بنام خدا

مقدمه‌ای بر الگوریتم‌ها

مترجمان :

گروه مهندسی ـ پژوهشی خوارزمی

مهدی رواخواه ـ محمود عالمی

محسن اسدی ـ شبنم جعفرخانی

شادی لنگری ـ الهام صدر

نشر درخشش – تابستان ۱۳۸۶

Cormen. Themas

کورمن، تامس

مقدمه‌ای بر الگوریتم‌ها / نویسنده تامس کورمن، چارلز لیزرسان، رونالد دیوست، کلیفورد اشتاین؛ مترجمان گروه مهندسی - پژوهشی خوارزمی (مهدی رواخواه؛ محمود عالمی، محسن اسدی، شبنم جعفرخانی، شادی لنگری، الهام صدر، مشهد، درخشش، ۱۳۸۳.

۷۳۶ ص: مصور. جدول. نمودار.

ISBN : 964 - 94855 - 1 - 1

فهرستنویسی براساس اطلاعات فیپا

Introduction to Algorithms.

ص.ع. به انگلیسی

۱. برنامه‌نویسی کامپیوتری. ۲. الگوریتم‌های کامپیوتری. الف. لیزرسان، چارلز،...... نویسندهٔ همکار. ب. دیوست، رونالد، نویسنده همکار. ج. اشتاین، کلیفورد، نویسنده همکار. د. رواه‌خواه، مهدی، ۱۳۶۱ - مترجم. ه عالمی، محمود، ۱۳۶۱ - مترجم همکار. و. اسدی، محسن، ۱۳۶۲ - مترجم همکار. ز. جعفرخانی، شبنم، مترجم همکار. ح. لنگری، شادی، ۱۳۶۲ - مترجم همکار. ط. صدر، الهام، ۱۳۶۲ - مترجم همکار. ی. عنوان.

QA ۷۶/۶ الف ۱۹

۰۰۵/۱ ۱۳۸۳

نشر درخشش مشهد ـ خیابان سعدی ـ بازارچه کتاب ـ شماره ۲۵ ـ تلفن ۲۲۵۱۹۲۳ ـ ۰۵۱۱

نام کتاب : **مقدمه ای بر الگوریتم‌ها**

عنوان اصلی : **Introduction to Algorithms**

مترجمین : **گروه مهندسی - پژوهشی خوارزمی**

ناشر : **درخشش**

چاپ سوم : **تابستان ۱۳۸۶**

تیراژ : **۲۰۰۰ جلد**

لیتوگرافی : **شایان**

چاپ : **دقت**

قیمت : **۶۰۰۰ تومان**

شابک : ۷ - ۹ - ۹۴۸۵۵ - ۹۶۴
ISBN : 964 - 94855 - 9 - 7

* کلیه حقوق قانونی این اثر، برای مترجمین و ناشر محفوظ است. نقل مطالب فقط بصورت معمول در مقالات تحقیقی با ذکر کامل نام ناشر و مترجمین این کتاب میسر است.

پیشگفتار مترجمان

حمد و سپاس خداوند منان را که به ما توفیق داد تا گامی کوچک در راه پیشبرد اهداف علمی برداریم. کلمهٔ **الگوریتم** از نام ریاضیدان برجسته ایرانی **ابوجعفر محمدبن موسی‌الخوارزمی** و بپاس خدمات گسترده او به توسعهٔ دانش بشری گرفته شده است. کلمهٔ **الجبرا** در انگلیسی نیز برگرفته از روی کتاب مشهور او **الجبر و المقابله** است.

اثر حاضر ترجمه کتاب Introduction to Algorithms و حاصل کوشش **گروه مهندسی - پژوهشی خوارزمی** است. کتاب مذکور هم اکنون به عنوان مرجع در بسیاری از دانشگاه‌های مطرح و معتبر جهان تدریس می‌شود.

این کتاب علاوه بر تمام مباحث **الگوریتم‌ها** و **ساختمان داده‌ها**، سایر مباحث علوم کامپیوتر از جمله **ذخیره و بازیابی اطلاعات** و **ساختمانهای گسسته** را دربر می‌گیرد.

به‌دلیل نبود مرجعی کامل در زمینه الگوریتم‌ها و نیاز مبرم اساتید و دانشجویان، بر آن شدیم تا آن را ترجمه کنیم و در اختیار دانش‌پژوهان عزیز قرار دهیم. اما باید یادآوری کنیم که از ترجمه بخش انتهایی کتاب اصلی با عنوان Selected Topics چشمپوشی کردیم، زیرا آنرا شامل مباحث پیچیده و بسیار فراتر از مقطع کارشناسی دیدیم.

امید است مطالب این کتاب مورد استفاده شما عزیزان قرار گیرد و راه‌گشای بخشی از نیازهای علمی شما باشد.

در پایان از اساتید محترم دانشگاه آزاد اسلامی واحد مشهد، **سرکار خانم دکتر قمرناز تدین تبریزی، جناب آقای مهندس حسن شاکری، جناب آقای مهندس سعید ابریشمی و جناب آقای مهندس سید آرش استادزاده** که از مساعدت و راهنمایی‌های ارزنده و بی‌دریغشان بهره فراوان بردیم، سپاسگزاری می‌کنیم.

گروه مهندسی - پژوهشی خوارزمی

دانشجویان رشته کامپیوتر - نرم‌افزار دانشگاه آزاد اسلامی واحد مشهد

مهدی رواخواه - محمود عالمی

محسن اسدی - شبنم جعفرخانی

شادی لنگری - الهام صدر

پیشگفتار مؤلفان

این کتاب مقدمه‌ای گسترده برای مطالعه مدرن الگوریتم‌های کامپیوتر فراهم می‌کند. در این کتاب الگوریتم‌های زیادی ارائه و به‌طور عمیق پوشش داده شده‌اند، که باعث می‌شود طراحی و تحلیل این الگوریتم‌ها برای تمام سطوح خوانندگان کتاب، امکان‌پذیر باشد. سعی کرده‌ایم بدون از دست دادن عمق مطلب یا دقت ریاضی، توضیحات را به زبان ساده بیان کنیم.

هر فصل یک الگوریتم، یک تکنیک طراحی، یک زمینه کاربردی، یا یک موضوع مرتبط را بیان می‌کند. الگوریتم‌ها بصورت شبه کد و به زبان انگلیسی طراحی شده‌اند تا برای کسانی که کمی برنامه‌نویسی کرده‌اند، قابل فهم باشند. کتاب شامل بیش از ۱۴۵ شکل جهت توصیف نحوه عملکرد الگوریتم‌ها می‌باشد. از آنجا که به **کارآیی** به عنوان یک معیار طراحی تأکید داریم، تحلیل‌های دقیقی از زمان اجرای همه الگوریتم‌ها ارائه کرده‌ایم.

این کتاب در ابتدا جهت استفاده در واحدهای درسی الگوریتم‌ها یا ساختمان داده‌ها برای دانشجویان مقطع کارشناسی و مقاطع بالاتر درنظر گرفته شده است. اما از آنجا که شامل مطالب فنی در طراحی الگوریتم‌ها بهمراه جنبه‌های ریاضی است، می‌تواند به عنوان خودآموز نیز توسط افراد حرفه‌ای استفاده شود.

سخنی با اساتید

این کتاب به‌گونه‌ای طراحی شده است که جامع و کامل باشد، و می‌تواند برای انواع واحدهای درسی، از واحدهای درسی کارشناسی در ساختمان داده‌ها تا واحدهای درسی مقاطع بالاتر در الگوریتم‌ها، مفید باشد. از آنجا که مطالب این کتاب بسیار فراتر از یک واحد درسی است، باید این کتاب را به عنوان یک «بوفه» یا «سلف» در نظر بگیرید که می‌توانید مطالبی که بیشترین تطبیق را با واحد درسی شما دارند، انتخاب و ارائه کنید.

ارائه واحد درسی با استفاده از برخی از فصل‌ها که به آنها احتیاج دارید، کار آسانی است. فصل‌های کتاب نسبتاً مستقل هستند، بطوریکه نیازی به نگرانی در مورد وابستگی غیرمنتظره و غیرضروری یک فصل به فصل دیگر نیست. هر فصل ابتدا مطالب آسانتر و سپس مطالب مشکل‌تر را بیان می‌کند و بخش‌ها در داخل هر فصل، مرز مطالب را مشخص می‌کنند. در یک واحد درسی دوره کارشناسی ممکن است تنها از بخش‌های ابتدایی یک فصل استفاده کنید، اما در واحدهای درسی مقاطع بالاتر می‌توانید تمام فصل را ارائه نمایید.

بیش از ۵۵۰ تمرین و ۱۰۰ مسئله در کتاب طرح شده است. هر بخش با تعدادی تمرین و هر فصل با

تعدادی مسئله خاتمه می‌یابد. تمرین‌ها عموماً صورت‌های کوتاهی دارند که مبانی اصلی مطالب را بررسی می‌کنند. بعضی از تمرین‌ها ساده و برای خودآزمایی مناسبند، در حالیکه برخی از آنها دشوار بوده و برای تکلیف منزل مناسبند. مسائل، مطالب پرکارتری هستند و اغلب، مباحث جدیدی را معرفی می‌کنند؛ و معمولاً شامل پرسش‌هایی هستند که دانشجویان را در طی مراحلی به سمت جواب رهنمون می‌کنند.

بخش‌ها و تمرین‌هایی که بیشتر برای دانشجویان مقاطع بالاتر از کارشناسی مناسبند را با (*) مشخص کرده‌ایم. یک بخش ستاره‌دار لزوماً مشکل‌تر از یک بخش بدون ستاره نیست، اما ممکن است نیاز به فهم ریاضیات پیشرفته‌تری داشته باشد. به همین شکل تمرین‌های ستاره‌دار، پیش‌زمینهٔ پیشرفته‌تر و خلاقیتی بیش از حد میانگین را نیاز دارند.

سخنی با دانشجویان

امیدواریم که این کتاب برای شما مقدمه‌ای دلپذیر در زمینه الگوریتم‌ها باشد. سعی شده است که هر الگوریتم، جذاب و قابل حصول باشد. برای کمک به شما در هنگامی که با الگوریتم‌های ناآشنا یا مشکل مواجه می‌شوید، هر الگوریتم را بصورت گام به گام شرح داده‌ایم. همچنین توضیحات دقیق ریاضی که برای درک تحلیل الگوریتم‌ها ضروری هستند فراهم شده‌اند. اگر از قبل با موضوعی آشنا هستید، متوجه می‌شوید که فصل‌ها به گونه‌ای سازماندهی شده‌اند که می‌توانید بخش‌های ابتدایی را سریع و به‌طور سطحی مطالعه کرده و بسرعت به مطالب پیشرفته‌تر برسید.

حجم این کتاب زیاد است و احتمالاً کلاس درسی شما بخشی از مطالب آنرا پوشش خواهد داد. اما سعی شده است این کتاب در حال حاضر بعنوان یک کتاب درسی و پس از آن بعنوان یک مرجع یا یک کتاب فنی در طول زندگیتان مفید باشد.

پیش‌نیازها برای خواندن این کتاب چیست؟

- باید مقداری تجربه برنامه‌نویسی داشته باشید. بویژه باید روالهای بازگشتی و ساختمان داده‌های ساده مانند آرایه‌ها و لیست‌های پیوندی را نیز درک کنید.
- باید در اثبات بوسیله استقرای ریاضی مهارت داشته باشید. قسمتهایی از کتاب به دانش محاسبات اولیه متکی هستند. علاوه بر آن، قسمت I این کتاب، تکنیکهای ریاضی که نیاز خواهید داشت را آموزش می‌دهد.

THOMAS H. CORMEN — *Honover, New Hampshire*
CHARLES E. LEISERSON — *Cambridge, Massachusetts*
RONALD L. RIVEST — *Cambridge, Massachusetts*
CLIFFORD STEIN — *Honover, New Hampshire*

مقدمه

این قسمت، نقطه آغاز در تفکر راجع به طراحی و تحلیل الگوریتم‌ها می‌باشد. این فصل بعنوان مقدمه‌ای گام به گام در نظر گرفته شده است بدین شکل که چگونه الگوریتم‌ها، بعضی از راهکارهای طراحی که در طول کتاب استفاده خواهیم کرد، و بسیاری از ایده‌های بنیادی که در تحلیل الگوریتم استفاده شده است، را تعیین کنیم. قسمت‌های بعدی این کتاب بر این اساس بنا نهاده خواهد شد.

فصل ۱ نگاهی کلّی بر الگوریتم‌ها و جایگاهشان در سیستم‌های کامپیوتری جدید دارد. این فصل بیان می‌کند که یک الگوریتم چیست و نمونه‌هایی از آن را بیان می‌کند. همچنین این فصل حالتی را تشریح می‌کند که الگوریتم‌ها یک تکنولوژی محسوب می‌گردند همانند سخت‌افزار سریع، رابط گرافیکی کاربر *(GUI)*، سیستم‌های شیی‌گرا، و شبکه‌ها.

در فصل ۲، اولین الگوریتمهایمان را که مسئله مرتب‌ساز یک توالی از n عدد را حل می‌کند، خواهیم دید که بصورت شبه کد نوشته شده‌اند، گر چه مستقیماً به هیچ زبان برنامه‌نویسی قرار دادی قابل ترجمه نیستند، ساختار الگوریتم را به اندازه‌ی کافی واضح و روشن منتقل می‌کنند که یک برنامه نویس مجرب می‌تواند آن را با زبان انتخابیش پیاده‌سازی کند. الگوریتم‌های مرتب‌سازی که بررسی می‌کنیم عبارتند از:

مرتب‌سازی درجی، که از روش افزایشی و مرتب‌سازی ادغام، که از یک تکنیک بازگشتی موسوم به "تقسیم و حل" استفاده می‌کند. گرچه زمان مورد نیاز هر کدام با مقدار n افزایش می‌یابد، میزان افزایش دو الگوریتم با هم فرق می‌کند. در فصل ۲ زمان اجرای این الگوریتم‌ها را مشخص می‌کنیم و علامت گذاری مناسبی را برای بیان آنها ارائه می‌کنیم.

فصل ۳ این علامت گذاری را بطور دقیق تعریف می‌کند، که ما آن را علامت گذاری مجانبی می‌نامیم. این فصل با تعریف علامت گذاری‌های مجانبی متعددی شروع می‌شود، که برای محدود کردن زمان‌های اجرای الگوریتم‌ها از بالا و یا پایین استفاده می‌کنیم. ادامه فصل ۳ در اصل، نمایش علامت گذاری ریاضی است. هدف بیشتر این است که تطابق و همخوانی استفاده شما از علامت گذاری را با آنچه که در این کتاب است تضمین نماید تا آن که بخواهد مفاهیم جدید ریاضی را به شما بیاموزد.

فصل ۴ بیشتر در مورد روش تقسیم و حل که در فصل ۲ معرفی شد، می‌باشد. بخصوص فصل ۴ شامل روش‌هایی برای حل رابطه‌های بازگشتی است، که برای توصیف زمانهای اجرای الگوریتم‌های بازگشتی مفید هستند. یک تکنیک قدرتمند، "روش اصلی" است که می‌تواند برای حل رابطه‌های بازگشتی

I

بنیادها

فهرست

I بنیادها

مقدمه ۱۲

۱ نقش الگوریتم‌ها در محاسبه ۱۵
- ۱.۱ الگوریتم‌ها ۱۵
- ۱.۲ الگوریتم‌ها بعنوان یک تکنولوژی ۲۰

۲ شروع ۲۵
- ۲.۱ مرتب سازی درجی ۲۵
- ۲.۲ تحلیل الگوریتم‌ها ۳۱
- ۲.۳ طراحی الگوریتم‌ها ۳۹

۳ رشد توابع ۵۳
- ۳.۱ نمادگذاری مجانبی ۵۳
- ۳.۲ نمادهای استاندارد و توابع عمومی ۶۳

۴ رابطه‌های بازگشتی ۷۴
- ۴.۱ روش جایگذاری ۷۵
- ۴.۲ روش درخت بازگشت ۸۰
- ۴.۳ روش اصلی ۸۵
- * ۴.۴ اثبات قضیه اصلی ۸۸

۵ تحلیل احتمالی و الگوریتم‌های تصادفی ۱۰۴
- ۵.۱ مسئله استخدام ۱۰۴
- ۵.۲ متغیرهای تصادفی شاخص ۱۰۸
- ۵.۳ الگوریتم‌های تصادفی ۱۱۲
- * ۵.۴ تحلیل احتمالی و استفاده‌های دیگر متغیرهای تصادفی شاخص ۱۱۹

II مرتب‌سازی و شاخص‌های آمار ترتیبی

مقدمه ۱۳۶

۶ مرتب‌سازی heap ۱۴۰
- ۶.۱ Heapها ۱۴۰
- ۶.۲ خفظ ویژگی heap ۱۴۳
- ۶.۳ ساختن یک heap ۱۴۵
- ۶.۴ الگوریتم مرتب‌سازی heap ۱۴۹
- ۶.۵ صف اولویت ۱۵۱

۷ مرتب‌سازی سریع ۱۵۸
۷.۱ توصیف مرتب‌سازی سریع ۱۵۸
۷.۲ کارایی مرتب‌سازی سریع ۱۶۲
۷.۳ صورت تصادفی مرتب‌سازی سریع ۱۶۷
۷.۴ تحلیل مرتب‌سازی سریع ۱۶۸
۸ مرتب‌سازی در زمان خطی ۱۷۸
۸.۱ حدهای پایین برای مرتب‌سازی ۱۷۸
۸.۲ مرتب‌سازی شمارشی ۱۸۱
۸.۳ مرتب‌سازی مبنایی ۱۸۴
۸.۴ مرتب‌سازی پیمانه‌ای ۱۸۸
۹ میانه‌ها و شاخص‌های آمار ترتیبی ۱۹۷
۹.۱ مینیمم و ماکزیمم ۱۹۸
۹.۲ انتخاب در زمان خطی مورد انتظار ۱۹۹
۹.۳ انتخاب در بدترین حالت زمان خطی ۲۰۳

III ساختمان داده‌ها

مقدمه ۲۱۲
۱۰ ساختمان داده‌های مقدماتی ۲۱۷
۱۰.۱ پشته‌ها و صف‌ها ۲۱۷
۱۰.۲ لیست‌های پیوندی ۲۲۱
۱۰.۳ پیاده‌سازی اشاره‌گرها و اشیاء ۲۲۶
۱۰.۴ نمایش درخت‌های مشتق شده ۲۳۱
۱۱ جدول درهم‌سازی ۲۳۸
۱۱.۱ جدول‌های آدرس مستقیم ۲۳۹
۱۱.۲ جدول‌های درهم‌سازی ۲۴۱
۱۱.۳ توابع درهم‌سازی ۲۴۷
۱۱.۴ آدرس‌دهی باز ۲۵۵
* ۱۱.۵ درهم‌سازی کامل ۲۶۴
۱۲ درخت جستجوی دودویی ۲۷۲
۱۲.۱ درخت جستجوی دودویی چیست؟ ۲۷۳
۱۲.۲ پرس‌وجوی یک درخت جستجوی دودویی ۲۷۶
۱۲.۳ درج و حذف ۲۸۰
* ۱۲.۴ درخت جستجوی دودویی تصادفی ساخته شده ۲۸۴
۱۳ درخت‌های قرمز ـ سیاه ۲۹۲
۱۳.۱ ویژگی‌های درخت‌های قرمز ـ سیاه ۲۹۲
۱۳.۲ چرخش‌ها ۲۹۶
۱۳.۳ درج ۲۹۹
۱۳.۴ حذف ۳۰۷
۱۴ بهبود ساختمان داده‌ها ۳۲۱

۱۴.۱ آمار ترتیبی پویا ۳۲۱
۱۴.۲ چطور یک ساختمان داده را بهبود بخشیم ۳۲۷
۱۴.۳ درخت‌های بازه‌ای ۳۳۱

IV *طراحی و تکنیک‌های تحلیل پیشرفته*

مقدمه ۳۴۰
۱۵ برنامه‌سازی پویا ۳۴۳
۱۵.۱ زمان‌بندی خطوط مونتاژ ۳۴۴
۱۵.۲ ضرب زنجیره‌ای ماتریس‌ها ۳۵۲
۱۵.۳ عناصر برنامه‌سازی پویا ۳۶۰
۱۵.۴ طولانی‌ترین زیررشته مشترک ۳۷۲
۱۵.۵ درخت‌های جستجوی دودویی بهینه ۳۷۸
۱۶ الگوریتم‌های حریصانه ۳۹۲
۱۶.۱ مسئله انتخاب فعالیت ۳۹۳
۱۶.۲ عناصر تدبیر حریصانه ۴۰۲
۱۶.۳ کد **Huffman** ۴۰۸
* ۱۶.۴ پایه‌های نظری روش‌های حریصانه ۴۱۷
* ۱۶.۵ مسئله زمان‌بندی کارها ۴۲۴
۱۷ تحلیل سرشکن شده ۴۳۱
۱۷.۱ تحلیل جمعی ۴۳۲
۱۷.۲ روش حسابداری ۴۳۶
۱۷.۳ روش پتانسیل ۴۳۹
۱۷.۴ جداول پویا ۴۴۳

V *ساختمان داده‌های پیشرفته*

مقدمه ۴۶۰
۱۸ B-Treeها ۴۶۳
۱۸.۱ تعریف **B-tree** ۴۶۸
۱۸.۲ اعمال اصلی روی **B-tree**ها ۴۷۰
۱۸.۳ حذف یک کلید از **B-tree** ۴۷۸
۱۹ heapهای دو جمله‌ای ۴۸۵
۱۹.۱ درخت‌های دو جمله‌ای و **heap**های دو جمله‌ای ۴۸۷
۱۹.۲ اعمال بر روی **heap**های دو جمله‌ای ۴۹۲
۲۰ heapهای فیبوناچی ۵۰۶
۲۰.۱ ساختار **heap**های فیبوناچی ۵۰۷
۲۰.۲ اعمال **heap** قابل ادغام ۵۱۰
۲۰.۳ کاهش کلید و حذف یک گره ۵۲۰
۲۰.۴ محدود کردن ماکزیمم درجه ۵۲۵
۲۱ ساختمان داده‌ها برای مجموعه‌های جدا از هم ۵۳۰

۲۱.۱ اعمال روی مجموعه جدا از هم ۵۳۰
۲۱.۲ نمایش لیست پیوندی مجموعه‌های جدا از هم ۵۳۳
۲۱.۳ جنگل‌های مجموعه جدا از هم ۵۳۷
* ۲۱.۴ تحلیل واحدسازی بر حسب مرتبه به همراه فشرده‌سازی مسیر ۵۴۱

VI الگوریتم‌های گراف

مقدمه ۵۵۶
۲۲ **الگوریتم‌های اولیه گراف** ۵۵۹
۲۲.۱ نمایش‌های گراف ۵۵۹
۲۲.۲ جستجوی اول سطح ۵۶۳
۲۲.۳ جستجوی اول عمق ۵۷۳
۲۲.۴ مرتب‌سازی موضعی ۵۸۳
۲۲.۵ اجزای همبند قوی ۵۸۶
۲۳ **درخت پوشای مینیمم** ۵۹۴
۲۳.۱ رشد درخت پوشای مینیمم ۵۹۵
۲۳.۲ الگوریتم‌های Prim و Kruskal ۶۰۱
۲۴ **کوتاهترین مسیرها از مبدأ واحد** ۶۱۳
۲۴.۱ الگوریتم Bellman-Ford ۶۲۲
۲۴.۲ کوتاهترین مسیر از مبدأ واحد در گراف‌های جهتدار بدون دور ۶۲۶
۲۴.۳ الگوریتم Dijkstra ۶۲۹
۲۴.۴ محدودیت‌های تفاضلی و کوتاهترین مسیرها ۶۳۵
۲۴.۵ اثبات ویژگی‌های کوتاهترین مسیرها ۶۴۱
۲۵ **کوتاهترین مسیرها بین همه جفت‌ها** ۶۵۴
۲۵.۱ کوتاهترین مسیرها و ضرب ماتریس ۶۵۶
۲۵.۲ الگوریتم Floyd-Warshall ۶۶۳
۲۵.۳ الگوریتم Johnson برای گراف‌های پراکنده (خلوت) ۶۷۰
۲۶ **ماکزیمم جریان** ۶۷۷
۲۶.۱ شبکه‌های جریان ۶۷۸
۲۶.۲ روش Ford-Fulkerson ۶۸۵
۲۶.۳ تطبیق دو بخشی ماکزیمم ۶۹۹
* ۲۶.۴ الگوریتم‌های راندن ـ برچسب‌دهی مجدد ۷۰۴
* ۲۶.۵ الگوریتم برچسب‌دهی مجدد ـ به ـ جلو ۷۱۷

که از الگوریتم‌های تقسیم و حل سرچشمه می‌گیرند استفاده شود. بیشتر قسمتهای فصل ۴ به اثبات درستی روش اصلی اختصاص یافته است، اگر چه می‌توانیم بدون هیچ ضرری از این اثبات بگذریم.

فصل ۵، تجزیه و تحلیل احتمالی و الگوریتم‌های تصادفی را معرفی می‌کند. نوعاً تجزیه و تحلیل احتمالی را برای تعیین زمان اجرای یک الگوریتم در حالت‌هایی استفاده می‌کنیم که در آنها بواسطهٔ وجود یک توزیع احتمال ذاتی، زمان اجرا ممکن است روی ورودیهای متفاوت با اندازه یکسان فرق کند. در بعضی موارد فرض می‌کنیم که ورودیها مطابق با یک توزیع احتمال معین هستند، بنابراین داریم روی زمان اجرای همه ورودیهای ممکن میانگین می‌گیریم. در موارد دیگر، توزیع احتمال نه از ورودیها، بلکه از انتخابهای تصادفی که در طول جریان الگوریتم به وجود می‌آیند، حاصل می‌شود. الگوریتمی که رفتارش نه فقط بوسیله ورودیش بلکه با مقادیری که توسط یک مولد عدد تصادفی تولید شده‌اند، تعیین شده باشد، الگوریتمی تصادفی است. می‌توانیم الگوریتم‌های تصادفی را برای اعمال یک توزیع احتمال روی ورودیها -که به موجب آن، تضمین می‌کند که هیچ ورودی خاص همیشه موجب کارآیی ضعیف نمی‌گردد- یا حتی برای محدود کردن میزان خطای الگوریتم‌هایی که سبب می‌شوند تا نتایج نادرستی روی یک پایه محدود تولید شود، استفاده کنیم.

۱ نقش الگوریتم‌ها در محاسبه

الگورتیم‌ها چه هستند؟ چرا مطالعه الگوریتم‌ها با ارزش است؟ نقش الگوریتم‌ها نسبت به دیگر تکنولوژی‌های مورد استفاده در کامپیوترها چیست؟ در این فصل به این سؤالات پاسخ خواهیم داد.

۱.۱ الگوریتم‌ها

بطور غیر رسمی یک **الگوریتم**[1] ، هر روال محاسباتی خوش تعریف است که مقداری، یا مجموعه‌ای از مقادیر را بعنوان **ورودی**[2] می‌گیرد و مقداری، یا مجموعه‌ای از مقادیر را بعنوان **خروجی**[3] تولید می‌کند. بنابراین یک الگوریتم، یک توالی از گام‌های محاسباتی است که ورودی را به خروجی تبدیل می‌کند.

همچنین می‌توانیم یک الگوریتم را بعنوان وسیله‌ای جهت حل یک **مسئله محاسباتی**[4] خوش تعریف در نظر بگیریم. طرز بیان مسئله در جملات عمومی، رابطهٔ ورودی/خروجی خواسته شده را مشخص می‌کند.

برای مثال، ممکن است کسی نیاز داشته باشد تا یک توالی از اعداد را به ترتیب غیر نزولی مرتب کند. این مسئله در عمل به وفور رخ می‌دهد و زمینه مناسبی برای معرفی بسیاری از تکنیک‌های طراحی استاندارد و ابزار تحلیل فراهم می‌کند. در اینجا نحوه تعریف رسمی **مسئله مرتب‌سازی**[5] آمده است:

ورودی: یک توالی از n عدد $\langle a_1, a_2, \ldots, a_n \rangle$.

خروجی: یک جایگشت (ترتیب مجدد) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ از توالی ورودی، به طوری که $a'_1 \le a'_2 \le \ldots \le a'_n$

برای مثال، توالی ورودی $\langle 31, 41, 59, 26, 41, 58 \rangle$ داده شده، یک الگوریتم مرتب‌سازی بعنوان

---

1. *Algorithm*
2. *Input*
3. *Output*
4. *Computational problem*
5. *Sorting Problem*

خروجی، توالی <26, 31, 41, 41, 58, 59> را بر می‌گرداند. چنین توالی ورودی، یک **نمونه**[1] مسئله مرتب‌سازی نامیده می‌شود. بطور کلی، یک **نمونه مسئله**[2] تشکیل شده است از ورودی (که در محدودیت‌های اعمال شده در مسئله صدق می‌کند) که جهت محاسبه راه‌حلی برای مسئله مورد نیاز می‌باشد.

مرتب‌سازی عملی بنیادی در علم کامپیوتر است (برنامه‌های زیادی از آن بعنوان یک مرحله میانی استفاده می‌کنند)، و در نتیجه تعداد زیادی از الگوریتم‌های مرتب‌سازی مناسب بهبود یافته‌اند. اینکه کدام الگوریتم برای یک کاربرد مورد نظر بهتر است - در میان عوامل دیگر - بستگی دارد به تعداد اقلام اطلاعاتی که بایستی مرتب شوند، میزان مرتب بودن اقلام، محدودیت‌های ممکن روی مقادیر اقلام، و نوع دستگاه ذخیره‌سازی که استفاده شده: حافظه اصلی، دیسکها یا نوارها.

الگوریتمی **صحیح**[3] گفته می‌شود که برای هر نمونه ورودی، با خروجی صحیحی متوقف شود. می‌گوییم که یک الگوریتم صحیح، مسئله محاسباتی داده شده را **حل می‌کند**[4]. یک الگوریتم نادرست ممکن است روی برخی نمونه‌های ورودی هرگز متوقف نشود، یا ممکن است با جوابی نامطلوب متوقف شود. بر خلاف انتظار، الگوریتم‌های نادرست گاهی اوقات می‌توانند مفید باشند، اگر بتوان میزان خطایشان را کنترل کرد. اگر چه معمولاً تنها به الگوریتم‌های صحیح معطوف می‌شویم.

یک الگوریتم می‌تواند در زبان انگلیسی بصورت یک برنامه کامپیوتری یا حتی بصورت یک طرح سخت‌افزاری مشخص شود. تنها لازم است که این مشخص‌سازی، توصیفی دقیق از روال محاسباتی را که باید دنبال شود فراهم کند.

چه نوع مسائلی توسط الگوریتم‌ها حل می‌شوند؟

مرتب‌سازی به هیچ وجه تنها مسئله محاسباتی که در آن الگوریتم‌ها توسعه یافته‌اند نمی‌باشد. (ممکن است وقتی که حجم این کتاب را دیدید بسیار مردّد شده باشید.) کاربردهای عملی الگوریتم‌ها در همه جا وجود دارند و شامل نمونه‌های زیر می‌باشند:

- پروژه ژن انسان دارای این اهداف است: شناسایی *100,000* ژن در *DNA* انسان، تعیین یک توالی از *3* بیلیون جفت پایه شیمیایی که *DNA* انسان را تشکیل می‌دهند، ذخیره‌سازی این اطلاعات در پایگاه داده‌ها و توسعه ابزارهایی برای تحلیل داده‌ها. هر کدام از این گام‌ها به الگوریتم‌های پیچیده‌ای نیاز دارد. از آنجا که راه حل‌های مسائل متعدد مربوط فراتر از محدوده این کتاب هستند ایده‌های بسیاری از فصل‌های این کتاب در حل مسائل زیست شناسی استفاده شده‌اند که بموجب آن دانشمندان قادر به تکمیل کارها بطور مؤثر در هنگام استفاده از منابع می‌شوند. صرفه‌جویی‌ها

---

1. instance
2. instance of a problem
3. correct
4. solve

در وقت انسان و زمان ماشین، و در پول و هزینه صورت می‌پذیرد، همانطور که اطلاعات بیشتری می‌توانند از تکنیک‌های آزمایشگاهی استخراج شوند.

- اینترنت مردم را در سرتاسر دنیا قادر می‌سازد تا به مقادیر زیادی از اطلاعات بطور سریع دستیابی یافته و آنها را بازیابی کنند. بنابراین برای این کار، الگوریتم‌های هوشمند به منظور اداره و دستکاری این حجم زیاد از اطلاعات بکار گرفته می‌شوند. نمونه‌هایی از مسائلی که بایستی حل شوند عبارتند از: یافتن مسیرهای مناسب که در آن‌ها داده‌ها منتقل می‌شوند (تکنیک‌های حل این مسائل در فصل ۲۴ بیان می‌شود)، و استفاده از یک موتور جستجو برای یافتن سریع صفحاتی که روی آن‌ها اطلاعات خاصی قرار دارند (تکنیک‌های مربوطه در فصل ۱۱ بیان می‌شود).
- تجارت الکترونیک این امکان را فراهم می‌کند که کالاها و سرویس‌ها بطور الکترونیکی معامله و مبادله شوند. اگر تجارت الکترونیک بطور گسترده‌ای استفاده شود، توانایی نگهداری اطلاعات مانند اعداد کارت اعتباری، کلمات رمز و صورت حسابهای خصوصی بانک ضروری است. رمز گشایی کلید عمومی و امضاهای دیجیتالی در هسته تکنولوژیهای مورد استفاده قرار دارند و بر اساس الگوریتم‌های عددی و نظریه اعداد می‌باشند.
- در صنعت و دیگر محیط‌های تجاری، اغلب مهم است که منابع نادر به سودمندترین روش تخصیص یابند. یک شرکت نفتی ممکن است بخواهد تعیین کند که در چه مکانی چاههای نفتی را حفّاری کند تا سود مورد نظرش را به حداکثر برساند. کاندیدای ریاست جمهوری ایالات متحده ممکن است بخواهد محلی مناسب را برای خرید تبلیغات مبارزاتی برای به حداکثر رساندن شانس بردن یک رأی مشخص کند. یک خط هوایی ممکن است بخواهد خدمه هواپیما را برای پرواز به کم هزینه‌ترین روش ممکن تعیین کند، تضمین کند که همه پروازها پوشش داده شده‌اند و مقررات دولتی راجع به زمان بندی استفاده از خدمه‌ها رعایت شده است. یک تأمین کننده خدمات اینترنتی ممکن است بخواهد تعیین کند که در چه مکانی منابع اضافی‌اش را برای سرویس به مشتریانش بطور مؤثرتر قرار دهد. همه اینها نمونه‌هایی از مسائلی است که می‌توانند با استفاده از برنامه‌سازی خطی حل شوند.

از آنجا که بعضی جزئیات این مثالها فراتر از محدوده این کتاب است، تکنیک‌های اساسی زیر را که به این مسائل و مسائلی در این محدوده مربوط می‌شوند ارائه می‌کنیم. همچنین نشان می‌دهیم که چطور اکثر مسائل واقعی این کتاب را حل می‌کنیم که شامل موارد زیر می‌شود:

- یک نقشه جاده که در آن فاصله بین هر جفت تقاطع همجوار مشخص شده است را داریم، و هدفمان این است که کوتاهترین مسیر از یک تقاطع به تقاطع دیگر را معین کنیم. تعداد مسیرهای ممکن می‌توانند زیاد باشند، حتی اگر مسیرهایی که از روی خودشان عبور می‌کنند در نظر نگیریم. چگونه انتخاب کنیم که کدام یک از مسیرهای ممکن کوتاهترین است؟ در اینجا نقشه جاده را (که خودش مدلی از جاده‌های واقعی است) بصورت یک گراف مدل می‌کنیم (که در فصل ۱۰ با آن مواجه خواهیم شد)، و می‌خواهیم

کوتاهترین مسیر از یک رأس به رأس دیگر را در گراف پیدا کنیم. در فصل ۲۴ خواهیم دید که چگونه این مسئله را بطور کار آمد حل کنیم.

- توالی $\langle A_1, A_2, \ldots, A_n \rangle$ از n ماتریس را داریم و می‌خواهیم تا حاصلضرب آنها $A_1 A_2 \ldots A_n$ را تعیین کنیم. چون ضرب ماتریس شرکت‌پذیر است، ترتیب‌های ضرب معتبر متعددی وجود دارد. برای مثال، اگر $n=4$ باشد می‌توانیم ضرب ماتریسها را به صورت یکی از ترتیب‌های پرانتز گذاری شده در زیر انجام دهیم:
$(A_1(A_2(A_3A_4)))$، $(A_1((A_2A_3)A_4))$، $((A_1A_2)(A_3A_4))$، $((A_1(A_2A_3))A_4)$، یا $(((A_1A_2)A_3)A_4)$. اگر همه این ماتریس‌ها مربعی باشند (و از این رو هم اندازه) ترتیب ضرب تأثیری بر اینکه ضربهای ماتریس چقدر زمان می‌برند ندارد. ولی اگر این ماتریس‌ها در اندازه‌های متفاوتی باشند (با وجود این اندازه‌هایشان برای ضرب ماتریس سازگار است)، آنگاه ترتیب ضرب تفاوت زیادی پیدا می‌کند. تعداد ترتیب‌های ضرب ممکن، توانی در n است و بنابراین آزمایش همه ترتیب‌های ممکن، ممکن است زمان خیلی طولانی را صرف نماید. در فصل ۱۵ خواهیم دید که چگونه یک تکنیک کلی معروف به برنامه‌سازی پویا را بکار بریم تا این مسئله بطور بسیار مؤثرتری حل شود.
- معادله $ax \equiv b$ (به پیمانه n) را داریم که در آن a و b و n اعداد صحیح هستند. و می‌خواهیم همه اعداد صحیح x به پیمانه n که در معادله صدق می‌کند را بیابیم. ممکن است صفر، یک، یا بیشتر از یک جواب وجود داشته باشد. می‌توان بسادگی $x=0, 1, \ldots, n-1$ را بترتیب امتحان کرد.
- n نقطه در صفحه داده شده است و می‌خواهیم بدنه محدّب این نقاط را بیابیم. بدنه محدّب، کوچکترین چند ضلعی محدّبی است که شامل این نقاط می‌باشد. بطور شهودی می‌توان هر نقطه را به شکل یک میخ که در تخته بصورت برجسته واقع شده است در نظر گرفت. بدنه محدّب بصورت یک باند محکم که همه میخ‌ها را احاطه می‌کند نمایش داده می‌شود. هر میخ که حول آن باند پیچیده می‌شود، یک رأس بدنه محدّب است. هر کدام از 2^n زیر مجموعه نقاط، ممکن است رأسهای بدنه محدّب باشند. دانستن اینکه کدام یک از نقاط، رأسهای بدنه محدب هستند به تنهایی کافی نیست، چون نیاز داریم ترتیبی که در آنها ظاهر می‌شوند را نیز بدانیم. لذا انتخابهای زیادی برای رأسهای بدنه محدّب وجود دارد.

این لیست مسائل شامل تمام جزئیات نمی‌باشد (همانطور که شما احتمالاً دوباره از قطر این کتاب حدس زده‌اید)، اما دو ویژگی مشترک در همه الگوریتم‌های جالب توجه را ارائه می‌دهد:

۱. راه حل‌های کاندید زیادی وجود دارند که اکثر آنها آن چه که ما می‌خواهیم نیستند. یافتن یکی از آنها که ما می‌خواهیم، مسلماً می‌تواند یک مبارزه‌طلبی را بوجود آورد.

۲. کاربردهای عملی وجود دارد. از مسائل لیست بالا، مسئله کوتاهترین مسیرها ساده‌ترین نمونه است. یک شرکت حمل و نقل مانند شرکت راه آهن یا شرکت حمل و نقل با کامیون، دریافتن کوتاهترین

مسیرها در میان جاده‌ها یا شبکه راه آهن منفعت مالی دارد، زیرا استفاده از مسیرهای کوتاهتر، کار و هزینه‌های سوختگیری کمتری را در پی دارد. یا اینکه یک گره مسیر دهی روی اینترنت ممکن است نیاز داشته باشد تا کوتاهترین مسیر در شبکه را بیابد تا اینکه یک پیام به سرعت مسیر دهی شود.

ساختمان داده‌ها

این کتاب شامل ساختمان داده‌های متعددی است. یک ساختمان داده [1]، روشی برای ذخیره کردن و سازماندهی داده‌ها به منظور سهولت بخشیدن به اصلاحات و دسترسی است. هیچ ساختمان داده‌ای به تنهایی در همه مقاصد خوب کار نمی‌کند و بنابراین مهم است که تواناییها و محدودیتهای بسیاری از آن‌ها را بشناسیم.

تکنیک

گرچه می‌توانید این کتاب را بعنوان یک "کتاب آشپزی" برای الگوریتم‌ها استفاده کنید ممکن است روزی با مسئله‌ای روبرو شوید که برای آن نمی‌توانید الگوریتم بیان شده‌ای را به آسانی پیدا کنید (بعنوان مثال، بسیاری از تمرین‌ها و مسائل این کتاب). این کتاب تکنیک‌های طراحی و تحلیل الگوریتم را به شما آموزش می‌دهد تا اینکه بتوانید خودتان الگوریتم‌ها را توسعه دهید، نشان دهید که جواب درستی می‌دهند، و کارآیی آنها را درک کنید.

مسائل سخت [2]

اکثر این کتاب درباره الگوریتم‌های کارآمد است. مقیاس معمول ما از کارآیی سرعت است، یعنی تا چه اندازه یک الگوریتم زمان صرف می‌کند تا نتایج خود را تولید کند. هر چند مسائلی وجود دارند که برای هر کدام هیچ راه حل مؤثری شناخته نشده است. زیر مجموعه جالب توجهی از این مسائل که معروف به *NP-complete* هستند وجود دارد.

چرا مسائل *NP-complete* جالب توجه هستند؟ اول اینکه: گرچه هرگز هیچ الگوریتم کارآیی برای یک مسئله *NP-complete* شناخته نشده است، تا کنون هیچکس ثابت نکرده است که الگوریتمی کارآ برای چنین مسئله‌ای، نمی‌تواند وجود داشته باشد. به عبارت دیگر برای مسائل *NP-complete*، وجود یا عدم وجود الگوریتم‌های کارآ نامعلوم است. دوم اینکه: مجموعه مسائل *NP-complete* دارای ویژگی قابل ملاحظه‌ای هستند که اگر الگوریتمی کارآمد برای یکی از آن‌ها موجود باشد، آنگاه الگوریتم‌های کار آمدی برای همه آنها وجود دارند. این رابطه در میان مسائل *NP-complete*، عدم وجود راه حل‌های کار آمد را آزار دهنده‌تر می‌کند. سوم اینکه: چندین مسئله‌ی *NP-complete* شبیه اما

---

1. Data Structure

2. Hard problems

نه یکسان با مسائلی که برای آنها الگوریتم‌های کارآمدی را می‌شناسیم، وجود دارند. تغییر کوچکی در بیان مسئله می‌تواند موجب تغییر بزرگی در کارآیی بهترین الگوریتم شناخته شده شود.

شناخت مسائل *NP-complete* ارزشمند است، زیرا بعضی از آنها بطور شگفت‌انگیزی اغلب در کاربردهای حقیقی رخ می‌دهند. اگر از شما خواسته شده باشد تا برای یک مسئله *NP-complete* الگوریتمی کارآمد ارائه دهید، احتمال دارد زمان زیادی در جستجوی بدون نتیجه صرف کنید. اگر بتوانید نشان دهید مسئله، *NP-complete* است می‌توانید بجای آن وقت خود را صرف بهبود الگوریتمی کارآمد نمایید که جوابی مناسب امّا نه بهترین جواب را ارائه می‌دهد.

بعنوان یک مثال واقعی، یک شرکت حمل و نقل با کامیون با یک انبار مرکزی در نظر بگیرید. این شرکت هر روز کامیون را در انبار بارگیری می‌کند و آن را به محل‌های متعددی جهت تحویل بار می‌فرستد. در پایان روز کامیون بایستی به انبار برگردد تا برای بارگیری روز بعد آماده شود. برای کاهش هزینه‌ها شرکت می‌خواهد ترتیبی از ایستگاههای تحویلی را انتخاب کند که کوتاهترین فاصله پیموده شده توسط کامیون را حاصل کنند. این مسئله به «مسئله فروشنده دوره گرد» معروف است و *NP-complete* است. این مسئله هیچ الگوریتم کارآمد شناخته شده‌ای ندارد. هر چند با در نظر گرفتن فرضیات مشخص، الگوریتم‌های کار آمدی وجود دارند که فاصله‌ای کلی ارائه می‌دهند که با کوتاهترین مسیر ممکن فاصله چندانی ندارد.

تمرین‌ها

۱ - ۱.۱ نمونه‌ای از یک دنیای واقعی که یکی از مسائل محاسباتی زیر در آن ظاهر می‌گردد ارائه دهید: مرتب‌سازی، تعیین بهترین ترتیب ضرب ماتریس‌ها، یا یافتن بدنه محدّب.

۲ - ۱.۱ به غیر از سرعت، چه مقیاس دیگری از کارآیی ممکن است در یک دنیای واقعی در نظر گرفته شود؟

۳ - ۱.۱ یک ساختمان داده انتخاب کنید که قبلاً دیده‌اید و راجع به محدودیتها و تواناییهایش بحث کنید.

۴ - ۱.۱ مسائل فروشنده دوره گرد و کوتاهترین مسیر که در بالا ذکر شدند از چه نظر به هم شبیه‌اند؟ از چه لحاظ متفاوتند؟

۵ - ۱.۱ مسئله‌ای در دنیای واقعی بیابید که در آن فقط بهترین راه حل درست عمل می‌کند. سپس مسئله‌ای را بیابید که در آن راه حلی که «تقریباً» بهترین راه حل است، به حدّ کافی مناسب می‌باشد.

۱.۲ الگوریتم‌ها بعنوان یک تکنولوژی

فرض کنید کامپیوترها بی‌اندازه سریع بودند و حافظه کامپیوتر رایگان می‌بود. آیا هیچ دلیلی برای

مطالعه الگوریتم‌ها داشتید؟ پاسخ مثبت است، تنها به این دلیل که شما همچنان می‌خواهید نشان دهید که روش حل تان پایان می‌یابد و پاسخ درستی را بر می‌گرداند.

اگر کامپیوترها فوق العاده سریع بودند، هر روش درستی برای حل مسئله جواب می‌داد. شاید بخواهید که پیاده سازیتان مطابق با یک تمرین مهندسی نرم‌افزار خوب باشد (یعنی خوب طراحی و مستند شده باشد)، اما اغلب از روشی که ساده‌ترین پیاده‌سازی را دارد استفاده می‌کنید.

البته کامپیوترها ممکن است سریع باشند اما بی‌نهایت سریع نیستند. و حافظه ممکن است ارزان باشد اما رایگان نیست. زمان محاسبه از این جهت یک منبع محدود است و بنابراین زمان محاسبه و لذا فضا در حافظه، منابعی محدود هستند. این منابع باید بطور خردمندانه‌ای استفاده شوند و برای انجام این کار الگوریتم‌هایی که از لحاظ زمان یا فضا کارآمد می‌باشند به شما کمک خواهند کرد.

راندمان[1]

الگوریتم‌هایی که برای حل یک مسئله یکسان ابداع شده‌اند اغلب بطور برجسته‌ای در کارآیی با هم تفاوت دارند. این تفاوتها می‌توانند خیلی مهم‌تر از تفاوتهای ناشی از سخت‌افزار و نرم‌افزار باشند.

بعنوان یک مثال، در فصل ۲، دو الگوریتم برای مرتب‌سازی مشاهده خواهیم کرد. اولی به نام مرتب‌سازی درجی[2]، تقریباً زمانی برابر c_1n^2 برای مرتب کردن n داده صرف می‌کند، c_1 ثابتی است که وابسته به n نیست. بعبارت دیگر تقریباً زمانی متناسب با n^2 صرف می‌کند. دومی مرتب‌سازی ادغام[3]، تقریباً زمانی برابر c_2nlgn صرف می‌کند، که lgn مخفف log_2n است و c_2 ثابت دیگری است که آن نیز وابسته به n نیست. مرتب‌سازی درجی معمولاً دارای یک ضریب ثابتِ کوچکتر از مرتب‌سازی ادغام است، بعبارت دیگر: $c_1 < c_2$ خواهیم دید که ضرایب ثابت می‌توانند در زمان اجرا بسیار کم اهمیت‌تر از وابستگی الگوریتم به سایز ورودی n باشند. در جاییکه مرتب‌سازی ادغام دارای ضریبی از lgn در زمان اجرایش است مرتب‌سازی درجی دارای ضریبی از n است که بسیار بزرگتر می‌باشد. گر چه مرتب‌سازی درجی معمولاً برای سایزهای ورودی کوچک سریعتر از مرتب‌سازی ادغام است، زمانیکه سایز ورودی n به اندازه کافی بزرگ شود استفادهٔ مرتب‌سازی ادغام از lgn بجای n تمام تفاوتهای ضرایب ثابت را جبران می‌کند. اهمیتی ندارد که چقدر c_1 از c_2 کوچکتر است، همیشه نقطه‌ای وجود دارد که در ورای آن، مرتب‌سازی ادغام سریع‌تر است.

به عنوان یک مثال واقعی اجازه دهید یک کامپیوتر سریعتر (کامپیوتر A) که مرتب‌سازی درجی را اجرا می‌کند در مقابل یک کامپیوتر کندتر (کامپیوتر B) که مرتب‌سازی ادغام را اجرا می‌کند، قرار دهیم. هر کدام از آنها باید آرایه‌ای از یک میلیون عدد را مرتب کند. فرض کنید که کامپیوتر A یک میلیارد

---

1. *efficiency*
2. *Insertion Sort*
3. *Merge Sort*

دستورالعمل در هر ثانیه اجرا می‌کند و کامپیوتر B فقط 10 میلیون دستورالعمل در هر ثانیه اجرا می‌کند، بنابراین کامپیوتر A، 100 برابر سریعتر از کامپیوتر B در توان محاسباتی اولیه است. حتی برای برجسته‌تر کردن این تفاوت فرض کنید که ماهرترین برنامه‌نویس جهان مرتب‌سازی درجی را به زبان ماشین برای کامپیوتر A کد می‌کند و کد حاصل به $2n^2$ دستورالعمل برای مرتب کردن n عدد نیاز دارد. (در اینجا $c_1 = 2$). در سمت دیگر، مرتب‌سازی ادغام برای کامپیوتر B توسط یک برنامه نویس متوسط کد شده است که از زبانی سطح بالا با کامپایلری ناکارآمد استفاده می‌کند، با کد حاصل که $50nlgn$ دستورالعمل نیاز دارد ($C_2=50$). برای مرتب‌سازی یک میلیون عدد، کامپیوتر A

$$\frac{2.(10^6)^2 \text{ دستورالعمل}}{10^9 \text{ ثانیه/دستورالعمل}} = 2000 \text{ ثانیه}$$

زمان صرف می‌کند، در حالیکه کامپیوتر B

$$\frac{50.10^6 lg10^6 \text{ دستورالعمل}}{10^7 \text{ ثانیه / دستورالعمل}} \approx 100 \text{ ثانیه}$$

زمان صرف می‌کند.

با استفاده از الگوریتمی که زمان اجرایش کندتر رشد می‌کند حتی با یک کامپایلر ضعیف، کامپیوتر B، 20 برابر سریعتر از کامپیوتر A عمل می‌کند! وقتی که 10 میلیون عدد را مرتب می‌کنیم مزیت مرتب‌سازی ادغام بسیار مشخص‌تر است: در حالیکه مرتب‌سازی درجی تقریباً $2/3$ روز زمان صرف می‌کند مرتب سازی ادغام زیر 20 دقیقه زمان صرف می‌کند. در حالت کلی، همانطور که سایز مسئله افزایش می‌یابد مزیت نسبی مرتب‌سازی ادغام نیز افزایش می‌یابد.

الگوریتم‌ها و تکنولوژی‌های دیگر

مثال بالا نشان می‌دهد که الگوریتم‌ها مانند سخت‌افزار کامپیوتر، یک تکنولوژی[1] هستند. کارایی کل سیستم به انتخاب الگوریتم‌های کارآ به اندازه انتخاب سخت‌افزار سریع وابسته است. همزمان که پیشرفت‌های سریعی در دیگر تکنولوژی‌های کامپیوتر صورت می‌گیرد این پیشرفت‌ها در الگوریتم‌ها هم به همان اندازه انجام می‌پذیرد.

ممکن است شگفت زده شوید که آیا بدرستی الگوریتم‌ها در کامپیوترهای معاصر جهت پیشبرد تکنولوژی‌های پیشرفته دیگر تا این اندازه مهم هستند، تکنولوژی‌هایی مانند:

- سخت‌افزار با سرعت ساعت بالا، عملکرد پردازش موازی، و معماری‌های فوق عددی،
- رابط گرافیکی بصری و آسان کاربر ($GUIs$)،
- سیستم‌های شیء گرا، و

---

1. technology

- شبکه بندی ناحیه محلی و ناحیه گسترده.

پاسخ مثبت است گرچه کاربردهایی وجود دارند که در سطح کاربردی صریحاً به محتوای الگوریتم نیاز ندارند (برای مثال، بعضی کاربردهای ساده مبتنی بر وب)، بیشتر آنها به درجه‌ای از محتوای الگوریتمی نیاز دارند. برای مثال، یک سرویس مبتنی بر وب را در نظر بگیرید که تعیین می‌کند چطور باید مسیری از یک مکان به مکان دیگر را پیمود. (اکنون که این مطلب نوشته می‌شود سرویس‌های متعدد این چنین وجود دارند.) پیاده‌سازی آن به سخت‌افزار سریع، یک رابط گرافیکی کاربر، شبکه بندی ناحیه گسترده و همچنین شاید به شیء گرایی وابسته باشد. ولی، همچنین به الگوریتم‌هایی برای عملکردهای مشخص نیاز دارد، مانند مسیریابی (احتمالاً با استفاده از الگوریتم کوتاهترین مسیرها)، پردازش نقشه‌ها، و تغییر آدرسها.

بعلاوه حتی کاربردی که به محتوای الگوریتمی در سطح کاربردی نیاز ندارد به شدّت به الگوریتم‌ها وابسته است. آیا این کاربرد به سخت‌افزار سریع وابسته است؟ طرح سخت‌افزاری از الگوریتم‌ها استفاده کرده است. آیا این کاربرد وابسته به رابط‌های گرافیکی کاربر است؟ طراحی هر *GUI* به الگوریتم‌ها وابسته است. آیا این کاربرد به شبکه بندی وابسته است؟ مسیریابی در شبکه‌ها به شدّت به الگوریتم‌ها وابسته است. آیا این کاربرد به زبانی غیر از کد ماشین نوشته شده است؟ بعداً بوسیله یک کامپایلر، مفسّر یا اسمبلر، که همه آنها بطور گسترده‌ای از الگوریتم‌ها استفاده می‌کنند، پردازش می‌شود. الگوریتم‌ها در هسته اکثر تکنولوژیهایی که در کامپیوترهای معاصر استفاده شده‌اند، قرار دارند.

بعلاوه، با ظرفیت همواره در حال افزایش کامپیوترها، از آنها در حل مسائل بزرگتر از قبل استفاده می‌کنیم. همانطور که در فوق هنگام مقایسه بین مرتب‌سازی درجی و مرتب‌سازی ادغام دیدیم این مورد در سایزهای بزرگتر مسئله وجود دارد که تفاوت در کارآیی بین الگوریتم‌ها بطور خاص معلوم می‌شود.

داشتن پایه‌ای قوی از تکنیک و دانش الگوریتمی خصوصیتی است که برنامه نویسان حقیقتاً ماهر را از مبتدیان مجزا می‌سازد. با تکنولوژی محاسباتی جدید می‌توانید بعضی از امور را بدون اینکه مطالب زیادی راجع به الگوریتم‌ها بدانید به انجام برسانید، اما با داشتن زمینه‌ای خوب در الگوریتم‌ها می‌توانید بسیار بهتر عمل کنید.

تمرین‌ها

۱ – ۱.۲ نمونه‌ای از یک کاربرد که به محتوای الگوریتمی در سطح کاربردی نیاز دارد ارائه دهید و درباره عملکرد الگوریتم‌های مربوط بحث کنید.

۲ – ۱.۲ فرض کنید که پیاده‌سازی مرتب‌سازی درجی و مرتب‌سازی ادغام را روی ماشینی یکسان با هم مقایسه می‌کنیم. برای ورودیهای با سایز n مرتب‌سازی درجی $8n^2$ گام اجرا می‌کند، در حالیکه مرتب‌سازی ادغام $64n\lg n$ مرحله را اجرا می‌کند. برای چه مقادیری از n مرتب‌سازی درجی،

مرتب‌سازی ادغام را شکست می‌دهد؟

۳ - ۱.۲ کوچکترین مقدار n که به ازای آن الگوریتمی با زمان اجرای $100n^2$ سریعتر از الگوریتمی با زمان اجرای 2^n روی ماشین یکسانی اجرا می‌شود چیست؟

مسائل

۱ - ۱ مقایسه زمان‌های اجرا

برای هر تابع $f(n)$ و زمان t در جدول زیر بزرگترین اندازه n مسئله‌ای که می‌تواند در زمان t حل شود را تعیین کنید. فرض می‌شود که الگوریتم حل مسئله $f(n)$ میکرو ثانیه زمان صرف می‌کند.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

۲ شروع

این فصل شما را با چارچوبی که ما در طول این کتاب برای بررسی تحلیل و طراحی الگوریتم‌ها استفاده خواهیم کرد آشنا می‌کند. این فصل مستقل و کامل است اما شامل ارجاع‌های متعددی به مواردی است که در فصل‌های ۳ و ۴ معرفی خواهند شد. (همچنین شامل حاصلجمع‌های متعددی است). با بررسی الگوریتم مرتب‌سازی درجی برای حل مسئله مرتب‌سازی معرفی شده در فصل ۱ شروع می‌کنیم. یک "شبه کد" تعریف می‌کنیم که بایستی برای خوانندههایی که برنامه نویسی کامپیوتری انجام داده‌اند آشنا باشد و از آن استفاده می‌کنیم تا نشان دهیم که چطور الگوریتمهایمان را تعیین می‌کنیم. با تعیین الگوریتم راجع به این که مرتب‌سازی بدرستی انجام شده بحث می‌کنیم و زمان اجرایش را تحلیل می‌کنیم. این تحلیل، یک علامت گذاری معرفی می‌کند که متمرکز می‌شود بر آنکه چگونه این زمان با تعداد داده‌هایی که باید مرتب شود افزایش می‌یابد. با دنبال نمودن بحث مرتب‌سازی درجی، روش تقسیم و حل را برای طراحی الگوریتم‌ها معرفی می‌کنیم و این روش را برای بهبود الگوریتمی بنام مرتب‌سازی ادغام استفاده می‌کنیم. با تحلیل زمان اجرای مرتب‌سازی ادغام، فصل را به پایان می‌رسانیم.

۲.۱ مرتب‌سازی درجی

اولین الگوریتم ما مرتب‌سازی درجی، مسئله مرتب‌سازی معرفی شده در فصل ۱ را حل می‌کند:

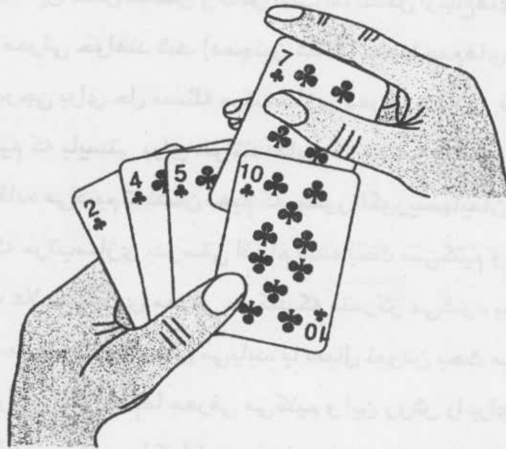
ورودی: یک توالی از n عدد $<a_1, a_2, \ldots, a_n>$.

خروجی: یک جایگشت (ترتیب مجدد) $<a'_1, a'_2, \ldots, a'_n>$ از توالی ورودی بطوریکه $a'_1 \le a'_2 \ldots \le a'_n$.

اعدادی که می‌خواهیم مرتب کنیم، بعنوان کلیدها[1] نیز شناخته می‌شوند.

1. Keys

در این کتاب نوعاً الگوریتمها را بصورت برنامه‌هایی که بشکل شبه کد [1] نوشته شده‌اند توصیف می‌کنیم که از خیلی جهات شبیه C، پاسکال یا جاوا می‌باشند. اگر با یکی از این زبانها آشنا شده باشید در هنگام خواندن الگوریتمهای ما مشکل کمی خواهید داشت. آنچه که شبه کد را که از کد "واقعی" مجزا می‌سازد این است که ما در شبه کد هر روش گویا که روشنتر و مختصرتر است را بکار می‌گیریم تا الگوریتم داده شده را مشخص کنیم. گاهی اوقات روشنترین روش زبان انگلیسی است، بنابراین اگر به یک عبارت یا جمله انگلیسی قرار گرفته درون یک بخش از کد "واقعی" برخوردید تعجّب نکنید. تفاوت دیگر شبه کد و کد واقعی این است که شبه کد نوعاً مرتبط با مباحث مهندسی نرم‌افزار نمی‌باشد. مباحث تجرید داده‌ها، پیمانه‌ای کردن و خطا گردانی اغلب برای بیان اصل الگوریتم بطور مختصرتر، نادیده گرفته می‌شوند.



شکل ۲.۱ مرتب‌سازی دسته‌ای از کارتها با استفاده از مرتب‌سازی درجی

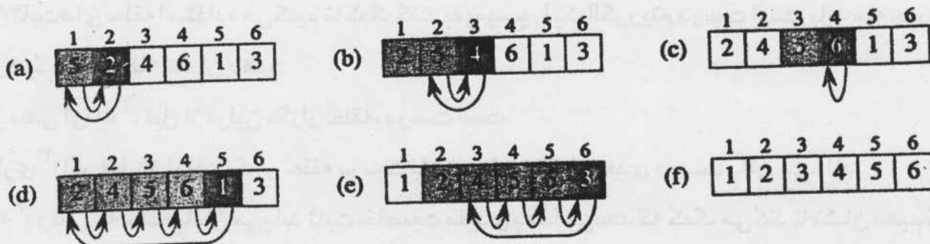
با مرتب‌سازی درجی [2] شروع می‌کنیم که الگوریتم کارآمدی برای مرتب‌سازی تعداد کمی از داده‌ها است. مرتب‌سازی درجی به روشی که اکثر مردم، دسته‌ای از کارتهای بازی را مرتب می‌کنند عمل می‌کند. با دست چپ خالی شروع می‌کنیم و کارت‌ها به پشت روی میز قرار دارند. سپس در هر زمان کارتی را از روی میز برداشته و آن را در مکان صحیح در دست چپ قرار می‌دهیم. برای یافتن مکان صحیح یک کارت، آن را با هر کدام از کارتها که از قبل در دست است، از راست به چپ مقایسه می‌کنیم، مانند شکل ۲.۱. همواره کارتهای نگهداشته شده در دست چپ مرتب شده‌اند و این کارتها در ابتدا، کارتهای بالای دسته روی میز بوده‌اند.

شبه کد ما برای مرتب‌سازی درجی بصورت روالی بنام *INSERTION-SORT* ارائه شده است که

1. *Pseudocode*

2. *Insertion Sort*

بعنوان پارامتر، آرایه $A[1\ ..\ n]$ شامل یک توالی بطول n که بایستی مرتب شود را می‌گیرد. (تعداد عناصر آرایه A یعنی n با $length[A]$ نمایش داده می‌شود.) اعداد ورودی بصورت درجا مرتب شده[1] اند: اعداد در آرایه A باز چینی می‌شوند و حداکثر یک تعداد ثابت از آنها در هر زمان، خارج از آرایه ذخیره شده‌اند. وقتی که *INSETRION-SORT* پایان می‌یابد، ورودی آرایه A شامل توالی خروجی مرتب شده است.



شکل ۲.۲ عملکرد *INSERTION-SORT* روی آرایه $A = \langle 5, 2, 4, 6, 1, 3\rangle$ اندیس‌های آرایه، بالای مستطیل‌ها و مقادیر ذخیره شده در مکانهای آرایه درون مستطیل‌ها نشان داده شده‌اند. *(a)-(e)* تکرارهای حلقه *for* خطوط ۸-۱. در هر تکرار، مستطیل سیاه، کلیدی راکه از $A[j]$ گرفته شده است نگه می‌دارد که با مقادیر مستطیل‌های سایه دار سمت چپ خودش در تست خط ۵ مقایسه می‌شود. پیکان‌های سایه‌دار، مقادیر آرایه راکه در خط ۶، یک مکان به راست انتقال داده می‌شوند نشان می‌دهند و پیکان‌های سیاه جایی راکه کلید در خط ۸ به آن انتقال می‌یابد مشخص می‌کنند. *(f)* آرایه مرتب شده نهایی.

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 .. j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6             do A[i + 1] ← A[i]
7                i ← i − 1
8          A[i + 1] ← key
```

ثابت‌های حلقه و صحّت مرتب‌سازی درجی

شکل ۲.۲ نشان می‌دهد که چگونه این الگوریتم برای $\langle 5, 2, 4, 6, 1, 3\rangle$ عمل می‌کند. اندیس j، کارت جاری قرار داده شده در دست را نشان می‌دهد. در آغاز هر تکرارِ حلقه بیرونی *for* که با j اندیس‌دهی شده است زیر آرایه شامل عناصر $A[1\ ..\ j\text{-}1]$ دسته مرتب شده فعلی را تشکیل می‌دهد و عناصر $[j\text{+}1\ ..\ n]$، مربوط به دسته‌ای از کارتها هستند که هنوز روی میز قرار دارند. در حقیقت،

1. Sorted in place

عناصر $A[1\,..\,j-1]$ عناصری هستند که در ابتدا در مکانهای 1 تا $j-1$ قرار داشته‌اند اما اکنون بصورت مرتب شده می‌باشند. این ویژگی‌های $A[1\,..\,j-1]$ را بطور رسمی بصورت یک **ثابت حلقه**[1] بیان می‌کنیم:

در شروع هر تکرار حلقه *for* خطوط ۸-۱، زیر آرایه $A[1\,..\,j-1]$ شامل عناصر ابتدایی $A[1\,..\,j-1]$، امّا بصورت مرتب می‌باشد.

از ثابت‌های حلقه استفاده می‌کنیم تا کمک کنند بفهمیم چرا یک الگوریتم درست است. باید سه چیز را درباره یک ثابت حلقه نشان دهیم:

مقدار دهی اولیه[2]: قبل از اولین تکرار حلقه، درست است.

نگهداری[3]: اگر قبل از اولین تکرار حلقه درست باشد، قبل از تکرار بعدی درست باقی می‌ماند.

خاتمه[4]: وقتی که حلقه پایان می‌یابد ثابت خاصیت مفیدی به ما می‌دهد که کمک می‌کند تا نشان دهیم که الگوریتم درست است.

وقتی که دو خصوصیت اول بر قرار باشند ثابت حلقه قبل از هر تکرار حلقه، درست است. به تشابه آن با استقرای ریاضی توجه کنید که در آن برای اثبات برقراری یک خصوصیّت، یک حالت پایه و یک گام استقرایی را ثابت می‌کنید. در اینجا نشان دادن این که ثابت حلقه قبل از اولین تکرار برقرار است، شبیه حالت پایه و نشان دادن این که ثابت حلقه از یک تکرار به تکرار دیگر بر قرار است، شبیه گام استقرایی است.

خصوصیّت سوم شاید مهمترین خصوصیّت باشد، زیرا داریم از ثابت حلقه برای نشان دادن صحّت الگوریتم استفاده می‌کنیم. همچنین این عمل با استفاده معمول از استقرای ریاضی که در آن، گام استقرایی بطور نامتناهی استفاده می‌شود متفاوت است؛ در اینجا وقتی که حلقه تمام شد استقرا را متوقف می‌کنیم.

بگذارید تا ببینیم که چگونه این خصوصیّات برای مرتب‌سازی درجی بر قرارند.

مقدار دهی اولیه: با نشان دادن اینکه ثابت حلقه قبل از اولین تکرار حلقه بر قرار است، یعنی وقتی که $j = 2$، آغاز می‌کنیم.[5] بنابراین زیرآرایه $A[1\,..\,j-1]$، تنها شامل تک عنصر $A[1]$ است که در حقیقت عنصر اولیّه در $A[1]$ است. بعلاوه، این زیرآرایه مرتب شده است (البته بطور بدیهی)، که نشان می‌دهد ثابت حلقه قبل از اولین تکرار حلقه برقرار است.

نگهداری: بعد از آن به خاصیّت دوم می‌پردازیم: نشان دادن این که هر تکرار، ثابت حلقه را حفظ می‌کند. بطور غیر رسمی، بدنه حلقه *for* بیرونی با انتقال $A[j-1]$ و $A[j-2]$ و $A[j-3]$ و مانند آن به اندازه یک مکان به راست، کار می‌کند، تا زمانیکه مکان مناسب $A[j]$ پیدا شود (خطوط ۷-۴) که در این

---

1. *Loop invariant*
2. *Initialization*
3. *Maintenance*
4. *Termination*

۵- وقتی که حلقه مورد نظر، حلقه *for* است لحظه‌ای که ثابت حلقه را درست قبل از اولین تکرار حلقه چک می‌کنیم، بلافاصله بعد از انتساب اولیه به متغیر شمارنده حلقه و درست قبل از اولین تست در سر آمد حلقه است. در مورد *INSERTION-SORT* این زمان پس از انتساب 2 به متغیر j، اما قبل از اولین تستِ اینکه آیا $j \leq length[A]$، است.

مکان $A[j]$ درج می‌شود (خط ۸) رفتار رسمی‌تر خصوصیّت دوم سبب می‌شود تا یک ثابت حلقه برای حلقه *while* "درونی" بیان کرده و نشان دهیم. گرچه در این مرحله ترجیح می‌دهیم که بطور دقیق به این مسئله نپردازیم و بنابراین به همان تحلیل غیر رسمی‌مان برای نشان دادن بر قراری خصوصیّت دوم برای حلقه بیرونی اکتفا می‌کنیم.

خاتمه: سرانجام بررسی می‌کنیم وقتی که حلقه به پایان می‌رسد چه اتفاقی می‌افتد. برای مرتب‌سازی درجی، حلقه *for* بیرونی وقتی که j از n بیشتر می‌شود پایان می‌یابد، بعبارت دیگر وقتی که $j = n+1$ با جایگزینی $n+1$ برای j در بیان ثابت حلقه، داریم: زیرآرایه $A[1\,..\,n]$ شامل عناصری است که در ابتدا در $A[1\,..\,n]$ بوده‌اند، اما بصورت مرتب شده. اما زیرآرایه $A[1\,..\,n]$ کلّ آرایه است! از این رو کلّ آرایه کامل، مرتب شده است بدین معنی که الگوریتم درست است.

بعدها از روش ثابت‌های حلقه در این فصل و فصل‌های دیگر به همین ترتیب برای نشان دادن درستی الگوریتم‌ها استفاده خواهیم کرد.

قراردادهای شبه کد

از قراردادهای زیر در شبه کد مان استفاده می‌کنیم:

۱. تورفتگی بر ساختار بلاک دلالت می‌کند. برای مثال، بدنه حلقه *for* که در خط ۱ شروع می‌شود شامل خطوط ۸ - ۲ است، و بدنه حلقه *while* که در خط ۵ شروع می‌شود شامل خطوط ۷ - ۶ و نه خط ۸، می‌باشد. این شکل تورفتگی در مورد عبارات *if-then-else* نیز به همان ترتیب بکار می‌رود. استفاده از تورفتگی بجای نشانگرهای قراردادی ساختار بلاک، مانند عبارت **begin** و **end**، در حالیکه وضوح برنامه را حفظ و یا حتّی افزایش می‌دهد، پیچیدگی آن را نیز کاهش می‌دهد.[1]

۲. ساختارهای حلقه‌ای *while*، *for* و *repeat* و ساختارهای شرطی *if*، *then* و *else* دارای تفاسیری مشابه با آنچه که در پاسکال است می‌باشند[2]. هر چند در مورد حلقه‌های *for* تفاوت ظریفی وجود دارد: در پاسکال مقدار متغیر شمارندهٔ حلقه در خروج از حلقه تعریف نشده می‌باشد اما در این کتاب، شمارنده حلقه مقدارش را پس از خروج از حلقه حفظ می‌کند. بنابراین بلافاصله بعد از یک حلقه *for* مقدار شمارنده حلقه، مقداری است که اولین‌بار از حدّ حلقه *for* بیشتر شده است.

ازاین خصوصیّت در بحث صحّت مرتب‌سازی درجی استفاده کردیم. سرآمد حلقه *for* در خط ۱ بصورت $for\ j \leftarrow 2\ to\ length[A]$ است و بنابراین وقتی که این حلقه به پایان رسید، $j = length[A]$ (یا بطور معادل، $j = n+1$ زیرا $n = length[A]$).

---

۱ - در زبانهای برنامه‌نویسی واقعی بطور کلی مصلحت نیست که فقط نشانه گذاری برای مشخص کردن ساختار بلاک استفاده شود، زیرا وقتی کد در بین صفحات می‌شکند تشخیص سطوح نشانه گذاری سخت می‌شود.

۲ - اکثر زبانهای دارای ساختار بلاکی، ترکیبهای معادلی دارند، گرچه ممکن است ترکیب دقیق آن با آن چه که در پاسکال است متمایز باشد.

۳. نماد "$\triangleright$" نشان می‌دهد که باقی مانده خط یک توضیح است.

۴. یک انتساب چندگانه به شکل $i \leftarrow j \leftarrow e$ دو متغیر i و j را با مقدار عبارت e انتساب می‌دهد؛ این انتساب باید به طور معادل همانند انتساب $j \leftarrow e$ و به دنبال آن، انتساب $i \leftarrow j$ پنداشته شود.

۵. متغیرها (مانند i ، j و key) برای روال داده شده، محلی هستند. متغیرهای سراسری را بدون اشاره صریح استفاده نخواهیم کرد.

۶. عناصر آرایه بوسیله تعیین اسم آرایه و بدنبال آن اندیس واقع در زوج براکت‌ها، مورد دستیابی قرار می‌گیرند. برای مثال، $A[i]$ عنصر iام آرایه A را نشان می‌دهد. برای نشان دادن بازهٔ مقادیر داخل آرایه از علامت " . . " استفاده می‌شود. بنابراین $A[1\ .\ .\ j]$ زیر آرایه‌ای از A شامل j عنصر $A[1], A[2], \ldots, A[j]$ را نشان می‌دهد.

۷. داده‌های مرکب نوعاً بصورت اشیاء[1] سازمان یافته‌اند که هر کدام از اشیاء از صفات[2] یا فیلدها[3] یی تشکیل شده‌اند. یک فیلد خاص با استفاده از نام فیلد بدنبال نام شیءاش در زوج براکت‌ها مورد دستیابی قرار می‌گیرد. برای مثال، آرایه را بعنوان یک شیء با صفت $length$ که نشان می‌دهد شامل چند عنصر است در نظر می‌گیریم. برای مشخص نمودن تعداد عناصر آرایه A، می‌نویسیم: $length[A]$ گرچه زوج براکت‌ها را، هم برای اندیس‌گذاری آرایه و هم برای صفات شیء استفاده می‌کنیم، معمولاً از محتوای عبارت مشخص می‌شود که کدام تعبیر مورد نظر است.

متغیری که آرایه یا شی‌ای را نشان می‌دهد بعنوان اشاره‌گری به داده تلقی می‌شود که به آرایه یا شیء اشاره می‌کند. برای همه فیلدهای f یک شیء x، انتساب $y \leftarrow x$ موجب می‌شود که $f[y] = f[x]$ بعلاوه، اگر اکنون قرار دهیم $f[x] \leftarrow 3$، بعد از آن فقط $f[x] = 3$ نیست، بلکه به همان شکل، $f[y] = 3$ است. بعبارت دیگر بعد از انتساب $y \leftarrow x$، x و y به شیء یکسانی اشاره می‌کنند. (شیء یکسانی هستند.) گاهی اوقات، یک اشاره‌گر اصلاً به هیچ شیء‌ای اشاره نمی‌کند. در این حالت، به آن مقدار خاص NIL را نسبت می‌دهیم.

۸. پارامترها با مقدار[4] به روال فرستاده می‌شوند: روال فراخوانی شده، یک کپی از پارامترهایش را دریافت می‌کند و اگر مقداری به یک پارامتر انتساب داده شود، تغییری در روال فراخواننده دیده نمی‌شود. وقتی اشیاء فرستاده می‌شوند اشاره گر به داده که شیء را نشان می‌دهد کپی می‌شود، اما فیلدهای اشیاء، کپی نمی‌شوند. برای مثال، اگر x پارامتر روال فراخوانی شده باشد انتساب $x \leftarrow y$ داخل روال فراخوانی شده برای روال فراخواننده قابل رؤیت نیست. ولی انتساب $f[x] \leftarrow 3$ قابل رؤیت می‌باشد.

---

1. objects
2. attributes
3. fields
4. by value

۹. عملگرهای بولی "and" و "or"، عملگرهای مدار کوتاه (سری) [1] می‌باشند. به عبارت دیگر، وقتی عبارت "x and y" را ارزیابی می‌کنیم، ابتدا x را ارزیابی می‌کنیم. اگر x با *FALSE* ارزیابی شد آنگاه کلّ عبارت نمی‌تواند با *TRUE* ارزیابی شود، بنابر این y را ارزیابی نمی‌کنیم. از طرف دیگر اگر x با *TRUE* ارزیابی شود، باید y را ارزیابی کنیم تا مقدار کل عبارت را معین نماییم. بطور مشابه در عبارت "x or y"، عبارت y را فقط اگر x با *FALSE* ارزیابی شود ارزیابی می‌کنیم. عملگرهای سری به ما اجازه می‌دهد تا عبارت بولی را مانند "$x \neq NIL$ and $f[x]=y$" بنویسیم، بدون آنکه زمانیکه x برابر *NIL* است در مورد آنچه در هنگام ارزیابی $f[x]$ روی می‌دهد، نگران باشیم.

تمرین‌ها

۲.۱-۱ با استفاده از شکل ۲.۲ بعنوان یک مدل، عملکرد *INSERTION-SORT* را روی آرایه $A = \langle 31, 42, 59, 26, 41, 58 \rangle$ شرح دهید.

۲.۱-۲ روال *INSERTION-SORT* را بازنویسی کنید تا مرتب‌سازی را بصورت غیر صعودی، بجای ترتیب غیر نزولی انجام دهد.

۲.۱-۳ مسئله جستجو [2] را در نظر بگیرید:

ورودی: یک توالی از n عدد $A=\langle a_1, a_2, \ldots, a_n \rangle$ و یک مقدار v.

خروجی: اندیس i بطوریکه: $v = A[i]$ و یا مقدار خاص *NIL* اگر v در A ظاهر نشود.

شبه کدی برای جستجوی خطی [3] بنویسید که در طول توالی پویش می‌کند و به دنبال v می‌گردد. با استفاده از یک ثابت حلقه، ثابت کنید که الگوریتم شما صحیح است. اطمینان حاصل کنید که ثابت حلقه شما، سه خصوصیّت اصلی را دارا می‌باشد.

۲.۱-۴ مسئله جمع دو عدد صحیح دودویی n بیتی را در نظر بگیرید که در دو آرایه n عنصری A و B ذخیره شده‌اند. جمع دو عدد باید بفرم دودویی در آرایه $(n+1)$ عنصری C ذخیره شود. مسئله را بطور رسمی بیان کنید و شبه کدی برای جمع این دو عدد صحیح بنویسید.

۲.۲ تحلیل الگوریتم‌ها

تحلیل [4] یک الگوریتم به معنای برآورد منابعی است که الگوریتم نیاز دارد. گاهی اوقات منابعی مانند حافظه، پهنای باند ارتباطی، یا سخت‌افزار کامپیوتر منابع اصلی هستند، امّا اغلب این زمان محاسباتی است که می‌خواهیم اندازه بگیریم. بطور کلّی با تحلیل چندین الگوریتم نامزد برای یک

1. *short circuiting*
2. *Searching problem*
3. *Linear search*
4. *Analyzing*

مسئله، کارآمدترین الگوریتم می‌تواند به آسانی مشخص شود. چنین تحلیلی ممکن است بیش از یک الگوریتم نامزد که امکان موفقیّت را دارند مشخص کند، اما معمولاً بسیاری از الگوریتم‌های نامرغوب در پردازش کنار گذاشته می‌شوند.

پیش از آنکه بتوانیم الگوریتمی را تحلیل کنیم باید مدلی از تکنولوژی پیاده‌سازی که استفاده خواهد شد را داشته باشیم، که شامل مدلی برای منابع آن تکنولوژی و هزینه هایش است. در بیشتر موارد در این کتاب یک تک پردازنده عمومی، مدل **ماشین با دستیابی تصادفی** *(RAM)*[1] محاسبه را بعنوان تکنولوژی پیاده‌سازی خود فرض خواهیم کرد و می‌دانیم که الگوریتم‌های ما بصورت برنامه‌های کامپیوتری پیاده‌سازی خواهند شد. در مدل *RAM*، دستورالعمل‌ها بدون هیچ اعمال همزمانی، یکی پس از دیگری اجرا می‌شود. امّا در فصل‌های بعدی فرصتی جهت رسیدگی به مدلهایی برای سخت‌افزار دیجیتالی خواهیم داشت.

به بیان دقیق‌تر، باید به دقت دستورالعمل‌های مدل *RAM* و هزینه هایشان را تعریف کرد. امّا انجام چنین کاری ممکن است خسته کننده باشد و به آگاهی کمی نسبت به طراحی و تحلیل الگوریتم منجر شود. با این وجود باید مراقب باشیم که از مدل *RAM* استفاده‌ی نا بجا نکنیم. برای مثال، اگر یک *RAM*، دستورالعملی داشت که عمل مرتب‌سازی را انجام می‌داد، چه می‌شد؟ آنگاه تنها با یک دستورالعمل می‌توانستیم مرتب‌سازی را انجام دهیم. چنین *RAM* ای غیر واقعی خواهد بود، زیرا کامپیوترهای واقعی دارای چنین دستورالعمل‌هایی نمی‌باشند. بنابراین راهنمای ما، چگونگی طراحی کامپیوترهای واقعی است. مدل *RAM* شامل دستورالعمل‌هایی است که عموماً در کامپیوترهای واقعی یافت می‌شوند: دستورات محاسباتی (جمع، تفریق، ضرب، تقسیم، باقیمانده،کف، سقف)، دستورات انتقال داده (بار کردن، ذخیره کردن، کپی کردن)، دستورات کنترلی (انشعاب شرطی و غیر شرطی، فراخوانی زیر روال و بازگشت). هر یک از چنین دستورالعمل‌هایی مقدار زمان ثابتی را صرف می‌کنند.

انواع داده در مدل *RAM* عدد صحیح و ممیز شناور می‌باشند، گرچه معمولاً در این کتاب نگران دقّت نیستیم، امّا در بعضی کارها دقت الزامی است. همچنین محدودیّتی روی اندازه‌ی هر کلمه داده فرض می‌کنیم. برای مثال، در هنگام کار با ورودیهایی با اندازه n معمولاً فرض می‌کنیم که اعداد صحیح بوسیله $c\lg n$ بیت به ازای ثابت $c\geq 1$ نشان داده می‌شوند. نیاز داریم $c\geq 1$ باشد تا آنکه هر کلمه بتواند مقدار n را نگه دارد، که ما را قادر می‌سازد تا عناصر ورودی را تک به تک فهرست کنیم و c را محدود کنیم تا اندازه کلمه بطور دلخواه رشد نکند. (اگر اندازه کلمه می‌توانست بطور دلخواه رشد کند، می‌توانستیم حجم زیادی از داده را در یک کلمه ذخیره کرده و روی این کلمه تنها در زمان ثابتی کار کنیم – یک مطلب غیر واقع بینانه آشکار.)

کامپیوترهای واقعی شامل دستورالعمل‌هایی هستند که در بالا بیان نشده‌اند و چنین دستورالعمل‌هایی

---

1. random-access machine

بمنزله‌ی ناحیهٔ خاکستری در مدل RAM هستند. برای مثال، آیا توان‌رسانی، یک دستورالعمل با زمان ثابت است؟ در حالت کلی، خیر؛ این دستورالعمل، دستورالعمل‌های متعددی را جهت محاسبه x^y وقتی x و y اعداد حقیقی هستند بکار می‌برد. هر چند در وضعیّت‌های محدود شده، توان رسانی عملی با زمان ثابت است. اکثر کامپیوترها یک دستورالعمل "شیفت به چپ" دارند که در زمان ثابت، بیتهای یک عدد صحیح را k مکان به چپ شیفت می‌دهد. در اکثر کامپیوترها، شیفت دادن بیتهای یک عدد صحیح یک مکان به چپ، معادل ضرب آن عدد در 2 است. شیفت بیت‌ها k مکان به چپ معادل ضرب آن عدد در 2^k است. بنابراین چنین کامپیوترهایی می‌توانند 2^k را در یک دستورالعمل با زمان ثابت، بوسیله شیفت عدد صحیح 1 به اندازه k مکان به چپ، تا زمانیکه k بیشتر از تعداد بیت‌های یک کلمه‌ی کامپیوتر نباشد محاسبه کنند. سعی خواهیم کرد تا از چنین ناحیه‌های خاکستری در مدل RAM اجتناب کنیم امّا محاسبه‌ی 2^k را بعنوان یک عمل با زمان ثابت، وقتی که k عدد صحیح مثبت به اندازه کافی کوچک است، در نظر می‌گیریم.

در مدل RAM، سعی به مدل کردن سلسله مراتب حافظه که در کامپیوترهای معاصر رایج است نمی‌کنیم. بعبارت دیگر، *حافظه‌های پنهان*[1] یا حافظهٔ مجازی (که اغلب با *صفحه‌بندی بر حسب نیاز*[2] پیاده‌سازی می‌شود) را مدل نمی‌کنیم. مدل‌های محاسباتی متعددی سعی می‌کنند تا تأثیرات سلسله مراتب حافظه که گاهی اوقات در برنامه‌های واقعی روی ماشین‌های واقعی با اهمیت هستند را توضیح دهند. مسائل معدودی در این کتاب، اثرات سلسله مراتب حافظه را بررسی می‌کنند امّا در اکثر قسمتها، تحلیل‌های این کتاب آنها را در نظر نمی‌گیرند. مدلهایی که سلسله مراتب حافظه را شامل می‌شوند تقریباً اندکی پیچیده‌تر از مدل RAM هستند به طوری که کار کردن با آنها می‌تواند مشکل باشد. بعلاوه تحلیل‌های مدل RAM معمولاً پیش‌بینی کننده‌هایی عالی از اجرا روی ماشین‌های واقعی هستند.

حتّی تحلیل یک الگوریتم ساده در مدل RAM می‌تواند سؤال برانگیز باشد. ابزار ریاضی مورد نیاز ممکن است شامل ترکیب‌شناسی، نظریه احتمال، مهارت جبری، و توانایی تشخیص مهمترین عبارات در یک فرمول باشد. چون رفتار یک الگوریتم ممکن است برای هر ورودی ممکن متفاوت باشد، به ابزاری جهت خلاصه‌سازی این رفتار بصورت فرمولهایی که به راحتی درک می‌شوند نیاز داریم.

حتی گرچه تنها یک مدل ماشین را برای تحلیل یک الگوریتم مورد نظر انتخاب می‌کنیم، باز هم با انتخابهای زیادی در تصمیم راجع به نحوهٔ بیان تحلیل خود مواجه می‌شویم. روشی را ترجیح می‌دهیم که برای نوشتن و دستکاری ساده باشد، نیازمندیهای منبع یک الگوریتم را نشان دهد، و جزئیات خسته کننده را از بین ببرد.

---

1. Caches

2. demand paging

تحلیل مرتّب‌سازی درجی

زمان صرف شده توسط روال *INSERTION-SORT* به ورودی بستگی دارد: مرتب‌سازی هزار عدد، زمان بیشتری را نسبت به مرتب‌سازی سه عدد صرف می‌کند. بعلاوه، *INSERTION-SORT* می‌تواند مقادیر مختلفی از زمان را برای مرتب‌سازی دو توالی از ورودیها با اندازه یکسان، بسته به اینکه تقریباً آنها تاکنون به چه میزان مرتّب شده‌اند صرف کند. بطور کلّی، زمان صرف شده توسط یک الگوریتم با توجه به اندازه ورودی رشد می‌کند، بنابراین مرسوم است که زمان اجرای یک برنامه را بصورت تابعی از اندازه ورودی‌اش تعریف کنیم. برای انجام این کار لازم است اصطلاحات "زمان اجرا" و "اندازه ورودی" را دقیق‌تر تعریف کنیم.

بهترین تصور برای **اندازه ورودی**، به مسئله‌ای که در حال بررسی است بستگی دارد. برای اکثر مسائل مانند مرتب‌سازی یا محاسبه تبدیلات گسسته فوریه، طبیعی‌ترین اندازه‌گیری، اندازه‌گیری تعداد عناصر در ورودی است - برای مثال، آرایه‌ای با اندازه n برای مرتب‌سازی. برای اکثر مسائل دیگر مانند ضرب دو عدد صحیح، بهترین اندازه‌گیری اندازه ورودی، تعداد کل بیتهایی است که جهت نمایش ورودی بصورت دودویی معمولی لازم می‌باشند. گاهی اوقات، مناسب‌تر آن است که اندازه ورودی به جای یک عدد با دو عدد بیان شود. برای مثال، اگر ورودی یک الگوریتم، یک گراف باشد اندازه ورودی می‌تواند بوسیله تعداد رئوس و یالهای گراف بیان شود. در هر مسئله‌ای که بررسی می‌کنیم، نشان خواهیم داد که از چه نوع اندازه‌گیری برای اندازه ورودی استفاده می‌شود.

زمان اجرای یک الگوریتم روی یک ورودی خاص برابر تعداد اعمال یا "گام‌ها"ی اجرا شده است. بهتر است تا مقصود یک گام را طوری تعریف کنیم که تا حدّ ممکن مستقل از ماشین باشد. اینک اجازه دهید دیدگاه زیر را بپذیریم. مقدار زمان ثابتی لازم است تا هر خط از شبه کد ما اجرا شود. یک خط ممکن است مقدار زمان متفاوتی نسبت به خط دیگر صرف کند، اما فرض خواهیم کرد که هر اجرای خط iام، زمان c_i را صرف می‌کند که c_i یک ثابت است. این دیدگاه موافق با مدل *RAM* است، و همچنین چگونگی پیاده‌سازی شبه کد روی اکثر کامپیوترهای واقعی را نشان می‌دهد.[1]

در بحث بعدی، بیان ما برای زمان اجرای *INSERTION-SORT* از یک فرمول در هم و برهم شکل می‌گیرد که همه هزینه‌های c_i عبارتها را با یک نمادگذاری بسیار ساده‌تر بکار می‌برد که مختصرتر بوده و آسانتر دستکاری می‌شود. همچنین این نماد گذاری ساده‌تر، تعیین اینکه آیا یک الگوریتم،

---

۱ - در اینجا چند نکتهٔ ظریف وجود دارد. گام‌های محاسباتی که ما به زبان انگلیسی مشخص می‌کنیم، اغلب گونه‌های متفاوت یک روال هستند که زمانی بیشتر از یک مقدار زمان ثابت نیاز دارد. برای مثال، بعداً در این کتاب ممکن است بگوییم «نقاط را براساس مؤلفه x مرتب کنید»، که همانطور که خواهیم دید بیشتر از یک مقدار زمان ثابت را صرف می‌کند. همچنین توجه کنید عبارتی که زیروالی را فراخوانی می‌کند زمان ثابتی را صرف می‌کند، گرچه زیروالی که یکبار فراخوانی شده است ممکن است زمان بیشتری را صرف کند. بعبارت دیگر، فرآیند فراخوانی زیرروال - ارسال پارامترها به آن و غیره - را از فرآیند اجرای زیرروال جدا می‌کنیم.

کارآمدتر از یک الگوریتم دیگر است را آسان می‌کند.

با ارائه روال *INSERTION-SORT* بهمراه "هزینه" زمانی هر عبارت و تعداد دفعاتی که هر عبارت اجرا می‌شود شروع می‌کنیم. برای هر $j = 2, 3, \ldots, n$ که $n = length[A]$، t_j را تعداد دفعاتی که تست حلقه *while* در خط ۵ برای آن مقدارِ j اجرا می‌شود قرار می‌دهیم. وقتی حلقه‌ی *for* یا *while* طبق معمول خارج می‌شود (یعنی بدلیل تست ابتدای حلقه)، عمل تست یک بار بیشتر از بدنه اصلی حلقه اجرا می‌شود. فرض می‌کنیم که توضیحات، عبارتهای قابل اجرا نیستند و بنابراین زمانی را صرف نمی‌کنند.

	INSERTION-SORT(A)	*cost*	*times*
1	**for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2	**do** *key* $\leftarrow A[j]$	c_2	$n-1$
3	▷ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] >$ *key*	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] \leftarrow$ *key*	c_8	$n-1$

زمان اجرای الگوریتم برابر مجموع زمانهای اجرای هر عبارت اجرا شده است؛ یک عبارت که c_i گام برای اجرا صرف می‌کند و n بار اجرا می‌شود، زمانی برابر $c_i n$ را به زمان اجرای کل می‌افزاید.[1] برای محاسبه $T(n)$ یعنی زمان اجرای *INSERTION-SORT* حاصلضرب‌های ستونهای هزینه و تعداد دفعات اجرا را با هم جمع می‌کنیم، که $T(n)$ بصورت زیر بدست می‌آید:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

حتی برای ورودیهایی با یک اندازه داده شده، زمان اجرای الگوریتم ممکن است به ورودی که با آن اندازه داده می‌شود، بستگی داشته باشد. برای مثال، در *INSERTION-SORT* بهترین حالت، زمانی رخ می‌دهد که آرایه قبلاً مرتب شده باشد. سپس برای هر $j = 2, 3, \ldots, n$ در خط ۵ در می‌یابیم وقتی i دارای مقدار اولیّه $j-1$ است، $A[i] \le key$ است. بنابراین برای $j = 2, 3, \ldots, n$ داریم $t_j = 1$، و زمان

۱- این ویژگی لزوماً برای منبعی مانند حافظه برقرار نیست. یک عبارت که m کلمه از حافظه را استفاده می‌کند و n بار اجرا می‌شود، لزوماً mn کلمه از حافظه را در کل مصرف نمی‌کند.

اجرا در بهترین حالت برابر است با

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

این زمان اجرا می‌تواند بصورت $an+b$ بازای ثابت‌های a و b که به هزینه‌های c_i عبارتها بستگی دارند، بیان شود؛ بنابراین **تابعی خطّی**[1] از n است.

اگر آرایه در یک ترتیب معکوس مرتب شده باشد - بعبارت دیگر در یک ترتیب نزولی - بدترین حالت نتیجه می‌شود. باید هر عنصر $A[j]$ را با هر عنصر در تمام زیر آرایه مرتب شده $A[1\,..\,j-1]$ مقایسه کنیم، و بنابراین برای $j = 2, 3, \ldots, n$ داریم $t_j = j$. با توجه به اینکه

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

و

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

درمی‌یابیم که در بدترین حالت، زمان اجرای *INSERTION-SORT* برابر است با

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

این زمان اجرا می‌تواند بصورت an^2+bn+c بازای ثابتهای a و b و c که دوباره به هزینه‌های c_i عبارتها بستگی دارند بیان شود؛ لذا **تابعی درجه دو** از n است.

معمولاً همانند مرتب‌سازی درجی زمان اجرای الگوریتم برای یک ورودی داد شده ثابت است، گر چه در فصل‌های بعد، برخی از الگوریتم‌های "تصادفی" جالب را خواهیم دید که رفتارشان می‌تواند حتی برای یک ورودی ثابت تغییر کند.

تحلیل بدترین حالت و حالت میانگین

در تحلیل مرتب‌سازی درجی، هم بهترین حالت که در آن آرایه ورودی قبلاً مرتب شده بود، و هم بدترین

1. *linear function*

حالت که در آن آرایه‌ی ورودی بصورت معکوس مرتب شده بود، را بررسی کردیم. گرچه در ادامه این کتاب، معمولاً تنها به یافتن **زمان اجرا در بدترین حالت** خواهیم پرداخت، بعبارت دیگر، طولانی‌ترین زمان اجرا برای یک ورودی با اندازه n.

سه دلیل برای این گرایش بیان می‌کنیم:

- زمان اجرای یک الگوریتم در بدترین حالت، یک حد بالا روی زمان اجرا برای هر ورودی است. دانستن این مطلب به ما ضمانت می‌دهد که الگوریتم هرگز زمانی طولانی‌تر از این را صرف نمی‌کند. نیازی به حدس زدنهای بررسی شده و قریب به یقین در مورد زمان اجرا نداریم و امیدواریم که زمان اجرا هرگز بدتر از این نمی‌شود.
- برای بعضی الگوریتم‌ها، اغلب بدترین حالت واقعاً اتفاق می‌افتد. برای مثال، در جستجوی یک پایگاه داده برای قطعه بخصوصی از اطلاعات، بدترین حالت الگوریتم جستجو اغلب وقتی که اطلاعات در پایگاه داده موجود نیست اتفاق می‌افتد. در برخی کاربردهای جستجو، جستجوها برای اطلاعات غیر موجود، ممکن است فراوان باشند.
- اغلب "حالت میانگین" تقریباً به همان بدی بدترین حالت است. فرض کنید بطور تصادفی n عدد را انتخاب می‌کنیم و مرتب‌سازی درجی را بکار می‌بریم. این الگوریتم برای تعیین مکانی جهت درج عنصر $A[j]$ در زیرآرایه‌ی $A[1 \,..\, j-1]$، چه زمانی را صرف می‌کند؟ به طور میانگین، نیمی از عناصر در $A[1 \,..\, j-1]$ کوچکتر از $A[j]$ هستند و نیمی از عناصر بزرگترند. بنابراین، بطور میانگین نیمی از زیرآرایه $A[1 \,..\, j-1]$ را بررسی می‌کنیم، در نتیجه $t_j = j/2$ اگر زمان اجرای حاصله در حالت میانگین را محاسبه کنیم تابعی درجه دو از اندازه ورودی می‌شود، درست مانند زمان اجرا در بدترین حالت.

در بعضی از حالت‌های خاص، علاقه‌مند به زمان اجرای **مورد انتظار**[1] یا زمان اجرا در **حالت میانگین** یک الگوریتم خواهیم شد؛ در فصل ۵، تکنیک **تحلیل احتمالی**[2] را خواهیم دید که بوسیله آن، زمان‌های اجرای مورد انتظار را تعیین می‌کنیم. مشکلی که با انجام یک تحلیل حالت میانگین وجود دارد این است که ممکن است مشخص نباشد چه چیز یک ورودی "میانگین" را برای مسئله‌ای خاص تشکیل می‌دهد. اغلب فرض خواهیم کرد که همه ورودیها با یک اندازه داده شده، احتمال یکسانی دارند. در عمل، این فرض ممکن است نقض شود اما می‌توانیم گاهی اوقات از یک **الگوریتم تصادفی**[3] که انتخابهایی تصادفی انجام می‌دهد برای ممکن ساختن یک تحلیل احتمالی استفاده کنیم.

---

1. expected
2. probabilistic analysis
3. randomized algorithm

مرتبه رشد

از بعضی تجریدهای ساده کننده استفاده کردیم تا تحلیل روال *INSERTION-SORT* را آسان کنیم. ابتدا هزینه واقعی هر عبارت را نادیده گرفته و از ثابت‌های c_i برای بیان این هزینه‌ها استفاده کردیم. سپس مشاهده کردیم که حتّی این ثابتها نیز جزئیات بیشتری را از آنچه واقعاً نیاز داریم به ما می‌دهند: زمان اجرا در بدترین حالت بازای ثابت‌های a و b و c که به هزینه‌های c_i عبارتها بستگی دارند، برابر an^2+bn+c است. بنابراین نه تنها هزینه‌های واقعی عبارتها را نادیده گرفتیم، بلکه هزینه‌های c_i تجریدی را نیز نادیده گرفتیم.

اکنون یک تجرید ساده کننده‌تر را بوجود می‌آوریم. این تجرید، **نرخ رشد**[1] یا **مرتبه رشد**[2] زمان اجرا است که حقیقتاً توجه ما را جلب می‌کند. بنابراین تنها جمله اول و اصلی فرمول را در نظر می‌گیریم (یعنی an^2)، زیرا جمله‌های با مرتبه کمتر برای n های بزرگ، نسبتاً کم اهمیت هستند. همچنین ضریب ثابت جمله اصلی را نادیده می‌گیریم زیرا ضرایب ثابت نسبت به نرخ رشد در تعیین کارآیی محاسباتی، برای ورودیهای بزرگ کم اهمیت‌تر هستند. بنابراین برای مثال می‌گوییم مرتب‌سازی درجی دارای زمان اجرای $\Theta(n^2)$ در بدترین حالت است (تلفظ می‌شود "تِتای مربّع n"). در این فصل، نماد Θ را بصورت غیر رسمی استفاده خواهیم کرد؛ این نماد در فصل ۳ دقیقاً تعریف خواهد شد.

معمولاً یک الگوریتم را کارآمدتر از الگوریتم دیگر در نظر می‌گیریم اگر زمان اجرای آن در بدترین حالت مرتبه رشد کوچکتری داشته باشد. بدلیل ضرایب ثابت و جملات با مرتبه‌ی کمتر، این ارزیابی ممکن است برای ورودیهای کوچک اشتباه باشد. اما برای ورودیهای به اندازه کافی بزرگ، یک الگوریتم با $\Theta(n^2)$، در بدترین حالت سریعتر از یک الگوریتم با $\Theta(n^3)$ اجرا خواهد شد.

تمرین‌ها

۱ - ۲.۲ تابع $n^3/1000-100n^2-100n+3$ را بر حسب نماد Θ بیان کنید.

۲ - ۲.۲ مرتب‌سازی n عدد ذخیره شده در آرایه A را در نظر بگیرید که ابتدا کوچکترین عنصر A را یافته و آن را با عنصر $A[1]$ تعویض می‌کنیم. سپس دومین عنصر کوچکتر A را یافته و آن را با $A[2]$ تعویض می‌کنیم. این روند را برای $n-1$ عنصر اول A ادامه می‌دهیم. شبه کدی برای این الگوریتم، که به مرتب‌سازی انتخابی معروف است بنویسید. این الگوریتم چه ثابت حلقه‌ای را نگه می‌دارد؟ چرا لازم است این الگوریتم فقط برای $n-1$ عنصر اول، به جای همه n عنصر اجرا شود؟

زمان اجرای مرتب‌سازی انتخابی را در بدترین حالت و بهترین حالت با نماد Θ بیان کنید.

۳ - ۲.۲ دوباره جستجوی خطی را در نظر بگیرید (تمرین ۳ - ۲.۱ را ملاحظه نمایید). بطور میانگین چند عنصر از توالی ورودی لازم است بررسی شود؟ با این فرض که بطور یکسان احتمال دارد هر یک

---

1. *rate of growth* 2. *order of growth*

از عناصر آرایه، عنصر در حال جستجو باشد. زمانهای اجرای جستجوی خطی در حالت میانگین و بدترین حالت برحسب نماد Θ چیست؟ پاسخ‌های خود را توجیه کنید.

۲.۲ - ۴ چطور می‌توانیم تقریباً هر الگوریتم را تغییر دهیم تا زمان اجرای خوبی در بهترین حالت داشته باشد؟

۲.۳ طراحی الگوریتم‌ها

روشهای زیادی برای طراحی الگوریتم‌ها وجود دارد، مرتب‌سازی درجی از یک روش افزایشی[1] استفاده می‌کند: پس از مرتب‌سازی زیرآرایه $A[1 \; . \; . \; j\text{-}1]$، تک عنصر $A[j]$ را در مکان مناسبش درج می‌کنیم که به زیر آرایه مرتب شده $A[1 \; . \; . \; j\text{-}1]$ منجر می‌شود.

در این بخش، روش طراحی دیگری بنام "تقسیم و حل" را مورد بررسی قرار می‌دهیم. از تقسیم و حل استفاده خواهیم کرد تا یک الگوریتم مرتب‌سازی طراحی کنیم که زمان اجرای آن در بدترین حالت بسیار کمتر از الگوریتم مرتب‌سازی درجی است. یک مزیت الگوریتم‌های تقسیم و حل این است که زمانهای اجرای آنها اغلب با استفاده از تکنیک‌هایی که در فصل ۴ معرفی خواهند شد، به آسانی تعیین می‌شود.

۲.۳-۱ روش تقسیم و حل

ساختار الگوریتم‌های مفید زیادی، بازگشتی است: برای حل یک مسئله مفروض، آنها خودشان را یک یا چند بار بطور بازگشتی فراخوانی می‌کنند تا دقیقاً به زیرمسائل مربوطه رسیدگی کنند. این الگوریتم‌ها نوعاً از روش تقسیم و حل[2] پیروی می‌کنند: آنها مسائل را به چندین زیرمسئله که مشابه مسئله اصلی هستند اما اندازه کوچکتری دارند می‌شکنند، بطور بازگشتی زیرمسائل را حل می‌کنند و سپس این حل‌ها را جهت ایجاد یک جواب برای مسئله اصلی ترکیب می‌کنند.

الگوی تقسیم و حل در هر سطح بازگشت شامل سه گام است:

تقسیم مسئله به تعدادی زیرمسئله.

حل زیر مسائل بوسیله حل بازگشتی آنها. امّا اگر اندازه‌های زیرمسائل به اندازه کافی کوچک باشند، زیرمسائل را تنها با روشی ساده و مستقیم حل می‌کنیم.

ترکیب حل‌های زیر مسائل در جوابی برای مسئله اصلی.

الگوریتم مرتب‌سازی ادغام دقیقاً از الگوی تقسیم و حل پیروی می‌کند. بطور شهودی، این الگوریتم بصورت زیر عمل می‌کند:

تقسیم: توالی n عنصری را تقسیم کنید تا به دو زیر توالی $n/2$ عنصری مرتب شود.

حل: با استفاده از مرتب‌سازی ادغام بصورت بازگشتی، دو زیرتوالی را مرتب کنید.

---

1. *incremental*

2. *divide-and-conquer*

ترکیب: دو زیرتوالی مرتب شده را برای تولید پاسخ مرتب شده ادغام کنید.

وقتی توالی که باید مرتب شود طولی برابر ۱ داشته باشد، بازگشت "خاتمه" می‌یابد، که در این حالت کاری برای انجام شدن وجود ندارد چون هر توالی با طول ۱ قبلاً مرتب شده است.

عمل کلیدی الگوریتم مرتب‌سازی ادغام، ادغام دو توالی مرتب شده در گام "ترکیب" است. برای انجام ادغام، از روال کمکی $MERGE(A, p, q, r)$ استفاده می‌کنیم، که A یک آرایه و p، q و r اندیس‌های عناصر آرایه هستند بطوریکه $p \leq q < r$ روال فرض می‌کند که زیرآرایه‌های $A[p\ .\ .\ q]$ و $A[q+1\ .\ .\ r]$ بصورت مرتب شده هستند. این روال آنها را برای تشکیل یک زیر آرایه مرتب شده، *ادغام می‌کند* که جایگزین زیرآرایه فعلی $A[p\ .\ .\ r]$ می‌شود.

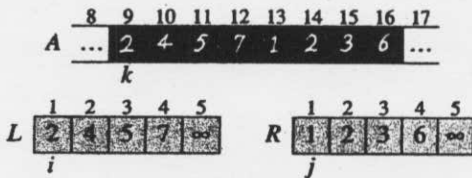
روال $MERGE$ ما زمان $\Theta(n)$ را صرف می‌کند، که $n = r\text{-}p+1$ تعداد عناصر ادغام شده است و بصورت زیر کار می‌کند. به موضوع کارت بازی بر می‌گردیم، فرض کنید دو دسته کارت داریم که به رو، روی میز قرار دارند. هر دسته مرتّب شده است، بطوریکه کوچکترین کارتها در بالا قرار دارند. می‌خواهیم دو دسته را در یک دسته‌ی مرتب شده خروجی ادغام کنیم که باید به پشت روی میز قرار گیرد. گام اصلی ما تشکیل شده است از انتخاب کارت کوچکتر از بین دو کارت بالایی دسته‌های به رو، حذف آن از دسته‌اش (که سبب می‌شود یک کارت جدید در بالا قرار گیرد)، و قرار دادن این کارت به پشت بر روی دسته خروجی. این گام را تا زمانی که یکی از دسته‌های ورودی خالی شود ادامه می‌دهیم، که در آن زمان فقط دسته ورودی باقیمانده را برداشته و بر روی دسته خروجی به پشت قرار می‌دهیم. از نظر محاسباتی، هر گام اصلی زمان ثابتی را صرف می‌کند، زیرا تنها دو کارت بالایی را بررسی می‌کنیم. از آنجا که حداکثر n گام اصلی را اجرا می‌کنیم، ادغام، زمان $\Theta(n)$ را صرف می‌کند.

شبه کد زیر، طرح فوق را پیاده‌سازی می‌کند، امّا با یک تغییر اضافی که از بررسی اینکه آیا در هر گام اصلی دسته‌ای خالی است یا نه، جلوگیری می‌کند. تدبیر مورد نظر به این شکل است که در انتهای هر دسته، یک *کارت نگهبان*[1] قرار دهیم که دارای مقدار خاصی است که از آن برای ساده‌سازی کد خود استفاده می‌کنیم. در اینجا از ∞ بعنوان مقدار نگهبان استفاده می‌کنیم، تا زمانیکه یک کارت با ∞ در بالا قرار گرفت نتواند کارت کوچکتر باشد، مگر آنکه هر دو دسته دارای کارتهای نگهبان خود در بالا باشند. امّا هنگامی که چنین اتفاقی رخ می‌دهد، تمام کارتهای غیر نگهبان در دسته خروجی قرار گرفته‌اند. چون از قبل می‌دانیم که دقیقاً $r\text{-}p+1$ کارت در دسته خروجی قرار خواهد گرفت، می‌توانیم هنگامی که این تعداد گامهای اصلی را اجرا کردیم به کار خود پایان دهیم.

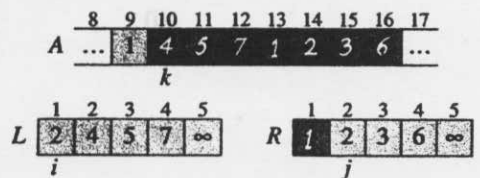
---

1. *sentinel card*

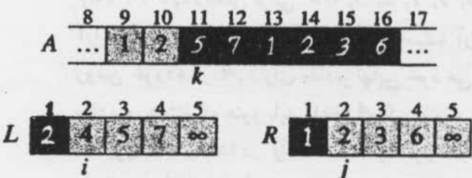
```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14             then A[k] ← L[i]
15                  i ← i + 1
16             else A[k] ← R[j]
17                  j ← j + 1
```



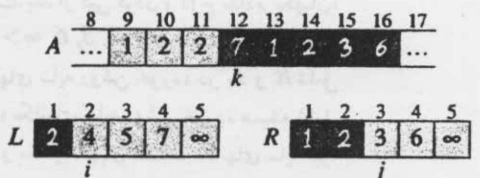
(a)



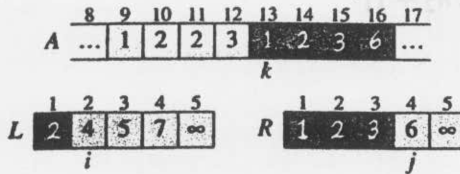
(b)



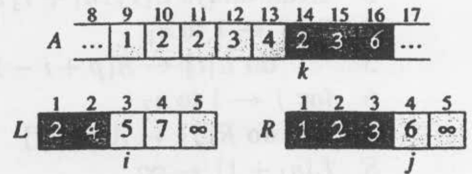
(c)



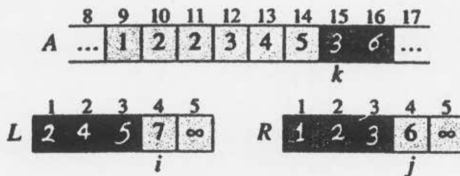
(d)



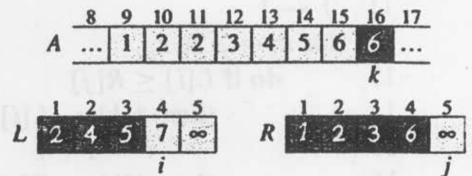
(e)



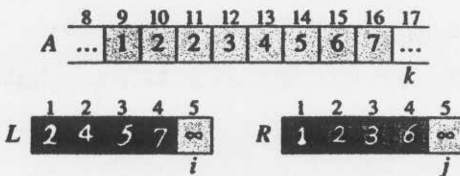
(f)



(g)



(h)



(i)

شکل ۲.۳ عملکرد خطوط ۱۷ - ۱۰ در فراخوانی *MERGE(A, 9, 12, 16)*، وقتی که زیرآرایه *A[9 . . 16]* شامل توالی <2, 4, 5, 7, 1, 2, 3, 6> است. بعد از کپی کردن و درج مقادیر نگهبان، آرایه L شامل <2, 4, 5, 7, ∞> است، آرایه R شامل <1, 2, 3, 6, ∞> است. مکانهای سایه روشن خورده در A دارای مقادیر نهایی خود می‌باشند، و مکانهای سایه روشن خورده در L و R شامل مقادیری هستند که هنوز باید داخل A کپی شوند. روی هم رفته، مکانهای سایه روشن خورده همیشه شامل مقادیری که در ابتدا در *A[9 . . 16]* قرار دارند، به همراه دو مقدار نگهبان هستند. مکانهای سایه تیره خورده در A شامل مقادیری هستند که روی آنها کپی خواهد شد، و مکانهای سایه تیره خورده در L و R شامل مقادیری هستند که تاکنون در داخل A کپی شده‌اند. *(a)-(h)* آرایه‌های A، L و R، و اندیس‌های k، i و j مربوطه قبل از هر تکرار حلقه خطوط ۱۷-۱۲. *(i)* آرایه‌ها و اندیس‌ها در خاتمه. در این مرحله، زیرآرایه *A[9 . . 16]* مرتب شده است و دو مقدار نگهبان در L و R، تنها دو عنصر این آرایه‌ها هستند که داخل A کپی نشده‌اند.

روال *MERGE* بصورت زیر کار می‌کند. خط ۱، طول زیرآرایه $A[p\,..\,q]$ یعنی n_1 و خط ۲، طول زیر آرایه‌ی $A[q+1\,..\,r]$، یعنی n_2 را محاسبه می‌کند. در خطّ ۳، آرایه‌های L و R ("چپ" و "راست") را بترتیب با طولهای n_1+1 و n_2+1 ایجاد می‌کنیم. حلقه *for* خطوط ۵ - ۴، زیرآرایه $A[p\,..\,q]$ را در $L[1\,..\,n_1]$ کپی می‌کند و حلقهٔ *for* خطوط ۷ - ۶، زیرآرایه $A[q+1\,..\,r]$ را در $R[1\,..\,n_2]$ کپی می‌کند. خطوط ۹ - ۸، مقادیر نگهبان را در انتهای آرایه‌های L و R قرار می‌دهند. خطوط ۱۷ - ۱۰ تشریح شده در شکل ۲.۳، $r-p+1$ گام اصلی را با نگهداری ثابت حلقه زیر، اجرا می‌کنند:

در آغاز هر تکرار حلقه *for* خطوط ۱۷ - ۱۲، زیرآرایه $A[p\,..\,k-1]$ شامل $k-p$ عنصر کوچکتر $L[1\,..\,n_1+1]$ و $R[1\,..\,n_2+1]$ بصورت مرتّب شده است. بعلاوه، $L[i]$ و $R[j]$ کوچکترین عناصر آرایه‌های خود هستند که در A کپی نشده‌اند.

باید نشان دهیم این ثابت حلقه قبل از اولین تکرار حلقه *for* خطوط ۱۷ - ۱۲ برقرار است، که هر تکرار حلقه، ثابت را نگه می‌دارد و آنکه این ثابت، یک ویژگی مفید را جهت نشان دادن صحّت، وقتی حلقه خاتمه می‌یابد فراهم می‌آورد.

مقدار دهی اولیه: قبل از اولین تکرار حلقه داریم $k = p$ بنابراین زیرآرایه $A[p\,..\,k-1]$ خالی است. این زیرآرایه خالی شامل $k-p = 0$ عنصر از کوچکترین عنصرهای L و R است، و از آنجا که $i=j=1$، $L[i]$ و $R[j]$ هر دو کوچکترین عنصر آرایه خود هستند که در داخل A کپی نشده‌اند.

نگهداری: برای مشاهده اینکه هر تکرار، ثابت حلقه را نگه می‌دارد اجازه دهید ابتدا فرض کنیم که $L[i] \leq R[j]$. پس $L[i]$ کوچکترین عنصری است که هنوز در A کپی نشده است. از آنجا که $A[p\,..\,k-1]$ شامل $k-p$ عنصر از کوچکترین عناصر است، پس از خط ۱۴ که $L[i]$ داخل $A[k]$ کپی می‌شود، زیرآرایه $A[p\,..\,k]$ شامل $k-p+1$ عنصر از کوچکترین عناصر خواهد بود. افزایش k (در به‌روزرسانی حلقه *for*) و i (در خط ۱۵) به اندازه یک واحد، ثابت حلقه را برای تکرار بعدی بازسازی مجدّد می‌کند. در عوض اگر $L[i] > R[j]$، آنگاه خطوط ۱۷ - ۱۶ عملکرد مناسبی را برای نگهداری ثابت حلقه انجام می‌دهند.

خاتمه: در پایان، $k = r+1$. بنا به ثابت حلقه، زیرآرایه $A[p\,..\,k-1]$ که برابر $A[p\,..\,r]$ است، شامل $k-p = r-p+1$ عنصر از کوچکترین عناصر $L[1\,..\,n_1+1]$ و $R[1\,..\,n_2+1]$ بصورت مرتب است. آرایه‌های L و R با هم شامل $n_1+n_2+2 = r-p+3$ عنصر هستند. همه بجز دو عنصر که بزرگترین عناصر هستند، در A کپی شده‌اند و این دو عنصر، مقادیر نگهبان هستند.

برای مشاهده اینکه روال *MERGE* در زمان $\Theta(n)$ اجرا می‌شود، که $n = r-p+1$ مشاهده کنید که هر کدام از خطوط ۳ - ۱ و ۱۱ - ۸ زمان ثابتی را صرف می‌کند، حلقه‌های *for* خطوط ۷-۴، زمان $\Theta(n_1+n_2)=\Theta(n)$ [1] را صرف می‌کند، و n تکرار از حلقه *for* در خطوط ۱۷ - ۱۲ وجود دارند که هر کدام

۱- در فصل ۳ خواهیم دید که چطور معادله‌های شامل نماد Θ را بطور رسمی تعبیر کنیم.

زمان ثابتی را صرف می‌کند.

اکنون می‌توانیم از روال *MERGE* بعنوان یک زیرروال در الگوریتم مرتب‌سازی ادغام استفاده کنیم. روال *MERGE-SORT(A, p, r)* عناصر زیرآرایه *A[p . . r]* را مرتب می‌کند. اگر $p \leq r$ زیرآرایه حداکثر یک عنصر دارد و بنابراین هم اکنون مرتب شده می‌باشد. در غیر اینصورت گام تقسیم به سادگی، اندیس q را که *A[p . . r]* را به دو زیرآرایه افراز می‌کند محاسبه می‌کند: *A[p . . q]*، شامل $\lceil n/2 \rceil$ عناصر و *A[q+1 . . r]*، شامل $\lfloor n/2 \rfloor$ عناصر.[۱]

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

برای مرتب کردن تمام توالی *A=<A[1], A[2], . . . , A[n]>*، فراخوانی اولیه *MERGE-SORT(A, L, length[A])* را انجام می‌دهیم، که یکبار دیگر $length[A] = n$ است. شکل ۲.۴ عملکرد این روال پایین به بالا را، زمانیکه n توانی از 2 است نشان می‌دهد. الگوریتم تشکیل شده است از ادغام جفت توالی‌های 1 عنصری جهت تشکیل توالی‌های مرتب شده با طول 2 ، ادغام جفت توالی‌های با طول 2 جهت تشکیل توالی‌های با طول 4 و. . .، تا وقتیکه دو توالی با طول $n/2$ ادغام می‌شوند تا توالی مرتب شدهٔ نهایی با طول n را تشکیل دهند.

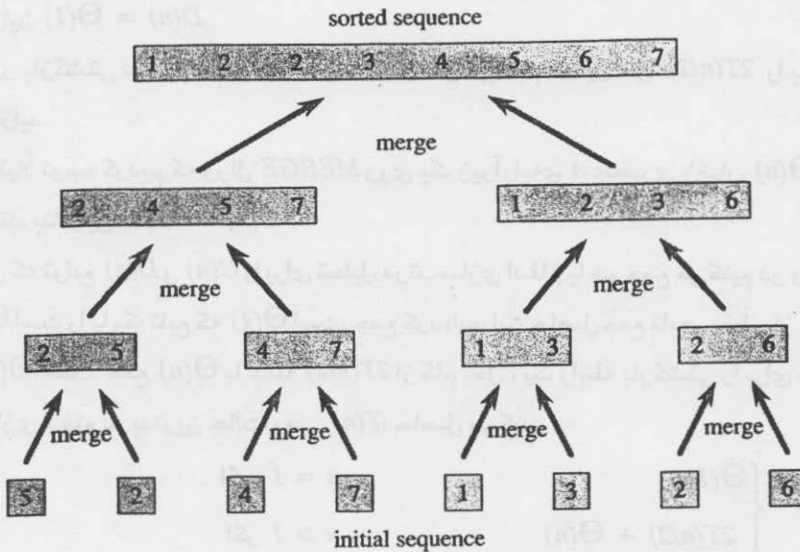
۲.۳.۲ تحلیل الگوریتم‌های تقسیم و حل

هنگامیکه یک الگوریتم شامل یک فراخوانی بازگشتی از خودش است، زمان اجرایش اغلب می‌تواند بوسیله یک **معادله بازگشتی** یا **رابطه بازگشتی** بیان شود، که زمان اجرای کل روی یک مسئله با اندازه n را بر حسب زمان اجرا روی ورودیهای کوچکتر بیان می‌کند. پس می‌توانیم از ابزار ریاضی برای حل رابطه بازگشتی و تهیّه حدودی روی کارآیی الگوریتم استفاده کنیم.

یک بازگشت برای زمان اجرای یک الگوریتم تقسیم و حل مبتنی بر سه گام الگوی اصلی است. مانند قبل، $T(n)$ را زمان اجرا روی یک مسئله با اندازه n قرار می‌دهیم. اگر اندازه مسئله به اندازه کافی کوچک

۱- عبارت $\lceil x \rceil$ به کوچکترین عدد صحیحی که بزرگتر یا مساوی x است دلالت می‌کند، و $\lfloor x \rfloor$ به بزرگترین عدد صحیح کوچکتر یا مساوی x دلالت می‌کند. این نماد گذاریها در فصل ۳ تعریف می‌شوند. ساده‌ترین روش برای اثبات آنکه مقداردهی q با $\lfloor (p+r)/2 \rfloor$ زیرآرایه‌های *A[p . . q]* و *A[q+1 . . r]* را بترتیب با اندازه‌های $\lceil n/2 \rceil$ و $\lfloor n/2 \rfloor$ حاصل می‌کند، این است که چهار حالتی که براساس زوج یا فرد بودن هر کدام از p و r رخ می‌دهند را بررسی کنیم.

باشد، که بازای ثابت c می‌گوییم $n \le c$ راه حل ساده و مستقیم، زمان ثابتی را صرف می‌کند که آن را بصورت $\Theta(1)$ می‌نویسیم. فرض کنید که از تقسیم مسئله، a زیر مسئله بوجود می‌آید، که هر کدام $1/b$ اندازه اصلی هستند. (برای مرتب‌سازی ادغام، a و b هر دو برابر 2 هستند، امّا الگوریتم‌های تقسیم و حل بسیاری را خواهیم دید که در آنها $a \neq b$ است.)



شکل ۲.۴ عملکرد مرتب‌سازی ادغام روی آرایه $A = <5, 2, 4, 7, 1, 3, 2, 6>$. طول توالی‌های مرتب شده که ادغام شده‌اند، با پیشروی الگوریتم از پایین به بالا افزایش می‌یابد.

اگر زمان $D(n)$ را برای تقسیم مسئله به زیرمسئله‌ها و زمان $C(n)$ را برای ترکیب جواب‌های زیرمسئله‌ها برای جواب مسئله اصلی صرف کنیم، رابطه بازگشتی زیر را بدست می‌آوریم

$$T(n) = \begin{cases} \Theta(1) & \text{اگر } n \le c \\ aT(n/b) + D(n) + C(n) & \text{در غیر اینصورت} \end{cases}$$

در فصل ۴، مشاهده خواهیم کرد که چطور رابطه‌های بازگشتی معمولی به این شکل را حل کنیم.

تحلیل مرتب‌سازی ادغام

اگر چه شبه کد *MERGE-SORT* هنگامیکه تعداد عناصر زوج نیستند به درستی کار می‌کند، اگر فرض کنیم اندازه مسئله اصلی توانی از 2 است، تحلیل مبتنی بر رابطه بازگشتی ما ساده می‌شود. پس هر گام تقسیم، دو زیر توالی با اندازه دقیقاً $n/2$ حاصل می‌کند. در فصل ۴، مشاهده خواهیم کرد که این

فرض تأثیری بر مرتبه رشد جواب رابطه بازگشتی ندارد. برای محاسبه رابطه بازگشتی $T(n)$ که زمان اجرای مرتّب‌سازی ادغام روی n عدد در بدترین حالت است، بصورت زیر استدلال می‌کنیم.

مرتب‌سازی ادغام روی تنها یک عنصر، زمان ثابتی را صرف می‌کند. وقتی $n > 1$ عنصر داریم، زمان اجرا را به صورت زیر تجزیه می‌کنیم.

تقسیم: گام تقسیم تنها وسط زیرآرایه را محاسبه می‌کند، این کار زمان ثابتی را صرف می‌کند. بنابراین $D(n) = \Theta(1)$

حل: بطور بازگشتی دو زیر مسئله با اندازه $n/2$ را حل می‌کنیم که این حل، $2T(n/2)$ را به زمان اجرا می‌افزاید.

ترکیب: قبلاً توجه کردیم که روال *MERGE* روی یک زیرآرایه‌ی n عنصری زمان $\Theta(n)$ را صرف می‌کند، بنابراین $C(n) = \Theta(n)$.

وقتی که توابع $D(n)$ و $C(n)$ را برای تحلیل مرتب‌سازی ادغام با هم جمع می‌کنیم در واقع یک تابع که $\Theta(n)$ است را با یک تابع که $\Theta(1)$ است، جمع کرده‌ایم. این حاصل‌جمع تابعی خطی از n به عبارت دیگر $\Theta(n)$، است. جمع $\Theta(n)$ با جمله $2T(n/2)$ از گام "حل"، یک رابطه بازگشتی را برای زمان اجرای مرتب‌سازی ادغام در بدترین حالت، یعنی $T(n)$، حاصل می‌کند:

$$T(n) = \begin{cases} \Theta(1) & \text{اگر } n = 1 \\ 2T(n/2) + \Theta(n) & \text{اگر } n > 1 \end{cases} \qquad (۲.۱)$$

در فصل ۴، "قضیه اصلی" را مشاهده خواهیم کرد که می‌توان از آن برای اثبات اینکه $T(n)$ برابر $\Theta(nlgn)$ است استفاده کرد، که lgn نشان دهندهٔ $log_2 n$ است. چون تابع لگاریتم برای ورودی‌های به اندازه کافی بزرگ از هر تابع خطی آهسته‌تر رشد می‌کند، در بدترین حالت، مرتب‌سازی ادغام با زمان اجرای $\Theta(nlgn)$ از مرتب‌سازی درجی با زمان اجرای $\Theta(n^2)$ بهتر عمل می‌کند.

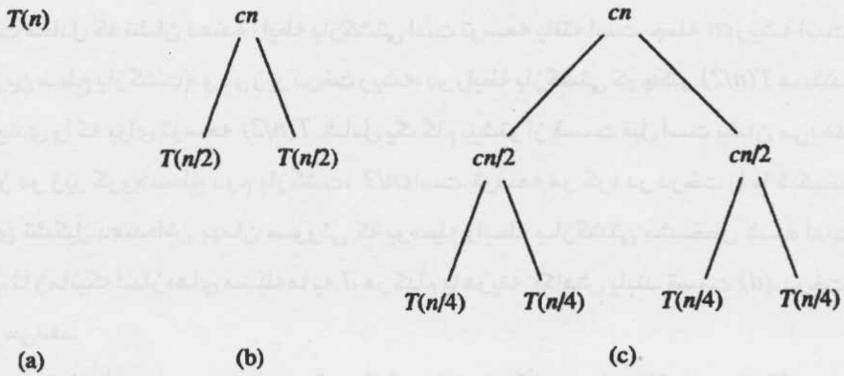
برای درک شهودی اینکه چرا جواب رابطه بازگشتی (۲.۱) برابر $T(n) = \Theta(nlgn)$ می‌شود، به قضیه اصلی نیازی نداریم. اجازه دهید رابطه بازگشتی (۲.۱) را بصورت زیر بازنویسی کنیم:

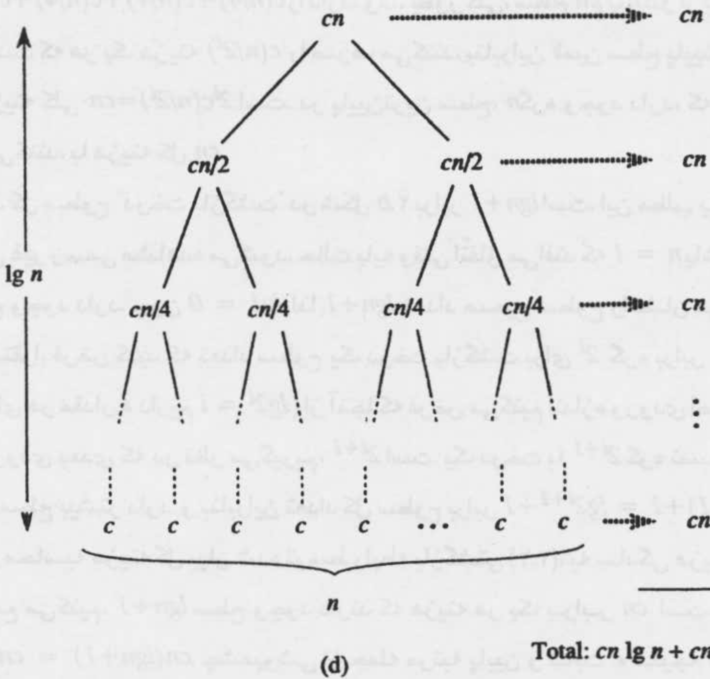
$$T(n) = \begin{cases} c & \text{اگر } n = 1 \\ 2T(n/2) + cn & \text{اگر } n > 1 \end{cases} \qquad (۲.۲)$$

که در آن ثابت c بیانگر زمان مورد نیاز برای حل مسئله‌های با اندازه 1 بعلاوه زمان مصرفی گام‌های تقسیم و ترکیب هر عنصر آرایه است.[۱]

---

۱ - بعید است که یک ثابت، دقیقاً هم زمان مورد نیاز برای حل مسئله‌های با اندازه 1 و هم زمان مصرفی گام‌های تقسیم و ترکیب هر عنصر آرایه را نشان دهد. این مشکل را می‌توانیم به صورت زیر حل کنیم. c را بزرگتر از این زمانها قرار می‌دهیم و می‌دانیم که رابطهٔ بازگشتی، یک حد پایین روی زمان اجرا ارائه می‌دهد. مرتبهٔ هر دو حد، $nlgn$ است، که روی هم رفته زمان $\Theta(nlgn)$ را می‌دهند.





شکل ۲.۵ ساختار یک درخت بازگشت برای رابطه بازگشتی $T(n) = 2T(n/2)+cn$. قسمت (a)، $T(n)$ را نشان می‌دهد که تدریجاً در (b)-(d) جهت تشکیل درخت بازگشت توسعه می‌یابد. درخت کاملاً توسعه یافته در قسمت (d)، $lgn+1$ سطح دارد (یعنی همان طور که نشان داده شده، ارتفاع آن lgn است)، و هر سطح دارای یک هزینه کلی cn است. بنابراین هزینه کل، $cnlgn+cn$ است که برابر $\Theta(nlgn)$ می‌باشد.

شکل ۲.۵ نشان می‌دهد که چگونه می‌توانیم رابطه بازگشتی (۲.۲) را حل کنیم. بمنظور سهولت، فرض می‌کنیم که n توان صحیحی از 2 است. قسمت (a) از شکل، $T(n)$ را نشان می‌دهد که در قسمت (b) به یک درخت معادل که نشان دهنده رابطه بازگشتی است توسعه یافته است. جمله cn ریشه است (هزینه در بالاترین سطح بازگشت)، و دو زیر درخت ریشه، دو رابطهٔ بازگشتی کوچکتر $T(n/2)$ هستند. قسمت (c)، فرآیندی را که برای توسعه $T(n/2)$ شامل یک گام بیشتر از قسمت قبل است نشان می‌دهد. هزینه هر یک از دو زیر گروه سطح دوم بازگشت، $cn/2$ است. توسعه هر گره در درخت را با شکستن آن به قسمتهای تشکیل دهنده‌اش بهمان صورتی که بوسیله رابطه بازگشتی مشخص شده است ادامه می‌دهیم، تا زمانیکه اندازه‌های مسئله‌ها به 1، هر کدام با هزینه c کاهش یابند. قسمت (d)، درخت حاصل را نشان می‌دهد.

سپس، هزینه‌های روی هر سطح درخت را با هم جمع می‌کنیم. سطح بالایی، هزینه کلی cn را دارد، سطح پایینی بعدی دارای هزینه کلی $c(n/2)+c(n/2) = cn$ است، سطح بعد از آن هزینهٔ کلی $c(n/4)+c(n/4)+c(n/4)+c(n/4) = cn$ را دارد، و.... بطور کلی، سطح iام پایینتر از بالاترین سطح، دارای 2^i گره است که هر یک هزینه $c(n/2^i)$ را صرف می‌کنند، بنابراین iامین سطح پایینتر از بالاترین سطح، دارای هزینه کلی $2^i c(n/2^i)=cn$ است. در پایین‌ترین سطح، n گره وجود دارد، که هر یک هزینه c را صرف می‌کنند، با هزینه کل cn.

تعداد کل سطوح "درخت بازگشت" در شکل ۲.۵ برابر $lgn+1$ است. این مطلب به آسانی با یک اثبات استقرایی غیر رسمی مشاهده می‌شود. حالت پایه وقتی اتّفاق می‌افتد که $n = 1$ باشد، در این حالت فقط یک سطح وجود دارد. چون $lg1 = 0$ لذا $lgn+1$ تعداد صحیح سطوح را نشان می‌دهد. اکنون بعنوان فرض استقرا، فرض کنید که تعداد سطوح یک درخت بازگشت برای 2^i گره برابر $lg2^i+1 = i+1$ است. (چون برای هر مقدار i داریم $lg2^i = i$) از آنجا که فرض می‌کنیم اندازه ورودی اصلی توانی از 2 است، اندازه ورودی بعدی که در نظر می‌گیریم، 2^{i+1} است. یک درخت با 2^{i+1} گره نسبت به یک درخت با 2^i گره، یک سطح بیشتر دارد و بنابراین تعداد کل سطوح برابر $(i+1)+1 = lg2^{i+1}+1$ است.

برای محاسبه هزینه کل بیان شده توسط رابطه بازگشتی (۲.۲)، به سادگی هزینه‌های همه سطوح را با هم جمع می‌کنیم. $lgn+1$ سطح وجود دارند که هزینه هر یک برابر cn است، با یک هزینه کلی $cn(lgn+1) = cnlgn+cn$ چشمپوشی از جمله مرتبه پایین و ثابت c نتیجه مطلوب $\Theta(nlgn)$ را حاصل می‌کند.

تمرین‌ها

۱ - ۲.۳ با استفاده از شکل ۲.۴ بعنوان یک مدل، عملکرد مرتب‌سازی ادغام را روی آرایه $A = <3, 41, 52, 26, 38, 57, 9, 49>$ شرح دهید.

۲ - ۲.۳ روال *MERGE* را بازنویسی کنید بطوریکه از مقادیر نگهبان استفاده نکند، در عوض هنگامیکه آرایه L یا R، همه عناصرشان در A کپی شده باشد متوقف شده و سپس باقیمانده آرایه دیگر را در A کپی کند.

۳ - ۲.۳ با استفاده از استقرای ریاضی نشان دهید زمانیکه n توان صحیحی از 2 است، جواب رابطه بازگشتی

$$T(n) = \begin{cases} 2 & \text{اگر } n=2 \\ 2T(n/2) + n & \text{اگر برای } n=2^k ، k>1 \end{cases}$$

برابر $T(n) = nlgn$ است.

۴ - ۲.۳ مرتب‌سازی درجی می‌تواند بصورت یک روال بازگشتی بشرح زیر بیان شود. به منظور مرتب کردن $A[1 \,.\,.\, n]$، آرایه $A[1 \,.\,.\, n-1]$ را بطور بازگشتی مرتب کرده و سپس $A[n]$ را در آرایه مرتب شده $A[1 \,.\,.\, n-1]$ درج می‌کنیم. یک رابطه بازگشتی برای زمان اجرای این نسخه بازگشتی از مرتب‌سازی درجی بنویسید.

۵ - ۲.۳ با مراجعه به مسئله جستجو (تمرین ۳-۲.۱ را ملاحظه نمایید)، مشاهده می‌کنید که اگر توالی A مرتب شده باشد، می‌توانیم عنصر میانی توالی را با v مقایسه کرده و نیمی از توالی که مورد نظر نیست را حذف کنیم.

جستجوی دودویی الگوریتمی است که این روال را هر بار با نصف کردن اندازه قسمت باقیمانده توالی، تکرار می‌کند. برای جستجوی دودویی، یک شبه کد تکراری و یا بازگشتی بنویسید. ثابت کنید زمان اجرای جستجوی دودویی در بدترین حالت برابر $\Theta(lgn)$ است.

۶ - ۲.۳ مشاهده می‌کنید که حلقه *while* خطوط ۷-۵ روال *INSERTION-SORT* در بخش ۲.۱، از جستجوی خطی استفاده می‌کند تا در طول زیرآرایه $A[1 \,.\,.\, j-1]$ (به عقب) پویش کند. آیا می‌توانیم به جای جستجوی خطی، از جستجوی دودویی (تمرین ۵-۲.۳ را ملاحظه نمایید) استفاده کنیم تا زمان کل اجرای مرتب‌سازی درجی در بدترین حالت را با $\Theta(nlgn)$ بهبود بخشیم؟

۷ - ۲.۳* یک الگوریتم با مرتبه زمانی $\Theta(nlgn)$ ارائه نمایید که عدد صحیح x و مجموعهٔ S شامل n عدد صحیح را گرفته، و تعیین کند که آیا دو عنصر در S وجود دارند که حاصلجمع آنها دقیقاً برابر x شود یا خیر؟

مسائل

۱-۲ مرتب‌سازی درجی روی آرایه‌های کوچک در مرتب‌سازی ادغام

با وجود اینکه مرتب‌سازی ادغام در بدترین حالت در زمان $\Theta(nlgn)$ و مرتب‌سازی درجی در بدترین حالت در زمان $\Theta(n^2)$ اجرا می‌شوند، ضرایب ثابت در مرتب‌سازی درجی آن را برای nهای کوچک

سریعتر می‌کنند. بنابراین معقول است وقتی زیرمسائل به اندازه کافی کوچک می‌شوند از مرتب‌سازی درجی در مرتب‌سازی اغام استفاده کنیم. یک تغییر در مرتب‌سازی ادغام را در نظر بگیرید که در آن n/k زیرلیست با طول k با استفاده از مرتب‌سازی درجی، مرتب شده و سپس با استفاده از فرآیند ادغام استاندارد ادغام می‌شوند و k مقداری است که باید مشخص شود.

a. نشان دهید که n/k زیرلیست هر یک با طول k می‌توانند بوسیله مرتب‌سازی درجی در بدترین حالت در زمان $\Theta(n/k)$ مرتب شوند.

b. نشان دهید که زیر لیستها می‌توانند در بدترین حالت در زمان $\Theta(nlg(n/k))$ ادغام شوند.

c. الگوریتم تغییر یافته که در بدترین حالت در زمان $\Theta(nk+nlg(n/k))$ اجرا می‌شود داده شده است، بزرگترین مقدار مجانبی (نماد Θ) k بصورت تابعی از n که برای آن الگوریتم تغییر یافته دارای زمان اجرای مجانبی یکسانی با مرتب‌سازی ادغام استاندارد است، چیست؟

d. در عمل k باید چطور انتخاب شود؟

۲-۲ درستی مرتب‌سازی حبابی

مرتب‌سازی حبابی، یک الگوریتم مرتب‌سازی مشهور است. این الگوریتم با تعویض پی در پی عناصر مجاور که نامرتب هستند کار می‌کند.

```
BUBBLESORT(A)
1  for i ← 1 to length[A]
2      do for j ← length[A] downto i + 1
3          do if A[j] < A[j − 1]
4              then exchange A[j] ↔ A[j − 1]
```

a. A' را خروجی $BUBBLESORT(A)$ در نظر بگیرید. برای اثبات درستی $BUBBLESORT$ لازم است ثابت کنیم این الگوریتم خاتمه می‌یابد و اینکه

$$A'[1] \leq A'[2] \leq \ldots \leq A'[n] \qquad (۲.۳)$$

که در آن $n = length[A]$ است. چه چیز دیگری باید ثابت شود تا نشان دهد که $BUBBLESORT$ عمل مرتب‌سازی را واقعاً انجام می‌دهد؟

دو قسمت بعدی، نامساوی ۲.۳ را ثابت خواهند کرد.

b. یک ثابت حلقه برای حلقه for خطوط ۴ - ۲ بطور دقیق بیان کنید و ثابت کنید که این ثابت حلقه برقرار است. اثبات شما باید از ساختار اثبات ثابت حلقه بیان شده در این فصل استفاده کند.

c. با استفاده از شرط خاتمهٔ ثابت حلقه که در قسمت (b) ثابت شد، یک ثابت حلقه برای حلقه for خطوط ۴-۱ بیان کنید که به شما اجازه دهد تا نامساوی (۲.۳) را ثابت کنید. اثبات شما باید از ساختار اثبات

ثابت حلقه بیان شده در این فصل استفاده کند.

d.زمان اجرای مرتب‌سازی حبابی در بدترین حالت چیست؟ این زمان اجرا در مقایسه با زمان اجرای مرتب‌سازی درجی چگونه است؟

۲-۳ درستی قانون *Horner*

قطعه کد زیر قانون *Horner* را برای ارزشیابی چند جمله‌ای

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n)\cdots)) , \end{aligned}$$

با ضرایب داده شده $a_0, a_1, \ldots, a_n$ و یک مقدار برای x پیاده‌سازی می‌کند:

```
1  y ← 0
2  i ← n
3  while i ≥ 0
4       do y ← a_i + x · y
5          i ← i − 1
```

a.زمان اجرای مجانبی این قطعه کد برای قانون *Horner* چیست؟

b.شبه کدی برای پیاده‌سازی الگوریتم ارزشیابی ساده چند جمله‌ای بنویسید که هر جمله از چند جمله‌ای را از ابتدا محاسبه می‌کند. زمان اجرای این الگوریتم چیست؟ در مقایسه با قانون *Horner* چگونه است؟

c.ثابت کنید که ثابت زیر، یک ثابت حلقه برای حلقه *while* در خطوط ۵ – ۳ است.

در شروع هر تکرار حلقه *while* خطوط ۵ – ۳،

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k .$$

یک حاصل‌جمع بدون جمله‌های برابر صفر بیان کنید. اثبات شما باید از ساختار اثبات ثابت حلقه ارائه شده در این فصل پیروی کند و باید نشان دهید که در پایان، $y = \sum_{k=0}^{n} a_k x^k$.

d.با اثبات اینکه قطعه کد داده شده به درستی، یک چند جمله‌ای مشخص شده بوسیله ضرایب $a_0, a_1, \ldots, a_n$ را ارزشیابی می‌کند به کار خود پایان دهید.

۲-۴ وارونگی

$A[1\ ..\ n]$ را آرایه‌ای از n عدد مجزا در نظر بگیرید. اگر $i < j$ و $A[i] > A[j]$، آنگاه زوج (i, j) یک وارونگی[1] از A خوانده می‌شود.

a. پنج وارونگی از آرایه‌ی $<2, 3, 8, 6, 1>$ را لیست کنید.

b. چه آرایه‌ای با عناصر مجموعه $\{1, 2, \ldots, n\}$ بیشترین وارونگی‌ها را دارد؟ این آرایه چند وارونگی دارد؟

c. چه رابطه‌ای بین زمان اجرای مرتب‌سازی درجی و تعداد وارونگی‌ها در آرایه ورودی وجود دارد؟ پاسخ خود را توجیه کنید.

d. الگوریتمی ارائه دهید که تعداد وارونگی‌ها در یک جایگشت روی n عنصر را در بدترین حالت در زمان $\Theta(n\lg n)$ تعیین کند. (راهنمایی: مرتب‌سازی ادغام را تغییر دهید.)

1. inversion

۳ رشد توابع

مرتبه رشد زمان اجرای یک الگوریتم که در فصل ۲ تعریف شد، توصیف ساده‌ای را از کارآیی الگوریتم ارائه می‌دهد و همچنین ما را قادر می‌سازد تا کارآیی نسبی الگوریتم‌های دیگر را بسنجیم. زمانیکه اندازه ورودی n بقدر کافی بزرگ شود، مرتب سازی ادغام، با زمان اجرای $\Theta(n\lg n)$ در بدترین حالت، بر مرتب سازی درجی که زمان اجرایش در بدترین حالت $\Theta(n^2)$ است غلبه می‌کند. اگر چه می‌توانیم گاهی اوقات زمان اجرای دقیق یک الگوریتم را تعیین کنیم، امّا همانطور که در فصل ۲ برای مرتب سازی درجی انجام دادیم، معمولاً دقت زیاد، ارزش تلاش برای محاسبه آن را ندارد. برای ورودیهای به اندازه کافی بزرگ، ضرایب ثابت و جملات مرتبه پایین‌تر یک زمان اجرای دقیق بوسیله تأثیرات اندازه ورودی، تحت‌الشعاع قرار می‌گیرند.

هنگامیکه اندازه‌های ورودی را برای ساختن مرتبه رشد مناسب زمان اجرا، به اندازه کافی بزرگ در نظر بگیریم، کارآیی مجانبی الگوریتم‌ها را بررسی کرده‌ایم. بعبارت دیگر، معطوف این موضوع می‌شویم که چگونه زمان اجرای یک الگوریتم با اندازه ورودی بصورت حدی افزایش می‌یابد، همان‌طور که اندازه ورودی ، بدون حد افزایش می‌یابد. معمولاً یک الگوریتم که بطور مجانبی کارآمدتر است، برای همه ورودیها به استثنای ورودیهای بسیار کوچک بهترین انتخاب خواهد بود.

این فصل روش‌های استاندارد متعددی را برای ساده سازی تحلیل مجانبی الگوریتم‌ها ارائه می‌دهد. بخش بعد با تعریف انواع مختلف "نمادگذاری مجانبی" که قبلاً نمونه‌ای از آن را در مثالی با نماد Θ دیده‌ایم شروع می‌شود. سپس قراردادهای نمادگذاری متعددی که در سرتاسر این کتاب استفاده شده‌اند ارائه می‌شوند و در نهایت، رفتار توابعی که معمولاً در تحلیل الگوریتم‌ها وجود دارند را بررسی می‌کنیم.

۳.۱ نمادگذاری مجانبی

نمادگذاریهایی را که برای توصیف زمان اجرای مجانبی یک الگوریتم استفاده می‌کنیم بر حسب

توابعی که دامنه هایشان مجموعه‌ای از اعداد طبیعی $N=\{0, 1, 2, ...\}$ است، تعریف می‌شوند. این نمادها برای بیان $T(n)$، یعنی تابع زمان اجرا در بدترین حالت که معمولاً فقط روی اندازه‌های صحیح ورودی تعریف می‌شود، مناسب هستند. هر چند گاهی اوقات مناسب است از نمادگذاری مجانبی به روش‌های مختلف استفاده نابجا کنیم. برای مثال، نمادگذاری به آسانی برای دامنه اعداد حقیقی تعمیم داده می‌شود، یا از سوی دیگر به زیر مجموعه‌ای از اعداد طبیعی محدود می‌شود. امّا مهم آن است که معنای دقیق نمادگذاری را بفهمید تا وقتی از این نمادگذاری استفاده نابجا می‌شود بطور نادرست به کاربرده نشود. این بخش نمادگذاریهای مجانبی اصلی را تعریف می‌کند و همچنین بعضی استفاده‌های نابجای معمول را معرفی می‌کند.

نماد Θ

در فصل ۲، فهمیدیم که زمان اجرای مرتب سازی درجی در بدترین حالت $T(n)=\Theta(n^2)$ است. اجازه دهید معنای این نماد را تعریف کنیم. برای تابع داده شده $g(n)$ مجموعه توابع را با $\Theta(g(n))$ نشان می‌دهیم.

$\Theta(g(n))=\{f(n): n\geq n_0$ ثابت‌های مثبت $n_0 c_2 c_1$ وجود دارند، به طوری که برای همه
$0\leq c_1g(n)\leq f(n)\leq c_2g(n)\}$ [1]

تابع $f(n)$ به مجموعه $\Theta(g(n))$ تعلق دارد اگر ثابت‌های مثبت c_1 و c_2 ای وجود داشته باشد بطوریکه این تابع بتواند برای n به اندازه کافی بزرگ، بین $c_1g(n)$ و $c_2g(n)$ "ساندویچ" شود. چون $\Theta(g(n))$ یک مجموعه است، برای نشان داده اینکه $f(n)$ عضو $\Theta(g(n))$ است، می‌توانیم بنویسیم "$f(n)\in\Theta(g(n))$". در عوض، معمولاً برای بیان این مطلب می‌نویسیم "$f(n)=\Theta(g(n))$". این استفاده نابجا از تساوی برای اشاره کردن به عضویت مجموعه ممکن است در ابتدا گیج کننده باشد اما بعداً در این بخش خواهیم دید که این استفاده مزایایی دارد.

شکل ۳.۱(a) یک تصویر شهودی از توابع $f(n)$ و $g(n)$ را ارائه می‌دهد، که در آن داریم $f(n)=\Theta(g(n))$ برای همه مقادیر n در سمت راست n_0 مقدار $f(n)$ در $c_1g(n)$ یا بالای آن و در $c_2g(n)$ یا پایین آن قرار دارد. بعبارت دیگر، برای تمام $n\geq n_0$ تابع $f(n)$ با یک ضریب ثابت با تابع $g(n)$ برابر است. می‌گوییم $g(n)$ **یک حد قوی مجانبی** [2] برای $f(n)$ است.

تعریف $\Theta(g(n))$ نیازمند این است که هر عضو $f(n)\in\Theta(g(n))$ **بطور مجانبی غیر منفی** [3] باشد، بعبارت دیگر $f(n)$ هنگامیکه n به اندازه کافی بزرگ است، غیر منفی باشد.

---

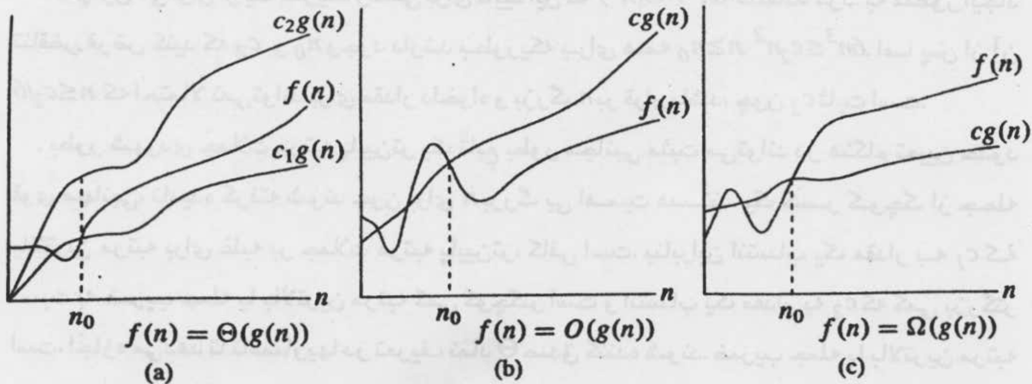
۱ - در نمادگذاری مجموعه، دو نقطه (:) باید بصورت "بطوریکه" خوانده شود.

2. asymptotically tight bound

3. asymptotically nonnegative

(یک تابع **بطور مجانبی مثبت**[1] تابعی است که برای همه nهای به اندازه کافی بزرگ، مثبت است.) در نتیجه خود تابع $g(n)$ باید بطور مجانبی غیر منفی باشد، یا در غیر این صورت مجموعه $\Theta(g(n))$ تهی است. بنابراین فرض خواهیم کرد که هر تابع استفاده شده در نماد Θ بطور مجانبی غیر منفی است. این فرض برای نمادگذاریهای مجانبی دیگر که در این فصل تعریف شده‌اند نیز برقرار است.

در فصل ۲، یک برداشت غیر رسمی از نماد Θ را معرفی کردیم که به مثابه کنار گذاشتن جملات مرتبه پایین‌تر و نادیده گرفتن ضریب اصلی جمله با بالاترین مرتبه بود.



شکل ۳.۱ نمونه‌های گرافیکی نمادهای Θ، O، و Ω. در هر قسمت مقدار n_0 نشان داده شده، مینیمم مقدار ممکن است؛ که هر مقدار بزرگتر نیز کار خواهد کرد. *(a)* نماد Θ یک تابع را در بین ضرایب ثابت محدود می‌کند. می‌نویسیم $f(n)=\Theta(g(n))$ اگر ثابت‌های مثبت n_0 و c_1 و c_2 وجود داشته باشند بطوریکه در سمت راست n_0 مقدار $f(n)$ همواره بین $c_1g(n)$ و $c_2g(n)$ قرار می‌گیرد. *(b)* نماد O یک حد بالا برای یک تابع با یک ضریب ثابت ارائه می‌دهد. می‌نویسیم $f(n)=O(g(n))$ اگر ثابتهای مثبت n_0 و c وجود داشته باشند بطوریکه در سمت راست n_0 مقدار $f(n)$ همیشه در رو یا پایین $cg(n)$ قرار می‌گیرد. *(c)* علامت Ω یک حد پایین برای یک تابع با یک ضریب ثابت ارائه می‌دهد. می‌نویسیم $f(n)=\Omega(g(n))$ اگر ثابت‌های مثبت c و n_0 وجود داشته باشند، بطوریکه در سمت راست n_0 مقدار $f(n)$ همواره در رو یا بالای $cg(n)$ قرار می‌گیرد.

اجازه دهید این شهود را مختصراً با استفاده از تعریف رسمی جهت نشان دادن اینکه $1/2n^2-3n=\Theta(n^2)$ توجیه کنیم. برای انجام این کار، باید ثابت‌های مثبت c_1، c_2 و n_0 را تعیین کنیم بطوریکه برای تمام $n\geq n_0$،

$$c_1n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2n^2$$

با تقسیم بر n^2 داریم:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 .$$

1. asymptotically positive

سمت راست نامساوی می‌تواند برای هر مقدار $n \geq 1$ با انتخاب $c_2 \geq 1/2$ برقرار شود. همچنین سمت چپ نامساوی می‌تواند برای هر مقدار $n \geq 7$ با انتخاب $c_1 \leq 1/14$ بر قرار شود. بنابراین با انتخاب $c_1=1/14$، $c_2=1/2$ و $n_0=7$ می‌توانیم تأیید کنیم که $1/2n^2-3n=\Theta(n^2)$. مسلماً انتخابهای دیگری برای ثابت‌ها وجود دارند، اما مهم این است که یک انتخاب وجود دارد. توجه کنید که این ثابت‌ها به تابع $1/2n^2-3n$ بستگی دارند؛ یک تابع متفاوت متعلق به $\Theta(n^2)$ معمولاً به ثابت‌های متفاوتی نیاز خواهد داشت.

همچنین می‌توان از یک تعریف رسمی برای تأیید این که $6n^3 \neq \Theta(n^2)$ استفاده کرد. به منظور ایجاد تناقض فرض کنید که c_2 و n_0 وجود دارند بطوریکه برای همه $n \geq n_0$ $6n^3 \leq c_2n^2$. اما پس از آن $n \leq c_2/6$ که احتمالاً نمی‌تواند برای مقدار دلخواه و بزرگ n بر قرار باشد، چون c_2 ثابت است.

بطور شهودی جملات مرتبه پایین‌تر یک تابعِ بطور مجانبی مثبت می‌تواند در هنگام تعیین حدود قوی مجانبی، نادیده گرفته شوند چون برای n بزرگ بی اهمیت هستند. یک کسر کوچک از جمله بالاترین مرتبه برای غلبه بر جملات مرتبه پایین‌تر، کافی است. بنابراین انتساب یک مقدار به c_1 که نسبت به ضریب جمله با بالاترین مرتبه کمی کوچکتر است و انتساب یک مقدار به c_2 که کمی بزرگتر است، اجازه می‌دهد تا نامساویها در تعریف نماد Θ صدق کننده شوند. ضریب جمله با بالاترین مرتبه نیز می‌تواند نادیده گرفته شود، زیرا فقط c_1 و c_2 را با یک فاکتور ثابت مساوی با این ضریب تغییر می‌دهد.

بعنوان مثال، تابع درجه دو $f(n)=an^2+bn+c$ را در نظر بگیرید که در آن a، b و c ثابت هستند و $a>0$ با کنار گذاشتن جملات مرتبه پایین‌تر و نادیده گرفتن ثابت، $f(n)=\Theta(n^2)$ می‌شود.

بطور رسمی برای نمایش این مطلب، ثابت‌ها را بصورت $c_1=a/4$، $c^2=7a/4$ و

$$n_0 = 2 \cdot \max((|b|/a), \sqrt{(|c|/a)})$$

می‌گیریم. می‌توانید ثابت کنید که برای همه $n \geq n_0$ $0 \leq c_1n^2 \leq an^2+bn+c \leq c_2n^2$. بطور کلی، برای هر چند جمله‌ای

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

که در آن a_i ها ثابت هستند و $a_d>0$ داریم $p(n)=\Theta(n^d)$ (مسئله ۱-۳ را ملاحظه نمایید).

چون هر ثابت یک چند جمله‌ای درجه صفر است، می‌توانیم هر تابع را بصورت $\Theta(n^0)$، یا $\Theta(1)$ بیان کنیم. امّا این نمادگذاری اخیر یک استفاده مختصراً نابجا است، چون واضح نیست که چه متغیری به بی‌نهایت میل می‌کند.[1] اغلب از نماد $\Theta(1)$ برای دلالت بر یک ثابت یا یک تابع ثابت با توجه به متغیر

---

۱. مشکل واقعی این است که نمادگذاری معمول برای توابع، توابع را از مقادیر متمایز نمی‌کند. در حساب λ پارامترهای یک تابع بطور واضح تعیین می‌شوند:
تابع n^2 می‌تواند بصورت $\lambda n.n^2$ یا حتی $\lambda r.r^2$ نوشته شود. امّا پذیرفتن یک نمادگذاری دقیق‌تر، دستکاریهای جبری را پیچیده می‌کند و بنابراین این استفاده نابجا را جایز می‌شماریم.

استفاده می‌کنیم.

نماد O

نماد O بطور مجانبی یک تابع را از بالا و پایین محدود می‌کند. وقتی که فقط یک حد بالای مجانبی داریم از نماد O استفاده می‌کنیم. برای تابع مفروض $g(n)$ توسّط $O(g(n))$ (تلفظ می‌شود "اُی بزرگ ِ‌ای n" یا گاهی اوقات فقط "اُی ِای n") به مجموعه توابع اشاره می‌کنیم.

$O(g(n)) = \{f(n)$: ثابت های c و n_0 وجود دارندبطوریکه برای همه $n \geq n_0$
$0 \leq f(n) \leq cg(n)\}$

با استفاده از نماد O یک حد بالا روی یک تابع با یک ضریب ثابت ارائـه مـی‌دهیم. شکل ۳.۱(b) شهودی که در پس نماد O است را نشان می‌دهد. برای همه مقادیر n در سمت راست n_0 مقدار تابع $f(n)$ در رو یا پایین $g(n)$ است.

برای بیان اینکه تابع $f(n)$ عضوی از مجموعه $O(g(n))$ است، می‌نویسیم $f(n) = O(g(n))$. توجه کنید که $f(n) = \Theta(g(n))$ بطور ضمنی بر $f(n) = O(g(n))$ دلالت می‌کند، زیرا نماد Θ نسبت به نماد O قویتر است. اگر بصورت تئوری مجموعه‌ها بنویسیم داریم $\Theta(g(n)) \subseteq O(g(n))$. بنابراین این اثبات که هر تابع درجه دوم an^2+bn+c که $a>0$ مرتبه $\Theta(n^2)$ دارد نشان می‌دهد که هر تابع درجه دوم مرتبه $O(n^2)$ نیز دارد. آنچه شگفت‌آورتر است، این است که هر تابع خطی $an+b$ مرتبه $O(n^2)$ دارد، که به آسانی با گرفتن ثابتها بصورت $c=a+|b|$ و $n_0=1$ اثبات می‌شود.

بعضی از خوانندگان که نماد O را قبلاً دیده‌اند ممکن است این مطلب را عجیب بدانند که برای مثال بنویسیم $n=O(n^2)$. گاهی اوقات در متون علمی نماد O بطور غیر رسمی برای توصیف حدود قوی مجانبی استفاده می‌شود. بعبارت دیگر آنچه که در استفاده از نماد Θ تعریف کرده‌ایم. بهرحال، وقتی در این کتاب می‌نویسیم $f(n) = O(g(n))$ صرفاً ادعا کرده‌ایم که مضرب ثابتی از $g(n)$ یک حد بالای مجانبی روی $f(n)$ است، بدون هیچ ادعایی راجع به اینکه یک حد بالا چقدر قوی است. در حال حاضر جدا کردن حدود بالای مجانبی از حدود قوی مجانبی در متون علمی الگوریتم‌ها استاندارد شده است.

با استفاده از نماد O، اغلب می‌توانیم زمان اجرای یک الگوریتم را تنها با بـررسی سـاختار کـلی الگوریتم بیان کنیم. برای مثال، ساختار حلقه تو در توی دو طرفه الگوریتم مرتب سازی درجی از فصل ۲، بلافاصله یک حد بالای $O(n^2)$ روی زمان اجرا در بدترین حالت حاصل می‌کند: هزینه هر تکرار حلقه داخلی از بالا با $O(1)$ (ثابت) محدود می‌شود، اندیس‌های i و j هر دو حداکثر n و حلقه داخلی حداکثر یکبار هر n^2 جفت مقدار i و j اجرا می‌شود.

از آنجا که نماد O وقتیکه از آن استفاده می‌کنیم تا زمان اجرای یک الگوریتم را در بدترین حالت محدود کنیم، یک حد بالا را بیان می‌کند، حدی روی زمان اجرای یک الگوریتم روی هر ورودی داریم.

بنابراین حد $O(n^2)$ روی زمان اجرای مرتب سازی درجی در بدترین حالت برای زمان اجرای آن روی هر ورودی نیز بکار می‌رود. هر چند حد $\Theta(n^2)$ روی زمان اجرای مرتب سازی درجی در بدترین حالت، به‌طور ضمنی بر یک حد $\Theta(n^2)$ روی زمان اجرای مرتب‌سازی درجی روی هر ورودی دلالت نمی‌کند. برای مثال، در فصل ۲ دیدیم زمانیکه ورودی از قبل مرتب شده باشد، مرتب سازی درجی در زمان $\Theta(n)$ اجرا می‌شود.

بطور تکنیکی، گفتن این مطلب که زمان اجرای مرتب سازی درجی $O(n^2)$ است، یک استعمال نابجا است زیرا برای یک n مفروض، زمان اجرای واقعی بسته به ورودی خاص با اندازه n تغییر می‌کند. وقتی می‌گوییم "زمان اجرا $O(n^2)$ است"، منظور این است که یک تابع $f(n)$ وجود دارد که $O(n^2)$ است بطوریکه برای هر مقدار n صرف نظر از اینکه چه ورودی خاصّی با اندازه n انتخاب می‌شود، زمان اجرا روی این ورودی از بالا با مقدار $f(n)$ محدود می‌شود. بطور معادل، منظور این است که زمان اجرا در بدترین حالت $O(n^2)$ است.

نماد Ω

درست همانطور که نماد O یک حد بالای مجانبی روی یک تابع فراهم می‌کند، نماد Ω هم یک حد پایین مجانبی فراهم می‌کند. برای یک تابع مفروض $g(n)$ توسط $\Omega(g(n))$ (تلفظ می‌شود "اُمگای بزرگ gی n" یا گاهی اوقات "اُمگای gی n") به مجموعه توابع اشاره می‌کنیم

$$\Omega(g(n)) = \{f(n) : \text{ثابت های } c \text{ و } n_0 \text{ وجود دارندبطوریکه برای همه } n \geq n_0 \quad 0 \leq cg(n) \leq f(n)\}$$

شهودی که در پس نماد Ω است در شکل ۳-۱(c) نشان داده شده است. برای همه مقادیر n در سمت راست n_0 مقدار $f(n)$ در رو یا بالای $cg(n)$ است.

بنابراین از تعاریف نمادهای مجانبی مشاهده کرده‌ایم که اثبات قضیه مهم زیر آسان است (تمرین ۵ - ۳.۱ را ملاحظه نمایید).

قضیه ۳.۱

برای دو تابع $f(n)$ و $g(n)$ داریم $f(n)=\Theta(g(n))$ اگر و فقط اگر $f(n)=O(g(n))$ و $f(n)=\Omega(g(n))$ ■

بعنوان یک نمونه از کاربرد این قضیه، اثبات اینکه برای هر ثابت a، b و c که $a>0$، $an^2+bn+c=\Theta(n^2)$ مستقیماً دلالت می‌کند بر اینکه $an^2+bn+c=\Omega(n^2)$ و $an^2+bn+c=O(n^2)$ در عمل، بجای استفاده از قضیه ۳.۱ برای بدست آوردن حدهای پایین و بالای مجانبی از حدود قوی مجانبی، همانطور که در این نمونه انجام دادیم، معمولاً از آن به منظور اثبات حدهای قوی مجانبی از حدهای پایین و بالای مجانبی استفاده می‌کنیم.

از آنجا که نماد Ω، وقتی که از آن برای محدود کردن زمان اجرای یک الگوریتم در بهترین حالت استفاده می‌کنیم، یک حد پایین را بیان می‌کند، بطور ضمنی، زمان اجرای یک الگوریتم را روی ورودیهای دلخواه نیز محدود می‌کنیم. بعنوان مثال، زمان اجرای مرتب سازی درجی در بهترین حالت $\Omega(n)$ است، که بطور ضمنی بر این دلالت می‌کند که زمان اجرای مرتب سازی درجی $\Omega(n)$ است.

بنابراین زمان اجرای مرتب سازی درجی بین $\Omega(n)$ و $O(n^2)$ قرار می‌گیرد، زیرا در جایی بین یک تابع خطی بر حسب n و یک تابع درجه دوم از n قرار می‌گیرد. بعلاوه، این حدود به طور مجانبی تا حد ممکن قوی هستند: بعنوان نمونه، زمان اجرای مرتب سازی درجی $\Omega(n^2)$ نیست، زیرا برای مرتب‌سازی درجی یک ورودی وجود دارد که در زمان $\Theta(n)$ اجرا می‌شود (یعنی وقتی که ورودی از قبل مرتب شده است). امّا گفتن اینکه زمان اجرای مرتب سازی درجی در بدترین حالت $\Omega(n^2)$ است متناقض با جمله قبل نیست، زیرا یک ورودی وجود دارد که باعث می‌شود الگوریتم زمان $\Omega(n^2)$ را صرف کند. وقتی می‌گوییم زمان اجرای یک الگوریتم $\Omega(g(n))$ است، منظورمان این است که صرف نظر از اینکه چه ورودی خاصی باندازه n برای هر مقدار n انتخاب شود، زمان اجرا روی این ورودی برای n به اندازه کافی بزرگ حداقل برابر یک ثابت، ضرب در $g(n)$ است.

نمادگذاری مجانبی در تساویها و نامساویها

قبلاً دیدیم که چگونه نماد مجانبی می‌تواند در فرمولهای ریاضی استفاده شود. برای مثال، در معرفی نماد O، نوشتیم "$n=O(n^2)$". همچنین ممکن است بنویسیم $2n^2+3n+1=2n^2+\Theta(n)$. چگونه چنین فرمولهایی را تفسیر کنیم؟

وقتی که نماد مجانبی تنها در سمت راست یک تساوی (یا نامساوی) قرار می‌گیرد، مانند $n=O(n^2)$ علامت مساوی را برای مفهوم عضویت مجموعه تعریف کرده‌ایم: $n\in O(n^2)$. گرچه بطور کل، وقتی نماد مجانبی در یک فرمول ظاهر می‌شود، آن را بصورتی تفسیر می‌کنیم که بیانگر یک تابع ناشناخته باشد که به نام آن توجهی نمی‌کنیم. بعنوان مثال، فرمول $2n^2+3n+1=2n^2+\Theta(n)$ به این معنی است که $2n^2+3n+1=2n^2+f(n)$، که $f(n)$ تابعی در مجموعه $\Theta(n)$ است. در این حالت، $f(n)=3n+1$ که در حقیقت در $\Theta(n)$ قرار دارد.

استفاده از نماد مجانبی بدین گونه می‌تواند به حذف جزئیات غیر ضروری در یک معادله کمک کند. بعنوان مثال، در فصل ۲، زمان اجرای مرتب سازی ادغام در بدترین حالت را بصورت رابطه بازگشتی $T(n)=2T(n/2)+\Theta(n)$ بیان کردیم. اگر فقط به رفتار مجانبی $T(n)$ توجه کنیم، هیچ نکته‌ای در تعیین همه جملات مرتبه پایینتر بطور دقیق وجود ندارد؛ چنین نتیجه می‌شود که تمام آنها باید در تابع ناشناخته‌ای که با جمله $\Theta(n)$ به آن اشاره می‌شود قرار بگیرند.

در نتیجه تعداد توابع ناشناخته در یک عبارت باید مساوی تعداد دفعات ظاهر شدن نماد مجانبی

باشد. بعنوان مثال در عبارت

$$\sum_{i=1}^{n} O(i) ,$$

فقط یک تابع ناشناخته (یک تابع از i) وجود دارد. بنابراین این عبارت مشابه $O(1)+O(2)+...+O(n)$ نیست و واقعاً یک تفسیر مشخص ندارد.

در بعضی حالتها، نماد مجانبی در سمت چپ یک تساوی نمایان می‌شود مانند

$$2n^2+\Theta(n)=\Theta(n^2)$$

این تساویها را با استفاده از قانون زیر تفسیر می‌کنیم: بدون توجه به اینکه توابع ناشناخته چگونه در سمت چپ علامت تساوی انتخاب می‌شوند، روشی برای انتخاب توابع ناشناخته در سمت راست علامت تساوی جهت معتبر ساختن تساوی وجود دارد. بنابراین، مفهوم مثال ما این است که برای هر تابع $f(n)\in\Theta(n)$ تابع $g(n)\in\Theta(n^2)$ وجود دارد بطوریکه برای همه nها، $2n^2+f(n)=g(n)$. بعبارت دیگر، سمت راست یک تساوی نسبت به سمت چپ آن دارای جزئیات کمتری است.

تعدادی از این روابط می‌توانند بصورت زنجیره وار به هم مرتبط شوند، مانند

$$\begin{aligned} 2n^2+3n+1 &= 2n^2+\Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

می‌توانیم هر تساوی را بطور مجزا با استفاده از قانون فوق تفسیر کنیم. تساوی اول می‌گوید که تابع $f(n)\in\Theta(n)$ وجود دارد بطوریکه به ازای تمام nها، $2n^2+3n+1=2n^2+f(n)$. تساوی دوم می‌گوید که برای هر تابع $g(n)\in\Theta(n)$ (مانند $f(n)$ که ذکر شد)، تابع $h(n)\in\Theta(n^2)$ وجود دارد بطوریکه به ازای تمام nها، $2n^2+g(n)=h(n)$. توجه کنید که این تفسیر دلالت می‌کند بر اینکه $2n^2+3n+1=\Theta(n^2)$، که همان چیزی است که ارتباط زنجیره وار تساویها بطور شهودی به ما می‌دهد.

نماد o

حد بالای مجانبی که با استفاده از نماد O بدست آمده است ممکن است بطور مجانبی قوی باشد یا نباشد. حد $2n^2=O(n^2)$ بطور مجانبی قوی است، اما حد $2n=O(n^2)$ قوی نیست. با استفاده از نماد o به یک حد بالا اشاره می‌کنیم که بطور مجانبی قوی نیست. بطور رسمی $o(g(n))$ ("اُی کوچک جی n") را بصورت مجموعه زیر تعریف می‌کنیم

$o(g(n))=\{f(n):$ برای هر ثابت مثبت $c>0$ یک ثابت $n_0>0$ وجود دارد بطوریکه به ازای تمام $n\geq n_0$

$0\leq f(n)<cg(n)\}$

بعنوان مثال، $2n=o(n^2)$، اما $2n^2\neq o(n^2)$

تعاریف نماد O و نماد o مشابه هستند. تفاوت اصلی آنها این است که در $f(n)=O(g(n))$، حد $0\le f(n)\le cg(n)$ برای ثابت $c>0$ بر قرار است، اما در $f(n)=o(g(n))$، حد $0\le f(n)<cg(n)$ برای هر ثابت $c>0$ بر قرار است. بطور شهودی در نماد o، تابع $f(n)$ نسبت به $g(n)$ در صورتیکه n به بی نهایت میل کند بی اهمیت می‌شود؛ بعبارت دیگر

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 . \qquad (۳.۱)$$

بعضی از نویسنده‌ها این حد را بعنوان تعریفی از نماد o استفاده می‌کنند؛ تعریف موجود در این کتاب نیز توابع گمنام را محدود می‌کند تا بطور مجانبی غیر منفی باشد.

نماد ω

در مقایسه، نماد ω در برابر نماد Ω مانند تماد o در برابر نماد O است. از نماد ω برای نشان دادن یک حد پایین که بطور مجانبی قوی نیست استفاده می‌کنیم. یک روش برای تعریف آن به شکل زیر است

$f(n)\in\omega(g(n))$، اگر و فقط اگر $g(n)\in o(f(n))$

اما بطور رسمی، $\omega(g(n))$ ("اُمگای کوچک g ی n") را بصورت مجموعه زیر تعریف می‌کنیم

$\omega(g(n)) = \{f(n)$: برای‌هرثابت‌مثبت $c>0$، یک‌ثابت $n_0>0$ وجوددارد به طوری که برای تمام $n \ge n_0$
$0 \le cg(n) \le f(n)\}$

بعنوان مثال $n^2/2=\omega(n)$ امّا $n^2/2\ne\omega(n^2)$ رابطه $f(n)=\omega(g(n))$ دلالت می‌کند بر اینکه

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty ,$$

اگر حد موجود باشد. بعبارت دیگر، $f(n)$ نسبت به $g(n)$ در صورتیکه n به بی نهایت میل کند بطور دلخواه بزرگ می‌شود.

مقایسه توابع

اکثر ویژگی‌های رابطه‌ای اعداد حقیقی در مقایسه‌های مجانبی نیز بکار می‌روند. برای اثبات، فرض کنید که $f(n)$ و $g(n)$ بطور مجانبی مثبت هستند.

تعدی:

$f(n) = \Theta(g(n))$ و $g(n) = \Theta(h(n))$ دلالت می‌کنند بر اینکه $f(n) = \Theta(h(n))$

$f(n) = O(g(n))$ و $g(n) = O(h(n))$ دلالت می‌کنند بر اینکه $f(n) = O(h(n))$
$f(n) = \Omega(g(n))$ و $g(n) = \Omega(h(n))$ دلالت می‌کنند بر اینکه $f(n) = \Omega(h(n))$
$f(n) = o(g(n))$ و $g(n) = o(h(n))$ دلالت می‌کنند بر اینکه $f(n) = o(h(n))$
$f(n) = \omega(g(n))$ و $g(n) = \omega(h(n))$ دلالت می‌کنند بر اینکه $f(n) = \omega(h(n))$

انعکاسی:

$$f(n) = \Theta(f(n)),$$
$$f(n) = O(f(n)),$$
$$f(n) = \Omega(f(n)).$$

تقارن:

$f(n) = \Theta(g(n))$ اگر و فقط اگر $g(n) = \Theta(f(n))$.

ترانهادهٔ تقارن:

$f(n) = O(g(n))$ اگر و فقط اگر $g(n) = \Omega(f(n))$
$f(n) = o(g(n))$ اگر و فقط اگر $g(n) = \omega(f(n))$

چون این ویژگی‌ها برای نمادهای مجانبی بر قرار هستند، می‌توان یک تشبیه بین مقایسه مجانبی دو تابع f و g و مقایسه دو عدد حقیقی a و b ایجاد نمود:

$$f(n) = O(g(n)) \quad \approx \quad a \le b,$$
$$f(n) = \Omega(g(n)) \quad \approx \quad a \ge b,$$
$$f(n) = \Theta(g(n)) \quad \approx \quad a = b,$$
$$f(n) = o(g(n)) \quad \approx \quad a < b,$$
$$f(n) = \omega(g(n)) \quad \approx \quad a > b.$$

می‌گوییم $f(n)$ بطور مجانبی کوچکتر از $g(n)$ است اگر $f(n)=o(g(n))$ و $f(n)$ بطور مجانبی بزرگتر از $g(n)$ است اگر $f(n) = \omega(g(n))$.

گرچه یک ویژگی اعداد حقیقی، برای نماد مجانبی برقرار نیست.

سه قسمتی[1]: برای هر دو عدد حقیقی a و b دقیقاً یکی از روابط زیر باید برقرار باشد:

$a<b, a=b,$ یا $a>b$

---

1. Trichotomy

گرچه هر دو عدد حقیقی می‌توانند مقایسه شوند امّا همه توابع بطور مجانبی قابل مقایسه نیستند. بعبارت دیگر برای دو تابع $f(n)$ و $g(n)$ ممکن است حالتی وجود داشته باشد که نه $f(n)=O(g(n))$ و نه $f(n)=\Omega(g(n))$ بر قرار باشد. بعنوان مثال، توابع n و $n^{1+\sin n}$ نمی‌توانند با استفاده از نماد مجانبی مقایسه شوند زیرا مقدار توان در $n^{1+\sin n}$ بین $2,0$ با در نظر گرفتن تمام مقادیر در این میان تـغییر می‌کند.

تمرین‌ها

۳.۱-۱ فرض کنید $f(n)$ و $g(n)$ بطور مجانبی توابع غیر منفی باشند. با استفاده از تعریف اصلی نماد Θ، ثابت کنید که $max(f(n),g(n))=\Theta(f(n)+g(n))$.

۳.۱-۲ نشان دهید هر ثابت حقیقی a و b که $b>0$

$$(n+a)^b=\Theta(n^b). \quad (۳.۲)$$

۳.۱-۳ توضیح دهید چرا عبارت "زمان اجرای الگوریتم A حداقل $O(n^2)$ است"، بی معنی است؟

۳.۱-۴ آیا $2^{n+1} = O(n^2)$؟ آیا $2^{2n}=O(2^{2n})$؟

۳.۱-۵ قضیه ۳.۱ را اثبات کنید.

۳.۱-۶ ثابت کنید که زمان اجرای یک الگوریتم $\Theta(g(n))$ است اگر و فقط اگر زمان اجـرای آن در بدترین حالت $O(g(n))$ و زمان اجرای آن در بهترین حالت $\Omega(g(n))$ باشد.

۳.۱-۷ ثابت کنید که $o(g(n)) \cap w(g(n))$ یک مجموعه تهی است.

۳.۱-۸ می‌توانیم نمادگذاری خود را برای حالتی که دو پارامتر n و m می‌توانند به طور مستقل با سرعت‌های متفاوتی به بی‌نهایت میل کنند بسط دهیم. برای یک تابع $g(n,m)$ مفروض، مجموعه توابع را با $O(g(n,m))$ نشان می‌دهیم.

$O(g(n,m))=\{f(n,m):$ ثابت‌های مثبت c n_0 m_0 وجود دارند، بطوریکه برای تمام $m\geq n_0$ و $m\geq m_0$ $0\leq f(n,m)\leq cg(n,m)\}$

تعاریف متناظری برای $\Omega(g(n,m))$ و $\Theta(g(n,m))$ ارائه دهید.

۳.۲ نمادهای استاندارد و توابع عمومی

این بخش بعضی از توابع ریاضی استاندارد و نمادها را بیان کرده و رابطه بین آنها را بررسی می‌کند. همچنین استفاده از نمادهای مجانبی را توضیح می‌دهد.

یکنواختی[1]

تابع $f(n)$ صعودی یکنواخت است، اگر $m \leq n$ دلالت کند بر اینکه $f(m) \leq f(n)$. بطور مشابه، این تابع نزولی یکنواخت است اگر $m \leq n$ دلالت کند بر اینکه $f(m) \geq f(n)$ تابع $f(n)$ اکیداً صعودی است اگر $m < n$ دلالت کند بر اینکه $f(m) < f(n)$ و اکیداً نزولی است اگر $m < n$ دلالت کند بر اینکه $f(m) > f(n)$.

کف و سقف

برای هر عدد حقیقی x بزرگترین عدد صحیح کوچکتر یا مساوی x را با $\lfloor x \rfloor$ (بخوانید "کفِ x") و کوچکترین عدد صحیح بزرگتر یا مساوی x را با $\lceil x \rceil$ (بخوانید "سقف x") نشان می‌دهیم. برای همه xهای حقیقی،

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1 . \qquad (۳.۳)$$

به ازای هر عدد صحیح n

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$$

و برای هر عدد حقیقی $n \geq 0$ و اعداد صحیح $a,b > 0$

$$\lceil \lceil n/a \rceil /b \rceil = \lceil n/ab \rceil , \qquad (۳.۴)$$

$$\lfloor \lfloor n/a \rfloor /b \rfloor = \lfloor n/ab \rfloor , \qquad (۳.۵)$$

$$\lceil a/b \rceil \leq (a + (b - 1))/b , \qquad (۳.۶)$$

$$\lfloor a/b \rfloor \geq ((a - (b - 1))/b . \qquad (۳.۷)$$

تابع کف $f(x) = \lfloor x \rfloor$ همانند تابع سقف $f(x) = \lceil x \rceil$ صعودی یکنواخت است.

حساب پیمانه‌ای[2]

برای هر عدد صحیح a و هر عدد صحیح مثبت n مقدار a به پیمانه n ($a \bmod n$)، باقیمانده (یا مانده[3]) خارج قسمت a/n است:

$$a \bmod n = a - \lfloor a/n \rfloor \, n . \qquad (۳.۸)$$

با ارائه یک برداشت خوش تعریف از باقیمانده یک عدد صحیح وقتی که به عدد دیگر تقسیم می‌شود، تهیّه نماد خاصی برای نمایش تساوی باقیمانده‌ها راحت است. اگر $(a \bmod n) = (b \bmod n)$ می‌نویسیم $a \equiv b$ (به پیمانه n) و می‌گوییم a هم‌ارز[4] b است به پیمانه n بعبارت دیگر، $a \equiv b$ (به پیمانه n) اگر a و b پس از تقسیم بر n باقیمانده یکسانی داشته باشند. بطور معادل، $a \equiv b$ (به پیمانه n) اگروفقط‌اگر n یک‌مقسوم‌علیه $b-a$ باشد.اگر a هم‌ارز b به پیمانه n نباشد، می‌نویسیم $a \not\equiv b$ (به‌پیمانه n).

---

1. Monotonicity
2. Modular arithmetic
3. Residue
4. equivalent

چند جمله‌ای‌ها

اگر d یک عدد صحیح غیر منفی باشد، یک چند جمله‌ای بر حسب n از درجه d یک تابع $p(n)$ به شکل

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

است، که ثابتهای $a_0, a_1, \ldots, a_d$ ضرایب چند جمله‌ای هستند و $a_d \neq 0$ یک چند جمله‌ای بطور مجانبی مثبت است اگر و فقط اگر $a_d > 0$ برای یک چند جمله‌ای بطور مجانبی مثبت $p(n)$ از درجه d داریم $p(n) = \Theta(n^d)$. برای هر ثابت حقیقی $a \geq 0$ تابع n^a صعودی یکنواخت است، و برای هر ثابت حقیقی $a \leq 0$ تابع n^a نزولی یکنواخت است. می‌گوییم تابع $f(n)$ بصورت چند جمله‌ای محدود است اگر به ازای ثابت k $f(n) = O(n^k)$

نمایی‌ها

برای همه اعداد حقیقی $n,m,a > 0$ تساویهای زیر را داریم:

$$\begin{aligned}
a^0 &= 1 , \\
a^1 &= a , \\
a^{-1} &= 1/a , \\
(a^m)^n &= a^{mn} , \\
(a^m)^n &= (a^n)^m , \\
a^m a^n &= a^{m+n} .
\end{aligned}$$

به ازای تمام nها و $a \geq 1$ تابع a^n در n صعودی یکنواخت است. به منظور سهولت، فرض خواهیم کرد که $0^0 = 1$.

سرعت رشد چند جمله‌ایها و نمایی‌ها می‌تواند به واقعیت زیر ربط داشته باشد. به ازای تمام ثابت‌های حقیقی a و b که $a > 1$

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 , \tag{۳.۹}$$

که از آن می‌توانیم نتیجه بگیریم که

$$n^b = o(a^n) .$$

بنابراین، هر تابع نمایی با پایه اکیداً بزرگتر از 1 سریعتر از هر تابع چند جمله‌ای رشد می‌کند.

با استفاده از e برای نشان دادن $2.71828...$ که مبنای تابع لگاریتم طبیعی است، به ازای همه xهای حقیقی داریم

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} , \tag{۳.۱۰}$$

که "!" تابع فاکتوریل را نشان می‌دهد که بعداً در این بخش تعریف می‌شود. برای همه اعداد حقیقی x نامساوی زیر را داریم

$$e^x \geq 1 + x \;, \tag{۳.۱۱}$$

که تساوی فقط زمانیکه $x=0$ است برقرار می‌باشد. وقتی که $|x| \leq 1$، تقریب زیر را داریم:

$$1 + x \leq e^x \leq 1 + x + x^2 \;. \tag{۳.۱۲}$$

هنگامیکه $x \to 0$، تقریب e^x با $1+x$ کاملاً مناسب است:

$$e^x = 1 + x + \Theta(x^2) \;.$$

(در این معادله، از نماد مجانبی برای توصیف رفتار حدّی به شکل $x \to 0$ به جای $x \to \infty$ استفاده می‌شود). برای همه xها داریم

$$\lim_{n\to\infty}\left(1 + \frac{x}{n}\right)^n = e^x \;. \tag{۳.۱۳}$$

لگاریتم‌ها

از نمادهای زیر استفاده خواهیم کرد:

$$\begin{aligned} \lg n &= \log_2 n && \text{(لگاریتم دودویی)} \\ \ln n &= \log_e n && \text{(لگاریتم طبیعی)} \\ \lg^k n &= (\lg n)^k && \text{(توان رسانی)} \\ \lg\lg n &= \lg(\lg n) && \text{(ترکیب)} \end{aligned}$$

یک قرارداد مهم نمادگذاری که قبول خواهیم کرد این است که توابع لگاریتمی فقط برای جمله بعدی فرمول بکار می‌روند، بنابراین $\lg n + k$ به معنی $(\lg n)+k$ خواهد بود و نه $\lg(n+k)$ اگر ثابت $b>1$ بر قرار باشد آنگاه برای $n>0$ تابع $\log_b n$ اکیداً صعودی است.

برای همه اعداد حقیقی $a>0$ $b>0$ $c>0$ و n،

$$\begin{aligned} a &= b^{\log_b a} \;, \\ \log_c(ab) &= \log_c a + \log_c b \;, \\ \log_b a^n &= n \log_b a \;, \\ \log_b a &= \frac{\log_c a}{\log_c b} \;, \end{aligned} \tag{۳.۱۴}$$

$$\begin{aligned} \log_b(1/a) &= -\log_b a \;, \\ \log_b a &= \frac{1}{\log_a b} \;, \\ a^{\log_b c} &= c^{\log_b a} \;, \end{aligned} \tag{۳.۱۵}$$

که در هر کدام از تساویهای فوق، مبنای لگاریتمها 1 نیست.

بنا به تساوی (۳.۱۴)، تغییر مبنای یک لگاریتم از یک ثابت به ثابتی دیگر فقط مقدار لگاریتم را با یک ضریب ثابت تغییر می‌دهد و بنابراین اغلب از نماد "lgn" وقتیکه به ضرایب ثابت توجه نمی‌کنیم استفاده خواهیم کرد، مانند نماد O. دانشمندان کامپیوتر دریافته‌اند که 2، طبیعی‌ترین مبنای لگاریتمها است زیرا بسیاری از الگوریتمها و ساختمان داده‌ها شامل شکستن یک مسئله به دو قسمت هستند.

یک بسط سری ساده برای $ln(1+x)$ وقتی که $|x| < 1$ وجود دارد:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots .$$

همچنین، برای $x > -1$ نامساویهای زیر را داریم:

$$\frac{x}{1+x} \leq \ln(1+x) \leq x , \qquad (۳.۱۶)$$

که تساوی فقط برای $x=0$ بر قرار است.

می‌گوییم تابع $f(n)$ **بطور چند لگاریتمی محدود** است اگر برای ثابت k، $f(n)=O(lg^k n)$. می‌توانیم رشد چند جمله‌ایها و چند لگاریتمها را با استفاده از جانشینی lgn برای n و 2^a برای a در تساوی (۳.۹) بیان کنیم، که تساوی زیر بدست می‌آید:

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0 .$$

از این حد، می‌توانیم نتیجه بگیریم که برای هر ثابت $a>0$،

$$\lg^b n = o(n^a)$$

بنابراین هر تابع چند جمله‌ای مثبت سریعتر از هر تابع چند لگاریتمی رشد می‌کند.

فاکتوریل

نماد $n!$ (بخوانید "n فاکتوریل") برای اعداد صحیح $n \geq 0$ بصورت زیر تعریف می‌شود:

$$n! = \begin{cases} 1 & \text{if } n = 0 , \\ n \cdot (n-1)! & \text{if } n > 0 . \end{cases}$$

بنابراین، $n! = 1 .2 .3. . n$.

یک حد بالای ضعیف روی تابع فاکتوریل، $n! \leq n^n$ است، زیرا هر یک از n جمله در حاصلضرب فاکتوریل حداکثر n است. تقریب *stirling*،

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1+\Theta\left(\frac{1}{n}\right)\right), \tag{۳.۱۷}$$

که e مبنای لگاریتم طبیعی است، یک حد بالای قویتر و نیز یک حد پایین می‌دهد. می‌توان ثابت کرد (تمرین ۳-۲.۳ را ملاحظه نمایید)

$$\begin{aligned} n! &= o(n^n), \\ n! &= \omega(2^n), \\ \lg(n!) &= \Theta(n \lg n), \end{aligned} \tag{۳.۱۸}$$

که تقریب *stirling* در اثبات تساوی (۳.۱۸) مفید است. معادله زیر نیز به ازای تمام $n \geq 1$ برقرار است:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \tag{۳.۱۹}$$

که در آن

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}. \tag{۳.۲۰}$$

تکرار تابعی

از نماد $f^{(i)}(n)$ برای نشان دادن تابع $f(n)$ که بطور تکراری i مرتبه برای یک مقدار اولیه n بکار می‌رود استفاده می‌کنیم. بطور رسمی، فرض کنید $f(n)$ تابعی روی اعداد حقیقی باشد. برای اعداد صحیح غیر منفی i بطور بازگشتی تعریف می‌کنیم

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0, \\ f(f^{(i-1)}(n)) & \text{if } i > 0. \end{cases}$$

بعنوان مثال اگر $f(n)=2n$ آنگاه $f^{(i)}(n) = 2^i n$.

تابع لگاریتم تکراری

از نماد $lg^* n$ (بخوانید "لُوگْ اِستار n") برای نشان دادن لگاریتم تکراری استفاده می‌کنیم که بصورت زیر تعریف می‌شود. اجازه دهید که $lg^{(i)}n$ بصورتی که در بالا تعریف شد، باشد، با $f(n)=lgn$

چون لگاریتم یک عدد غیر مثبت تعریف نشده است، $lg^{(i)}n$ فقط اگر $lg^{(i-1)}n>0$ تعریف می‌شود. حتماً بین $lg^{(i)}n$ (تابع لگاریتم که i مرتبه با شروع از آرگومان n بطور متوالی بکار می‌رود) و $lg^i n$

(لگاریتم n که به توان i رسیده است) تمایز قائل شوید. تابع لگاریتم تکراری بصورت زیر تعریف می‌شود.

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\} .$$

لگاریتم تکراری تابعی است که بسیار کند رشد می‌کند:

$$\begin{aligned} \lg^* 2 &= 1 , \\ \lg^* 4 &= 2 , \\ \lg^* 16 &= 3 , \\ \lg^* 65536 &= 4 , \\ \lg^*(2^{65536}) &= 5 . \end{aligned}$$

از آنجا که تعداد اتمهای جهان قابل مشاهده حدود 10^{80} تخمین زده می‌شود، که بسیار کمتر از 2^{65536} است، بندرت با یک اندازه ورودی n بطوریکه $\lg^* n > 5$ مواجه شویم.

اعداد فیبوناچی

اعداد فیبوناچی با رابطه بازگشتی زیر تعریف می‌شوند:

$$\begin{aligned} F_0 &= 0 , \\ F_1 &= 1 , \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2 . \end{aligned} \tag{۳.۲۱}$$

بنابراین، هر عدد فیبوناچی، جمع دو عدد قبلی است، که توالی زیر حاصل می‌شود:

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots .$

اعداد فیبوناچی با نسبت طلایی[1] ϕ و مزدوجش $\widehat{\phi}$، که بوسیله فرمولهای زیر ارائه می‌شوند رابطه دارند:

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \\ &= 1.61803\ldots , \\ \widehat{\phi} &= \frac{1-\sqrt{5}}{2} \\ &= -.61803\ldots . \end{aligned} \tag{۳.۲۲}$$

بویژه داریم

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} , \tag{۳.۲۳}$$

1. golden ratio

که می‌تواند بوسیله استقرا اثبات شود (تمرین ۶-۲.۲). از آنجا که $|\hat{\phi}| < 1$، داریم $|\hat{\phi}^i|/\sqrt{5} < 1/\sqrt{5} < 1/2,$ بنابراین عدد iام فیبوناچی F_i برابر $\phi^i/\sqrt{5}$ که نزدیکترین عدد صحیح گرد شده است، می‌باشد. بنابراین، اعداد فیبوناچی بصورت نمایی رشد می‌کنند.

تمرین‌ها

۱-۳.۲ نشان دهید اگر $f(n)$ و $g(n)$ توابع صعودی یکنواخت باشند، آنگاه توابع $f(n)+g(n)$ و $f(g(n))$ نیز صعودی یکنواخت هستند، و اگر علاوه بر آن $f(n)$ و $g(n)$ غیر منفی نیز باشند، آنگاه $f(n).g(n)$ صعودی یکنواخت است.

۲-۳.۲ تساوی (۳.۱۵) را ثابت کنید.

۳-۳.۲ تساوی (۳.۱۸) را ثابت کنید. همچنین ثابت کنید $n! = \omega(2^n)$ و $n! = o(n^n)$

*۴-۳.۲ آیا تابع $\lceil lg\, n \rceil !$ بطور چند جمله‌ای محدود است؟ آیا تابع $\lceil lg\, lgn \rceil !$ بطور چند جمله‌ای محدود می‌شود؟

*۵-۳.۲ کدام یک بطور مجانبی بزرگتر است:

$lg(lg^* n)$ یا $lg^*(lgn)$

۶-۳.۲ با استفاده از استقرا ثابت کنید که عدد iام فیبوناچی در تساوی زیر صدق می‌کند

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} ,$$

که در آن ϕ نسبت طلایی و $\hat{\phi}$ مزدوجش است.

۷-۳.۲ ثابت کنید برای $i \geq 0$، $(i + 2)$اُمین عدد فیبوناچی در $F_{i+2} \geq \phi^i$ صدق می‌کند.

مسائل

۳-۱ رفتار مجانبی چند جمله‌ای‌ها

فرض کنید

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

که $a_d > 0$ یک چند جمله‌ای درجه d بر حسب n باشد، و فرض کنید k یک ثابت باشد. از تعاریف نمادهای مجانبی برای اثبات ویژگیهای زیر استفاده کنید.

a. اگر $k \geq d$ آنگاه $p(n) = O(n^k)$

b. اگر $k \leq d$ آنگاه $p(n) = \Omega(n^k)$

.c اگر $k = d$ آنگاه $p(n) = \Theta(n^k)$

.d اگر $k > d$ آنگاه $p(n) = o(n^k)$

.e اگر $k < d$ آنگاه $p(n) = \omega(n^k)$

۳-۲ رشد مجانبی نسبی

برای هر جفت از عبارات $(A\ ,\ B)$ در جدول زیر، نشان دهید که A برابر O، o، Ω، یا Θی Bاست. فرض کنید که $k\geq 1$ $\varepsilon>0$ و $c>1$ ثابت هستند. پاسخ شما باید بصورت "بله" یا "خیر" در هر خانه جدول نوشته شود.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

۳-۳ رتبه‌بندی بر حسب سرعت‌های رشد مجانبی

a. توابع زیر را بر حسب مرتبه رشد، رتبه بندی کنید؛ به عبارت دیگر، یک آرایش $g_1, g_2, \ldots, g_{30}$ از توابعی که در $g_1=\Omega(g_2)$ $g_2=\Omega(g_3)$ ، . . . $g_{29}=\Omega(g_{30})$ صدق می‌کنند پیدا کنید. لیست خود را به کلاس‌های هم‌ارزی افراز کنید بطوریکه $f(n)$ و $g(n)$ در کلاس یکسانی باشند اگر و فقط اگر $f(n)=\Theta(g(n))$

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln\ln n$	$\lg^* n$	$n\cdot 2^n$	$n^{\lg\lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n\lg n$	$2^{2^{n+1}}$

b. نمونه‌ای از یک تابع غیر منفی $f(n)$ ارائه دهید که برای همه توابع $g_i(n)$ در قسمت (a)، $f(n)$ نه $O(g_i(n))$ و نه $\Omega(g_i(n))$ باشد.

۴-۳ ویژگی‌های نماد مجانبی

فرض کنید $f(n)$ و $g(n)$ بطور مجانبی توابعی مثبت باشند. هر یک از حدس‌های زیر را ثابت کرده و یا رد کنید.

a. $f(n) = O(g(n))$ بر این دلالت می‌کند که $g(n) = O(f(n))$.

b. $f(n)+g(n) = \Theta\ (min\ \ (f(n)\ ,\ g(n)))$.

c. $f(n) = O(g(n))$ بر این دلالت می‌کند که $lg(f(n)) = O(lg(g(n)))$ که برای همه nهای به اندازه کافی بزرگ، $lg(g(n))\geq 1$ و $f(n)\geq 1$.

d. $f(n) = O(g(n))$ بر این دلالت می‌کند که $2^{f(n)} = O(2^{g(n)})$.

e. $f(n) = O((f(n))^2)$.

f. $f(n) = O(g(n))$ بر این دلالت می‌کند که $g(n) = \Omega(f(n))$.

g. $f(n) = \Theta\ (f(n/2))$.

h. $f(n)+o(f((n)) = \Theta(f(n))$.

۵-۳ انواع متفاوت O و Ω

برخی از نویسندگان، Ω را اندکی متفاوت با آنچه تعریف کردیم تعریف می‌کنند؛ از $\overset{\infty}{\Omega}$ (بخواهید "اُمگا بی نهایت") برای این تعریف متفاوت استفاده می‌کنیم. می‌گوییم $f(n)=\overset{\infty}{\Omega}\ (g(n))$ اگر ثابت مثبت c وجود داشته باشد بطوریکه برای بسیاری از اعداد صحیح بی‌نهایت n $f(n)\geq cg(n)\geq 0$.

a. نشان دهید برای هر دو تابع $f(n)$ و $g(n)$ که بطور مجانبی غیر منفی هستند، یا $f(n) = O(g(n))$ یا $f(n) = \overset{\infty}{\Omega}\ (g(n))$ یا هر دو تساوی برقرار است، در حالیکه اگر از $\overset{\infty}{\Omega}$ بجای Ω استفاده کنیم این مطلب درست نیست.

b. مزایا و مضرّات بالقوّه استفاده از $\overset{\infty}{\Omega}$ بجای Ω را برای توصیف زمان اجرای برنامه‌ها بیان کنید.

بعضی از نویسندگان نیز نماد O را اندکی متفاوت تعریف می‌کنند؛ اجازه دهید تا از نماد O' برای این تعریف استفاده کنیم. می‌گوییم $f(n) = O'(g(n))$ اگر و فقط اگر $|f(n)\ | = O(g(n))$.

c. اگر O' را در قضیه ۳.۱ جایگزین O کرده اما باز هم از Ω استفاده نماییم، هر سمت "اگر و فقط اگر" چه تغییری می‌کند؟

بعضی از نویسندگان (بخوانید "اُوی ساده") را تعریف می‌کنند تا مفهوم O را بنحوی که ضرایب لگاریتمی نادیده گرفته شده‌اند، بیان کند:

$\tilde{O}(g(n)) = \{f(n)$: ثابت‌های مثبت k،c و n_0 وجود دارند بطوریکه برای تمام $n\geq n_0$
$0\leq f(n)\leq cg(n)lg^k(n)\}$

d. $\tilde{\Omega}$ و $\tilde{\Theta}$ را به روشی مشابه تعریف کنید. قضیه ۳.۱ را با این نمادها تعریف کرده و آن را ثابت کنید.

۳-۶ توابع تکراری

عملگر تکرار * که در تابع lg^* استفاده شده است می‌تواند برای هر تابع صعودی یکنواخت $f(n)$ روی اعداد حقیقی بکار رود. برای یک ثابت $c \in R$ مفروض، تابع تکراری f_c^* را بصورت زیر تعریف می‌کنیم

$$f_c^*(n) = min \{i \geq 0: f^{(i)}(n) \leq c\}$$

که لازم نیست در همه حالتها خوش تعریف باشد. بعبارت دیگر، کمیت $f_c^*(n)$ تعداد استعمال‌های تکراری تابع f است که برای کاهش آرگومانش به c یا کمتر از آن نیاز است.

برای هر یک از توابع $f(n)$ زیر و ثابت‌های c، یک حد روی $f_c^*(n)$ ارائه دهید که تا حدّ ممکن قوی باشد.

	$f(n)$	c	$f_c^*(n)$
a.	$n-1$	0	
b.	$\lg n$	1	
c.	$n/2$	1	
d.	$n/2$	2	
e.	$\sqrt{n}$	2	
f.	$\sqrt{n}$	1	
g.	$n^{1/3}$	2	
h.	$n/\lg n$	2	

۴ رابطه‌های بازگشتی

همانطور که در بخش ۲.۳.۲ ذکر شد، وقتی یک الگوریتم شامل یک فراخوانی بازگشتی به خودش است، زمان اجرایش اغلب می‌تواند بوسیله یک رابطه بازگشتی بیان شود. یک رابطه بازگشتی یک تساوی یا نامساوی است که یک تابع را بر حسب مقدار آن روی ورودیهای کوچکتر توصیف می‌کند. بعنوان مثال، در بخش ۲.۳.۲ مشاهده کردیم که زمان اجرای *T(n)* روال *MERGE-SORT* در بدترین حالت می‌توانست بوسیله رابطه بازگشتی زیر تعریف شود:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,, \end{cases} \qquad (۴.۱)$$

که ادعا شد جوابش $T(n) = \Theta(nlgn)$ است.

این فصل سه روش برای حل رابطه‌های بازگشتی ارائه می‌کند - بعبارت دیگر برای بدست آوردن حدهای "O" یا "Θ" مجانبی روی جواب. در **روش جایگذاری**، یک حد حدس می‌زنیم و سپس از استقرای ریاضی برای اثبات درستی حدس خود استفاده می‌کنیم. **روش درخت بازگشت**، رابطه بازگشتی را به یک درخت تبدیل می‌کند که گره‌هایش بیانگر هزینه‌های ایجاد شده در سطوح مختلف بازگشت می‌باشند؛ از تکنیکهایی برای محدود کردن حاصلجمع‌ها برای حل رابطه بازگشتی استفاده می‌کنیم. **روش اصلی** حدهایی برای رابطه‌های بازگشتی به شکل زیر فراهم می‌کند:

$$T(n) = aT(n/b) + f(n),$$

که $a \geq 1$، $b > 1$ و $f(n)$ یک تابع داده شده است؛ نیاز است این سه حالت را بخاطر بسپارید، امّا وقتی که این کار را انجام دادید تعیین حدهای مجانبی برای بسیاری از رابطه‌های بازگشتی ساده، آسان خواهد بود.

نکات تکنیکی

در عمل، وقتی که رابطه‌های بازگشتی را بیان و حل می‌کنیم، جزئیات تکنیکی مشخصی را نادیده

می‌گیریم. مثالی خوب از یک مورد جزئی که اغلب توضیح داده نمی‌شود، فرض آرگومان‌های صحیح برای توابع است. بطور معمول، زمان اجرای $T(n)$ یک الگوریتم فقط وقتی که n یک مقدار صحیح است تعریف می‌شود، زیرا برای اکثر الگوریتم‌ها اندازه ورودی همیشه یک مقدار صحیح است. برای مثال، رابطه بازگشتی که زمان اجرای $MERGE\text{-}SORT$ را بدترین حالت بیان می‌کند به شکل زیر است:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\ , \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1\ . \end{cases} \qquad (۴.۲)$$

شرایط مرزی، دسته دیگری از جزئیات را که نوعاً نادیده می‌گیریم بیان می‌کنند. از آنجا که زمان اجرای یک الگوریتم روی یک ورودی با اندازه ثابت برابر یک ثابت است، رابطه‌های بازگشتی که از زمان‌های اجرای الگوریتم‌ها بوجود می‌آیند عموماً برای nهای به اندازه کافی کوچک، دارای $T(n) = \Theta(1)$ هستند. در نتیجه، به منظور سهولت، بطور کلی عبارات شرایط مرزی رابطه‌های بازگشتی را ذکر نکرده و فرض می‌کنیم که $T(n)$ برای nهای کوچک ثابت است. بعنوان مثال، معمولاً رابطه بازگشتی (۴.۱) را بصورت زیر بیان می‌کنیم:

$$T(n) = 2T(n/2) + \Theta(n)\ , \qquad (۴.۳)$$

بدون اینکه صریحاً مقادیر را برای nهای کوچک ارائه دهیم. دلیل آن است که گرچه تغییر مقدار $T(1)$، جواب رابطه بازگشتی را تغییر می‌دهد امّا جواب معمولاً با بیشتر از یک ضریب ثابت تغییر نمی‌کند، بنابراین مرتبه رشد تغییر نمی‌کند.

وقتی که رابطه‌های بازگشتی را بیان و حل می‌کنیم اغلب کف‌ها، سقف‌ها، و شرایط مرزی را حذف می‌کنیم. بدون این جزئیات جلو می‌رویم و بعداً مشخص می‌کنیم که آیا آنها اهمیت دارند یا خیر. آنها معمولاً اهمیتی ندارند، اما مهم است که بدانیم چه موقع اهمیت دارند. تجربه کمک می‌کند و همینطور بیان بعضی از قضا یا مبنی بر اینکه این جزئیات تأثیری بر حدهای مجانبی بسیاری از رابطه‌های بازگشتی که در تحلیل الگوریتم‌ها با آنها مواجه می‌شویم، ندارند. (قضیه ۴.۱ را ملاحظه نمایید). بهر حال، در این فصل، بعضی از این جزئیات را مورد توجه قرار می‌دهیم تا نکات ریز روش‌های حل رابطه بازگشتی را نشان دهیم.

۴.۱ روش جایگذاری

روش جایگذاری برای حل رابطه‌های بازگشتی مستلزم دو گام می‌باشد:

۱. شکل جواب را حدس بزنید.

۲. از استقراری ریاضی به منظور یافتن ثابت‌ها و نشان دادن اینکه جواب کار می‌کند، استفاده کنید.

این نام از جایگذاری پاسخ حدس زده شده برای تابع وقتی که فرض استقرا برای مقادیر کوچکتر بکار برده می‌شود بدست آمده است. این روش قدرتمند است، اما بدیهی است که تنها می‌تواند در حالتهایی که حدس زدن شکل جواب آسان است، بکار رود.

روش جایگذاری می‌تواند برای بوجود آوردن حدهای بالا یا پایین روی یک رابطه بازگشتی استفاده شود. بعنوان مثال، اجازه دهید یک حد بالا روی رابطه بازگشتی

$$T(n) = 2T(\lfloor n/2 \rfloor)+n \qquad (۴.۴)$$

تعیین کنیم، که مشابه رابطه‌های بازگشتی (۴.۲) و (۴.۳) است. حدس می‌زنیم که جواب، $T(n) = O(n\lg n)$ است. روش ما این است که ثابت کنیم به ازای انتخاب مناسب ثابت $c > 0$ $T(n) \leq cn\lg n$ با این فرض که این حد برای $\lfloor n/2 \rfloor$ برقرار است شروع می‌کنیم، بعبارت دیگر، اینکه $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. با جایگذاری در رابطه بازگشتی داریم

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn\lg(n/2) + n \\ &= cn\lg n - cn\lg 2 + n \\ &= cn\lg n - cn + n \\ &< cn\lg n , \end{aligned}$$

که گام آخر تا زمانیکه $c \geq 1$ باشد بر قرار است.

اکنون برای نشان دادن اینکه جواب ما برای شرایط مرزی بر قرار است به استقرای ریاضی نیاز داریم. معمولاً این کار را با نشان دادن این که شرایط مرزی بعنوان حالتهای پایه برای اثبات استقرایی مناسب هستند انجام می‌دهیم. برای رابطه بازگشتی (۴.۴)، باید نشان دهیم که می‌توانیم ثابت c را به اندازه کافی بزرگ انتخاب کنیم تا حد $T(n) \leq cn\lg n$ برای شرایط مرزی نیز بدرستی کار کند. این نیاز می‌تواند گاهی اوقات به مشکلاتی منجر شود. برای اثبات اجازه دهید فرض کنیم تنها شرط مرزی رابطه بازگشتی $T(1) = 1$ است. آنگاه برای $n = 1$، حد $T(n) \leq cn\lg n$، $T(1) \leq c1\lg 1 = 0$ را بوجود می‌آورد، که با $T(1)=1$ اختلاف دارد. در نتیجه، حالت پایه اثبات استقرایی ما بر قرار نمی‌شود.

این اشکال در اثبات یک فرض استقرا برای یک شرط مرزی خاص می‌تواند به آسانی برطرف شود. بعنوان مثال، در رابطه بازگشتی (۴.۴)، با بهره‌گیری از نماد مجانبی تنها لازم است ثابت کنیم که برای $n \geq n_0$ $T(n) \leq cn\lg n$، که n_0 یک ثابت انتخابی ما است. ایده آن است که شرط مرزی $T(1)=1$ را در اثبات استقرایی مورد ملاحظه قرار ندهیم. توجه کنید که به ازای $n > 3$ رابطه بازگشتی مستقیماً به $T(1)$ بستگی ندارد. بنابراین، می‌توانیم با مقدار دهی $n_0=2$ $T(1)$ را با $T(2)$ و $T(3)$ بعنوان حالتهای پایه در اثبات استقرایی جایگزین کنیم. توجه کنید که تمایزی بین حالت پایه رابطه بازگشتی $(n=1)$ و

حالتهای پایه اثبات استقرایی ($n=2$ و $n=3$) قائل می‌شویم. از رابطه بازگشتی نتیجه می‌گیریم که $T(2)=4$ و $T(3)=5$. اکنون اثبات استقرایی که برای یک ثابت $c\geq1$، $T(n)\leq cnlgn$، می‌تواند با انتخاب c به اندازه کافی بزرگ بطوریکه $T(2)\leq c2lg2$ و $T(3)\leq c3lg3$، کامل شود. همانطور که انجام شد، هر انتخاب $c\geq2$ برای برقراری حالتهای پایه $n=2$ و $n=3$ کافی است. برای اکثر رابطه‌های بازگشتی که بررسی خواهیم کرد، گسترش شرایط مرزی برای ایجاد فرض استقرایی که برای n کوچک کار می‌کند، ساده است.

یک حدس خوب

متأسفانه یک روش کلی برای حدس زدن جواب‌های صحیح رابطه‌های بازگشتی وجود ندارد. حدس زدن یک جواب، به تجربه و گهگاه خلاقیت نیاز دارد. گرچه خوشبختانه، بعضی از مکاشفه‌ها وجود دارند که می‌توانند به شما کمک کنند تا یک حدس زننده خوب شوید. همچنین شما می‌توانید از درختهای بازگشت برای انجام حدس‌های خوب استفاده کنید، که آنها را در بخش ۴.۲ خواهیم دید.

اگر یک رابطه بازگشتی مشابه رابطه‌ای باشد که قبلاً دیده‌اید آنگاه حدس یک جواب مشابه، معقول است. بعنوان مثال، رابطه بازگشتی زیر را در نظر بگیرید

$$T(n) = 2T(\lfloor n/2 \rfloor +17)+n$$

که بخاطر جمع "17" در آرگومان T در سمت راست، سخت بنظر می‌آید. گرچه بطور شهودی این جمله اضافی نمی‌تواند بطور اساسی در جواب رابطه بازگشتی تأثیری بگذارد. وقتی n بزرگ است، تفاوت بین $T(\lfloor n/2 \rfloor)$ و $T(\lfloor n/2 \rfloor +17)$ زیاد نیست: هر دو تقریباً n را بطور مساوی به دو نیمه تقسیم می‌کنند. در نتیجه حدس می‌زنیم که $T(n)=O(nlgn)$، که شما می‌توانید با استفاده از روش جایگذاری صحت آن را بررسی کنید. (تمرین ۵-۴.۱ را ملاحظه نمایید).

روش دیگر برای یک حدس خوب این است که حدهای پایین و بالای ضعیفی را روی رابطه بازگشتی ثابت کنیم و سپس بازه عدم قطعیت را کاهش دهیم.

بعنوان مثال، با یک حد پایین $T(n)=\Omega(n)$ برای رابطه بازگشتی (۴.۴) شروع می‌کنیم، چون جمله n را در رابطه بازگشتی داریم، و می‌توانیم یک حد بالای اولیه $T(n)=O(n^2)$ را ثابت کنیم. آنگاه می‌توانیم بطور تدریجی حد بالا را کوچک کرده و حد پایین را بزرگ کنیم تا زمانیکه به جواب قوی مجانبی $T(n)=\Theta(nlgn)$ برسیم.

نکات ظریف

مواقعی وجود دارد که شما می‌توانید به درستی یک حد مجانبی روی جواب یک رابطه بازگشتی حدس بزنید، اما به دلیلی بنظر نمی‌آید که ریاضی در استقرا به نتیجه برسد. معمولاً مشکل این است که فرض

استقرا برای اثبات حد با جزئیات، به اندازه کافی قوی نیست، وقتی با چنین مانعی برخورد می‌کنید تجدیدنظر در حدس بوسیله تفریق یک جمله مرتبه پایین‌تر اجازه می‌دهد تا ریاضی به نتیجه برسد.

رابطه بازگشتی زیر را در نظر بگیرید

$$T(n)=T(\lfloor n/2\rfloor)+T(\lceil n/2\rceil)+1$$

حدس می‌زنیم که جواب $O(n)$ است، وسعی می‌کنیم تا نشان دهیم که به ازای انتخاب مناسب ثابت c، $T(n)\le cn$. با جایگذاری حدس خود در رابطه بازگشتی، بدست می‌آوریم

$$\begin{aligned} T(n) &\le c\lfloor n/2\rfloor + c\lceil n/2\rceil +1 \\ &=cn+1 \end{aligned}$$

که برای هر انتخاب از c، بر $T(n) \le cn$ دلالت نمی‌کند. وسوسه‌انگیز است که یک حدس بزرگتر $T(n)=O(n^2)$ بزنیم، که بتواند به درستی بکار رود، اما در حقیقت، این حدس که جواب $T(n)=O(n)$ است درست می‌باشد. هر چند به منظور نشان دادن این مطلب، باید از یک فرض استقرای قویتر استفاده کنیم.

بطور شهودی، حدس ما تقریباً درست است: فقط بخاطر جمله مرتبه پایین‌تر، یعنی ثابت 1 جواب ما اشتباه است. با وجود این، استقرای ریاضی کار نمی‌کند مگر اینکه شکل دقیق فرض استقرا را اثبات کنیم.

مشکلمان را با تفریق یک جمله مرتبه پایین‌تر از حدس قبلی خود رفع می‌کنیم. حدس جدید ما، $T(n)\le cn-b$ است، که $b\ge 0$ ثابت است. اکنون تا هنگامیکه $b\ge 1$، داریم

$$\begin{aligned} T(n) &\le (c\lfloor n/2\rfloor - b) + (c\lceil n/2\rceil - b) + 1 \\ &= cn - 2b + 1 \\ &\le cn - b\,, \end{aligned}$$

مانند قبل، ثابت c باید به اندازه کافی بزرگ انتخاب شود تا شرایط مرزی را مدیریّت کنیم.

اکثر مردم ایده تفریق یک جمله مرتبه پایین‌تر را غیر معقول می‌دانند. با این وجود، اگر ریاضی به نتیجه نرسد آیا نباید حدس خود را افزایش دهیم؟ کلید درک این مرحله این است که به یاد داشته باشیم در حال استفاده از استقرای ریاضی هستیم: می‌توانیم یک چیز قویتر را برای یک مقدار داده شده با فرض یک چیز قویتر برای مقادیر کوچکتر اثبات کنیم.

اجتناب از تله‌ها

خطا کردن در استفاده از نماد مجانبی آسان است. بعنوان مثال، در رابطه بازگشتی (۴.۴) می‌توانیم به اشتباه با حدس $T(n)\le cn$ "ثابت کنیم" $T(n)=O(n)$ و سپس ثابت کنیم

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2\rfloor)+n \\ &\le cn+n \\ &= O(n), \end{aligned}$$
⟸ اشتباه!!

زیرا c یک ثابت است. اشتباه این است که شکل دقیق فرض استقرا را ثابت نکرده‌ایم، بعبارت دیگر این که $T(n) \le cn$.

تغییر متغیرها

گاهی اوقات، یک دستکاری جبری کوچک می‌تواند یک رابطه بازگشتی ناآشنا را مشابه یک رابطه که قبلاً دیده‌اید، نماید. بعنوان مثال، رابطه بازگشتی زیر را در نظر بگیرید

$$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \lg n,$$

که بنظر مشکل می‌آید. اگر چه می‌توانیم این رابطه بازگشتی را با تغییر متغیرها ساده کنیم. به منظور سهولت، نگران مقادیری که به اعداد صحیح گرد نمی‌شوند، مانند $\sqrt{n}$، نیستیم. با تغییر نام $m=lgn$ داریم

$$T(2^m)=2T(2^{m/2})+m$$

اکنون می‌توانیم تغییر نام $S(m)=T(2^m)$ را انجام دهیم تا رابطه بازگشتی جدید

$$S(m)=2S(m/2)+m$$

ایجاد شود، که بسیار مشابه رابطه بازگشتی (۴.۴) است. در حقیقت، این رابطه بازگشتی جدید جواب یکسانی دارد: $S(m) = O(mlgm)$. با تغییر معکوس از $S(m)$ به $T(n)$، بدست می‌آوریم:

$$T(n) = T(2^m) = S(m) = O(m\lg m) = O(\lg n \lg\lg n).$$

تمرین‌ها

۴.۱-۱ نشان دهید که جواب $T(n)=T(\lfloor n/2\rfloor)+1$ برابر $O(lgn)$ است.

۴.۱-۲ مشاهده کردیم که جواب $T(n)=2T(\lfloor n/2\rfloor)+n$ برابر $O(nlgn)$ است. نشان دهید که جواب این رابطه بازگشتی، $\Omega(nlgn)$ نیز است. نتیجه بگیرید که جواب $\Theta(nlgn)$ است.

۴.۱-۳ نشان دهید که با ایجاد یک فرض استقرای متفاوت می‌توانیم مشکل شرط مرزی $T(1)=1$ برای رابطه بازگشتی (۴.۴) را بدون تطبیق شرایط مرزی برای اثبات استقرایی برطرف کنیم.

۴.۱-۴ نشان دهید که $\Theta(nlgn)$ جواب رابطه بازگشتی "دقیق" (۴.۲) برای مرتب‌سازی ادغام است.

۴.۱-۵ نشان دهید که جواب $T(n)=2T(\lfloor n/2\rfloor+17)+n$ برابر $O(nlgn)$ است.

۴.۱-۶ رابطه بازگشتی $T(n)=2T(\sqrt{n})+1$ را با ایجاد تغییری در متغیرها حل کنید. جواب شما باید بطور مجانبی قوی باشد. نگران این که آیا متغیرها صحیح هستند یا خیر نباشید.

۴.۲ روش درخت بازگشت

اگر چه روش جایگذاری می‌تواند اثبات مختصر و مفیدی را فراهم کند که جواب یک رابطه بازگشتی درست است، گاهی اوقات جلو رفتن با یک حدس خوب مشکل است. رسم یک درخت بازگشت، همانطور که در تحلیل رابطه بازگشتی مرتب سازی ادغام در بخش ۲.۳.۲انجام دادیم، روشی ساده برای انجام یک حدس خوب است. در یک **درخت بازگشت**، هر گره بیانگر هزینه یک زیر مسئله در مجموعه فراخوانی‌های تابع بازگشتی را نشان می‌دهد. هزینه‌های داخل هر سطح درخت را جمع می‌کنیم تا مجموعه هزینه‌های هر سطح را بدست آوریم، و سپس تمام هزینه‌های هر سطح را جمع می‌کنیم تا هزینه کل همه سطوح رابطه بازگشتی را تعیین کنیم. درختهای بازگشت بویژه زمانیکه رابطه بازگشتی، زمان اجرای یک الگوریتم تقسیم و حل را توصیف می‌کند مفید هستند.

یک درخت بازگشت به بهترین شکل برای تولید یک حدس خوب، که بعداً بوسیله روش جایگذاری تأیید و اثبات می‌شود، استفاده می‌گردد. وقتی از درخت بازگشت برای تولید یک حدس خوب استفاده می‌کنید، اغلب می‌توانید کمی "بی دقتی" را جایز بشمارید، زیرا صحّت حدس خود را بعداً تعیین خواهید کرد. امّا اگر زمانیکه یک درخت بازگشت را رسم کرده و هزینه‌ها را جمع می‌کنید. خیلی دقیق هستید، می‌توانید از درخت بازگشت بعنوان اثبات مستقیم جواب یک رابطه بازگشت استفاده کنید. در این بخش، از درختهای بازگشت برای ایجاد حدس‌های خوب استفاده خواهیم کرد، و در بخش ۴.۴ از درختهای بازگشت مستقیماً برای اثبات قضیه‌ای که پایه و اساس روش اصلی را شکل می‌دهد، استفاده خواهیم کرد.

بعنوان مثال، اجازه دهید ببینیم که چطور یک درخت بازگشت یک حدس خوب برای رابطه بازگشتی $T(n)=3T(\lfloor n/4\rfloor)+\Theta(n^2)$ ایجاد می‌کند. با تمرکز روی یافتن یک حد بالا برای جواب شروع می‌کنیم. چون می‌دانیم که کف‌ها و سقف‌ها معمولاً در حل رابطه‌های بازگشتی قابل توجه نیستند (در اینجا نمونه‌ای از بی‌دقّتی که می‌توانیم جایز بدانیم وجود دارد)، با نوشتن ضریب ثابت غیر صریح $c>0$ ، درخت بازگشت برای رابطه بازگشتی $T(n)=3T(n/4)+cn^2$ را ایجاد می‌کنیم.

شکل ۴.۱ اشتقاق درخت بازگشت برای $T(n)=3T(n/4)+cn^2$، را نشان می‌دهد. به منظور سهولت، فرض می‌کنیم که n توان دقیقی از 4 است (نمونه دیگری از بی‌دقّتی مجاز). قسمت (a) شکل، $T(n)$ را نشان می‌دهد که در قسمت (b) به شکل یک درخت معادل برای نمایش رابطه بازگشتی گسترش می‌یابد. جمله cn^2 در ریشه، هزینه بالاترین سطح رابطه بازگشتی را نمایش می‌دهد و سه زیر درخت ریشه، هزینه‌های ایجاد شده توسط زیر مسائل با اندازه $n/4$ را نمایش می‌دهند. قسمت (c) با گسترش هر گره

با هزینه $T(n/4)$ از قسمت (b)، فرآیند گام بعدی را نشان می‌دهد. هزینه هر کدام از سه فرزند ریشه، $c(n/4)^2$ است. گسترش هر گره در درخت را با شکستن آن گره به قسمتهای سازنده‌اش، بصورتی که توسط رابطه بازگشتی تعیین می‌شود، ادامه می‌دهیم.

چون همانطور که از ریشه دور می‌شویم اندازه‌های زیر مسئله‌ها کاهش می‌یابند، در نهایت باید به یک شرط مرزی برسیم. در چه فاصله‌ای از ریشه به شرط مرزی می‌رسیم؟ اندازه زیر مسئله برای یک گره در عمق i، $n/4^i$ است. بنابراین، وقتی که $n/4^i=1$ یا بطور معادل، وقتی که $i=\log_4 n$ اندازه زیر مسئله به $n=1$ می‌رسد. بنابراین، درخت $\log_4 n+1$ سطح $(0, 1, 2, \ldots, \log_4^n)$ دارد.

آنگاه هزینه هر سطح درخت را تعیین می‌کنیم. تعداد گره‌ها در هر سطح سه برابر تعداد گره‌ها در سطح قبلی است و بنابراین تعداد گره‌های عمق i، 3^i است. چون اندازه‌های زیر مسئله‌ها برای هر سطح که از ریشه به پایین می‌آییم با ضریب 4 کاهش می‌یابند، برای $i = 0, 1, 2, \ldots, \log_4 n-1$، هر گره در عمق i هزینه‌ای برابر $c(n/4^i)^2$ دارد. با ضرب کردن می‌بینیم که برای $i = 0, 1, 2, \ldots, \log_4 n-1$، هزینه کل تمام گره‌ها در عمق i برابر $3^i c(n/4^i)^2=(3/16)^i cn^2$ است. سطح آخر، در عمق $\log_4 n$،

$3^{\log_4 n} = n^{\log_4 3}$

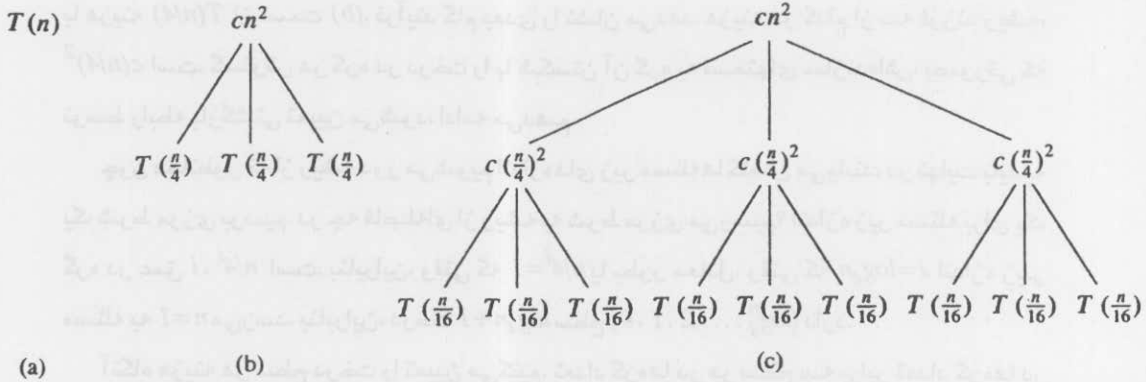
گره دارد که هر کدام هزینه $T(1)$ را دارند و هزینه کل برابر $n^{\log_4 3}T(1)$ است که $\Theta(n^{\log_4 3})$ می‌باشد.

اکنون هزینه‌های همه سطح‌ها را جمع می‌کنیم تا هزینه کل درخت را تعیین کنیم:

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) . \end{aligned}$$

فرمول اخیر، تا آن زمان که دوباره از کمی بی‌دقتی سود برده و از یک سری هندسی نزولی نامتناهی بعنوان حدّ بالا استفاده کنیم، تا اندازه‌ای در هم و بر هم بنظر می‌رسد، با یک گام بازگشت به عقب بکارگیری معادله $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ داریم:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\ &= O(n^2) . \end{aligned}$$



شکل ۴.۱ ساختار درخت بازگشت برای رابطه بازگشتی $T(n)=3T(n/4)+cn^2$. قسمت *(a)*، $T(n)$ را نشان می‌دهد، که رفته رفته در *(b)-(d)* برای تشکیل درخت بازگشت گسترش می‌یابد. درخت کاملاً گسترش یافته در قسمت *(d)* دارای ارتفاع $log^n{}_4$ است (این درخت $log^n{}_4+1$ سطح دارد).

بنابراین، به حدس $T(n) = O(n^2)$ برای رابطه بازگشتی اصلی $T(n)=3T(\lfloor n/4 \rfloor)+\Theta(n^2)$ رسیده‌ایم.

در این مثال، ضرایب cn^2 یک سری هندسی نزولی را می‌سازند، و بنابر معادله $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$

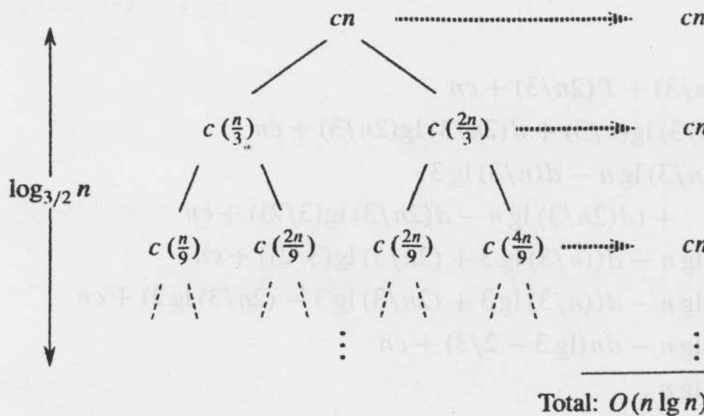
مجموع این ضرایب از بالا با ثابت $16/13$ محدود می‌شود. از آنجا که سهم ریشه در هزینه کل، cn^2 است ریشه کسر ثابتی از هزینه کل را صرف می‌کند. بعبارت دیگر، هزینه کلّی درخت تحت‌الشعاع هزینه ریشه قرار می‌گیرد.

در حقیقت، اگر $O(n^2)$ واقعاً یک حد بالا برای رابطه بازگشتی باشد (همانطوریکه بزودی بررسی خواهیم کرد)، آنگاه باید یک حد قوی باشد. چرا؟ اولین فراخوانی بازگشتی، هزینه $\Theta(n^2)$ را صرف می‌کند و بنابراین $\Omega(n^2)$ باید یک حد پایین برای رابطه بازگشتی باشد.

اکنون می‌توانیم از روش جایگذاری برای تعیین این که آیا حدسمان درست بوده است استفاده کنیم، بعبارت دیگر، $T(n)=O(n^2)$، یک حد بالا برای رابطه بازگشتی $T(n)=3T(\lfloor n/4 \rfloor)+\Theta(n^2)$ است. می‌خواهیم نشان دهیم که برای ثابت $d>0$، $T(n)\le dn^2$. با استفاده از همان ثابت $c>0$ مانند قبل داریم

$$\begin{aligned}
T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\
&\le 3d\lfloor n/4 \rfloor^2 + cn^2 \\
&\le 3d(n/4)^2 + cn^2 \\
&= \frac{3}{16}dn^2 + cn^2 \\
&\le dn^2 ,
\end{aligned}$$

که گام آخر تا زمانیکه $d\ge(16/13)$ برقرار است.



شکل ۴.۲ یک درخت بازگشت برای رابطهٔ بازگشتی $T(n)=T(n/3)+T(2n/3)+cn$.

بعنوان یک مثال پیچیده‌تر دیگر، شکل(۴.۲) درخت بازگشت برای $T(n)=T(n/3)+T(2n/3)+O(n)$ را نشان می‌دهد. (دوباره توابع سقف و کف را برای سادگی حذف می‌کنیم.) مانند قبل، c را ضریب ثابتی در جمله $O(n)$ در نظر می‌گیریم. هنگامیکه مقادیر روی سطوح درخت بازگشت را جمع می‌کنیم، به مقدار cn برای هر سطح می‌رسیم. طولانی‌ترین مسیر از ریشه به یک برگ

$n\rightarrow(2/3)n\rightarrow(2/3)^2n\rightarrow...\rightarrow 1$

است. وقتی $k=log^n{}_{3/2}$ داریم $(2/3)^k\ n=1$، بنابراین ارتفاع درخت $log^n{}_{3/2}$است.

بطور شهودی، انتظار داریم جواب رابطه بازگشتی حداکثر برابر تعداد سطوح ضرب در هزینه هر سطح یا بعبارتی $O(cnlog^n{}_{3/2})=O(nlgn)$ باشد. هزینه کل بطور یکنواخت در طول سطوح درخت بازگشت توزیع می‌شود. یک پیچیدگی در اینجا وجود دارد: باز هم باید هزینه برگها را در نظر بگیریم. اگر این درخت بازگشت یک درخت دودویی کامل با ارتفاع $log^n{}_{3/2}$ بود، $2^{logn}{}_{3/2}=n^{log2}{}_{3/2}$ برگ وجود داشت. از آنجا که هزینه هر برگ ثابت است، پس هزینه کل همه برگها $\Theta(n^{log2}{}_{3/2})$ خواهد بود، که $w(nlgn)$ است. اما این درخت بازگشت، یک درخت دودویی کامل نیست و بنابراین کمتر از $n^{log2}{}_{3/2}$ برگ دارد. بعلاوه، همان‌طور که از ریشه پایین می‌آییم برگ‌های داخلی، بیشتر و بیشتر پنهان می‌شوند. در نتیجه، همه سطح‌ها هزینه دقیق cn را صرف نمی‌کنند؛ سطح‌های پایینی هزینه کمتری را صرف می‌کنند. می‌توانستیم همه هزینه‌ها را بطور دقیق حساب کنیم، اما بخاطر آورید که تنها سعی می‌کنیم تا با یک حدس برای استفاده در روش جایگذاری پیش برویم. اجازه دهید تا بی‌دقتی را جایز شمرده و سعی کنیم نشان دهیم که حدس $O(nlgn)$ برای حد بالا درست است.

در حقیقت می‌توانیم از روش جایگذاری برای بررسی صحّت این که $O(nlgn)$ یک حد بالا برای جواب رابطه بازگشتی است استفاده کنیم. نشان می‌دهیم $T(n)\le dnlgn$، که در آن d یک ثابت مثبت مناسب است. تا زمانیکه $d\ge c/(lg3-(2/3))$، داریم

$$\begin{aligned}
T(n) &\le T(n/3)+T(2n/3)+cn \\
&\le d(n/3)\lg(n/3)+d(2n/3)\lg(2n/3)+cn \\
&= (d(n/3)\lg n-d(n/3)\lg 3) \\
&\qquad +(d(2n/3)\lg n-d(2n/3)\lg(3/2))+cn \\
&= dn\lg n-d((n/3)\lg 3+(2n/3)\lg(3/2))+cn \\
&= dn\lg n-d((n/3)\lg 3+(2n/3)\lg 3-(2n/3)\lg 2)+cn \\
&= dn\lg n-dn(\lg 3-2/3)+cn \\
&\le dn\lg n\,,
\end{aligned}$$

بنابراین، مجبور نبودیم که هزینه‌ها را در درخت بازگشت بطور دقیق محاسبه کنیم.

تمرین‌ها

۴.۲-۱ از یک درخت بازگشت برای تعیین یک حد بالای مجانبی مناسب روی رابطه بازگشتی $T(n) = 3T(\lfloor n/2 \rfloor) + n$ استفاده کنید. از روش جایگذاری برای تعیین صحّت پاسخ خود استفاده کنید.

۴.۲-۲ ثابت کنید جواب رابطه بازگشتی $T(n) = T(n/3)+T(2n/3)+cn$، که در آن c یک ثابت است، با استفاده از درخت بازگشت، $\Omega(nlgn)$ است.

۴.۲-۳ درخت بازگشت را برای $T(n) = 4T(\lfloor n/2 \rfloor) + cn$، که در آن c یک ثابت است، رسم کنید. و یک حد مجانبی قوی روی جواب آن بدست آورید. درستی حد خود را با روش جایگذاری تعیین کنید.

۴.۲-۴ از یک درخت بازگشت برای ارائه یک جواب بطور مجانبی قوی برای رابطه بازگشتی $T(n) = T(n-a) + T(a) + cn$، که در آن $a \geq 1$ و $c > 0$ ثابت هستند، استفاده کنید.

۴.۲-۵ از یک درخت بازگشت برای ارائه یک جواب بطور مجانبی قوی برای رابطه بازگشتی $T(n)=T(\alpha n)+T((1-\alpha)n)+cn$، که در آن α یک ثابت دربازه $0<\alpha<1$ و $c>0$ نیز یک ثابت است، استفاده کنید.

۴.۳ روش اصلی

روش اصلی یک روش مستقیم و چارچوب بندی شده برای حل رابطه‌های بازگشتی به شکل

$$T(n)=aT(n/b)+f(n) \qquad (۴.۵)$$

فراهم می‌کند، که در آن $a\geq1$ و $b>1$ ثابت هستند و $f(n)$ یک تابع بطور مجانبی مثبت است. در روش اصلی نیاز به حفظ سه حالت است، اما پس از آن جواب بسیاری از رابطه‌های بازگشتی می‌تواند بسیار ساده، اغلب بدون قلم و کاغذ تعیین شود.

رابطه بازگشتی (۴.۵) زمان اجرای یک الگوریتم را بیان می‌کند که مسئله‌ای با اندازه n را به a زیر مسئله هر کدام با اندازه n/b تقسیم می‌کند، که a و b ثابت‌های مثبتی هستند. a زیر مسئله بصورت بازگشتی، هر کدام در زمان $T(n/b)$ حل می‌شوند. هزینه تقسیم مسئله و ترکیب نتایج زیر مسائل بوسیله تابع $f(n)$ بیان می‌شوند. (بعبارت دیگر، با استفاده از نمادگذاری از بخش ۲.۳.۲، $f(n)=D(n)+C(n)$). برای مثال، رابطه بازگشتی که از روال *MERGE-SORT* ناشی می‌شود دارای $a=2$، $b=2$، و $f(n)=\Theta(n)$ است.

بعنوان یک مطلب از صحت تکنیکی، رابطه بازگشتی، واقعاً خوش تعریف نیست زیرا ممکن است n/b یک عدد صحیح نباشد. اگر چه جایگذاری هر کدام از a جمله $T(n/b)$ با $T(\lfloor n/b \rfloor)$ یا $T(\lceil n/b \rceil)$ تأثیری بر رفتار مجانبی رابطه بازگشتی ندارد. (این مطلب را در بخش بعد اثبات خواهیم کرد.) بنابراین، معمولاً

حذف توابع سقف و کف را در هنگام نوشتن رابطه‌های بازگشتی تقسیم و حل به این شکل، مناسب می‌یابیم.

قضیهٔ اصلی

قضیه اصلی وابسته به قضیه زیر است.

قضیه ۴.۱ (قضیه اصلی)

فرض کنید $a \geq 1$ و $b > 1$ ثابت باشند و $f(n)$ یک تابع باشد، و فرض کنید $T(n)$ روی اعداد صحیح غیرمنفی توسط رابطه بازگشتی زیر تعریف شده باشد

$$T(n) = aT(n/b) + f(n)$$

که در آن n/b را به مفهوم $\lfloor n/b \rfloor$ یا $\lceil n/b \rceil$ تفسیر می‌کنیم. پس $T(n)$ می‌تواند بطور مجانبی بصورت زیر محدود شده باشد.

۱. اگر به ازای ثابت $\varepsilon > 0$، $f(n) = O(n^{\log_b a - \varepsilon})$، آنگاه $T(n) = \Theta(n^{\log_b a})$.

۲. اگر $f(n) = \Theta(n^{\log_b a})$، آنگاه $T(n) = \Theta(n^{\log_b a} \lg n)$.

۳. اگر به ازای ثابت $\varepsilon > 0$، $f(n) = \Omega(n^{\log_b a + \varepsilon})$، و اگر به ازای ثابت $c < 1$ و همه n های به اندازه کافی بزرگ، $af(n/b) \leq cf(n)$، آنگاه $T(n) = \Theta(f(n))$. ■

قبل از بکارگیری قضیه اصلی در مثال‌ها، اجازه دهید بفهمیم که این قضیه چه می‌گوید. در هر کدام از سه حالت، در حال مقایسه تابع $f(n)$ با تابع $n^{\log_b a}$ هستیم. بطور شهودی، جواب رابطه بازگشتی بوسیله بزرگترین تابع تعیین می‌شود. اگر همانند حالت ۱، تابع $n^{\log_b a}$ بزرگتر باشد آنگاه جواب برابر $T(n) = \Theta(n^{\log_b a})$ است. اگر همانند حالت ۳، تابع $f(n)$ بزرگتر باشد آنگاه جواب برابر $T(n) = \Theta(f(n))$ است. اگر همانند حالت ۲، دو تابع هم اندازه باشند در یک ضریب لگاریتمی ضرب می‌کنیم، و جواب برابر $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$ است.

ماورای این شهود، تکنیک‌هایی وجود دارند که باید فهمیده شوند. در حالت اول، نه تنها باید $f(n)$ کوچکتر از $n^{\log_b a}$ باشد بلکه باید بصورت چند جمله‌ای کوچکتر باشد. بعبارت دیگر، $f(n)$ باید بطور مجانبی کوچکتر از $n^{\log_b a}$ با یک ضریب n^{ε} به ازای ثابت $\varepsilon > 0$، باشد. در حالت سوم، نه تنها باید $f(n)$ بزرگتر از $n^{\log_b a}$ باشد، بلکه باید بصورت چند جمله‌ای بزرگتر باشد و بعلاوه در شرط "نظم" که $af(n/b) \leq cf(n)$، صدق کند. اکثر توابع بصورت چند جمله‌ای محدود که با آنها مواجه خواهیم شد، در این شرط صدق می‌کنند. مهم است که بدانید این سه حالت، همه حالات ممکن برای $f(n)$ را در بر نمی‌گیرند. فاصله‌ای بین حالت‌های ۱ و ۲ وقتی که $f(n)$ کوچکتر از $n^{\log_b a}$ است، اما نه بصورت چند جمله‌ای، وجود دارد. بطور مشابه، فاصله‌ای بین حالت‌های ۲ و ۳ وقتی که $f(n)$ بزرگتر از $n^{\log_b a}$ است،

اما نه بصورت چند جمله‌ای، وجود دارد. اگر تابع $f(n)$ در یکی از این فاصله‌ها قرار گیرد یا اگر شرط نظم در حالت ۳ برقرار نشود، روش اصلی نمی‌تواند برای حل رابطه بازگشتی استفاده شود.

استفاده از روش اصلی

برای استفاده از روش اصلی، بسادگی تعیین می‌کنیم که کدام حالت از قضیه اصلی (اگر باشد) بکار می‌رود و پاسخ را می‌نویسیم.

بعنوان اولین مثال،

$$T(n) = 9T(n/3)+n$$

را در نظر بگیرید. برای این رابطه بازگشتی، داریم $a=9$، $b=3$، $f(n)=n$، و بنابراین داریم $n^{\log_b a}=n^{\log_3 9}=\Theta(n^2)$. از آنجا که $f(n) = O(n^{\log_3 9-\varepsilon})$، که $\varepsilon=1$، می‌توانیم حالت ۱ قضیه اصلی را بکار ببریم و نتیجه بگیریم که جواب برابر $T(n) = \Theta(n^2)$ است.

اکنون،

$$T(n) = T(2n/3)+1$$

را در نظر بگیرید که در آن $a=1$، $b=3/2$، $f(n)=1$، و $n^{\log_b a}=n^{\log_{3/2} 1}=n^0=1$. حالت ۲ بکار می‌رود زیرا $f(n)=\Theta(n^{\log_b a})=\Theta(1)$، و بنابراین جواب رابطه بازگشتی برابر $T(n)=\Theta(\lg n)$ است.

برای رابطه بازگشتی

$$T(n)=3T(n/4)+n\lg n$$

داریم $a=3$، $b=4$، $f(n)=n\lg n$، و $n^{\log_b a}=n^{\log_4 3}=O(n^{0.793})$. از آنجا که $f(n)=\Omega(n^{\log_4 3+\varepsilon})$ که $\varepsilon \approx 0.2$، اگر بتوانیم نشان دهیم که شرط نظم برای $f(n)$ بر قرار است، حالت ۳ بکار می‌رود. برای n به اندازه کافی بزرگ و برای $c=3/4$،

$$af(n/b)=3(n/4)\lg(n/4)\le(3/4)n\lg n = cf(n)$$

در نتیجه بنا به حالت ۳، جواب رابطه بازگشتی $T(n)=\Theta(n\lg n)$ است.

روش اصلی برای رابطه بازگشتی زیر بکار نمی‌رود.

$$T(n)=2T(n/2)+n\lg n,$$

با آنکه دارای شکل مناسب است: $a=2$، $b=2$، $f(n)=n\lg n$، و $n^{\log_b a}=n$ است. ممکن است بنظر رسد که حالت ۳ باید بکار رود، زیرا $f(n)=n\lg n$ بطور مجانبی بزرگتر از $n^{\log_b a}=n$ است. مشکل آن است که بصورت چند جمله‌ای بزرگتر نیست. نسبت $f(n)/n^{\log_b a}=(n\lg n)/n=\lg n$ بطور مجانبی کمتر از n^{ε} به ازای هر ثابت مثبت ε است. در نتیجه، رابطه بازگشتی در فاصله بین حالت ۲ و حالت ۳ قرار می‌گیرد. (تمرین ۲-۴.۴ را برای جواب ملاحظه نمایید.)

تمرین‌ها

۱-۴.۳-۱ از روش اصلی برای ارائه حدهای مجانبی قوی برای رابطه‌های بازگشتی زیر استفاده کنید.

a. $T(n) = 4T(n/2) + n.$

b. $T(n) = 4T(n/2) + n^2.$

c. $T(n) = 4T(n/2) + n^3.$

۴.۳-۲ رابطه بازگشتی $T(n)=7T(n/2) + n^2$ زمان اجرای الگوریتم A را بیان می‌کند. الگوریتم رقیب A' دارای زمان اجرای $T'(n)=aT'(n/4) + n^2$ است. بزرگترین مقدار صحیح برای a بطوریکه A' بطور مجانبی سریعتر از A باشد، چیست؟

۴.۳-۳ با استفاده از روش اصلی نشان دهید که جواب رابطه بازگشتی $T(n) = T(n/2) + \Theta(1)$ مربوط به جستجوی دودویی برابر $T(n)=\Theta(\lg n)$ است. (تمرین ۵-۲.۳ برای تعریف جستجوی دودویی ملاحظه نمایید.)

۴.۳-۴ آیا روش اصلی می‌تواند در رابطه بازگشتی $T(n)=4T(n/2)+n^2\lg n$ بکار برده شود؟ چرا بله یا چرا خیر؟ یک حد بالای مجانبی برای این رابطه بازگشتی ارائه دهید.

*۴.۳-۵ شرط نظم $af(n/b) \leq cf(n)$ را به ازای ثابت $c<1$ در نظر بگیرید، که قسمتی از حالت ۳ قضیه اصلی است. نمونه‌ای از ثابت‌های $a\geq 1$ و $b>1$ و یک تابع $f(n)$ ارائه دهید که در همه شرایط حالت ۳ قضیه اصلی غیر از شرط نظم صدق کنند.

---

* ۴.۴ اثبات قضیه اصلی

این بخش شامل اثبات قضیه اصلی (قضیه ۴.۱) است. نیازی به فهمیدن این اثبات به منظور بکار بردن این قضیه نمی‌باشد.

اثبات در دو قسمت است. قسمت اول رابطه بازگشتی "اصلی" (۴.۵) را تحلیل می‌کند، با این فرض ساده کننده که $T(n)$ فقط روی توانهای کاملی از $b>1$، بعبارت دیگر به ازای $n=1, b, b^2, \ldots$ تعریف می‌شود. این قسمت تمام شهود لازم برای فهمیدن این مطلب که چرا قضیه اصلی درست است را ارائه می‌کند. قسمت دوم نشان می‌دهد که چگونه تحلیل می‌تواند به همه اعداد صحیح مثبت n تعمیم داده شود و تکنیکی صرفاً ریاضی است که برای مسئله مدیریت کف‌ها و سقف‌ها بکار می‌رود.

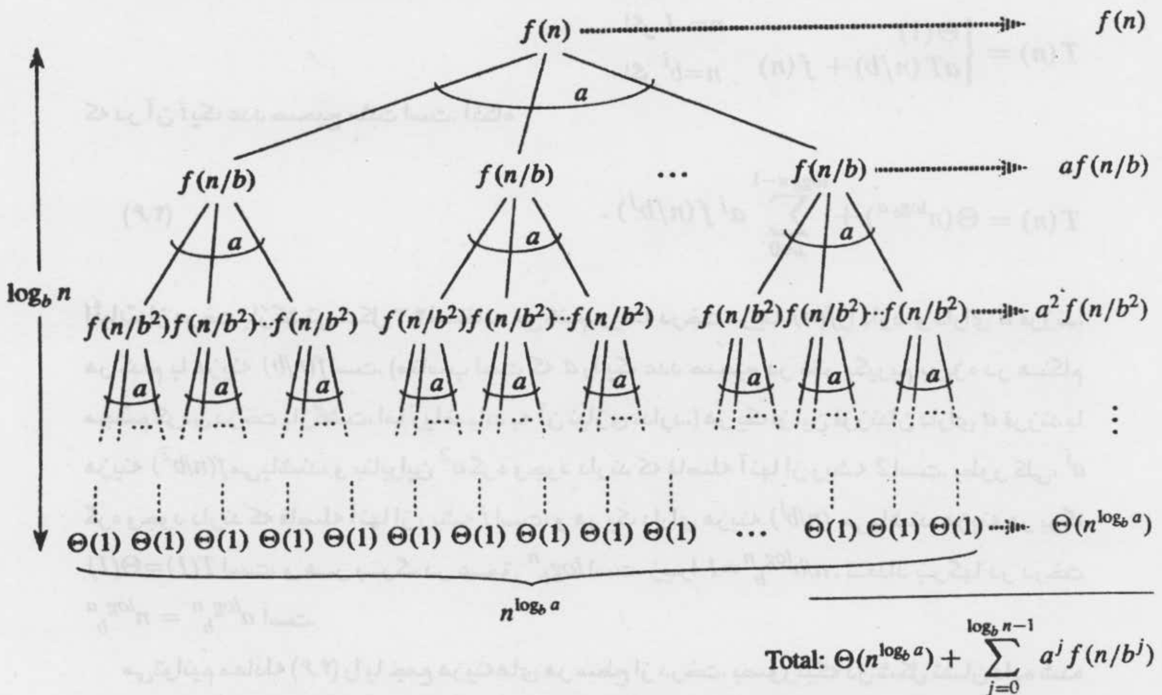
در این بخش، گاهی اوقات از نماد مجانبی استفاده نابجای مختصری می‌کنیم. بدین صورت که از آن برای بیان رفتار توابعی که فقط روی توان‌های کامل b تعریف می‌شوند استفاده می‌نمائیم. بخاطر آورید که در تعاریف نمادهای مجانبی نیاز است که حدها برای همه اعداد به اندازه کافی بزرگ اثبات شده باشند، نه فقط آنهایی که توانهایی از b هستند. از آنجا که می‌توانیم نمادهای مجانبی جدیدی

بسازیم که در مجموعه $\{b^i : i = 0, 1, \ldots\}$ بجای اعداد صحیح غیر منفی بکار می‌روند، استفاده نابجا مختصرتر است.

با وجود این، زمانی که در حال استفاده از نماد مجانبی روی یک دامنه محدود هستیم باید همیشه مواظب باشیم که نتایج نامناسبی را بدست نیاوریم. بعنوان مثال، اثبات این مطلب که وقتی n یک توان کامل از 2 است، $T(n) = O(n)$، تضمین نمی‌کند که $T(n) = O(n)$. تابع $T(n)$ می‌توانست بصورت زیر تعریف شود.

$$T(n) = \begin{cases} n & \text{اگر } n=1, 2, 4, 8, \ldots \\ n^2 & \text{در غیر اینصورت} \end{cases}$$

که در این حالت، بهترین حد بالا که می‌تواند ثابت شود $T(n) = O(n^2)$ است. بخاطر وجود چنین پیامدهای حادّی، هرگز از نمادگذاری مجانبی روی یک دامنه محدود بدون مشخص کردن دقیق آن از روی موضوعی که به آن می‌پردازیم، استفاده نخواهیم کرد.



شکل ۴.۳ درخت بازگشت تولید شده توسط $T(n) = aT(n/b)+f(n)$. این درخت یک درخت a- تایی کامل با $n^{\log_b a}$ برگ و ارتفاع $\log_b n$ است. هزینه هر سطح در سمت راست نشان داده شده است، و مجموع آنها در معادله (۴.۶) ارائه شده است.

۴.۴.۱ اثبات برای توانهای کامل

قسمت اول اثبات قضیه اصلی، رابطه بازگشتی (۴.۵)

$T(n) = aT(n/b) + f(n)$

را برای روش اصلی، با این فرض که n یک توان کامل از $b>1$ است تحلیل می‌کند، که در آن نیاز نیست b یک عدد صحیح باشد. این تحلیل به سه لم شکسته می‌شود. لم اول مسئله حل رابطه بازگشتی اصلی را به مسئله ارزشیابی یک عبارت که شامل یک حاصلجمع است، تبدیل می‌کند. دومین لم، حدهایی روی این حاصلجمع تعیین می‌کند. سومین لم، برای اثبات صورتی از قضیه اصلی برای حالتی که در آن n توان کاملی از b است، دو لم اول را با هم ترکیب می‌کند.

لم ۴.۲

فرض کنید $a \le 1$ و $b>1$ ثابت باشند، و $f(n)$ را یک تابع غیر منفی تعریف شده روی توانهای کاملی از b در نظر بگیرید. $T(n)$ را روی توانهای کاملی از b توسط رابطه بازگشتی زیر تعریف کنید

$$T(n) = \begin{cases} \Theta(1) & \text{اگر } n=1 \\ aT(n/b) + f(n) & \text{اگر } n=b^i \text{ ،} \end{cases}$$

که در آن i یک عدد صحیح مثبت است. آنگاه

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) . \qquad (۴۶)$$

اثبات از درخت بازگشتی شکل ۴.۳ استفاده می‌کنیم. ریشه درخت هزینه $f(n)$ را دارد، و دارای a فرزند، هر کدام با هزینه $f(n/b)$ است. (مناسب است که a را یک عدد صحیح در نظر بگیریم بویژه در هنگام مجسّم کردن درخت بازگشت، اما ریاضیات به آن نیازی ندارد.) هر یک از این فرزندان دارای a فرزند با هزینه $f(n/b^2)$ می‌باشند، و بنابراین a^2 گره وجود دارند که فاصله آنها از ریشه 2 است. بطور کلی، a^j گره وجود دارند که فاصله آنها از ریشه j است، و هر یک دارای هزینه (n/b^j) می‌باشند. هزینه هر برگ $T(1)=\Theta(1)$ است، و هر برگ در عمق $\log_b n$ است، زیرا $n/b^{\log_b n}=1$. تعداد برگها در درخت $a^{\log_b n} = n^{\log_b a}$ است.

می‌توانیم معادله (۴.۶) را با جمع هزینه‌های هر سطح از درخت، بصورتیکه در شکل نشان داده شده است، بدست آوریم. هزینه برای سطح j از گره‌های داخلی برابر $a^j f(n/b^j)$ است، و بنابراین هزینه کل سطوح گره‌های داخلی برابر است با

$$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j) .$$

در الگوریتم تقسیم و حل بنیادی، این حاصلجمع، هزینه‌های تقسیم مسائل به زیر مسائل و سپس ترکیب مجدد زیر مسائل را نشان می‌دهد. هزینه همه برگها، که هزینه انجام همه $n^{\log_b a}$ زیر مسئله با اندازه 1 است، برابر $\Theta(n^{\log_b a})$ می‌باشد. ■

بسته به درخت بازگشت، سه حالت قضیه اصلی متناظر است با حالتهایی که در آن‌ها هزینه کل درخت (۱) تحت‌الشعاع هزینه‌های برگها قرار می‌گیرد، (۲) بطور یکنواخت در سر تا سر سطوح درخت توزیع می‌شود، (۳) تحت‌الشعاع هزینه ریشه قرار می‌گیرد.

حاصلجمع در معادله (۴.۶) هزینه گام‌های تقسیم و ترکیب را در الگوریتم تقسیم و حل بنیادی بیان می‌کند. لم بعد حدهای مجانبی را روی رشد حاصلجمع ایجاد می‌کند.

لم ۴.۳

فرض کنید $a \geq 1$ و $b>1$ ثابت باشند، و $f(n)$ را یک تابع غیر منفی تعریف شده روی توان‌های کاملی از b در نظر بگیرید. تابع $g(n)$ روی توانهای کامل b به شکل زیر تعریف می‌شود

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \qquad (۴.۷)$$

آنگاه می‌تواند بطور مجانبی برای توانهای کامل b بصورت زیر محدود شود:

۱. اگر به ازای ثابت $\varepsilon>0$، $f(n) = O(n^{\log_b a-\varepsilon})$، آنگاه $g(n) = O(n^{\log_b a})$.

۲. اگر $f(n) = \Theta(n^{\log_b a})$، آنگاه $g(n) = \Theta(n^{\log_b a} \lg n)$.

۳. اگر به ازای ثابت $c<1$ و برای تمام $n\geq b$، $af(n/b)\leq cf(n)$، آنگاه $g(n) = \Theta(f(n))$.

اثبات برای حالت ۱ داریم $f(n)=O(n^{\log_b a-\varepsilon})$، که دلالت می‌کند بر اینکه $f(n/b^j) = O((n/b^j)^{\log_b a-\varepsilon})$ با جایگذاری در معادله (۴.۷) داریم

$$g(n) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right) \qquad (۴.۸)$$

حاصلجمع داخل نماد O را با فاکتورگیری جملات و ساده سازی، محدود می‌کنیم که به یک سری هندسی صعودی منجر می‌شود:

$$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} = n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j$$
$$= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} (b^\epsilon)^j$$
$$= n^{\log_b a-\epsilon} \left(\frac{b^{\epsilon \log_b n}-1}{b^\epsilon-1}\right)$$
$$= n^{\log_b a-\epsilon} \left(\frac{n^\epsilon-1}{b^\epsilon-1}\right).$$

از آنجا که bو εثابت هستند، می‌توانیم عبارت آخر را بصورت $n^{\log_b a-\varepsilon}O(n^\varepsilon)=O(n^{\log_b a})$ بازنویسی کنیم. با جایگذاری این عبارت برای حاصلجمع در معادله (۴.۸) داریم

$$g(n) = O(n^{\log_b a}),$$

و حالت ۱ ثابت می‌شود.

با این فرض که برای حالت ۲، $f(n) = \Theta(n^{\log_b a})$ داریم

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right). \qquad (۴.۹)$$

حاصلجمع در Θ را مانند حالت ۱ محدود می‌کنیم، اما این بار یک سری هندسی بدست نمی‌آوریم. در عوض، درمی‌یابیم که جمله حاصلجمع یکسان است:

$$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} = n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j$$
$$= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1$$
$$= n^{\log_b a} \log_b n.$$

با جایگذاری این عبارت برای حاصلجمع در معادله (۴.۹) داریم:

$$g(n) = \Theta(n^{\log_b a} \log_b n)$$
$$= \Theta(n^{\log_b a} \lg n),$$

و حالت ۲ اثبات می‌شود.

حالت ۳ بطور مشابه ثابت می‌شود. از آنجا که $f(n)$ در تعریف (۴.۷) تابع $g(n)$ قرار دارد و همه جملات $g(n)$ غیر منفی هستند، می‌توانیم نتیجه بگیریم که برای توانهای کامل b، $g(n) = \Omega(f(n))$. با این فرض که برای ثابت $c<1$ و تمام $n \geq b$، $af(n/b) \leq cf(n)$، داریم $f(n/b) \leq (c/a)f(n)$. با j بار تکرار، داریم $f(n/b^j) \leq (c/a)f(n)$ یا به طور معادل $a^j f(n/b^j) \leq c^j f(n)$ با جایگذاری در معادله (۴.۷) و ساده‌سازی، یک سری هندسی بدست می‌آید اما بر خلاف سری حالت ۱، این سری دارای جملات نزولی است:

$$\begin{aligned} g(n) &= \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\ &\leq \sum_{j=0}^{\log_b n-1} c^j f(n) \\ &\leq f(n)\sum_{j=0}^{\infty} c^j \\ &= f(n)\left(\frac{1}{1-c}\right) \\ &= O(f(n)), \end{aligned}$$

زیرا c ثابت است. بنابراین می‌توانیم نتیجه بگیریم که برای توانهای کاملی از b، $g(n)=\Theta(f(n))$ حالت ۳ اثبات می‌شود، که اثبات لم را کامل می‌کند. ■

اکنون می‌توانیم صورتی از قضیه اصلی را برای حالتی که n توان کاملی از b است ثابت کنیم.

لم ۴.۴

فرض کنید $a \geq 1$ و $b>1$ ثابت باشند و $f(n)$ را یک تابع غیر منفی تعریف شده روی توانهای کاملی از b در نظر بگیرید. $T(n)$ را روی توانهای کاملی از b توسط رابطه بازگشتی زیر تعریف کنید

$$T(n) = \begin{cases} \Theta(1) & \text{اگر } n=1 \\ aT(n/b) + f(n) & \text{اگر } n=b^i \end{cases}$$

که در آن، i یک عدد صحیح مثبت است. آنگاه $T(n)$ می‌تواند بطور مجانبی برای توانهای کاملی از b بصورت زیر محدود شود:

۱. اگر به ازای ثابت $\varepsilon>0$، $f(n) = O(n^{\log_b a-\epsilon})$ آنگاه $T(n) = \Theta(n^{\log_b a})$.

۲. اگر $f(n) = \Theta(n^{\log_b a})$ آنگاه $T(n) = \Theta(n^{\log_b a}\lg n)$.

۳. اگر به ازای ثابت $\varepsilon > ۰$ $f(n) = \Omega(n^{\log_b a+\epsilon})$

و به ازای ثابت $c<1$ و تمام n های به اندازه کافی بزرگ $af(n/b)\leq cf(n)$ ، آنگاه $T(n)=\Theta(f(n))$.

اثبات از حدهای لم ۴.۳ برای ارزشیابی حاصلجمع (۴.۶) از لم ۴.۲ استفاده می‌کنیم.

برای حالت ۱ داریم

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}) , \end{aligned}$$

برای حالت ۲،

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) \\ &= \Theta(n^{\log_b a} \lg n) . \end{aligned}$$

برای حالت ۳،

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(f(n)) \\ &= \Theta(f(n)) , \end{aligned}$$

زیرا

$$f(n) = \Omega(n^{\log_b a+\epsilon}).$$

■

۴.۴.۲ کف‌ها و سقف‌ها

برای تکمیل اثبات قضیه اصلی، اکنون باید تحلیل خود از موقعیتی که در آن کف‌ها و سقف‌ها در رابطه بازگشتی اصلی استفاده می‌شوند را گسترش دهیم تا آنکه رابطه بازگشتی برای همه اعداد صحیح تعریف شود، نه فقط برای توانهای کامل b. فراهم کردن یک حد پایین روی

$$T(n) = aT(\lceil n/b \rceil) + f(n) \tag{۴.۱۰}$$

و یک حد بالا روی

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \tag{۴.۱۱}$$

معمول است، زیرا حد $\lceil n/b \rceil \geq n/b$ می‌تواند در حالت اول گنجانده شود تا نتیجه مطلوب را حاصل کند، و حد $\lfloor n/b \rfloor \leq n/b$ می‌تواند در حالت دوم گنجانده شود. محدود کردن رابطه بازگشتی (۴.۱۱) از پایین بسیار شبیه تکنیک محدود کردن رابطه بازگشتی (۴.۱۰) از بالا است، بنابراین فقط حد دوم را بیان خواهیم کرد.

درخت بازگشت شکل (۴.۳) را تغییر می‌دهیم تا درخت بازگشت شکل (۴.۴) ایجاد شود. همانطور که در درخت بازگشت پایین می‌رویم، به یک توالی از فراخوانی‌های بازگشتی روی آرگومانهای زیر می‌رسیم

n ,
$\lceil n/b \rceil$,
$\lceil \lceil n/b \rceil /b \rceil$,
$\lceil \lceil \lceil n/b \rceil /b \rceil /b \rceil$,

$f(n)$ .... $f(n)$
a
$f(n_1)$ $f(n_1)$ ... $f(n_1)$ .... $af(n_1)$
$\lfloor \log_b n \rfloor$
$f(n_2)$ $f(n_2)$ ... $f(n_2)$ $f(n_2)$ $f(n_2)$ ... $f(n_2)$ $f(n_2)$ $f(n_2)$ ... $f(n_2)$ .... $a^2 f(n_2)$
$\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ ... $\Theta(1)$ $\Theta(1)$ $\Theta(1)$ .... $\Theta(n^{\log_b a})$
$\Theta(n^{\log_b a})$

Total: $\Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor -1} a^j f(n_j)$

شکل ۴.۴ درخت بازگشت تولید شده توسط $aT(\lceil n/b \rceil)+f(n)$. آرگومان بازگشتی n_j توسط معادله (۴.۱۲) بدست می‌آید.

فرض کنید که jامین عنصر توالی را با n_j نشان دهیم، که

$$n_j = \begin{cases} n & j=0 \text{ اگر} \\ \lceil n_{j-1}/b \rceil & j>0 \text{ اگر} \end{cases} \quad (۴.۱۲)$$

اولین هدف ما این است که عمق k را بطوریکه n_k یک ثابت است تعیین کنیم. با استفاده از نامساوی $\lceil x \rceil \le x+1$، داریم

$$\begin{aligned} n_0 &\le n , \\ n_1 &\le \frac{n}{b}+1 , \\ n_2 &\le \frac{n}{b^2}+\frac{1}{b}+1, \\ n_3 &\le \frac{n}{b^3}+\frac{1}{b^2}+\frac{1}{b}+1 , \\ &\vdots \end{aligned}$$

بطور کلی

$$\begin{aligned} n_j &\le \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} \\ &< \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} \\ &= \frac{n}{b^j} + \frac{b}{b-1} . \end{aligned}$$

با قرار دادن $j = \lfloor log_b^n \rfloor$، بدست می‌آوریم

$$\begin{aligned} n_{\lfloor \log_b n \rfloor} &< \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1} \\ &\le \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} \\ &= \frac{n}{n/b} + \frac{b}{b-1} \\ &= b + \frac{b}{b-1} \\ &= O(1) , \end{aligned}$$

و بنابراین می‌بینیم که در عمق $\lfloor log_b^n \rfloor$، اندازه مسئله حداکثر یک ثابت است.

از شکل (۴.۴)، مشاهده می‌کنیم که

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) , \qquad (۴.۱۳)$$

که بسیار شبیه معادله (۴.۶) است، بجز اینکه n یک عدد صحیح دلخواه است و به توان کاملی از b محدود نمی‌شود.

اکنون می‌توانیم حاصلجمع

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \qquad (۴.۱۴)$$

از (۴.۱۳) به روشی مشابه اثبات لم ۴.۳ ارزشیابی کنیم. با آغاز از حالت ۳، اگر برای $n > b + b/(b-1)$ داشته باشیم $af(\lceil n/b \rceil) \le cf(n)$ که $c<1$ یک ثابت است، آنگاه ثابت می‌شود که $a^j f(n_j) \le c^j f(n)$. بنابراین، حاصلجمع در معادله (۴.۱۴) می‌تواند درست مانند لم ۴.۳ ارزشیابی شود. برای حالت ۲ داریم

$f(n)=\Theta(n^{\log_b a})$. اگر بتوانیم نشان دهیم که

$$f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a}),$$

آنگاه اثبات برای حالت ۲ از لم ۴.۳، انجام خواهد شد. دقت کنید که $j \le \lfloor \log_b n \rfloor$، بر $b^j/n \le 1$ دلالت می‌کند. حد $f(n)=O(n^{\log_b a})$ دلالت می‌کند بر اینکه یک ثابت $c>0$ وجود دارد بطوریکه برای همه n_jهای به اندازه کافی بزرگ،

$$\begin{aligned}
f(n_j) &\le c\left(\frac{n}{b^j}+\frac{b}{b-1}\right)^{\log_b a} \\
&= c\left(\frac{n}{b^j}\left(1+\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a} \\
&= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\left(\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a} \\
&\le c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\frac{b}{b-1}\right)^{\log_b a} \\
&= O\left(\frac{n^{\log_b a}}{a^j}\right),
\end{aligned}$$

زیرا $c(1+b/(b-1))^{\log_b a}$ یک ثابت است. بنابراین حالت ۲ اثبات می‌شود. اثبات حالت ۱ تقریباً مشابه است. کلید اثبات این است که حد $f(n_j)=O(n^{\log_b a-\varepsilon})$ اثبات شود، که شبیه اثبات متناظرش در حالت ۲ است، گرچه محاسبات جبری پیچیده‌تر است.

اینک حدهای بالا را در قضیه اصلی برای همه اعداد صحیح n اثبات کرده‌ایم. اثبات حدهای پایین نیز مشابه است.

تمرین‌ها

۴.۴-۱* یک عبارت کامل و ساده برای n_j در معادله (۴.۱۲) برای حالتی که در آن b، به جای یک عدد حقیقی دلخواه، یک عدد صحیح مثبت است، ارائه دهید.

۴.۴-۲* نشان دهید که اگر $f(n) = \Theta(n^{\log_b a} \lg^k n)$، که در آن $k \ge 0$، آنگاه رابطه بازگشتی اصلی دارای جواب $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$ است. به منظور سهولت، تحلیل خود را به توانهای کامل b محدود کنید.

۴.۴-۳* نشان دهید که حالت ۳ قضیه اصلی بصورت اغراق‌آمیزی بیان شده است، به این معنی که برای

ثابت $c<1$، شرط نظم $af(n/b) \leq cf(n)$ دلالت می‌کند بر اینکه یک ثابت $\varepsilon>0$ وجود دارد، بطوریکه

$$f(n) = \Omega(n^{\log_b a+\epsilon}).$$

مسائل

۴-۱ مثالهایی از رابطه‌های بازگشتی

حدهای پایین و بالای مجانبی را برای $T(n)$ در هر کدام از رابطه‌های بازگشتی زیر ارائه دهید. فرض کنید برای $n \leq 2$ $T(n)$ ثابت است. حدود خود را به اندازه ممکن قوی سازید و پاسخهایتان را توجیه کنید.

a. $T(n) = 2T(n/2) + n^3$.

b. $T(n) = T(9n/10) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 7T(n/3) + n^2$.

e. $T(n) = 7T(n/2) + n^2$.

f. $T(n) = 2T(n/4) + \sqrt{n}$.

g. $T(n) = T(n-1) + n$.

h. $T(n) = T(\sqrt{n}) + 1$.

۴-۲ یافتن اعداد صحیح گم شده

آرایه $A[1\ .\ .\ n]$ شامل همه اعداد صحیح از 0 تا n بجز یک عدد می‌باشد. تعیین عدد صحیح گمشده در زمان $O(n)$ با استفاده از یک آرایه کمکی $B[0\ .\ .\ n]$ که اعداد موجود در A را ثبت می‌کند، آسان خواهد بود. امّا در این مسئله نمی‌توانیم به یک عدد صحیح در A با یک عمل دسترسی پیدا کنیم. عناصر A بصورت دودویی نمایش داده می‌شوند و تنها عملی که می‌توان برای دستیابی به آنها استفاده کرد "برداشتن jامین بیت $A[i]$" است که زمان ثابتی را صرف می‌کند.

نشان دهید که اگر فقط از این عمل استفاده کنیم، می‌توانیم باز هم عدد صحیح گمشده را در زمان $O(n)$ تعیین کنیم.

۴-۳ هزینه‌های ارسال پارامتر

در طول این کتاب، فرض می‌کنیم که ارسال پارامتر طی فراخوانی‌های روال، زمان ثابتی را صرف

می‌کند، حتی اگر یک آرایه n عنصری ارسال شود. این فرض در اکثر سیستم‌ها معتبر است چون یک اشاره‌گر به آرایه، و نه خود آرایه، فرستاده می‌شود. این مسئله سه استراتژی ارسال پارامتر را بررسی می‌کند:

۱. یک آرایه بوسیله اشاره‌گر ارسال می‌شود. زمان = $\Theta(1)$.

۲. یک آرایه بوسیله کپی کردن ارسال می‌شود. زمان = $\Theta(N)$، که در آن N اندازه آرایه است.

۳. یک آرایه بوسیله کپی کردن تنها زیر بازه‌ای که ممکن است توسط روال فراخوانی شده مورد دسترسی قرار گیرد، ارسال می‌شود. اگر زیر آرایه $A[p..q]$ ارسال شود، زمان = $\Theta(q-p+1)$.

a. الگوریتم جستجوی دودویی بازگشتی را برای یافتن یک عدد در آرایه مرتب شده در نظر بگیرید (تمرین ۵-۲.۳ را ملاحظه نمایید). رابطه‌های بازگشتی برای زمان‌های اجرای جستجوی دودویی در بدترین حالت، هنگامی که آرایه‌ها با استفاده از هر کدام از سه روش فوق ارسال می‌شوند، ارائه دهید و حدهای بالای مناسبی روی جواب‌های رابطه‌های بازگشتی ارائه دهید. N را اندازه مسئله اصلی و n را روی اندازه یک زیر مسئله در نظر بگیرید.

b. قسمت (a) را برای الگوریتم *MERGE-SORT* از بخش ۲.۳.۱ مجدداً انجام دهید.

۴-۴ مثالهای بیشتری از رابطه‌های بازگشتی

حدهای پایین و بالا مجانبی را برای $T(n)$ در هر کدام از رابطه‌های بازگشتی زیر ارائه دهید. فرض کنید $T(n)$ برای n به اندازه کافی کوچک ثابت است. حدود خود را به اندازه ممکن قوی سازید و پاسخهایتان را توجیه کنید.

a. $T(n) = 3T(n/2) + n \lg n.$

b. $T(n) = 5T(n/5) + n/\lg n.$

c. $T(n) = 4T(n/2) + n^2\sqrt{n}.$

d. $T(n) = 3T(n/3 + 5) + n/2.$

e. $T(n) = 2T(n/2) + n/\lg n.$

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n.$

g. $T(n) = T(n-1) + 1/n.$

h. $T(n) = T(n-1) + \lg n.$

i. $T(n) = T(n-2) + 2\lg n.$

j. $T(n) = \sqrt{n}T(\sqrt{n}) + n.$

۴-۵ اعداد فیبوناچی

این مسئله ویژگی‌های اعداد فیبوناچی را، که توسط رابطه بازگشتی (۳.۲۱) تعریف می‌شوند، توسعه می‌دهد. از تکنیک تولید توابع استفاده خواهیم کرد تا رابطه بازگشتی فیبوناچی را حل کنیم. **تابع مولد** (یا **سری‌های توانی رسمی**) $\mathcal{F}$ را بصورت زیر تعریف می‌کنیم

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots ,\end{aligned}$$

که در آن F_i iامین عدد فیبوناچی است.

a.نشان دهید $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

b.نشان دهید

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{(1-\phi z)(1-\widehat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\widehat{\phi} z}\right) ,\end{aligned}$$

که در آن

$$\phi = \frac{1+\sqrt{5}}{2} = 1.61803\ldots$$

و

$$\widehat{\phi} = \frac{1-\sqrt{5}}{2} = -0.61803\ldots .$$

c.نشان دهید

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \widehat{\phi}^i)z^i .$$

d.ثابت کنید برای $i > 0$، $F_i = \phi^i/\sqrt{5}$ به نزدیکترین عدد صحیح گرد می‌شود. (راهنمایی: دقت کنید که $|\widehat{\phi}| < 1$).

e.ثابت کنید برای $i \geq 0$، $F_{i+2} \geq \phi^i$.

۴-۶ تست تراشه *VLSI*

پروفسور *Diogenes* n تراشه *VLSI*[1] فرضاً یکسان دارد که قادر به تست یکدیگر هستند. دستگاه تست پروفسور در یک زمان دو تراشه را در خود جای می‌دهد. هنگامیکه دستگاه بارگذاری می‌شود، هر تراشه، تراشه دیگر را تست می‌کند و گزارش می‌دهد که آیا تراشه خوب است یا بد. یک تراشه خوب همیشه گزارش دقیقی راجع به اینکه آیا تراشه دیگر خوب است یا بد ارائه می‌دهد. اما پاسخ یک تراشه بد نمی‌تواند مورد اطمینان باشد. بنابراین چهار پیشامد ممکن در یک تست بصورت زیر است:

تراشه A گزارش می‌دهد	تراشه B گزارش می‌دهد	نتیجه
B خوب است	A خوب است	هر دو خوبند یا هر دو بدند
B خوب است	A بد است	حدّاقل یکی بد است
B بد است	A خوب است	حدّاقل یکی بد است
B بد است	A بد است	حدّاقل یکی بد است

a. نشان دهید که اگر بیشتر از $n/2$ تراشه بد باشند، پروفسور نمی‌تواند لزوماً با استفاده از هر استراتژی مبتنی بر این نوع تست جفت به جفت، تعیین کند که کدام یک از تراشه‌ها خوب هستند. فرض کنید تراشه‌های بد می‌توانند برای گمراه کردن پروفسور همکاری کنند.

b. مسئله یافتن یک تراشه خوب از بین n تراشه را در نظر بگیرید، فرض کنید که بیشتر از $n/2$ تراشه خوب هستند. نشان دهید $\lfloor n/2 \rfloor$ تست جفت به جفت برای تبدیل مسئله به یک مسئله با اندازه تقریباً نصف کافی است.

c. نشان دهید که تراشه‌های خوب می‌توانند با $\Theta(n)$ تست جفت به جفت مشخص شوند، با فرض اینکه بیشتر از تراشه $n/2$ خوب هستند. یک رابطه بازگشتی که تعداد تستها را بیان می‌کند ارائه کرده و آن را حل نمایید.

۴-۷ آرایه‌های *Monge*

آرایه A با ابعاد $m \times n$ از اعداد حقیقی یک **آرایه** *Monge* است اگر برای همه i، j، k و l بطوریکه $1 \le i < k \le m$ و $1 \le j < l \le n$ داشته باشیم

$$A[i , j] + A[k , l] \le A[i , l] + A[k , j]$$

---

۱ - *VLSI* مخفف "*Very Large Scale Integration*" به معنی "مجتمع سازی در مقیاس خیلی بزرگ" می‌باشد که تکنولوژی تراشه مدار مجتمع است و امروزه برای ساخت اکثر ریز پردازنده‌ها استفاده می‌شود.

بعبارت دیگر، زمانیکه دو سطر و دو ستون از یک آرایه *Monge* را انتخاب کنیم و چهار عنصر واقع در محل برخورد سطرها و ستون‌ها را در نظر بگیریم، مجموع عناصر بالا-چپ و پایین -راست، کوچکتر یا مساوی مجموع عناصر بالا -راست و پایین -چپ است. بعنوان مثال، آرایه زیر *Monge* است:

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

a. ثابت کنید یک آرایه *Monge* است اگر و فقط اگر برای همه $i = 1, 2, \ldots, m-1$ و $j = 1, 2, n-1$ داشته باشیم

$$A[i,j]+A[i+1,j+1] \leq A[i,j+1] + A[i+1,j]$$

(راهنمایی: برای قسمت "فقط اگر"، بطور مجزا از استقرا روی سطرها و ستونها استفاده کنید.)

b. آرایه زیر *Monge* نیست. یک عنصر را جهت *Monge* ساختن آن تغییر دهید.

(راهنمایی: از قسمت *(a)* استفاده کنید.)

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

c. فرض کنید $f(i)$ اندیس ستونی که شامل سمت چپ‌ترین عنصر کوچک سطر i است، باشد. ثابت کنید که برای هر آرایه *Monge* با ابعاد $m \times n$ داریم

$$f(1) \leq f(2) \leq \ldots \leq f(m)$$

d. در اینجا توصیفی از یک الگوریتم تقسیم و حل وجود دارد که سمت چپ‌ترین عنصر کوچک در هر سطر از یک آرایه *Monge* به نام A با ابعاد $m \times n$ را محاسبه می‌کند:

یک زیر ماتریس A' از A شامل سطرهای با شماره زوج A بسازید. بطور بازگشتی سمت چپ‌ترین مینیمم را برای هر سطر از A' تعیین کنید. آنگاه سمت چپ‌ترین مینیمم در سطرهای با شماره فرد A را محاسبه نمایید.

توضیح دهید چگونه سمت چپ‌ترین مینیمم در سطرهای با شماره فرد A (سمت چپ‌ترین مینیمم در سطرهای با شماره زوج مشخص شده است) در زمان $O(m+n)$ محاسبه می‌شود.

e. یک رابطه بازگشتی بنویسید که زمان اجرای الگوریتم توصیف شده در قسمت (d) را بیان کند. نشان دهید که جواب آن $O(m+n\log m)$ است.

۵ تحلیل احتمالی و الگوریتم‌های تصادفی

این فصل تحلیل احتمالی و الگوریتم‌های تصادفی را معرفی می‌کند. در طول این کتاب به دفعات با تحلیل احتمالی و الگوریتم‌های تصادفی مواجه خواهیم شد.

5.1 مسئله استخدام

فرض کنید نیاز به استخدام یک معاون اداری جدید دارید. تلاش‌های قبلی شما در استخدام ناموفق بوده‌اند و تصمیم می‌گیرید از یک آژانس کاریابی استفاده کنید. آژانس کاریابی هر روز یک کاندیدا برای شما خواهد فرستاد. شما با این شخص مصاحبه خواهید کرد و سپس تصمیم می‌گیرید که آن شخص را استخدام کنید یا خیر. شما باید حق‌الزحمه کمی به آژانس کاریابی برای مصاحبه با یک متقاضی بپردازید. اما استخدام واقعی یک متقاضی بسیار پرهزینه است، زیرا باید معاون اداری فعلی خود را اخراج کرده و حق‌الزحمه استخدام زیادی به آژانس کاریابی پرداخت کنید. شما متعهد می‌شوید تا همواره بهترین شخص ممکن برای این شغل را داشته باشید. بنابراین تصمیم می‌گیرید که پس از مصاحبه با هر متقاضی، اگر این متقاضی نسبت به معاون اداری فعلی شایسته‌تر باشد، معاون اداری فعلی را اخراج کرده و متقاضی جدید را استخدام کنید. شما مایل به پرداخت هزینه حاصل از این استراتژی هستید، اما می‌خواهید تخمین بزنید که این هزینه چقدر خواهد بود.

روال *HIRE-ASSISTANT* ارائه شده در زیر، این استراتژی استخدام را بصورت شبه کد بیان می‌کند. فرض می‌شود که کاندیداهای شغل معاون اداری از 1 تا n شماره گذاری شده‌اند. روال فرض می‌کند که شما قادر هستید تا، پس از مصاحبه با کاندیدای i تعیین کنید که آیا کاندیدای i بهترین کاندیدایی است که تا کنون دیده‌اید یا خیر. برای مقدار دهی اولیه، روال یک کاندیدای فرضی با شماره 0 ایجاد می‌کند که نسبت به تمام کاندیداهای دیگر، شایستگی کمتری دارد.

```
HIRE-ASSISTANT(n)
1  best ← 0          ▷ candidate 0 is a least-qualified dummy candidate
2  for i ← 1 to n
3       do interview candidate i
4          if candidate i is better than candidate best
5             then best ← i
6                  hire candidate i
```

مدل هزینه برای این مسئله با مدل توصیف شده در فصل ۲ تفاوت دارد. به زمان اجرای *HIRE-ASSISTANT* نمی‌پردازیم، اما در عوض هزینه‌ای که برای مصاحبه و استخدام متحمّل می‌شویم را مورد بررسی قرار می‌دهیم. در ظاهر، تحلیل هزینه این الگوریتم ممکن است بسیار متفاوت با تحلیل زمان اجرای عملی باشد که مرتب سازی ادغام می‌نامیم اما تکنیک‌های تحلیل استفاده شده، چه در حال تحلیل هزینه و چه در حال تحلیل زمان اجرا باشیم، یکسان هستند. در هر دو حالت، تعداد دفعاتی که اعمال اصلی مشخص اجرا می‌شوند را محاسبه می‌کنیم.

مصاحبه دارای هزینه کمی است، که c_i می‌نامیم، در صورتیکه استـ[illegible] c_h است. فرض کنید m تعداد افراد استخدام شده باشد، پس کل هزینه مربوط به ایـ[illegible] $O(nc_i+mc_h)$ است. صرف نظر از تعداد افرادی که استخدام می‌کنیم، همواره با n کاندیدا مصاحبه می‌کنیم و بنابراین همیشه هزینه nc_i مربوط به مصاحبه را متحمل می‌شویم. لذا به تحلیل mc_h می‌پردازیم. این کمیت با هر بار اجرای الگوریتم تغییر می‌کند.

این مطالب بعنوان یک الگوی محاسباتی عادی کاربرد دارند. اغلب این گونه است که برای پیدا کردن ماکزیمم یا مینیمم مقدار در یک توالی نیاز به بررسی هر عنصر از توالی و نگهداری یک "برنده" فعلی داریم. مسئله استخدام این موضوع را [illegible]کند که چند بار برداشت خود از اینکه کدام عنصر در حال حاضر برنده است را [illegible] روز رسانی می‌کنیم.

تحلیل بدترین حالت

در بدترین حالت، هر کاندیدایی را که با وی مصاحبه می‌کنیم واقعاً استخدام می‌کنیم. این وضعیت هنگامی رخ می‌دهد که کاندیداها در یک ترتیب صعودی از شایستگی وارد شوند، که در این حالت n بار استخدام را انجام می‌دهیم و هزینه استخدام کل برابر $O(nc_h)$ است.

اما منطقی است انتظار داشته باشیم که کاندیداها همواره در یک ترتیب صعودی از شایستگی قرار ندارند. در حقیقت هیچ ایده‌ای در مورد ترتیبی که آنها برای استخدام می‌آیند نداشته و نیز هیچ کنترلی روی این ترتیب نداریم. بنابراین طبیعی است این سؤال را بپرسیم که در یک حالت میانگین یا یک حالت معمول، انتظار وقوع چه اتفاقی را داریم.

تحلیل احتمالی

تحلیل احتمالی، استفاده از احتمال در تحلیل مسائل است. معمولاً از تحلیل احتمالی برای تحلیل زمان اجرای یک الگوریتم استفاده می‌کنیم. گاهی اوقات از تحلیل احتمالی برای تحلیل کمیتهای دیگر مانند هزینه استخدام در روال *HIRE-ASSISTANT* استفاده می‌کنیم. به منظور انجام یک تحلیل احتمالی باید از دانش خود در مورد توزیع ورودیها، یا انجام فرض هایی در مورد توزیع ورودیها استفاده کنیم. سپس الگوریتم خود را با محاسبه یک زمان اجرای مورد انتظار تحلیل می‌کنیم. انتظار در مورد توزیع ورودیهای ممکن است. بنابراین عملاً زمان اجرا روی تمام ورودیهای ممکن را میانگین می‌گیریم.

در تصمیم‌گیری روی توزیع ورودیها باید بسیار دقیق باشیم. برای بعضی از مسائل، مناسب است فرض‌هایی در مورد مجموعه تمام ورودیهای ممکن انجام دهیم و می‌توانیم از تحلیل احتمالی بعنوان تکنیکی برای طرّاحی یک الگوریتم کارآمد و بعنوان وسیله‌ای جهت کسب اطلاعات راجع به یک مسئله استفاده کنیم. برای مسائل دیگر نمی‌توانیم یک توزیع ورودی مناسب را توصیف کنیم و در این موارد نمی‌توانیم از تحلیل احتمالی استفاده کنیم.

برای مسئله استخدام می‌توانیم فرض کنیم که متقاضی‌ها با یک ترتیب تصادفی می‌آیند. این مطلب برای این مسئله به چه معنا است؟ فرض می‌کنیم که می‌توانیم هر کاندیدا را با هر کاندیدای دیگر مقایسه کرده و تصمیم بگیریم که کدام یک شایسته‌تر است. بنابراین با استفاده از *rank(i)* برای نمایش رتبه متقاضی i و پذیرفتن این قرارداد که یک رتبه بالاتر متناظر با یک متقاضی شایسته‌تر است، می‌توانیم هر کاندیدا را با یک عدد منحصر از *1* تا n رتبه بندی کنیم. لیست مرتب *<rank(1), rank(2),. . ..rank(n)>* یک جایگشت از لیست *<1, 2,. . . , n>* است. این مطلب که متقاضیان با یک ترتیب تصادفی می‌آیند معادل است با این مطلب که احتمال برابری دارد این لیست رتبه‌ها هر یک از $n!$ جایگشت اعداد *1* تا n باشد. متناوباً می‌گوییم رتبه‌ها یک **جایگشت تصادفی یکنواخت** را تشکیل می‌دهند؛ به عبارت دیگر هر کدام از $n!$ جایگشت ممکن با احتمال یکسانی رخ می‌دهند.

بخش ۵.۲ شامل یک تحلیل احتمالی از مسئله استخدام است.

الگوریتم‌های تصادفی

به منظور استفاده از تحلیل احتمالی، نیاز است مطالبی در مورد توزیع روی ورودیها بدانیم. بیشتر موارد در مورد توزیع ورودی، آگاهی مختصری داریم. حتی اگر مطالبی در مورد توزیع بدانیم قادر نخواهیم بود که این دانش را بصورت محاسباتی مدل کنیم. با این وجود اغلب می‌توانیم از احتمال و تصادفی‌سازی بعنوان ابزاری برای طراحی و تحلیل الگوریتم استفاده کنیم، بدین شکل که رفتار قسمتی از الگوریتم را تصادفی می‌کنیم.

در مسئله استخدام، ممکن است چنین به نظر برسد که کاندیداها در یک ترتیب تصادفی به ما نشان داده می‌شوند، اما هیچ راهی برای اطلاع از اینکه آیا آنها واقعاً اینگونه هستند یا خیر نداریم. بنابراین به منظور توسعه یک الگوریتم تصادفی برای مسئله استخدام باید کنترل بیشتری روی ترتیبی که با کاندیداها مصاحبه می‌کنیم داشته باشیم. لذا مدل را اندکی تغییر خواهیم داد. خواهیم گفت که آژانس کاریابی دارای n کاندیدا است، و لیستی از کاندیداها را از قبل برای ما می‌فرستند. هر روز به طور تصادفی انتخاب می‌کنیم که با کدام کاندیدا، مصاحبه نماییم. اگر چه هیچ اطلاعی در مورد کاندیداها (به غیر از نام آنها) نداریم، اما تغییر مهمی را ایجاد کرده‌ایم. به جای تکیه بر این حدس که کاندیداها در یک ترتیب تصادفی نزد ما خواهند آمد، کنترل فرآیند را بدست گرفته و یک ترتیب تصادفی را اعمال کرده‌ایم.

بطور کلی‌تر، یک الگوریتم را **تصادفی** می‌نامیم اگر رفتارش نه تنها با ورودی‌اش بلکه همچنین با مقادیر تولید شده توسّط یک **مولد عدد تصادفی** تعیین شود. فرض خواهیم کرد که مولد عدد تصادفی *RANDOM* را در اختیار داریم. فراخوانی *RANDOM(a,b)* یک عدد صحیح بین a و b را برمی‌گرداند که با احتساب a و b احتمال هر چنین عدد صحیحی یکسان است. برای مثال، *RANDOM(0,1)*، 0 را با احتمال $1/2$ و 1 را با احتمال $1/2$ تولید می‌کند. فراخوانی *RANDOM(3,7)* یکی از اعداد 3، 4، 5، 6 یا 7 را با احتمال $1/5$ بر می‌گرداند. هر عدد صحیح که بوسیله *RANDOM* بر گردانده می‌شود، مستقل از اعداد صحیح برگردانده شده در فراخوانی‌های قبلی است. می‌توانید *RANDOM* را مانند چرخاندن یک $(b-a+1)$-وجهی برای بدست آوردن خروجی‌اش تصور کنید. (در عمل، اکثر محیط‌های برنامه‌نویسی یک **مولد عدد شبه تصادفی** ارائه می‌کنند: یک الگوریتم مشخص که اعدادی را بر می‌گرداند که از لحاظ آماری تصادفی "به نظر می‌رسند".)

تمرین‌ها

۵.۱-۱ نشان دهید این فرض که همواره در خط ۴ از روال *HIRE-ASSISTANT* قادر به تعیین بهترین کاندیدا هستیم، دلالت می‌کند بر اینکه می‌توانیم هر رتبه مربوط به یک کاندیدا را با هر رتبه مربوط به یک کاندیدای دیگر مقایسه کنیم.

۵.۱-۲* یک پیاده سازی از روال *RANDOM(a,b)* بیان کنید که تنها فراخوانی‌های *RANDOM(0, 1)* را انجام دهد. زمان اجرای مورد انتظار روال شما بصورت تابعی از a و b چیست؟

۵.۱-۳* فرض کنید می‌خواهید 0 را با احتمال $1/2$ و 1 را با احتمال $1/2$ در خروجی داشته باشید. روال *BIASED-RANDOM* در اختیار شما قرار دارد که یا 0 یا 1 را در خروجی می‌دهد. این روال، 1 را با احتمال p و 0 را با احتمال $1-p$ می‌دهد که $0<p<1$ ، اما نمی‌دانید که p چه مقدار است. الگوریتمی ارائه کنید که از *BIASED-RANDOM* بعنوان یک زیر روال استفاده کند و با برگرداندن 0 با احتمال $1/2$ و 1

با احتمال $1/2$، یک پاسخ بدون انحراف را برگرداند. زمان اجرای مورد انتظار الگوریتم شما بصورت تابعی از p چیست؟

۵.۲ متغیرهای تصادفی شاخص

به منظور تحلیل بیشتر الگوریتم‌ها از جمله مسئله استخدام، از متغیرهای تصادفی شاخص استفاده خواهیم کرد. متغیرهای تصادفی شاخص، یک روش مناسب برای تبدیل بین احتمالها و انتظارها را فراهم می‌کنند. فرض کنید فضای نمونه S و پیشامد A را داریم. **متغیر تصادفی شاخص** $I\{A\}$ مربوط به پیشامد A بصورت زیر تعریف می‌شود:

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{اگر } A \text{ رخ دهد} \\ 0 & \text{اگر } A \text{ رخ ندهد} \end{cases} \qquad (۵.۱)$$

بعنوان یک مثال ساده، اجازه دهید تعداد مورد انتظار شیرهایی که در پرتاب یک سکه نا اریب بدست می‌آیند را تعیین کنیم. فضای نمونه ما $S = \{H, T\}$ است، متغیر تصادفی Y را تعریف می‌کنیم که دارای مقادیر H و T، هر یک با احتمال $1/2$ است. سپس می‌توانیم یک متغیر تصادفی شاخص X_H تعریف کنیم که مربوط به سکه‌ای است که شیر می‌آید و می‌توانیم بصورت پیشامد $Y = H$ بیان کنیم. این متغیر تعداد شیرهای بدست آمده در این پرتاب را می‌شمارد. اگر سکّه شیر بیاید X_H برابر 1 و در غیر صورت برابر 0 است. می‌نویسیم

$$X_H = \mathrm{I}\{Y = H\} = \begin{cases} 1 & \text{اگر } Y=H \\ 0 & \text{اگر } Y=T \end{cases}$$

تعداد مورد انتظار شیرهای بدست آمده، در یک پرتاب سکّه بسادگی برابر مقدار مورد انتظار متغیر شاخص X_H است:

$$\begin{aligned} \mathrm{E}[X_H] &= \mathrm{E}[\mathrm{I}\{Y = H\}] \\ &= 1 \cdot \Pr\{Y = H\} + 0 \cdot \Pr\{Y = T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 . \end{aligned}$$

بنابراین تعداد مورد انتظار شیرهای بدست آمده با یک پرتاب سکه نا اریب برابر $1/2$ است. همانطور که لم زیر نشان می‌دهد، تعداد مورد انتظار یک متغیر تصادفی شاخص مربوط به پیشامد A برابر با احتمالی است که A رخ می‌دهد.

لم ۵.۱

S فضای نمونه و A پیشامدی در فضای نمونه S است. قرار دهید $X_A = I\{A\}$ آنگاه $E[X_A] = Pr\{A\}$

اثبات بنا به تعریف معتبر تصادفی شاخص از معادله (۵.۱) و تعریف مقدار مورد انتظار، داریم

$$\begin{aligned} \mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\overline{A}\} \\ &= \Pr\{A\} \ , \end{aligned}$$

که $\overline{A}$ نشان دهنده S-A، یعنی متمم A است. ■

اگر چه استفاده از متغیرهای تصادفی شاخص برای کاربردی مانند شمردن تعداد شیرهایی مورد انتظار در یک پرتاب سکّه واحد ممکن است عملی افراط‌آمیز به نظر برسد، امّا برای تحلیل وضعیت‌هایی که در آنها آزمایشهای تصادفی تکراری را انجام می‌دهیم مفید می‌باشند. برای روشن‌تر کردن این مطلب، می‌توانیم X_i را متغیر تصادفی شاخص مربوط به پیشامدی در نظر بگیریم که در آن، i امین پرتاب شیر می‌آید. Y_i را متغیر تصادفی که نتیجه i امین پرتاب را نشان می‌دهد در نظر می‌گیریم، داریم $X_i = I\{Y_i = H\}$. فرض می‌کنیم X متغیر تصادفی باشد که تعداد کل شیرها در n پرتاب سکه را نشان می‌دهد، لذا

$$X = \sum_{i=1}^{n} X_i \ .$$

می‌خواهیم تعداد مورد انتظار شیرها را محاسبه کنیم، لذا از هر دو طرف معادله بالا انتظار (امید ریاضی) می‌گیریم، داریم

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \ .$$

سمت چپ معادله فوق، انتظار مجموع n متغیر تصادفی است. بنا به لم (۵.۱)، بسادگی می‌توانیم انتظار هر متغیر تصادفی را محاسبه کنیم. بنا به خطی بودن انتظار، محاسبه انتظار مجموع آسان است: انتظار مجموع مساوی با مجموع انتظارهای n متغیر تصادفی است. خطی بودن انتظار، استفاده از متغیرهای تصادفی شاخص را به تکنیک تحلیلی قدرتمندی تبدیل می‌کند؛ حتی زمانیکه بین متغیرهای تصادفی وابستگی وجود دارد خطی بودن انتظار بر قرار است. اکنون بسادگی می‌توانیم تعداد مورد انتظار شیرها را محاسبه کنیم:

$$
\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} 1/2 \\
&= n/2 .
\end{aligned}
$$

در سرتاسر این کتاب از متغیرهای تصادفی شاخص استفاده خواهیم کرد.

تحلیل مسئله استخدام با استفاده از متغیرهای تصادفی شاخص

به مسئله استخدام بر می‌گردیم. اکنون می‌خواهیم تعداد دفعات مورد انتظار که یک معاون اداری جدید را استخدام می‌کنیم محاسبه نماییم. برای استفاده از یک تحلیل احتمالی، فرض می‌کنیم که کاندیداها با یک ترتیب تصادفی، همانطور که در بخش قبلی بحث شد، وارد می‌شوند. (در بخش ۵.۳ چگونگی حذف این فرض را خواهیم دید.) فرض کنید X متغیر تصادفی باشد که مقدارش برابر تعداد دفعاتی است که یک معاون اداری جدید را استخدام می‌کنیم. بنا به تعریف، مقدار مورد انتظار متغیر تصادفی X برابر است با

$$
\mathrm{E}[X] = \sum_{x=1}^{n} x \, \Pr\{X = x\} ,
$$

اما محاسبه این عبارت دشوار است. در عوض از متغیرهای تصادفی شاخص استفاده می‌کنیم تا محاسبات را بسیار ساده کنیم.

برای استفاده از متغیرهای تصادفی شاخص، بجای محاسبه $E[X]$ با تعریف یک متغیر مربوط به تعداد دفعاتی که یک دستیار اداری جدید را استخدام می‌کنیم، n متغیر مربوط به اینکه آیا هر کاندیدای خاص استخدام می‌شود یا خیر تعریف می‌کنیم. بخصوص فرض می‌کنیم X_i متغیر تصادفی شاخص مربوط به پیشامدی باشد که در آن i امین کاندیدا استخدام می‌شود. بنابراین،

$$
X_i = I\{\text{کاندیدای } i \text{ استخدام شود}\} = \begin{cases} 1 & \text{اگر کاندیدای } i \text{ استخدام شود} \\ 0 & \text{اگر کاندیدای } i \text{ استخدام نشود} \end{cases} \qquad (۵.۲)
$$

و

$$
X = X_1 + X_2 + \cdots + X_n . \qquad (۵.۳)
$$

بنا به لم ۵.۱ داریم

$E[X_i] = Pr$ {کاندیدای i انتخاب شود}

و بنابراین باید احتمال اینکه خطهای ۶ - ۵ از *HIRE-ASSISTANT* اجرا می‌شوند را محاسبه کنیم.

کاندیدای i در خط ۵، دقیقاً وقتی بهتر از هر کدام از کاندیداهای 1 تا $i-1$ است استخدام می‌شود. چون فرض کرده‌ایم که کاندیداها با یک ترتیب تصادفی وارد می‌شوند، i کاندیدای اول در یک ترتیب تصادفی قرار گرفته‌اند. تا اینجا احتمال شایسته‌ترین بودن هر یک از این i کاندیدای اول برابر است. احتمال شایسته‌تر بودن کاندیدای i نسبت به کاندیداهای 1 تا $i-1$ برابر $1/i$ است و بنابراین احتمال استخدام شدن آن $1/i$ است. بنا به لم ۵.۱، نتیجه می‌گیریم که

$$E[X_i] = 1/i \tag{۵.۴}$$

اکنون می‌توانیم $E[X]$ را محاسبه کنیم:

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \qquad \text{(بنا به معادله (۵.۳))} \tag{۵.۵}$$

$$= \sum_{i=1}^{n} \mathrm{E}[X_i] \qquad \text{(بنا به خطی بودن انتظار)}$$

$$= \sum_{i=1}^{n} 1/i \qquad \text{(بنا به معادله (۵.۴))}$$

$$= \ln n + O(1) \tag{۵.۶}$$

حتی گرچه با n فرد مصاحبه می‌کنیم، اما تنها واقعاً به طور تقریبی $\ln n$ نفر از آنها را بصورت میانگین استخدام می‌کنیم. این نتیجه را در لم زیر خلاصه می‌کنیم.

لم ۵.۲

با فرض اینکه کاندیداها در یک ترتیب تصادفی قرار می‌گیرند، الگوریتم *HIRE-ASSISTANT* دارای هزینه کلی استخدام برابر $O(c_h \ln n)$ است.

اثبات این حد، مستقیماً از تعریف هزینه استخدام و معادله (۵.۶) اثبات می‌شود. ■

هزینه مصاحبه مورد انتظار، بهبود مهمی در هزینه استخدام $O(nc_h)$ در بدترین حالت است.

تمرین‌ها

۱-۵.۲ در *HIRE-ASSISTANT* با فرض اینکه کاندیداها در یک ترتیب تصادفی ارائه شده‌اند، احتمال اینکه دقیقاً یک بار استخدام را انجام دهید چیست؟ احتمال اینکه دقیقاً n بار استخدام را انجام دهید چیست؟

۲-۵.۲ در *HIRE-ASSISTANT* با فرض اینکه کاندیداها در یک ترتیب تصادفی ارائه شده‌اند، احتمال اینکه دقیقاً دوبار استخدام را انجام دهید چیست؟

۳-۵.۲ با استفاده از متغیرهای تصادفی شاخص، مقدار مورد انتظار جمع n تاس را محاسبه کنید.

۴-۵.۲ از متغیرهای تصادفی شاخص برای حل مسئله زیر، که به **مسئله تست کلاه**[1] معروف است، استفاده کنید. در یک رستوران هر یک از n مشتری یک کلاه به مسئول کلاه‌ها می‌دهد. مسئول کلاه‌ها، کلاهها را با یک ترتیب تصادفی به مشتریان برمی‌گرداند. تعداد مورد انتظار مشتریانی که کلاه خود را پس می‌گیرند چیست؟

۵-۵.۲ فرض کنید $A[1 \,..\, n]$ یک آرایه از n عدد متمایز باشد. اگر $i<j$ و $A[i]>A[j]$ آنگاه زوج (i, j) یک **وارونگی** از A نامیده می‌شود.

(مسئله ۴-۲ را برای مشاهده مطالب بیشتری در مورد وارونگی ملاحظه نمایید.) فرض کنید که هر عنصر از A بطور تصادفی، مستقل و یکنواخت از بازه 1 تا n انتخاب می‌شود. از متغیرهای تصادفی شاخص برای محاسبه تعداد وارونگی‌های مورد انتظار استفاده نمایید.

۵.۳ الگوریتم‌های تصادفی

در بخش قبل، نشان دادیم که چگونه اطلاع از نحوه توزیع روی ورودیها می‌تواند به ما کمک کند تا رفتار یک الگوریتم را در حالت میانگین تحلیل کنیم. بیشتر اوقات چنین اطلاعی نداشته و تحلیل در حالت میانگین امکان‌پذیر نیست. همانطور که در بخش ۵.۱ ذکر شد، ممکن است قادر به استفاده از یک الگوریتم تصادفی باشیم.

برای مسئله‌ای مانند مسئله استخدام که در آن انجام این فرض که همه جایگشتهای ورودی احتمال برابری دارند مفید است، یک تحلیل احتمالی، توسعه یک الگوریتم تصادفی را هدایت خواهد کرد. بجای آنکه توزیعی از ورودیها را فرض کنیم، یک توزیع را اعمال می‌کنیم. بخصوص قبل از اجرای الگوریتم برای اعمال این ویژگی که احتمال هر جایگشت برابر است، کاندیداها را بصورت تصادفی جابجا می‌کنیم. این تغییر، انتظار استخدام یک معاون اداری جدید را که تقریباً برابر $\ln n$ است تغییر نمی‌دهد. گرچه این بدان معنی است که برای هر ورودی، انتظار داریم این حالت بجای حالتی که ورودیها مطابق

1. hat-check problem

با یک توزیع خاص هستند رخ دهد.

اکنون تفاوت بین تحلیل احتمالی و الگوریتم‌های تصادفی را بیشتر بررسی می‌کنیم. در بخش ۵.۲ ادعا کردیم که با انجام این فرض که کاندیداها در یک ترتیب تصادفی قرار گرفته‌اند، تعداد مورد انتظار دفعاتی که یک معاون اداری جدید را استخدام می‌کنیم حدوداً $ln n$ است. توجه کنید که الگوریتم در اینجا مشخص است؛ برای هر ورودی خاص، تعداد دفعاتی که یک معاون اداری جدید استخدام می‌شود همواره همان تعداد خواهد بود. بعلاوه تعداد دفعاتی که یک معاون اداری جدید را استخدام می‌کنیم برای ورودیهای متفاوت فرق می‌کند و به رتبه‌های کاندیداهای مختلف بستگی دارد. از آنجا که این تعداد تنها به رتبه‌های کاندیداها بستگی دارد، می‌توانیم یک ورودی خاص را با لیست کردن رتبه‌های کاندیداها بصورت مرتب، یعنی $<rank(1), rank(2), \ldots, rank(n)>$، نمایش دهیم. با دریافت لیست رتبه‌های $A_1 = <1,2,3,4,5,6,7,8,9,10>$، یک معاون اداری جدید همواره 10 بار استخدام خواهد شد، زیرا هر کاندیدای بعدی بهتر از کاندیدای قبلی است، و خطوط ۶-۵ در هر تکرار الگوریتم اجرا خواهند شد. با دریافت لیست رتبه‌های $A_2 = <10,9,8,7,6,5,4,3,2,1>$، یک معاون اداری جدید، تنها یکبار استخدام خواهد شد، در اولین تکرار. با دریافت لیست رتبه‌های $A_3 = <5,2,1,8,4,7,10,9,3,6>$، یک معاون اداری جدید سه بار پس از مصاحبه با کاندیداهای با رتبه‌های 5، 8 و 10، استخدام خواهد شد. به یاد آورید که هزینه الگوریتم ما به تعداد دفعاتی که یک معاون اداری جدید را استخدام می‌کنیم بستگی دارد، می‌بینیم که ورودیهای پرهزینه‌ای مانند A_1، ورودیهای کم هزینه‌ای مانند A_2، وورودیهایی با هزینه متوسط، مانند A_3، وجود دارند.

از سوی دیگر، الگوریتم تصادفی را در نظر بگیرید که ابتدا کاندیداها را جایگشت کرده و سپس بهترین کاندیدا را تعیین می‌کند. در این حالت، تصادفی سازی در الگوریتم وجود دارد، نه در توزیع ورودی. با دریافت یک ورودی خاص مانند A_3، نمی‌توانیم بگوییم که به طور ماکزیمم چند مرتبه به روزرسانی خواهد شد، زیرا این کمیت با هر اجرای الگوریتم فرق می‌کند. اولین باری که الگوریتم را روی A_3 اجرا می‌کنیم، ممکن است جایگشت A_1 را تولید کرده و 10 به روز رسانی انجام دهد. در حالیکه مرتبه دوم که الگوریتم را اجرا می‌کنیم، ممکن است جایگشت A_2 را تولید کنیم و فقط یک به روز رسانی انجام دهیم. سومین مرتبه که آن را اجرا می‌کنیم، ممکن است تعداد به روز رسانی‌های دیگری را انجام دهیم. هر بار که الگوریتم را اجرا کنیم، اجرا به انتخاب‌های تصادفی انجام شده بستگی دارد و به احتمال زیاد با اجرای قبلی الگوریتم متفاوت است. برای این الگوریتم و بیشتر الگوریتم‌های تصادفی دیگر، هیچ ورودی خاصی باعث رفتار **بدترین حالت** آن نمی‌شود. حتی بدترین دشمن شما هم نمی‌تواند یک آرایه ورودی بد تولید کند، زیرا جایگشت تصادفی ترتیب ورودی را بدون ربط می‌سازد. الگوریتم تصادفی فقط اگر مولد عدد تصادفی، یک جایگشت "بدشانس" تولید کند، بد انجام می‌شود.

برای مسئله استخدام تنها تغییر لازم در کد، جایگشت کردن آرایه بطور تصادفی است.

```
RANDOMIZED-HIRE-ASSISTANT(n)
1  randomly permute the list of candidates
2  best ← 0          ▷ candidate 0 is a least-qualified dummy candidate
3  for i ← 1 to n
4      do interview candidate i
5         if candidate i is better than candidate best
6            then best ← i
7                 hire candidate i
```

با این تغییر ساده، یک الگوریتم تصادفی ایجاد کرده‌ایم که کارآیی آن همانند آنچه است که با فرض ارائه شدن کاندیداها، در یک ترتیب تصادفی حاصل می‌شود.

لم ۵.۳

هزینه استخدام مورد انتظار روال *RANDOMIZED-HIRE-ASSISTANT* برابر $O(c_h \ln n)$ است.

اثبات پس از جایگشت کردن آرایه ورودی، به وضعیتی یکسان با وضعیت تحلیل احتمالی *HIRE-ASSISTANT* رسیده‌ایم. ▫

مقایسه بین لم ۵.۲ و ۵.۳ تفاوت بین تحلیل احتمالی و الگوریتم‌های تصادفی را حاصل می‌کند. در لم ۵.۲ فرضی در مورد ورودی انجام می‌دهیم. در لم ۵.۳ چنین فرضی را انجام نمی‌دهیم، اگر چه تصادفی کردن ورودی، زمان اضافی را صرف می‌کند. در ادامه این بخش، در مورد بعضی مطالب مربوط به ورودیهای بطور تصادفی جایگشت شده بحث می‌کنیم.

آرایه‌های جایگشت شده بصورت تصادفی

بسیاری از الگوریتم‌های تصادفی، ورودی را بوسیله جایگشت کردن آرایه ورودی داده شده، تصادفی می‌کنند. (راههای دیگری برای استفاده از تصادفی سازی وجود دارند.) در اینجا، در مورد دو روش انجام این کار بحث خواهیم کرد. بدون از دست دادن کلّیت فرض می‌کنیم آرایه A، که شامل عناصر 1 تا n است، به ما داده می‌شود. هدف ما، تولید یک جایگشت تصادفی از آرایه است.

یک روش معمول این است که به هر عنصر $A[i]$ از آرایه، یک اولویت تصادفی $P[i]$ نسبت دهیم، و سپس عناصر A را متناظر با این اولویت‌ها مرتب کنیم. بعنوان مثال، اگر آرایه اولیه ما $A = \langle 1,2,3,4 \rangle$ باشد و اولویت‌های تصادفی $P = \langle 36,3,97,19 \rangle$ را انتخاب کنیم، آرایه $B = \langle 2,4,1,3 \rangle$ را ایجاد خواهیم کرد زیرا دومین اولویت، کوچکترین اولویت است که پس از آن چهارمین اولویت و سپس اولین، و در نهایت سومین اولویت قرار دارند. این روال را *PERMUTE-BY-SORTING* می‌نامیم:

```
PERMUTE-BY-SORTING(A)
1  n ← length[A]
2  for i ← 1 to n
3      do P[i] = RANDOM(1, n^3)
4  sort A, using P as sort keys
5  return A
```

خط ۳ عددی تصادفی بین 1 و n^3 انتخاب می‌کند. از بازه 1 تا n^3 استفاده می‌کنیم تا احتمال آنکه همه اولویت‌ها در P منحصر بفرد هستند را زیاد کنیم.

(تمرین ۵-۵.۳.۳ از شما می‌خواهد تا ثابت کنید احتمال اینکه همه ورودیها منحصر بفرد هستند حداقل $1-1/n$ است، و تمرین ۶-۵.۳ چگونگی پیاده سازی الگوریتم را حتی اگر دو اولویت یا بیشتر یکسان باشند از شما می‌خواهد.) اجازه دهید فرض کنیم که همه اولویت‌ها منحصر بفرد هستند.

مرحله زمانبر در این روال مرتب‌سازی در خط ۴ است. همانطور که در فصل ۸ خواهیم دید اگر از مرتب سازی مقایسه‌ای استفاده کنیم، مرتب‌سازی زمان $\Omega(nlgn)$ را صرف می‌کند. می‌توانیم به این حد پایین برسیم، زیرا مشاهده کرده‌ایم که مرتب‌سازی ادغام زمان $\Theta(nlgn)$ را صرف می‌کند. (مرتب‌سازیهای مقایسه‌ای دیگری که زمان $\Theta(nlgn)$ را صرف می‌کنند در قسمت ۲ خواهیم دید.) پس از مرتب سازی اگر $P[i]$، j امین اولویت کوچکتر باشد، آنگاه $A[i]$ در مکان j خروجی خواهد بود. با این روند یک جایگشت بدست می‌آوریم. تنها باقی می‌ماند ثابت کنیم که این روال یک **جایگشت تصادفی یکنواخت** را تولید می‌کند، بعبارت دیگر، تولید هر جایگشت از اعداد 1 تا n احتمال یکسانی دارد.

لم ۵.۴

روال *PERMUTE-BY-SORTING* یک جایگشت تصادفی یکنواخت از ورودی تولید می‌کند، با این فرض که همه اولویت‌ها متمایز هستند.

اثبات با در نظر گرفتن جایگشت خاصی که در آن هر عنصر $A[i]$ کوچکترین اولویت i ام را دریافت می‌کند، شروع می‌کنیم. نشان خواهیم داد که این جایگشت با احتمال دقیقاً $1/n!$ رخ می‌دهد. برای $i = 1, 2, \ldots, n$ فرض کنید X_i پیشامدی است که عنصر $A[i]$ کوچکترین اولویت i ام را دریافت می‌کند. آنگاه می‌خواهیم احتمال این که برای همه i ها، پیشامد X_i رخ می‌دهد را محاسبه کنیم که برابر است با

$$\Pr\{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\} .$$

این احتمال برابر است با

$$\Pr\{X_1\} \cdot \Pr\{X_2 \mid X_1\} \cdot \Pr\{X_3 \mid X_2 \cap X_1\} \cdot \Pr\{X_4 \mid X_3 \cap X_2 \cap X_1\}$$
$$\cdots \Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} \cdots \Pr\{X_n \mid X_{n-1} \cap \cdots \cap X_1\} .$$

داریم $Pr\{X_1\}=1/n$ زیرا برابر است با احتمالی که یک اولویت انتخاب شده بصورت تصادفی از مجموعه n، کوچکترین باشد. سپس، مشاهد می‌کنیم که $Pr\ \{X_2/X_1\}=1/(n-1)$ زیرا با دریافت عنصر $A[1]$ که کوچکترین اولویت را دارد، هر کدام از $(n-1)$ عنصر باقی‌مانده، شانس یکسانی برای داشتن کوچکترین اولویت دوم را دارد. بطور کلی، برای $i= 2, 3,. . .,n$ داریم که

$$\Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} = 1/(n-i+1),$$

زیرا با دریافت عناصر $A[1]$ تا $A[i-1]$ که (بترتیب) دارای $(i-1)$ اولویت کوچکتر هستند، هر کدام از $n-(i-1)$ عنصر باقی مانده، شانس یکسانی برای داشتن کوچکترین اولویت i ام را دارند. بنابراین داریم

$$\begin{aligned}\Pr\{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\} &= \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right)\cdots\left(\frac{1}{2}\right)\left(\frac{1}{1}\right)\\ &= \frac{1}{n!},\end{aligned}$$

و نشان داده‌ایم که احتمال بدست آوردن جایگشت همانی برابر $1/n!$ است.

می‌توانیم این اثبات را برای هر جایگشت از اولویت‌ها توسعه دهیم. جایگشت ثابت $\sigma = <\sigma(1), \sigma(2), . . . , \sigma(n)>$ از مجموعه $\{1, 2, . . . , n\}$ را در نظر بگیرید. رتبه اولویت نسبت داده شده به عنصر $A[i]$ را با r_i نشان می‌دهیم، که عنصر با کوچکترین اولویت j ام دارای رتبه j است. اگر X_i را پیشامدی که در آن عنصر $A[i]$ کوچکترین اولویت $\sigma(i)$ ام را دریافت می‌کند، تعریف کنیم، یا $r_i = \sigma(i)$ ، همان اثبات هنوز بکار می‌رود. بنابراین اگر احتمال بدست آوردن هر جایگشت خاص را حساب کنیم، محاسبه مشابه محاسبه بالا است، بنابراین احتمال بدست آوردن این جایگشت نیز $1/n!$ است. ■

ممکن است این طور تصور شود که برای اثبات اینکه یک جایگشت، یک جایگشت تصادفی یکنواخت است، کافی است نشان دهیم که برای هر عنصر $A[i]$ احتمال اینکه در مکان j ام خاتمه یابد، $1/n$ است. تمرین ۴-۵.۳ نشان می‌دهد که این شرط ضعیف‌تر، واقعاً، ناکافی است.

روش بهتر برای تولید یک جایگشت تصادفی این است که آرایه داده شده را بصورت درجا، جایگشت کنیم. روال *RANDOM-IN-PLACE* این کار را در زمان $O(n)$ انجام می‌دهد. در تکرار i ، عنصر $A[i]$ بطور تصادفی از بین عناصر $A[i]$ تا $A[n]$ انتخاب می‌شود. پس از تکرار i ، $A[i]$ هرگز

تغییر نمی‌کند.

```
RANDOMIZE-IN-PLACE(A)
1  n ← length[A]
2  for i ← 1 to n
3       do swap A[i] ↔ A[RANDOM(i, n)]
```

از ثابت حلقه استفاده می‌کنیم تا نشان دهیم که روال *RANDOMIZE-IN-PLACE* یک جایگشت تصادفی یکنواخت تولید می‌کند. با دریافت مجموعه‌ای از n عنصر، یک k- جایگشت، یک توالی شامل k عنصر از n عنصر است. $n!/(n-k)!$ تعداد از چنین k- جایگشت‌های ممکن وجود دارند.

لم ۵.۵

روال *RANDOMIZE-IN-PLACE* یک جایگشت تصادفی یکنواخت را محاسبه می‌کند.

اثبات ثابت حلقه زیر را استفاده می‌کنیم:

درست قبل از iامین تکرار حلقه *for* خطوط ۳-۲، برای هر $(i-1)$- جایگشت ممکن، زیر آرایه $A[1 \,..\, i-1]$ با احتمال $(n-i+1)! \,/\, n!$، شامل این $(i-1)$- جایگشت است.

لازم است نشان دهیم که این ثابت قبل از اولین تکرار حلقه درست است، و هر تکرار حلقه ثابت را حفظ می‌کند، و این که ثابت، یک ویژگی مفید برای نشان دادن صحت وقتی که حلقه به پایان می‌رسد فراهم می‌کند.

مقدار دهی اولیه: وضعیت قبل از اولین تکرار حلقه را در نظر بگیرید، که $i=1$. ثابت حلقه بیان می‌کند که برای هر 0– جایگشت ممکن، زیر آرایه $A[1 \,..\, 0]$ با احتمال $(n-i+1)!/n!=n!/n!=1$ شامل این 0–جایگشت است. زیرا آرایه $A[1 \,..\, 0]$ یکی زیرآرایه خالی است و یک 0–جایگشت هیچ عنصری ندارد. بنابراین $A[1 \,..\, 0]$ شامل هر 0–جایگشت با احتمال 1 است، و ثابت حلقه قبل از اولین تکرار برقرار است.

نگهداری: فرض می‌کنیم درست قبل از $(i-1)$ امین تکرار، هر $(i-1)$– جایگشت ممکن با احتمال $(n-i+1)!/n!$، در زیر آرایه $A[1 \,..\, i-1]$ قرار دارد. و ما نشان خواهیم داد که بعد از تکرار iام، هر i- جایگشت ممکن با احتمال $(n-i)!/n!$ در زیر آرایه $A[1-i]$ قرار دارد. لذا افزایش i برای تکرار بعدی، ثابت حلقه را حفظ خواهد کرد.

اجازه دهید iامین تکرار را بررسی کنیم. یک i– جایگشت خاص را در نظر بگیرید، و عناصر آن را با $\langle x_1 , x_2 , \ldots , x_i \rangle$ نشان دهید. این جایگشت تشکیل شده است از یک $(i-1)$– جایگشت $\langle x_1, \ldots , x_{i-1} \rangle$، بهمراه مقدار x_i، در ادامه، که الگوریتم آن را در $A[i]$ قرار می‌دهد. فرض کنید E_1

پیشامدی است که در آن اولین $i-1$ تکرار، $(i-1)$- جایگشت خاص $<x_1, . . . , x_{i-1}>$ را در $A[1 . . i]$ ایجاد کرده‌اند. بنا به ثابت حلقه داریم $Pr\{E_1\}=(n-i+1)! / n!$. فرض کنید E_2 پیشامدی است که تکرار iام، x_i را در مکان $A[i]$ قرار می‌دهد. دقیقاً وقتی که E_1 و E_2 هر دو رخ می‌دهند، i- جایگشت $<x_1, . . . , x_i>$ در $A[1 . . i]$ تشکیل می‌شود، و بنابراین می‌خواهیم $Pr\{E_2 \cap E_1\}$ را محاسبه کنیم. داریم

$$Pr\{E_2 \cap E_1\} = Pr\{E_2 \mid E_1\} \, Pr\{E_1\}$$

احتمال $Pr\{E_2 \mid E_1\}$ برابر $1/(n-i+1)$ است، چون در خط ۳، الگوریتم بطور تصادفی x_i را از $n-i+1$ مقدار در مکانهای $A[i . . n]$ انتخاب می‌کند. بنابراین داریم

$$\begin{aligned} \Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \\ &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\ &= \frac{(n-i)!}{n!} . \end{aligned}$$

خاتمه: در پایان، $i = n + 1$ و زیر آرایه $A[1 . . n]$ یک n- جایگشت داده شده با احتمال $(n - n)! / n! = 1 / n!$ است.

بنابراین روال *RANDOMIZE-IN-PLACE* یک جایگشت تصادفی یکنواخت تولید می‌کند. ■

یک الگوریتم تصادفی اغلب ساده‌ترین و کارآمدترین راه برای حل یک مسئله است. گاهی اوقات در سرتاسر کتاب از الگوریتم‌های تصادفی استفاده خواهیم کرد.

تمرین‌ها

۵.۳-۱ پروفسور *Marceau* به ثابت حلقه استفاده شده در اثبات لم ۵.۵ اعتراض دارد. او درست بودن آن را قبل از اولین تکرار، زیر سوال می‌برد. استدلال وی اینست که می‌توان درست به همان سادگی بیان داشت که یک زیر آرایه خالی شامل هیچ 0- جایگشتی نیست. بنابراین احتمال این که یک زیر آرایه خالی شامل یک 0- جایگشت باشد صفر خواهد بود، لذا ثابت حلقه قبل از اولین تکرار، نامعتبر است. روال *RANDOMIZE-IN-PLACE* را بازنویسی کنید تا ثابت حلقه مربوط به آن برای زیر آرایه غیرخالی قبل از اولین تکرار بکار رود، و اثبات لم ۵.۵ را برای روال خود تغییر دهید.

۵.۳-۲ پروفسور *Kelp* تصمیم می‌گیرد روالی بنویسد که علاوه بر جایگشت همانی، بصورت تصادفی یک جایگشت را تولید کند. او زیرروال زیر را پیشنهاد می‌کند:

```
PERMUTE-WITHOUT-IDENTITY(A)
1  n ← length[A]
2  for i ← 1 to n
3       do swap A[i] ↔ A[RANDOM(i + 1, n)]
```

آیا این کد هر آنچه پروفسور *Kelp* در نظر دارد را انجام می‌دهد؟

۵.۳-۳ فرض کنید که به جای تعویض عنصر *A[i]* با یک عنصر تصادفی از زیرآرایه *A[i . . n]* آن را با این عنصر تصادفی از هر جای آرایه تعویض می‌کردیم:

```
PERMUTE-WITH-ALL(A)
1  n ← length[A]
2  for i ← 1 to n
3       do swap A[i] ↔ A[RANDOM(1, n)]
```

آیا این کد جایگشت تصادفی یکنواخت تولید می‌کند؟ چرا بله یا چرا خیر؟

۵.۳-۴ پرفسور *Armstrong* روال زیر را برای تولید یک جایگشت تصادفی یکنواخت پیشنهاد می‌کند:

```
PERMUTE-BY-CYCLIC(A)
1  n ← length[A]
2  offset ← RANDOM(1, n)
3  for i ← 1 to n
4      do dest ← i + offset
5          if dest > n
6              then dest ← dest − n
7          B[dest] ← A[i]
8  return B
```

نشان دهید که هر عنصر *A[i]* دارای *1/n* احتمال خاتمه یافتن در هر مکان خاص در B است. سپس نشان دهید که پروفسور *Armstrong* با نشان دادن اینکه جایگشت حاصل بطور یکنواخت تصادفی نیست دچار اشتباه شده است.

۵.۳-۵ توضیح دهید که چگونه الگوریتم *PERMUTE-BY-SORTING* را برای مدیریت حالتی که در آن دو اولویت یا بیشتر، یکسان باشند پیاده‌سازی کنیم. به عبارت دیگر، الگوریتم شما باید یک جایگشت تصادفی یکنواخت، حتی اگر دو اولویت یا بیشتر یکسان باشند تولید کند.

* ۵.۴ تحلیل احتمالی و استفاده‌های دیگر متغیرهای تصادفی شاخص

این بخش پیشرفته تحلیل احتمالی را بوسیله چهار مثال، بیشتر توضیح می‌دهد. مثال اول، احتمالی که در یک اتاق از *k* نفر، دو نفر روز تولد یکسانی داشته باشند را تعیین می‌کند. مثال دوم، پرتاب تصادفی توپها به داخل جعبه‌ها را بررسی می‌کند. مثال سوم به «توالی‌های» شیرهای متوالی در پرتاب سکه می‌پردازد. مثال آخر، شکل دیگر مسئله استخدام را تحلیل می‌کند، که در آن شما مجبورید تصمیماتی را

بدون آنکه واقعاً با تمام کاندیداها مصاحبه کنید بگیرید.

۵.۴.۱ متناقض‌نمای روز تولد

مثال اول ما متناقض‌نمای روز تولد [1] است. چند نفر باید در یک اتاق باشند، تا یک شانس 50% وجود داشته باشد که دو نفر از آنها در روز یکسانی از سال متولد شده باشند؟ جواب بطور تعجب آوری کم است. تناقض نمایی که وجود دارد این است که در واقع همانطور که خواهیم دید جواب بسیار کمتر از تعداد روزهای یک سال، یا حتی نیمی از تعداد روزها در یک سال است.

برای پاسخ دادن به این سؤال، افراد داخل اتاق را با اعداد صحیح $1, 2, \ldots, k$ که k تعداد افراد داخل اتاق است، اندیس گذاری می‌کنیم. سالهای کبیسه را نادیده گرفته و فرض می‌کنیم که همه سالها دارای $n=365$ روز می‌باشند. فرض کنید برای $i=1, 2, \ldots, k$ b_i روزی از سال باشد که روز تولد فرد i در آن روز است، که $1 \le b_i \le n$. همچنین فرض می‌کنیم که روزهای تولد بطور یکنواخت روی n روز از سال توزیع شده‌اند، بطوریکه برای $i=1,2,\ldots,k$ و $r=1, 2, \ldots, n$ داریم $\Pr\{b_i = r\}=1/n$

احتمال اینکه دو فرد i و j روزهای تولد یکسانی داشته باشند، به اینکه انتخاب تصادفی روزهای تولد مستقل باشد بستگی دارد. از اکنون به بعد فرض می‌کنیم که روزهای تولد مستقل هستند، بنابراین احتمال اینکه روز تولد i و روز تولد j هر دو در روز r باشند برابر است با

$$\begin{aligned} \Pr\{b_i = r \text{ and } b_j = r\} &= \Pr\{b_i = r\}\Pr\{b_j = r\} \\ &= 1/n^2 . \end{aligned}$$

بنابراین احتمال اینکه هر دو در روز یکسانی متولد شده باشند برابر است با

$$\begin{aligned} \Pr\{b_i = b_j\} &= \sum_{r=1}^{n} \Pr\{b_i = r \text{ and } b_j = r\} \\ &= \sum_{r=1}^{n} (1/n^2) \\ &= 1/n . \end{aligned} \tag{۵.۷}$$

بطور شهودی‌تر، وقتی b_i انتخاب می‌شود احتمال اینکه b_j با همان روز تولد انتخاب شود $1/n$ است. بنابراین احتمال اینکه i و j روز تولد یکسانی داشته باشند مشابه است با احتمال اینکه روز تولد یکی از آنها در یک روز مشخص شده باشد. اما توجه کنید که این همسانی به این فرض که روزهای تولد مستقل هستند بستگی دارد.

می‌توانیم احتمال اینکه حداقل 2 نفر از میان k نفر روزهای تولد یکسانی دارند را با توجه به پیشامد متمم تحلیل کنیم. احتمال اینکه حداقل دو روز تولد یکسان باشند برابر است با 1 منهای احتمال اینکه همه روزهای تولد متفاوت باشند. پیشامدی که k فرد دارای روزهای تولد متفاوت هستند برابر است با

---

1. birthday paradox

$$B_k = \bigcap_{i=1}^{k} A_i ,$$

که در آن A_i پیشامدی است که برای تمام $j < i$، روز تولد فرد i با فرد j تفاوت دارد. از آن جا که می‌توانیم بنویسیم $B_k = A_k \cap B_{k-1}$ رابطه بازگشتی زیر را بدست می‌آوریم

$$\Pr\{B_k\} = \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\} , \qquad (۵.۸)$$

که $Pr\{B_1\} = Pr\{A_1\} = 1$ را به عنوان یک شرط اولیه می‌گیریم. بعبارت دیگر، احتمال اینکه $b_1, b_2, \ldots, b_k$ روزهای تولد متفاوتی باشند برابر است با احتمال این که $b_1, b_2, \ldots, b_{k-1}$ روزهای تولد متفاوت باشند ضرب در احتمال اینکه برای $i = 1, 2, \ldots, k-1$ $b_k \neq b_i$ با این فرض که $b_1, b_2, \ldots, b_{k-1}$ متفاوت هستند. اگر $b_1, b_2, \ldots, b_{k-1}$ متفاوت باشند، احتمال شرطی که برای $i = 1, 2, \ldots, b_{k-1}$، $b_k \neq b_i$، برابر $Pr\{A_k \mid B_{k-1}\} = (n-k+1)/n$ است، زیرا از میان n روز، $n-(k-1)$ روز وجود دارند که استفاده نشده‌اند. مکرراً رابطه بازگشتی (۵.۸) را بکار می‌بریم تا بدست آوریم

$$\begin{aligned}
\Pr\{B_k\} &= \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\} \\
&= \Pr\{B_{k-2}\} \Pr\{A_{k-1} \mid B_{k-2}\} \Pr\{A_k \mid B_{k-1}\} \\
&\vdots \\
&= \Pr\{B_1\} \Pr\{A_2 \mid B_1\} \Pr\{A_3 \mid B_2\} \cdots \Pr\{A_k \mid B_{k-1}\} \\
&= 1 \cdot \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right) \cdots \left(\frac{n-k+1}{n}\right) \\
&= 1 \cdot \left(1 - \frac{1}{n}\right)\left(1 - \frac{2}{n}\right) \cdots \left(1 - \frac{k-1}{n}\right) .
\end{aligned}$$

با استفاده از نامساوی (۳.۱۱)، $1+x \leq e^x$، داریم

$$\begin{aligned}
\Pr\{B_k\} &\leq e^{-1/n} e^{-2/n} \cdots e^{-(k-1)/n} \\
&= e^{-\sum_{i=1}^{k-1} i/n} \\
&= e^{-k(k-1)/2n} \\
&\leq 1/2
\end{aligned}$$

که $-k(k-1)/2n \leq ln1/2$. احتمال اینکه تمام k روز تولد متفاوت باشند، زمانیکه $k(k-1) \geq 2nln2$ حداکثر $1/2$ است، یا حل معادله درجه دوم وقتی که $k \geq (1+\sqrt{1+(8ln2)n})/2$. برای $n=365$ باید داشته باشیم، $k \geq 23$. بنابراین اگر حداقل 23 نفر در یک اتاق باشند، احتمال اینکه حداقل دو نفر روز

تولد یکسانی داشته باشند حداقل $1/2$ است. در مریخ، طول یکسال 669 روز مریخی است؛ بنابراین 31 روز مریخی نیاز است تا به نتیجه یکسانی برسیم.

تحلیل با استفاده از متغیرهای تصادفی شاخص

می‌توانیم از متغیرهای تصادفی شاخص برای فراهم کردن یک تحلیل ساده‌تر اما تقریبی از تناقض نمای روز تولد استفاده کنیم. برای هر زوج (i, j) از k فرد در اتاق، متغیر تصادفی شاخص X_{ij} را برای $1 \le i < j \le k$ بصورت زیر تعریف می‌کنیم

$$X_{ij} = \mathrm{I}\{\text{فرد } i \text{ و فرد } j \text{ روز تولد یکسانی داشته باشند}\}$$
$$= \begin{cases} 1 & \text{اگر فرد } i \text{ و فرد } j \text{ روز تولد یکسانی داشته باشند} \\ 0 & \text{در غیر اینصورت} \end{cases}$$

بنا به معادله (۵.۷)، احتمال اینکه دو نفر روزهای تولد یکسانی داشته باشند $1/n$ است، و بنابراین بنا به لم ۵.۱، داریم

$$E[X_{ij}] = Pr\ \{\text{فرد } i \text{ و فرد } j \text{ روز تولد یکسانی داشته باشند}\} = 1/n$$

X را متغیر تصافی در نظر بگیرید که تعداد زوج افرادی که روز تولد یکسانی دارند را حساب می‌کند، داریم

$$X = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij} .$$

با گرفتن انتظار (امید ریاضی) از هر دو طرف و بکارگیری خطی بودن انتظار، بدست می‌آوریم

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}\right] \\ &= \sum_{i=1}^{k} \sum_{j=i+1}^{k} \mathrm{E}[X_{ij}] \\ &= \binom{k}{2}\frac{1}{n} \\ &= \frac{k(k-1)}{2n} . \end{aligned}$$

بنابراین، وقتیکه $k(k-1) \ge 2n$ تعداد مورد انتظار زوج افراد با روز تولد یکسان حداقل 1 است. بنابراین، اگر حداقل $\sqrt{2n}+1$ نفر در اتاق داشته باشیم، می‌توانیم انتظار داشته باشیم که حداقل دو نفر روز تولد یکسانی داشته باشند. برای $n= 365$، اگر $k=28$، تعداد مورد انتظار زوج‌های با روز تولد

یکسان برابر $1.0365 \approx (28.27)/(2.365)$ است. بنابراین، با حداقل 28 نفر، انتظار داریم تا حداقل یک زوج با روز تولد یکسان پیدا کنیم. در مریخ، که یکسال 669 روز طول می‌کشد، حداقل به 38 روز مریخی نیاز داریم.

تحلیل اول که فقط از احتمالات استفاده می‌کرد، تعداد افراد لازم برای اینکه احتمال وجود یک زوج با روز تولد یکسان از $1/2$ تجاوز کند را تعیین کرد و تحلیل دوم که از متغیرهای تصادفی شاخص استفاده می‌کرد، تعداد افراد لازم بطوریکه تعداد مورد انتظار روزهای تولد یکسان برابر 1 باشد را تعیین کرد. اگر چه تعداد دقیق افراد برای این دو وضعیت متفاوت است، اما آنها بطور مجانبی یکسان هستند: $\Theta(\sqrt{n})$.

۵.۴.۲ توپها و جعبه‌ها

فرآیند پرتاب تصادفی توپهای یکسان در b جعبه با شماره‌های $1, 2, \ldots, b$ را در نظر بگیرید. پرتاب‌ها مستقل هستند، و در هر پرتاب احتمال قرار گرفتن توپ در هر جعبه مساوی است. احتمال اینکه توپ پرتاب شده در یک جعبه مشخّص قرار گیرد $1/b$ است. بنابراین فرآیند پرتاب توپ یک توالی از آزمایشهای برنولی با احتمال موفقیت $1/b$ است، که موفقیت بمعنی این است که توپ در جعبه مشخص شده بیفتد. این مدل بویژه برای تحلیل درهم سازی مفید است (فصل ۱۱ را ملاحظه نمایید)، و می‌توانیم به سؤالات جالب گوناگونی در مورد فرآیند پرتاب توپ پاسخ دهیم.

چه تعداد توپ در یک جعبه مشخص شده می‌افتند؟ تعداد توپهایی که در یک جعبه مشخص شده می‌افتند از توزیع دو جمله‌ای $b(k;n,1/b)$ پیروی می‌کند. اگر n توپ پرتاب شوند، تعداد مورد انتظار توپ‌هایی که در جعبه مشخص شده می‌افتند، n/b است.

بطور میانگین چه تعداد توپ باید پرتاب شوند تا یک جعبه مشخص شده شامل یک توپ شود؟ تعداد پرتاب‌ها تا زمانیکه جعبه مشخص شده یک توپ را دریافت کند از توزیع هندسی با احتمال $1/b$ پیروی می‌کند و تعداد مورد انتظار پرتابها تا زمان موفقیت، $1/(1/b)=b$ است.

چه تعداد توپ باید پرتاب شوند تا هر جعبه حداقل شامل یک توپ باشد؟ اجازه دهید پرتابی را که در آن یک توپ درون یک جعبه خالی می‌افتد "برخورد" بنامیم. می‌خواهیم تعداد مورد انتظار n از پرتابهای لازم برای بدست آوردن b برخورد را بدانیم.

برخوردها می‌توانند جهت افراز n پرتاب به چند مرحله استفاده شوند. مرحله i ام از پرتابهای بعد از $(i-1)$ امین برخورد تا برخورد i ام تشکیل شده است. مرحله اول از اولین پرتاب تشکیل شده است، زیرا تضمین می‌شود زمانیکه تمام جعبه‌ها خالی هستند یک برخورد داریم. برای هر پرتاب در طول مرحله i ام، $i-1$ جعبه وجود دارند که شامل توپ هستند و $b-i+1$ جعبه خالی وجود دارند. بنابراین، برای هر پرتاب در مرحله i ام، احتمال بدست آوردن یک برخورد $(b-i+1)/b$ است.

فرض کنید n_i تعداد پرتابها در مرحله i ام را نشان می‌دهد. بنابراین تعداد پرتابهای لازم برای بدست آوردن b برخورد برابر $n = \sum_{i=1}^{b} n_i$ است. هر متغیر تصادفی n_i دارای یک توزیع هندسی با احتمال موفقیت $(b-i+1)/b$ است و لذا داریم

$$E[n_i] = \frac{b}{b-i+1} .$$

بنا به خطی بودن انتظار

$$\begin{aligned} E[n] &= E\left[\sum_{i=1}^{b} n_i\right] \\ &= \sum_{i=1}^{b} E[n_i] \\ &= \sum_{i=1}^{b} \frac{b}{b-i+1} \\ &= b\sum_{i=1}^{b} \frac{1}{i} \\ &= b(\ln b + O(1)) . \end{aligned}$$

بنابراین تقریباً $blnb$ پرتاب قبل از اینکه بتوانیم انتظار داشته باشیم که هر جعبه دارای یک توپ باشد، نیاز است. این مسئله به **مسئله گردآور کوپن**، نیز معروف است و بیان می‌کند فردی که تلاش دارد هر کدام از b کوپن مختلف را گردآوری کند باید تقریباً $bLnb$ کوپن بصورت تصادفی بدست آمده را به منظور موفق شدن جمع آوری کند.

۵.۴.۳ توالی‌ها

فرض کنید یک سکه ناaریب را n بار پرتاب می‌کنید. طولانی‌ترین توالی از شیرهای متوالی که انتظار دارید مشاهده کنید چیست؟ همانطور که تحلیل زیر نشان می‌دهد پاسخ $\Theta(lgn)$ است.

ابتدا ثابت می‌کنیم که طول مورد انتظار طولانی‌ترین توالی از شیرها $O(lgn)$ است. احتمال اینکه هر پرتاب سکه یک شیر باشد $1/2$ است. فرض کنید A_{ik} پیشامدی باشد که یک توالی از شیرهای با طول حداقل k با i امین پرتاب سکه شروع می‌شود یا بطور دقیق‌تر، پیشامدی که k پرتاب متوالی سکه، i $i+1, \ldots, i+k-1$ و فقط به شیر منجر می‌شود که $1 \le k \le n$ و $1 \le i \le n-k+1$. از آنجا که پرتابهای سکه بطور متقابل مستقل هستند، برای هر پیشامد مفروض A_{ik}، احتمال اینکه تمام k پرتاب شیر بیایند برابر است با

$$\Pr\{A_{ik}\} = 1/2^k . \qquad (۵.۹)$$

برای $k = 2\lceil lgn \rceil$،

$$\begin{aligned}\Pr\{A_{i,2\lceil \lg n\rceil}\} &= 1/2^{2\lceil \lg n\rceil} \\ &\le 1/2^{2\lg n} \\ &= 1/n^2 ,\end{aligned}$$

و بنابراین احتمال اینکه یک توالی از شیرها با طول حداقل $2\lceil lgn\rceil$ از مکان i شروع شود بسیار کم است. حداکثر $n\text{-}2\lceil lgn\rceil+1$ مکان وجود دارند که چنین توالی می‌تواند شروع شود. بنابراین احتمال اینکه یک توالی از شیرها با طول حداقل $2\lceil lgn\rceil$ از هر جایی شروع شود برابر است با

$$\begin{aligned}\Pr\left\{\bigcup_{i=1}^{n-2\lceil \lg n\rceil+1} A_{i,2\lceil \lg n\rceil}\right\} &\le \sum_{i=1}^{n-2\lceil \lg n\rceil+1} 1/n^2 \\ &< \sum_{i=1}^{n} 1/n^2 \\ &= 1/n ,\end{aligned} \qquad (5.10)$$

زیرا بنا به **نامساوی** *Boole*[1] احتمال اجتماع پیشامدها حداکثر برابر حاصلجمع احتمال‌های تک‌تک پیشامدها است. (توجه کنید که این مطلب حتی برای چنین پیشامدهایی که مستقل نیستند بر قرار است.)

اکنون با استفاده از نامساوی (۵.۱۰) طول طولانی‌ترین توالی را محدود می‌کنیم. فرض کنید برای $j=0,2,\ldots,n$، L_j پیشامدی باشد که طولانی‌ترین توالی از شیرها دقیقاً طول j را داشته باشد و L را طول طولانی‌ترین توالی در نظر بگیرید. بنا به تعریف مقدار مورد انتظار،

$$\mathrm{E}[L] = \sum_{j=0}^{n} j\Pr\{L_j\} \qquad (5.11)$$

می‌توانیم با استفاده از حدهای بالا روی هر $Pr\{L_j\}$ مشابه آنهایی که در نامساوی (۵.۱۰) محاسبه شدند، این جمع را ارزشیابی کنیم. متأسفانه این روش حدهای ضعیفی حاصل خواهد کرد. امّا می‌توانیم شهود بدست آمده بوسیله تحلیل فوق را استفاده کنیم تا حد خوبی بدست آوریم. بطور غیر رسمی، مشاهده می‌کنیم که برای هیچ جمله جداگانه در حاصلجمع معادله (۵.۱۱) هر دو ضریب j و $Pr\{L_j\}$ بزرگ نیستند. چرا؟ وقتی $j\ge 2\lceil lgn\rceil$، آنگاه $Pr\{L_j\}$ بسیار کوچک است و وقتی $j<2\lceil lgn\rceil$، آنگاه j نسبتاً کوچک است. بطور رسمی‌تر، توجه داریم که برای $j=0,1,\ldots,n$، پیشامدهای L_j جدا از هم (مجزا) هستند و بنابراین احتمال اینکه یک توالی از شیرها با طول حداقل $2\lceil lgn\rceil$ از هر جایی شروع شود، $\sum_{i=2\lceil \lg n\rceil}^{n} \Pr\{L_j\}$ است. بنا به نامساوی (۵.۱۰)، داریم $\sum_{i=2\lceil \lg n\rceil}^{n} \Pr\{L_j\} < 1/n$.

---

۱ - نامساوی *Boole*: برای هر توالی محدود یا نامحدود شما را از پیشامدهای $A_1, A_2, \ldots$ داریم:

$Pr\{A_1\cup A_2\cup\ldots\}\le Pr\{A_1\}+Pr\{A_2\}+\ldots$

همچنین با توجه به اینکه $\sum_{j=0}^{n} \Pr\{L_j\} = 1$ داریم $\sum_{j=0}^{2\lceil \lg n \rceil -1} \Pr\{L_j\} \leq 1$.

بنابراین، بدست می‌آوریم

$$\begin{aligned}
E[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{2\lceil \lg n \rceil -1} j \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} j \Pr\{L_j\} \\
&< \sum_{j=0}^{2\lceil \lg n \rceil -1} (2\lceil \lg n \rceil) \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} n \Pr\{L_j\} \\
&= 2\lceil \lg n \rceil \sum_{j=0}^{2\lceil \lg n \rceil -1} \Pr\{L_j\} + n \sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} \\
&< 2\lceil \lg n \rceil \cdot 1 + n \cdot (1/n) \\
&= O(\lg n) .
\end{aligned}$$

شانس اینکه یک توالی از شیرها از $r\lceil lgn \rceil$ پرتاب بیشتر باشد، به سرعت با r کم می‌شود. برای $r \geq 1$ احتمال اینکه یک توالی از $r\lceil lgn \rceil$ شیر از موقعیت i شروع شود برابر است با

$$\begin{aligned}
\Pr\{A_{i,r\lceil \lg n \rceil}\} &= 1/2^{r\lceil \lg n \rceil} \\
&\leq 1/n^r .
\end{aligned}$$

بنابراین، احتمال اینکه طولانی‌ترین توالی حداقل $r\,[lgn]$ باشد حداکثر $n/n^r = 1/n^{r-1}$ است، یا بطور معادل، احتمال اینکه طولانی‌ترین توالی دارای طولی کمتر از $r\lceil lgn \rceil$ باشد، حداقل $1-1/n^{r-1}$ است. بعنوان مثال برای $n=1000$ پرتاب سکه، احتمال داشتن یک توالی با حداقل $2\lceil lgn \rceil = 20$ شیر حداکثر برابر $1/n=1/1000$ است. شانس داشتن یک توالی طولانی‌تر از $3\lceil lgn \rceil = 30$ شیر حداکثر برابر $1/n=1/1{,}000{,}000$ است.

اکنون یک حد پایین متمم را ثابت می‌کنیم: طول مورد انتظار طولانی‌ترین توالی از شیرها در n پرتاب سکه برابر $\Omega(lgn)$ است. برای اثبات این حد، با افراز n پرتاب به تقریباً n/s گروه هر یک شامل s پرتاب دنبال توالی‌های با طول s می‌گردیم. اگر انتخاب کنیم $s = \lfloor (lgn)/2 \rfloor$ می‌توانیم نشان دهیم احتمال اینکه حداقل یکی از این گروهها تماماً شامل شیرها باشد، زیاد است و از آنجا احتمال اینکه طولانی‌ترین توالی دارای طول حداقل $s=\Omega(lgn)$ باشد زیاد است. سپس نشان خواهیم داد که طول مورد انتظار طولانی‌ترین توالی برابر $\Omega(lgn)$ است.

n پرتاب سکه را به حداقل $\lfloor n/\lfloor (lgn)/2 \rfloor \rfloor$ گروه هر یک شامل $\lfloor (lgn)/2 \rfloor$ پرتاب متوالی افراز می‌کنیم، و احتمال اینکه هیچ گروهی تماماً شامل شیرها نباشد را محدود می‌کنیم. بنا به معادله (۵.۹)، احتمال اینکه گروهی که از موقعیت i شروع می‌شود تماماً شامل شیرها باشد برابر است با

$$\begin{aligned} \Pr\{A_{i,\lfloor(\lg n)/2\rfloor}\} &= 1/2^{\lfloor(\lg n)/2\rfloor} \\ &\geq 1/\sqrt{n} \end{aligned}$$

بنابراین احتمال اینکه یک توالی از شیرها با طول حداقل $\lfloor(lgn)/2\rfloor$ از موقعیت i شروع نشود، حداکثر $1\text{-}1/\sqrt{n}$ است. از آنجا که این $\lfloor n/\lfloor(lgn)/2\rfloor\rfloor$ گروه از پرتاب‌های مستقل و متقابلاً منحصر تشکیل شده‌اند، احتمال اینکه هر یک از گروهها یک توالی با طول $\lfloor(lgn)/2\rfloor$ نباشد حداکثر برابر است با

$$\begin{aligned} (1-1/\sqrt{n})^{\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor} &\leq (1-1/\sqrt{n})^{n/\lfloor(\lg n)/2\rfloor-1} \\ &\leq (1-1/\sqrt{n})^{2n/\lg n-1} \\ &\leq e^{-(2n/\lg n-1)/\sqrt{n}} \\ &= O(e^{-\lg n}) \\ &= O(1/n) \end{aligned}$$

برای این اثبات از نامساوی (۳.۱۱)، $1+x\leq e^x$، و این حقیقت که ممکن است بخواهید صحت آنرا تأیید کنید، که برای n به اندازه کافی بزرگ $(2n/lgn\text{-}1)/\sqrt{n}\geq lgn$ استفاده کردیم.

بنابراین، احتمال اینکه طولانی‌ترین توالی از $\lfloor(lgn)/2\rfloor$ تجاوز کند برابر است با

$$\sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} \Pr\{L_j\} \geq 1-O(1/n) \qquad (۵.۱۲)$$

اکنون می‌توانیم یک حد پایین روی طول مورد انتظار طولانی‌ترین توالی را با شروع از معادله (۵.۱۱) و پیش روی در یک روند مشابه با تحلیل حد بالا محاسبه کنیم:

$$\begin{aligned} \mathrm{E}[L] &= \sum_{j=0}^{n} j\Pr\{L_j\} \\ &= \sum_{j=0}^{\lfloor(\lg n)/2\rfloor} j\Pr\{L_j\} + \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} j\Pr\{L_j\} \\ &\geq \sum_{j=0}^{\lfloor(\lg n)/2\rfloor} 0\cdot\Pr\{L_j\} + \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} \lfloor(\lg n)/2\rfloor\Pr\{L_j\} \\ &= 0\cdot\sum_{j=0}^{\lfloor(\lg n)/2\rfloor}\Pr\{L_j\} + \lfloor(\lg n)/2\rfloor\sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n}\Pr\{L_j\} \\ &\geq 0+\lfloor(\lg n)/2\rfloor(1-O(1/n)) \qquad \text{(بنابه نامساوی (۵.۱۲))} \\ &= \Omega(\lg n)\,. \end{aligned}$$

همانند تناقض نمای روز تولد، می‌توانیم یک تحلیل ساده‌تر اما تقریبی را با استفاده از متغیرهای تصادفی شاخص بدست آوریم. $X_{ik}=I\{A_{ik}\}$ را متغیر تصادفی شاخص مربوط به یک توالی از شیرها با طول حداقل k که با iامین پرتاب سکّه شروع می‌شود، در نظر می‌گیریم. برای محاسبه تعداد کل چنین توالی‌هایی، تعریف می‌کنیم

$$X = \sum_{i=1}^{n-k+1} X_{ik}$$

با گرفتن انتظار (امید ریاضی) و استفاده از خطی بودن انتظار، داریم

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-k+1} X_{ik}\right] \\ &= \sum_{i=1}^{n-k+1} \mathrm{E}[X_{ik}] \\ &= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\} \\ &= \sum_{i=1}^{n-k+1} 1/2^k \\ &= \frac{n-k+1}{2^k}. \end{aligned}$$

با قرار دادن مقادیر متفاوتی برای k می‌توانیم تعداد مورد انتظار توالی‌ها با طول k را محاسبه کنیم. اگر این تعداد بزرگ باشد (بسیار بزرگتر از 1)، آنگاه انتظار می‌رود بسیاری از توالی‌ها با طول k رخ دهند و احتمال اینکه یکی رخ دهد زیاد است. اگر این تعداد کوچک باشد (بسیار کوچکتر از 1)، آنگاه انتظار می‌رود تعداد کمی از توالی‌ها با طول k رخ دهند و احتمال اینکه یکی رخ دهد کم است. اگر برای ثابت مثبت c، $k=c\lg n$، بدست می‌آوریم

$$\begin{aligned} \mathrm{E}[X] &= \frac{n-c\lg n+1}{2^{c\lg n}} \\ &= \frac{n-c\lg n+1}{n^c} \\ &= \frac{1}{n^{c-1}} - \frac{(c\lg n-1)/n}{n^{c-1}} \\ &= \Theta(1/n^{c-1}). \end{aligned}$$

اگر c بزرگ باشد، تعداد مورد انتظار توالی‌ها با طول $c\lg n$ بسیار کوچک است و نتیجه می‌گیریم که احتمال رخ دادن آنها کم است. از طرف دیگر، اگر $c<1/2$، آنگاه بدست می‌آوریم

$$E[X] = \Theta(1/n^{1/2-1}) = \Theta(n^{1/2})$$

و انتظار داریم که تعداد زیادی از توالی‌ها با طول $(1/2)\lg n$ وجود خواهند داشت. بنابراین، احتمال رخ دادن یک توالی با چنین طولی زیاد است. از این تخمین‌های غیر دقیق می‌توانیم نتیجه بگیریم که طول طولانی‌ترین توالی $\Theta(\lg n)$ است.

۵.۴.۴ مسئله استخدام پیوسته

بعنوان آخرین مثال، شکل دیگری از مسئله استخدام را در نظر می‌گیریم. اکنون فرض کنید که نمی‌خواهیم با همه کاندیداها به منظور یافتن بهترین شخص مصاحبه کنیم. همچنین نمی‌خواهیم همینکه متقاضیان بهتر و بهتر را پیدا کردیم، استخدام و اخراج را انجام دهیم. در عوض می‌خواهیم برای کاندیدایی که تقریباً بهترین است، جایی در نظر بگیریم تا آنکه دقیقاً یکبار استخدام را انجام دهیم. باید از درخواست یک شرکت پیروی کنیم: بعد از هر مصاحبه یا باید بلافاصله موقعیت را به متقاضی پیشنهاد کنیم یا باید به آنها بگوییم که شغل را دریافت نخواهند کرد. چه تبادلی بین مینیمم کردن میزان مصاحبه و ماکزیمم کردن کیفیت کاندیدای استخدام شده است؟

می‌توانیم این مسئله را به روش زیر مدل کنیم. پس از ملاقات با یک متقاضی، قادریم به هر یک امتیازی بدهیم؛ فرض کنید $score(i)$ امتیازی باشد که به متقاضی iام داده شده است، و فرض کنید هیچ دو متقاضی امتیاز یکسانی را دریافت نمی‌کنند. بعد از اینکه j متقاضی را دیدیم، می‌دانیم کدامیک از آنها بالاترین امتیاز را دارد، اما نمی‌دانیم کدامیک از $n-j$ متقاضی باقی مانده امتیاز بالاتری خواهد داشت.

تصمیم می‌گیریم استراتژی انتخاب یک عدد صحیح مثبت $k<n$، مصاحبه و سپس رد کردن k متقاضی اول، و پس از آن استخدام متقاضی اول که بالاترین امتیاز نسبت به تمام متقاضیان قبلی دارد را بپذیریم. اگر شایسته‌ترین متقاضی در بین k شخص اول که با آنها مصاحبه شده است باشد، آنگاه متقاضی nام را استخدام خواهیم کرد. این استراتژی در روال *ON-LINE-MAXIMUM* فرموله شده است، که در ذیل قرار دارد. روال *ON-LINE-MAXIMUM* اندیس کاندیدایی که می‌خواهیم استخدام کنیم را برمی‌گرداند.

```
ON-LINE-MAXIMUM(k, n)
1  bestscore ← −∞
2  for i ← 1 to k
3       do if score(i) > bestscore
4             then bestscore ← score(i)
5  for i ← k + 1 to n
6       do if score(i) > bestscore
7             then return i
8  return n
```

می‌خواهیم برای هر مقدار ممکن k احتمال اینکه شایسته‌ترین متقاضی را استخدام کنیم تعیین نماییم. سپس بهترین k ممکن را انتخاب خواهیم کرد و استراتژی را با این مقدار پیاده سازی می‌کنیم. اکنون فرض کنید k تعیین شده است. فرض کنید $M(j)=max_{1\le i\le j}\{score(i)\}$ امتیاز ماکزیمم در بین متقاضیان 1 تا j را نشان می‌دهد. S را پیشامد موفقیت ما در انتخاب شایسته‌ترین متقاضی و S_i را

پیشامد موفقیت ما هنگامیکه شایسته‌ترین متقاضی، i امین شخص مصاحبه شده است، در نظر بگیرید. از آنجا که S_iهای متفاوت، جدا از هم (مجزا) هستند، داریم $\Pr\{S\} = \sum_{i=1}^{n} \Pr\{S_i\}$ توجه کنید هنگامیکه شایسته‌ترین متقاضی یکی از k نفر اول است، برای $i = 1, 2, \ldots, k$ داریم $Pr\{S_i\}=0$. بنابراین، بدست می‌آوریم

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\} \,. \tag{۵.۱۳}$$

اکنون $Pr\{S_i\}$ را محاسبه می‌کنیم. به منظور موفق شدن وقتیکه شایسته‌ترین متقاضی، i امین شخص است، دو چیز باید اتفاق بیفتد. اول آنکه شایسته‌ترین متقاضی باید در موقعیت i باشد، پیشامدی که با B_i نشان می‌دهیم. دوم آنکه الگوریتم نباید یکی از متقاضیان در موقعیت‌های $k+1$ تا $i-1$ را انتخاب کند، که زمانی‌اتفاق می‌افتد که برای‌هر j بطوریکه $k+1 \leq j \leq i-1$ در خط ۶ داشته باشیم $score(j) < bestscore$ (از آنجا که امتیازها منحصر به فرد هستند، می‌توانیم احتمال اینکه $score(j) = bestscore$ را نادیده بگیریم.) به عبارت دیگر، باید حالتی باشد که تمام مقادیر $score(k+1)$ تا $score(i-1)$ کوچکتر از $M(k)$ باشند؛ در عوض اگر مقادیر بزرگتری از $M(k)$ وجود داشته باشند، اندیس اولین مقداری که بزرگتر است را برمی‌گردانیم. از O_i بعنوان پیشامدی که هیچ یک از متقاضیان در موقعیّت‌های $k+1$ تا $i-1$ انتخاب نمی‌شوند، استفاده می‌کنیم. خوشبختانه، دو پیشامد B_i و O_i مستقل هستند. پیشامد O_i تنها به ترتیب نسبی مقادیر در موقعیتهای 1 تا $i-1$ بستگی دارد، در حالی که B_i فقط به اینکه آیا مقدار در موقعیت i بزرگتر از تمام مقادیر 1 تا $i-1$ است یا خیر، بستگی دارد. ترتیب موقعیت‌های 1 تا $i-1$ تأثیری بر اینکه i بزرگتر از تمام آنها باشد نمی‌گذارد، و مقدار i تأثیری بر ترتیب موقعیتهای 1 تا $i-1$ نمی‌گذارد. بنابراین با توجه به استقلال B_i و O_i داریم

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\}\Pr\{O_i\} \,.$$

احتمال $Pr\{B_i\}$ به وضوح $1/n$ است، زیرا احتمال قرار داشتن ماکزیمم در هر یک از n موقعیت یکسان است. برای اینکه پیشامد O_i رخ دهد، مقدار ماکزیمم در موقعیت‌های 1 تا $i-1$ باید در یکی از k موقعیت اول باشد، و احتمال قرار داشتن آن در هر یک از این $i-1$ موقعیت یکسان است. در نتیجه، $Pr\{O_i\}=k/(i-1)$ و $Pr\{S_i\}=k/(n(i-1))$. با استفاده از معادله (۵.۱۳) داریم

$$\begin{aligned}\Pr\{S\} &= \sum_{i=k+1}^{n} \Pr\{S_i\} \\ &= \sum_{i=k+1}^{n} \frac{k}{n(i-1)}\end{aligned}$$

$$= \frac{k}{n}\sum_{i=k+1}^{n}\frac{1}{i-1}$$
$$= \frac{k}{n}\sum_{i=k}^{n-1}\frac{1}{i}\,.$$

بوسیله انتگرال تقریب می‌زنیم تا این حاصلجمع را از بالا و پایین محدود کنیم[1]، داریم

$$\int_{k}^{n}\frac{1}{x}\,dx \le \sum_{i=k}^{n-1}\frac{1}{i} \le \int_{k-1}^{n-1}\frac{1}{x}\,dx\,.$$

با حل این انتگرالهای معیّن، حدود زیر را بدست می‌آوریم

$$\frac{k}{n}(\ln n - \ln k) \le \Pr\{S\} \le \frac{k}{n}(\ln(n-1) - \ln(k-1))\,,$$

که یک حد نسبتاً قوی برای $Pr\{S\}$ فراهم می‌کند. چون می‌خواهیم احتمال موفقیّت خود را ماکزیمم کنیم، اجازه دهید تا روی انتخاب مقدار k که حد پایین روی $Pr\{S\}$ را ماکزیمم می‌کند متمرکز شویم. (علاوه بر این، عبارت حد پایین برای ماکزیمم کردن آسانتر از عبارت حد بالا است.) با مشتق گرفتن از عبارت $(k/n)(\ln n-\ln k)$ نسبت به k، بدست می‌آوریم

$$\frac{1}{n}(\ln n - \ln k - 1)\,.$$

با مساوی قرار دادن این مشتق با 0 ، مشاهده می‌کنیم که حد پایین روی احتمال هنگامیکه $\ln k=\ln n-1=\ln(n/e)$ یا بطور معادل هنگامیکه $k=n/e$، ماکزیمم می‌شود. بنابراین اگر استراتژی خود را با $k=n/e$ پیاده سازی کنیم، با احتمال حداقل $1/e$ موفق خواهیم شد.

تمرین‌ها

۵.۴-۱ چند نفر باید در یک اتاق باشند تا احتمال اینکه فردی دارای روز تولد یکسانی با شما باشد، حداقل $1/2$ باشد؟ چند نفر باید باشند تا احتمال اینکه حداقل دو نفر در روز 4 جولای بدنیا آمده باشند،

---

۱ - اگر $f(k)$ یک تابع نزولی یکنواخت باشد، آنگاه داریم

$$\int_{m}^{n+1} f(x)\,dx \le \sum_{k=m}^{n} f(k) \le \int_{m-1}^{n} f(x)\,dx\,.$$

بزرگتر از $1/2$ باشد؟

۵.۴-۲ فرض کنید توپها داخل b جعبه پرتاب می‌شوند. هر پرتاب مستقل است و احتمال اینکه یک توپ در هر جعبه قرار گیرد، برابر است. تعداد مورد انتظار پرتابهای توپها قبل از اینکه حداقل یکی از جعبه‌ها شامل دو توپ باشد چیست؟

۵.۴-۳* برای تحلیل تناقض نمای روز تولد، آیا مهم است که روزهای تولد به طور متقابل مستقل باشند، یا استقلال جفت به جفت کافی است؟ پاسخ خود را توجیح کنید.

۵.۴-۴* چند نفر باید به یک مهمانی دعوت شوند تا احتمال اینکه سه نفر روز تولد یکسانی داشته باشند زیاد شود؟

۵.۴-۵* احتمال اینکه یک k- رشته روی یک مجموعه با اندازه n واقعاً یک k - جایگشت باشد چیست؟ این سؤال چگونه به تناقض نمای روز تولد مربوط می‌شود؟

۵.۴-۶* فرض کنید n توپ داخل n جعبه پرتاب می‌شوند، بطوریکه هر پرتاب مستقل است و احتمال اینکه یک توپ در هر جعبه قرار گیرد برابر است. تعداد مورد انتظار جعبه‌های خالی چیست؟ تعداد مورد انتظار جعبه‌ها با دقیقاً یک توپ چیست؟

۵.۴-۷* حد پایین روی طول توالی را قوی کنید، با نشان دادن اینکه در n پرتاب یک سکه نا اریب، احتمال اینکه هیچ توالی طولانی‌تر از $lgn-2lglgn$ شیر متوالی رخ نداده باشد، کمتر از $1/n$ است.

مسائل

۱-۵ شمارش احتمالی

با یک شمارنده b- بیتی، می‌توانیم به طور عادی تنها تا 2^b-1 بشماریم. با **شمارش احتمالی** *R.Morris* می‌توانیم تا مقدار بسیار بزرگتری به قیمت از دست دادن اندکی دقّت بشماریم.

فرض می‌کنیم برای $i=0, 1, 2, \ldots, 2^b-1$، مقدار شمارنده یعنی i شمارش n_i را نشان می‌دهد، که n_i یک توالی صعودی از مقادیر نامنفی را تشکیل می‌دهد. فرض می‌کنیم که مقدار اولیه شمارنده 0 است، که شمارش $n_0=0$ را نشان می‌دهد. عمل *INCREMENT* روی شمارنده‌ای که شامل مقدار i است در یک روند احتمالی کار می‌کند. اگر $i=2^b-1$، آنگاه خطای سر ریز گزارش می‌شود. در غیر این صورت شمارنده با احتمال $1/(n_{i+1}-n_i)$ به اندازه 1 واحد افزایش می‌یابد، و با احتمال $1/1(n_{i+1}-n_i)$ بدون تغیر باقی می‌ماند.

اگر برای تمام $i\geq0$ انتخاب کنیم $n_i=i$ آنگاه شمارنده یک شمارنده عادی است. اگر به ازای $i>0$ یا $n_i=F_i$ (iامین عدد فیبوناچی - بخش ۳.۲ را ملاحظه نمائید)، انتخاب کنیم $n_i=2^{i-1}$، وضعیت‌های جالب‌تری بوجود می‌آیند.

برای این مسئله، فرض کنید n^b_{2-1} آنقدر بزرگ است که احتمال خطای سر ریز ناچیز می‌باشد.

a. نشان دهید که مقدار مورد انتظار نشان داده شده بوسیله شمارنده بعد از انجام n عمل *INCREMENT* دقیقاً n است.

b. تحلیل واریانس شمارشی که بوسیله شمارنده نمایش داده می‌شود، به توالی n_i بستگی دارد. اجازه دهید یک حالت ساده را در نظر بگیریم: به ازای همه $i \geq 0$ ، $n_i = 100i$. واریانس مقدار نمایش داده شده بوسیله ثبّات بعد از اجرای n عمل *INCREMENT* را تخمین بزنید.

۵-۲ جستجوی یک آرایه نامرتب

این مسئله سه الگوریتم برای جستجوی مقدار x در یک آرایه نامرتب A شامل n عنصر را بررسی می‌کند.

استراتژی تصادفی زیر را در نظر بگیرید: یک اندیس تصادفی i از داخل A انتخاب کنید. اگر $A[i]=x$ آنگاه به کار خود خاتمه می‌دهیم؛ در غیر این صورت، جستجو را با انتخاب یک اندیس تصادفی جدید از داخل A ادامه می‌دهیم. انتخاب اندیس‌های تصادفی از داخل A را تا هنگامیکه یک اندیس j پیدا کنیم بطوریکه $A[j]=x$ یا تا وقتیکه هر عنصر A را بررسی کرده باشیم ادامه می‌دهیم. توجه کنید که هر بار از کل مجموعه اندیس‌ها انتخاب را انجام می‌دهیم، لذا ممکن است یک عنصر را بیش از یکبار بررسی کنیم.

a. شبه کدی برای روال *RANDOM-SEARCH* جهت پیاده سازی استراتژی فوق بنویسید. مطمئن شوید الگوریتم شما وقتی که همه اندیس‌های داخل A انتخاب شده باشند خاتمه می‌یابد.

b. فرض کنید دقیقاً یک اندیس i وجود دارد بطوریکه $A[i]=x$ تعداد مورد انتظار اندیس‌های داخل A که باید قبل از اینکه x پیدا شود و *RANDOM-SEARCH* خاتمه یابد انتخاب شده باشند چیست؟

c. با تعمیم حل قسمت (b)، فرض کنید که $k \geq 1$ اندیس i وجود دارند بطوریکه $A[i]=x$ تعداد مورد انتظار اندیس‌های داخل A که باید قبل از اینکه x پیدا شود *RANDOM-SEARCH* خاتمه یابد انتخاب شده باشند، چیست؟ پاسخ شما باید تابعی از n و k باشد.

d. فرض کنید که هیچ اندیس i ی وجود ندارد بطوریکه $A[i] = x$ تعداد مورد انتظار اندیس‌های داخل A که باید قبل از اینکه همه عناصر A بررسی شده باشند و *RANDOM-SEARCH* خاتمه یابد انتخاب شده باشند، چیست؟

اکنون الگوریتم جستجوی خطی جبری را در نظر بگیرید، که با *DETERMINISTIC-SEARCH* به آن اشاره می‌کنیم. این الگوریتم، A را برای x به ترتیب با در نظر گرفتن $A[1]$, $A[2]$, $A[3]$, . . ., $A[n]$ تا زمانیکه $A[i]=x$ پیدا شود یا به انتهای آرایه برسد جستجو می‌کند. فرض کنید که احتمال همه جایگشت‌های ممکن آرایه ورودی برابر است.

e. فرض کنید دقیقاً یک اندیس i وجود دارد بطوریکه $A[i]=x$. زمان اجرای مورد انتظار *DETERMINISTIC-SEARCH* چیست؟ زمان اجرای *DETERMINISTIC-SEARCH* در بدترین حالت چیست؟

f. با تعمیم حلّ قسمت *(e)*، فرض کنید که $k\geq 1$ اندیس i وجود دارند بطوریکه $A[i]=x$. زمان اجرای مورد انتظار *DETERMINISTIC-SEARCH* چیست؟
زمان اجرای *DETERMINISTIC-SEARCH* در بدترین حالت چیست؟
پاسخ شما باید تابعی از n و k باشد.

g. فرض کنید که هیچ اندیس i ای وجود ندارد بطوریکه $A[i]=x$. زمان اجرای مورد انتظار *DETERMINISTIC-SEARCH* چیست؟ زمان اجرای *DETERMINISTIC-SEARC* در بدترین حالت چیست؟

در نهایت، الگوریتم تصادفی *SCRAMBLE-SEARCH* را در نظر بگیرید که ابتدا آرایه ورودی را بصورت تصادفی جایگشت می‌کند و سپس جستجوی خطی جبری ارائه شده در فوق را روی آرایه جایگشت شده حاصل اجرا می‌کند.

h. k را تعداد اندیس‌های i بطوریکه $A[i]=x$ در نظر بگیرید. زمان اجرا در بدترین حالت و زمان اجرای مورد انتظار *SCRAMBLE-SEARCH* برای حالت‌هایی که در آن‌ها $k=0$ و $k=1$ ارائه دهید. حلّ خود را برای مدیریت حالتی که در آن $k\geq 1$، تعمیم دهید.

کدام یک از این سه الگوریتم جستجو را استفاده خواهید کرد؟ پاسخ خود را توضیح دهید.

II

مرتب‌سازی و شاخص‌های آمار ترتیبی

مقدمه

این قسمت، الگوریتم‌های متعددی را بیان می‌کند که ***مسئله مرتب سازی*** زیر را حل می‌کنند:

ورودی:یک توالی از nعدد $<a_1, a_2, \ldots, a_n>$.

خروجی: یک جایگشت (ترتیب دوباره) $<a'_1, a'_2, \ldots, a'_n>$ از توالی ورودی، به طوری که $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

توالی ورودی معمولاً یک آرایهٔ n- عنصری است، گر چه ممکن است به شیوه‌های دیگری مانند یک لیست پیوندی نمایش داده شود.

ساختار داده‌ها

در عمل، اعدادی که باید مرتّب شوند بندرت مقادیر منفردی هستند. معمولاً هر یک از آنها قسمتی از یک مجموعه داده بنام **رکورد** می‌باشند. هر رکورد شامل یک***کلید*** است که همان مقداری است که باید مرتّب شود. باقیماندهٔ رکورد از ***داده‌های فرعی*** که معمولاً در اطراف کلید هستند تشکیل شده است. در عمل هنگامی که یک الگوریتم مرتّب سازی، اعداد را جایگشت می‌کند باید داده‌های فرعی را نیز جایگشت کند. اگر هر رکورد شامل داده‌های فرعی زیادی باشد به منظور مینیمم کردن انتقال داده‌ها اغلب یک آرایه از اشاره‌گرها به رکوردها را به جای خود رکوردها جایگشت می‌کنیم.

به عبارتی، جزئیات پیاده‌سازی ،یک الگوریتم را از یک برنامهٔ کامل مشخص می‌کنند. این که اعداد منفرد یا رکودهای بزرگی که شامل اعداد هستند را مرتّب کنیم، به روشی که روال مرتّب‌سازی ترتیب مرتّب را تعیین می‌کنیم ربط ندارد. بنابراین هنگامی که به مسئله مرتب‌سازی می‌پردازیم، معمولاً فرض می‌کنیم که ورودی تنها از اعداد تشکیل شده است. در نتیجه، ترجمه یک الگوریتم برای مرتب‌سازی اعداد به یک برنامه برای مرتب‌سازی رکوردهای ساده و صریح است، اگر چه در یک وضعیت خاص ممکن است جزئیات دیگری وجود داشته باشد که کار برنامه‌سازی واقعی را دچار مشکل کنند.

چرا مرتب‌سازی

بسیاری از دانشمندان علوم کامپیوتر مرتب‌سازی را به عنوان بنیادی‌ترین مسأله در مطالعه

الگوریتم‌هادر نظر می‌گیرند. دلایل مختلفی وجود دارد:

- برخی اوقات نیاز به مرتب‌سازی اطلاعات در یک کاربرد، ذاتی است. به عنوان مثال، به منظور آماده کردن شرح حال مشتریان، بانکها نیاز به مرتّب‌سازی چکها بر اساس شماره چک دارند.
- الگوریتم‌ها اغلب از مرتب‌سازی به عنوان یک زیر روال کلیدی استفاده می‌کنند. به عنوان مثال، برنامه‌ای که اشیاء گرافیکی را که بالای همدیگر لایه بندی شده‌اند پردازش می‌کند، ممکن است مجبور باشد اشیاء را بر طبق رابطه ذکر شده مرتب کند تا بتواند اشیاء را از پایین با بالا رسم کند. الگوریتم‌های متعددی را در این کتاب مشاهده خواهیم کرد که از مرتّب‌سازی به عنوان یک زیر روال استفاده می‌کنند.
- الگوریتم‌های مرتب‌سازی متعددی وجود دارند و این الگوریتم‌ها از مجموعه قدرتمندی از تکنیک‌ها استفاده می‌کنند. در حقیقت، تکنیک‌های مهم بسیاری که در طی طراحی الگوریتم استفاده می‌شوند، در بدنه الگوریتم‌های مرتّب‌سازی که در این سال‌ها توسعه یافته‌اند، ارائه شده‌اند. به همین جهت، مرتّب‌سازی از قدیم یکی از مسائل جالب بوده است.
- مرتّب‌سازی مسأله‌ای است که برای آن می‌توانیم یک حد پایین کلی تعیین کنیم. (همان طور که در فصل ۸ انجام خواهیم داد). بهترین حدود بالا به طور مجانبی با حد پایین همخوانی دارند و بنابراین می‌دانیم که الگوریتم‌های مرتّب‌سازی به صورت مجانبی بهینه هستند. علاوه بر این، می‌توانیم از حد پایین مرتّب‌سازی برای اثبات حدود پایین مسائل مشخص دیگر استفاده کنیم.
- بسیاری از جنبه‌های مهندسی در هنگام پیاده‌سازی الگوریتم‌های مرتّب‌سازی پیش می‌آیند. سریع‌ترین برنامه مرتب‌سازی برای یک شرایط خاص ممکن است به عوامل زیادی بستگی داشته باشد، مانند دانش قبلی در مورد کلیدها و داده‌های وابسته، سلسله مراتب حافظه (حافظه مجازی و پنهان) کامپیوترهای میزبان، و محیط نرم‌افزاری. بهتر است بسیاری از این موارد را در سطح الگوریتمی بررسی کرده و کد را پیچیده نکنیم.

الگوریتم‌های مرتّب‌سازی

دو الگوریتم که n عدد حقیقی را مرتب می‌کنند، در فصل ۲ معرفی کردیم. مرتّب‌سازی درجی در بدترین حالت، زمان $\Theta(n^2)$ را صرف می‌کند. هر چند، به دلیل این که حلقه‌های داخلی آن فشرده هستند، برای ورودی‌های با اندازه کوچک یک الگوریتم مرتّب‌سازی درجا *(inplace)* است. (به یاد آورید که یک الگوریتم تنها در صورتی درجا مرتب می‌کند که از قبل تعداد ثابتی از اعضای آرایه ورودی، خارج از آرایه ذخیره شده باشند.) مرتّب‌سازی ادغام زمان اجرای مجانبی بهتری دارد، $\Theta(n\lg n)$، اما روال *MERGE* که در مرتّب‌سازی ادغام به کار می‌رود، به صورت درجا عمل نمی‌کند.

در این قسمت، دو الگوریتم دیگر را معرفی خواهیم کرد که اعداد حقیقی دلخواه را مرتب می‌کنند.

مرتّب‌سازی *heap* که در فصل ۶ معرفی می‌شود n عدد را در زمان $O(nlgn)$ به صورت درجا مرتب می‌کند. این روش از یک ساختمان داده مهم استفاده می‌کند که *heap* نام دارد و با آن می‌توانیم یک صف اولویت را نیز پیاده‌سازی کنیم.

مرتّب‌سازی سریع در فصل ۷ نیز n عدد را به صورت درجا مرتب می‌کند، اما زمان اجرای آن در بدترین حالت $\Theta(n^2)$ است. گر چه زمان اجرای آن در حالت میانگین $\Theta(nlgn)$ است، و به صورت کلی مرتّب‌سازی *heap* را انجام می‌دهد. همانند مرتّب‌سازی درجی، مرتّب‌سازی سریع کد فشرده‌ای دارد، بنابراین ضریب ثابت پنهان در زمان اجرای آن کوچک است. این الگوریتم برای مرتّب‌سازی آرایه‌های بزرگ ورودی، یک الگوریتم رایج می‌باشد.

مرتّب‌سازی درجی، مرتّب‌سازی ادغام، مرتّب‌سازی *heap* و مرتّب‌سازی سریع همگی مرتّب‌سازی‌های مقایسه‌ای هستند: این الگوریتم‌ها با مقایسه عناصر، ترتیب مرتب شدهٔ آرایه ورودی را تعیین می‌کنند. فصل ۸ با معرفی مدل درخت تصمیم، به منظور مطالعه محدودیت‌های اجرایی مرتّب‌سازی‌های مقاسیه‌ای آغاز می‌شود. با استفاده از این مدل، حد پایین $\Omega(nlgn)$ را در بدترین حالت برای زمان اجرای هر مرتّب‌سازی مقایسه‌ای روی n ورودی اثبات می‌کنیم. بنابراین نشان می‌دهیم که مرتّب‌سازی‌های *heap* ادغام، مرتّب‌سازی‌های مقایسه‌ای هستند که به طور مجانبی بهینه‌اند.

فصل ۸ نشان خواهد داد اگر بتوانیم اطلاعاتی دربارهٔ ترتیب مرتب شده ورودی به طریقی غیر از مقایسه اعضا جمع آوری کنیم می‌توانیم حد پایین $\Omega(nlgn)$ را کاهش دهیم.

به عنوان مثال، الگوریتم مرتّب‌سازی شمارشی فرض می‌کند که اعداد ورودی در مجموعه $\{1, 2, . . ., k\}$ قرار دارند. با استفاده از اندیس گذاری آرایه به عنوان وسیله‌ای جهت تعیین ترتیب نسبی، مرتّب‌سازی شمارشی می‌تواند n عدد را در زمان $\Theta(k+n)$ مرتب نماید. بنابراین وقتی $k = O(n)$ مرتّب‌سازی شمارشی در زمانی اجرا شود که در اندازه آرایه ورودی خطی است. یک الگوریتم مربوط به نام مرتّب‌سازی مبنایی می‌تواند برای توسعه دامنه مرتّب‌سازی شمارشی مورد استفاده قرار گیرد.

اگر n عدد صحیحی برای مرتب کردن وجود داشته باشد، هر عدد صحیح d رقم داشته باشد، و هر رقم در مجموعه $\{1, 2, . . . , k\}$ قرار داشته باشد آنگاه مرتّب‌سازی مبنایی می‌تواند اعداد را در زمان $\Theta(d(n+k))$ مرتب کند. وقتی d ثابت و k برابر $O(n)$ است، مرتّب‌سازی مبنایی در زمان خطی اجرا می‌شود. الگوریتم سوم، مرتّب‌سازی پیمانه‌ای، به دانش توزیع احتمالات اعداد در آرایه ورودی نیاز دارد. این الگوریتم می‌تواند n عدد حقیقی را که به طور یکنواخت در بازه نیمه باز $[0,1)$ توزیع شده‌اند در حالت میانگین در زمان $O(n)$ مرتب نماید.

شاخص‌های آماری ترتیبی

i امین شاخص آماری ترتیبی یک مجموعه از n عدد، i امین عدد کوچک در مجموعه می‌باشد. البته می‌توانیم i امین شاخص آماری ترتیبی را با استفاده از مرتّب‌سازی ورودی واندیس گذاری i امین عنصر خروجی انتخاب کنیم. همان طور که حد پایین ثابت شده در فصل ۸ نشان می‌دهد بدون هیچ فرضی در مورد توزیع ورودی، این روش در زمان $\Omega(nlgn)$ اجرا می‌شود.

در فصل ۹ نشان می‌دهیم که می‌توانیم i امین عنصر کوچک را در زمان $O(n)$ پیدا کنیم، حتی وقتی عناصر، اعداد حقیقی دلخواه هستند. الگوریتمی را با شبه کد فشرده که در بدترین حالت در زمان ممکن $\Theta(n^2)$ و در حالت میانگین در زمان خطی اجرا می‌شود ارائه می‌کنیم. همچنین الگوریتم پیچیده‌تری را بیان می‌کنیم که در بدترین حالت در زمان $O(n)$ اجرا می‌شود.

پس زمینه

اگر چه بیشتر این فصل به ریاضیات پیچیده‌ای متّکی نیست، برخی بخش‌ها به استدلال‌های ریاضی نیاز دارند. به ویژه تحلیل حالت میانگین مرتّب‌سازی سریع، مرتّب‌سازی پیمانه‌ای و الگوریتم شاخص‌های آماری ترتیبی از احتمال استفاده می‌کنند، موارد تحلیل احتمال و الگوریتم‌های تصادفی در فصل ۵ بررسی شده‌اند. تحلیل زمان خطی الگوریتم در بدترین حالت برای شاخص‌های آمار ترتیبی، شامل ریاضیاتی است که تا حدی پیچیده‌تر نسبت به دیگر تحلیل‌های بدترین حالت در این بخش می‌باشد.

۶ مرتّب‌سازی heap

در این فصل، الگوریتم مرتّب‌سازی دیگری را معرفی می‌کنیم. زمان اجرای مرتّب‌سازی *heap* برخلاف مرتّب‌سازی درجی، همانند مرتّب‌سازی ادغام برابر با $O(nlgn)$ است. نحوه مرتّب‌سازی آن همانند مرتّب‌سازی درجی و برخلاف مرتب سازی ادغام به صورت درجا است: در هر زمان تنها تعداد ثابتی از عناصر آرایه خارج از آرایه ورودی ذخیره می‌شوند. مرتّب‌سازی *heap* خصوصیات بهتر دو الگوریتم مرتّب‌سازی که قبلاً بحث شد را ترکیب می‌کند.

همچنین مرتّب‌سازی *heap* تکنیک دیگری از طراحی الگوریتم را معرفی می‌کند: استفاده از یک ساختمان داده، که در این حالت به آن *heap* می‌گوییم جهت مدیریت اطلاعات در طول اجرای الگوریتم. ساختمان داده *heap* تنها برای مرتّب‌سازی *heap* مفید نیست بلکه یک صف اولویت کارآمد نیز ایجاد می‌کند. ساختمان داده *heap* در الگوریتم‌های فصل‌های بعدی دوباره ظاهر خواهد شد.

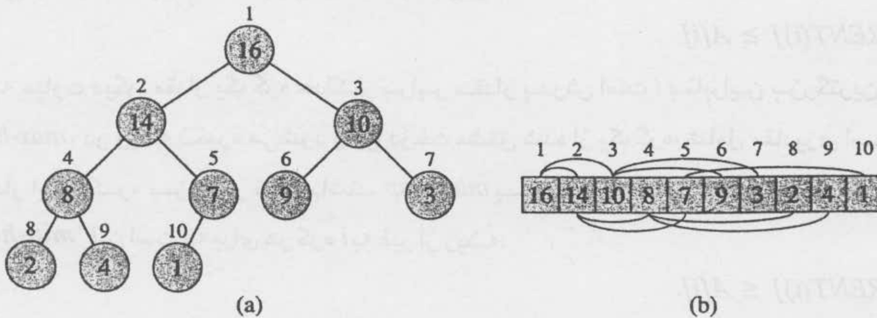
توجه داریم که واژه "*heap*" در اصل در ارتباط با مرتّب‌سازی *heap* ابداع شد اما از آن زمان جهت اشاره به "حافظهٔ داده‌های زائد جمع آوری شده" بکار می‌رود. مانند آنچه زبان‌های برنامه نویسی *lisp* و *Java* فراهم می‌کنند ساختمان داده *heap*، حافظه داده‌های زائد جمع آوری شده نیست، و هر گاه در این کتاب به *heap*ها اشاره نمودیم، منظورمان ساختاری است که در این بخش تعریف می‌شود.

۶.۱ *Heap*ها

همان طور که در شکل ۶.۱ نشان داده شده است ساختمان داده *heap*(دودویی) [1] یک شیء آرایه‌ای است که می‌تواند به شکل یک درخت دودویی کاملاً کامل در نظر گرفته شود. هر گره درخت متناظر با یک عضو از آرایه است که مقدار را در گره ذخیره می‌کند و درخت به طور کامل در همه سطوح پر می‌شود. به جز احتمالاً پایین‌ترین سطح، که از چپ تا یک نقطه پر شده است آرایه A که *heap* را نشان می‌دهد شیئی با دو خصوصیت است: $length[A]$، تعداد عناصر آرایه است و $heap\text{-}size[A]$ تعداد عناصر *heap* که در آرایه A ذخیره شده‌اند می‌باشد. به عبارت دیگر اگر چه $A[1 \,..\, length[A]]$ ممکن

1.(binary) heap

است شامل اعداد معتبری باشد، اما هیچ عنصری بعد از *A[heap-size[A]]* که *heap-size[A] ≤ length[A]* یک عضو *heap* نمی‌باشد.



شکل ۶.۱ یک *max-heap* که در *(a)* به صورت یک درخت دودویی و در *(b)* به صورت یک آرایه، مشاهده می‌شود. عدد داخل دایره در هر گره درخت مقدار ذخیره شده در آن گره است. عدد بالای هر گره اندیس متناظر در آرایه است. در بالا و پایین آرایه خطوطی وجود دارند که ارتباط پدر - فرزند را نشان می‌دهند؛ پدرها همیشه در سمت چپ فرزندانشان قرار دارند. درخت ارتفاعی برابر ۳ دارد ؛ گره با اندیس *4* (با مقدار *8*) ارتفاعی برابر یک دارد.

ریشه درخت، *A[1]* است و با داشتن اندیس *i* مربوط به یک گره اندیس‌های پدر *PARENT(i)*، فرزند چپ *LEFT(i)* و فرزند راست *RIGHT(i)* آن به سادگی می‌توانند محاسبه گردند.

```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

در اغلب کامپیوترها، روال *LEFT* می‌تواند *2i* را در یک دستور به سادگی با شیفت نمایش دودویی *i* یک مکان به سمت چپ، محاسبه کند. به طور مشابه، روال *RIGHT* می‌تواند به سرعت *2i+1* را با شیفت نمایش دودویی *i* یکی به چپ و اضافه کردن یک به بیت کم ارزش، محاسبه کند. روال *PARENT* می‌تواند $\lfloor i/2 \rfloor$ را با شیفت *i*، یک بیت به راست بدست آورد. در یک پیاده‌سازی مناسب مرتّب‌سازی

heap ، این سه روال اغلب به شکل ماکرو، یا روالهای درون خطی پیاده‌سازی می‌شوند.

دو نوع *heap* دودویی وجود دارد: *max-heap*ها و *min-heap*ها. در هر دو نوع، مقادیر درون گره‌ها ویژگی *heap* را ارضاء می‌کنند که ویژگی‌های هر کدام به نوع *heap* بستگی دارند. در یک *max-heap*، ویژگی *max-heap* این است که برای هر گره i به جز ریشه:

$$A[PARENT(i)] \geq A[i]$$

به عبارت دیگر، مقدار یک گره حداکثر برابر مقدار پدرش است ؛ بنابراین بزرگترین عضو *max-heap*، در ریشه ذخیره می‌شود و زیردرخت مشتق شده از یک گره، شامل مقادیری است که از مقدار این گره بزرگ‌تر نمی‌باشند. *min-heap* به صورت عکس شکل می‌گیرد؛ **ویژگی *min-heap***[1] این است که برای هر گره i به غیر از ریشه:

$$A[PARENT(i)] \leq A[i].$$

کوچک ترین عضو در *min-heap* در ریشه قرار دارد.

برای الگوریتم مرتّب‌سازی *heap*، از *max-heap* استفاده می‌کنیم. *min-heap*ها عموماً در صف‌های اولویت استفاده می‌شوند که از بخش ۶.۵ در مورد آنهابحث خواهیم کرد. در مشخص نمودن این که برای کاربرد خاصی به *max-heap* نیاز داریم یا *min-heap* دقت خواهیم نمود و زمانی که ویژگی‌هایی برای *max-heap* و *min-heap* به کار برده می‌شوند، صرفاً از واژه *heap* استفاده می‌کنیم.

با در نظر گرفتن *heap* به عنوان یک درخت، ***ارتفاع***[2] یک گره در *heap* را تعدا یال‌ها در طولانی‌ترین مسیر ساده به سمت پایین، از گره تا یک برگ تعریف می‌کنیم، و ارتفاع *heap* را برابر ارتفاع ریشه‌اش تعریف می‌کنیم. از آنجا که یک *heap* با n عنصر مبنای یک درخت دودویی کامل است، ارتفاع آن برابر $\Theta(lgn)$ است (تمرین ۲- ۶.۱ را ملاحظه نمائید.) خواهیم دید که اعمال اصلی روی *heap* از نظر زمانی حداکثر متناسب با ارتفاع درخت اجرا می‌شوند و بنابراین زمان $O(lgn)$ را صرف می‌کنند. ادامه این فصل، پنج روال اصلی را ارائه می‌کند و نشان می‌دهد که این روال‌ها چگونه در یک الگوریتم مرتب‌سازی و یک ساختمان داده صف اولویت استفاده می‌شوند.

- روال *MAX-HEAPIFY* که در زمان $O(lgn)$ اجرا می‌شود کلید نگهداری ویژگی *max-heap* است.
- روال *BUILD-MAX-HEAP* که در زمان خطی اجرا می‌شود یک *max-heap* را از روی یک آرایه ورودی نامرتب تولید می‌کند.
- روال *HEAPSORT* که در زمان $O(nlgn)$ اجرا می‌شود به صورت درجا یک آرایه را مرتب می‌کند.
- روال‌های *MAX-HEAP-INSERT*، *HEAP-EXTRACT-MAX*، *HEAP-INCREASE-KEY* و *HEAP-MAXIMUM* که در زمان $O(lgn)$ اجرا می‌شوند. به ما امکان استفاده از ساختمان داده *heap* را به عنوان یک صف اولویت می‌دهند.

1.min-heap property

2.height

تمرین‌ها

۱- ۶.۱ تعداد حداکثر و حداقل اعضا در یک *heap* با ارتفاع *h* چقدر است؟

۲- ۶.۱ نشان دهید که یک *heap* با *n* عضو ارتفاع $\lfloor lgn \rfloor$ دارد.

۳- ۶.۱ نشان دهید که در هر زیردرخت یک *max-heap* ریشه زیردرخت، بزرگ‌ترین مقداری که در آن زیردرخت قرار می‌گیرد را در بر دارد.

۴- ۶.۱ در یک *max-heap* کوچک‌ترین عنصر در کجا ممکن است قرار داشته باشد، با این فرض که کلیه عناصر متمایز هستند؟

۵- ۶.۱ آیا آرایه‌ای که مرتب است یک *min-heap* است؟

۶- ۶.۱ آیا توالی زیر یک *max-heap* است؟ $<23, 17, 14, 6, 13, 10, 1, 5, 7, 12>$.

۷- ۶.۱ نشان دهید در نمایش آرایه‌ای ذخیره‌سازی یک *heap* با *n* عضو برگ‌ها گره هایی هستند که با اعداد $\lfloor n/2 \rfloor +1$ و $\lfloor n/2 \rfloor +2, \ldots, n$ اندیس گذاری شده‌اند.

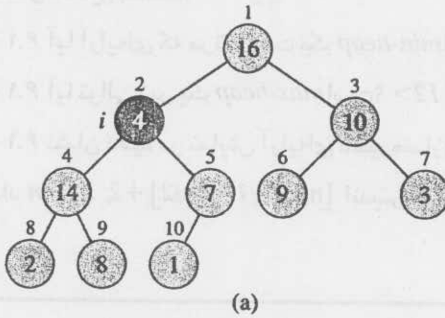
۶.۲ حفظ ویژگی heap

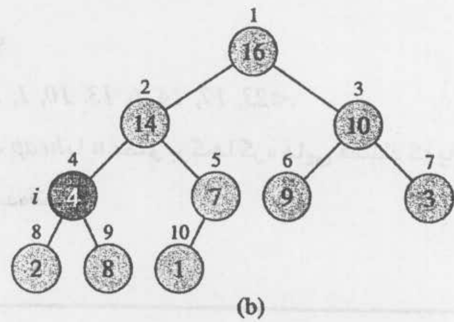
MAX-HEAPIFY یک زیرروال مهم برای دستکاری *max-heap*ها است. ورودی‌های آن آرایه *A* و اندیس *i* در آرایه است. زمانی که *MAX-HEAPIFY* فراخوانی می‌شود، فرض می‌شود که درخت‌های دودویی مشتق شده از *LEFT(i)* و *RIGHT(i)*، *max-heap* هستند، ولی عنصر *A[i]* ممکن است کوچک‌تر از فرزندانش باشد، بنابراین ویژگی *max-heap* مورد تخطی قرار می‌گیرد. وظیفه *MAX-HEAPIFY* این است که مقدار موجود در *A[i]* را در *max-heap* به پائین حرکت دهد تا زیردرخت مشتق شده از *i*، یک *max-heap* شود.

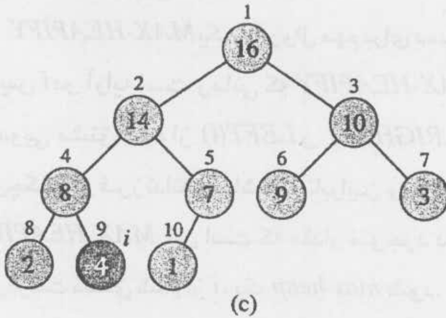
```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] and A[l] > A[i]
 4     then largest ← l
 5     else largest ← i
 6  if r ≤ heap-size[A] and A[r] > A[largest]
 7     then largest ← r
 8  if largest ≠ i
 9     then exchange A[i] ↔ A[largest]
10           MAX-HEAPIFY(A, largest)
```

شکل ۶.۲ عمل *MAX-HEAPIFY* را توضیح می‌دهد. در هر مرحله، بزرگ‌ترین عضو بین *A[RIGHT(i)]*، *A[LEFT(i)]* و *A[i]* مشخص می‌شود و اندیس آن در *largest* ذخیره می‌شود. اگر

$A[i]$ بزرگ‌ترین باشد، آنگاه زیردرخت مشتق شده از i ، یک *max-heap* است و روال پایان می‌یابد. در غیر این صورت یکی از دو فرزند بزرگترین عنصر را دارد و $A[i]$ با $A[largest]$ جا به جا می‌شود که باعث می‌شود گره i و فرزندانش، ویژگی *max-heap* را برقرار کنند، اما گره‌ای که با *largest* اندیس گذاری شده، اکنون مقدار اولیه $A[i]$ را دارد و بنابراین زیردرخت مشتق شده از *largest* ممکن است ویژگی *max-heap* را مورد تخطی قرار دهد. در نتیجه *MAX-HEAPIFY* بایستی به صورت بازگشتی برای آن زیردرخت فراخوانی شود.







شکل ۶.۲ عمل *MAX-HEAPIFY(A, 2)* زمانی که $heap\text{-}size[A] = 0$ (a) پیکربندی اولیه با $A[2]$ در گره $i = 2$، ویژگی *max-heap* را مورد تخطی قرار می‌دهد. چون از هر دو فرزندش بزرگ‌تر نیست ویژگی *max-heap* برای گره 2 در (b) با عوض کردن $A[2]$ با $A[4]$ بازیابی می‌شود، که در این صورت ویژگی *max-heap* برای گره 4 را به هم می‌زند. اکنون فراخوانی بازگشتی *MAX-HEAPIFY(A, 4)* دارای $i = 4$ است. بعد از جابه‌جایی $A[4]$ با $A[9]$ که در (c) نشان داده شده است گره 4 ثابت می‌شود و فراخوانی بازگشتی *MAX-HEAPIFY (A, 9)* هیچ تغییر دیگری را در ساختمان داده ایجاد نمی‌کند.

زمان اجرای *MAX-HEAPIFY* روی زیردرختی به اندازه n که از گره داده شده i مشتق شده برابر است با $\Theta(1)$ که برای تعیین رابطه میان عناصر $A[i]$، $A[LEFT(i)]$ و $A[RIGHT[i]]$ صرف می‌شود، به‌علاوه زمان اجرای *MAX-HEAPIFY* روی زیردرختی مشتق شده از یکی از فرزندان گره i

هر یک از زیردرختهای فرزندان دارای اندازه حداکثر $2n/3$ است - بدترین حالت هنگامی رخ می‌دهدکه آخرین سطر درخت دقیقاً نیمه پر باشد و بنابراین زمان اجرای *MAX-HEAPIFY* می‌تواند توسط رابطه بازگشتی زیر تعریف شود

$$T(n) \leq T(2n/3)+\Theta(1)$$

جواب این رابطه بازگشتی، با استفاده از حالت دوم قضیه اصلی (قضیه ۴.۱) برابر $T(n) = O(lgn)$ است. متناوباً می‌توانیم زمان اجرای *MAX-HEAPIFY* را روی گره‌ای با ارتفاع h به صورت $O(h)$ مشخص کنیم.

تمرین‌ها

۱- ۶.۲ با استفاده از شکل ۶.۲ به عنوان نمونه، عملکرد *MAX-HEAPIFY(A, 3)* را روی آرایه $A = <27, 17, 3, 16, 13, 15, 1, 2, 7, 12, 4, 8, 9, 0>$ نمایش دهید.

۲- ۶.۲ با شروع از روال *MAX-HEAPIFY*، شبه کدی برای روال *MAX-HEAPIFY(A, i)* بنویسید که دستکاری مشابهی را روی داده‌های *min-heap* انجام دهد. زمان اجرای *MAX-HEAPIFY* چطور با زمان اجرای *MIN-HEAPIFY* مقایسه می‌شود.

۳- ۶.۲ تأثیر فراخوانی *MAX-HEAPIFY(A, i)* زمانی که عنصر $A[i]$ بزرگ‌تر از فرزندانش است چیست؟

۴- ۶.۲ تأثیر فراخوانی *MAX-HEAPIFY(A, i)* برای $i > heap\text{-}size[A]/2$ چیست؟

۵- ۶.۲ برنامه *MAX-HEAPIFY* از نظر ضرایب ثابت کارآمد است، بجز احتمالاً فراخوانی بازگشتی در خط ۱۰ که ممکن است باعث شود بعضی کامپایلرها برنامه نامناسبی ایجاد کنند. یک *MAX-HEAPIFY* کارآمد بنویسید که از یک ساختار کنترلی تکراری (حلقه) به جای بازگشت استفاده می‌کند.

۶- ۶.۲ نشان دهید که زمان اجرای *MAX-HEAPIFY* روی یک *heap* با اندازه n در بدترین حالت برابر $\Omega(lgn)$ است. (*راهنمایی:* برای یک *heap* با n گره، مقادیری که باعث می‌شود *MAX-HEAPIFY* برای هر گره در مسیری از ریشه تا برگ به صورت بازگشتی فراخوانی شود را به دست آورد.)

۶.۳ ساختن یک heap

می‌توانیم برای تبدیل آرایه $A[1\ .\ .\ n]$ به یک *max-heap* که $n = length[A]$ از روال *MAX-HEAPIFY* به روش پایین به بالا استفاده کنیم.

با توجه به تمرین ۱- ۶.۱ عناصر زیرآرایه $A[(\lfloor n/2 \rfloor +1)\ .\ .\ n]$ همگی برگ‌های درخت هستند و

بنابراین هر کدام یک *heap* یک عنصری برای شروع است. روال *BUILD-MAX-HEAP* در راستای گره‌های باقیمانده درخت حرکت کرده و برای هر یک *MAX-HEAPIFY* را اجرا می‌کند.

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
```

شکل ۶.۳ مثالی از عملکرد *BUILD-MAX-HEAP* را نشان می‌دهد. برای این که نشان دهیم چرا *BUILD-MAX-HEAP* درست کار می‌کند از ثابت حلقه زیر استفاده می‌کنیم: در شروع هر تکرار حلقه *for* در خطوط ۳-۲، هر گره $i+1, i+2, \ldots, n$ ریشه یک *max-heap* است.

لازم است نشان دهیم که این ثابت قبل از اولین تکرار حلقه صحیح است، هر تکرار حلقه این ثابت را حفظ می‌کند و این ثابت ویژگی مفیدی را به وجود می‌آورد که: زمانی که حلقه پایان می‌یابد، درستی را نشان می‌دهد.

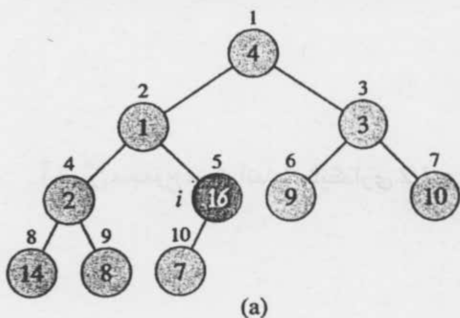
مقداردهی اولیه: قبل از اولین تکرار حلقه، $i=\lfloor n/2 \rfloor$ است. هر گره $\lfloor n/2 \rfloor+1, \lfloor n/2 \rfloor+2, \ldots, n$ یک برگ است و بنابراین ریشه یک *max-heap* کوچک‌تر می‌باشد.

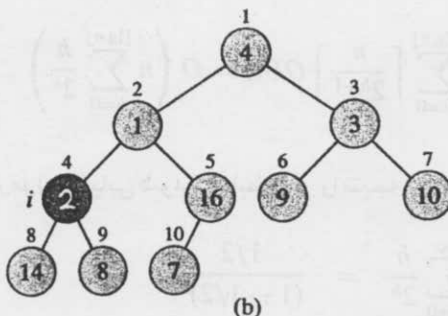
نگهداری: برای این که ببینیم هر تکرار، ثابت حلقه را حفظ می‌کند مشاهده می‌شود که فرزندان گره i زودتر از i شماره‌گذاری شده‌اند. بنابراین با استفاده از ثابت حلقه، هر دوی آنها ریشه‌های *max-heap* هستند. این دقیقاً شرایطی است که برای فراخوانی $MAX\text{-}HEAPIFY\ (A, i)$ نیاز است تا گره i را ریشه یک *max-heap* کند. علاوه بر این، فراخوانی *MAX-HEAPIFY* این ویژگی که گره‌های $i+1, i+2, \ldots, n$ همگی ریشه‌های *max-heap* هستند را حفظ می‌کند. کاهش دادن i در حلقه *for* ثابت حلقه را برای تکرار بعدی بازسازی می‌کند.

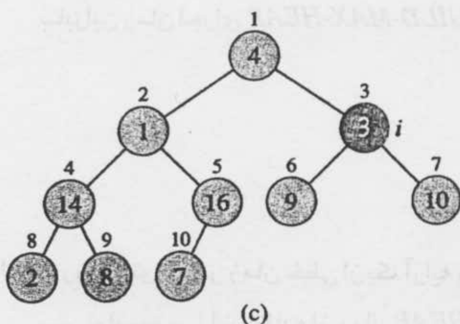
خاتمه: در پایان $i=0$ است. با توجه به ثابت حلقه، هر گره $1, 2, \ldots, n$ ریشه یک *max-heap* است به ویژه گره 1.

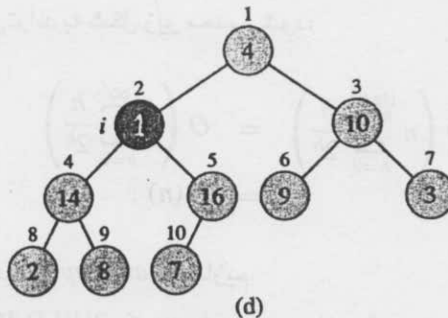
می‌توانیم یک حد بالای ساده برای زمان اجرای *BUILD-MAX-HEAP* به صورت زیر محاسبه کنیم. هر فراخوانی *MAX-HEAPIFY* زمان $O(lgn)$ را صرف می‌کند و $O(n)$ فراخوانی وجود دارد. بنابراین زمان اجرا $O(nlgn)$ است. این حد بالا، اگر چه صحیح است ولی دقیق نیست. با در نظر گرفتن این که زمان اجرای *MAX-HEAPIFY* در یک گره با ارتفاع گره در درخت تغییر می‌کند و ارتفاع اکثر گره‌ها کوچک است، می‌توان حد دقیق‌تری را نتیجه گرفت. تحلیل دقیق‌تر بر مبنای این ویژگی است که هر *heap* با n عنصر دارای ارتفاع (lgn) (به تمرین ۲-۶.۱ مراجعه کنید) و حداکثر $\lceil n/2^{h+1} \rceil$ گره با ارتفاع h است (به تمرین ۳-۶.۳ مراجعه کنید).

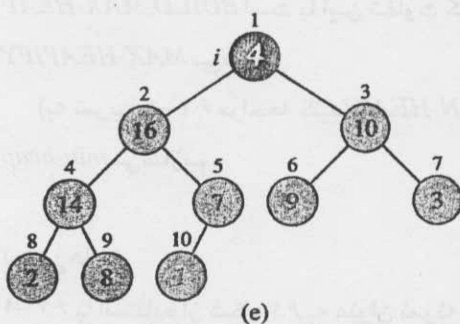
A	4	1	3	2	16	9	10	14	8	7

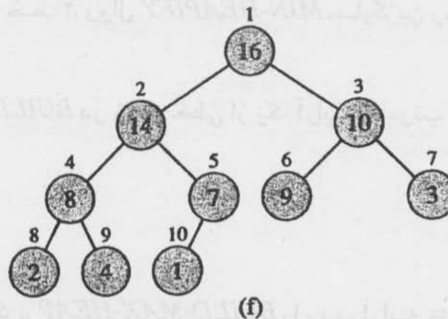












شکل ۶.۳ عملکرد *BUILD-MAX-HEAP*، نمایش ساختمان داده قبل از فراخوانی *MAX-HEAPIFY* در خط ۳ از *BUILD-MAX-HEAP* *(a)* آرایه ۱۰ عنصری *A* به عنوان ورودی و درخت دودویی که توسط آرایه *A* ارائه می‌شود. شکل نشان می‌دهد که اندیس حلقه یعنی i قبل از فراخوانی *MAX-HEAPIFY* به گره *5* اشاره می‌کند. *(b)* ساختمان داده‌ای که به عنوان نتیجه بدست می‌آید. اندیس حلقه، i، برای تکرار بعدی به گره *4* اشاره می‌کند. *(c)-(e)* نتیجه تکرار متوالی حلقه *for* در *BUILD-MAX-HEAP* مشاهده می‌کنید که هرگاه *MAX-HEAPIFY* روی یک گره فراخوانی می‌شود، هر دو زیردرخت گره *max-heap* هستند. *(f)* *max-heap* پس از اتمام *BUILD-MAX-HEAP*

زمان مورد نیاز *MAX-HEAPIFY* هنگامی که برای گره‌ای با ارتفاع h فراخوانی می‌شود $O(h)$ است. بنابراین می‌توانیم هزینه *BUILD-MAX-HEAP* را به صورت زیر بیان کنیم.

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) .$$

آخرین مجموع می‌تواند با جایگذاری $x=1/2$ در فرمول ارزیابی شود که رابطه زیر را نتیجه می‌دهد.

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

بنابراین زمان اجرای *BUILD-MAX-HEAP* می‌تواند به شکل زیر محدود شود:

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n) . \end{aligned}$$

از این رو، می‌توانیم در زمان خطی از یک آرایه نامرتب یک *max-heap* بسازیم.

می‌توانیم با استفاده از روال *BUILD-MIN-HEAP* یک *min-heap* بسازیم که مشابه *BUILD-MAX-HEAP* است با این تفاوت که در خط ۳، روال *MIN-HEAPIFY* جایگزین روال *MAX-HEAPIFY* می‌شود.

(به تمرین ۲- ۶.۲ مراجعه کنید). *BUILD-MIN-HEAP* در زمان خطی از یک آرایه نامرتب یک *min-heap* می‌سازیم .

تمرین‌ها

۱- ۶.۳ با استفاده از شکل ۶.۳ به عنوان نمونه، عملکرد *BUILD-MAX-HEAP* را روی آرایه $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$ نمایش دهید.

۲- ۶.۳ چرا اندیس حلقه i را در خط ۲ روال *BUILD-MAX-HEAP* به جای این که از 1 به $\lfloor length[A]/2 \rfloor$ افزایش دهیم از $\lfloor length[A]/2 \rfloor$ به 1 کاهش می‌دهیم؟

۳- ۶.۳ نشان دهید که در هر *heap* با n عنصر، حداکثر $\lceil n/2^{h+1} \rceil$ گره با ارتفاع h وجود دارد.

۶.۴ الگوریتم مرتّب‌سازی *heap*

الگوریتم مرتّب‌سازی *heap* برای ساختن یک *max-heap* روی آرایه ورودی *A[1 . . n]* که *n = length[A]* با استفاده از *BUILD-MAX-HEAP* آغاز می‌شود. از آنجا که بزرگ‌ترین عضو آرایه در ریشه یعنی *A[1]* ذخیره شده، می‌تواند با جابه‌جا شدن با *A[n]* در موقعیت صحیح نهایی خود قرار گیرد. اگر اکنون گره *n* را از *heap* حذف کنیم (با کم کردن یک واحد از *heap-size[A]*) می‌بینیم که *A[1 . . (n-1)]* می‌تواند به راحتی به یک *max-heap* تبدیل شود. فرزندان ریشه *max-heap* باقی خواهند ماند ولی عنصر ریشه جدید، ممکن است ویژگی *max-heap* را مورد تخطی قرار دهد. از این رو کل کاری که برای بازیابی ویژگی *max-heap* وجود دارد یک فراخوانی *MAX-HEAPIY(A, 1)* است که برای آرایه *A[1 . . (n-1)]* یک *max-heap* ایجاد می‌کند. سپس الگوریتم مرتّب‌سازی *heap* این روند را برای *max-heap* با اندازه *n-1* تا *heap* با اندازه *2* تکرار می‌کند. (برای ثابت دقیق حلقه تمرین ۲-۶.۴ را ملاحظه کنید.)

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3       do exchange A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          MAX-HEAPIFY(A, 1)
```

شکل ۶.۴ نمونه‌ای از مرتّب‌سازی *heap* بعد از این که *max-heap* در حالت اولیه ساخته شده است را نشان دهد. هر *max-heap* در ابتدای تکرار حلقه *for* در خطوط ۵-۲ نشان داده شده است.

روال *HEAPSORT* زمان *O(nlgn)* را صرف می‌کند چون فراخوانی *BUILD-MAX-HEAP* زمان *O(n)* و هر یک از *n-1* فراخوانی *MAX-HEAPIY* زمان *O(lgn)* را صرف می‌کند.

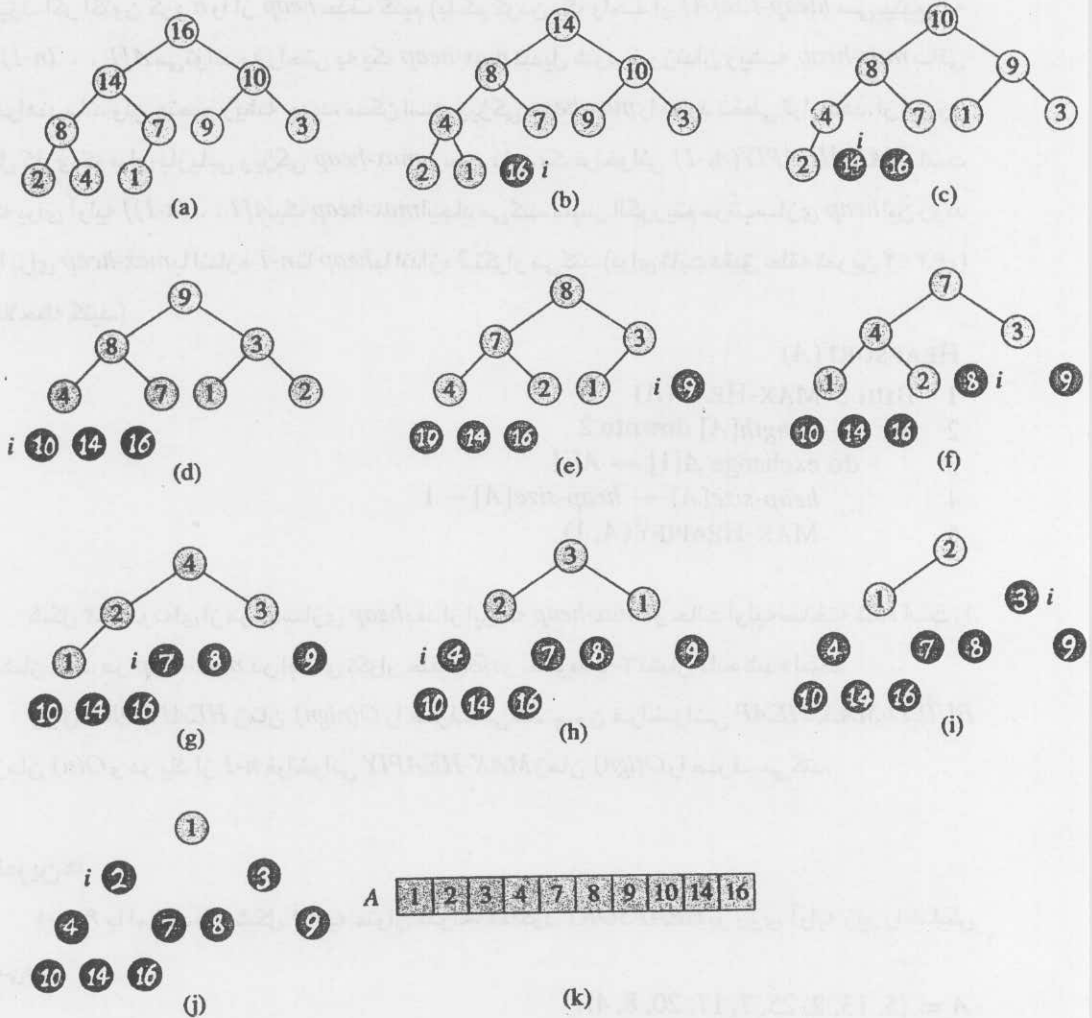
تمرین‌ها

۱-۶.۴ با استفاده از شکل ۶.۴ به عنوان نمونه، عملکرد *HEAPSORT* بر روی آرایه زیر را نمایش می‌دهد.

$A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle.$

۲-۶.۴ با استفاده از ثابت حلقه زیر، در مورد درستی *HEAPSORT* بحث کنید:

در ابتدای هر تکرار حلقه *for* در خطوط ۵-۲، زیر آرایه *A[1 . . i]* یک *max-heap* است که شامل *i* تا از کوچک‌ترین عناصر *A[1 . . n]* به صورت مرتب است و زیر آرایه *A[i+1 . . n]* شامل *n-i* تا از بزرگ‌ترین عناصر *A[1 . . n]* به صورت مرتب است.



شکل ۶.۴ عملکرد *HEAPSORT* *(a)* ساختمان *max-heap*، درست پس از این که توسط *BUILD-MAX-HEAP* ساخته می‌شود. *(b)-(i)* *max-heap* درست پس از هر فراخوانی *MAX-HEAPIFY* در خط ۵. مقدار *i* در آن زمان نشان داده شده. فقط گره هایی که روشن سایه زده شده‌اند در *heap* باقی می‌مانند. *(k)* آرایه *A* مرتب شدهٔ نهایی.

۳- ۶.۴ زمان اجرای مرتب‌سازی *heap* روی آرایه A با طول n که قبلاً با ترتیبی صعودی مرتب شده چیست؟ در ترتیب نزولی چگونه است؟

۴- ۶.۴ نشان دهید زمان اجرای مرتب‌سازی *heap* در بدترین حالت $\Omega(nlgn)$ است.

۵- ۶.۴ نشان دهید زمانی که همهٔ عناصر متفاوتند، زمان اجرای مرتّب‌سازی *heap* در بهترین حالت $\Omega(nlgn)$ است.

۶.۵ صف اولویت

مرتب‌سازی *heap* الگوریتم بسیار خوبی است، ولی یک پیاده‌سازی خوب مرتّب‌سازی سریع، که در فصل ۷ ارائه شده است معمولاً در عمل، بر آن مقدم است. با این وجود خود ساختمان داده *heap* کاربرد زیادی دارد. در این بخش یکی از رایج‌ترین کاربردهای *heap* را عنوان می‌کنیم: استفاده از آن به عنوان یک صف اولویت کارآمد. همانند *heap*ها، دو نوع صف اولویت وجود دارد: صف اولویت ماکزیمم و صف اولویت مینیمم. در اینجا توجّهمان را روی چگونگی پیاده‌سازی صف‌های اولویت ماکزیمم متمرکز می‌کنیم که آنها نیز به نوبه خود بر پایه *max-heap* هستند؛ تمرین ۳- ۶.۵ از شما می‌خواهد که روال‌ها را برای صف‌های اولویت مینیمم بنویسید.

یک ***صف اولویت***[1] ساختمان داده‌ای برای نگه داشتن مجموعه S از عناصری است که هر یک دارای یک مقدار مربوطه بنام *key* است. یک ***صف اولویت ماکزیمم***[2] اعمال زیر را پشتیبانی می‌کند.

INSERT(S, x) عنصر x را در مجموعه S درج می‌کند. این عملکرد می‌تواند به شکل $S \leftarrow S \cup \{x\}$ نوشته شود.

MAXIMUM(S) عنصری از S با بزرگ‌ترین کلید را برمی‌گرداند.

EXTRACT-MAX(S) عنصری از S با بزرگ‌ترین کلید را حذف کرده برمی‌گرداند.

INCREASE-KEY(S, x, k) مقدار کلید عنصر x را به مقدار جدید k افزایش می‌دهد، که فرض شده مقدار x حداقل به بزرگی مقدار فعلی کلید عنصر x است.

یکی از کاربردهای صف اولویت ماکزیمم دسته‌بندی کارها در یک کامپیوتر مشترک است. صف اولویت ماکزیمم، انجام شدن کارها و اولویت نسبی آنها را دنبال می‌کند. هنگامی که کاری پایان یافت یا در آن وقفه‌ای ایجاد شد، کاری که اولویتی بالاتر از بقیه دارد با استفاده از *EXTRACT-MAX* از میان آنها انتخاب می‌شود. در هر زمان می‌توان با استفاده از *INSERT* یک کار جدید اضافه کرد.

متقابلاً، یک ***صف اولویت مینیمم***[3] نیز اعمال *INSERT*، *MINMUM*، *EXTRACT-MIN* و

1.priority queue　　2.max-priority queue

3.min-priority queue

DECREASE-KEY را پشتیبانی می‌کند. یک صف اولویت مینیمم می‌تواند در یک شبیه ساز رویداد استفاده شود. اقلام درون صف رویدادهایی هستند که هر کدام توسط زمان وابسته که به عنوان کلید آن در نظر گرفته می‌شود شبیه سازی می‌شوند. رویدادها بایستی به ترتیب زمان رخدادشان شبیه‌سازی شوند، زیرا شبیه سازی یک رویداد می‌تواند باعث شبیه سازی رویدادهای دیگر در آینده شود. برنامه شبیه سازی در هر مرحله از *EXTRACT-MIN* استفاده می‌کند تا رویداد بعدی برای شبیه‌سازی انتخاب کند. به محض این که رویدادهای جدید ایجاد شوند، با استفاده از *INSERT* در صف اولویت مینیمم درج می‌شوند. کاربردهای دیگری از صف اولویت مینیمم را خواهیم دید و عمل *DECREASE-KEY* را به طور دقیق‌تر بررسی خواهیم کرد.

تعجبی ندارد که می‌توانیم از *heap* برای پیاده‌سازی صف اولویت استفاده کنیم. در یک عمل داده شده مانند زمان‌بندی کارها یا شبیه سازی رویدادها، عناصر صف اولویت متناظر با اشیاء در آن عمل هستند. اغلب لازم است تعیین کنیم که کدام شیء کاربردی متناظر با عنصر صف اولویت داده شده است، و بر عکس. بنابراین هنگامی که *heap* برای پیاده‌سازی صف اولویت به کار می‌رود. اغلب لازم است یک اتصال برای شی کاربردی متناظر در هر عنصر *heap* ذخیره کنیم. ساختار دقیق این اتصال بستگی به عمل دارد. (به عنوان مثال یک اشاره‌گر، یک عدد صحیح و...)به طور مشابه لازم است در هر شی کاربردی یک اتصال برای عنصر *heap* متناظر ذخیره کنیم. در این جا اتصال معمولاً یک اندیس آرایه می‌باشد. چون عناصر *heap* در طی عملیات *heap*، در آرایه تغییر مکان می‌دهند یک پیاده‌سازی واقعی طبق جایگزینی مجدد یک عنصر *heap* بایستی اندیس آرایه را در شیء کاربردی متناظر به روز درآورد. چون جزئیات دستیابی اشیاء کاربردی شدیداً به کاربرد و پیاده‌سازی آن بستگی دارد، آنها را در این جا دنبال نخواهیم کرد جز توجه به این مورد که در عمل لازم است این اتصالها به درستی نگهداری شوند.

اکنون در مورد چگونگی پیاده‌سازی اعمال یک صف اولویت ماکزیمم بحث می‌کنیم. *HEAP-MAXIMUM* عمل *MAXIMUM* را در زمان $O(1)$ پیاده‌سازی می‌کند.

```
HEAP-MAXIMUM(A)
1  return A[1]
```

روال *HEAP-EXTRACT-MAX* عمل *EXTRACT-MAX* را پیاده‌سازی می‌کند. این روال شبیه بدنه حلقه *for* (خطوط ۵–۳) روال *HEPSORT* است.

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A] < 1
2     then error "heap underflow"
3  max ← A[1]
```

```
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

زمان اجرا *HEAP-EXTRACT-MAX*، *O(lgn)* است، چون فقط مقدار ثابتی کار را علاوه بر زمان *O(lgn)* برای *MAX-HEAPIFY* انجام می‌دهد.

روال *HEAP-INCREASE-KEY* عمل *INCREASE-KEY* را پیاده‌سازی می‌کند. عنصری از صف اولویت که کلید آن افزایش یافته، با اندیس i در آرایه مشخص می‌شود. روال در ابتدا کلید عنصر *A[i]* را به مقدار جدیدش تغییر می‌دهد. چون افزایش کلید *A[i]* ممکن است ویژگی *max-heap* را مورد تخطی قرار دهد آنگاه روال در حالتی مشابه حلقه درج (خطوط ۷–۵) روال *INSERTION-SORT* از بخش ۲.۱ مسیری از این گره تا ریشه می‌پیماید تا جای مناسبی برای کلید که جدیداً افزایش یافته پیدا کند. در طی این پیمایش مکرراً یک عنصر را با پدرش مقایسه می‌کند، اگر کلید عنصر، بزرگ‌تر باشد جای کلیدها را عوض کرده و ادامه می‌دهد. اگر کلید عنصر کوچک‌تر باشد پایان می‌یابد، چون ویژگی *max-heap* برقرار می‌شود(برای ثابت دقیق حلقه تمرین ۵– ۶.۵ را ملاحظه کنید.)

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2     then error "new key is smaller than current key"
3  A[i] ← key
4  while i > 1 and A[PARENT(i)] < A[i]
5      do exchange A[i] ↔ A[PARENT(i)]
6         i ← PARENT(i)
```

شکل ۶.۵ مثالی از عملکرد *HEAP-INCREASE-KEY* را نشان می‌دهد. زمان اجرای *HEAP-INCREASE-KEY* روی یک *heap* با n عنصر، *O(lgn)* است، چون مسیری که از گره تا ریشهٔ به روز درآمده در خط ۳، دنبال شده است دارای طول *O(lgn)* است.

روال *MAX-HEAP-INSERT* عمل *INSERT* را پیاده‌سازی می‌کند. این روال به عنوان ورودی، کلید عنصر جدیدی که بایستی در *A max-heap* درج شود را می‌گیرد. روال در ابتدا با اضافه کردن یک برگ جدید به درخت که کلید آن ∞- است، *max-heap* را بسط می‌دهد. سپس *HEAP-INCREASE-KEY* را برای تنظیم کلید این گره جدید به مقدار صحیح خود و حفظ ویژگی *max-heap* فراخوانی می‌کند.

```
MAX-HEAP-INSERT(A, key)
1  heap-size[A] ← heap-size[A] + 1
2  A[heap-size[A]] ← −∞
3  HEAP-INCREASE-KEY(A, heap-size[A], key)
```

زمان اجرای *MAX-HEAP-INSERT* روی یک *heap* با *n* عنصر *O(lgn)* است. به طور خلاصه، *heap* می‌تواند هر عمل صف اولویت را روی مجموعه‌ای با اندازه *n* در زمان *O(lgn)* پشتیبانی کند.

تمرین‌ها

۱-۶.۵ عملکرد *HEAP-EXTRACT-MAX* را بر روی *heap* زیر نمایش دهید

$A = <5, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>$

۲-۶.۵ عملکرد *MAX-HEAP-INSERT* را روی *heap* $A = <15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>$ نمایش دهید. از شکل ۵-۶ به عنوان نمونه برای فراخوانی *HEAP-INCREASE-KEY* استفاده کنید.

۳-۶.۵ برای روال‌های *HEAP-MINIMUM*، *HEAP-EXTRACT-MIN*، *HEAP-DECREASE-KEY*، *MIN-HEAP-INSERT* شبه کدی بنویسید که توسط یک *min-heap* صف اولویت مینیمم را پیاده‌سازی کند.

۴-۶.۵ چرا برای قرار دادن مقدار ∞- در کلید گره درج شده در خط ۲ از *MAX-HEAP-INSERT* به خود زحمت می‌دهیم، در حالی که عمل بعدی که انجام می‌دهیم افزایش کلید آن به مقدار مطلوب است؟

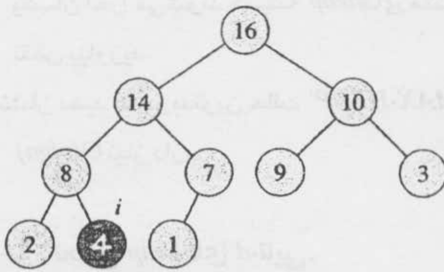
۵-۶.۵ با استفاده از ثابت حلقه زیر درستی *HEAP-INCREASE-KEY* را ثابت کنید:

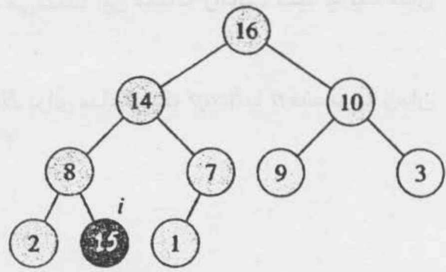
در ابتدای هر تکرار حلقه *While* خطوط ۶-۴، آرایه *A[1 . . heap-size[A]]* ویژگی *max-heap* را حفظ می‌کند، به جز یک تخطی که ممکن است وجود داشته باشد: *A[i]* ممکن است بزرگ‌تر از *A[PARENT(i)]* باشد.

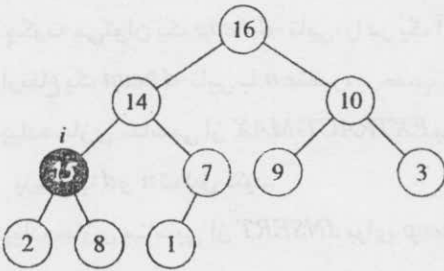
۶-۶.۵ چگونگی پیاده‌سازی یک صف *FIFO* - اولین ورودی اولین خروجی - توسط یک صف اولویت رانشان دهید. نشان دهید با صف اولویت چگونه می‌توان پشته را پیاده‌سازی کرد. صف و پشته در بخش ۱۰.۱ تعریف شده‌اند.)

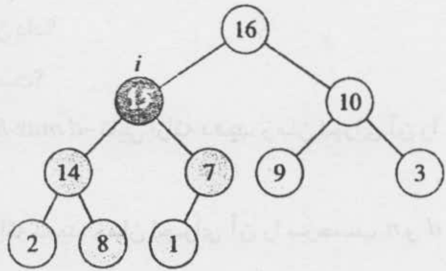
۷-۶.۵ عمل *HEAP-DELETE* شیء که در گره *i* از *heap* به نام *A* قرار دارد را حذف می‌کند. برای *HEAP-DELETE* یک پیاده‌سازی ارائه دهید که روی یک *max-heap* با *n* عنصر در زمان *O(lgn)* اجرا شود.

۸-۶.۵ الگوریتمی با زمان اجرای *O(nlgk)* ارائه دهید که *k* لیست مرتب شده را در یک لیست مرتب ادغام کند، به طوری که *n* تعداد کل اعضای در همه لیست‌های ورودی است. (*راهنمایی:* برای ادغام *k* طرفه از *min-heap* استفاده کنید.)









شکل ۶.۵ عمل *HEAP-INCREASE-KEY* . *max-heap(a)* شکل *(a)* ۶-۴ همراه با گره‌ای با اندیس *i* که کاملاً سیاه شده. *(b)* کلید این گره به *15* افزایش یافته. *(c)* پس از یک تکرار حلقهٔ *While* در خطوط ۶-۴، کلیدهای گره و پدر آن تعویض شده‌اند و اندیس *i* به پدر منتقل می‌شود. *(d)* پس از تکرار بعدی حلقه *While*. در اینجا $A[PARENT(i)] \ge A[i]$ اکنون ویژگی *max-heap* برقرار شده و روال پایان می‌یابد.

مسائل

۱-۶ ساختن *heap* با استفاده از درج

می‌توان روال *BUILD-MAX-HEAP* در بخش ۶.۳ را با استفاده مکرر از *MAX-HEAP-INSERT* برای درج عناصر در *heap*، پیاده‌سازی کرد. پیاده‌سازی زیر را در نظر بگیرید.

```
BUILD-MAX-HEAP'(A)
1  heap-size[A] ← 1
2  for i ← 2 to length[A]
3      do MAX-HEAP-INSERT(A, A[i])
```

a. آیا زمانی که روال‌های *BUILD-MAX-HEAP', BUILD-MAX-HEAP* روی یک آرایه ورودی

یکسان اجرا می‌شوند همیشه *heap*های مشابه ایجاد می‌کنند؟ این مسأله راثابت کنید یا یک مثال نقض بیاورید.

b. نشان دهید که در بدترین حالت *BUILD-MAX-HEAP'* برای ساخت یک *heap* با n عنصر به زمان $O(nlgn)$ نیاز دارد.

۶-۲ تحلیل *heap*های *d*-تایی

یک *d heap*- تایی مانند یک *heap* دودویی است، (ولی ممکن است یک استثناء وجود داشته باشد) گره‌های غیر برگ، به جای 2 فرزند d فرزند دارند.

a. چگونه می‌توان یک *d heap*- تایی، را در یک آرایه نشان داد؟

b. ارتفاع یک *d heap*- تایی با n عنصر، برحسب n و d چیست؟

c. پیاده‌سازی مناسبی از *EXTRACT-MAX* برای *d max-heap*- تایی ارائه دهید. زمان اجرای آن را برحسب d و n تحلیل کنید.

d. پیاده‌سازی مناسبی از *INSERT* برای *max-heap* ارائه دهید. زمان اجرای آن را برحسب n و d تحلیل کنید.

e. پیاده‌سازی مناسبی از *INCREASE-KEY(A, i, k)* ارائه دهید که ابتدا قرار می‌دهد $A[i] \leftarrow max(A[i], k)$ و سپس ساختار *d max-heap*- تایی را به شکل مناسبی بازسازی می‌نماید. زمان اجرای آن را برحسب d و n تحلیل کنید.

۶-۳ جداول *Young*

یک جدول *YOUNG* با ابعاد $m{\times}n$ یک ماتریس $m{\times}n$ است که ورودی‌های هر سطر از چپ به راست و ورودی‌های هر ستون از بالا به پایین مرتب شده‌اند. ممکن است بعضی از ورودی‌های جدول *YOUNG* مقدار ∞ می‌باشند، که با آنها به عنوان عناصری که وجود ندارند برخورد می‌کنیم. بنابراین، یک جدول *Young* می‌تواند برای نگهداری $r \leq mn$ عدد متناهی استفاده می‌شود.

a. یک جدول *Young* با ابعاد $4{\times}4$ رسم کنید که شامل عناصر $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$ باشد.

b. در مورد این که در جدول *Young* به نام Y با ابعاد $m{\times}n$ اگر $Y[1,1] = \infty$ باشد Y خالی و اگر $Y[m,n] < \infty$ باشد Y پر است (شامل mn عنصر است) بحث کنید.

c. الگوریتمی برای پیاده‌سازی *EXTRACT-MIN* روی یک جدول *YOUNG* ناتهی ارائه دهید که در زمان $O(m+n)$ اجرا شود. الگوریتم شما بایستی از زیرروالی بازگشتی استفاده کند که یک مسئله $m{\times}n$ را با حل بازگشتی زیر مسئله $(m-1){\times}n$ یا $m{\times}(n-1)$ حل می‌کند. (*راهنمایی*: از *MAX-HEAPIFY* کمک بگیرید) $T(p)$ را که $p=m+n$ است به عنوان حداکثر زمان اجرای

EXTRACT-MIN روی یک جدول *Young* با ابعاد $m \times n$ تعریف کنید. یک رابطه بازگشتی برای $T(p)$، که حد زمان $O(m+n)$ را نتیجه می‌دهد. ارائه کرده و آن را حل نمایید.

d. نشان دهید چطور یک عنصر جدید، در زمان $O(m+n)$ در جدول *Young* با ابعاد $m \times n$ غیر پر درج می‌شود.

e. بدون استفاده از هیچ روش مرتّب‌سازی دیگری به عنوان زیرروال نشان دهید که چگونه می‌توان از یک جدول *Young* با ابعاد $n \times n$ برای مرتّب‌سازی n^2 عدد در زمان $O(n^3)$ استفاده کرد.

f. الگوریتمی با زمان $O(m+n)$ ارائه دهید که مشخص کند آیا عدد داده شده در جدول *Young* با ابعاد $m \times n$ داده شده ذخیره شده است یا نه.

۷ مرتّب سازی سریع

مرتّب‌سازی سریع یک الگوریتم مرتّب‌سازی است که زمان اجرای آن روی یک آرایه ورودی با n عدد، در بدترین حالت $\Theta(n^2)$ است. علی رغم بالا بودن زمان اجرا، در بدترین حالت، اغلب مرتّب‌سازی سریع بهترین انتخاب عملی برای مرتّب‌سازی است. زیرا در حالت متوسط به طور قابل ملاحظه‌ای کارآمد است: زمان اجرای مورد انتظار $\Theta(nlgn)$ است، و ضرایت ثابت مخفی در نماد $\Theta(nlgn)$ کوچکند. همچنین این روش، خصوصیت مرتّب‌سازی درجا را دارد (بخش ۲.۱ را مشاهده کنید) و حتی در محیط‌های حافظه مجازی هم به خوبی عمل می‌کند.

بخش ۷.۱ الگوریتم و زیر برنامه‌های مهمی را که برای تقسیم بندی توسط مرتّب‌سازی سریع استفاده می‌شود توصیف می‌کند. به دلیل این که رفتار مرتّب‌سازی سریع پیچیده است در بخش ۷.۲ با بحثی شهودی در مورد کارایی آن کار را آغاز کرده و تحلیل دقیق آن را به انتهای فصل موکول می‌کنیم. بخش ۷.۳ نوعی از مرتّب‌سازی سریع که از نمونه برداری تصادفی استفاده می‌کند را ارائه می‌دهد. این الگوریتم در حالت متوسط زمان اجرای خوبی دارد و هیچ ورودی خاصی، باعث فراخوانی رفتار بدترین حالت نمی‌شود. الگوریتم تصادفی در بخش ۷.۴ تحلیل شده است که در آنجا نشان داده شده که زمان اجرای آن در بدترین حالت $\Theta(n^2)$ و در حالت متوسط $O(nlgn)$ است.

۷.۱ توصیف مرتّب‌سازی سریع

مرتّب‌سازی سریع، مانند مرتّب‌سازی به روش ادغام، بر اساس نمونه‌ای از تقسیم و حل که در بخش ۲.۳.۱ معرفی شد می‌باشد. در این جا ۳ مرحله فرآیند تقسیم و حل برای مرتّب‌سازی یک نمونه زیر آرایه $A[p\ ..\ r]$ آمده است.

تقسیم: تقسیم آرایه $A[p\ ..\ r]$ به دو زیر آرایه (ممکن است خالی باشند) $A[p\ ..\ q-1]$ و $A[q+1\ ..\ r]$ به طوری که هر عنصر $A[p\ ..\ q-1]$ کوچک‌تر یا مساوی با $A[q]$ است و $A[q]$ به نوبه خود کوچک‌تر یا مساوی با هر یک از عناصر $A[q+1\ ..\ r]$ می‌باشد. اندیس q را به عنوان بخشی

از این روال تقسیم‌بندی، محاسبه می‌کنیم.

حل: دو زیر آرایه $A[p\ ..\ q-1]$ و $A[q+1\ ..\ r]$ را با فراخوانی‌های بازگشتی مرتّب‌سازی سریع، مرتب می‌کنیم.

ترکیب: از آنجایی که زیر آرایه‌ها درجا مرتب می‌شوند لازم نیست برای ترکیب آنها کار دیگری انجام شود. اکنون آرایه $A[p\ ..\ r]$ مرتب می‌شود.

روال زیر مرتّب‌سازی سریع را پیاده‌سازی می‌کند.

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

برای مرتّب‌سازی کل آرایه A، فراخوانی اولیه $QUICKSORT(A, 1, length[A])$ است.

تقسیم آرایه

کلید این الگوریتم روال $PARTITION$ است که ترتیب زیر آرایه $A[p\ ..\ r]$ را درجا تغییر می‌دهد.

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5            then i ← i + 1
6                 exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

شکل ۷.۱ عمل $PARTITION$ را روی یک آرایه 8 عضوی نشان می‌دهد. $PARTITION$ همیشه عنصر $x = A[r]$ را به عنوان یک عنصر محوری برای تقسیم بندی زیرآرایه $A[p\ ..\ r]$، انتخاب می‌کند. هنگامی که روال اجرا می‌شود آرایه به 4 ناحیه (ممکن است خالی باشد) تقسیم می‌شود. در ابتدای هر تکرار حلقه for خطوط ۶-۳، در هر ناحیه ویژگی‌های خاصی برقرار است که می‌توانیم به عنوان ثابت حلقه در نظر بگیریم:

در آغاز هر تکرار حلقه خطوط ۶-۳، برای هر اندیس k از آرایه

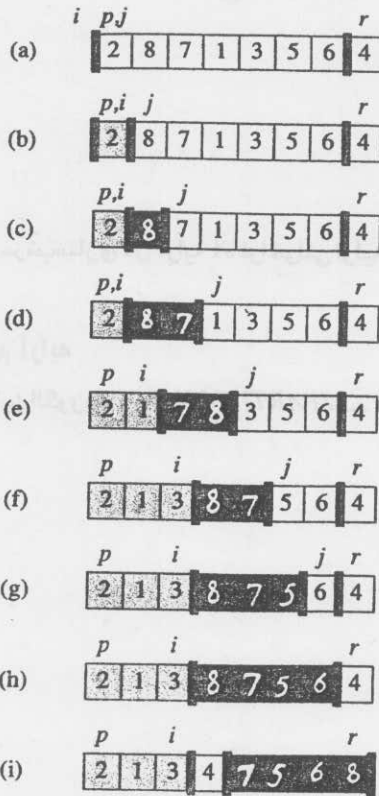
۱. اگر $p \le k \le i$ آنگاه $A[k] \le x$

۲. اگر $i+1 \leq k \leq j-1$ آنگاه $A[x] > x$

۳. اگر $k = r$ آنگاه $A[k] = x$

شکل ۷.۲ این ساختار را خلاصه می‌کند. اندیس‌هایی که بین j و $r-1$ هستند در هیچ یک از ۳ حالت بالا قرار نمی‌گیرند و مقادیر این ورودی‌ها هیچ گونه ارتباط خاصی با عنصر محوری x ندارند.

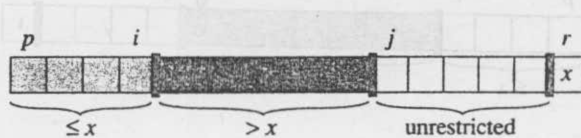
لازم است نشان دهیم که این ثابت حلقه قبل از اولین تکرار درست است و هر تکرار حلقه، ثابت را حفظ می‌کند و ثابت ویژگی مفیدی برای نشان دادن الگوریتم در هنگام پایان کار حلقه ارائه می‌دهد.



شکل ۷.۱ عمل $PARTITION$ روی یک آرایه نمونه. عناصری از آرایه که سایه روشن خورده‌اند همگی در قسمت اول هستند و مقادیر آنها کوچک‌تر یا مساوی x است. عناصری که سایه تیره خورده‌اند در قسمت دوم هستند و مقادیر آنها بزرگ‌تر از x است. عناصری که سایه نخورده‌اند هنوز در هیچ یک از دو قسمت اول قرار نگرفته‌اند و آخرین عنصر سفید، عنصر محوری است. (a) آرایه اولیه و مقدار دهی متغیرها. هیچ یک از عناصر در یکی از دو قسمت اول قرار گرفته‌اند. (b) مقدار 2 با خودش تعویض و در قسمت مقادیر کوچک‌تر قرار گرفته است. (c)-(d) مقادیر 8 و 7 به قسمت مقادیر بزرگتر اضافه شده‌اند. (e) مقادیر 1 و 8 تعویض شده‌اند و قسمت کوچک‌تر افزایش می‌یابد. (f) مقادیر 3 و 8 عوض شده و قسمت کوچک‌تر افزایش می‌یابد. (g)-(h) قسمت بزرگ‌تر افزایش می‌یابد تا 5 و 6 را نیز شامل شده و حلقه پایان می‌یابد. (i) در خطوط ۸-۷ عنصر محوری عوض شده تا بین دو قسمت قرار گیرد.

مقداردهی اولیه: قبل از اولین تکرار حلقه، $i = p\text{-}1$ و $j = p$ است. بین p و i و بین $i+1$ و $j\text{-}1$ هیچ مقداری وجود ندارد، بنابراین دو شرط اول ثابت حلقه به طور بدیهی برقرار است. سومین شرط نیز توسط انتساب در خط یک، برقرار می‌شود.

نگهداری: همان طور که شکل ۷.۳ نشان می‌دهد، بسته به نتیجه تست خط ۴، دو حالت در نظر گرفته می‌شود. شکل ۷.۳ (a) نشان می‌دهد هنگامی که $A[j] > x$ است چه اتفاقی می‌افتد؛ تنها عمل در حلقه افزایش j است. بعد از این که j افزایش یافت شرط 2 برای $A[j\text{-}1]$ برقرار شده و کلیه ورودی‌های دیگر بدون تغییر باقی می‌مانند. شکل ۷.۳ (b) نشان می‌دهد هنگامی که $A[j] \le x$ است چه اتفاقی می‌افتد؛ i افزایش یافته، $A[i]$ و $A[j]$ تعویض شده و سپس j افزایش می‌یابد. به دلیل این جابه‌جایی اکنون داریم: $A[i] \le x$ و شرط 1 برقرار می‌شود. به طور مشابه، $A[j\text{-}1] > x$ را نیز داریم، چون عضوی که با $A[j\text{-}1]$ عوض شده، با توجه به ثابت حلقه بزرگ‌تر از x است.



شکل ۷.۲ چهار ناحیه‌ای که توسط روال $PARTITION$ روی زیر آرایه $A[p\ .\ .\ r]$ ایجاد می‌شوند. مقادیر موجود در $A[p\ .\ .\ r]$ همگی کوچکتر یا مساوی x، مقادیر موجود در $A[i+1\ .\ .\ j\text{-}1]$ همگی بزرگ‌تر از x و $A[r] = x$ است. مقادیر موجود در $A[j\ .\ .\ r\text{-}1]$ می‌توانند هر مقداری داشته باشند.

خاتمه: در خاتمه، $j = r$ است. بنابراین، هر ورودی در آرایه، در یکی از سه مجموعه‌ای که توسط ثابت تعریف شده وجود دارد و ما مقادیر آرایه را به سه مجموعه تقسیم کرده‌ایم: آنهایی که کوچک‌تر یا مساوی x هستند، آنهایی که بزرگ‌تر از x هستند و یک مجموعه عضوی شامل x.

دو خط پایانی $PARTITION$ به وسیله جابه‌جا کردن عنصر محوری با سمت چپ‌ترین عنصر بزرگ‌تر از x، عنصر محوری را به مکانش در وسط آرایه منتقل می‌کند. اکنون خروجی $PARTITION$ شرایطی که برای مرحله تقسیم تعیین شده بود را برقرار می‌کند.

زمان اجرای $PARTITION$ روی زیر آرایه $A[p\ .\ .\ r]$، $\Theta(n)$ است که $n = r\text{-}p+1$ می‌باشد(تمرین ۷.۱.۳ را ملاحظه کنید.)

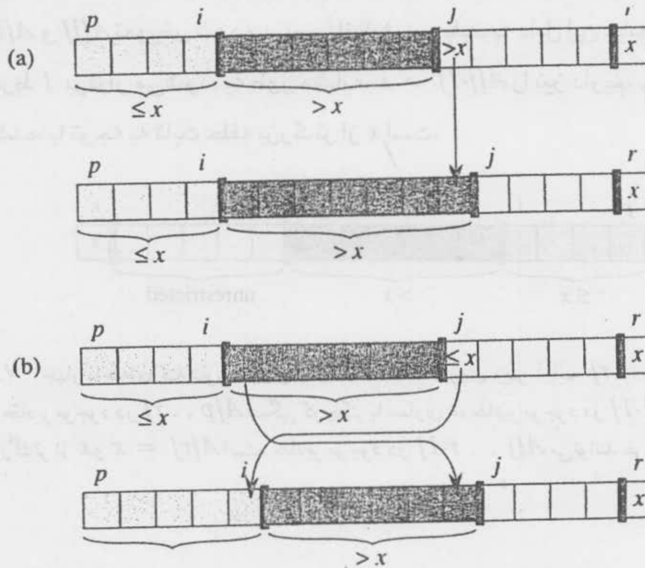
تمرین‌ها

۱– ۷.۱ با استفاده از شکل ۷.۱ به عنوان نمونه، عمل $PARTITION$ را بر روی آرایه $A=\langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\rangle$ نمایش دهید.

۷.۱-۲ هنگامی که کلیه اعضاء آرایه $A[p\;.\;.\;r]$ مقدار یکسانی دارند، روال *PARTITION* چه مقداری را برای q برمی‌گرداند؟ *PARTITION* را طوری تغییر دهید که وقتی کلیه مقادیر آرایه $A[p\;.\;.\;r]$ یکسانند $q = (p+r)/2$ باشد.

۷.۱-۳ در مورد زمان اجرای *PARTITION* که روی زیر آرایه‌ای به اندازه n $\Theta(n)$ است مختصراً بحث کنید.

۷.۱-۴ *QUICKSORT* را طوری تغییر دهید که آرایه را به ترتیب غیر صعودی مرتب کند.



شکل ۷.۳ دو حالت برای یک تکرار روال *PARTITION*. (a) اگر $A[j] > x$ باشد تنها عمل، اضافه کردن j است که ثابت حلقه را حفظ می‌کند. (b) اگر $A[j] \le x$، اندیس i افزایش یافته، $A[i]$ و $A[j]$ تعویض شده و سپس j افزایش می‌یابد. باز هم ثابت حلقه حفظ می‌شود.

۷.۲ کارایی مرتب‌سازی سریع

زمان اجرای مرتب سازی سریع به این بستگی دارد که آیا تقسیم بندی متوازن است یا نامتوازن، و این مورد نیز به نوبه خود به این بستگی دارد که چه عناصری برای تقسیم‌بندی مورد استفاده قرار گرفته‌اند. اگر تقسیم‌بندی متوازن باشد الگوریتم به طور مجانبی با سرعتی برابر با مرتّب‌سازی ادغام اجرا می‌شود. با این وجود اگر تقسیم بندی نامتوازن باشد، این مرتّب‌سازی می‌تواند به طور مجانبی به کندی مرتّب‌سازی درجی اجرا شود. در این بخش، چگونگی اجرای مرتّب‌سازی سریع را با فرض تقسیم‌بندی متوازن در مقابل تقسیم بندی نامتوازن به طور غیر رسمی بررسی می‌کنیم.

بدترین حالت تقسیم‌بندی

بدترین حالت برای مرتّب‌سازی سریع هنگامی رخ می‌دهد که روال تقسیم بندی یک زیرمسئله با $n-1$ عنصر و یک زیر مسئله با 0 عنصر ایجاد کند. (این ادعا در بخش ۷.۴.۱ ثابت می‌شود) فرض کنیم که این تقسیم‌بندی نامتوازن در فراخوانی بازگشتی به وجود آید. تقسیم‌بندی زمان $\Theta(n)$ را صرف می‌کند. چون فراخوانی بازگشتی روی آرایه‌ای با اندازه 0، دقیقاً $T(0)=\Theta(1)$ را برگرداننده و رابطه بازگشتی برای زمان اجرا به صورت زیر می‌باشد:

$$T(n) = T(n-1)+T(0)+\Theta(n)$$
$$=T(n-1)+\Theta(n).$$

به طور شهودی اگر هزینه‌های ایجادشده در هر مرحله بازگشت را با هم جمع کنیم یک سری حسابی به دست می‌آوریم که برابر $\Theta(n^2)$ است. در واقع، ساده‌تر این است که برای اثبات این که جواب رابطه بازگشتی $T(n) = T(n-1)+\Theta(n)$، $T(n) = \Theta(n^2)$ است، از روش جایگذاری استفاده کنیم.(تمرین ۱-۷.۲ را ملاحظه کنید.)

بنابراین اگر تقسیم‌بندی در هر سطح بازگشتی الگوریتم به طور حداکثر نامتوازن باشد، زمان اجرا برابر $\Theta(n^2)$ است. بنابراین زمان اجرای مرتّب‌سازی سریع در بدترین حالت بهتر از مرتّب‌سازی درجی نیست. علاوه بر این، زمان اجرای $\Theta(n^2)$ زمانی رخ می‌دهد که آرایه ورودی از قبل به طور کامل مرتب شده باشد - در شرایطی مشابه، مرتّب‌سازی درجی در زمان $O(n)$ اجرا می‌شود.

بهترین حالت تقسیم‌بندی

در اکثر تقسیمات ممکن، *PARTITION* دو زیر مسئله ایجاد می‌کند که اندازه هر یک از آنها بیشتر از $n/2$ نیست چون اندازه یکی $\lfloor n/2 \rfloor$ و اندازه دیگری $\lceil n/2 \rceil -1$ است. در این حالت، مرتّب‌سازی سریع خیلی سریع‌تر انجام می‌شود. بنابراین رابطه بازگشتی برای زمان اجرا به صورت زیر است

$$T(n) \le 2T(n/2)+\Theta(n)$$

که با استفاده از حالت دوم قضیه اصلی (قضیه ۴.۱) دارای جواب $T(n)=O(nlgn)$ می‌باشد. لذا موازنه برابر دو طرف تقسیم در هر مرحله از بازگشت، الگوریتمی ایجاد می‌کند که به طور مجانبی سریع‌تر است.

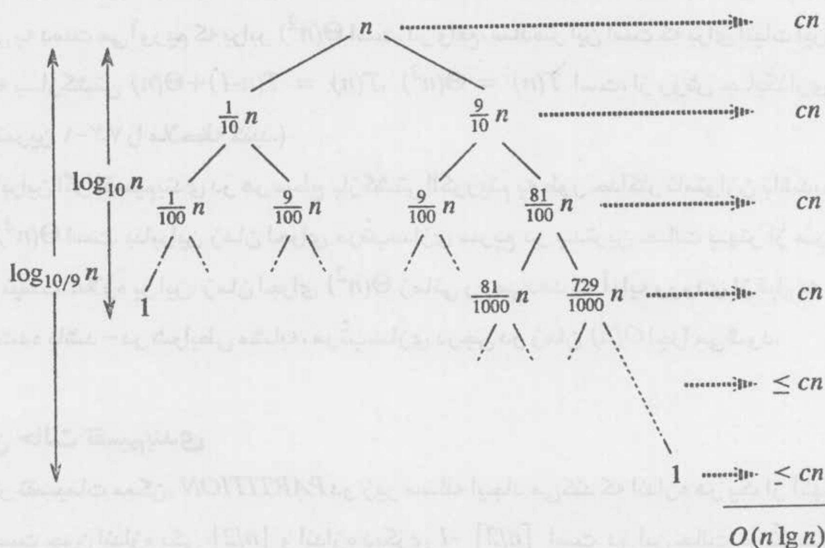
تقسیم‌بندی متوازن

همان طور که در تحلیل‌های بخش ۷.۴ نشان خواهیم داد زمان اجرای مرتّب سازی سریع در حالت متوسط، به بهترین حالت نزدیک‌تر است تا به بدترین حالت. کلید درک علت این موضوع، درک این مطلب است که توازن تقسیم‌بندی چگونه در رابطه بازگشتی که زمان اجرا را بیان می‌کند منعکس می‌شود.

به عنوان مثال فرض کنید الگوریتم تقسیم، همیشه تقسیم بندی با نسبت 9 به 1 ایجاد می‌کند که در اولین نظر کاملاً نامتوازن به نظر می‌رسد. سپس به رابطه بازگشتی زیر برای زمان اجرای مرتّب‌سازی سریع می‌رسیم

$$T(n) \le T(9n/10) + T(n/10) + cn$$

که به طور صریح ثابت c را در عبارت $\Theta(n)$ به طور پنهان قرار داده‌ایم. شکل ۷.۴ درخت بازگشتی برای این رابطه بازگشتی را نشان می‌دهد. توجه داشته باشید که هر سطح درخت هزینه cn را دارد، تا این که در عمق $log_{10}n = \Theta(lgn)$ به یک شرایط مرزی برسیم و سپس هزینه سطوح، حداکثر cn می‌شود. رابطه بازگشتی در عمق $log_{10/9}n = \Theta(lgn)$ پایان می‌یابد.



شکل ۷.۴ درختی بازگشتی برای $QUICKSORT$ که در آن $PARTITION$ دو قسمت به نسبت 9 به 1 ایجاد می‌کند که زمان اجرای $O(nlgn)$ را نتیجه می‌دهد. گره‌ها اندازه زیر مسئله‌ها را همراه با هزینه هر سطح در سمت راست نشان می‌دهند. هزینه هر سطح شامل ثابت ضمنی c در عبارت $\Theta(n)$ می‌باشد.

از این رو هزینه کل مرتّب‌سازی سریع $O(nlgn)$ است. بنابراین با یک تقسیم به نسبت 9 به 1 در هر سطح از بازگشت، که به طور شهودی کاملاً نامتوازن به نظر می‌رسد، مرتّب‌سازی سریع در زمان $O(nlgn)$ اجرا می‌شود - به طور مجانبی مشابه زمانی است که قسمت، درست زیر میانه باشد. در حقیقت، حتی تقسیم به نسبت 99 به 1 نیز زمان اجرایی برابر با $O(nlgn)$ را نتیجه می‌دهد. علت این است که هر تقسیم با نسبت ثابت، یک درخت بازگشتی با عمق $\Theta(lgn)$ را نتیجه می‌دهد که هزینه هر سطح $O(n)$ است. بنابراین هر جا که تقسیم دارای نسبت ثابت باشد، زمان اجرا برابر $O(nlgn)$ است.

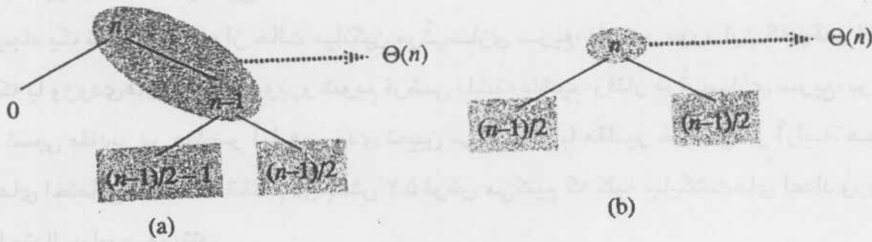
به دست آوردن حالت میانگین

برای ایجاد یک تصور واضح از حالت میانگین مرتّب‌سازی سریع، باید در مورد این که چقدر انتظار داریم که با ورودی‌های متفاوت روبرو شویم فرضی داشته باشیم. رفتار مرتّب‌سازی سریع، بر طبق ترتیب نسبی مقادیر در عناصر آرایه ورودی تعیین می‌شود نه با مقادیر خاصی در آرایه. همانند تحلیل‌های احتمالی مسئله استخدام در بخش ۵.۲ فرض می‌کنیم که کلیه جایگشت‌های اعداد ورودی، دارای احتمال برابری هستند.

هنگامی که مرتّب‌سازی سریع را روی یک آرایه ورودی تصادفی اجرا می‌کنیم، همان طور که تحلیل غیر رسمی ما فرض می‌کند بعید است که تقسیم بندی همیشه در هر سطح به یک طریق انجام می‌شود. انتظار داریم که تعدادی از تقسیمات به شکل معقولی، خوب موازنه شده باشند و تعدادی نسبتاً نامتوازن باشند. به عنوان مثال، تمرین ۶-۷.۲ از شما می‌خواهد نشان دهید که حدود 80% اوقات *PARTITION* تقسیم‌بندی به وجود می‌آورد که توازن آن بیشتر از نسبت 9 به 1 است و حدود 20% اوقات تقسیم بندی ایجاد می‌کند که توازن آن کمتر از نسبت 9 به 1 است.

در حالت میانگین *PARTITION* ترکیبی از تقسیم بندی "خوب" و "بد" ایجاد می‌کند. در یک درخت بازگشتی برای اجرای *PARTITION* در حالت میانگین، تقسیم بندی خوب و بد به طور تصادفی در درخت توزیع می‌شوند. به منظور درک مطلب فرض کنید تقسیم‌بندی‌های خوب و بد یکی در میان در سطوح درخت ظاهر می‌شوند و تقسیم‌بندی‌های خوب، بهترین حالت تقسیمات و تقسیم‌بندی‌های بد، بدترین حالت تقسیمات باشند. شکل ۷.۵ (a) تقسیمات در دو سطح متوالی در درخت بازگشتی را نمایش می‌دهد. در ریشه درخت، هزینه تقسیم بندی n است و اندازه زیر آرایه هایی که ایجاد شده‌اند $n-1$ و 0 است: بدترین حالت. در سطح بعدی برای زیرآرایه با اندازه $n-1$ بهترین حالت، تقسیم‌بندی به دو زیر آرایه با اندازه‌های $(n-1)/2-1$ و $(n-1)/2$ است. فرض می‌کنیم که هزینه شرایط مرزی برای زیرآرایه با اندازه 0 برابر 1 است.

ترکیب تقسیم‌بندی بد که به دنبال آن تقسیم‌بندی مناسبی انجام شده است، سه زیر آرایه با اندازه‌های 0، $(n-1)/2-1$ و $(n-1)/2$ ایجاد می‌کند که هزینه ترکیب تقسیم بندی $\Theta(n)+\Theta(n-1)=\Theta(n)$ است. یقیناً این شرایط، بدتر از حالتی که در شکل ۷.۵ (b) یک سطح از تقسیم‌بندی، دو زیر آرایه با اندازه $(n-1)/2$ با هزینه $\Theta(n)$ ایجاد می‌کرد نیست. همچنان این حالت دوم متوازن است! به طور شهودی هزینه $\Theta(n)$ برای تقسیم‌بندی خوب و مناسب می‌تواند هزینه $\Theta(n-1)$ برای تقسیم‌بندی بد را در برگیرد و لذا تقسیم‌بندی حاصل مناسب است. بنابراین زمان اجرای مرتّب‌سازی سریع، هنگامی که سطوح به شکل یکی در میان دارای تقسیم‌بندی خوب و بد هستند، همانند زمان اجرا برای حالتی است که تقسیم‌بندی‌ها تنها به صورت مناسب انجام شده‌اند: همچنان $O(nlgn)$، ولی با ثابتی که کمی بزرگتر می‌باشد و در نماد O پنهان است. در بخش ۷.۴.۲ تحلیلی از حالت میانگین ارائه خواهیم کرد.



شکل ۷.۵ (a). دو سطح از یک درخت بازگشتی برای مرتّب‌سازی سریع. تقسیم‌بندی در ریشه دارای هزینه n است و تقسیم‌بندی نامناسبی را به وجود می‌آورد: دو زیر آرایه با اندازه‌های 0 و $n-1$ تقسیم زیر آرایه با اندازه $n-1$ دارای هزینه $n-1$ بوده ویک تقسیم‌بندی خوب ایجاد می‌کند: زیر آرایه هایی با اندازه‌های $(n-1)/2-1$ و $(n-1)/2$. (b) یک سطح از درخت بازگشتی که بسیار متوازن است. در هر دو بخش، هزینه تقسیم برای زیر مسئله‌هایی که با منحنی سایه‌دار نشان داده شده‌اند برابر $\Theta(n)$ است. زیر مسئله‌هایی که برای حل در (a) باقیمانده‌اند و با مستطیل‌های سایه‌دار مشخص شده‌اند، بزرگ‌تر از زیر مسئله‌های متناظر برای حل در (b) باقیمانده‌اند نیستند.

تمرین‌ها

۷.۲-۱ همان طور که در ابتدای بخش ۷.۲ ادعا شد، با استفاده از روش جایگذاری ثابت کنید که $T(n)=\Theta(n^2)$ جواب رابطه بازگشتی $T(n)=T(n-1)+\Theta(n)$ است.

۷.۲-۲ زمان اجرای $QUICKSORT$ هنگامی که همه عناصر آرایه یک مقدار دارند چیست؟

۷.۲-۳ نشان دهید که زمان اجرای $QUICKSORT$ هنگامی که عناصر آرایه A متفاوت بوده و با ترتیبی نزولی ذخیره شده‌اند، $\Theta(n^2)$ است.

۷.۲-۴ بانک‌ها اغلب نقل و انتقالات یک حساب را بر طبق زمان‌های نقل و انتقال ثبت می‌کنند ولی بسیاری از مردم تمایل دارند گزارش بانکی خود را با چک هایی که بر طبق شماره چک لیست شده‌اند دریافت کنند. مردم معمولاً چک‌ها را بر طبق شماره چک می‌نویسند و تجار به وسیله ارسال چک آنها را نقد می‌کنند. بنابراین مسئله تبدیل ترتیب زمان انتقال به ترتیب شماره چک، مسئله مرتّب‌سازی ورودی تقریباً مرتب است. ثابت کنید $INSERTION\text{-}SORT$ در این مسئله بر $QUICKSORT$ غلبه می‌کند.

۷.۲-۵ فرض کنید تقسیم بندی‌ها در هر سطح مرتب‌سازی سریع با نسبت $1-\alpha$ به α انجام شده‌اند به طوری که $0 < \alpha \le 1/2$ عددی ثابت است. نشان دهید که کمترین عمق یک برگ در درخت بازگشتی تقریباً $-lgn/lg\alpha$ و بیشترین عمق تقریباً $-lgn/-lg(1-\alpha)$ می‌باشد. (نگران گرد کردن اعداد صحیح نباشید.)

۷.۲-۶* ثابت کنید برای هر ثابت $0 < \alpha \le 1/2$ احتمال اینکه روی یک آرایه ورودی تصادفی، $PARTITION$ تقسیمی با توازن بیش از $1-\alpha$ به α ایجاد کند برابر $1-2\alpha$ است.

۷.۳ صورت تصادفی مرتّب‌سازی سریع

در بررسی رفتار مرتّب‌سازی سریع در حالت میانگین فرض کردیم که تمام جایگشت‌های اعداد ورودی دارای احتمال برابری هستند. با این وجود، در یک شرایط تخصصی نمی‌توان همیشه انتظار داشت که این مطلب برقرار باشد. (تمرین ۴- ۷.۲ را ملاحظه کنید.) همان طور که در بخش ۵.۳ دیدیم، گاهی اوقات می‌توانیم به منظور دستیابی به اجرای خوب در حالت میانگین روی همه ورودی‌ها، تصادفی سازی را به الگوریتم اضافه کنیم. بسیاری از مردم شکل تصادفی مرتّب‌سازی سریع حاصل را به عنوان الگوریتم مرتّب‌سازی برای ورودی‌های به اندازه کافی بزرگ انتخاب می‌کنند.

در بخش ۵.۳، الگوریتم خود را با جایگشت کردن ورودی، تصادفی کردیم. ممکن است بتوانیم برای مرتّب‌سازی سریع نیز این کار را انجام دهیم، ولی یک روش تصادفی کردن متفاوت که **نمونه‌برداری تصادفی**[1] نام دارد، تحلیل ساده‌تری را نتیجه می‌دهد. به جای این که همیشه از $A[r]$ به عنوان عنصر محوری استفاده کنیم، از عنصری که به طور تصادفی از زیرآرایه $A[p \,..\, r]$ انتخاب می‌شود استفاده می‌کنیم. این کار را با تعویض عنصر $A[r]$ با عنصری که از $A[p \,..\, r]$ به طور تصادفی انتخاب می‌شود انجام می‌دهیم. این تغییر که در آن به طور تصادفی از ادامه $p \,..\, r$ یک نمونه انتخاب می‌کنیم، اطمینان می‌دهد که عنصر محوری $x = A[r]$ با احتمال برابر می‌تواند هر یک از $r-p+1$ عنصر زیر آرایه باشد. چون عنصر محوری بشکلی تصادفی انتخاب می‌شود انتظار داریم تقسیمات آرایه ورودی در حالت میانگین به شکل مناسبی متوازن باشند.

تغییرات *PARTITION* و *QUICKSORT* اندک هستند. در روال جدید تقسیم‌بندی قبل از تقسیم‌بندی واقعی، به سادگی تعویض را پیاده‌سازی می‌کنیم:

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  exchange A[r] ↔ A[i]
3  return PARTITION(A, p, r)
```

مرتّب‌سازی سریع جدید، بجای *PARTITION*، *RANDOMIZED-PARTITION* را فراخوانی می‌کند:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2     then q ← RANDOMIZED-PARTITION(A, p, r)
3          RANDOMIZED-QUICKSORT(A, p, q − 1)
4          RANDOMIZED-QUICKSORT(A, q + 1, r)
```

در بخش بعد به تحلیل این الگوریتم خواهیم پرداخت.

---

[1] random sampling

تمرین‌ها

۱- ۷.۳ چرا کارآیی الگوریتم تصادفی را در حالت میانگین تحلیل می‌کنیم نه در بدترین حالت؟

۲- ۷.۳ در طی اجرای روال *RANDOMIZED-QUICKSORT*، در بدترین حالت چند بار *RANDOM* که اعداد تصادفی تولید می‌کند را فراخوانی می‌کنیم. در بهترین حالت چطور؟ پاسخ خود را برحسب نمادگذاری Θ ارائه دهید.

۷.۴ تحلیل مرتّب‌سازی سریع

بخش ۷.۲ مشاهداتی در مورد رفتار مرتّب‌سازی سریع در بدترین حالت و این که چرا انتظار داریم به سرعت عمل کند را ارائه داد. در این بخش، رفتار مرتّب‌سازی سریع را دقیق‌تر تحلیل می‌کنیم. با تحلیل‌بدترین‌حالت شروع می‌کنیم، که برای *QUICKSORT* و *RANDOMIZED-QUICKSORT* بیان می‌شود و با تحلیل حالت میانگین *RANDOMIZED-QUICKSORT* به کار خود خاتمه می‌دهیم.

۷.۴.۱ تحلیل بدترین حالت

در بخش ۷.۲ دیدیم که یک تقسیم بندی در بدترین حالت در هر سطح بازگشت مرتّب‌سازی سریع، زمان اجرای $\Theta(n^2)$ را ایجاد می‌کند که این زمان به طور شهودی بدترین حالت زمان اجرای الگوریتم می‌باشد. اکنون این ادعا را اثبات می‌کنیم.

با استفاده از روش جایگذاری (بخش ۴.۱ را ملاحظه نمایید) می‌توانیم نشان دهیم که زمان اجرای مرتّب‌سازی سریع $O(n^2)$ است. فرض کنید $T(n)$ بدترین حالت زمان اجرا برای روال *QUICKSORT* روی ورودی با اندازه n باشد. رابطه بازگشتی زیر را داریم

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) , \qquad (۷.۱)$$

که پارامتر q بین 0 تا $n\text{-}1$ تغییر می‌کند زیرا روال *PARTITION* دو زیر مسئله با اندازه کلی $n\text{-}1$ ایجاد می‌کند. حدس می‌زنیم که برای یک ثابت c، $T(n) \le cn^2$ باشد. با جایگذاری این حدس در رابطه بازگشتی (۷.۱) بدست می‌آوریم.

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) . \end{aligned}$$

همان طور که مشاهده می‌شود حداکثر مقدار عبارت $q^2+(n-q-1)^2$ در نقاط انتهایی بازه $0 \le q \le n-1$ بدست می‌آید، چون مشتق دوم عبارت بر حسب q مثبت است (تمرین ۳-۷.۴ را ملاحظه کنید). این دیدگاه، حد

$$\max_{0 \le q \le n-1}(q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$$

را به ما می‌دهد. با ادامه محدود کردن $T(n)$ داریم

$$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

زیرا می‌توانیم ثابت c را به اندازه کافی بزرگ بگیریم تا عبارت $c(2n-1)$ عبارت $\Theta(n)$ را در برگیرد. بنابراین $T(n)=O(n^2)$. در بخش ۷.۲ حالت خاصی را دیدیم که مرتّب‌سازی سریع، زمان $\Omega(n^2)$ را صرف می‌کرد: هنگامی که تقسیم‌بندی نامتوازن است. تمرین ۱-۷.۴ از شما می‌خواهد که نشان دهید رابطه بازگشتی (۷.۱) دارای جواب $T(n)=\Omega(n^2)$ است. بنابراین زمان اجرای مرتّب‌سازی سریع (در بدترین حالت) برابر $\Theta(n^2)$ است.

۷.۴.۲ زمان اجرای مورد انتظار

قبلاً اثباتی شهودی در مورد این که چرا زمان اجرای *RANDOMIZED-QUICKSORT* در حالت میانگین $O(nlgn)$ است ارائه دادیم: اگر در هر سطح از بازگشت، تقسیم‌بندی که با استفاده از *RANDOMIZED-PARTITION* صورت گرفته است، کسر ثابتی از عناصر را در یک طرف تقسیم‌بندی قرار دهد، آنگاه عمق درخت بازگشت $\Theta(lgn)$ بوده و $O(n)$ عمل در هر سطح انجام می‌شود. حتی اگر سطح جدیدی با نامتوازن‌ترین تقسیم ممکن را در بین این سطوح اضافه کنیم زمان کل، $O(nlgn)$ باقی می‌ماند. می‌توانیم ابتدا با دانستن چگونگی عملکرد روال تقسیم‌بندی و سپس با استفاده از آن برای به دست آوردن حد $O(nlgn)$ روی زمان اجرای مورد انتظار، زمان اجرایی که برای *RANDOMIZED-QUICKSORT* انتظار می‌رود را دقیق‌تر تحلیل کنیم. این حد بالای زمان اجرای مورد انتظار در ترکیب با حد بهترین حالت که در بخش ۷.۲ دیدیم، زمان اجرای مورد انتظار $\Theta(nlgn)$ را نتیجه می‌دهد.

زمان اجرا و مقایسه‌ها

زمان اجرای *QUICKSORT* به زمانی که در روال *PARTITION* صرف می‌شود بستگی دارد. هر بار که روال *PARTITION* فراخوانی می‌شود، یک عنصر محوری انتخاب می‌شود و این عنصر هیچگاه در

فراخوانی‌های بعدی *QUICKSORT* و *PARTITION* ظاهر نمی‌شود. بنابراین حداکثر *n* فراخوانی *PARTITION* در کل اجرای الگوریتم مرتّب‌سازی سریع وجود دارد و یک فراخوانی *PARTITION* دارای زمان $O(1)$ به اضافه مقدار زمانی که متناسب با تعداد تکرارهای حلقه *for* در خطوط ۶-۳ است می‌باشد. هر تکرار حلقه *for* یک مقایسه در خط ۴ انجام می‌دهد، مقایسه‌ای بین عنصر محوری و عنصر دیگری از آرایه *A*. بنابراین اگر بتوانیم تعداد کل دفعاتی که خط ۴ اجرا می‌شود را محاسبه کنیم، می‌توانیم کل زمانی که در حلقه *for* در طی اجرای کامل *QUICKSORT* صرف می‌شود را محدود کنیم.

لم 7.1

فرض کنید X تعداد مقایسه هایی باشد که در خط ۴ روال *PARTITION* در کل اجرای *QUICKSORT* روی یک آرایه *n* عنصری انجام می‌شود. آنگاه زمان اجرای *QUICKSORT* برابر $O(n+X)$ خواهد بود.

اثبات بنا به بحث فوق، *n* فراخوانی *PARTITION* وجود دارد که هر یک مقدار کار ثابتی را انجام می‌دهد و سپس حلقه *for* را چندین بار اجرا می‌کند. هر تکرار حلقه *for* خط ۴ را اجرا می‌کند. ■

بنابراین هدف ما، محاسبه X یعنی تعداد کل مقایسه هایی است که در *PARTITION* صورت گرفته است. ما نمی‌خواهیم بررسی کنیم که چند مقایسه در هر فراخوانی *PARTITION* صورت می‌گیرد بلکه می‌خواهیم یک حد کلی برای تعداد کل مقایسه‌ها پیدا کنیم. بدین منظور، باید بفهمیم چه موقع الگوریتم، دو عنصر از آرایه را مقایسه کرده و چه موقع نمی‌کند. برای راحتی تحلیل، عناصر آرایه *A* را به صورت $z_1, z_2, \ldots, z_n$ نامگذاری می‌کنیم بطوریکه z_i، *i* امین عنصر کوچک است. همچنین مجموعه $Z_{ij}=\{z_i, z_{i+1}, \ldots, z_j\}$ را به عنوان مجموعه‌ای که عناصر بین z_i و z_j را شامل می‌شود تعریف می‌کنیم.

چه هنگام، الگوریتم z_j و z_i را مقایسه می‌کند؟ برای جواب دادن به این سؤال، در ابتدا می‌بینیم که هر جفت از عناصر حداکثر یک بار مقایسه می‌شوند. چرا؟ عناصر فقط با عنصر محوری مقایسه می‌شوند و بعد از اینکه یک فراخوانی خاص *PARTITION* پایان یافت، عنصر محوری که در آن فراخوانی استفاده شده بود هیچگاه دوباره با عناصر دیگر مقایسه نمی‌شود.

و تحلیل ما از متغیرهای تصادفی شاخص استفاده می‌کند (بخش ۵.۲ را ملاحظه نمایید).

تعریف می‌کنیم $X_{ij}=I$ {z_i با z_j مقایسه می‌شود}

که در آن در نظر می‌گیریم که آیا مقایسه در یک زمان طی اجرای الگوریتم، نه فقط در طی یک تکرار یا یک فراخوانی *PARTITION* رخ داده است یا خیر. از آنجایی که هر جفت حداکثر یکبار مقایسه می‌شود، می‌توانیم براحتی تعداد کل مقایسه‌هایی که توسط الگوریتم صورت گرفته‌اند را مشخص

کنیم:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} .$$

با گرفتن انتظار (امید ریاضی) از طرفین و سپس استفاده از خطی بودن انتظار و لم ۵.۱، نتیجه می‌شود.

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{\text{مقایسه می‌شود } z_j \text{ با } z_i\} . \end{aligned} \quad (۷.۲)$$

تنها محاسبه $Pr\{$ z_i با z_j مقایسه می‌شود $\}$ باقی می‌ماند.

خوب است در مورد زمانی که دو داده مقایسه نمی‌شوند فکر کنیم. یک ورودی از اعداد بین *1* تا *10* (در هر ترتیبی) برای مرتّب‌سازی سریع در نظر بگیرید و فرض کنید اولین عنصر محوری *7* است. سپس اولین فراخوانی *PARTITION*، اعداد را به دو مجموعه $\{1, 2, 3, 4, 5, 6\}$ و $\{8, 9, 10\}$ تقسیم می‌کند. بنابراین با این کار، عنصر محوری *7* با همه عناصر دیگر مقایسه شده است ولی هیچکدام از اعداد مجموعه اول (به عنوان مثال *2*) هیچگاه با هیچکدام از عناصر مجموعه دوم (به عنوان مثال *9*) مقایسه نشده است و نخواهد شد.

در حالت کلی، هنگامیکه عنصر محوری x به طوریکه $z_i < x < z_j$ انتخاب می‌شود، می‌دانیم که z_i و z_j نمی‌توانند در هیچ زمانی مقایسه شوند. از طرف دیگر اگر z_i به عنوان عنصر محوری، قبل از هر عنصر دیگری در Z_{ij} انتخاب شود، آنگاه z_i با هر عنصری در Z_{ij} به جز با خودش مقایسه می‌شود. بطور مشابه، اگر z_j به عنوان عنصر محوری قبل از هر عنصر دیگری در Z_{ij} انتخاب شود، آنگاه z_j با هر عنصر دیگری در Z_{ij} به جز خودش مقایسه خواهد شد. در مثال ما، مقادیر *7* و *9* مقایسه می‌شوند، زیرا *7* اولین عنصر از $Z_{7,9}$ است که به عنوان عنصر محوری انتخاب می‌شود. در مقابل، *2* و *9* هیچگاه مقایسه نخواهند شد زیرا اولین عنصر محوری که از $Z_{2,9}$ انتخاب شده است *7* می‌باشد. بنابراین z_i و z_j مقایسه می‌شوند اگر و تنها اگر اولین عنصر که به عنوان عنصر محوری از Z_{ij} انتخاب می‌شود، z_i یا z_j باشد.

اکنون احتمال رخداد این امر را محاسبه می‌کنیم. قبل از این که عنصری از Z_{ij} به عنوان عنصر محوری انتخاب شود، کل مجموعه Z_{ij} در یک تقسیم‌بندی قرار می‌گیرد. بنابراین هر عنصر Z_{ij} احتمال یکسانی برای انتخاب شدن به عنوان اولین عنصر محوری دارد. به دلیل اینکه مجموعه Z_{ij}، *j-i+1*

عنصر دارد احتمال اینکه هر عنصر داده شده اولین انتخاب به عنوان عنصر محوری باشد $1/(j-i+1)$ است. پس داریم

$Pr\{$ مقایسه شود z_j با $z_i\} = Pr\{$ هستند Z_{ij} اولین عنصر محوری انتخاب شده از z_j یا $z_i\}$

$= Pr\{$ است Z_{ij} اولین عنصر محوری انتخاب شده از $z_i\}$

$+ Pr\{$ است Z_{ij} اولین عنصر محوری انتخاب شده از $z_j\}$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1}$$

$$= \frac{2}{j-i+1} \tag{۷.۳}$$

خط دوم ثابت می‌شود زیرا دو رویداد ناسازگار هستند. با ترکیب معادلات (۷.۲) و (۷.۳) داریم:

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

می‌توانیم با تغییر متغیر $(k=j-i)$ و حد سریهای هارمونیک این مجموع را محاسبه کنیم:

$$\begin{aligned} \mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\ &< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\ &= \sum_{i=1}^{n-1} O(\lg n) \\ &= O(n \lg n) . \end{aligned} \tag{۷.۴}$$

بنابراین نتیجه می‌گیریم که با استفاده از *RANDOMIZED-PARTITION*، زمان اجرای مورد انتظار برای مرتب‌سازی سریع برابر $O(n\lg n)$ است.

تمرین‌ها

۱- ۷.۴ نشان دهید در رابطه بازگشتی

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

$$T(n) = \Omega(n^2).$$

۷.۴-۲ نشان دهید زمان اجرا در بهترین حالت برای مرتب‌سازی سریع $\Omega(nlgn)$ است.

۷.۴-۳ نشان دهید که $q^2+(n-q-1)^2$ روی $q=0, 1, . . . , n-1$ هنگامی که $q = 0$ یا $q = n-1$ می‌باشد، ماکزیمم است.

۷.۴-۴ نشان دهید زمان اجرای مورد انتظار برای *RANDOMIZED-QUICKSORT*، برابر $\Omega(ngln)$ است.

۷.۴-۵ زمان اجرای مرتب‌سازی سریع در عمل می‌تواند با استفاده از زمان اجرای سریع مرتّب‌سازی درجی هنگامیکه آرایه ورودی "تقریباً" مرتب شده است، بهبود یابد. فرض کنید هنگامیکه مرتب‌سازی سریع برای یک زیر آرایه با کمتر از k عنصر فراخوانی می‌شود بدون مرتب کردن زیر آرایه بازگشت داده شود پس از اینکه فراخوانی سطح بالای مرتب‌سازی سریع بازگشت می‌کند مرتّب‌سازی درجی را در کل آرایه برای پایان دادن به فرآیند مرتب‌سازی اجرا کنید. ثابت کنید این الگوریتم مرتّب‌سازی در زمان مورد انتظار $O(nk+nlg(n/k))$ اجرا می‌شود. در تئوری و در عمل، مقدار k چگونه باید در نظر گرفته شود؟

۷.۴-۶* تغییر روال *PARTITION* را در نظر بگیرید، به صورتی که 3 عنصر از آرایه A را بصورت تصادفی انتخاب و تقسیم‌بندی را حول میانه آنها (مقدار وسط 3 عنصر) انجام دهیم. احتمال تقسیم‌بندی با نسبت α به $(1-\alpha)$ را در بدترین حالت، به صورت تابعی از α در بازه $o<\alpha<1$ تقریب بزنید.

مسائل

۷-۱ درستی تقسیم‌بندی *Hoare*

نسخه‌ای از *PARTITION* که در این بخش ارائه شده است، الگوریتم تقسیم‌بندی اصلی نیست. در اینجا الگوریتم تقسیم‌بندی اصلی آورده شده که منتسب به *T.Hoare* است:

```
HOARE-PARTITION(A, p, r)
 1  x ← A[p]
 2  i ← p − 1
 3  j ← r + 1
 4  while TRUE
 5       do repeat j ← j − 1
 6            until A[j] ≤ x
 7          repeat i ← i + 1
 8            until A[i] ≥ x
 9          if i < j
10            then exchange A[i] ↔ A[j]
11            else return j
```

a.عمل *HOARE -PARTITION* را روی آرایه $A=<13,19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21>$ با نشان دادن مقادیر آرایه و مقادیر کمکی بعد از هر تکرار حلقه *for* خطوط ۱۱-۴ شرح دهید.

سه سؤال بعد از شما می‌خواهند که بطور دقیق درستی روال *HOARE-PARTITION* را اثبات کنید. عبارات زیر را ثابت کنید.

b.اندیس‌های i و j به شکلی هستند که هیچگاه به عنصری از A خارج از زیر آرایه $A[p..r]$ دسترسی نداریم.

c.هنگامیکه *HOARE-PARTITION* پایان می‌یابد، مقدار j را برمی‌گرداند که $p \le j < r$.

d.هنگامیکه *HOARE-PARTITION* پایان می‌یابد، هر عنصر $A[p..j]$ کوچک‌تر یا مساوی با هر عنصر از $A[j+1..r]$ است. روال *PARTITION* در بخش ۷.۱ مقدار محوری (در ابتدا مقدار موجود در $A[r]$) را از دو تقسیم‌بندی که به وجود می‌آورد جدا می‌کند. از طرف دیگر روال *HOARE-PARTITION* همیشه مقدار محوری (در ابتدا مقدار موجود در $A[p]$) را در یکی از دو تقسیم‌بندی $A[p..j]$ و $A[j+1..r]$ قرار می‌دهد. از آنجا که $p \le j < r$ این تقسیم‌بندی همیشه کل آرایه را در برمی‌گیرد.

e.*QUICKSORT* را با استفاده از *HOARE-PARTITION* بازنویسی کنید.

۷-۲ تحلیل دیگر مرتّب‌سازی

یک تحلیل دیگر زمان اجرای مرتب‌سازی سریع تصادفی، به جای تأکید بر تعداد مقایسه‌های انجام شده، به زمان اجرای مورد انتظار در هر تک فراخوانی بازگشتی *QUICKSORT* تأکید دارد.

a.ثابت کنید در آرایه‌ای با اندازه n احتمال اینکه هر عنصر خاص به عنوان عنصر محوری انتخاب شود $1/n$ است. از این موضوع برای تعریف متغیرهای تصادفی شاخص $X_i=I\{i$ امین عنصر کوچک به عنوان عنصر محوری انتخاب می‌شود$\}$ استفاده کنید. $E[X_i]$ چیست؟

b.فرض کنید $T(n)$ متغیری تصادفی است که زمان اجرای مرتّب‌سازی سریع را روی آرایه n عنصری مشخص می‌کند. ثابت کنید

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q\left(T(q-1)+T(n-q)+\Theta(n)\right)\right] \qquad (۷.۵)$$

c.نشان دهید معادله (۷.۵) به صورت زیر ساده می‌شود

$$\mathrm{E}[T(n)] = \frac{2}{n}\sum_{q=0}^{n-1}\mathrm{E}[T(q)] + \Theta(n)\,. \qquad (۷.۶)$$

d.نشان دهید که

$$\sum_{k=1}^{n-1} k \lg k \le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 . \qquad (۷.۷)$$

(راهنمایی: سری را به دو بخش تقسیم کنید، یکی برای $k = 1,2, \ldots , \lceil n/2 \rceil -1$ و یکی برای $k = \lceil n/2 \rceil , \ldots , n-1$).

e.با استفاده از حد معامله (۷.۷) نشان دهید که رابطه بازگشتی معادله (۷.۶) دارای جواب $E[T(n)] = \Theta(n\lg n)$ می‌باشد. (راهنمایی: با استفاده از جایگذاری نشان دهید که برای ثابت‌های مثبت a,b $E[T(n)] \le an \lg n - bn$ است.)

۷-۳ مرتّب سازی ساده لوحانه[1]

پروفسور *Howard* و پروفسور *Fine* الگوریتم مرتب‌سازی "عالی" زیر را پیشنهاد کرده‌اند.

```
STOOGE-SORT(A, i, j)
1  if A[i] > A[j]
2      then exchange A[i] ↔ A[j]
3  if i + 1 ≥ j
4      then return
5  k ← ⌊(j − i + 1)/3⌋              ▷ Round down.
6  STOOGE-SORT(A, i, j − k)         ▷ First two-thirds.
7  STOOGE-SORT(A, i + k, j)         ▷ Last two-thirds.
8  STOOGE-SORT(A, i, j − k)         ▷ First two-thirds again.
```

a.ثابت کنید اگر $n = length[A]$ باشد، آنگاه فرمول به درستی آرایه ورودی $A[1 \,..\, n]$ را مرتب کند.

b.یک رابطه بازگشتی برای زمان اجرای بدترین حالت *STOOGE-SORT* و یک حد مجانبی دقیق (نماد Θ) برای زمان اجرای بدترین حالت ارائه دهید.

c.زمان اجرای بدترین حالت *STOOGE-SORT* را با زمان اجرای بدترین حالت مرتب‌سازی‌های درجی، ادغام، *heap* و مرتب‌سازی سریع مقایسه کنید.

۷.۴ عمق پشته برای مرتب‌سازی سریع

الگوریتم مرتّب‌سازی سریع بخش ۷.۱ شامل دو فراخوانی بازگشتی به خودش است. بعد از فراخوانی *PARTITION* زیرآرایه چپ و سپس زیرآرایه راست به صورت بازگشتی مرتب می‌شوند. دومین

---

1. *Stooge sort*

فراخوانی بازگشتی در *QUICKSORT* در حقیقت لازم نیست. می‌توان با استفاده از یک ساختار کنترل تکراری از این عمل صرف‌نظر کرد. این تکنیک که ***بازگشت در انتها***[1] نامیده می‌شود، به صورت خود کار با کامپایلرهای خوب ایجاد می‌شود. نسخه زیر از مرتب‌سازی سریع را در نظر بگیرید، که بازگشت در انتها را شبیه‌سازی می‌کند.

```
QUICKSORT'(A, p, r)
1  while p < r
2      do ▷ Partition and sort left subarray.
3         q ← PARTITION(A, p, r)
4         QUICKSORT'(A, p, q − 1)
5         p ← q + 1
```

a. ثابت کنید که *QUICKSORT'(A, 1, length[A])* به درستی آرایه *A* را مرتب می‌کند.

کامپایلرها معمولاً روال‌های بازگشتی را با استفاده از پشته‌ای که اطلاعات وابسته، شامل مقادیر پارامترها، را در برمی‌گیرد، برای هر فراخوانی بازگشتی اجرا می‌کنند. اطلاعات آخرین فراخوانی در بالای پشته و اطلاعات اولین فراخوانی در انتهای آن قرار می‌گیرد. هنگامی که یک روال فراخوانی می‌شود اطلاعات آن وارد پشته می‌شود؛ هنگامیکه پایان می‌یابد اطلاعات آن از پشته خارج می‌شود. از آنجایی که فرض کردیم پارامترهای آرایه توسط اشاره‌گرها نمایش داده می‌شوند، اطلاعات برای هر فراخوانی روال پشته نیاز به فضای پشته *O(1)* دارد. عمق پشته، ماکزیمم مقدار فضای پشته‌ای است که در هر زمان در طی یک محاسبه استفاده می‌شود.

b. روندی را توضیح دهید که در آن عمق پشته *QUICKSORT'* روی آرایه ورودی *n* عضوی برابر *Θ(n)* است.

c. برنامه *QUICKSORT'* را طوری تغییر دهید که عمق پشته در بدترین حالت *Θ(lgn)* باشد. زمان اجرای مورد انتظار *O(nlgn)* برای الگوریتم را حفظ کنید.

۷-۵ تقسیم‌بندی میانه 3 عنصر

یک راه برای بهبود روال *RANDOMIZED-QUICKSORT* این است که تقسیم، حول محوری که دقیق‌تر از انتخاب تصادفی یک عنصر از زیر آرایه انتخاب شده است، انجام گیرد. یک روش معمول روش میانه *3* عنصر است: محور را به عنوان میانه (عنصر وسطی) مجموعه‌ای از *3* عنصر که به صورت تصادفی از زیرآرایه انتخاب شده‌اند انتخاب کنید. (تمرین ۶- ۷.۴ را ملاحظه کنید.) برای این مسئله، فرض می‌کنیم که عناصر آرایه ورودی *A[1 . . n]* متفاوتند و *n ≥ 3* است. آرایه خروجی مرتب

1.tail recursion

شده را با $A'[1 \ . \ . \ n]$ مشخص می‌کنیم و با استفاده از روش میانه 3 عنصر، برای انتخاب عنصر محوری x را $p_i = Pr\{x = A'[i]\}$ تعریف می‌کنیم.

a. فرمول دقیقی برای p_i به صورت تابعی از n و i برای $i = 2, 3, . . ., n-1$ ارائه دهید. (توجه داشته باشید که $p_1 = p_n = 0$)

b. در مقایسه با پیاده‌سازی معمولی، تا چه مقدار شانس انتخاب $x = A' [(n+1)/2]$ میانه $A[1 \ . \ . \ n]$ را به عنوان عنصر محوری را افزایش می‌دهیم؟ فرض کنید $n \to \infty$، نسبت حدی این احتمالات را بیان کنید.

c. اگر یک تقسیم‌بندی خوب به شکل انتخاب $x=A'[i]$ به عنوان عنصر محوری تعریف شود، به طوریکه $n/3 \le i \le 2n/3$، تا چه حد در مقایسه با پیاده‌سازی معمول، شانس بدست آوردن یک تقسیم خوب را افزایش می‌دهیم؟ (راهنمایی: مجموع را با یک عدد صحیح تقریب بزنید.)

d. ثابت کنید هنگامیکه زمان اجرای مرتب‌سازی $\Omega(nlgn)$ می‌باشد، روش میانه 3 عنصر فقط روی ضریب ثابت تاثیر می‌گذارد.

۶-۷ مرتب‌سازی فازی بازه‌ها

مسئله مرتب‌سازی را در نظر بگیرید که اعداد دقیقاً مشخص نیستند. در عوض برای هر عدد بازه‌ای از اعداد حقیقی داریم که عدد به آن تعلق دارد. به عبارت دیگر n بازه بسته به شکل $[a_i \ , \ b_i]$ که $a_i \le b_i$، داده شده است. هدف این است که این بازه‌ها را به صورت فازی مرتب کنیم. یعنی یک جایگشت $\langle i_1, i_2 , . . . , i_n \rangle$ از فواصل ایجاد کنیم، به طوریکه $c_j \in [a_{ij}, b_{ij}]$ وجود دارد که در $c_1 \le c_2 \le . . . \le c_n$ صدق می‌کند.

a. الگوریتمی برای مرتب‌سازی فازی n بازه طراحی کنید. الگوریتم شما باید ساختار کلی الگوریتمی را داشته باشد که نقاط پایانی چپ (a_iها) را با مرتب‌سازی سریع مرتب کند، ولی از تداخل بازه برای بهبود زمان اجرا بهره ببرد. (هر چه تداخل بیشتروبیشتر شود، مسئله مرتب‌سازی فازی آسان و آسان‌تر می‌شود. الگوریتم شما باید از این تداخل تا اندازه‌ای که وجود دارد استفاده کند.)

b. ثابت کنید که الگوریتم شما در حالت کلی در زمان مورد انتظار $\Theta(nlgn)$ اجرا می‌شود، ولی هنگامیکه تمام فواصل متداخل باشند در زمان $\Theta(n)$ اجرا می‌شود. (یعنی هنگامیکه مقدار xای وجود داشته باشد به طوریکه برای همه iها داشته باشیم $x \in [a_i, b_i]$) الگوریتم شما نباید در این مورد به دقت چک شود؛ به جای آن هنگامیکه مقدار تداخل‌ها افزایش می‌یابد طبیعتاً کارآیی آن نیز بهتر می‌شود.

۸ مرتب‌سازی در زمان خطی

تاکنون چندین الگوریتم مرتب‌سازی که می‌توانند n عدد را در زمان $O(nlgn)$ مرتب کنند معرفی کرده‌ایم. مرتب‌سازی ادغام و مرتب‌سازی *heap* در بدترین حالت به این حد بالا می‌رسند و مرتب‌سازی سریع در حالت میانگین به این حد می‌رسد. به علاوه، برای هر یک از این الگوریتم‌ها می‌توانیم یک توالی از n عدد ورودی ایجاد کنیم که باعث می‌شود الگوریتم در زمان $\Omega(nlgn)$ اجرا شود.

این الگوریتم‌ها در یک ویژگی جالب مشترکند: توالی مرتب شده‌ای که آنها تعیین می‌کنند، فقط بر اساس مقایسه‌هایی بین عناصر ورودی است. به چنین الگوریتم‌های مرتب‌سازی، *مرتب‌سازی‌های مقایسه‌ای* می‌گوییم. تمام الگوریتم‌های مرتب‌سازی که تاکنون معرفی شده‌اند مرتب‌سازی‌های مقایسه‌ای هستند.

در بخش ۸.۱، ثابت خواهیم کرد که هر مرتب‌سازی مقایسه‌ای برای مرتب کردن n عنصر، در بدترین حالت باید $\Omega(nlgn)$ مقایسه انجام دهد. بنابراین مرتب‌سازی‌های ادغام و *heap* به طور مجانبی بهینه هستند و هیچ مرتب‌سازی مقایسه‌ای وجود ندارد که با بیشتر از یک ضریب ثابت، سریع‌تر باشد.

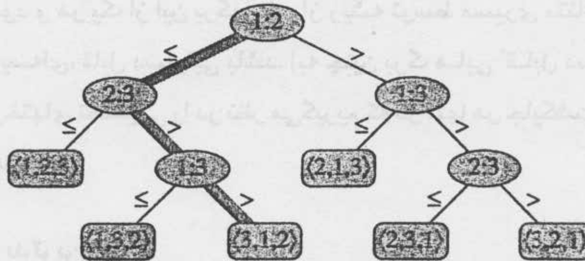
بخش‌های ۸.۲، ۸.۳ و ۸.۴ سه الگوریتم مرتب‌سازی را بررسی می‌کنند - مرتب‌سازی شمارشی، مرتب‌سازی مبنایی و مرتب‌سازی پیمانه‌ای - که در زمان خطی اجرا می‌شوند. بدیهی است که این الگوریتم‌ها به جای مقایسه از اعمال دیگری برای ایجاد توالی مرتب شده استفاده می‌کنند. در نتیجه، حد پایین $\Omega(nlgn)$ در آنها صدق نمی‌کند.

8.1 حدهای پایین برای مرتب‌سازی

در یک مرتب‌سازی مقایسه‌ای، تنها از مقایسه‌های بین عناصر برای دستیابی به اطلاعاتی در مورد ترتیب توالی ورودی $\langle a_1, a_2, \ldots, a_n \rangle$ استفاده می‌کنیم. به عبارت دیگر برای تعیین ترتیب نسبی دو عنصر داده شده a_i و a_j، یکی از بررسی‌های $a_i < a_j$، $a_i \le a_j$، $a_i = a_j$، $a_i \ge a_j$ یا $a_i > a_j$ را انجام می‌دهیم.

ممکن است مقادیر عناصر را بررسی نکنیم یا از هر راه دیگری اطلاعاتی در مورد ترتیب آنها بدست آوریم.

در این بخش بدون از دست دادن کلیّت، فرض می‌کنیم تمام عناصر ورودی متفاوت هستند. با در نظر گرفتن این فرض مقایسه‌هایی به شکل $a_i = a_j$ بی‌فایده هستند، بنابراین می‌توانیم فرض کنیم که هیچ مقایسه‌ای به این شکل صورت نمی‌گیرد. همچنین توجه داریم که مقایسه‌های $a_i \leq a_j$، $a_i \geq a_j$، $a_i > a_j$، $a_i < a_j$ همگی معادلند، چرا که اطلاعات یکسانی در مورد ترتیب نسبی a_i و a_j به ما می‌دهند. بنابراین فرض می‌کنیم تمام مقایسه‌ها به شکل $a_i \leq a_j$ هستند.



شکل ۸.۱ درخت تصمیم برای مرتب‌سازی درجی که روی سه عنصر اعمال شده است. گره داخلی که با $i{:}j$ نشان داده شده است، مقایسه‌ای بین a_i و a_j رانشان می‌دهد. برگی که توسط جایگشت $<\pi(1), \pi(2), \ldots, \pi(n)>$ نشان داده شده است، ترتیب $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$ را نشان می‌دهد. مسیر سایه‌خورده بیانگر تصمیم‌های گرفته شده در هنگام مرتب‌سازی توالی ورودی $<a_1=6,\ a_2=8,\ a_3=5>$ می‌باشد؛ جایگشت $<3, 1, 2>$ در برگ نشان می‌دهد که ترتیب مرتب شده، $a_3=5 \leq a_1=6 \leq a_2=8$ است. $3! = 6$ جایگشت ممکن از عناصر ورودی وجود دارد. بنابراین درخت تصمیم باید حداقل 6 برگ داشته باشد.

مدل درخت تصمیم

مرتب‌سازی‌های مقایسه‌ای را می‌توان به طور خلاصه برحسب *درخت‌های تصمیم* [1] در نظر گرفت. درخت تصمیم، یک درخت دودویی پر است که مقایسه‌های بین عناصر را نمایش می‌دهد. این مقایسه‌ها توسط یک الگوریتم مرتب‌سازی خاص که روی یک ورودی با اندازه داده شده اعمال شده است، ایجاد می‌شوند. از کنترل، جابه جایی داده‌ها و همه جنبه‌های دیگر الگوریتم صرف نظر شده است. شکل ۸.۱ درخت تصمیم را نشان می‌دهد که متناظر است با الگوریتم مرتب‌سازی درجی بخش ۲.۱ که روی یک توالی ورودی با 3 عنصر عمل می‌کند.

در یک درخت تصمیم هر گره داخلی با $i{:}j$ برای یک i و j در بازه $1 \leq i$ و $j \leq n$ نشان داده می‌شود، که n تعداد عناصر در توالی ورودی است. هر برگ با یک جایگشت $<\pi(1), \pi(2), \ldots, \pi(n)>$ نشان داده می‌شود.

1.decision trees

اجرای الگوریتم مرتب‌سازی، متناظر با طی مسیری از ریشه درخت تصمیم تا یک برگ می‌باشد. در هر گره داخلی، مقایسه $a_i \le a_j$ صورت می‌گیرد. آنگاه زیردرخت چپ نتیجه مقایسه $a_i \le a_j$ و زیردرخت راست نتیجه مقایسه $a_i > a_j$ را می‌دهد. هنگامیکه به برگ می‌رسیم، الگوریتم مرتب‌سازی ترتیب $a_{\pi(1)} \le a_{\pi(2)} \le \ldots \le a_{\pi(n)}$ را ایجاد کرده است. چون هر الگوریتم مرتب‌سازی صحیح باید قادر باشد که هر جایگشتی از ورودی هایش را ایجاد کند، یک شرط لازم برای درستی یک مرتب‌سازی مقایسه‌ای این است که هر یک از $n!$ جایگشت روی n عنصر، باید به عنوان یکی از برگ‌های درخت تصمیم ظاهر شود و هر یک از این برگها باید از ریشه توسط مسیری متناظر با یک اجرای واقعی مرتب‌سازی مقایسه‌ای، قابل دستیابی باشد. (به چنین برگ هایی "قابل دستیابی" خواهیم گفت.) بنابراین، فقط درختهای تصمیمی را در نظر می‌گیریم که در آنها هر جایگشت به صورت یک برگ قابل دستیابی ظاهر می‌شود.

حد پایین برای بدترین حالت

طول طولانی‌ترین مسیر از ریشه یک درخت تصمیم تا هر یک از برگ‌های قابل دستیابی آن، تعداد مقایسه‌هایی که الگوریتم مرتب‌سازی متناظر در بدترین حالت انجام می‌دهد را نشان می‌دهد. در نتیجه تعداد مقایسه‌ها برای یک الگوریتم مرتب‌سازی مقایسه‌ای برابر با ارتفاع درخت تصمیم آن می‌باشد. بنابراین حد پایین ارتفاع همه درخت‌های تصمیم که در آنها هر جایگشت بصورت یک برگ قابل دستیابی ظاهر می‌شود، یک حد پایین برای زمان اجرای هر الگوریتم مرتب‌سازی مقایسه‌ای می‌باشد. قضیه زیر چنین حد پایینی را بیان می‌کند.

قضیه ۸.۱

هر الگوریتم مرتب‌سازی مقایسه‌ای در بدترین حالت به $\Omega(nlgn)$ مقایسه نیاز دارد.

اثبات با توجه به بحث قبل، کافی است ارتفاع درخت تصمیم که در آن هر جایگشت بصورت یک برگ قابل دستیابی ظاهر شده است را تعیین کنیم. درخت تصمیمی با ارتفاع h با l برگ قابل دستیابی که متناظر با یک مرتب‌سازی مقایسه‌ای روی n عنصر است را در نظر بگیرید. به دلیل این که هر یک از $n!$ جایگشت ورودی بصورت یک برگ ظاهر می‌شود داریم: $n! \le l$. از آنجا که یک درخت دودویی با ارتفاع h بیشتر از 2^h برگ ندارد داریم

$$n! \le l \le 2^h$$

که با گرفتن لگاریتم خواهیم داشت

$$h \ge lg(n!)$$ (چون تابع lg بصورت یکنواخت افزایش می‌یابد.)

$$= \Omega(nlgn)$$ (بنا به معادله (۳.۱۸))

■

قضیه فرعی ۸.۲

مرتب‌سازی‌های *heap* و ادغام به طور مجانبی مرتب‌سازی‌های مقایسه‌ای بهینه‌ای هستند.

اثبات حدود بالای $O(n\lg n)$ برای زمان‌های اجرای مرتب‌سازی‌های *heap* و ادغام با حد پایین $\Omega(n\lg n)$ در بدترین حالت که از قضیه ۸.۱ حاصل می‌شود، مطابق است. ■

تمرین‌ها

۸.۱-۱ کمترین عمق ممکن یک برگ در یک درخت تصمیم برای مرتب‌سازی‌های مقایسه‌ای چیست؟

۸.۱-۲ حدود به طور مجانبی دقیق روی $\lg(n!)$ را بدون استفاده از تقریب استرلینگ[1] *(Stirling)* بدست آورید. در عوض، مجموع $\sum_{k=1}^{n} \lg k$ را محاسبه نمایید.

۸.۱-۳ نشان دهید که هیچ مرتب‌سازی مقایسه‌ای وجود ندارد که زمان اجرای آن برای حداقل نصف $n!$ ورودی با طول n، خطی باشد. در مورد $1/n$ از ورودی‌ها با طول n چطور؟ در مورد کسر $1/2^n$ چطور؟

۸.۱-۴ یک توالی از n عنصر برای مرتب شدن داده شده است. توالی ورودی شامل n/k زیر توالی می‌باشد، که هر یک شامل k عنصر می‌باشد. همه عناصر یک زیرتوالی داده شده، کوچکتر از عناصر زیرتوالی‌های بعدی و بزرگتر از عناصر زیرتوالی قبلی هستند. بنابراین کل کاری که لازم است برای مرتب کردن کل توالی با طول n انجام دهیم، مرتب کردن k عنصر در هر یک از n/k زیرتوالی است. یک حد پایین $\Omega(n\lg k)$ روی تعداد مقایسه‌هایی که برای حل این مسئله مرتب‌سازی متفاوت نیاز است نشان دهید. (*راهنمایی:* ترکیب کردن حدود پایین برای تک‌تک زیرتوالی‌ها کار دقیقی نیست.)

۸.۲ مرتب‌سازی شمارشی

مرتب‌سازی شمارشی[2] فرض می‌کند که هر یک از n عنصر ورودی یک عدد صحیح دربازه 0 تا k می‌باشد، که k مقداری صحیح است. زمانیکه $k = O(n)$ است الگوریتم مرتب‌سازی در زمان $\Theta(n)$ اجرا می‌شود.

ایده اصلی مرتب‌سازی شمارشی اینست که برای هر عنصر ورودی x، تعداد عناصر کوچکتر از x مشخص شود. این اطلاعات می‌تواند برای قرار دادن x بصورت مستقیم درجایگاهش در آرایه خروجی مورد استفاده قرار گیرد. به عنوان مثال اگر 17 عنصر کوچکتر از x وجود داشته باشد، x به موقعیت 18 خروجی تعلق دارد. این طرح باید بطور دقیقی اصلاح شود تا شرایطی که در آن چندین عنصر، یک مقدار دارند را کنترل کنیم. چون نمی‌خواهیم که همه آنها را در یک موقعیت قرار دهیم.

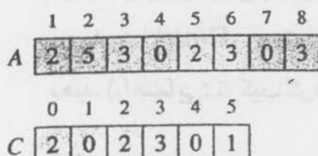
---

1. $n! \approx \sqrt{2\pi}\, n^{(n+1/2)} e^{-n}$

2. counting sort

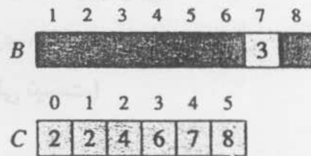
در کد مرتب‌سازی شمارشی، فرض می‌کنیم که آرایه ورودی $A[1\ ..\ n]$ باشد و بنابراین $length[A] = n$ است. به دو آرایه دیگر نیز احتیاج داریم: آرایه $B[1\ ..\ n]$ که خروجی مرتب شده را نگهداری می‌کند و آرایه $C[0\ ..\ k]$ که حافظه کاری موقتی را ایجاد می‌کند.

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 to k
 2       do C[i] ← 0
 3  for j ← 1 to length[A]
 4       do C[A[j]] ← C[A[j]] + 1
 5  ▷ C[i] now contains the number of elements equal to i.
 6  for i ← 1 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] now contains the number of elements less than or equal to i.
 9  for j ← length[A] downto 1
10       do B[C[A[j]]] ← A[j]
11           C[A[j]] ← C[A[j]] − 1
```

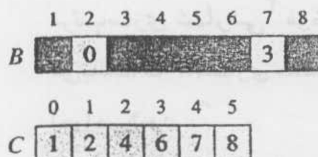


(a)

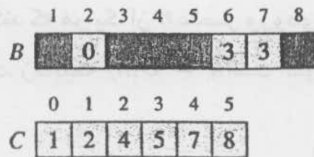
(b)



(c)



(d)



(e)

(f)

شکل ۸.۲ عملکرد *COUNTING-SORT* روی آرایه ورودی $A[1\ ..\ 8]$، که هر عنصر A یک عدد صحیح غیر منفی است که بزرگتر از $k = 5$ نمی‌باشد. *(a)* آرایه A و آرایه کمکی C بعد از خط ۴. *(b)* آرایه C بعد از خط ۷. *(c)-(e)* آرایه خروجی B و آرایه کمکی C بعد از به ترتیب 1، 2 و 3 تکرار حلقه خطوط ۹-۱۱. فقط عناصری از آرایه B که سایه روشن زده شده‌اند پرشده‌اند. *(f)* آرایه نهایی B که خروجی مرتب شده است.

شکل ۸.۲ مرتب‌سازی شمارشی را شرح می‌دهد .بعد از مقدار دهی اولیه در حلقه *for* خطوط ۲-۱، هر عنصر ورودی را در حلقه *for* خطوط ۴-۳ بررسی می‌کنیم. اگر مقدار یک عنصر ورودی، i باشد $C[i]$ را یکی افزایش می‌دهیم. بنابراین بعد از خط ۴، برای هر عدد صحیح $i = 0, 1, \ldots, k$ ، $c[i]$ تعداد عناصر ورودی برابر با i را نگه می‌دارد. در خطوط ۷-۶ با نگهداری یک مجموع در حال اجرای آرایه C، برای هر $i = 0, 1, 2, \ldots, k$ ، تعداد عناصر ورودی کوچکتر یا مساوی با i را مشخص می‌کنیم.

در انتها، در حلقه *for* خطوط ۱۱-۹، هر عنصر $A[j]$ را در موقعیت مرتب شده صحیح خود، در آرایه خروجی B قرار می‌دهیم. اگر تمام n عنصر متفاوت باشند، آنگاه وقتی برای اولین بار به خط ۹ وارد می‌شویم، برای هر $A[j]$ مقدار $C[A[j]]$ موقعیت نهایی صحیح $A[j]$ در آرایه خروجی است، زیرا $C[A[j]]$ عنصر کوچکتر یا مساوی با $A[j]$ وجود دارد. بدلیل اینکه عناصر ممکن است متفاوت نباشند، هر بار که یک مقدار $A[j]$ را در آرایه B قرار می‌دهیم، $C[A[j]]$ را یکی کاهش می‌دهیم. کاهش $C[A[j]]$ باعث می‌شود اگر عنصر ورودی بعدی در صورت وجود دارای مقداری برابر با $A[j]$ است، به موقعیتی برود که بلافاصله قبل از $A[j]$ در آرایه خروجی قرار دارد.

مرتب‌سازی شمارشی، چقدر زمان نیاز دارد؟ حلقه *for* خطوط ۲-۱ زمان $\Theta(k)$، حلقه *for* خطوط ۴-۳ زمان $\Theta(n)$ ، حلقه *for* خطوط ۷-۶ زمان $\Theta(k)$ و حلقه *for* خطوط ۱۱-۹ زمان $\Theta(n)$ را صرف می‌کنند. بنابراین زمان کل برابر $\Theta(k+n)$ است. در عمل، معمولاً از مرتب‌سازی شمارشی زمانی استفاده می‌کنیم که داریم $k = O(n)$، که در این حالت زمان اجرا $\Theta(n)$ است.

مرتب‌سازی شمارشی بر حد پایین $\Omega(nlgn)$ که در بخش ۸.۱ ثابت شد غلبه می‌کند، زیرا یک مرتب‌سازی مقایسه‌ای نیست. در حقیقت، هیچ مقایسه‌ای بین عناصر ورودی در هیچ کجای کد برنامه رخ نمی‌دهد. در عوض مرتب‌سازی شمارشی از مقدار واقعی عناصر برای پیدا کردن اندیس در آرایه استفاده می‌کند. هنگامیکه از مدل مرتب‌سازی مقایسه‌ای خارج می‌شویم، حد پایین $\Omega(nlgn)$ برای مرتب‌سازی بکار نمی‌رود.

یک ویژگی مهم مرتب‌سازی شمارشی اینست که **پایدار**[1] است: اعداد با مقدار یکسان در آرایه خروجی به همان ترتیبی که درآرایه ورودی هستند ظاهر می‌شوند. به این معنا که گره‌های بین دو عدد، توسط قانونی شکسته می‌شوند که می‌گوید هر کدام از اعداد که ابتدا در آرایه ورودی ظاهر می‌شوند، در آرایه خروجی نیز اول ظاهر می‌شوند. معمولاً ویژگی پایداری فقط زمانی مهم است که عناصر مرتب شده همراه با داده‌های وابسته باشند. پایداری مرتب‌سازی شمارشی به دلیل دیگری نیز اهمیت دارد: مرتب‌سازی شمارشی اغلب به عنوان زیرروالی در مرتب‌سازی مبنایی استفاده می‌شود. همانطور که در بخش بعد خواهیم دید، پایداری مرتب‌سازی شمارشی برای صحت مرتب‌سازی مبنایی بسیار مهم است.

---

1. stable

تمرین‌ها

۱- ۸.۲ با استفاده از شکل ۸.۲ به عنوان نمونه، عملکرد *COUNTING-SORT* را روی آرایه $A = <6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2>$ نمایش دهید.

۲- ۸.۲ ثابت کنید *CONUNTING-SORT* پایدار است.

۳- ۸.۲ فرض کنید ابتدای حلقه *for* در خط ۹ روال *CONUNTING-SORT* بصورت زیر بازنویسی شده است.

9 for j ← 1 to length[A]

نشان دهید که الگوریتم هنوز به درستی کار می‌کند. آیا الگوریتم تغییر یافته پایدار است؟

۴- ۸.۲ الگوریتمی را بیان کنید که با دریافت n عدد صحیح در بازه 0 تا k، ورودی‌اش را پیش پردازش کند و سپس به هر پرس‌وجو در مورد اینکه چه تعداد از این n عدد صحیح در بازه $[a\ ..\ b]$ قرار دارند، در زمان $O(1)$ پاسخ دهد. الگوریتم شما باید از زمان پیش‌پردازش $\Theta(n+k)$ استفاده کند.

۸.۳ مرتب‌سازی مبنایی

مرتب‌سازی مبنایی[1] الگوریتمی است که توسط ماشین‌های مرتب‌سازی کارت که اکنون آنها را تنها در موزه‌های کامپیوتر پیدا می‌کنید، استفاده می‌شد. کارتها در *80* ستون سازماندهی می‌شوند و در هر ستون، یک سوراخ می‌تواند در یکی از *12* مکان ایجاد شود. مرتب کننده می‌تواند بطور مکانیکی طوری برنامه‌ریزی شود که ستون داده شده هر کارت در یک دسته را بررسی کند و با توجه به اینکه کدام مکان سوراخ شده است، کارت را در یکی از *12* مکان قرار دهد. یک اپراتور می‌تواند کارتها را مکان به مکان جمع‌آوری کند، بطوریکه کارتهایی که در اولین مکان سوراخ شده قرار دارند، در بالای کارتهایی که در دومین مکان سوراخ شده هستند قرار می‌گیرند و به همین ترتیب.

329		720		720		329
457		355		329		355
657		436		436		436
839	→	457	→	839	→	457
436		657		355		657
720		329		457		720
355		839		657		839

شکل ۸.۳ عملکرد مرتب‌سازی مبنایی روی لیستی شامل هفت عدد *3* رقمی. سمت چپ‌ترین ستون، ورودی است. ستونهای دیگر، لیست را پس از مرتب سازی‌های پیاپی روی مکانهای بطور صعودی با ارزش رقم‌ها نشان می‌دهند. قسمت سایه خورده، مکان رقمی که برای تولید هر لیست از لیست قبلی، مرتب شده است را نشان می‌دهد.

[1] *radix sort*

برای ارقام دهدهی، فقط 10 مکان در هر ستون استفاده می‌شود. (دو مکان دیگر برای کدگذاری کاراکترهای غیر عددی استفاده می‌شوند.) پس یک عدد d رقمی، فیلدی با d ستون را اشغال می‌کند. از آنجایی که مرتب کننده کارت در هر زمان فقط می‌تواند یک ستون را بررسی کند، مسئله مرتب‌سازی n کارت روی یک عدد d رقمی، نیاز به یک الگوریتم مرتب‌سازی دارد.

بطور شهودی، ممکن است کسی بخواهد اعداد را بر مبنای با ارزش‌ترین رقم آنها مرتب کند، کارتهای هر یک از مکان‌های حاصله را بصورت بازگشتی مرتب کرده و سپس دسته‌ها را به ترتیب، ترکیب کند. متأسفانه از آنجا که کارتها در 9 مکان از 10 مکان باید برای مرتب کردن کارتهای هر یک از مکان‌ها کنار گذاشته شوند، این روال دسته کارتهای میانی بسیاری را ایجاد می‌کند که باید از آنها مطلع باشیم. (تمرین ۵- ۸.۳ را ملاحظه نمایید.)

مرتب‌سازی مبنایی، مسئله مرتب‌سازی کارت را بطور شمارشی ابتدا با استفاده از مرتب‌سازی بر مبنای کم ارزش‌ترین رقم حل می‌کند. سپس کارتها در یک دسته ترکیب می‌شوند، بطوریکه کارتها در مکان 0 نسبت به کارتهایی که در مکان 1 قرار دارند مقدم هستند و کارت هایی که در مکان 1 هستند نسبت به کارت هایی که در مکان 2 هستند تقدم دارند و . . . . سپس کل دسته دوباره بر مبنای کم ارزش‌ترین رقم دوم مرتب شده و به روشی مشابه ترکیب می‌شود. این فرآیند تا زمانی ادامه می‌یابد که کارت‌ها بر مبنای تمام d رقم مرتب شوند. بطور مشخص در آن هنگام، کارتها بطور کامل بر مبنای d رقم عدد مرتب شده‌اند. بنابراین فقط d گذر در دسته لازم است تا کارت‌ها مرتب شوند. شکل ۸.۳ نشان می‌دهد که چطور مرتب‌سازی مبنایی روی "دسته"ای با هفت عدد 3 رقمی عمل می‌کند.

لازم است رقمی که در این الگوریتم مرتب می‌شود پایدار باشد. مرتب‌سازی که توسط یک مرتب کننده کارت انجام می‌گیرد پایدار است، ولی اپراتور باید مواظب باشد که ترتیب کارتها به همان ترتیبی باشد که از یک مکان بیرون آمده‌اند، حتی اگر همه کارتهای یک مکان در ستون انتخاب شده دارای یک رقم مشابه باشند.

در یک کامپیوتر معمولی، که یک ماشین با دستیابی تصادفی ترتیبی است، گاهی اوقات مرتب‌سازی مبنایی برای مرتّب کردن رکوردهای اطلاعاتی که توسط چندین فیلد کلید گذاری می‌شوند استفاده می‌شود. به عنوان مثال، ممکن است بخواهیم تاریخ را با سه کلید مرتب کنیم: سال، ماه و روز. می‌توانیم یک الگوریتم مرتب‌سازی را با یک تابع مقایسه اجرا کنیم، که با دریافت دو تاریخ، سال‌ها را مقایسه می‌کند و اگر برابر باشند، ماه‌ها را مقایسه می‌کند و اگر آنها نیز با هم برابر باشند، روزها را مقایسه می‌کند. متناوباً می‌توانیم اطلاعات را 3 بار با یک مرتب‌سازی پایدار مرتب کنیم: ابتدا بر مبنای روز، سپس بر مبنای ماه و در آخر بر مبنای سال.

کد مرتب‌سازی مبنایی ساده است. در روال زیر فرض شده است که هر عنصر در آرایه n عضوی A، d رقم دارد که رقم 1، رقم با پایین‌ترین مرتبه و رقم d، رقم با بالاترین مرتبه است.

```
RADIX-SORT(A, d)
1  for i ← 1 to d
2       do use a stable sort to sort array A on digit i
```

لم ۸.۳

با دریافت اعداد n رقمی که در آنها هر رقم می‌تواند تا k مقدار ممکن را بگیرد، *RADIX-SORT* این اعداد را به درستی در زمان $\Theta(d(n+k))$ مرتب می‌کند.

اثبات صحت مرتب‌سازی مبنایی با استقرا روی ستون‌هایی که مرتب شده‌اند از اثبات می‌شود. (تمرین ۳- ۸.۳ را ملاحظه کنید.) تحلیل زمان اجرا به مرتب‌سازی پایداری که به عنوان الگوریتم مرتب‌سازی واسطه استفاده شده است بستگی دارد. هنگامیکه هر رقم در بازه 0 تا $k-1$ است (بطوریکه می‌تواند k مقدار ممکن را بگیرد) و k بسیار بزرگ است، مرتب‌سازی شمارشی انتخابی بدیهی است. پس هر گذر روی n عدد d رقمی، زمان $\Theta(n+k)$ را صرف می‌کند. d گذر وجود دارد لذا زمان کل برای مرتب‌سازی مبنایی برابر $\Theta(d(n+k))$ است. ■

هنگامیکه d ثابت است و $k = O(n)$، مرتب‌سازی مبنایی در زمانی خطی اجرا می‌شود. در حالت کلی‌تر، در چگونگی شکستن هر کلید به ارقام، انعطاف‌پذیری داریم.

لم ۸.۴

با دریافت n عدد b بیتی و هر عدد صحیح مثبت r که $r \le b$، *RADIX-SORT* این اعداد را به درستی در زمان $\Theta((b/r)(n+2^r))$ مرتب می‌کند.

اثبات برای یک مقدار $r \le b$ مشاهده می‌کنیم که هر کلید، $d = \lceil b/r \rceil$ رقم r بیتی دارد. هر رقم، یک عدد صحیح در بازه 0 تا 2^n-1 است، لذا می‌توانیم از مرتب‌سازی شمارشی با $k = 2^r-1$ استفاده کنیم. (به عنوان مثال، می‌توانیم یک کلمه 32 بیتی را بصورت یک کلمه که دارای 4 رقم 8 بیتی است در نظر بگیریم، بطوریکه $b=32$ ، $r=8$ ، $k=2^r-1=255$ و $d=b/r=4$.) هر گذر مرتب‌سازی شمارشی زمان $\Theta(n+k)=\Theta(n+2^r)$ را صرف می‌کند و d گذر وجود دارد، که زمان کل اجرا برابر $\Theta(d(n+2^r))=\Theta((b/r)(n+2^r))$ است. ■

برای مقادیر داده شده n و b می‌خواهیم مقداری از r که $r \le b$ را انتخاب کنیم که عبارت $(b/r)(n+2^r)$ را مینیمم کند. اگر $b < \lfloor lgn \rfloor$ باشد آنگاه برای هر مقدار $r \le b$ داریم $(n+2^r)=\Theta(n)$. بنابراین انتخاب $r=b$، زمان اجرای $(b/b)(n+2^b)=\Theta(n)$ را نتیجه می‌دهد، که بطور مجانبی بهینه است. اگر $b \ge \lfloor lgn \rfloor$ باشد، آنگاه انتخاب $r=\lfloor lgn \rfloor$ بهترین زمان اجرا را با یک ضریب ثابت حاصل می‌کند که بصورت زیر

می‌توانیم مشاهده کنیم. با انتخاب $r=\lfloor lgn \rfloor$، زمان اجرای $\Theta(bn/lgn)$ بدست می‌آید. همان طور که r را به مقداری بیشتر از $\lfloor lgn \rfloor$ افزایش می‌دهیم، جمله 2^r در صورت، سریعتر از جمله r در مخرج افزایش می‌یابد و بنابراین افزایش r به مقداری بیشتر از $\lfloor lgn \rfloor$، زمان اجرای $\Omega(bn/lgn)$ را نتیجه می‌دهد. در عوض اگر r را به کمتر از $\lfloor lgn \rfloor$ کاهش دهیم، آنگاه جمله b/r افزایش می‌یابد و جمله $n+2^r$ در $\Theta(n)$ باقی می‌ماند.

آیا مرتب‌سازی مبنایی به یک الگوریتم مرتب‌سازی بر مبنای مقایسه، مانند مرتب‌سازی سریع، قابل ترجیح است؟ اگر $b=O(lgn)$، چنانکه اغلب این حالت رخ می‌دهد، و انتخاب کنیم $r\approx lgn$، آنگاه زمان اجرای مرتب‌سازی مبنایی برابر $\Theta(n)$ است که به نظر می‌رسد از زمان اجرای مرتب‌سازی سریع در حالت میانگین یعنی $\Theta(nlgn)$ بهتر است. هر چند ضرایب ثابت که در نماد Θ پنهان هستند، متفاوتند. اگر چه مرتب‌سازی مبنایی ممکن است گذرهای کمتری نسبت به مرتب‌سازی سریع روی n کلید انجام دهد، هر گذر مرتب‌سازی مبنایی ممکن است بطور قابل توجهی طولانی‌تر باشد. اینکه کدام الگوریتم مرتب‌سازی ارجح است، به مشخصات پیاده‌سازی‌های ماشین مورد استفاده (مثلاً مرتب‌سازی سریع اغلب از حافظه‌های پنهان سخت‌افزاری به شکل کارآمدتری نسبت به مرتب‌سازی مبنایی استفاده می‌کند) و داده ورودی بستگی دارد. علاوه براین نسخه‌ای از مرتب‌سازی مبنایی که از مرتب‌سازی شمارشی به عنوان مرتب سازی پایدار واسطه استفاده می‌کند، به صورت *درجا* مرتب‌سازی را انجام نمی‌دهد، در صورتی که بسیاری از مرتب سازی‌های مقایسه‌ای با زمان $\Theta(nlgn)$، این کار را انجام می‌دهند. بنابراین هنگامیکه ذخیره حافظه اصلی با ارزش است، یک الگوریتم درجا مانند مرتب‌سازی سریع ممکن است ارجح باشد.

تمرین‌ها

۸.۳-۱ با استفاده از شکل ۸.۳ به عنوان نمونه، عملکرد *RADIX-SORT* را روی لیست کلمات انگلیسی زیر شرح دهید.

COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

۸.۳-۲ کدامیک از الگوریم‌های مرتب‌سازی زیر پایدارند:

مرتب‌سازی درجی، مرتب‌سازی ادغام، مرتب‌سازی *heap* و مرتب‌سازی سریع؟ یک طرح ساده ارائه کنید که هر الگوریتم مرتب‌سازی را پایدار می‌کند. طرح شما مستلزم چقدر زمان و فضای اضافی است؟

۸.۳-۳ از استقرا برای اثبات اینکه مرتب‌سازی مبنایی به درستی کار می‌کند استفاده کنید. اثبات شما در کجا به این فرض نیاز دارد که مرتب‌سازی واسطه پایدار است؟

۸.۳-۴ نشان دهید چطور می‌توان n عدد صحیح در بازه 0 تا n^2-1 را در زمان $O(n)$ مرتب کرد.

۵- ۸.۳* در اولین الگوریتم مرتب‌سازی کارت در این بخش، دقیقاً چند گذر مرتب‌سازی برای مرتب کردن اعداد دهدهی d رقمی در بدترین حالت لازم است؟ یک اپراتور لازم است از چند دسته کارت در بدترین حالت مطلع باشد؟

۸.۴ مرتب‌سازی پیمانه‌ای

مرتب‌سازی پیمانه‌ای[1] هنگامیکه ورودی دارای یک توزیع یکنواخت است در زمانی خطی اجرا می‌شود. مانند مرتب‌سازی شمارشی، مرتب‌سازی پیمانه‌ای سریع است، زیرا بعضی چیزها را در مورد ورودی فرض می‌کند. در حالیکه مرتب‌سازی شمارشی فرض می‌کند که ورودی از اعداد صحیح در یک بازه کوچک تشکیل شده است، مرتب‌سازی پیمانه‌ای فرض می‌کند که ورودی با یک فرآیند تصادفی که بطور یکنواخت عناصر را روی بازه $[0, 1)$ توزیع می‌کند ایجاد می‌شود.

ایده مرتب‌سازی پیمانه‌ای این است که بازه $[0, 1)$ را به n زیربازه یا *پیمانه*[2] با اندازه برابر تقسیم کند و سپس n عدد ورودی را در پیمانه‌ها توزیع کند. از آنجایی که ورودی‌ها بطور یکنواخت روی $[0, 1)$ توزیع می‌شوند، انتظار نداریم که اعداد زیادی در هر پیمانه قرار گیرند. برای ایجاد خروجی، بسادگی اعداد در هر پیمانه را مرتب می‌کنیم و سپس به ترتیب به داخل پیمانه‌ها رفته و عناصر را در هر یک، لیست می‌کنیم.

کد ما برای مرتب‌سازی پیمانه‌ای فرض می‌کند که ورودی، آرایه A با n عنصر است و هر عنصر $A[i]$ در آرایه در $0 \le A[i] < 1$ صدق می‌کند. این کد به یک آرایه کمکی $B[0 \,.\,.\, n-1]$ از لیست‌های پیوندی (پیمانه‌ها) نیاز دارد و فرض می‌کند که مکانسیمی برای نگهداری چنین لیست هایی وجود دارد. (بخش ۱۰.۲ چگونگی پیاده‌سازی اعمال پایه روی لیست‌های پیوندی را توضیح می‌دهد.)

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3       do insert A[i] into list B[⌊nA[i]⌋]
4  for i ← 0 to n − 1
5       do sort list B[i] with insertion sort
6  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

شکل ۸.۴ عملکرد مرتب‌سازی پیمانه‌ای روی یک آرایه ورودی با 10 عدد را نشان می‌دهد.

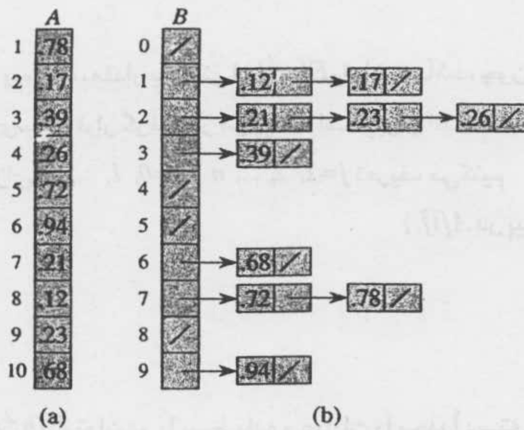
برای اینکه ببینید این الگوریتم کار می‌کند، دو عنصر $A[i]$ و $A[j]$ را در نظر بگیرید. بدون از دست دادن کلّیت فرض می‌کنیم که $A[i] \le A[j]$ است. از آنجا که $\lfloor nA[i] \rfloor \le \lfloor nA[j] \rfloor$ ، عنصر $A[i]$ یا در

---

1. bucket sort   2. buckets

همان پیمانه‌ای که $A[j]$قرار دارد یا در پیمانه‌ای با اندیس کوچکتر قرار می‌گیرد. اگر $A[i]$و $A[j]$در یک پیمانه قرار بگیرند، آنگاه حلقه *for*خطوط ۵-۴، آنها را در ترتیبی صحیح قرار می‌دهد. اگر $A[i]$و $A[j]$ در پیمانه‌های مختلفی قرار بگیرند، آنگاه خط ۶ آنها را در ترتیبی صحیح قرار می‌دهد. بنابراین مرتب‌سازی پیمانه‌ای بدرستی کار می‌کند.

برای تحلیل زمان اجرا، مشاهده می‌کنید که تمام خطوط جز خط ۵ در بدترین حالت زمان $O(n)$ را صرف می‌کنند. فقط باقی می‌ماند کل زمانی که صرف nفراخوانی مرتب‌سازی درجی در خط ۵ می‌شود را محاسبه کنیم.



شکل ۸.۴ عملکرد *BUCKET-SORT*. *(a)* آرایه ورودی $A[1\ .\ .\ 10]$ *(b)* آرایه $B[0\ .\ .\ 9]$ از لیست‌های (پیمانه‌های) مرتب شده بعد از خط ۵ الگوریتم. پیمانه i مقادیر موجود در بازه نیمه باز $[i/10,\ (i+1)/10)$ را نگهداری می‌کند. خروجی مرتب شده از الحاق لیست‌های $B[0], B[1], . . . , B[9]$ به ترتیب، تشکیل شده است.

برای تحلیل هزینه فراخوانی‌های مرتب‌سازی درجی، فرض می‌کنیم n_iیک متغییر تصادفی است که تعداد عناصر موجود در پیمانه $B[i]$ را مشخص می‌کند. از آنجا که مرتب‌سازی درجی در زمان درجهٔ دو اجرا می‌شود (بخش ۲.۲ را ملاحظه کنید) زمان اجرای مرتب‌سازی پیمانه‌ای برابر است با

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

با گرفتن انتظار (امید ریاضی) از هر دو طرف و استفاده از خطی بودن انتظار داریم

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] \qquad \text{(بنا به خطی بودن انتظار)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}[n_i^2]\right) \qquad (۸.۱)
\end{aligned}
$$

برای $i=0, 1, \ldots, n-1$ ادعا می‌کنیم که

$$\mathrm{E}[n_i^2] = 2 - 1/n \qquad (۸.۲)$$

تعجبّی ندارد که هر پیمانه i، مقدار یکسانی از $E[n_i^2]$ را داشته باشد، چون هر مقدار در آرایه ورودی A دارای احتمال برابری برای قرار گرفتن در هر پیمانه است. برای اثبات معادله (۸.۲) متغیرهای تصادفی شاخص زیر را به ازای $i=0, 1, \ldots, n$ و $j=1, 2, \ldots, n$ تعریف می‌کنیم

$X_{ij} = I\{$ در پیمانه i قرار می‌گیرد $A[i]$ $\}$

بنابراین

$$n_i = \sum_{j=1}^{n} X_{ij} \; .$$

برای محاسبه $E[n_i^2]$ توان دو را بسط داده و عبارات را مجدداً دسته بندی می‌کنیم:

$$
\begin{aligned}
\mathrm{E}[n_i^2] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\ne j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} \mathrm{E}\left[X_{ij}^2\right] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\ne j}} \mathrm{E}[X_{ij}X_{ik}] \; , \qquad (۸.۳)
\end{aligned}
$$

که آخرین خط به وسیله خطی بودن انتظار ثابت می‌شود. این دو مجموع را بصورت مجزا ارزشیابی می‌کنیم. متغیر تصادفی شاخص X_{ij} با احتمال $1/n$ برابر با 1 و در غیر اینصورت برابر با 0 است، و بنابراین

$$E[X_{ij}^2] = 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) = \frac{1}{n}.$$

هنگامیکه $k \neq j$ است متغیرهای X_{ij} و X_{ik} مستقلند و از اینرو

$$E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}.$$

با قرار دادن این دو مقدار مورد انتظار در معادله (۸.۳) داریم

$$E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \neq j}} \frac{1}{n^2} = n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n},$$

که معادله (۸.۲) را ثابت می‌کند.

با استفاده از این مقدار مورد انتظار در معادله (۸.۱)، نتیجه می‌گیریم که زمان مورد انتظار برای مرتب سازی پیمانه‌ای برابر $\Theta(n)+n \cdot O(2-1/n)=\Theta(n)$ است. بنابراین کل الگوریتم مرتب‌سازی پیمانه‌ای در زمان مورد انتظار خطی اجرا می‌شود.

حتی اگر ورودی از یک توزیع یکنواخت انتخاب نشود، مرتب‌سازی پیمانه‌ای ممکن است باز هم در زمان خطی اجرا شود. تا زمانیکه ورودی دارای این ویژگی است که مجموع مربعات اندازه‌های پیمانه در کل تعداد عناصر خطی است، معادله (۸.۱) به ما می‌گوید که مرتب‌سازی پیمانه‌ای در زمان خطی اجرا خواهد شد.

تمرین‌ها

۸.۴-۱ با استفاده از شکل ۸.۴ به عنوان مدل، عملکرد *BUCKET-SORT* را روی آرایه $A=<.79, .13, .16, .64, .39, .20, .89, .53, .71, .42>$ شرح دهید.

۸.۴-۲ زمان اجرای بدترین حالت الگوریتم مرتب‌سازی پیمانه‌ای چیست؟ چه تغییر ساده‌ای در الگوریتم، زمان اجرای مورد انتظار خطی آن را حفظ و زمان اجرای بدترین حالت آن را تبدیل به $O(nlgn)$ می‌کند؟

۸.۴-۳ فرض کنید X یک متغیر تصادفی است که برابر با تعداد شیرها در دو پرتاب یک سکه نااریب است. $E[X^2]$ چیست؟ $E^2[X]$ چیست؟

*۸.۴-۴ n نقطه در دایره واحد $p_i=(x_i\ ,\ y_i)$ داده شده‌اند، بطوریکه برای $i=1,\ 2,\ ...,\ n$ $0<x_i^2+y_i^2\leq 1$. فرض کنید که نقاط بصورت یکنواخت توزیع شده‌اند؛ به عبارت دیگر، احتمال یافتن یک نقطه در هر ناحیه از دایره، متناسب با مساحت آن ناحیه است. یک الگوریتم با زمان مورد انتظار $\Theta(n)$ طراحی کنید که n نقطه را بر حسب فاصله‌هایشان از مبدأ، $d_i=\sqrt{x^2_i+y^2_i}$، مرتب کند. (*راهنمایی:* اندازه‌های پیمانه‌ها را طوری طراحی کنید که منعکس کننده توزیع یکنواخت نقاط در دایره واحد باشد.)

*۸.۴-۵ *تابع توزیع احتمال* $P(x)$ [1] برای متغیر تصادفی X با $P(x)=Pr\{X\leq x\}$ تعریف می‌شود. فرض کنید یک لیست از n متغیر تصادفی $X_1,\ X_2,\ ...\ ,\ X_n$ از تابع توزیع احتمال پیوسته P انتخاب شود که در زمان $O(1)$ قابل محاسبه است. نشان دهید چگونه این اعداد را در زمان مورد انتظار خطی مرتب کنیم.

مسائل

۸-۱ حدود پایین حالت میانگین روی مرتب سازی مقایسه‌ای

در این مسئله، حد پایین $\Omega(nlgn)$ را روی زمان اجرای مورد انتظار هر مرتب سازی مقایسه‌ای تصادفی یا قطعی روی n عنصر ورودی مجزا، ثابت می‌کنیم. با بررسی مرتب سازی مقایسه‌ای قطعی A با درخت تصمیم T_A شروع می‌کنیم. فرض می‌کنیم که هر جایگشت از ورودی‌های A دارای احتمال یکسان است.

a. فرض کنید هر برگ T_A با این احتمال که برای یک ورودی تصادفی، مورد دستیابی قرار گیرد، برچسب گذاری می‌شود. ثابت کنید که دقیقاً $n!$ برگ با $1/n!$ برچسب گذاری می‌شوند و برگهای باقیمانده با 0 برچسب گذاری می‌شوند.

b. فرض کنید $D(T)$ طول مسیر خارجی درخت تصمیم T را مشخص می‌کند؛ به عبارت دیگر $D(T)$ برابر با مجموع عمق‌های همه برگهای T است. فرض کنید T یک درخت تصمیم با $k>1$ برگ باشد و LT و RT نیز درختهای چپ و راست T باشند. نشان دهید که

$$D(T) = D(LT) + D(RT) + k$$

---

1. probability distribution function

c.فرض کنید $d(k)$ مینیمم مقدار $D(T)$ در کل درخت‌های تصمیم T با $k>1$ برگ باشد. نشان دهید که $d(k) = min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$

(*راهنمایی:* درخت تصمیم T با k برگ را در نظر بگیرید که به مینیمم می‌رسد. فرض کنید i_0 تعداد برگهای LT و $k-i_0$ تعداد برگهای RT باشد.)

d.ثابت کنید که برای مقدار داده شده $k > 1$ و i در بازه $1 \le i \le k-1$، تابع $ilgi + (k - i) lg (k - i)$ در $i = k/2$ مینیمم می‌شود. نتیجه بگیرید که $d(k) = \Omega(klgk)$ است.

e.ثابت کنید $D(T_A) = \Omega(n! lg(n!))$ و نتیجه بگیرید که زمان مورد انتظار برای زمان مرتب سازی n عنصر برابر $\Omega(nlgn)$ است.

اکنون مرتب سازی مقایسه‌ای تصادفی B را در نظر بگیرید. می‌توانیم مدل درخت تصمیم را برای مدیریت تصادفی سازی، با استفاده از پیوند دو نوع گره توسعه دهیم: گره‌های مقایسه‌ای معمولی و گره‌های "تصادفی ساز". یک گره تصادفی ساز، یک انتخاب تصادفی به شکل $RANDOM(l, r)$ را که توسط الگوریتم B انجام می‌شود مدل سازی می‌کند؛ این گره r فرزند دارد، هر یک از آنها در طول اجرای الگوریتم دارای احتمال یکسانی برای انتخاب شدن هستند.

f.نشان دهید برای هر مرتب سازی مقایسه‌ای تصادفی B، یک مرتب سازی مقایسه‌ای قطعی A وجود دارد که بطور میانگین نسبت به B مقایسه‌های بیشتری را انجام نمی‌دهد.

۸-۲ مرتب سازی درجا در زمان خطی

فرض کنید آرایه‌ای با n رکورد داده برای مرتب سازی داریم و کلید هر رکورد دارای مقدار 0 یا 1 است. یک الگوریتم برای مرتب سازی چنین مجموعه‌ای از رکوردها ممکن است دارای زیر مجموعه‌ای از ۳ مشخصه مطلوب زیر باشد:

۱. الگوریتم در زمان $O(n)$ اجرا می‌شود.

۲. الگوریتم پایدار است.

۳. الگوریتم بصورت درجا مرتب سازی را انجام می‌دهد، تنها با استفاده از مقدار ثابتی از حافظه به علاوه حافظه آرایه اصلی.

a.الگوریتمی ارائه دهید که در موارد ۱ و ۲ بالا صدق کند.

b.الگوریتمی ارائه دهید که در موارد ۱ و ۳ بالا صدق کند.

c.الگوریتمی ارائه دهید که در موارد ۲ و ۳ بالا صدق کند.

d. آیا هر یک از الگوریتمهای مرتب سازی شما در قسمتهای (a) تا (c) می‌تواند برای مرتب سازی n رکورد با کلیدهای b بیتی با استفاده از مرتب سازی مبنایی در زمان $O(bn)$ استفاده شود؟ چگونگی انجام آن یا عدم امکان آن را توضیح دهید.

e. فرض کنید n رکورد، کلیدهایی در بازه 1 تا k دارند. نشان دهید چطور می‌توان مرتب سازی شمارشی را تغییر داد تا رکوردها بتوانند بصورت درجا در زمان $O(n+k)$ مرتب شوند. ممکن است از $O(k)$ حافظه علاوه بر آرایه ورودی استفاده کنید. آیا الگوریتم شما پایدار است؟ (راهنمایی: برای $k = 3$ چطور اینکار را انجام می‌دهید؟)

۸-۳ مرتب سازی اقلام با طول متغیر

a. به شما آرایه‌ای از اعداد صحیح داده شده است، که اعداد صحیح مختلف ممکن است تعداد ارقام متفاوتی داشته باشند، ولی تعداد کل ارقام تمام اعداد صحیح در آرایه برابر n است. نشان دهید چگونه آرایه را در زمان $O(n)$ مرتب کنیم؟

b. به شما آرایه‌ای از رشته‌ها داده شده است، که رشته‌های مختلف ممکن است تعداد کاراکترهای متفاوتی داشته باشند، ولی تعداد کل کاراکترها در همه رشته‌ها n است. نشان دهید چطور آرایه در زمان $O(n)$ مرتب می‌شود؟

(توجه داشته باشید که ترتیب مورد نظر در اینجا ترتیب الفبایی استاندارد است، برای مثال $a<ab<b$.)

۸-۴ پارچهای آب

فرض کنید به شما n پارچ آب قرمز و n پارچ آب آبی داده می‌شود که تمامی آنها دارای شکل و اندازه متفاوت هستند. همه پارچهای قرمز و همه پارچهای آبی حجم آب متفاوتی را نگه می‌دارند. علاوه براین، به ازای هر پارچ قرمز یک پارچ آبی وجود دارد که همان حجم آب را نگه می‌دارد و برعکس.

وظیفه شما اینست که پارچها را در جفتهایی از پارچهای قرمز و آبی که مقدار آب یکسانی را نگه می‌دارند گروه‌بندی کنید. برای اینکار ممکن است عمل زیر را انجام دهید: یک جفت از پارچها که در آن یکی قرمز و یکی آبی است را بردارید، پارچ قرمز را با آب پر کنید و سپس آب را در پارچ آبی بریزید. این عمل به شما می‌گوید که پارچ آبی می‌تواند آب بیشتری نگه دارد یا پارچ قرمز، یا اینکه گنجایش یکسانی دارند. فرض کنید که چنین مقایسه‌ای یک واحد زمان صرف کند. هدف شما پیدا کردن الگوریتمی است که مینیمم تعداد مقایسه را برای تعیین گروه بندی بکار برد. بیاد داشته باشید که نمی‌توانید بطور مستقیم ۲ پارچ آبی یا ۲ پارچ قرمز را مقایسه کنید.

a. یک الگوریتم قطعی که از $\Theta(n^2)$ مقایسه برای گروه بندی پارچها به جفتها استفاده می‌کند بیان کنید.

b. برای تعداد مقایسه‌هایی که یک الگوریتم که این مسئله را حل می‌کند باید انجام دهد، حد پایین $\Omega(nlgn)$ را ثابت کنید.

c. الگوریتمی تصادفی ارائه کنید که تعداد مقایسه‌های مورد انتظار آن $O(nlgn)$ باشد و ثابت کنید که این حد صحیح است. تعداد مقایسه‌های الگوریتم شما در بدترین حالت چیست؟

۸-۵ مرتب سازی میانگین

فرض کنید به جای مرتب سازی یک آرایه، فقط لازم است که عناصر بطور میانگین افزایش یابند. بطور دقیقتر، آرایه A با n عنصر را k- مرتب[1] می‌نامیم اگر برای تمام $i = 1, 2, \ldots, n\text{-}k$ رابطه زیر برقرار باشد:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} .$$

a. اینکه یک آرایه 1- مرتب باشد یعنی چه؟

b. یک جایگشت از اعداد $1, 2, \ldots, 10$ ارائه دهید که 2- مرتب باشد ولی مرتب شده نباشد.

c. ثابت کنید که یک آرایه n عنصری k- مرتب است، اگر و تنها اگر برای تمام $i = 1, 2, \ldots, n\text{-}k$، $A[i] \leq A[i+k]$ باشد.

d. الگوریتمی ارائه دهید که یک آرایه n عنصری را در زمان $O(n\lg(n/k))$، k - مرتب کند. همچنین می‌توانیم زمانی که k ثابت است، یک حد پایین روی زمان ایجاد یک آرایه k - مرتب نشان دهیم.

e. نشان دهید که یک آرایه k - مرتب با طول n می‌تواند در زمان $O(n\lg k)$ مرتب شود. (*راهنمایی*: از حل تمرین ۸-۵.۶ استفاده کنید.)

f. نشان دهید هنگامیکه k ثابت است، زمان لازم برای k - مرتب کردن یک آرایه n عنصری برابر $\Omega(n\lg n)$ است. (*راهنمایی*: از حل قسمت قبلی همراه با حد پایین روی مرتب سازی مقایسه‌ای استفاده کنید.)

۸-۶ حد پایین روی ادغام لیستهای مرتب

مسئله ادغام دو لیست مرتب اغلب پیش می‌آید. از آن به عنوان یک زیرروال *MERGE-SORT* استفاده می‌شود، روال ادغام دو لیست مرتب به عنوان *MERGE* در بخش ۲.۳.۱ ارائه شد. در این مسئله، نشان خواهیم داد که در بدترین حالت، حد پایین $2n\text{-}1$ روی تعداد مقایسه‌ها برای ادغام دو لیست مرتب که هر یک شامل n داده می‌باشد، لازم است.

ابتدا حد پایین $2n\text{-}o(n)$ برای مقایسه‌ها را، با استفاده از درخت تصمیم نشان خواهیم داد.

a. نشان دهید که، برای $2n$ عدد داده شده $\binom{2n}{n}$ راه ممکن وجود دارد که آنها را به دو لیست مرتب، هر یک شامل n عدد تقسیم کنیم.

b. با استفاده از یک درخت تصمیم، نشان دهید هر الگوریتم که بدرستی دو لیست مرتب را ادغام می‌کند حداقل از $2n\text{-}o(n)$ مقایسه استفاده می‌کند.

---

1. k-sorted

حال حد قویتر $2n-1$ را نشان خواهیم داد.

c. نشان دهید اگر دو عنصر در ترتیب مرتب، متوالی بوده و از دو لیست مختلف باشند، آنگاه باید با هم مقایسه شوند.

d. از پاسَخ خود در قسمت قبلی برای نشان دادن حد پایین $2n-1$ برای مقایسه‌ها جهت ادغام دو لیست مرتب استفاده کنید.

۹ میانه‌ها و شاخص‌های آمار ترتیبی

iامین *شاخص آمار ترتیبی*[۱] یک مجموعه nعضوی، iامین عضو کوچک است. به عنوان مثال، **مینیمم**[۲] یک مجموعه از اعضا، اولین شاخص آمار ترتیبی آن $(i = 1)$ است و ***ماکزیمم***[۳]، nامین شاخص آمار ترتیبی $(i = n)$ است. **میانه**[۴]، بطور غیر رسمی، نقطه میانی مجموعه است. هنگامیکه n فرد است، میانه منحصر بفرد است که در $i = (n+1)/2$ رخ می‌دهد. وقتی n زوج است، دو میانه وجود دارند که در $i = n/2$ و $i = n/2+1$ رخ می‌دهند. بنابراین بدون در نظر گرفتن زوج یا فرد بودن n میانه‌ها در $i = \lfloor(n+1)/2\rfloor$ *(میانه پایین)*[۵] و $i = \lceil(n+1)/2\rceil$ *(میانه بالا)*[۶] رخ می‌دهند. اما برای سادگی در این متن، همواره از عبارت "میانه" برای اشاره به میانه پایین استفاده می‌کنیم.

این فصل مسئله انتخاب iامین شاخص آمار ترتیبی از یک مجموعه با nعضو مجزا را بیان می‌کند. برای راحتی کار فرض می‌کنیم این مجموعه شامل اعداد مجزا باشد، اگر چه بطور مجازی هر چه انجام می‌دهیم، برای وضعیتی که در آن یک مجموعه شامل مقادیر تکراری است نیز توسعه می‌یابد. ***مسئله انتخاب***[۷] می‌تواند بطور رسمی به شکل زیر تعیین شود:

ورودی: مجموعه A با n عدد (مجزا) و عدد i، که $1 \le i \le n$ است.

خروجی: عضو $x \in A$ که بزرگتر از دقیقاً $i-1$ عضو دیگر A می‌باشد.

مسئله انتخاب می‌تواند در زمان $O(nlgn)$ حل شود، چون می‌توانیم اعداد را با استفاده از مرتب‌سازی *heap* یا مرتب‌سازی ادغام مرتب کنیم و سپس بسادگی iامین عنصر در آرایه خروجی را مشخص کنیم. اما الگوریتم‌های سریعتری وجود دارند.

در بخش ۹.۱، مسئله انتخاب مینیمم و ماکزیمم یک مجموعه از اعضا را بررسی می‌کنیم. مسئله جالبتر، مسئله انتخاب کلی است، که در نتیجه دو بخش بررسی می‌شود. بخش ۹.۲ یک الگوریتم عملی را تحلیل می‌کند که در حالت میانگین به زمان اجرای $O(n)$ می‌رسد. بخش ۹.۳ شامل یک الگوریتم است که جنبه‌های نظری بیشتری داشته و در بدترین حالت به زمان اجرای $O(n)$ می‌رسد.

---

1. order statistic
2. minimum
3. maximum
4. median
5. lower median
6. upper median
7. selection problem

۹.۱ مینیمم و ماکزیمم

چه تعداد مقایسه برای تعیین مینیمم یک مجموعه n عضوی لازم است؟ می‌توانیم بسادگی به حد بالای $n-1$ برای مقایسه‌ها برسیم: هر عضو مجموعه را به ترتیب بررسی کرده و کوچکترین عضوی که تاکنون دیده شده است را نگه می‌داریم. در روال زیر، فرض می‌کنیم مجموعه در آرایه A قرار دارد، که $length[A] = n$ است.

```
MINIMUM(A)
1  min ← A[1]
2  for i ← 2 to length[A]
3       do if min > A[i]
4              then min ← A[i]
5  return min
```

قطعاً یافتن ماکزیمم می‌تواند با $n-1$ مقایسه نیز انجام شود.

آیا این بهترین کاری است که می‌توانیم انجام دهیم؟ بله، چون می‌توانیم به حد پایین $n-1$ برای مقایسه‌ها، برای مسئله تعیین مینیمم برسیم. الگوریتم را در نظر بگیرید که مینیمم را بصورت مسابقه‌ای بین عناصر تعیین می‌کند. هر مقایسه یک بازی در مسابقه است که در آن عنصر کوچکتر از میان دو عنصر، برنده می‌شود. نگرش اصلی اینست که هر عنصر بجز برنده باید حداقل یک بازی را ببازد. از اینرو $n-1$ مقایسه برای تعیین مینیمم لازم است و الگوریتم *MINIMUM* از لحاظ تعداد مقایسه‌های انجام شده، بهینه است.

مینیمم و ماکزیمم همزمان

در برخی کاربردها، باید هم مینیمم و هم ماکزیمم یک مجموعه از n عضو را پیدا کنیم. به عنوان مثال، یک برنامه گرافیکی ممکن است نیاز داشته باشد که مقیاس یک مجموعه از داده‌های $(x\ ,\ y)$ را، برای قرار گرفتن در یک صفحه نمایش چهار ضلعی یا وسایل خروجی گرافیکی دیگر، تغییر دهد. برای این کار، برنامه باید ابتدا مینیمم و ماکزیمم هر مختصات را تعیین کند.

ارائه الگوریتمی که بتواند هم مینیمم و هم ماکزیمم n عضو را با استفاده از $\Theta(n)$ مقایسه، که بطور مجانبی بهینه است، پیدا کند سخت نیست. بسادگی مینیمم و ماکزیمم را بطور مستقل، با استفاده از $n-1$ مقایسه برای هر یک پیدا می‌کند، که در کل $2n-2$ مقایسه انجام می‌دهد.

در حقیقت، حداکثر $3\lfloor n/2 \rfloor$ مقایسه برای پیدا کردن مینیمم و ماکزیمم کافی است. استراتژی اینست که اعضای مینیمم و ماکزیمم را که تا اینجا دیده شده‌اند نگه داریم. به جای اینکه هر عضو ورودی را با

مقایسه با مینیمم و ماکزیمم فعلی پردازش کنیم، که هزینه 2 مقایسه برای هر عضو را صرف می‌کند، اعضا را جفت به جفت مقایسه می‌کنیم. ابتدا جفت عضوها از ورودی را با یکدیگر مقایسه می‌کنیم و سپس عضو کوچکتر را با مینیمم جاری و عضو بزرگتر را با ماکزیمم جاری مقایسه می‌کنیم که هزینه 3 مقایسه برای هر 2 عضو را موجب می‌گردد.

مقدار دهی اولیه مینیمم و ماکزیمم جاری، به زوج یا فرد بودن n بستگی دارد. اگر n فرد باشد، هم مینیمم و هم ماکزیمم را با مقدار اولین عضو مقدار دهی می‌کنیم و سپس باقیمانده اعضا را دو به دو پردازش می‌کنیم. اگر n زوج باشد، یک مقایسه روی 2 عضو اول برای تعیین مقادیر اولیه مینیمم و ماکزیمم انجام می‌دهیم و سپس باقیمانده اعضا را همانند حالت فرد بودن n، دو به دو پردازش می‌کنیم.

حال تعداد کل مقایسه‌ها را تحلیل می‌کنیم. اگر n فرد باشد، آنگاه $3\lfloor n/2 \rfloor$ مقایسه انجام می‌دهیم. اگر n زوج باشد، یک مقایسه اولیه و به دنبال آن $3(n-2)/2$ مقایسه انجام می‌دهیم، که در کل $3n/2-2$ مقایسه انجام می‌دهیم. بنابراین در هر یک از دو حالت، تعداد کل مقایسه‌ها حداکثر $3\lfloor n/2 \rfloor$ است.

تمرین‌ها

۹.۱-۱ نشان دهید که دومین عضو کوچک از بین n عضو می‌تواند با $n+\lceil lgn \rceil-2$ مقایسه در بدترین حالت پیدا شود. (**راهنمایی:** کوچکترین عضو را نیز پیدا کنید.)

۹.۱-۲ * نشان دهید $\lceil 3n/2 \rceil-2$ مقایسه در بدترین حالت برای پیدا کردن مینیمم و ماکزیمم n عدد لازم است. (**راهنمایی:** در نظر بگیرید که چند عدد بصورت بالقوه مینیمم یا ماکزیمم هستند و بررسی کنید که چطور یک مقایسه بر این تعداد تأثیر می‌گذارد.)

۹.۲ انتخاب در زمان خطی مورد انتظار

مسئله انتخاب کلی به نسبت به مسئله ساده پیدا کردن یک مینیمم سخت‌تر به نظر می‌آید. همچنان بصورت شگفت آوری زمان اجرای مجانبی هر دو مسئله یکی است: $\Theta(n)$. در این بخش، یک الگوریتم تقسیم و حل را برای مسئله انتخاب ارائه می‌دهیم. الگوریتم *RANDOMIZED-SELECT* بعد از الگوریتم مرتب سازی سریع در بخش ۷ مدل می‌شود. همانند مرتب سازی سریع، ایده اینست که آرایه ورودی را بطور بازگشتی تقسیم کنیم. ولی بر خلاف مرتب سازی سریع، که هر دو طرف تقسیم‌بندی را بصورت بازگشتی پردازش می‌کند، *RANDOMIZED-SELECT* فقط روی یک طرف تقسیم‌بندی عمل می‌کند. این تفاوت در تحلیل آشکار می‌شود: در حالیکه زمان اجرای مورد انتظار مرتب سازی سریع، $\Theta(nlgn)$ است، زمان مورد انتظار *RANDOMIZED-SELECT*، $\Theta(n)$ می‌باشد.

RANDOMIZED-SELECT از روال *RANDOMIZED-PARTITION* که در بخش ۷.۳ معرفی شد، استفاده می‌کند. بنابراین مانند *RANDOMIZED-QUICKSORT*، این الگوریتم یک الگوریتم

تصادفی است، چون رفتار آن تا یک اندازه توسط خروجی یک مولد اعداد تصادفی تعیین می‌شود. کد زیر برای *RANDOMIZED-SELECT*، i امین عنصر کوچک آرایه $A[p..r]$ را برمی‌گرداند.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p = r
2     then return A[p]
3  q ← RANDOMIZED-PARTITION(A, p, r)
4  k ← q − p + 1
5  if i = k          ▷ the pivot value is the answer
6     then return A[q]
7  elseif i < k
8     then return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

بعد از اینکه *RANDOMIZED-PARTITION* در خط ۳ الگوریتم اجرا می‌شود، آرایه $A[p..r]$ به دو زیرآرایه (شاید خالی) $A[p..q-1]$ و $A[q+1..r]$ تقسیم می‌شود بطوریکه هر عنصر $A[p..q-1]$ کوچکتر یا مساوی با $A[q]$ است، که $A[q]$ نیز به نوبه خود کوچکتر از هر عنصری از $A[q+1..r]$ می‌باشد. همانند مرتب سازی سریع، به $A[q]$ به عنوان عنصر محوری[1] اشاره می‌کنیم. خط ۴ از *RANDOMIZED-SELECT* تعداد k عنصر در زیرآرایه $A[p..q]$ را محاسبه می‌کند، به عبارت دیگر تعداد عناصر در طرف کمتر تقسیم‌بندی، به علاوه یک برای عنصر محوری. سپس خط ۵ چک می‌کند که آیا $A[q]$، i امین عنصر کوچک هست یا نه. اگر باشد آنگاه $A[q]$ برگردانده می‌شود. در غیر اینصورت، الگوریتم تعیین می‌کند که i امین عنصر کوچک در کدام یک از دو زیرآرایه $A[p..q-1]$ و $A[q+1..r]$ قرار دارد. اگر $i < k$ باشد، آنگاه عنصر مورد نظر در طرف کمتر تقسیم‌بندی قرار گرفته و در خط ۸ بطور بازگشتی از زیرآرایه انتخاب می‌شود. اما اگر $i > k$ باشد، آنگاه عنصر مورد نظر در طرف بزرگتر تقسیم‌بندی قرار دارد. از آنجا که تا کنون k مقدار که کوچکتر از i امین عنصر کوچک $A[p..r]$ هستند را می‌شناسیم - یعنی عناصر $A[p..q]$ - عنصر مورد نظر، $(i-k)$ امین عنصر کوچک $A[q+1..r]$ است، که در خط ۹ بطور بازگشتی پیدا می‌شود. به نظر می‌رسد که کد، فراخوانی‌های بازگشتی به زیرآرایه‌ها با 0 عنصر را مجاز نمی‌داند. ولی تمرین ۱-۹.۲ از شما می‌خواهد نشان دهید که این وضعیت نمی‌تواند اتفاق بیفتد.

زمان اجرای *RANDOMIZED-SELECT* در بدترین حالت $\Theta(n^2)$ است، حتی برای یافتن مینیمم، زیرا می‌توانستیم خیلی بد شانس باشیم و همیشه تقسیم‌بندی را حول بزرگترین عنصر باقیمانده انجام دهیم و تقسیم‌بندی، زمان $\Theta(n)$ را صرف می‌کند. اگر چه الگوریتم بخوبی در حالت میانگین کار می‌کند و چون تصادفی است، هیچ ورودی خاصی باعث رفتار بدترین حالت نمی‌شود.

---

1. pivot

زمان مورد نیاز *RANDOMIZED-SELECT* روی یک آرایه ورودی $A[p \,..\, r]$ با n عنصر، یک متغیر تصادفی است که آنرا با $T(n)$ نشان می‌دهیم و یک حد بالا روی $E[T(n)]$ را بصورت زیر تعیین می‌کنیم. روال *RANDOMIZED-PARTITION* برای برگرداندن یک عنصر به عنوان عنصر محوری دارای احتمال یکسانی است. بنابراین برای هر k که $1 \le k \le n$ باشد، زیر آرایه $A[p \,..\, q]$، k عنصر (همگی کوچکتر یا مساوی عنصر محوری) با احتمال $1/n$ دارد. برای $k = 1, 2, \ldots, n$ متغیرهای تصادفی شاخص X_k را تعریف می‌کنیم که

$X_k = I\{$زیرآرایه $A[p \,..\, q]$ دقیقاً k عنصر دارد$\}$

و بنابراین داریم

$$E[X_k] = 1/n \tag{9.1}$$

هنگامیکه *RANDOMIZED-SELECT* را فراخوانی می‌کنیم و $A[q]$ را به عنوان عنصر محوری انتخاب می‌کنیم، از قبل نمی‌دانیم که آیا بلافاصله به جواب درست می‌رسیم، به زیرآرایه $A[p \,..\, q-1]$ بازگشت می‌کنیم، یا به زیرآرایه $A[q+1 \,..\, r]$ بازگشت می‌کنیم. این تصمیم به مکانی که i امین عنصر کوچک نسبت به $A[q]$ قرار می‌گیرد بستگی دارد. با این فرض که $T(n)$ بصورت یکنواخت افزایش می‌یابد، می‌توانیم فراخوانی بازگشتی را توسط زمان مورد نیاز برای فراخوانی بازگشتی روی بزرگترین ورودی ممکن، محدود کنیم. به عبارت دیگر برای بدست آوردن یک حد بالا فرض می‌کنیم i امین عنصر همیشه در طرفی از تقسیم‌بندی قرار گرفته که تعداد عناصر بیشتری قرار دارند. برای یک فراخوانی داده شده از *RANDOMIZED-SELECT*، متغیر تصادفی شاخص X_k برای دقیقاً یک مقدار k، 1 می‌باشد و برای بقیه kها، 0 می‌باشد. وقتی $X_k = 1$ است، دو زیرآرایه که به آنها ممکن است بازگشت کنیم، دارای اندازه‌های $k-1$ و $n-k$ هستند. از اینرو، رابطه بازگشتی زیر را داریم

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^{n} (X_k \cdot T(\max(k-1, n-k)) + O(n)) . \end{aligned}$$

با بکاربردن مقادیر مورد انتظار داریم[1]

$$\begin{aligned} &\mathrm{E}[T(n)] \\ &\le \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\ &= \sum_{k=1}^{n} \mathrm{E}[X_k \cdot T(\max(k-1, n-k))] + O(n) \end{aligned}$$ (بنا به خطی بودن انتظار(امیدریاضی))

---

۱ ـ هنگامیکه متغیرهای تصادفی $X_1, X_2, \ldots, X_n$ متقابلاً مستقل هستند، داریم

$E[X_1, X_2 \ldots, X_n] = E[X_1]\, E[X_2] \ldots E[X_n]$

$$= \sum_{k=1}^{n} \mathrm{E}[X_k] \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n)$$

$$= \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) \qquad \text{(بنا به معادله (۹.۱))}.$$

به منظور به کار بردن معادله ذکر شده در پاورقی به این مطلب که X_k و $T(max(k-1, n-k))$ متغیرهای تصادفی مستقل می‌باشند متکی می‌شویم. تمرین ۲-۹.۲ از شما می‌خواهد که این ادعا را ثابت کنید.

عبارت $max(k-1, n-k)$ را در نظر می‌گیریم، داریم

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil . \\ n-k & \text{if } k \le \lceil n/2 \rceil . \end{cases}$$

اگر n زوج باشد، هر جمله از $T(\lceil n/2 \rceil)$ تا $T(n-1)$ دقیقاً دوبار در مجموع ظاهر می‌شود و اگر n فرد باشد، تمام این جملات دوبار و $T(\lfloor n/2 \rfloor)$ یکبار ظاهر می‌شود. بنابراین داریم

$$\mathrm{E}[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}[T(k)] + O(n) .$$

رابطه بازگشتی را با جایگذاری حل می‌کنیم. فرض کنید برای ثابت c که در شرایط اولیه رابطه بازگشتی صدق می‌کند، $T(n) \le cn$ است. فرض کنید برای n کوچکتر از یک ثابت، $T(n) = O(1)$ است؛ این ثابت را بعداً انتخاب خواهیم کرد. همچنین ثابت a را چنان انتخاب می‌کنیم که تابعی که با جمله $O(n)$ در بالا بیان شده است (و اجزاء غیر بازگشتی زمان اجرای الگوریتم را توضیح می‌دهد)، برای همه $n>0$ از بالا با an محدود شود. با استفاده از این فرض استقرایی داریم

$$\begin{aligned}
\mathrm{E}[T(n)] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\
&= \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + an \\
&\le \frac{2c}{n} \left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + an \\
&= \frac{2c}{n} \left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an
\end{aligned}$$

$$
\begin{aligned}
&= \frac{c}{n}\left(\frac{3n^2}{4}+\frac{n}{2}-2\right)+an \\
&= c\left(\frac{3n}{4}+\frac{1}{2}-\frac{2}{n}\right)+an \\
&\le \frac{3cn}{4}+\frac{c}{2}+an \\
&= cn-\left(\frac{cn}{4}-\frac{c}{2}-an\right) .
\end{aligned}
$$

به منظور کامل کردن اثبات، لازم است نشان دهیم که برای n به اندازه کافی بزرگ، این عبارت آخر حداکثر cn است، یا بطور معادل اینکه $cn/4 - c/2 - an \ge 0$. اگر $c/2$ را با دو طرف جمع کنیم و از n فاکتور بگیریم، $n(c/4 - a) \ge c/2$ بدست می‌آید. تا زمانیکه ثابت c را طوری انتخاب کنیم که $c/4 - a > 0$، یعنی $c > 4a$، می‌توانیم هر دو طرف را بر $c/4 - a$ تقسیم کنیم، داریم

$$
n \ge \frac{c/2}{c/4-a} = \frac{2c}{c-4a} .
$$

بنابراین، اگر فرض کنیم که برای $n < 2c/(c - 4a)$، $T(n) = O(1)$ است، داریم $T(n)=O(n)$. نتیجه می‌گیریم که هر شاخص آمار ترتیبی و بویژه میانه، بطور میانگین می‌تواند در زمانی خطی تعیین شود.

تمرین‌ها

۹.۲ -۱ نشان دهید در *RANDOMIZED-SELECT*، هیچ فراخوانی بازگشتی حتی به آرایه‌ای با طول 0 انجام نمی‌شود.

۹.۲ -۲ ثابت کنید متغیر تصادفی شاخص X_k و مقدار $T(max(k-1, n-k))$ مستقلند.

۹.۲ -۳ یک نسخه تکراری *RANDOMIZED-SELECT* را بنویسید.

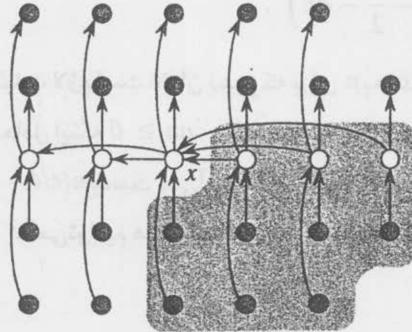
۹.۲ -۴ فرض کنید از *RANDOMIZED-SELECT* برای انتخاب عنصر مینیمم آرایه $A<3, 2, 9, 0, 7, 5, 4, 8, 6, 1>$ استفاده می‌کنیم. یک توالی از تقسیم‌بندی‌هایی که در بدترین حالت اجرای *RANDOMIZED-SELECT* حاصل می‌شوند را بیان کنید.

۹.۳ انتخاب در بدترین حالت زمان خطی

اکنون الگوریتم انتخابی را بررسی می‌کنیم که زمان اجرای آن در بدترین حالت $O(n)$ است. مانند *RANDOMIZED-SELECT*، الگوریتم *SELECT* عنصر مورد نظر را با تقسیم‌بندی بازگشتی آرایه ورودی پیدا می‌کند. اما ایده‌ای که پشت این الگوریتم وجود دارد، اینست که یک قسمت خوب را در هنگامیکه آرایه تقسیم می‌شود تضمین کند. *SELECT* از الگوریتم تقسیم‌بندی قطعی *PARTITION*

مربوط به مرتب سازی سریع (بخش ۷.۱ را ملاحظه کنید) استفاده می‌کند که طوری تغییر یافته است که عنصری که تقسیم‌بندی حول آن انجام می‌شود را به عنوان پارامتر ورودی بگیرد.

الگوریتم $SELECT$، iامین عنصر کوچک از آرایه ورودی با $n > 1$ عنصر را با اجرای مراحل زیر تعیین می‌کند. (اگر $n = 1$ باشد آنگاه $SELECT$ بطور مطلق، تنها مقدار ورودی‌اش را به عنوان iامین عنصر کوچک بر می‌گرداند.)



شکل ۹.۱ تحلیل الگوریتم $SELECT$. n عنصر با دایره‌های کوچک نشان داده می‌شوند و هر گروه یک ستون را اشغال می‌کند. میانه‌های گروه‌ها سفید شده‌اند و میانهٔ میانه‌ها، با x برچسب گذاری شده است. (هنگام پیدا کردن میانهٔ تعداد زوجی از عناصر، از میانه پایین استفاده می‌کنیم.) پیکان‌ها از عناصر بزرگتر به عناصر کوچکتر کشیده شده‌اند که توسط آن‌ها می‌توان دید که 3 عنصر از هر گروه 5 عنصری کامل در سمت راست x از x بزرگترند و 3 عنصر از هر گروه 5 عنصری در سمت چپ x کوچکتر از x هستند. عناصر بزرگتر از x روی زمینهٔ سایه خورده نشان داده شده‌اند.

۱. n عنصر آرایه ورودی را به $\lfloor n/5 \rfloor$ گروه 5 عنصری تقسیم کنید و حداکثر یک گروه از $n \bmod 5$ عنصر باقیمانده ساخته می‌شود.

۲. میانه هر یک از $\lceil n/5 \rceil$ گروه را ابتدا با مرتب سازی درجی عناصر هر گروه (که حداکثر 5 عنصر در هر یک وجود دارد) و سپس انتخاب میانه از لیست مرتب شده عناصر گروه پیدا کنید.

۳. از $SELECT$ بصورت بازگشتی برای پیدا کردن میانه x از $\lceil n/5 \rceil$ میانه‌ای که در مرحله ۲ پیدا شدند استفاده کنید. (اگر تعداد میانه‌ها زوج باشد، طبق قراردادمان، x میانه پایین است.)

۴. آرایه ورودی را حول میانهٔ میانه‌ها (یعنی x) با استفاده از نسخه تغییر یافته $PARTITION$ تقسیم کنید. فرض کنید k یک واحد بیشتر از تعداد عناصر در طرف کمتر تقسیم‌بندی باشد، بنابراین x، k امین عنصر کوچک است و $n-k$ عنصر در طرف بیشتر تقسیم‌بندی موجود است.

۵. اگر $i = k$ باشد، آنگاه x را برگردانید. در غیر اینصورت اگر $i < k$ باشد، از $SELECT$ بصورت بازگشتی برای پیدا کردن iامین عنصر کوچک در طرف کمتر استفاده کرده یا اگر $i > k$ باشد، از آن برای پیدا کردن $(i-k)$ امین عنصر کوچک در طرف بیشتر استفاده کنید.

برای تحلیل زمان اجرای $SELECT$، ابتدا یک حد پایین روی تعداد عناصر بزرگتر از عنصر تقسیم‌کننده x تعیین می‌کنیم. شکل ۹.۱ در تجسم این نحوه محاسبه به ما کمک می‌کند. حداقل نصفی از میانه‌های پیدا شده در مرحله ۲ بزرگتر از x[1] یعنی میانهٔ میانه‌ها هستند. بنابراین در حداقل نصف $\lceil n/5 \rceil$ گروه، 3 عنصر وجود دارند که از x بزرگترند، بجز برای گروهی که اگر 5 به n قابل قسمت نباشد، کمتر از 5 عنصر دارد و گروهی که خود شامل x است. با منظور نکردن این دو گروه ثابت می‌شود که تعداد عناصر بزرگتر از x حداقل برابر است با

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6 .$$

بطور مشابه، تعداد عناصری که کوچکتر از x هستند حداقل $3n/10-6$ است. بنابراین در بدترین حالت، $SELECT$ برای حداکثر $7n/10+6$ عنصر در مرحله ۵ بطور بازگشتی فراخوانی می‌شود.

اکنون می‌توانیم یک رابطه بازگشتی برای زمان اجرای الگوریتم $SELECT$ در بدترین حالت، یعنی $T(n)$، ایجاد کنیم. مراحل ۱، ۲ و ۴ زمان $O(n)$ را صرف می‌کنند. (مرحله ۲ شامل $O(n)$ فراخوانی مرتب‌سازی درجی روی مجموعه‌های با اندازه $O(1)$ است.) با فرض اینکه T بطور یکنواخت افزایش می‌یابد، مرحله ۳ زمان $T(\lceil n/5 \rceil)$ و مرحله ۵ حداکثر زمان $T(7n/10+6)$ را صرف می‌کند. فرض می‌کنیم که هر ورودی با 140 یا تعداد کمتری عنصر، به زمان $O(1)$ نیاز دارد که این فرض در ابتدا جالب به نظر نمی‌رسد؛ منشأ ثابت جادویی 140 بزودی مشخص خواهد شد. بنابراین می‌توانیم رابطه بازگشتی زیر را بدست آوریم

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 140 , \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n > 140 . \end{cases}$$

نشان می‌دهیم که زمان اجرا با استفاده از جایگذاری، خطی است. بطور دقیقتر، نشان خواهیم داد که برای ثابت به اندازه کافی بزرگ c و همه $n > 0$، $T(n) \leq cn$ است. با فرض $T(n) \leq cn$ برای ثابتهای به اندازه کافی بزرگ c و تمام $140 \leq n$ شروع می‌کنیم؛ اگر c به اندازه کافی بزرگ باشد این فرض برقرار است. همچنین یک ثابت a انتخاب می‌کنیم بطوریکه تابع بیان شده با جمله $O(n)$ در بالا (که اجزاء غیر بازگشتی زمان اجرای الگوریتم را بیان می‌کند) برای همه $n > 0$، از بالا توسط an محدود شود. با جایگذاری این فرض استقرایی در سمت راست رابطه بازگشتی خواهیم داشت

۱ـ از آنجا که فرض کرده‌ایم اعداد مجزا هستند، می‌توانیم بدون نگرانی در مورد تساوی بگوییم "بزرگتر از" و "کوچکتر از".

$$\begin{aligned} T(n) &\le c\lceil n/5\rceil + c(7n/10+6) + an \\ &\le cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an)\,, \end{aligned}$$

که حداکثر برابر cn است اگر

$$-cn/10 + 7c + an \le 0\,. \qquad (۹.۲)$$

وقتی $n > 70$ باشد، نامساوی (۹.۲) معادل با نامساوی $c \ge 10a(n/(n-70))$ است. بدلیل اینکه فرض می‌کنیم $n\ge140$ است، داریم $n/(n-70) \le 2$، و بنابراین انتخاب $c\ge20a$ نامساوی (۹.۲) را برقرار می‌کند. (توجه کنید که مطلب خاصی در مورد ثابت 140 وجود ندارد؛ می‌توانستیم آنرا با هر ثابت صحیح اکیداً بزرگتر از 70 جایگزین کرده و سپس c را بر طبق آن انتخاب کنیم.) بنابراین زمان اجرای *SELECT* در بدترین حالت، خطی است.

همانند مرتب سازی مقایسه‌ای (بخش ۸.۱ را ملاحظه کنید)، *SELECT* و *RANDOMIZED-SELECT* تنها با مقایسه عناصر، اطلاعاتی در مورد ترتیب نسبی عناصر را مشخص می‌کنند. از فصل ۸ بیاد آورید که مرتب سازی در مدل مقایسه ای حتی بطور میانگین نیاز به زمان $\Omega(nlgn)$ دارد (مسئله ۱–۸ را ملاحظه کنید). الگوریتم‌های مرتب سازی با زمان خطی در فصل ۸، فرضی را در مورد ورودی در نظر می‌گیرند. در مقابل، الگوریتم‌های انتخاب با زمان خطی در این فصل، به هیچ فرضی در مورد ورودی نیاز ندارند. آنها تابع حد پایین $\Omega(nlgn)$ نیستند، زیرا حل مسئله انتخاب را بدون مرتب سازی مدیریت می‌کنند.

بنابراین زمان اجرا خطی است، زیرا این الگوریتم‌ها مرتب سازی را انجام نمی‌دهند؛ رفتار زمان خطی، یکی از نتایج فرضیات در مورد ورودی نیست، در حالیکه حالتی برای الگوریتمهای مرتب سازی در فصل ۸ بود. مرتب سازی در مدل مقایسه‌ای حتی بطور میانگین به زمان $\Omega(nlgn)$ احتیاج دارد. (مسئله ۱–۸ را ملاحظه کنید)، و بنابراین روش مرتب سازی و اندیس گذاری که در مقدمه این بخش بیان شد، بطور مجانبی غیر کارآمد است.

تمرین‌ها

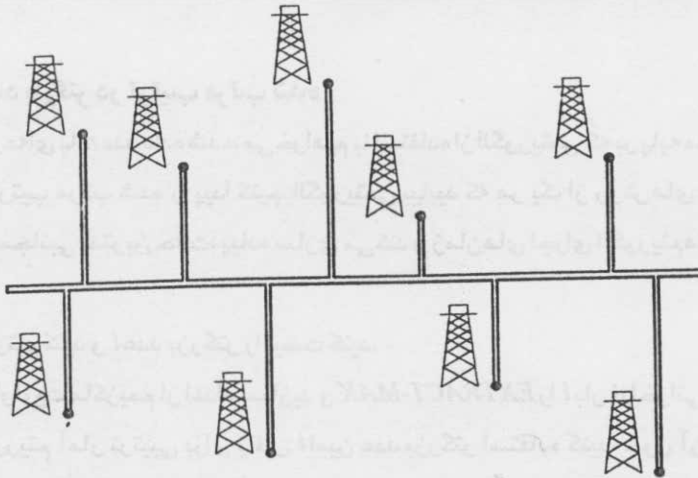
۱– ۹.۳ در الگوریتم *SELECT*، عناصر ورودی به گروه‌های 5 عنصری تقسیم می‌شوند. آیا اگر عناصر ورودی به گروه‌های 7 عنصری تقسیم شوند، الگوریتم در زمانی خطی کار خواهد کرد؟ ثابت کنید که اگر از گروه‌های 3 عنصری استفاده شود، *SELECT* در زمان خطی اجرا نمی‌شود.

۲– ۹.۳ *SELECT* را برای نشان دادن اینکه، اگر $n \ge 140$ باشد آنگاه حداقل $\lceil n/4\rceil$ عنصر بزرگتر از

میانهٔ میانه‌ها (یعنی x) و حداقل $\lceil n/4 \rceil$ عنصر کوچکتر از x هستند، تحلیل کنید.

۹.۳-۳ نشان دهید چگونه مرتب سازی سریع می‌تواند در بدترین حالت در زمان $O(nlgn)$ اجرا شود.

۹.۳-۴* فرض کنید که یک الگوریتم فقط از مقایسه برای پیدا کردن i امین عنصر کوچک در یک مجموعه n عضوی استفاده می‌کند. نشان دهید این الگوریتم همچنین می‌تواند $i-1$ عضو کوچکتر و $n-i$ عضو بزرگتر را بدون انجام هیچ مقایسه اضافی پیدا کند.



شکل ۹.۲ پروفسور *Olay* می‌خواهد موقعیت خط لوله نفت شرق - غرب را طوری تعیین کند که طول کل خطوط فرعی شمال - جنوب مینیمم شود.

۹.۳-۵ فرض کنید یک زیرروال میانه "جعبه سیاه" با زمان خطی در بدترین حالت دارید. یک الگوریتم ساده با زمان خطی ارائه کنید که مسئله انتخاب برای شاخص آمار ترتیبی دلخواه را حل کند.

۹.۳-۶ k امین *Quantiles*[1] یک مجموعه n عضوی، $k-1$ شاخص آمار ترتیبی هستند که مجموعه مرتب شده را به k مجموعه با اندازه‌های برابر تقسیم می‌کنند. الگوریتمی با زمان $O(nlgk)$ ارائه دهید که k امین *Quantiles* یک مجموعه را لیست کند.

۹.۳-۷ الگوریتمی با زمان $O(n)$ شرح دهید که، با دریافت مجموعه S با n عدد متفاوت و یک عدد صحیح مثبت $k \leq n$، k عدد در S که به میانه نزدیکترین هستند را مشخص کند.

۹.۳-۸ فرض کنید $X[1 \,..\, n]$ و $Y[1 \,..\, n]$ دو آرایه هستند، که هر یک شامل n عدد مرتب شده است. الگوریتمی با زمان $O(lgn)$ ارائه دهید تا میانه تمام $2n$ عنصر در آرایه X و Y را پیدا کند.

۹.۳-۹ پروفسور *Olay* مشاور یک شرکت نفت است، که برای یک خط لوله بزرگ از غرب تا شرق شامل n چاه برنامه ریزی می‌کند. همانطور که در شکل ۹.۲ نشان داده شده است، از هر چاه نفت باید یک خط

۱ ـ یک معیار آماری برای سنجش توزیع احتمال .

لوله فرعی مستقیماً به خط لوله اصلی از طریق کوتاهترین مسیر (که شمال یا جنوب است) وصل شود. مختصات x و y چاه‌های نفت داده شده است، پروفسور چگونه باید مکان بهینه خط لوله اصلی را تعیین کند (مکانی که کل طول خطوط فرعی را مینیمم می‌کند)؟ نشان دهید که مکان بهینه می‌تواند در زمانی خطی تعیین شود.

مسائل

۹-۱ i عدد بزرگتر در ترتیب مرتب شده

در مجموعه‌ای با n عدد داده شده، می‌خواهیم با استفاده از الگوریتمی که بر پایه مقایسه است، i عدد بزرگتر آن ترتیب مرتب شده را پیدا کنیم. الگوریتمی بیابید که هر یک از روش‌های زیر را با بهترین زمان اجرای مجانبی بدترین حالت، پیاده سازی می‌کند و زمان‌های اجرای الگوریتم‌ها را بر حسب n و i تحلیل کنید.

a. اعداد را مرتب کنید و i عدد بزرگتر را لیست کنید.

b. یک صف اولویت ماکزیمم از اعداد بسازید و $EXTRACT\text{-}MAX$ را i بار فراخوانی کنید.

c. از یک الگوریتم آمار ترتیبی برای یافتن i امین عدد بزرگتر استفاده کنید، حول آن عدد تقسیم‌بندی کرده و i عدد بزرگتر را مرتب کنید.

۹-۲ میانه وزن دار

برای n عنصر مختلف $x_1 , x_2 , . . . , x_n$ با وزنهای مثبت $w_1 , w_2 , . . . , w_n$ بطوریکه $\sum^n_{i=1} w_i=1$، میانه (پایین) وزن دار، عنصر x_k است که در

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

و

$$\sum_{x_i > x_k} w_i \leq \frac{1}{2}.$$

صدق می‌کند.

a. ثابت کنید که میانه $x_1 , x_2 , . . . , x_n$، برای میانه وزن دار x_i با وزن‌های $w_i=1/n$ برای $i = 1, 2, . . . , n$ می‌باشد.

b. نشان دهید که چگونه با استفاده از مرتب‌سازی، میانه وزن دار n عنصر در بدترین حالت زمانی در $O(nlgn)$ محاسبه شود.

c. نشان دهید چگونه با استفاده از یک الگوریتم میانه با زمانی خطی مانند $SELECT$ در بخش ۹.۳، میانه وزن دار را در بدترین حالت زمانی در زمان $\Theta(n)$ محاسبه کنیم.

مسئله مکان اداره پست به شکل زیر تعریف می‌شود. n نقطه $p_1, p_2, \ldots, p_n$ با وزن‌های مربوطه $w_1, w_2, \ldots, w_n$ داده می‌شوند. می‌خواهیم نقطه P (لازم نیست یکی از نقاط ورودی باشد) را پیدا کنیم که مجموع $\sum^{n}_{i=1} w_i\, d(p, p_i)$ که در آن $d(a,b)$ فاصله بین نقاط a و b است را مینیمم می‌کند.

d. ثابت کنید که میانه وزن دار، بهترین جواب برای مسئله مکان اداره پست یک بعدی است، که در آن نقاط، اعداد حقیقی ساده هستند و فاصله بین نقاط a و b، $d(a,b) = |a-b|$ می‌باشد.

e. بهترین جواب برای مسئله مکان اداره پست دو بعدی را بیابید، که در آن نقاط، جفتهای مختصاتی (x,y) هستند و فاصله بین نقاط $a = (x_1, y_1)$ و $b = (x_2, y_2)$ برابر **فاصله** *Manhattan*[1] است که از $d(a,b) = |x_1-x_2| + |y_1-y_2|$ بدست می‌آید.

۹-۳ شاخص آمار ترتیبی کوچک

نشان داده شد که تعداد $T(n)$ مقایسه توسط $SELECT$ برای انتخاب i امین شاخص آمار ترتیبی از n عدد در بدترین حالت در $T(n) = \Theta(n)$ صدق می‌کند، ولی ثابتی که در نمادگذاری Θ پنهان می‌شود نسبتاً بزرگ است. وقتی i نسبت به n کوچک است، می‌توانیم یک روال متفاوت را پیاده سازی کنیم که از $SELECT$ به عنوان یک زیرروال استفاده می‌کند، اما در بدترین حالت مقایسه‌های کمتری انجام می‌دهد.

a. الگوریتمی را شرح دهید که $U_i(n)$ مقایسه برای یافتن i امین عنصر کوچک n عنصر استفاده می‌کند، که

$$U_i(n) = \begin{cases} T(n) & \text{if } i \geq n/2\,, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{otherwise}\,. \end{cases}$$

(راهنمایی: با $\lfloor n/2 \rfloor$ مقایسه جفت به جفت مجزا شروع کنید، و روی مجموعه‌ای که شامل عنصر کوچکتر از هر جفت است بازگشت کنید.)

b. نشان دهید اگر $i < n/2$، آنگاه $U_i(n) = n + O(T(2i)\, lg(n/i))$.

c. نشان دهید اگر i یک ثابت کوچکتر از $n/2$ باشد، آنگاه $U_i(n) = n + O(lgn)$.

d. نشان دهید اگر برای $k \geq 2$ داشته باشیم $i = n/k$، آنگاه $U_i(n) = n + O(T(2n/k)\, lgk)$.

---

1. Manhattan distance

III

ساختمان داده‌ها

مقدمه

مجموعه‌ها همانطور که پایه ریاضیات هستند پایه و اساس علم کامپیوتر نیز می‌باشند. در حالیکه مجموعه‌های ریاضی تغییر نمی‌کنند، مجموعه‌هایی که توسط الگوریتم‌ها دستکاری می‌شوند می‌توانند رشد کنند، کاهش یابند، یا در طول زمان تغییر کنند. چنین مجموعه‌هایی را مجموعه‌های *پویا*[1] می‌نامیم. پنج فصل بعدی برخی تکنیک‌ها برای نمایش مجموعه‌های پویای متناهی و دستکاری آنها در کامپیوتر را ارائه می‌کند.

الگوریتم‌ها ممکن است به چندین نوع مختلف از اعمال که بایستی روی مجموعه‌ها انجام شوند احتیاج داشته باشند. به عنوان مثال، بسیاری از الگوریتمها ممکن است صرفاً به قابلیت درج کردن عناصر، حذف آنها و بررسی عضویت در یک مجموعه نیاز داشته باشند. مجموعه پویایی که این اعمال را پشتیبانی می‌کند یک *لغت نامه*[2] نامیده می‌شود. الگوریتم‌های دیگر به اعمال پیچیده‌تری احتیاج دارند. به عنوان مثال، صف‌های اولویت مینیمم که در فصل ۶ در قسمت ساختمان داده *heap* معرفی شده‌اند، اعمال درج کردن عنصر و خارج کردن کوچکترین عنصر یک مجموعه را پشتیبانی می‌کنند. بهترین راه برای پیاده سازی یک مجموعه پویا، به اعمالی که باید پشتیبانی شوند بستگی دارد.

اعضاء یک مجموعه پویا

در یک پیاده سازی معمول مجموعه پویا، هر عضو توسط یک شیء نمایش داده می‌شود که اگر اشاره‌گری به شیء داشته باشیم، فیلدهای آن می‌توانند بررسی و دستکاری شوند. (بخش ۱۰.۳ پیاده‌سازی اشیاء اشاره‌گر را در محیط‌های برنامه سازی مورد بحث قرار می‌دهند که اشیاء و اشاره‌گرها را به عنوان نوع دوره اصلی شامل نمی‌شوند). برخی از انواع مجموعه‌های پویا فرض می‌کنند که یکی از فیلدهای شیء *کلیدی*[3] شاخص است. اگر همه کلیدها متفاوتند، می‌توانیم مجموعه پویا را به عنوان مجموعه‌ای از مقادیر کلیدی در نظر بگیریم. شیء ممکن است دارای *داده‌های فرعی*[4] باشد، که در فیلدهای دیگر شیء قرار می‌گیرند، اما در پیاده سازی مجموعه استفاده نمی‌شوند. همچنین مجموعه، ممکن است فیلدهایی داشته باشد که توسط اعمال مجموعه دستکاری می‌شوند: این

---

1. *dynamic*
2. *Dictionary*
3. *key*
4. *satellite data*

فیلدها ممکن است شامل داده یا اشاره‌گر به اشیاء دیگر مجموعه باشند.

برخی مجموعه‌های پویا از پیش فرض می‌کنند که کلیدها در یک مجموعه مرتب مانند اعداد حقیقی، یا مجموعه تمام کلماتی که بر طبق ترتیب الفبایی معمول مرتب شده‌اند، انتخاب می‌گردند. (یک مجموعه مرتب در ویژگی سه قسمتی که در صفحه ٦٢ بیان شده صدق می‌کند). ترتیب کلی به ما اجازه این را می‌دهد که به عنوان مثال کوچکترین عضو مجموعه را تعریف کنیم، یا در مورد عضو بعدی که از یک عضو داده شده در مجموعه بزرگتر است صحبت کنیم.

اعمال بر روی مجموعه‌های پویا

اعمال روی مجموعه‌های پویا می‌توانند به دو گروه تقسیم شوند: ***پرس و جوها***[1]، که به سادگی اطلاعاتی در مورد مجموعه را بر می‌گرداند و ***اعمال تغییر دهنده***[2]، که مجموعه را تغییر می‌دهند. در این جا لیستی از اعمال معمول را مشاهده می‌کنیم. هر کاربرد خاص، معمولاً به پیاده سازی تعداد اندکی از این اعمال احتیاج دارد.

SEARCH(S,k)

یک پرس و جو که، با گرفتن مجموعه S و مقدار کلیدی k، اشاره گر x به یک عضو در S را بر می‌گرداند، بطوریکه $key[x] = k$، یا اگر چنین عضوی به S تعلق نداشته باشد، NIL را بر می‌گرداند.

INSERT(S,x)

یک عمل تغییردهنده که مجموعه S را با عضوی که x به آن اشاره می‌کند افزایش می‌دهد. معمولاً فرض می‌کنیم هر فیلد در عضو x که در پیاده‌سازی مجموعه مورد نیاز است از قبل مقدار دهی اولیه شده است.

DELETE(S,x)

یک عمل تغییردهنده که با دریافت اشاره‌گر x به عضوی از S، x را از S حذف می‌کند. (توجه داشته باشید که این عمل از اشاره گری به عضو x استفاده می‌کند نه از یک مقدار کلید.)

MINIMUN(S)

یک پرس و جو روی مجموعه کاملاً مرتب S که یک اشاره گر به عضوی از S که کمترین کلید را دارد برمی‌گرداند.

---

1. queries

2. modifying operations

MAXIMUM(S)

یک پرس و جو روی مجموعه کاملاً مرتب S که یک اشاره گر به عضوی از S با بزرگترین کلید را بر می‌گرداند.

SUCCESSOR(S,x)

یک پرس و جو که با دریافت عضو x که کلید آن در یک مجموعه کاملاً مرتب S قرار دارد، اشاره گری به عضو بزرگتر بعدی در S را بر می‌گرداند، یا اگر x عضو ماکزیمم است *NIL* بر می‌گرداند.

PREDECESSOR(S,x)

یک پرس و جو که با دریافت عضو x که کلید آن از مجموعه کاملاً مرتب S است، اشاره گری به عضو کوچکتر بعدی در S را بر می‌گرداند، یا اگر x عضو مینیمم است *NIL* بر می‌گرداند.

پرس و جوهای *SUCCESSOR* و *PREDECESSOR* اغلب برای مجموعه‌هایی با کلیدهای غیر متفاوت توسعه می‌یابند. برای مجموعه‌ای با n کلید، استنباط معمول اینست که یک فراخوانی *MINIMUM* که به دنبال آن $n-1$ فراخوانی *SUCCESSOR* صورت می‌گیرد، اعضاء مجموعه را در ترتیبی مرتب شده می‌شمارد.

زمانی که جهت اجرای یک عمل مجموعه بکار می‌رود، معمولاً بر حسب اندازه مجموعه‌ای که به عنوان آرگومان داده شده است اندازه‌گیری می‌شود. به عنوان مثال، بخش ۱۳ ساختمان داده‌ای را معرفی می‌کند که می‌تواند برای یک مجموعه با اندازه n هر یک از اعمالی که در بالا لیست شده‌اند را در زمان $O(lgn)$ پشتیبانی می‌کند.

مروری بر قسمت III

فصل‌های ۱۴-۱۰ چندین ساختمان داده که می‌توانند در پیاده سازی مجموعه‌های پویا استفاده شوند را شرح می‌دهند؛ بسیاری از آنها بعدها برای ساخت الگوریتم‌های کارا برای مسائل مختلفی استفاده خواهند شد. ساختمان داده مهم دیگر - *heap* - نیز قبلاً در فصل ۶ معرفی شده است.

فصل ۱۰ ضرورت کار با ساختمان داده‌های ساده‌ای مانند پشته‌ها، صف‌ها، لیست‌های پیوندی و درخت‌های مشتق شده را بیان می‌کند. این فصل همچنین نشان می‌دهد که اشیاء و اشاره گرها چگونه می‌توانند در محیط‌های برنامه نویسی که بصورت اولیه آنها را پشتیبانی نمی‌کنند پیاده سازی شوند. بسیاری از این موارد باید برای شخصی که واحد درسی برنامه نویسی مقدماتی را گذرانده است، آشنا باشد.

فصل ۱۱ جدول‌های در هم سازی را معرفی می‌کند. که اعمال لغت نامه‌ای *INSERT*، *DELETE* و

SEARCH را پشتیبانی می‌کنند. در بدترین حالت، در هم سازی به زمان $\Theta(n)$ برای انجام یک عمل *SEARCH* احتیاج دارد، ولی زمان مورد انتظار برای اعمال جدول درهم سازی برابر $O(1)$ است. تحلیل درهم سازی متکی بر احتمال است، ولی اکثر فصل‌ها به هیچ پیش زمینه‌ای در مورد احتمال احتیاج ندارند.

درختهای جستجوی دودویی، که در فصل ۱۲ پوشش داده شده‌اند، تمام اعمال مجموعه پویا که در بالا لیست شده‌اند را پشتیبانی می‌کنند. در بدترین حالت، هر عمل روی درختی با n عضو، زمان $\Theta(n)$ را صرف می‌کند ولی برای درخت جستجوی دودویی تصادفی ساخته شده ،زمان مورد انتظار برای هر عمل $O(lgn)$ است. درختهای جستجوی دودویی به عنوان پایه و اساس بسیاری از ساختمان داده‌های دیگر بکار برده می‌شوند.

درختهای قرمز - سیاه، که یک نوع متفاوت از درختهای جستجوی دودویی هستند، در فصل ۱۳ معرفی شده‌اند. بر خلاف درختهای جستجوی دودویی معمولی، تضمین شده است که درختهای قرمز-سیاه بسیار خوب عمل می‌کنند: اعمال در بدترین حالت زمان $O(lgn)$ را صرف می‌کنند. درخت قرمز - سیاه یک درخت جستجوی متوازن است؛ فصل ۱۸ نوع دیگری از درختهای جستجوی متوازن را ارائه می‌کند، که *B-Tree* نام دارد. اگر چه مکانیزم درختهای قرمز - سیاه تا حدی پیچیده است، در این فصل می‌توانید اندک اندک اکثر ویژگی‌های آن را بدون مطالعه جزئیات مکانیزم آن بدست آورید. با این وجود، حرکت در راستای کدها می‌تواند کاملاً آموزنده باشد.

در فصل ۱۴، نشان می‌دهیم که چگونه می‌توان درختهای قرمز - سیاه را توسعه داد تا علاوه بر اعمال لیست شده در فوق، اعمال دیگری را نیز پشتیبانی کنند. آنها را توسعه می‌دهیم که بتوان بطور پویا، شاخص‌های آماری را برای هر مجموعه از کلیدها نگهداری کرد. سپس آنها را برای نگهداری بازه‌های اعداد حقیقی، به شکل متفاوتی توسعه می‌دهیم.

۱۰ ساختمان داده‌های مقدّماتی

در این فصل نمایش مجموعه پویا با استفاده از ساختمان داده ساده‌ای که از اشاره‌گرها استفاده می‌کند را بررسی می‌کنیم. اگر چه ساختمان داده‌های پیچیده بسیاری می‌توانند با استفاده از اشاره‌گرها کاربرد زیادی پیدا کنند، امّا ما صرفاً آنهایی که اصلی هستند را ارائه می‌کنیم: پشته‌ها، صف‌ها، لیست‌های پیوندی و درختهای مشتق شده. همچنین در مورد روشی بحث می‌کنیم که توسط آن اشیاء و اشاره‌گرها، می‌توانند از آرایه‌ها ترکیب شوند.

۱۰.۱ پشته‌ها و صف‌ها

پشته‌ها و صف‌ها مجموعه‌هایی پویا هستند که در آن‌ها عضوی از مجموعه که با عمل *DELETE* حذف می‌شود از قبل مشخص شده است. در یک *پشته*[1]، عضوی که از مجموعه حذف می‌شود همان عضوی است که اخیراً درج شده است: پشته خط مشی *آخرین ورودی - اولین خروجی*[2] یا *LIFO* را پیاده سازی می‌کند. بطور مشابه، در *صف*[3] عنصری که حذف می‌شود همیشه همان عنصری است که در مجموعه، برای بیشترین مدت حضور داشته است: صف خط مشی *اولین ورودی - اولین خروجی*[4] یا *FIFO* را پیاده سازی می‌کند. چندین راه مؤثر برای پیاده سازی پشته‌ها و صف‌ها روی یک کامپیوتر وجود دارد. در این بخش نشان می‌دهیم که چگونه می‌توان از یک آرایه ساده برای پیاده سازی هر یک استفاده کرد.

پشته‌ها

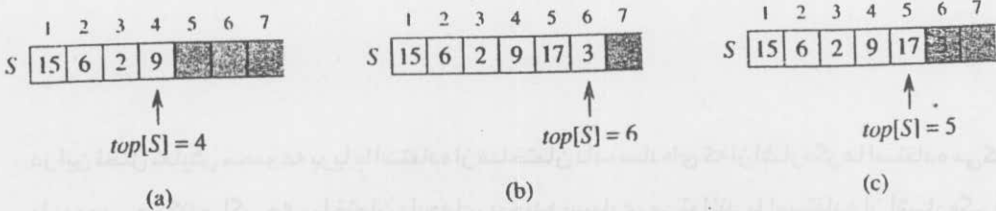
عمل *INSERT* روی یک پشته اغلب *PUSH* و عمل *DELETE* که هیچ آرگومانی نمی‌گیرد، اغلب *POP* نامیده می‌شود. این نامها تلمیحی هستند برای پشته‌های فیزیکی مانند پشته‌های فنردار بشقابها که در کافه تریاها استفاده می‌شوند. ترتیب برداشته شدن بشقابها از پشته، عکس ترتیب روی هم قرار دادن

---

1. *stack*
2. *last-in-first out*
3. *Queue*
4. *first in-first out*

آنها است زیرا تنها بشقابی که در بالا قرار دارد قابل دسترس است.

همانطور که در شکل ۱۰.۱ نشان داده شده است، می‌توانیم پشته‌ای با حداکثر n عنصر را با استفاده از آرایه $S[1..n]$ پیاده سازی کنیم. این آرایه دارای خصوصیت $top[S]$ است که آخرین عنصر درج شده را مشخص می‌کند. پشته از عناصر $S[1..top[S]]$ تشکیل می‌شود، که $S[1]$ عنصری است که در ته پشته و $S[top[S]]$ عنصری است که در بالای پشته قرار دارد.



شکل ۱۰.۱ پیاده سازی یک پشته با استفاده از چند آرایه. عناصر پشته فقط درموقعیت‌هایی که بصورت روشن سایه زده شده‌اند، ظاهر می‌شوند: *(a)* پشته *S*، *4* عنصر دارد. عنصر بالایی *9*است. *(b)* پشته *S* بعد از فراخوانی‌های *PUSH(S,17)* و *PUSH(S,3)* . *(c)* پشته *S* بعد از فراخوانی *pop(S)* که عنصر *3* را برگردانده است، که این عنصر همان عنصری است که آخرین بار به داخل پشته درج شده است. اگر چه عنصر *3* هنوز در آرایه دیده می‌شود اما دیگر در پشته قرار ندارد؛ *top*، عدد *17* است.

وقتی $top[S] = 0$ پشته هیچ عنصری ندارد و *خالی* [1] است. خالی بودن پشته می‌تواند توسط عمل پرس و جوی *STACK-EMPTY* بررسی شود. اگر از یک پشته خالی بخواهیم عنصری خارج کنیم، می‌گوییم پشته ته ریز [2] می‌شود، که معمولاً یک خطا است. اگر $top[S]$ بیشتر از n شود، پشته سرریز [3] می‌شود. (در پیاده‌سازی شبه‌کد، نگران سرریز پشته نمی‌باشیم.)

هر یک از اعمال پشته می‌توانند با چند خط کد پیاده‌سازی شوند.

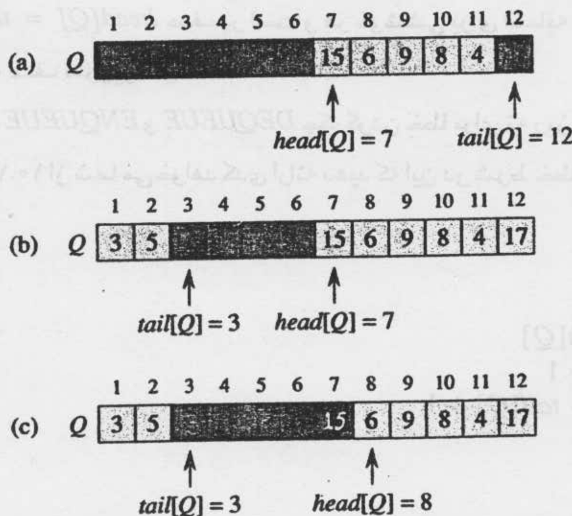
```
STACK-EMPTY(S)
1  if top[S] = 0
2      then return TRUE
3      else return FALSE

PUSH(S, x)
1  top[S] ← top[S] + 1
2  S[top[S]] ← x
```

1. empty
2. underflow
3. overflow

```
POP(S)
1  if STACK-EMPTY(S)
2     then error "underflow"
3     else top[S] ← top[S] − 1
4          return S[top[S] + 1]
```

شکل ۱۰.۱ تأثیر اعمال تغییر دهنده *PUSH* و *POP* روی پشته را نشان می‌دهد. هر یک از سه عمل پشته، زمان *O(1)* را صرف می‌کنند.



شکل ۱۰.۲ یک‌صف که بااستفاده از آرایه *Q[1 . . 12]* پیاده‌سازی شده‌است. عناصرصف صرفاًدر نواحی که سایه روشن خورده‌اند دیده می‌شوند. *(a)* صف، 5 عنصر در موقعیت‌های *Q[7. .11]* دارد. *(b)* شکل صف بعد از فراخوانی‌های *ENQUEUE(Q,17)*، *ENQUEUE(Q,3)* و *ENQUEUE(Q,5)* . *(c)* شکل صف پس از فراخوانی *DEQUEUE(Q)* که کلید *15* از ابتدای صف را بر می‌گرداند. *head* جدید *6* می‌باشد.

صف‌ها

عمل *INSERT* روی صف را *ENQUEUE* و عمل *DELETE* روی صف را *DEQUEUE* می‌نامیم؛ مانند عمل *POP* پشته، *DEQUEUE* هیچ آرگومانی نمی‌گیرد. ویژگی *FIFO* یک صف باعث می‌شود که مانند یک صف از آدم‌ها در اداره ثبت نام، عمل کند. صف یک *head* و یک *tail* دارد. وقتی یک عنصر وارد صف می‌شود مکانش در انتهای صف *(tail)* می‌باشد، دقیقاً همانطور که یک دانشجو تازه از راه رسیده در انتهای صف قرار می‌گیرد. عنصری که از صف خارج می‌شود همیشه عنصری است که در ابتدای *(head)* صف قرار دارد، درست مانند دانشجوی ابتدای صف، که بیشتر از همه منتظر مانده

است. (خوشبختانه، نباید نگران عناصری باشیم که از وسط صف بریده می‌شوند.)

شکل ۱۰.۲ روشی برای پیاده سازی یک صف با حداکثر *n-1* عضو با استفاده از آرایه *Q[1. .n]* را نشان می‌دهد. صف دارای خصوصیت *head[Q]* است که ابتدای آن را مشخص می‌کند یا به آن اشاره می‌کند. خصوصیت *tail[Q]*، موقعیت بعدی را مشخص می‌کند که عنصر تازه از راه رسیده، در آن موقعیت صف درج خواهد شد. عناصر صف در مکانهای *head[Q],head[Q]+1,...,tail[Q]-1*. قرار دارند، که اگر روی صف حرکت کنیم موقعیت *1* بلافاصله بعد از موقعیت *n* در ساختار حلقوی قرار می‌گیرد. وقتی *head[Q] = tail[Q]*، صف خالی است. در ابتدا داریم *head[Q] = tail[Q] = 1*. وقتی صف خالی است، هر کوششی برای خارج کردن یک عنصر از صف باعث ته ریز شدن می‌شود. وقتی *head[Q] = tail[Q]+1* صف پر است و هر کوششی برای اضافه نمودن یک عنصر به صف باعث می‌شود که صف سرریز شود.

در روال‌های *ENQUEUE* و *DEQUEUE* چک کردن خطا برای ته ریز و سرریز شدن حذف شده است. (تمرین ۴-۱۰.۱ از شما می‌خواهد کدی ارائه دهید که این دو شرط خطا را چک می‌کنند.)

```
ENQUEUE(Q, x)
1  Q[tail[Q]] ← x
2  if tail[Q] = length[Q]
3     then tail[Q] ← 1
4     else  tail[Q] ← tail[Q] + 1

DEQUEUE(Q)
1  x ← Q[head[Q]]
2  if head[Q] = length[Q]
3     then head[Q] ← 1
4     else  head[Q] ← head[Q] + 1
5  return x
```

شکل ۱۰.۲ تأثیرات اعمال *ENQUEUE* و *DEQUEUE* را نشان می‌دهد. هر عمل زمان *O(1)* را صرف می‌کند.

تمرین‌ها

۱-۱۰.۱ با استفاده از شکل ۱۰.۱ به عنوان الگو، نتیجه هر عمل از توالی زیر روی پشته *S* که در ابتدا خالی است و در آرایه *S[1. .6]* ذخیره شده را شرح دهید.

PUSH(S,4) , PUSH(S,1) , PUSH(S,3) , POP(S) , PUSH(S,8) , POP(S).

۲-۱۰.۱ توضیح دهید که چگونه دو پشته را روی یک آرایه *A[1..n]* پیاده سازی کنیم، بطوریکه هیچ

پشته‌ای سرریز نشود. مگر اینکه تعداد کل اعضاء در دو پشته روی هم n شود. اعمال $PUSH$ و POP باید در زمان $O(1)$ اجرا شوند.

۱۰.۱-۳ با استفاده از شکل ۱۰.۲ به عنوان الگو، نتیجه هر عمل از توالی زیر روی صف Q که در آرایه $Q[1..6]$ ذخیره شده و در ابتدا خالی است را شرح دهید.

ENQUEUE(Q,4), ENQUEUE(Q,1), ENQUEUE(Q,3), DEQUEUE(Q), ENQUEUE(Q,8) , DEQUEUE(Q).

۱۰.۱-۴ *ENQUEUE* و *DEQUEUE* را به نحوی بازنویسی کنید که ته ریز و سرریز شدن صف را کشف کند.

۱۰.۱-۵ در حالیکه یک پشته اعمال درج و حذف را فقط از یک طرف می‌تواند انجام دهد و یک صف می‌تواند درج را از یک طرف و حذف را در طرف دیگر انجام دهد، یک *deque* (صف دو طرفه) می‌تواند درج و حذف را از هر دو طرف انجام دهد. چهار روال با زمان $O(1)$ برای درج و حذف عنصر از هر دو طرف یک *deque* که از یک آرایه ساخته شده است، بنویسید.

۱۰.۱-۶ نشان دهید که چگونه یک صف را با استفاده از دو پشته پیاده‌سازی کنیم. زمان اجرای اعمال صف را تحلیل کنید.

۱۰.۱-۷ نشان دهید که چگونه یک پشته را با استفاده از دو صف پیاده سازی کنیم. زمان اجرای اعمال پشته را تحلیل کنید.

۱۰.۲ لیستهای پیوندی

لیست پیوندی، ساختمان داده‌ای است که اشیاء با یک ترتیب خطی در آن قرار گرفته‌اند. بر خلاف آرایه، که در آن ترتیب خطی توسط اندیسهای آرایه تعیین می‌شود، ترتیب در لیست پیوندی بوسیله یک اشاره‌گر در هر شیء تعیین می‌گردد. لیستهای پیوندی یک نمایش انعطاف‌پذیر و ساده برای مجموعه‌های پویا ایجاد می‌کنند، و تمام اعمالی که قبلاً لیست شد را (ولی نه لزوماً بصورت مؤثر) پشتیبانی می‌کنند.

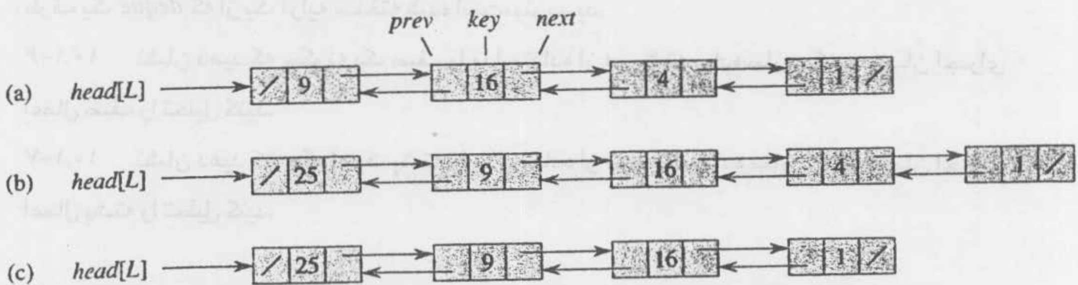
همانطور که در شکل ۱۰.۳ نشان داده شده است، هر عنصر *لیست پیوندی دو طرفه*[1]، بنام L شیئی است با یک فیلد کلید و دو فیلد اشاره گر دیگر: *next* و *prev* شیء ممکن است داده‌های فرعی دیگری را نیز در بر بگیرد. برای عنصر x در لیست پیوندی، $next[x]$ به عنصر ما بعد خود در لیست پیوندی و $prev[x]$ به عنصر ما قبل خود در لیست اشاره می‌کند. اگر $prev[x] = NIL$ عنصر x ما قبلی ندارد و بنابراین اولین عنصر لیست یا *head* لیست است. اگر $next[x] = NIL$ عنصر x ما بعدی ندارد و

---

1. *doubly linked list*

بنابراین آخرین عنصر لیست یا *tail* لیست است. خصوصیت *head[L]* به اولین عنصر لیست اشاره می‌کند. اگر *head[L] = NIL* لیست خالی است.

یک لیست ممکن است یکی از چندین شکل را داشته باشد. ممکن است تک پیوندی یا دو پیوندی باشد، ممکن است مرتب شده باشد یا نباشد، و ممکن است حلقوی باشد یا نباشد. اگر لیست، **تک پیوندی**[1] باشد، اشاره گر *prev* در هر عنصر را حذف می‌کنیم. اگر لیست **مرتب شده**[2] باشد، ترتیب خطی لیست مطابق با ترتیب خطی کلیدها که در عناصر لیست ذخیره شده‌اند می‌باشد؛ عنصر مینیمم در ابتدای لیست و عنصر ماکزیمم در انتهای لیست قرار گرفته است. اگر لیست **نامرتب**[3] باشد، عناصر می‌توانند با هر ترتیبی ظاهر شوند. در یک **لیست حلقوی**[4]، اشاره گر *prev* عنصر ابتدایی لیست به عنصر انتهایی لیست و اشاره‌گر *next* عنصر انتهایی لیست، به عنصر ابتدایی لیست اشاره می‌کند. بنابراین ممکن است لیست به شکل حلقه‌ای از عناصر به نظر برسد. در باقیمانده این بخش، فرض می‌کنیم لیستهایی که با آنها کار می‌کنیم نا مرتب و دو پیوندی هستند.



شکل ۱۰.۳ *(a)* لیست دو پیوندی *L* که مجموعه پویای *{1,4,9,16}* را نمایش می‌دهد. هر عنصر در لیست، شیئی با فیلد کلید و اشاره گرهایی (با پیکانها نشان دادن شده‌اند) به اشیاء ما قبل و ما بعد می‌باشد. فیلد *next* عنصر انتهایی و فیلد *prev* عنصر ابتدای لیست، *NIL* هستند، که با «/» نشان داده شده‌اند. خصوصیت *head[L]* به عنصر ابتدای لیست اشاره می‌کند. *(b)* با دنبال کردن اجرای *LIST-INSERT(L,x)* که در آن *key[x] = 25* لیست پیوندی دارای شیء جدیدی با کلید 25 به عنوان اولین عنصر ابتدایی جدید می‌باشد. این شیء جدید به عنصر ابتدایی قبلی با کلید 9 اشاره می‌کند. *(c)* نتیجه فراخوانی بعدی *LIST-DELETE(L,x)* که *x* به شیء با کلید 4 اشاره می‌کند.

جستجوی یک لیست پیوندی

روال *LIST-SEARCH(L,k)* اولین عنصر با کلید *k* در لیست *L* را با استفاده از یک جستجوی خطی ساده پیدا می‌کند و یک اشاره گر به این عنصر را بر می‌گرداند. اگر هیچ شیئی با کلید *k* در لیست موجود نباشد، *NIL* بر گردانده می‌شود. برای لیست پیوندی شکل ۱۰.۳*(a)*، فراخوانی *LIST-SEARCH(L,4)*

---

1. singly linked
2. sorted
3. unsorted
4. cricular list

اشاره‌گر به عنصر سوم لیست را برگرداند و فراخوانی *LIST-SEARCH(L,7)*، *NIL* را بر می‌گرداند.

```
LIST-SEARCH(L, k)
1  x ← head[L]
2  while x ≠ NIL and key[x] ≠ k
3      do x ← next[x]
4  return x
```

برای جستجوی لیستی با n شیء، روال *LIST-SEARCH* در بدترین حالت زمان $\Theta(n)$ را صرف می‌کند، زیرا ممکن است مجبور باشد کل لیست را جستجو کند.

درج در یک لیست پیوندی

شیء x که فیلد *key* آن قبلاً مقدار دهی شده است مفروض است، روال *LIST-INSERT* x را بنحوی که در شکل ۱۰.۳ *(b)* نشان داده شده است، در ابتدای لیست پیوندی قرار می‌دهد.

```
LIST-INSERT(L, x)
1  next[x] ← head[L]
2  if head[L] ≠ NIL
3      then prev[head[L]] ← x
4  head[L] ← x
5  prev[x] ← NIL
```

زمان اجرای *LIST-INSERT* روی لیست n عنصری برابر $O(1)$ است.

حذف از لیست پیوندی

روال *LIST-DELETE* عنصر x را از لیست پیوندی L حذف می‌کند. باید اشاره گری به x به آن داده شود، و روال با به روز رسانی اشاره گرها x را از لیست خارج می‌کند. اگر بخواهیم یک عنصر با کلید داده شده را حذف کنیم، باید ابتدا *LIST-SEARCH* را برای بازیابی اشاره‌گر به آن عنصر فراخوانی کنیم.

```
LIST-DELETE(L, x)
1  if prev[x] ≠ NIL
2      then next[prev[x]] ← next[x]
3      else  head[L] ← next[x]
4  if next[x] ≠ NIL
5      then prev[next[x]] ← prev[x]
```

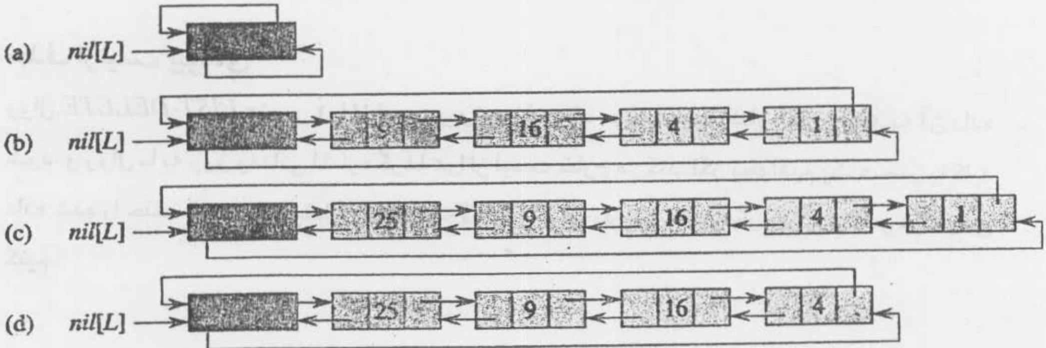
شکل (c) ۱۰.۳ نشان می‌دهد که چطور یک عنصر از لیست پیوندی حذف می‌شود. *LIST-DELETE* در زمان $O(1)$ اجرا می‌شود، ولی اگر بخواهیم عنصری با کلید داده شده را حذف کنیم در بدترین حالت به زمان $\Theta(n)$ نیاز داریم زیرا باید در ابتدا *LIST-SEARCH* را فراخوانی کنیم.

نگهبان‌ها

اگر می‌توانستیم شرایط حدی در ابتدا و انتهای لیست را نادیده بگیریم، کد *LIST-DELETE* ساده‌تر می‌شد.

```
LIST-DELETE'(L, x)
1  next[prev[x]] ← next[x]
2  prev[next[x]] ← prev[x]
```

یک **نگهبان**[1] شیئی فرضی است که با ما اجازه می‌دهد شرایط حدی را ساده کنیم. برای مثال فرض کنید که به همراه لیست L شیء *nil[L]* را داریم که *NIL* را نشان می‌دهد ولی تمام فیلدهای دیگر عناصر لیست را نیز دارا است. هر جا در لیست، ارجاعی به *NIL* داشته باشیم، آنرا با ارجاع به نگهبان *nil[L]* جایگزین می‌کنیم. همانطور که در شکل ۱۰.۴ نشان داده شده است، این تغییر لیست دو پیوندی با قاعده را به یک **لیست دو پیوندی حلقوی با یک نگهبان**[2] تبدیل می‌کند، که در آن نگهبان *nil[L]* بین عنصر ابتدایی و عنصر انتهایی لیست قرار می‌گیرد؛



شکل ۱۰.۴ یک لیست دو پیوندی حلقوی با یک نگهبان. نگهبان *nil[L]* بین عضو ابتدایی و انتهایی ظاهر شده است. به *head[L]* دیگر نیازی نیست، زیرا می‌توانیم به عضو ابتدایی لیست توسط *next[nil[L]]* دستیابی داشته باشیم. *(a)* یک لیست خالی. *(b)* لیست پیوندی شکل *(a)* ۱۰.۳ با کلید *9* در *head* و کلید *1* در *tail* *(c)* لیست پس از اجرای *LIST-INSERT' (L,x)* که *key[x]=25* شیء جدید، عضو ابتدایی لیست می‌باشد. *(d)* لیست پس از حذف شیء با کلید *1* عضو انتهایی جدید، شیئی با کلید *4* می‌باشد.

---

1. *Sentinel*

2. *circular doubly linked list with a sentinel*

فیلد *next[nil[L]]* به ابتدای لیست و *prev[nil[L]]* به انتهای آن اشاره می‌کند. بطور مشابه، فیلد *next* عنصر *tail* (انتهایی) و *prev* عنصر *head* (ابتدایی) به *nil[L]* اشاره می‌کنند. از آنجا که *next[nil[L]]* به *head* اشاره می‌کند، می‌توانیم *head[L]* را با جایگزین کردن ارجاع‌ها به *next[nil[L]]* به جای ارجاعها به آن بکلی حذف کنیم. یک لیست خالی فقط شامل نگهبان است، زیرا هم *next[nil[L]]* و هم *prev[nil[L]]* می‌توانند با *nil[L]* مقدار دهی شوند.

کد *LIST-SEARCH* به همان شکل قبلی باقی می‌ماند ولی تنها ارجاع‌ها به *NIL* و *head[L]* به همان شکلی که در بالا مشخص شده است تغییر می‌یابند.

```
LIST-SEARCH'(L, k)
1  x ← next[nil[L]]
2  while x ≠ nil[L] and key[x] ≠ k
3      do x ← next[x]
4  return x
```

از روال دو خطی *LIST-DELETE'* برای حذف یک عنصر از لیست استفاده می‌کنیم. از روال زیر برای درج یک عنصر در لیست استفاده می‌کنیم.

```
LIST-INSERT'(L, x)
1  next[x] ← next[nil[L]]
2  prev[next[nil[L]]] ← x
3  next[nil[L]] ← x
4  prev[x] ← nil[L]
```

شکل ۱۰.۴ اثرات *LIST-INSERT'* و *LIST-DELETE'* روی یک لیست نمونه را نشان می‌دهد.

نگهبان‌ها به ندرت از لحاظ مجانبی، حدهای نمایی اعمال ساختمان داده را کاهش می‌دهند می‌توانند ضرایب ثابت را کاهش دهند. فایده استفاده از نگهبان‌ها در حلقه‌ها معمولاً بیشتر در مورد وضوح کد می‌باشد تا سرعت آن؛ برای مثال، کد لیست پیوندی با استفاده از نگهبان‌ها ساده می‌شود، اما فقط زمان $O(1)$ را در روال‌های *LIST-INSERT'* و *LIST-DELETE'* صرفه جویی می‌کنیم. اما در وضعیت‌های دیگر، استفاده از نگهبان‌ها کمک می‌کند که کد را در یک حلقه به حداقل برسانیم، بنابراین ضریب n یا n^2 در زمان اجرا کاهش می‌یابد.

نگهبان‌ها نباید بصورت چشم بسته بکار روند. اگر لیست‌های کوچک زیادی وجود داشته باشند، حافظه اضافی که توسط نگهبان‌های آنها استفاده می‌شود می‌تواند حافظه تلف شده قابل توجهی باشد. در این کتاب، تنها زمانی از نگهبان‌ها استفاده می‌کنیم که سبب ساده شدن کد گردند.

تمرین‌ها

۱۰.۲-۱ آیا عمل *INSERT* مجموعه پویا روی یک لیست تک پیوندی می‌تواند در زمان $O(1)$ پیاده سازی شود؟ *DELETE* چطور؟

۱۰.۲-۲ یک پشته را با استفاده از لیست تک پیوندی L پیاده سازی کنید. اعمال *PUSH* و *POP* باید همچنان زمان $O(1)$ را صرف کنند.

۱۰.۲-۳ یک صف را با استفاده از یک لیست تک پیوندی L پیاده سازی کنید. اعمال *ENQUEUE* و *DEQUEUE* باید همچنان زمان $O(1)$ را صرف کنند.

۱۰.۲-۴ همانطور که نوشته شد، هر تکرار حلقه در روال *LIST-SEARCH'* به دو بررسی احتیاج دارد: یک بررسی برای $x \neq nil[L]$ و یکی برای $key[x] \neq k$ نشان دهید که چگونه عمل بررسی $x \neq nil[L]$ در هر تکرار را حذف کنیم.

۱۰.۲-۵ اعمال لغت نامه‌ای *INSERT*، *DELETE* و *SEARCH* را با استفاده از لیستهای حلقوی تک پیوندی پیاده سازی کنید. زمان‌های اجرای روال‌های شما چقدر است؟

۱۰.۲-۶ عمل *UNION* مجموعه پویا، دو مجموعه جدا از هم S_1 و S_2 را به عنوان ورودی دریافت می‌کند و مجموعه $S = S_1 \cup S_2$ که شامل تمام اعضای S_1 و S_2 است را برمی‌گرداند. مجموعه‌های S_1 و S_2 معمولاً توسط این عمل تخریب می‌شوند. نشان دهید که چگونه *UNION* با زمان $O(1)$ را با استفاده از ساختمان داده لیست مناسب، پیاده سازی کنیم.

۱۰.۲-۷ یک روال غیر بازگشتی با زمان $O(n)$ ارائه دهید که یک لیست تک پیوندی n عنصری را معکوس می‌کند. روال نباید بیش از حافظه ثابتی که برای خود لیست نیاز است حافظه بگیرد.

۱۰.۲-۸* توضیح دهید که چگونه لیست دو پیوندی را به جای دو اشاره گر معمول *next* و *prev* با استفاده از فقط یک مقدار اشاره گر $np[x]$ برای هر شیء پیاده‌سازی کنیم. فرض کنید تمام مقادیر اشاره‌گرها می‌توانند به شکل اعداد صحیح k بیتی تفسیر شوند و $np[x]$ را به شکل $np[x] = next[x]\ XOR\ Prev[x]$ تعریف کنید، "*or* انحصاری" k بیت از $next[x]$ و $prev[x]$ (مقدار *NIL* با 0 نشان داده می‌شود.) بیان کنید که چه اطلاعاتی برای دسترسی به عضو ابتدایی لیست (*head*) لازم است. نشان دهید که چگونه اعمال *SEARCH*، *INSERT* و *DELETE* بر روی چنین لیستی را پیاده سازی کنیم. همچنین نشان دهید که چگونه چنین لیستی در زمان $O(1)$ معکوس می‌شود.

۱۰.۳ پیاده سازی اشاره گرها و اشیاء

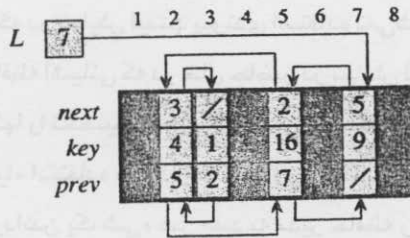
چگونه اشاره گرها و اشیاء را در زبانهایی، مانند *Fortran*، که شامل آنها نیست پیاده سازی می‌کنیم؟ در این بخش، دو راه پیاده سازی ساختمان داده‌های پیوندی بدون یک نوع داده اشاره گر صریح را خواهیم دید. اشیاء و اشاره گرها را از ترکیب آرایه‌ها و اندیس‌های آرایه بدست خواهیم آورد.

نمایش اشیاء با استفاده از چند آرایه

می‌توانیم مجموعه‌ای از اشیاء که دارای فیلدهای یکسانی هستند را با استفاده از یک آرایه برای هر فیلد، نمایش دهیم. به عنوان مثال، شکل ۱۰.۵ نشان می‌دهد که چگونه می‌توانیم لیست‌های پیوندی شکل *(a)* ۱۰.۳ را با سه آرایه پیاده سازی می‌کنیم. *key* آرایه مقادیر کلیدهای موجود در مجموعه پویا را نگه می‌دارد، و اشاره گرها در آرایه‌های *next* و *prev* ذخیره می‌شوند. برای اندیس آرایه داده شده *x*، *key[x]*، *next[x]* و *prev[x]* یک شیء در لیست پیوندی را نشان می‌دهند. بر اساس این تفسیر، اشاره گر *x* به سادگی یک اندیس مشترک برای آرایه‌های *key*، *next* و *prev* است.

در شکل *(a)*۱۰.۳ شیء با کلید *k* در لیست پیوندی به دنبال شیء با کلید *16* آمده است. در شکل ۱۰.۵، کلید *4* در *key[2]* و کلید *16* در *key[5]* ظاهر شده‌اند، بنابراین داریم *next[5]=2* و *prev[1]=5*. اگر چه ثابت *NIL* در فیلد *next* عنصر انتهایی و فیلد *prev* عنصر ابتدایی ظاهر شود، معمولاً از یک عدد صحیح (مانند *0* یا *1-*) که نمی‌تواند یک اندیس واقعی برای آرایه *A* را نشان دهد استفاده می‌کنیم. متغیر *L* اندیس عنصر ابتدای لیست را نگه می‌دارد.

در شبه کد خود از براکتهای مربعی هم برای اشاره به اندیس گذاری یک آرایه و هم برای انتخاب یک فیلد (دلخواه) شیء استفاده می‌کنیم. در هر حال که مفهوم *key[x]* ، *next[x]* و *prev[x]* مطابق با عمل پیاده سازی است.



شکل ۱۰.۵ لیست پیوندی شکل *(a)*۱۰.۳ که توسط آرایه‌های *key*، *next* و *prev* نمایش داده شده است. هر برش عمودی آرایه، یک شیء منفرد را نشان می‌دهد. اشاره گرهای ذخیره شده، متناظر با اندیس‌های آرایه نشان داده شده‌اند؛ پیکان‌ها نشان می‌دهند که چگونه آنها را تفسیر کنیم. موقعیت‌های اشیاء که سایه روشن زده شده‌اند عناصر لیست را شامل می‌شوند. متغیر *L*، اندیس عنصر ابتدایی را نگه می‌دارد.

نمایش اشیاء با استفاده از یک آرایه

کلمات در حافظه کامپیوتر نوعاً توسط اعداد صحیح از *0* تا *M-1* آدرس دهی می‌شوند، که *M* یک عدد صحیح بزرگ است. در بسیاری از زبان‌های برنامه نویسی، یک شیء، مجموعه‌ای از مکان‌های کنار هم در حافظه کامپیوتر را اشغال می‌کند. اشاره گر بسادگی آدرس اولین مکان حافظه شیء است و دیگر

مکان‌های حافظه درون شیء می‌توانند بوسیله اضافه کردن آفست به اشاره‌گر، اندیس دهی شوند.

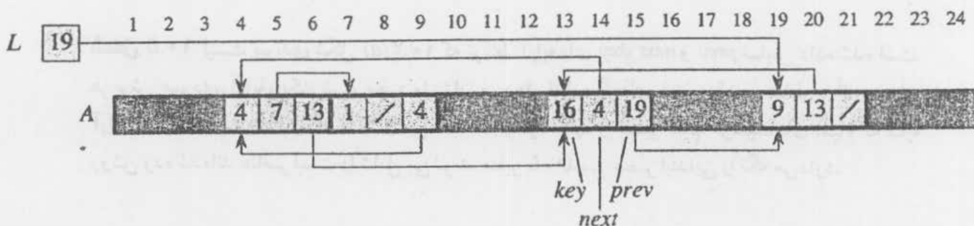
برای پیاده سازی اشیاء در محیط‌های برنامه نویسی که دارای نوع داده اشاره گر صریح نیستند از همان استراتژی استفاده می‌کنیم. به عنوان مثال، شکل ۱۰.۶ نشان می‌دهد که چگونه تک آرایه A می‌تواند برای ذخیره لیست پیوندی از شکل‌های (a) ۱۰.۳ و ۱۰.۵ استفاده شود. یک شیء، زیر آرایه $A[j\ ..\ k]$ را اشغال می‌کند. هر فیلد شی، متناظر با آفستی در دامنه تغییرات 0 تا $k-j$ است و یک اشاره‌گر به شیء، اندیس j است. در شکل ۱۰.۶، آفست‌های متناظر با key، $next$ و $prev$ به ترتیب 0، 1 و 2 هستند. برای خواندن مقدار $prev[i]$ برای اشاره گر داده شده i، مقدار i اشاره‌گر را به آفست 2 اضافه می‌کنیم، بنابراین خوانده می‌شود $A[i+2]$

نمایش با استفاده از یک آرایه انعطاف‌پذیر است، زیرا به اشیائی با طول‌های مختلف امکان داده می‌شود که در یک آرایه ذخیره شوند. مسئله مدیریت یک چنین مجموعه ناهمگنی از اشیا، سخت‌تر از مسئله مدیریت یک مجموعه همگن که در آن تمام اشیاء فیلدهای یکسانی دارند می‌باشد. از آنجا که اغلب ساختمان داده‌هایی که در نظر خواهیم گرفت از عناصر همگن تشکیل شده‌اند، برای رسیدن به مقصود کافیست که از نمایش اشیاء با استفاده از چند آرایه استفاده کنیم.

تخصیص و آزاد کردن اشیاء

برای وارد کردن یک کلید به یک مجموعه پویا که با یک لیست دو پیوندی نمایش داده شده است باید یک اشاره گر به شیء جاری که در نمایش لیست پیوندی استفاده نمی‌شود، اختصاص دهیم.

بنابراین بهتر است حافظه اشیائی که در حال حاضر در نمایش لیست پیوندی استفاده نمی‌شوند را مدیریت کنیم تا بتوانیم آنها را تخصیص دهیم. در برخی سیستم‌ها، یک *جمع کننده داده زائد*[1] مسئول تعیین اینست که کدام اشیاء استفاده نمی‌شوند. بسیاری از کاربردها، به اندازه کافی ساده هستند که می‌توانند مسئولیت بازگرداندن یک شیء غیر مفید به مدیر حافظه را به عهده بگیرند.



شکل ۱۰.۶ لیست پیوندی شکل های (a)۱۰.۳ و ۱۰.۵ که در آرایه A نمایش داده شده‌اند. هر عنصر لیست یک شیء است که یک زیر آرایه مجاور با طول 3 را اشغال می‌کند. 3 فیلد key، $next$ و $prev$ به ترتیب با آفست‌های 0، 1 و 2 متناظر می‌باشند. اشاره گر به یک شی، شاخص اولین عنصر شیء است. اشیائی که عناصر لیست را در بر دارند سایه روشن زده شده‌اند و پیکانها ترتیب لیست را نشان می‌دهند.

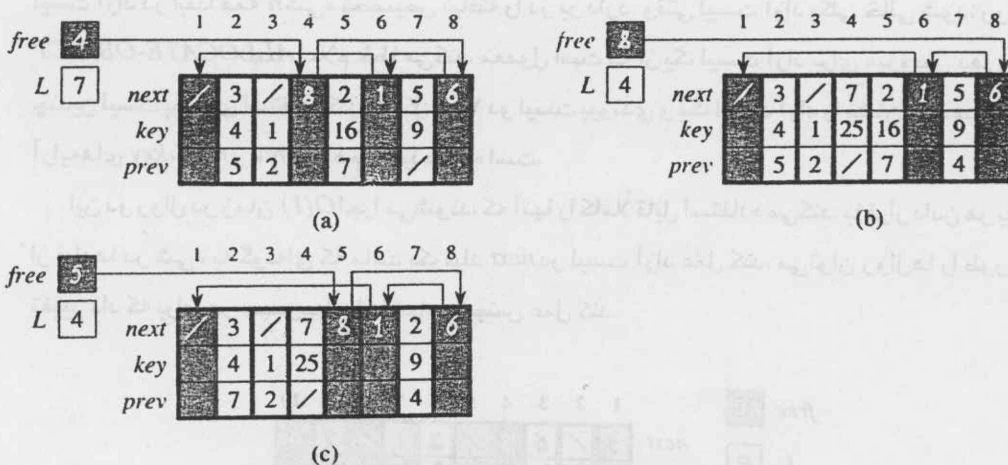
---

1. garbage collector

اینک مسئله تخصیص و آزاد کردن اشیاء همگن را با استفاده از نمونه‌ای از یک لیست دو پیوندی که توسط چند آرایه نشان داده شده است، حل خواهیم کرد.

فرض کنید آرایه‌ها در نمایش با استفاده از چند آرایه دارای طول m هستند و در بعضی اوقات مجموعه پویا $m \leq n$ عنصر دارد. بنابراین n شیء، عناصر جاری در مجموعه پویا را نمایش می‌دهند و m-n شیء باقیمانده *آزاد*[1] هستند؛ اشیاء آزاد می‌توانند برای نمایش عناصری که در آینده در مجموعه پویا درج می‌گردند استفاده شوند.

اشیاء آزاد را در یک لیست تک پیوندی نگه می‌داریم، که به آن *لیست آزاد*[2] می‌گوییم. لیست آزاد فقط از آرایه *next* که اشاره گرهای *next* را ذخیره می‌کند، استفاده کند. ابتدای لیست آزاد، در متغیر سراسری *free* نگهداری می‌شود. وقتی مجموعه پویا که توسط لیست پیوندی L نمایش داده می‌شود، غیر تهی است، لیست آزاد ممکن است مانند آنچه که در شکل ۱۰.۷ نشان داده شده، با لیست L در هم پیچیده شود. دقت داشته باشید که هر شیء در این نمایش، یا در لیستِ آزاد قرار دارد یا در لیستِ L نه در هر دو.



شکل ۱۰.۷ تأثیر روال‌های *ALLOCATE-OBJECT* و *FREE-OBJECT*. *(a)* لیست شکل ۱۰.۵ (سایه روشن زده شده است) و لیست آزاد (سایه تیره زده شده است). پیکانها ساختار لیست آزاد را نشان می‌دهند. *(b)* نتیجه فراخوانی *ALLOCATE-OBJECT* (که اندیس *4* را بر می‌گرداند)، *key[4]* با *25* مقدار دهی شده و *LIST-INSERT(L,4)* فراخوانی می‌شود. ابتدای جدید لیست آزاد، شیء *8* است که در لیست آزاد *next[4]* بوده است. *(c)* پس از اجرای *LIST-DELETE(L,5)*، *FREE-OBJECT(5)* را فراخوانی می‌کنیم. شیء *5* ابتدای جدید لیست آزاد می‌شود و شیء *8* در لیست آزاد به دنبال آن قرار می‌گیرد.

لیست آزاد یک پشته است: شیء بعدی که تخصیص می‌یابد آخرین شیئی است که آزاد شده است.

1. free
2. free list

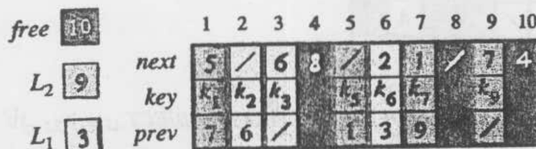
می‌توانیم از پیاده سازی لیستی از اعمال *PUSH* و *POP* پشته به ترتیب جهت پیاده سازی روال‌هایی برای تخصیص و آزاد کردن اشیاء، استفاده کنیم. فرض می‌کنیم متغیر سراسری *free* که در روال زیر استفاده شده است به اولین عنصر لیست آزاد اشاره می‌کند.

```
ALLOCATE-OBJECT()
1  if free = NIL
2     then error "out of space"
3     else x ← free
4          free ← next[x]
5          return x
```

```
FREE-OBJECT(x)
1  next[x] ← free
2  free ← x
```

لیست آزاد در ابتدا همه *n* شیء تخصیص نیافته را در بر دارد. وقتی لیست آزاد بکلی خالی شود، روال *ALLOCATE-OBJECT* اعلام خطا می‌کند. معمول است که از یک لیست آزاد برای سرویس دهی به چندین لیست پیوندی استفاده کنیم. شکل ۱۰.۸ دو لیست پیوندی و یک لیست آزاد را نشان می‌دهد که با آرایه‌های *key*، *next* و *prev* در هم پیچیده شده است.

این دو روال در زمان *O(1)* اجرا می‌شوند، که آنها را کاملاً قابل استفاده می‌کند. با قرار دادن هر یک از فیلدها در شیء به گونه‌ای که مانند یک فیلد *next* در لیست آزاد عمل کند، می‌توان روال‌ها را طوری تغییر داد که برای هر مجموعه‌ای از اشیاء همجنس عمل کند.



شکل ۱۰.۸ لیستهای دو پیوندی L_1 (سایه روشن زده شده) و L_2 (سایه تیره زده شده)، و لیست آزاد (کاملاً سیاه شده) که در هم پیچیده شده‌اند.

تمرین‌ها

۱۰.۳-۱ با استفاده از نمایش چند آرایه‌ای، تصویری از توالی <13,4,8,19,5,11> که به شکل یک لیست دو پیوندی ذخیره شده است رسم کنید. همین کار را برای نمایش تک آرایه‌ای انجام دهید.

۱۰.۳-۲ روال‌های *FREE-OBJECT* و *ALLOCATE-OBJECT* را برای یک مجموعه همجنس از

اشیاء که با نمایش تک آرایه‌ای پیاده سازی شده است، بنویسید.

۱۰.۳-۳ در پیاده سازی روال‌های *ALLOCATE-OBJECT* و *FREE-OBJECT* چرا به مقداردهی یا مقداردهی مجدد فیلدهای *prev* اشیاء نیاز نداریم؟

۱۰.۳-۴ اغلب مطلوبست که تمام عناصر یک لیست دو پیوندی را بصورت فشرده در حافظه نگهداریم، مثلاً در نمایش چند آرایه‌ای این کار با استفاده از m موقعیت اول صورت می‌گیرد. (این حالت در یک محیط محاسباتی حافظه مجازی صفحه بندی شده رخ می‌دهد.) توضیح دهید که روال‌های *ALLOCATE-OBJECT* و *FREE-OBJECT* چگونه می‌توانند پیاده سازی شوند تا این نمایش فشرده شود. فرض کنید که هیچ اشاره‌گری به عناصر لیست پیوندی در خارج از خود لیست وجود ندارد. (راهنمایی: از پیاده سازی آرایه‌ای پشته استفاده کنید.)

۱۰.۳-۵ فرض کنید L یک لیست دو پیوندی با طول m باشد که در آرایه‌های *key*، *prev* و *next* بطول n ذخیره شده است. فرض کنید که این آرایه‌ها توسط روال‌های *ALLOCATE-OBJECT* و *FREE-OBJECT* که لیست پیوندی آزاد F را نگه می‌دارند مدیریت می‌شوند. همچنین فرض کنید که از n شیء، دقیقاً m تا در لیست L و n-m تا در لیست آزاد قرار دارند. روال *COMPACTJFY-LIST(L,F)* را بنویسید که با دریافت لیست L و لیست آزاد F اشیاء را در L حرکت می‌دهد تا موقعیت‌های m , ... , 2 , 1 آرایه را اشغال کنند و لیست آزاد F را به ترتیبی تنظیم کند که صحیح باقی بماند و موقعیت‌های n , ... , $m+2$, $m+1$ آرایه را اشغال می‌کند. زمان اجرای روال شما باید $\Theta(m)$ باشد و باید فقط از یک مقدار ثابت فضای اضافی استفاده کند. اثبات دقیقی برای صحت روال خود ارائه دهید.

۱۰.۴ نمایش درختهای مشتق شده

روش‌های نمایش لیستها که در بخش قبل ارائه گردیدند برای هر ساختمان داده همجنس تعمیم می‌یابد. در این بخش، بطور خاص به مسئله نمایش درختهای مشتق شده توسط ساختمان داده‌های پیوندی می‌پردازیم. در ابتدا نگاهی به درختهای دودویی داریم و سپس برای درختهای مشتق شده که در آن‌ها گره‌ها می‌توانند تعداد دلخواهی فرزند داشته باشند روشی ارائه می‌دهیم.

هر گره از درخت را با یک شیء نشان می‌دهیم. همانند لیستهای پیوندی، فرض می‌کنیم که هر گره یک فیلد *key* دارد. بقیه فیلدهای مورد توجه، اشاره گرهایی به دیگر گره‌ها هستند و بسته به نوع درخت تغییر می‌کنند.

درختهای دودویی

همانطور که در شکل ۱۰.۹ نشان داده شده است، از فیلدهای p $left$ و $right$ برای ذخیره اشاره گرها به پدر، فرزند چپ و فرزند راست هر گره در درخت دودویی T استفاده می‌کنیم. اگر $p[x] = NIL$ آنگاه x ریشه است. اگر گره x هیچ فرزند چپی نداشته باشد آنگاه $left[x] = NIL$ و برای فرزند راست نیز به همین شکل است. بوسیله $root[T]$ به ریشه کل درخت T اشاره می‌شود. اگر $root[T] = NIL$ آنگاه درخت خالی است.

درختهای مشتق شده با انشعابهای نامحدود

طرح نمایش یک درخت دودویی می‌تواند به هر طبقه‌ای از درختهای که در آن‌ها تعداد فرزندان هر گره حداکثر عدد ثابت k است توسعه یابد: فیلدهای $left$ و $right$ را با $child_1, child_2, ..., child_n$ جایگزین می‌کنیم. هنگامیکه تعداد فرزندان یک گره نامحدود است این طرح کاربرد ندارد، زیرا نمی‌دانیم که چند فیلد (آرایه‌ها در نمایش چند آرایه‌ای) را باید تخصیص دهیم. به علاوه، حتی اگر تعداد فرزندان k بوسیله یک ثابت بزرگ محدود شود اما اکثر گره‌ها تعداد کمی فرزند داشته باشند، ممکن است حافظه زیادی را هدر دهیم.

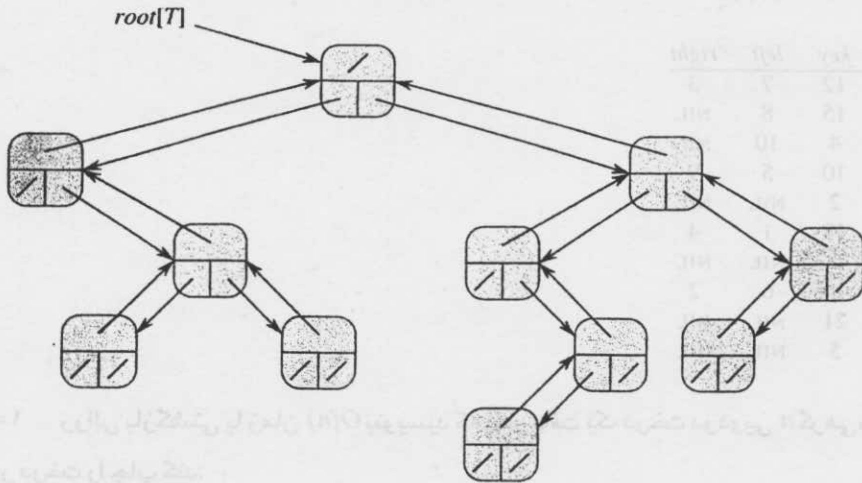
خوشبختانه، یک طرح هوشمند برای استفاده از درختهای دودویی جهت نمایش درختها با تعداد فرزند دلخواه وجود دارد و این فایده را دارد که برای هر درخت n گرهی، فقط از فضای $O(n)$ استفاده می‌شود. نمایش فرزند چپ – همزاد راست در شکل ۱۰.۱۰ نشان داده شده است. مانند قبل، هر گره دارای اشاره گر پدر p است و $root[T]$ به ریشه درخت T اشاره می‌کند. اما به جای داشتن یک اشاره‌گر به هر یک از فرزندان، هر گره x فقط دو اشاره‌گر دارد:

1. $left\text{-}child[x]$ به سمت چپ‌ترین فرزند گره x اشاره می‌کند، و

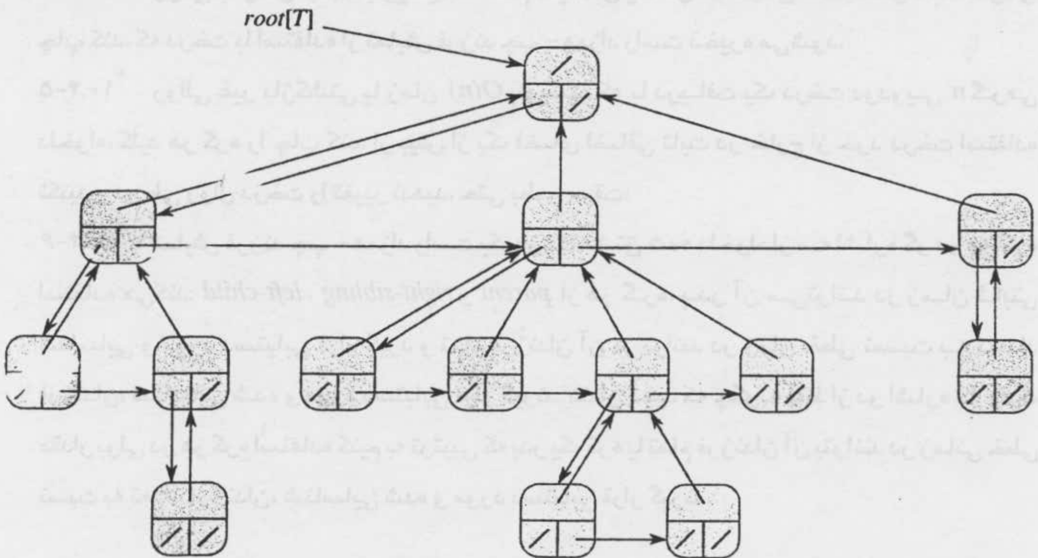
2. $right\text{-}sibling[x]$ به همزادی از x اشاره می‌کند که بلافاصله در طرف راست قرار گرفته است. اگر گره x هیچ فرزندی نداشته باشد، آنگاه $left\text{-}child[x] = NIL$ و اگر گره x سمت راست‌ترین فرزند پدرش باشد، آنگاه $right\text{-}sibling[x] = NIL$.

نمایش‌های دیگر درخت

گاهی اوقات درختهای مشتق شده را به اشکال دیگری نمایش می‌دهیم. به عنوان مثال در فصل ۶، heap که بر مبنای یک درخت دودویی کامل است را بوسیله یک آرایه منفرد به اضافه یک اندیس، نمایش دادیم. درختهایی که در فصل ۲۱ آورده شده‌اند فقط به طرف ریشه پیمایش می‌شوند، بنابراین فقط اشاره گرهای پدر ارائه می‌شود؛ اشاره‌گری به فرزندان وجود ندارد. شکل‌های زیاد دیگری نیز هستند. اینکه کدام طرح بهترین است به کاربرد بستگی دارد.



شکل ۱۰.۹ نمایش درخت دودویی *T*. هر گره *x* دارای فیلدهای *p[x]* در بالا، *left[x]* (پایین سمت چپ) و *right[x]* (پایین سمت راست) است. فیلدهای *key* نشان داده نشده‌اند.



شکل ۱۰.۱۰ نمایش فرزند چپ ـ همزاد راست درخت *T*. هر گره *x* دارای فیلدهای *p[x]* (در بالا) *left-child[x]* (پایین سمت چپ) و *right-sibling[x]* (پایین سمت راست) است. کلیدها نشان داده نشده‌اند.

تمرین‌ها

۱-۱۰.۴ یک درخت دودویی بکشید که از اندیس 6 مشتق شده و با فیلدهای زیر نمایش داده می‌شود.

index	*key*	*left*	*right*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

۲-۱۰.۴ روالی بازگشتی با زمان $O(n)$ بنویسید که با دریافت یک درخت دودویی n گرهی، کلید هر گره در درخت را چاپ کند.

۳-۱۰.۴ یک روال غیر بازگشتی با زمان $O(n)$ بنویسید که با دریافت یک درخت دودویی n گرهی، کلید هر گره در درخت را چاپ کند. از یک پشته به عنوان ساختمان داده کمکی استفاده کنید.

۴-۱۰.۴ روالی با زمان $O(n)$ بنویسید که تمام کلیدهای یک درخت مشتق شده دلخواه با n گره را چاپ کند، که درخت با استفاده از نمایش فرزند چپ - همزاد راست ذخیره می‌شود.

۵-۱۰.۴* روالی غیر بازگشتی با زمان $O(n)$ بنویسید که با دریافت یک درخت دودویی n گرهی دلخواه، کلید هر گره را چاپ کند. از بیش از یک فضای اضافی ثابت در خارج از خود درخت استفاده نکنید و در طی روال درخت را تغییر ندهید، حتی بطور موقت.

۶-۱۰.۴* نمایش فرزند چپ - همزاد راست یک درخت مشتق شده دلخواه از سه اشاره گر در هر گره استفاده می‌کند: *left-child*، *right-sibling* و *parent* از هر گره، پدر آن می‌توانند در زمان ثابتی شناسایی و مورد دستیابی قرار گیرد و تمام فرزندان آن می‌توانند در زمان خطی نسبت به تعداد فرزندان، شناسایی شده و مورد دستیابی قرار گیرند. نشان دهید که چگونه فقط از دو اشاره گر و یک مقدار بولی در هر گره استفاده کنیم به ترتیبی که پدر یک گره یا تمام فرزندان آن بتوانند در زمانی خطی نسبت به تعداد فرزندان، شناسایی شده و مورد دستیابی قرار گیرند.

مسائل

۱-۱۰ مقایسه بین لیستها

برای هر یک از ۴ نوع لیست که در جدول زیر آمده است، بدترین زمان اجرای مجانبی هر عمل لیست شده برای مجموعه پویا چیست؟

	تک پیوندی نامرتب	تک پیوندی مرتب	دو پیوندی نامرتب	دو پیوندی مرتب
SEARCH(L,K)				
INSERT(L,x)				
DELETE(L,x)				
SUCCESSOR(L,x)				
PREDECESSOR(L,x)				
MINIMUM(L)				
MAXIMUM(L)				

۲-۱۰ *heap* های قابل ادغام با استفاده از لیست پیوندی

یک *heap* قابل ادغام این اعمال را پشتیبانی می‌کند: *Make-Heap* (که یک *heap* قابل ادغام خالی ایجاد می‌کند)، *INSERT* ، *MINIMUM* ، *EXTRACT-MIN* و *UNION*. نشان دهید در هر یک از حالتهای زیر چگونه *heap* قابل ادغام را با استفاده از لیست پیوندی پیاده‌سازی کنیم. سعی کنید هر یک از اعمال فوق به کاراترین شکل ممکن انجام شود. زمان اجرای هر عمل را بر حسب اندازه مجموعه(های) پویا که عمل روی آن انجام می‌شود تحلیل کنید.

a.لیستها مرتب شده هستند.

b.لیستها غیر مرتبند.

c.لیستها غیر مرتبند و مجموعه‌های پویا که ادغام می‌شوند جدا از هم هستند.

۳-۱۰ جستجوی یک لیست فشرده مرتب شده

تمرین ۴-۳-۱۰ می‌پرسد که چگونه ممکن است یک لیست n عنصری را بطور فشرده در n موقعیت اول آرایه نگه داریم. فرض خواهیم کرد که تمام کلیدها متمایزند و لیست فشرده نیز مرتب شده است، بعبارت دیگر برای $i = 1,2, \ldots ,n$ است $key[i] < key[next[i]]$ بطوریکه $next[i] \neq NIL$ با این فرض‌ها نشان خواهید داد که الگوریتم تصادفی زیر می‌تواند برای جستجوی لیست در زمان مورد انتظار $(O\sqrt{n})$ استفاده شود.

```
COMPACT-LIST-SEARCH(L, n, k)
1  i ← head[L]
2  while i ≠ NIL and key[i] < k
3       do j ← RANDOM(1, n)
```

```
4          if key[i] < key[j] and key[j] ≤ k
5             then i ← j
6                  if key[i] = k
7                     then return i
8          i ← next[i]
9   if i = NIL or key[i] > k
10     then return NIL
11     else return i
```

اگر خطوط ۷-۳ روال را در نظر نگیریم، یک الگوریتم معمولی برای جستجوی یک لیست پیوندی مرتب داریم، که در آن اندیس i به هر موقعیت از لیست به نوبت اشاره می‌کند. جستجو وقتی که اندیس i از انتهای لیست خارج شود یا زمانیکه $key[i] \geq k$ شود، پایان می‌یابد. در حالت دوم اگر $key[i] = k$ باشد، مشخص است که کلیدی با مقدار k پیدا کرده‌ایم اما اگر $key[i] > k$ باشد، هیچگاه کلیدی با مقدار k پیدا نخواهیم کرد و بنابراین پایان دادن جستجو کار درستی بوده است.

خطوط ۷-۳ تلاش می‌کنند که به سمت جلو و به موقعیت j که بصورت تصادفی انتخاب شده است پرش کنند. در صورتی که $key[j]$ بزرگتر از $key[i]$ و کوچکتر یا مساوی k باشد این پرش سودمند است؛ در چنین حالتی، j موقعیتی در لیست که i می‌تواند در طی یک جستجوی معمولی لیست به آن برسد را علامت می‌زند. بدلیل فشرده بودن لیست، می‌دانیم که هر انتخاب j بین 1 و n به جای یک شکاف در لیست آزاد، یک شیء در لیست را مشخص می‌کند.

به جای تحلیل مستقیم کارایی *COMPACT-LIST-SEARCH* یک الگوریتم مربوط به آن را تحلیل خواهیم کرد، *'COMPACT-LIST-SEARCH*، که دو حلقه مجزا را اجرا می‌کند. این الگوریتم پارامتر اضافه t که یک حد بالا برای تعداد تکرارهای اولین حلقه را مشخص می‌کند، دریافت می‌کند.

```
COMPACT-LIST-SEARCH'(L, n, k, t)
1   i ← head[L]
2   for q ← 1 to t
3       do j ← RANDOM(1, n)
4          if key[i] < key[j] and key[j] ≤ k
5             then i ← j
6                  if key[i] = k
7                     then return i
8   while i ≠ NIL and key[i] < k
9       do i ← next[i]
10  if i = NIL or key[i] > k
11     then return NIL
12     else return i
```

بـــــــــرای مـــــــــقایسه اجـــــــــرای الگـــــــــوریتم‌های *COMPACT-LIST-SEARCH(L,k)* و *COMPAT-LIST-SEARCH' (L,k,t)*، فرض می‌کنیم که توالی اعداد صحیح بازگردانده شده بوسیله فراخوانی‌های *RANDOM(1,n)*، برای هر دو الگوریتم یکسانند.

a. فرض کنید *COMPACT-LIST-SEARCH(L,k)*، t تکرار حلقه *while* خطوط ۸-۲ را اعمال می‌کند. ثابت کنید *COMPACT-LIST-SEARCH' (L,k,t)* همان پاسخ را برمی گرداند و اینکه تعداد کل تکرارهای هر دو حلقه *for* و *while* در *COMPACT-LIST-SEARCH'* حداقل t است.

در فراخوانی *COMPACT-LIST-SEARCH' (L,k,t)* فرض کنید X_t متغیر تصادفی باشد که در لیست پیوندی (به عبارت دیگر در طول زنجیره اشاره‌گرهای *next*) فاصله بین موقعیت و کلید مطلوب k بعد از t تکرار حلقه *for* خطوط ۷-۲، را نشان می‌دهد.

b. ثـابت کـنید کـه زمـان اجـرای مـورد انـتظار *COMPACT-LIST-SEARCH' (L,k,t)* بـرابـر $O(t+E[X_t])$ است.

c. نشان دهید که

$$E[X_t] \le \sum_{r=1}^{n}(1-r/n)^t.$$

(راهنمایی: از معادله

$$E[X] = \sum_{i=1}^{\infty} \Pr\{X \ge i\}$$

استفاده کنید.)

d. نشان دهید که

$$\sum_{r=0}^{n-1} r^t \le n^{t+1}/(t+1).$$

e. ثابت کنید که $E[X_t] \le n/(t+1)$.

f. نشان دهید کـه *COMPACT-LIST-SEARCH' (L,k,t)* در زمـان مـورد انـتظار $O(t+n/t)$ اجرا می‌شود.

g. نتیجه بگیرید که زمان اجرای مورد انتظار *COMPACT-LIST-SEARCH* برابر $O(\sqrt{n})$ است.

h. چرا فرض مـی‌کنیم کـه تـمام کـلیدها در *COMPACT-LIST-SEARCH* مـتمایزند؟ ثـابت کـنید هنگامیکه لیست شامل مقادیر کلید تکراری است، پرسشهای تصادفی لزوماً از لحاظ مجانبی کمکی نمی‌کنند.

۱۱ جدول‌های درهم سازی

بسیاری از کاربردها، به یک مجموعه پویا نیاز دارند که فقط اعمال لغت نامه‌ای *SEARCH* *INSERT* و *DELETE* را پشتیبانی می‌کند. به عنوان مثال، یک کامپایلر برای یک زبان کامپیوتر یک جدول نماد را نگه می‌دارد که در آن کلیدهای عناصر رشته‌های کاراکتری دلخواهی هستند که مطابق با شناسه‌ها در زبان می‌باشند. یک جدول درهم سازی، ساختمان داده‌ای مفید برای پیاده سازی لغت نامه‌ها است. اگر چه جستجوی یک عنصر در یک جدول درهم سازی در عمل می‌تواند به اندازه جستجوی یک عنصر لیست پیوندی - در بدترین حالت $\Theta(n)$ - زمان ببرد، اما در هم سازی بسیار عالی می‌کند. تحت فرضیات معقول، زمان مورد انتظار برای جستجوی یک عنصر در یک جدول درهم سازی برابر $O(1)$ است.

جدول درهم سازی، تعمیمی از ایده ساده‌تر آرایه‌های معمولی است. آدرس دهی مستقیم به یک آرایه معمولی، از توانایی ما در بررسی یک موقعیت دلخواه در آرایه در زمان $O(1)$ استفاده مفیدی می‌کند. بخش ۱۱.۱ آدرس دهی مستقیم را با جزئیات بیشتری مورد بحث قرار می‌دهد. آدرس دهی مستقیم وقتی کاربرد دارد که بتوانیم یک آرایه که یک مکان برای هر کلید ممکن دارد را تخصیص دهیم.

هنگامیکه تعداد کلیدهایی که واقعاً ذخیره می‌شوند نسبت به تعداد کل کلیدهای ممکن کم است، جدول در هم سازی جایگزینی مناسب برای آدرس دهی مستقیم به آرایه می‌شود، زیرا جدول درهم سازی نوعاً از یک آرایه با اندازه‌ای متناسب با تعداد کلیدهایی که واقعاً ذخیره می‌شوند استفاده می‌کند. به جای استفاده مستقیم از کلید به عنوان اندیس آرایه، اندیس آرایه از کلید محاسبه می‌شود. بخش ۱۱.۲ ایده اصلی را ارائه می‌دهد، با تمرکز بر "زنجیره سازی" به عنوان راهی برای مدیریت "برخوردها" که در آن بیش از یک کلید به یک اندیس آرایه نگاشت می‌شود. بخش ۱۱.۳ بیان می‌کند که چگونه اندیس‌های آرایه می‌توانند از کلیدها و با استفاده از توابع درهم سازی محاسبه شوند. چندین گونه متفاوت در طرح اصلی را عنوان کرده و تحلیل می‌کنیم. بخش ۱۱.۴ نگاهی دارد به "آدرس دهی باز" که راهی دیگر برای مدیریت برخوردها است. مهمترین نکته اینست که در هم سازی تکنیکی عملی و

بسیار کارآمد است: اعمال لغت نامه‌ای اصلی در حالت میانگین تنها به زمان $O(1)$ نیاز دارند. بخش ۱۱.۵ توضیح می‌دهد که چگونه "درهم سازی کامل" می‌تواند جستجوها را در بدترین حالت در زمان $O(1)$ پشتیبانی کند.

۱۱.۱ جدول‌های آدرس مستقیم

آدرس دهی مستقیم یک تکنیک ساده است که وقتی مجموعه مرجع U از کلیدها به طور معقول کوچک است به خوبی عمل می‌کند. فرض کنید که یک کاربرد به مجموعه‌ای پویا نیاز دارد که در آن هر عنصر، کلیدی دارد که از مجموعه مرجع $U = \{0,1,...,m-1\}$ انتخاب می‌شود، که m خیلی بزرگ نیست. فرض خواهیم کرد که هیچ دو عنصری کلید یکسانی ندارند.

برای نشان دادن مجموعه پویا، از یک آرایه یا جدول آدرس مستقیم استفاده می‌کنیم، که با $T[0 \,..\, m-1]$ نشان داده می‌شود، و در آن هر موقعیت یا مکان متناظر با کلیدی در مجموعه مرجع U است. شکل ۱۱.۱ مسیر را شرح می‌دهد؛ مکان k به یک عضو از مجموعه با کلید k اشاره می‌کند. اگر مجموعه عنصری با کلید k نداشته باشد آنگاه $T[k] = NIL$.

```
DIRECT-ADDRESS-SEARCH(T, k)
    return T[k]

DIRECT-ADDRESS-INSERT(T, x)
    T[key[x]] ← x

DIRECT-ADDRESS-DELETE(T, x)
    T[key[x]] ← NIL
```

همه این اعمال سریعند: فقط زمان $O(1)$ نیاز است.

برای برخی کاربردها اعضای مجموعه پویا می‌توانند در خود جدول آدرس مستقیم ذخیره شوند. بعبارت دیگر، به جای ذخیره کردن کلید عنصر و داده‌های وابسته در یک شیء در خارج از جدول آدرس مستقیم، و قرار دادن یک اشاره گر به شیء در یک مکان از جدول، می‌توانیم خودش را در آن مکان ذخیره کنیم، و لذا در فضا صرف جویی کنیم. بعلاوه اغلب لازم نیست که فیلد کلید شیء را ذخیره کنیم، زیرا اگر اندیس یک شیء در جدول را داشته باشیم، کلید آن را داریم. اما اگر کلیدها ذخیره نشوند، باید راهی برای عنوان کردن اینکه آیا آن مکان خالی است داشته باشیم.

تمرین‌ها

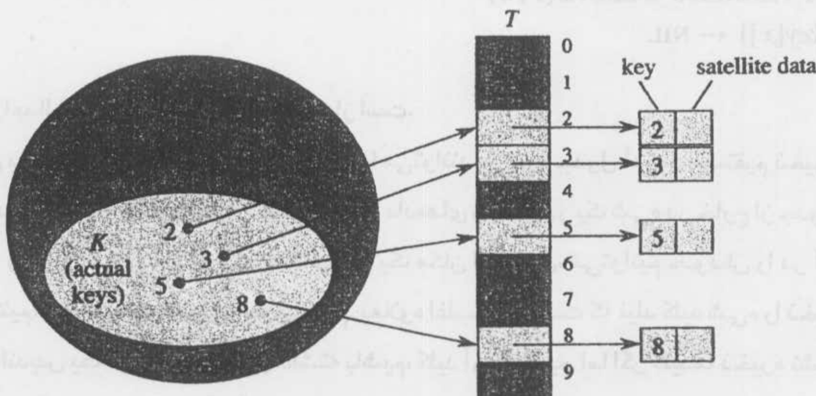
۱۱.۱-۱ فرض کنید که مجموعه پویای S توسط جدول آدرس مستقیم T بطول m نمایش داده شود.

روالی را شرح دهید که عنصر ماکزیمم S را پیدا می‌کند. بدترین حالت اجرای روال شما چیست؟

۱۱.۱-۲ یک بردار بیتی بصورت ساده، آرایه‌ای از بیت‌ها(0ها و 1ها) است. یک بردار بیتی بطول m، نسبت به آرایه‌ای با mاشاره‌گر فضای بسیار کمتری را اشغال می‌کند. نشان دهید که چگونه می‌توان از یک بردار بیتی برای نشان دادن یک مجموعه پویا از عناصری که هیچ داده وابسته‌ای ندارند استفاده کرد. اعمال لغت نامه‌ای باید در زمان $O(1)$ اجرا شوند.

۱۱.۱-۳ برای چگونگی پیاده سازی یک جدول آدرس مستقیم که در آن لازم نیست کلیدهای عناصر ذخیره شده متفاوت باشند و عناصر می‌توانند داده‌های وابسته داشته باشند، روشی پیشنهاد کنید، هر سه عمل لغت نامه‌ای (INSERT، DELETE و SEARCH) باید در زمان $O(1)$ اجرا شوند. (فراموش نکنید که DELETE به عنوان آرگومان، یک اشاره‌گر به شیئی که باید حذف شود را دریافت می‌کند نه کلید را)

*۱۱.۱-۴ می‌خواهیم یک لغت نامه را با استفاده از آدرس دهی مستقیم در یک آرایه بزرگ پیاده سازی کنیم. در ابتدا، ورودی‌های آرایه ممکن است شامل داده‌های بی‌ارزش باشند و مقدار دهی اولیه آرایه ورودی به خاطر اندازه‌اش غیر عملی است. طرحی برای پیاده سازی یک لغت نامه آدرس مستقیم روی یک آرایه بزرگ شرح دهید. هر شیء ذخیره شده باید از فضای $O(1)$ استفاده کند؛ هر یک از اعمال INSERT، DELETE و SEARCH باید زمان $O(1)$ را صرف کنند؛ و مقدار دهی اولیه ساختمان داده باید زمان $O(1)$ را صرف کند. (راهنمایی: برای کمک به تعیین اینکه آیا یک ورودی داده شده در آرایه بزرگ، معتبر است یا نه، از یک پشته اضافی استفاده کنید که اندازه‌اش برابر با تعداد کلیدهایی است که واقعاً در لغت نامه ذخیره شده‌اند.)



شکل ۱۱.۱ پیاده سازی یک مجموعه پویا با استفاده از جدول آدرس مستقیم T. هر کلید در مجموعه مرجع $U = \{0,1,...,9\}$ با یک اندیس در جدول متناظر است. مجموعه $k = \{2,3,5,8\}$ از کلیدهای واقعی، مکانهایی در جدول که شامل اشاره گرها به عناصر هستند را مشخص می‌کند. مکان‌های دیگر که تیره سایه زده شده‌اند شامل NIL هستند.

۱۱.۲ جدول‌های درهم سازی

مشکلات آدرس دهی مستقیم، روشن و واضح است: اگر مجموعه مرجع U بزرگ باشد، ذخیره کردن جدول T با اندازه $|U|$ با حافظه داده شده در یک کامپیوتر معمولی ممکن است عملی نباشد یا حتی غیر ممکن باشد. علاوه بر آن، مجموعه k از کلیدهایی که واقعاً ذخیره می‌شوند ممکن است نسبت به U بسیار کوچک باشد و مقدار زیادی از فضایی که برای T تخصیص یافته، ممکن است هدر برود.

وقتی مجموعه k از کلیدهای ذخیره شده در یک لغت نامه نسبت به مجموعه U از همه کلیدهای ممکن، بسیار کوچکتر است یک جدول درهم سازی نسبت به جدول آدرس مستقیم به فضای کمتری نیاز دارد. خصوصاً هنگامیکه این مزیت را حفظ می‌کنیم که جستجوی یک عنصر در جدول درهم سازی نیاز به زمان $O(1)$ دارد، می‌تواند به $\Theta(|k|)$ کاهش یابد. تنها دستاورد اینست که این حد برای زمان میانگین است، در حالیکه برای آدرس دهی مستقیم در بدترین حالت زمانی بر قرار است.

با آدرس دهی مستقیم، عنصر با کلید k در مکان k ذخیره می‌شود. با درهم سازی، این عنصر در مکان $h(k)$ ذخیره می‌شود؛ بعبارت دیگر، از ***تابع درهم سازی***[1] بنام h برای محاسبه مکان از روی کلید k استفاده می‌کنیم. در اینجا h مجموعه مرجع U از کلیدها را به مکانهایی از ***جدول درهم سازی***[2] $T[0 .. m-1]$ نگاشت می‌کند.

$$h: U \rightarrow \{0, 1, ..., m-1\}$$

می‌گوییم عنصری با کلید k به مکان $h(k)$ ***درهم سازی***[3] می‌شود؛ همچنین می‌گوییم که $h(k)$ ***مقدار درهم سازی شده***[4] کلید k است. شکل ۱۱.۲ ایده اصلی را شرح می‌دهد. ایده اصلی تابع درهم سازی کاهش دادن دامنه تغییرات اندیس‌های آرایه است که لازم است مدیریت شوند. به جای $|U|$ مقدار، نیاز به مدیریت فقط m مقدار داریم. نیازمندیهای حافظه‌ای نیز متقابلاً کاهش می‌یابد.

در اینجا یک مشکل وجود دارد: دو کلید ممکن است به یک مکان یکسان درهم سازی شوند. این وضعیت را یک **برخورد**[5] می‌نامیم. خوشبختانه تکنیک‌های کارایی برای حل این مشکلات که بوسیله برخوردها ایجاد شده‌اند، وجود دارد.

البته جواب ایده‌آل باید از برخورد با یکدیگر جلوگیری کند. ممکن است سعی کنیم با استفاده از انتخاب تابع درهم سازی مناسب h به این هدف دست بیابیم. یک ایده اینست که h را طوری بسازیم که تصادفی به نظر برسد، بنابراین از برخوردها جلوگیری می‌شود یا حداقل تعداد آن‌ها کم می‌شود. اصطلاح "درهم سازی" که تصویرهای درهم سازی و جدا سازی تصادفی را موجب می‌شود، روح این

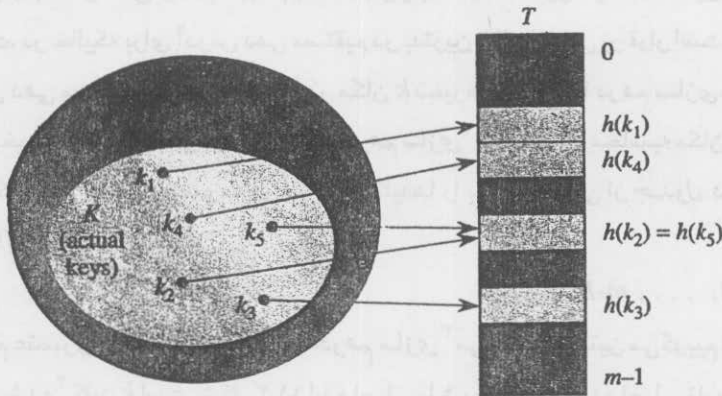
---

1. hash function
2. hash table
3. hash
4. hash value
5. collision

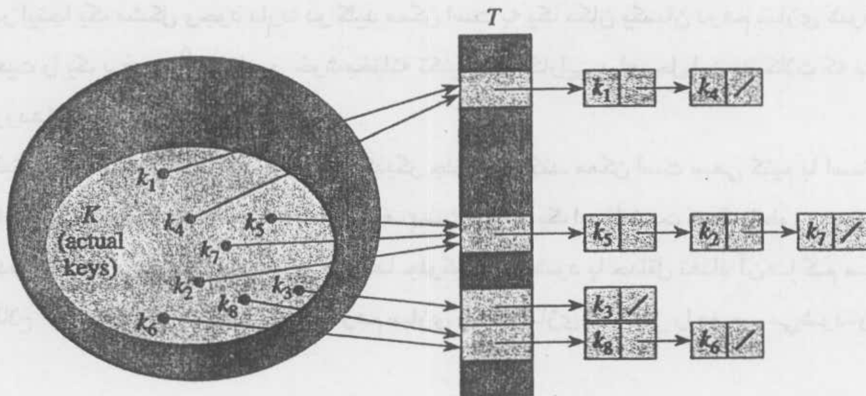
روش را تسخیر می‌کند. (البته رفتار تابع درهم سازی h باید معین باشد، یعنی ورودی k همیشه باید همان خروجی $h(k)$ را ایجاد کند.) اما از آنجا که $|U| > m$، باید حداقل دو کلید وجود داشته باشند که مقدار درهم سازی یکسانی دارند؛ پس جلوگیری از برخورد با هم غیر ممکن است.

بنابراین علیرغم اینکه یک تابع درهم سازی به نظر تصادفی و خوب طراحی شده است، می‌توانند تعداد برخوردها را کم کنند، همچنان به روشی برای حل مجدد برخوردهایی که رخ می‌دهند احتیاج داریم.

ادامه این فصل ساده‌ترین تکنیک‌های حل مجدد برخوردها را ارائه می‌دهد، که زنجیره سازی نامیده می‌شوند بخش ۱۱.۴ یک روش دیگر برای حل مجدد برخوردها معرفی می‌کند که آدرس دهی باز نامیده می‌شود.



شکل ۱۱.۲ استفاده از تابع درهم سازی h برای نگاشت کلیدها به مکان‌های جدول درهم سازی. کلیدهای k_2 و k_5 به یک مکان نگاشت شده‌اند، بنابراین با هم برخورد دارند.



شکل ۱۱.۳ حل مجدد برخورد با استفاده از زنجیره سازی هر مکان $T[j]$ دارای یک لیست پیوندی از تمام کلیدهایی است که مقدار درهم سازی شده آنها برابر j است. به عنوان مثال $h(k_1) = h(k_4)$ و $h(k_5) = h(k_2) = h(k_7)$

حل برخوردها با استفاده از زنجیره سازی

در *زنجیره سازی*[1] همان طور که در شکل ۱۱.۳ نشان داده شده است تمام عناصری که به یک مکان درهم‌سازی می‌شوند را در یک لیست پیوندی قرار می‌دهیم. مکان j دارای اشاره‌گری به ابتدای لیست تمام عناصر ذخیره شده‌ای است که به j درهم سازی شده‌اند؛ اگر چنین عناصری وجود نداشته باشند مکان j دارای مقدار *NIL* است.

هنگامیکه برخوردها با استفاده از زنجیره سازی حل شده باشند، پیاده‌سازی اعمال لغت نامه‌ای روی جدول درهم سازی T آسان است.

```
CHAINED-HASH-INSERT(T, x)
    insert x at the head of list T[h(key[x])]

CHAINED-HASH-SEARCH(T, k)
    search for an element with key k in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
    delete x from the list T[h(key[x])]
```

بدترین حالت زمان اجرا برای درج برابر $O(1)$ است. روال درج تا حدی سریع است زیرا فرض می‌کند عنصر x که درج می‌شود، قبلاً در جدول وجود نداشته است؛ این فرض، اگر لازم باشد می‌تواند با انجام یک جستجو قبل از درج بررسی شود (با هزینه اضافی). برای جستجو، بدترین حالت زمان اجرا متناسب با طول لیست است؛ این عمل را بطور دقیق‌تر در زیر تحلیل خواهیم کرد. اگر لیست‌ها دو پیوندی باشند، حذف عنصر x می‌تواند در زمان $O(1)$ انجام شود. (توجه داشته باشید که *CHAINED-HASH-DELETE* به عنوان ورودی عنصر x را می‌گیرد نه کلید k آن را، بنابراین در ابتدا مجبور نیستیم که x را جستجو کنیم. اگر لیست‌ها تک پیوندی بودند، گرفتن x به عنوان ورودی به جای کلید k کمک چندانی نمی‌کرد. هنوز باید x را در لیست $T[h(key[x])]$ پیدا کنیم، تا اتصال *next* عنصر ما قبل x برای جدا کردن x به درستی تنظیم شود. در این حالت، حذف و جستجو باید لزوماً زمان اجرای یکسانی داشته باشند.)

تحلیل درهم‌سازی با استفاده از زنجیره سازی

در هم سازی با زنجیره سازی به چه شکل کار می‌کند؟ مخصوصاً چقدر زمان می‌برد تا برای یک عنصر با کلید داده شده جستجو را انجام دهیم؟

---

1. chaining

جدول درهم سازی T با m مکان که n عنصر را ذخیره می‌کند داده شده است، *ضریب بار*[1] α برای T را به شکل n/m تعریف می‌کنیم، بعبارت دیگر میانگین تعداد عناصر ذخیره شده در یک زنجیره. تحلیل ما بر حسب α خواهد بود که می‌تواند کوچکتر، مساوی یا بزرگتر از یک باشد.

رفتار درهم سازی با استفاده از زنجیره سازی در بدترین حالت، بسیار نامناسب است: همه n کلید به یک مکان یکسان درهم سازی می‌شوند و لیستی به طول n ایجاد می‌کنند. بنابراین بدترین حالت زمانی برای جستجو برابر $\Theta(n)$ به اضافه زمان لازم برای محاسبه تابع درهم سازی است و در حالتی که از یک لیست پیوندی برای همه عناصر استفاده می‌کردیم بهتر نیست. واضح است که، جداول درهم‌سازی به خاطر اجرا در بدترین حالتشان استفاده نمی‌شوند (با این وجود درهم سازی کامل، که در بخش ۱۱.۵ توضیح داده شده است بدترین حالت کارآیی خوبی را فراهم می‌کند هنگامیکه مجموعه کلیدها ایستا است.)

اجرای میانگین درهم سازی به این بستگی دارد که تابع درهم سازی h در حالت میانگین چگونه مجموعه کلیدها را به منظور اینکه در m مکان ذخیره شوند توزیع می‌کند. بخش ۳-۱۱ این جنبه را بررسی می‌کند، ولی در اینجا ما فرض می‌کنیم که هر عنصر به شکلی برابر احتمال دارد که به هر یک از m مکان درهم‌سازی شود، بدون توجه به اینکه هر یک از عناصر دیگر به کجا درهم سازی شده‌اند. این فرض را درهم‌سازی *یکنواخت ساده*[2] می‌نامیم.

$j = 0, 1, ..., m-1$، برای اجازه دهید به طول لیست $T[j]$ توسط n_j اشاره کنیم بطوریکه

$$n = n_0+n_1+...+n_{m-1} \quad (۱۱.۱)$$

و مقدار میانگین n_j برابر $E[n_j] = \alpha = n/m$ است.

فرض می‌کنیم که مقدار درهم سازی $h(k)$ می‌تواند در زمان $O(1)$ محاسبه شود، تا زمان لازم برای جستجوی یک عنصر با کلید k به طور خطی به طول $n_{h(k)}$ از لیست $T[h(k)]$ بستگی داشته باشد. با کنار گذاشتن زمان $O(1)$ لازم برای محاسبه تابع درهم سازی و دستیابی به مکان $h(k)$ تعداد عناصری که انتظار می‌رود در الگوریتم جستجو بررسی شوند را در نظر می‌گیریم، بعبارت دیگر تعداد عناصر در لیست $T[h(k)]$ که کنترل می‌شوند، تا ببینیم آیا کلیدهای آنها با k برابر است. دو حالت را در نظر می‌گیریم. در حالت اول، جستجو موفقیت‌آمیز نیست: هیچ عنصری در جدول، کلید k را ندارد. در حالت دوم، جستجوی موفقیت‌آمیز عنصری با کلید k را پیدا می‌کند.

قضیه ۱۱.۱

در یک جدول درهم سازی که در آن برخوردها با استفاده از زنجیره سازی حل می‌شوند، با فرض اینکه درهم سازی یکنواخت ساده است، یک جستجوی ناموفق، زمان مورد انتظار $\Theta(1+\alpha)$ را صرف

---

1. load factor
2. simple uniform hashing

می‌کند.

اثبات با فرض درهم سازی یکنواخت ساده، هر کلید k که قبلاً در جدول ذخیره نشده است بطور مساوی احتمال دارد که در هر یک از m مکان قرار گیرد. زمان مورد انتظار برای جستجوی نا موفق یک کلید، زمان مورد انتظار برای جستجو تا انتهای لیست $T[h(x)]$ است، که طول آن $E[n_{h(k)}] = \alpha$ است. بنابراین تعداد عناصری که انتظار می‌رود بررسی شوند، در یک جستجوی ناموفق α است، و زمان کلی که لازم است (شامل زمان محاسبه $h(k)$) برابر $\Theta(1+\alpha)$ است. ■

شرایط برای یک جستجوی موفق تا حدی متفاوت است، زیرا هر لیست بطور مساوی احتمال ندارد که مورد جستجو قرار گیرد. در عوض، احتمال اینکه یک لیست جستجو شود با تعداد عناصری که دارد متناسب است. با این حال زمان جستجوی مورد انتظار همچنان $\Theta(1+\alpha)$ می‌باشد.

قضیه ۱۱.۲

در یک جدول درهم سازی که در آن برخوردها با استفاده از زنجیره سازی حل می‌شود، با فرض اینکه درهم سازی یکنواخت ساده است، یک جستجوی موفق در حالت میانگین، زمان $\Theta(1+\alpha)$ را صرف می‌کند.

اثبات فرض می‌کنیم عنصری که دنبال آن می‌گردیم به طوری برابر احتمال دارد که در هر یک از n عنصر ذخیره شده در جدول باشد. تعداد عناصری که در طی یک جستجوی موفق برای عنصر x بررسی می‌شوند یکی بیشتر از تعداد عناصری است که قبل از x در لیست x ظاهر می‌شوند. عناصر قبل از x در لیست، همگی بعد از درج x درج شده‌اند زیرا عناصر جدید در جلو (ابتدای) لیست قرار می‌گیرند. برای پیدا کردن تعداد عناصری که انتظار می‌رود بررسی شوند، روی n عنصر x در جدول میانگین می‌گیریم که برابر است با 1 به اضافه تعداد عناصری که انتظار می‌رود بعد از درج x در لیست x درج شده باشند. برای $i = 1, 2, \ldots, n$ فرض کنید x_i به i امین عنصری که در جدول درج شده است اشاره می‌کند، و فرض کنید $k_i = key[x_i]$ باشد. برای کلیدهای k_i و k_j متغیر تصادفی شاخص $X_{ij} = I\{h(k_i) = h(k_j)\}$ را تعریف می‌کنیم. با فرض درهم سازی یکنواخت ساده، داریم $Pr\{h(k_i) = h(k_j)\} = 1/m$ (بنا به خطی بودن انتظار (امید ریاضی)) و بنابراین تعداد عناصری که انتظار می‌رود در یک جستجوی موفق بررسی شوند برابر است با

$$
\begin{aligned}
& \mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)\right] \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}]\right) \quad \text{(بنا به خطی بودن انتظار(امیدریاضی))}
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)\\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)\\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)\\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)\\
&= 1+\frac{n-1}{2m}\\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\,.
\end{aligned}
$$

بنابراین زمان کل مورد نیاز برای جستجوی موفق (که شامل زمان محاسبه تابع درهم سازی نیز می‌باشد) برابر $\Theta(2+\alpha/2-\alpha/2n)=\Theta(1+\alpha)$ می‌باشد. ■

مفهوم تحلیل چیست؟ اگر تعداد مکان‌های جدول درهم سازی حداقل متناسب با تعداد عناصر جدول باشد، داریم $n = O(m)$ و در نتیجه $\alpha = n/m = O(m)/m = O(1)$. بنابراین، جستجو در حالت میانگین زمان ثابتی را صرف می‌کند. از آنجا که درج، زمان $O(1)$ را در بدترین حالت صرف می‌کند و حذف هم در بدترین حالت، هنگامیکه لیستها دوپیوندی هستند زمان $O(1)$ را صرف می‌کند، تمام اعمال لغت نامه‌ای می‌توانند در حالت میانگین در زمان $O(1)$ انجام شوند.

تمرین‌ها

۱۱.۲-۱ فرض کنید که از تابع درهم سازی h برای درهم سازی n کلید متفاوت به درون آرایه T با طول m استفاده می‌کنیم. با فرض درهم‌سازی یکنواخت ساده، تعداد برخوردهایی که انتظار می‌رود چقدر است؟

بطور دقیقتر، تعداد اعضاء مجموعه $\{\{k,l\}: k \neq l$ و $h(k) = h(l)\}$ چیست؟

۱۱.۲-۲ درج کلیدهای $5, 28, 19, 15, 20, 33, 12, 17, 10$ در یک جدول درهم سازی با برخوردهایی که توسط زنجیر سازی حل شده‌اند را شرح دهید. فرض کنید جدول 9 مکان داشته باشد و تابع درهم‌سازی $h(k) = k \bmod 9$ باشد.

۱۱.۲-۳ عقیده پروفسور *Marley* اینست که اگر طرح زنجیره سازی را چنان اصلاح کنیم که هر لیست به شکل مرتب نگهداری شود کارآیی بسیار افزایش می‌یابد. این اصلاح پروفسور، چگونه به زمان اجرای جستجوهای موفق، جستجوهای ناموفق، درج و حذف تأثیر می‌گذارد؟

۱۱.۲-۴ چگونه حافظه مورد نیاز عناصر می‌تواند با اتصال تمام مکانهایی که استفاده نمی‌شوند به یک لیست آزاد درون خود جدول درهم سازی تخصیص یابد و آزاد شود؟ فرض کنید که یک مکان می‌تواند یک پرچم و یک عنصر به اضافه یک یا دو اشاره گر را ذخیره کند. همه اعمال لغت نامه‌ای و لیست آزاد، باید در زمان مورد انتظار $O(1)$ اجرا شوند. آیا لیست آزاد لازم است دو پیوندی باشد، یا یک لیست تک پیوندی کافیست؟

۱۱.۲-۵ نشان دهید که اگر $|U| > nm$، یک زیر مجموعه از U با اندازه n وجود دارد که تشکیل شده است از کلیدهایی که همگی به یک مکان درهم سازی شده‌اند، بطوریکه زمان بدترین جستجو برای درهم سازیِ همراه با زنجیره سازی، $\Theta(n)$ است.

۱۱.۳ توابع درهم‌سازی

در این بخش، برخی از جنبه‌های طراحی یک تابع درهم سازی خوب را مورد بحث قرار می‌دهیم و سپس سه طرح برای تولید این توابع ارائه می‌دهیم. دو طرح از این طرحها که درهم سازی با استفاده از تقسیم و درهم سازی با استفاده از ضرب هستند، ذاتاً روشهای مکاشفه‌ای هستند، در حالیکه سومین طرح یعنی درهم‌سازی کلی، برای داشتن کارآیی مناسب که قابل اثبات است، از تصادفی سازی استفاده می‌کند.

تولید یک تابع درهم سازی خوب چگونه صورت می‌گیرد؟

یک تابع درهم سازی مناسب (تقریباً) فرض درهم سازی یکنواخت ساده را ارضا می‌کند: هر کلید بطور مساوی احتمال دارد که به هر یک از m مکان درهم سازی شود، بدون توجه به اینکه کلیدها به کجا درهم سازی شده‌اند. متأسفانه، نوعاً این امکان وجود ندارد که این شرط را چک کنیم زیرا دانستن توزیع احتمالی که بر طبق آن کلیدها انتخاب می‌شوند به ندرت رخ می‌دهد و کلیدها ممکن نیست بصورت مستقل انتخاب شوند.

گاهی اوقات نحوه توزیع کردن را می‌دانیم. به عنوان مثال، اگر بدانیم کلیدها اعداد حقیقی تصادفی k هستند و بطور مستقل و یکنواخت در بازه $0 \le k < 1$ توزیع شده‌اند، تابع درهم سازی

$$h(k) = \lfloor km \rfloor$$

در شرط درهم سازی یکنواخت ساده صدق می‌کند.

در عمل، تکنیک‌های مکاشفه‌ای اغلب می‌توانند برای ایجاد یک تابع درهم سازی که خوب عمل می‌کند به کار روند. اطلاعات کیفی در مورد توزیع کلیدها ممکن است در این فرآیند طراحی مفید باشند. به عنوان مثال، یک جدول نماد کامپایلر در نظر بگیرید، که در آن کلیدها رشته‌های کارکتری هستند که شناسه‌ها در یک برنامه را نشان می‌دهند. نمادهایی که بسیار به هم نزدیکند مانند pt و pts اغلب در یک

برنامه یکسان رخ می‌دهند. یک تابع درهم سازی خوب، شانس درهم سازی شدن این انواع به یک مکان یکسان را مینیمم می‌کند.

یک راه مناسب اینست که مقادیر درهم سازی را از راهی بدست آوریم که انتظار می‌رود مستقل از داده‌هایی است که امکان دارد وجود داشته باشند. به عنوان مثال، روش تقسیم (در بخش ۱۱.۳.۱ بحث شد) مقدار درهم سازی را به عنوان باقیمانده تقسیم کلید به یک عدد اول معین محاسبه می‌کند. این روش غالباً با این فرض که عدد اول انتخاب شده به هیچ الگویی در توزیع کلیدها مربوط نمی‌شود، نتایج مناسبی را حاصل می‌کند.

در نهایت، به کاربردهایی می‌پردازیم که ممکن است به ویژگی‌های قوی‌تری نسبت به آنچه توسط درهم سازی یکنواخت ساده ایجاد شده است نیاز داشته باشند. به عنوان مثال، ممکن است بخواهیم از کلیدهایی که تا اندازه‌ای به هم نزدیکند برای بدست آوردن مقادیر درهم سازی شده‌ای که دور از هم قرار گرفته‌اند استفاده کنیم. (این ویژگی خصوصاً وقتی از جستجوی خطی که در بخش ۱۱.۴ تعریف شده است استفاده می‌کنیم مطلوب است) درهم سازی جامع، که در بخش ۱۱.۳.۳ تشریح شده است اغلب این ویژگی‌های مطلوب را مهیا می‌کند.

تفسیر کلیدها به عنوان اعداد طبیعی

اکثر توابع درهم سازی فرض می‌کنند که مجموعه مرجع کلیدها مجموعه $N = \{0,1,2,...\}$ از اعداد طبیعی است. بنابراین اگر کلیدها اعداد طبیعی نباشند، راهی برای تفسیر آنها به عنوان اعداد طبیعی وجود دارد. به عنوان مثال، یک رشته کاراکتری می‌تواند به شکل یک عدد صحیح که بصورت نماد مبنایی مناسبی بیان شده است تفسیر گردد. بنابراین شناسه pt ممکن است به شکل جفت عدد دهدهی $(112,116)$ تفسیر شود، زیرا در مجموعه کاراکترهای ASCII، $p = 112$ و $t = 116$ است؛ آنگاه به عنوان یک عدد صحیح مبنای 128، pt برابر می‌شود با $(112 . 116) + 116 = 14425$. معمولاً در عمل بدست آوردن چنین روشهایی برای تفسیر هر کلید به شکل یک عدد طبیعی (که ممکن است بسیار بزرگ باشد) کار آسانی است. در ادامه، فرض می‌کنیم که کلیدها اعدادی طبیعی هستند.

۱۱.۳.۱ روش تقسیم

در روش تقسیم[1] برای ایجاد تابع درهم سازی، کلید k را با بدست آوردن باقیمانده تقسیم k بر m به یکی از m مکان نگاشت می‌کنیم. بعبارت دیگر تابع درهم سازی $h(k) = k \bmod m$ است. به عنوان مثال، اگر اندازه جدول درهم سازی $m = 12$ باشد و کلید $k = 100$، $h(x) = 4$ می‌شود. از آنجا که فقط یک عمل تقسیم لازم است، درهم سازی با استفاده از تقسیم بسیار سریع انجام می‌گیرد.

1. division method

وقتی از روش تقسیم استفاده می‌کنیم، معمولاً از برخی مقادیر m اجتناب می‌کنیم. مثلاً m نباید توانی از 2 باشد، زیرا اگر $m = 2^p$ باشد، آنگاه $h(k)$ فقط کم ارزش‌ترین p بیت k می‌شود. بهتر است که تابع درهم سازی را به تمام بیت‌های کلید وابسته کنیم، مگر آنکه تمام الگوهای کم ارزش p بیت، احتمال مساوی داشته باشند. تمرین ۳-۱۱.۳ از شما می‌خواهد تا نشان دهید انتخاب $m = 2^p-1$ وقتی k یک رشته کاراکتری است که در مبنای 2^p تفسیر شده است، ممکن است انتخاب ضعیفی باشد، زیرا ترتیب‌های مختلف کاراکترهای k مقدار درهم سازی آن را تغییر نمی‌دهد.

عدد اولی که به یکی از توانهای صحیح 2 بسیار نزدیک نیست، اغلب انتخاب مناسبی برای m است. به عنوان مثال، فرض کنید می‌خواهیم برای نگهداری $n = 2000$ رشته کاراکتری، که هر کاراکتر 8 بیت دارد، یک جدول درهم سازی تخصیص دهیم که در آن برخوردها توسط زنجیره سازی حل شده‌اند. به بررسی بطور متوسط 3 عنصر در یک جستجوی ناموفق توجّهی نداریم، بنابراین یک جدول درهم‌سازی با اندازه $m = 701$ را تخصیص می‌دهیم. عدد 701 انتخاب می‌شود زیرا یک عدد اول نزدیک به $2000/3$ است و نزدیک به هیچ توانی از 2 نیست. با در نظر گرفتن هر کلید x به عنوان یک عدد طبیعی، تابع درهم سازی ما چنین خواهد بود

$$h(k) = k \bmod 701$$

۱۱.۳.۲ روش ضرب

روش ضرب[1] برای ایجاد توابع درهم سازی، در دو مرحله عمل می‌کند. اول، کلید k را در یک عدد ثابت A در بازه $0 < A < 1$ ضرب می‌کنیم و بخش کسری kA را خارج می‌کنیم. سپس عدد را در m ضرب کرده و جزء صحیح نتیجه را در نظر می‌گیریم. بطور خلاصه، تابع چنین درهم سازی می‌شود

$$h(k) = \lfloor m\ (kA \bmod 1) \rfloor$$

که در آن $kA \bmod 1$ به معنای قسمت کسری kA است، بعبارت دیگر $kA - \lfloor kA \rfloor$.

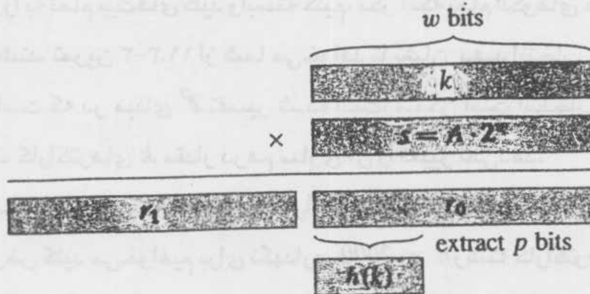
یک مزیت روش ضرب اینست که مقدار m بحرانی نیست. ما نوعاً آن را طوری انتخاب می‌کنیم که توانی از 2 باشد. (برای عدد صحیح p ،$m = 2^p$). زیرا با این کار می‌توانیم بسادگی این عمل را در اغلب کامپیوترها به شکل زیر پیاده سازی کنیم. فرض کنید اندازه کلمه ماشین، w بیت است و k در یک تک کلمه قرار می‌گیرد. A را طوری محدود می‌کنیم که کسری به شکل $s/2^w$ باشد، که در آن s یک عدد صحیح دربازه $0 < s < 2^w$ است. با ارجاع به شکل ۱۱.۴، اول k را در عدد w بیتی $s = A.2^w$ ضرب می‌کنیم. نتیجه یک عدد $2w$ بیتی با مقدار $r_1 2^w + r_0$ است، که r کلمه با ارزش‌تر و r_0 کلمه کم ارزش حاصلضرب است. مقدار درهم سازی p بیتی مطلوب، از پر اهمیت‌ترین p بیت r_0 تشکیل می‌گردد.

اگر چه این روش با هر مقدار ثابت A کار می‌کند، با برخی مقادیر بهتر از بقیه مقادیر عمل می‌کند.

---

1. *multiplication method*

انتخاب بهینه به خصوصیات داده‌ای که درهم سازی می‌شود بستگی دارد. *Knuth* پشتیبانی می‌کند که



شکل ۱۱.۴ روش ضرب عمل درهم سازی. نمایش w بیتی کلید k در مقدار w بیتی $s = A.2^w$ ضرب شده است. p بیت پرارزش‌تر از نیمه w بیتی کم ارزش‌تر حاصلضرب، مقدار درهم سازی مطلوب $h(k)$ را شکل می‌دهد.

$$A \approx (\sqrt{5}-1)/2 = 0.6180339887\ldots \qquad (11.2)$$

احتمال دارد به طور معقول خوب کار کند.

به عنوان مثال، فرض کنید داریم $k = 123456$، $p = 14$، $m = 2^{14}=16384$ و $w=32$. بر طبق پیشنهاد *knuth* A را کسری به شکل $\frac{s}{2^{32}}$ انتخاب می‌کنیم که نزدیک‌ترین عدد به $(\sqrt{5}-1)/2$ است، بنابراین $A = 2654435769/2^{32}$ پس $k\ .\ s=327706022297664=(76300\ .\ 2^{32})+17612864$ و $r_1=76300$ و $r_0=17612864$ بیت با ارزش‌تر r_0 مقدار $h(k) = 67$ را به ما می‌دهند.

۱۱.۳.۳* درهم‌سازی جامع

اگر فردی که قصدش آزار و اذیت ما است بخواهد کلیدهایی را برای درهم سازی شدن توسط یک تابع ثابت انتخاب کند، می‌تواند n کلیدی را انتخاب کند که همگی به یک مکان درهم سازی می‌شوند، که زمان بازیابی میانگین $\Theta(n)$ را حاصل می‌کند. هر تابعْ درهم سازی ثابت در برابر این حالت‌های مشکل‌ساز، آسیب‌پذیر است؛ تنها راه مؤثر برای بهبود این وضعیت آنست که تابع درهم سازی را بطور تصادفی به طریقی انتخاب کنیم که مستقل از کلیدهایی باشد که واقعاً می‌خواهند ذخیره شوند. این روش که، *درهم‌سازی جامع*[1] نامیده می‌شود، بدون توجه به اینکه چه کلیدهایی انتخاب شده‌اند می‌تواند در حالت میانگین کارآیی بالایی داشته باشد.

---

1. *universal hashing*

ایده اصلی درهم سازی جامع، انتخاب تابع درهم سازی در ابتدای اجرا است که بصورت تصادفی از روی کلاس توابع که به دقت طراحی شده است صورت می‌گیرد. همانند عملی که در مرتب‌سازی سریع صورت می‌گیرد، تصادفی سازی ضمانت می‌کند که هیچ داده ورودی منفردی همیشه بدترین حالت را ایجاد نمی‌کند. بدلیل تصادفی سازی، الگوریتم می‌تواند در هر اجرا حتی برای ورودی یکسان، رفتار متفاوتی داشته باشد، که برای هر ورودی در حالت میانگین کارآیی مناسبی را تضمین می‌کند. با برگشتن به مثال جدول نماد کامپایلر، متوجه می‌شویم که انتخابی که برنامه نویس از شناسه‌ها دارد در اینجا نمی‌تواند موجب کارآیی ضعیف عمل درهم سازی شود. کارآیی ضعیف فقط زمانی اتفاق می‌افتد که کامپایلر، یک تابع درهم سازی تصادفی انتخاب می‌کند که باعث می‌شود مجموعه شناسه‌ها بصورت بدی درهم سازی شوند، ولی احتمال پیش آمدن این وضعیت کم است و برای هر مجموعه دیگر از شناسه‌ها با همین اندازه، یکسان است.

فرض کنید $\mathcal{H}$ یک مجموعه متناهی از توابع درهم سازی باشد که مجموعه مرجع U از کلیدها را به بازه $\{0,1, ..., m-1\}$ نگاشت می‌کند. چنین مجموعه‌ای جامع نامیده می‌شود، اگر برای هر جفت متفاوت از کلیدهای $l \in U$ و k تعداد توابع درهم سازی $h \in \mathcal{H}$ که برای آنها $h(k) = h(l)$ حداکثر برابر $|\mathcal{H}|/m$ باشد. به عبارت دیگر، با استفاده از یک تابع درهم سازی که بطور تصادفی از $\mathcal{H}$ انتخاب شده است، شانس یک برخورد بین کلیدهای متمایز l و k بیشتر از شانس $1/m$ برخوردی که اگر $h(k)$ و $h(l)$ بطور تصادفی و مستقل از مجموعه $\{0, 1, . . . ,m-1\}$ انتخاب شوند صورت می‌گیرد، نیست.

قضیه زیر نشان می‌دهد که یک کلاس جامع از توابع درهم سازی، رفتار در حالت میانگین مناسبی را نتیجه خواهد. به خاطر آورید که n_i به طول لیست $T[i]$ اشاره دارد.

قضیه ۱۱.۳

فرض کنید که تابع درهم سازی h از یک مجموعه جامع از توابع درهم سازی انتخاب می‌شود و با عمل زنجیره سازی برای حل برخوردها برای درهم سازی n کلید به جدول T با اندازه m، استفاده می‌گردد. اگر کلید k در جدول نباشد، طول مورد انتظار $E[n_{h(k)}]$ از لیستی که کلید k به آن درهم سازی می‌شود حداکثر α است. اگر کلید k در جدول باشد، طول مورد انتظار $E[n_{h(k)}]$ لیستی که شامل k است، حداکثر $1+\alpha$ می‌باشد.

اثبات توجه داریم که انتظارات، فراتر از انتخاب تابع درهم سازی هستند و به هیچ فرضی در مورد توزیع کلیدها بستگی ندارند. برای هر جفت k و l از کلیدهای متفاوت، متغیر تصادفی شاخص $X_{kl} = I\{h(k) = h(l)\}$ را تعریف می‌کنیم. زیرا بنا به تعریف، یک جفت از کلیدها با احتمال حداکثر $1/m$ با هم برخورد می‌کنند، داریم $Pr\{h(k) = h(l)\} \leq 1/m$ و بنابراین لم ۵.۱ بیان می‌کند که $E[X_{kl}] \leq 1/m$

سپس برای هر کلید k مقدار تصادفی Y_k را تعریف می‌کنیم که برابر است با تعداد کلیدهای غیر از k که به همان مکان که x درهم سازی می‌شود درهم سازی می‌شوند، بنابراین

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl} \ .$$

لذا داریم

$$\begin{aligned} \mathrm{E}[Y_k] &= \mathrm{E}\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] \\ &= \sum_{\substack{l \in T \\ l \neq k}} \mathrm{E}[X_{kl}] \\ &\leq \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m} \ . \end{aligned}$$

(بنا به خطی بودن انتظار(امیدریاضی))

باقی اثبات به اینکه آیا کلید k در جدول T هست یا خیر بستگی دارد.

- اگر $k \notin T$ آنگاه $n_{h(k)} = Y_k$ و $|\{l k : l \in T$ و $l \neq k\}| = n$. بنابراین $E[n_{h(k)}] = E[Y_k] \leq n/m = \alpha$
- اگر $k \in T$ آنگاه به دلیل اینکه k در لیست $T[h(k)]$ قرار دارد و شمارش Y_k کلید k را شامل نمی‌شود داریم: $n_{h(k)} = Y_k + 1$ و $|\{l : l \in T$ و $l \neq k\}| = n - 1$. بنابراین

$$E[n_{h(k)}] = E[Y_k] + 1 \leq (n-1) / m + 1 = 1 + \alpha - 1 / m < 1 + \alpha \quad \blacksquare$$

قضیه فرعی زیر عنوان می‌کند که درهم سازی جامع مزیتهای مطلوبی دارد: امکان ندارد که یک توالی از اعمالی انتخاب شوند که بدترین حالت زمان اجرا را حاصل می‌کنند. با تصادفی سازی هوشمندانه انتخاب تابع درهم سازی در زمان اجرا، تضمین می‌کنیم که هر توالی از اعمال می‌تواند با زمان اجرای مورد انتظار مناسبی مدیریت شود.

قضیه فرعی ۱۱.۴

با استفاده از درهم سازی جامع و حل برخوردها با کمک زنجیره سازی در یک جدول که m مکان دارد، هر توالی از n عمل $INSERT$، $SEARCH$ و $DELETE$ که شامل $O(m)$ عمل $INSERT$ می‌باشند، زمان مورد انتظار $\Theta(n)$ را صرف می‌کند.

اثبات از آنجا که تعداد درجها $O(m)$ است، داریم $n = O(m)$ و بنابراین $\alpha = O(1)$. اعمال $INSERT$ و $DELETE$ زمان ثابتی را صرف می‌کنند و بنا به قضیه ۱۱.۳ زمان مورد انتظار برای هر عمل $SEARCH$ برابر $O(1)$ است. بنابراین با توجه به خطی بودن انتظار، زمان مورد انتظار برای کل توالی اعمال برابر $O(n)$ است. ■

طراحی یک کلاس جامع از توابع درهم سازی

طراحی یک کلاس جامع از توابع درهم سازی بسیار آسان است و نظریه اعداد در اثبات آن به ما کمک می‌کند. اگر با نظریه اعداد آشنا نمی‌باشید بهتر است ابتدا آن را مورد بررسی قرار دهید.

با انتخاب عدد اول p به اندازه کافی بزرگ به نحوی که هر کلید ممکن k در بازه 0 تا $p\text{-}1$ قرار می‌گیرد آغاز می‌کنیم. فرض کنید Z_p به مجموعه $\{0,1, \dots ,p\text{-}1\}$ و Z_p به مجموعه $\{1, 2, \dots ,p\text{-}1\}$ دلالت داشته باشد. از آنجا که p اول است، می‌توانیم همنهشتی به پیمانه p را حل کنیم. از آنجا که فرض می‌کنیم اندازه مجموعه مرجع کلیدها بیشتر از تعداد مکان‌ها در جدول درهم سازی است، داریم $p>m$ با استفاده از یک تبدیل خطی که به دنبال آن به ترتیب تبدیلات به پیمانه‌های p و m صورت می‌گیرند تابع درهم سازی $h_{a,b}$ را برای هر $a \in Z^*_p$ و هر $b \in Z_p$ تعریف می‌کنیم:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m \qquad (۱۱.۳)$$

به عنوان مثال با $p = 17$ و $m = 6$ داریم $h_{3,4}(8) = 5$ مجموعه چنین توابع درهم سازی به شکل زیر است

$$\mathcal{H}_{p,m} = \{h_{a,b} : a \in \mathbf{Z}_p^* \text{ and } b \in \mathbf{Z}_p\} \ . \qquad (۱۱.۴)$$

هر تابع درهم سازی $h_{a,b}$ Z_p را به Z_m نگاشت می‌کند. این کلاس توابع درهم سازی ویژگی جالبی دارد که اندازه m بازه خروجی اختیاری است - و لازم نیست که اول باشد - ویژگی که در بخش ۱۱.۵ از آن استفاده خواهیم کرد. از آنجا که $p\text{-}1$ انتخاب برای a و p انتخاب برای b داریم، $p(p\text{-}1)$ تابع درهم‌سازی در $\mathcal{H}_{p,m}$ وجود دارد.

قضیه ۱۱.۵

کلاس $\mathcal{H}_{p,m}$ از توابع درهم سازی که با استفاده از تساویهای (۱۱.۳) و (۱۱.۴) تعریف می‌شود جامع است.

اثبات دو کلید متفاوت k و l از Z_p را در نظر بگیرید، بطوریکه $k \neq l$ برای تابع درهم‌سازی داده شده $h_{a,b}$ فرض می‌کنیم

$r = (ak + b) \bmod p$

$s = (al + b) \bmod p$

در ابتدا توجه دارید که $r \neq s$ چرا؟ مشاهده می‌شود که

$r - s \equiv a(k - l) \bmod p$

در نتیجه $r \neq s$ زیرا p اول است و هم a و هم $(k\text{-}l)$ به پیمانه p غیر صفر هستند، و بنابراین حاصلضرب آنها به پیمانه p نیز باید غیر صفر باشد. بنابراین در طی محاسبه هر $h_{a,b}$ در $\mathcal{H}_{p,m}$، ورودی‌های متمایز c,k به دو مقدار متفاوت r و s به پیمانه p نگاشت می‌شوند؛ برخوردی در "باقیمانده به p" وجود ندارد، به

علاوه، هر یک از $p(p-1)$ انتخاب ممکن برای جفت (a,b) با $a \neq 0$ به جفت (r,s) متفاوت منجر می‌شود، زیرا می‌توانیم با r و s داده شده، a,b را بدست آوریم:

$$a = ((r-s) ((k-l)^{-1}) \bmod p)) \bmod p$$

$$b = (r-ak) \bmod p$$

که در آن $((k-l)^{-1} \bmod p)$ به معکوس $k-l$ به پیمانه p دلالت می‌کند. از آنجا که فقط $p(p-1)$ جفت (r,s) ممکن با شرط $r \neq s$ وجود دارد یک تناظر یک‌به‌یک بین جفت‌های (a,b) با شرط $a \neq 0$ و جفت‌های (r,s) با شرط $r \neq s$ وجود دارد. بنابراین برای هر جفت ورودی داده شده l, k، اگر (a,b) را بطور یکنواخت و تصادفی از $Z_p^* \times Z_p$ انتخاب کنیم، جفت بدست آمده (r,s) به طور مساوی احتمال دارد که هر جفتی از مقادیر متفاوت به پیمانه p باشد.

بنابراین هنگامیه r و s بطور تصادفی به عنوان مقادیر متمایزی به پیمانه p انتخاب شده‌اند، احتمال برخورد دو کلید متفاوت l و k برابر است با احتمال اینکه $r \equiv s$ (به پیمانه m). برای یک مقدار داده شده r که جزء $p-1$ مقدار باقیمانده ممکن برای s است، تعداد مقادیر s به طوریکه $s \neq r$ و $s \equiv r$ (به پیمانه m) حداکثر برابر است با

$$\lceil p/m \rceil -1 \leq ((p+m-1)/m)-1 \quad \text{((بنابه نامساوی (۳.۷)}$$

$$= (p - 1) / m$$

با تبدیل به پیمانه m احتمال برخورد s با r حداکثر برابر است با $((p-1)/m)/(p-1) = 1/m$.

بنابراین برای هر جفت مقادیر متمایز $k , l \in Z_p$

$$Pr\{h_{a,b}(k) = h_{a,b}(l)\} \leq 1/m$$

که نتیجه می‌دهد $\mathcal{H}_{p,m}$ در واقع جامع است.

تمرین‌ها

۱-۱۱.۳ فرض کنید می‌خواهیم یک لیست پیوندی بطول n را جستجو کنیم، بطوری که عنصر شامل کلید k به همراه مقدار درهم سازی $h(k)$ است. هر کلید یک رشته کاراکتری بلند است. در زمان جستجوی یک عنصر در لیست با یک کلید داده شده، چگونه می‌توانیم از مقادیر *hash* استفاده کنیم؟

۲-۱۱.۳ فرض کنید یک رشته از r کاراکتر به m مکان با استفاده از یک عدد در مبنای 128 و سپس استفاده از روش تقسیم، درهم سازی شود. عدد m براحتی به شکل یک کلمه کامپیوتری 32 بیتی نمایش داده می‌شود، ولی رشته r کاراکتری که به عنوان یک عدد مبنای 128 با آن برخورد می‌شود، کلمات زیادی را به کار می‌برد. چگونه می‌توانیم روش تقسیم را برای محاسبه مقدار درهم سازی رشته کاراکتری بدون استفاده از بیش از تعداد ثابتی از کلمات حافظه خارج از خود رشته، بکار بریم؟

۳-۱۱.۳ شکلی از روش تقسیم که در آن $h(k) = k \bmod m$ را در نظر بگیرید، که $m = 2^p-1$ و k

یک رشته کاراکتری است که در مبنای 2^p تفسیر شده است. نشان دهید اگر رشته x بتواند از رشته y با استفاده از جابجایی کارکترهایش بدست آید، آنگاه x و y به یک مقدار یکسان درهم سازی می‌شوند. مثالی از یک کاربرد که در آن این ویژگی در تابع درهم سازی نامطلوبست را ارائه دهید.

۱۱.۳-۴ یک جدول درهم سازی با اندازه $m = 100$ و یک تابع درهم سازی متناظر $h(k) = \lfloor m(kA \bmod 1) \rfloor$ برای $A = (\sqrt{5}-1)/2$ را در نظر بگیرید. موقعیت هایی که کلیدهای 61، 62، 63، 54 و 65 به آن نگاشت می‌شوند را محاسبه نمائید.

*۱۱.۳-۵ خانواده $\mathcal{H}$ از توابع درهم سازی را از روی مجموعه متناهی U به مجموعه متناهی B تعریف کنید که ε-*جامع*[1] باشد اگر برای همه جفتهای متمایز عناصر k و l در U داشته باشیم

$$Pr\{h(k) = h(l)\} \leq \varepsilon$$

که احتمال، از تابع درهم سازی h که به شکلی تصادفی از خانواده $\mathcal{H}$ انتخاب شده است بدست می‌آید. نشان دهید که در یک خانواده ε- جامع از توابع درهم سازی، باید داشته باشیم

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}.$$

*۱۱.۳-۶ فرض کنید U مجموعه n- گانه بدست آمده از مقادیر Z_p باشد و $B = Z_p$ که p اول است. تابع درهم سازی $h_b : U \rightarrow B$ را برای $b \in Z_p$ روی ورودی n-گانه $<a_0, a_1, \ldots, a_{n-1}>$ از U را به شکل زیر تعریف کنید

$$h_b(\langle a_0, a_1, \ldots, a_{n-1} \rangle) = \sum_{j=0}^{n-1} a_j b^j$$

و فرض کنید $\mathcal{H} = \{h_b : b \in Z_p\}$ ثابت کنید که $\mathcal{H}$ بر طبق تعریف ε- جامع در تمرین ۱۱.۳-۵، $((n-1)/p)$- جامع است.

۱۱.۴ آدرس دهی باز

در *آدرس دهی باز*[2]، همه عناصر در خود جدول درهم سازی ذخیره می‌شوند. بعبارت دیگر هر ورودی جدول یا یک عنصر از مجموعه پویا یا NIL را در بر دارد. در هنگام جستجوی یک عنصر، بطور اصولی مکان‌های جدول را تا زمانیکه عنصر مطلوب پیدا بشود یا روشن شود که عنصر در جدول نیست، بررسی می‌کنیم. هیچ لیست و هیچ عنصری بر خلاف شکل زنجیره‌ای در بیرون از جدول ذخیره نشده‌اند. بنابراین در آدرس دهی باز، جدول درهم سازی می‌تواند پر شود که در آن صورت هیچ درج دیگری نمی‌تواند صورت بگیرد. ضریب بار α هیچگاه بیشتر از 1 نمی‌شود.

---

1. ε-universal
2. open addressing

البته می‌توانستیم لیستهای پیوندی را برابر زنجیره سازی در جدول درهم سازی، در مکان‌های دیگر جدول درهم سازی که استفاده نشده‌اند ذخیره کنیم (تمرین ۴-۱۱.۲ را ملاحظه کنید)، ولی مزیت آدرس دهی باز این است که از اشاره گرها اجتناب می‌کند. به جای اشاره گرهی‌ی که به دنبال هم قرار دارند، توالی از مکانها را برای امتحان شدن محاسبه می‌کنیم. برای یک مقدار حافظه، حافظه اضافه که بعلت عدم استفاده از ذخیره سازی اشاره گرها آزاد شده است، جدول درهم سازی ایجاد می‌کند که شامل تعداد بیشتری از مکانها است و بطور بالقوه برخوردهای کمتر و بازیابی سریعتر، حاصل می‌کند.

برای انجام عمل درج با استفاده از آدرس دهی باز، بطور پیاپی جدول درهم سازی را تا زمانی که مکان خالی پیدا کنیم تا در آن کلیدها را قرار دهیم، امتحان یا *بررسی*[1] می‌کنیم. به جای ثابت بودن در ترتیب $0, 1, \ldots, m-1$ (که زمان جستجوی $\Theta(n)$ را نیاز دارد)، توالی موقعیت هایی که بررسی می‌شوند، به کلیدی که درج می‌شود وابسته است. برای تعیین اینکه کدام مکانها باید بررسی شوند، تابع درهم سازی را چنان توسعه می‌دهیم که تعداد بررسیها (با شروع از 0) را به عنوان ورودی دوم شامل شود. بنابراین تابع درهم سازی به شکل زیر می‌شود

$$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$$

با استفاده از آدرس دهی باز، نیاز داریم که برای هر کلید k *توالی بررسی*[2]

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

یک جایگشت از $\langle 0, 1, \ldots, m-1 \rangle$ باشد، بطوریکه هر موقعیت جدول درهم سازی هنگامیکه جدول پر می‌شود، در نهایت به عنوان یک مکان برای کلید جدید در نظر گرفته می‌شود. در شبه کد زیر فرض می‌کنیم که عناصر در جدول درهم سازی T کلیدهایی هستند که هیچگونه داده فرعی ندارند؛ کلید k شاخص عنصری است که کلید k را شامل می‌شود. هر مکان یا یک کلید یا مقدار NIL (اگر مکان خالی باشد) را در بردارد.

```
HASH-INSERT(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3            if T[j] = NIL
4               then T[j] ← k
5                      return j
6               else  i ← i + 1
7     until i = m
8  error "hash table overflow"
```

الگوریتم برای جستجوی کلید k همان توالی از مکانها را بررسی می‌کند که الگوریتم درج وقتی کلید

---

1. probe   2. probe sequence

k درج می‌شد امتحان کرده است. بنابراین جستجو (بصورت ناموفق) زمانی می‌تواند پایان یابد که یک مکان خالی را پیدا کند، زیرا k باید در این مکان درج می‌شده است نه در مکان بعدی توالی بررسی. (این بحث فرض می‌کند که کلیدها از جدول درهم سازی حذف نمی‌شوند.) روال *HASH-SEARCH* به عنوان ورودی جدول درهم سازی T و کلید k را می‌گیرد و اگر مکان j پیدا شد که کلید k را دارد j و اگر کلید k در جدول T موجود نباشد *NIL* را بر می‌گرداند.

```
HASH-SEARCH(T, k)
1  i ← 0
2  repeat j ← h(k, i)
3         if T[j] = k
4            then return j
5         i ← i + 1
6     until T[j] = NIL or i = m
7  return NIL
```

حذف از یک جدول درهم سازی آدرس باز مشکل است. وقتی کلیدی را از مکان i حذف می‌کنیم، نمی‌توانیم بسادگی آن مکان را با ذخیره *NIL* در آن به معنای خالی بودن، علامت گذاری کنیم. انجام این کار ممکن است باعث شود بازیابی کلید k که در زمان درجش مکان i را بررسی کرده و آن را اشغال شده یافته بودیم، غیر ممکن شود. یک راه حل اینست که مکان را با ذخیره کردن مقدار خاص *DELETED* به جای *NIL* علامت گذاری کنیم. بنابراین می‌توانیم روال *HASH-INSERT* را اصلاح کنیم بنحوی که با چنین مکانی همانند یک مکان خالی برخورد کند و بنابراین کلید جدید می‌تواند درج شود. هیچ تغییری در *HASH-SEARCH* لازم نیست، زیرا هنگام جستجو از روی مقادیر *DELETED* عبور می‌کند. با این حال وقتی از مقدار خاص *DELETED* استفاده می‌کنیم، زمانهای جستجوی دیگر وابسته به ضریب بار α نیستند، و به همین دلیل وقتی کلیدها باید حذف شوند، زنجیره‌سازی بطور معمول‌تری به عنوان تکنیک حل برخوردها انتخاب می‌شود.

در تحلیل خود، فرض ***درهم سازی یکنواخت***[1] را در نظر داریم: فرض می‌کنیم که هر کلید بطور مساوی احتمال دارد که هر یک از $m!$ جایگشت $<0,1, \ldots ,m-1>$ را به عنوان توالی بررسی خود داشته باشد. درهم سازی یکنواخت، مفهوم درهم سازی یکنواخت ساده را تعمیم می‌دهد، که درهم‌سازی یکنواخت ساده قبلاً برای شرایطی که در آن تابع درهم سازی، نه تنها یک عدد منفرد بلکه یک توالی بررسی کامل را تولید می‌کند تعریف شده است. اما درهم سازی یکنواخت صحیح پیاده‌سازی مشکلی دارد، و در عمل تقریبهای مناسبی (مانند درهم سازی مضاعف که در زیر تعریف شده است)

---

1. uniform hashing

استفاده می‌شوند.

معمولاً سه تکنیک برای محاسبه توالی‌های بررسی مورد نیاز آدرس دهی باز استفاده می‌شوند: بررسی خطی، بررسی درجه دوم و درهم سازی مضاعف. این تکنیک‌ها همگی تضمین می‌کنند که برای هر کلید k $<h(k,0),\ h(k,1),\ ...,\ h(k,m-1)>$ یک جایگشت از $<0,1,\ ...\ ,\ m-1>$ است. با این وجود هیچکدام از این تکنیک‌ها فرض درهم سازی یکنواخت را بطور کامل برآورده نمی‌کنند، زیرا هیچکدام از آنها قادر نیستند که بیش از m^2 توالی بررسی (به جای $m!$ که درهم سازی یکنواخت به آن نیاز دارد) تولید کنند. درهم سازی مضاعف بیشترین تعداد توالی‌های بررسی که انتظار می‌رود را دارد و به نظر می‌رسد که بهترین نتایج را حاصل کند.

بررسی خطی

یک تابع درهم سازی معمول $h':\ U \rightarrow \{0,1,\ ...\ ,m-1\}$ داده شده است، که به آن به عنوان یک ***تابع درهم‌سازی کمکی***[1] اشاره می‌کنیم، روش ***بررسی خطی***[2] برای $i = 0,\ 1,\ ...,\ m-1$، از تابع درهم سازی زیر استفاده می‌کند

$$h(k,i) = (h'(k)+i)\ mod\ m$$

عدد k داده شده است، اولین مکانی که بررسی می‌شود $T[h'(k)]$ است، یعنی مکانی که توسط تابع درهم‌سازی کمکی بدست آمده است. سپس مکان $T[h'(k)+1]$ را بررسی می‌کنیم، و همینطور تا مکان $T[m-1]$. بنابراین از مکان‌های $T[0],\ T[1],\ ...$ می‌گذریم تا اینکه در نهایت مکان $T[h'(k)-1]$ را بررسی کنیم. چون بررسی اولیه، توالی بررسی را تعیین می‌کند، فقط m توالی بررسی متفاوت موجود است.

بررسی خطی در پیاده سازی ساده است، ولی از یک مسئله دیگر بنام ***دسته بندی اولیه***[3] استفاده می‌کند. رانش‌های طولانی از مکان‌های اشغال شده ایجاد می‌شود، که زمان میانگین جستجو را افزایش می‌دهد. دسته‌ها زمانی پدید می‌آیند که یک مکان خالی که i مکان پر قبل از آن آمده است، با احتمال $(i+1)/m$ پر شود. رانش‌های بزرگِ مکان‌های اشغال شده سعی دارند که بزرگتر شوند و زمان میانگین جستجو افزایش می‌یابد.

بررسی درجه دوم

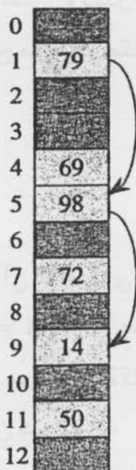
بررسی درجه دوم[4] از یک تابع درهم سازی به شکل

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m\ , \qquad (11.5)$$

1. auxiliary hash function
2. linear probing
3. primary clustering
4. quadratic probing

استفاده می‌کند، که h' یک تابع درهم سازی کمکی، c_1 و $c_2 \neq 0$ ثابتهای کمکی هستند و $i=0,1, \dots ,m-1$ است. مکان اولی که بررسی می‌شود $T[h'(k)]$ است؛ مکانهای بعدی که بررسی می‌شوند، با استفاده از مقداری که در حالت درجه دو بودن به بررسی شماره i بستگی دارد، تنظیم می‌گردند. این روش خیلی بهتر از بررسی خطی کار می‌کند، ولی برای استفاده کامل از جدول درهم سازی، مقادیر c_1، c_2 و m محدود می‌شوند.

مسئله ۳-۱۱ راهی برای انتخاب این پارامترها ارائه می‌دهد. همچنین اگر دو کلید، مکان بررسی اولیه یکسانی داشته باشند آنگاه توالی‌های جستجوی آنها یکسان هستند، زیرا $h(k_1,0) = h(k_2,0)$ نتیجه می‌دهد که $h(k_1,i) = h(k_2,i)$ این ویژگی یک شکل ملایم‌تر دسته بندی را نتیجه می‌دهد که دسته‌بندی *ثانویه*[1] نامیده می‌شود. مانند آنچه که در بررسی خطی وجود دارد، بررسی اولیه، کل توالی را تعیین می‌کند پس فقط m توالی بررسی متمایز استفاده می‌شود.



شکل ۱۱.۵ درج با استفاده از درهم سازی مضاعف. یک جدول درهم‌سازی با اندازه 13 با $h_1(k) = k \bmod 13$ و $h_2(k) = 1+(k \bmod 11)$ داریم. از آنجا که $14 \equiv 1$ (به پیمانه 13) و $14 \equiv 3$ (به پیمانه 11)، بعد از اینکه مکانهای 1 و 5 امتحان شدند و فهمیدیم که اشغال شده‌اند، کلید 14 در مکان خالی درج می‌شود.

درهم سازی مضاعف

درهم سازی مضاعف از بهترین روشهای موجود برای آدرس دهی باز است، زیرا جایگشتهای ایجاد

1. secondary clustering

شده بسیاری از مشخصات جایگشت هایی که تصادفاً انتخاب شده‌اند را دارا می‌باشند. **درهم‌سازی مضاعف**[1] از یک تابع درهم سازی به شکل

$$h(k,i) = (h_1(k)+ih_2(k)) \bmod m$$

استفاده می‌کند، که h_1 و h_2 توابع درهم سازی کمکی هستند. بررسی اوّلیه در مکان $T[h_1(k)]$ اتفاق می‌افتد؛ مکان‌های بررسی متوالی از مکان قبلی و با استفاده از مقدار $h_2(k)$ به پیمانه m تنظیم می‌شوند، بنابراین بر خلاف حالتهای بررسی خطی و بررسی درجه دو، در اینجا توالی بررسی از دو جهت به کلید k بستگی دارد زیرا مکان بررسی اولیه، مکانهای بعد یا هر دو ممکن است متفاوت باشند. شکل ۱۱.۵ نمونه‌ای از درج توسط درهم سازی مضاعف را ارائه می‌کند.

مقدار $h_2(k)$ باید برای کل جدول درهم سازی که جستجو می‌شود، نسبت به اندازه جدول درهم‌سازی یعنی m اول باشد. (تمرین ۳-۱۱.۴ را مشاهده کنید.) یک راه مناسب برای اینکه از این شرط مطمئن شویم اینست که m را توانی از 2 قرار دهیم و h_2 را طوری طراحی کنیم که همیشه یک عدد فرد را تولید کند. راه دیگر اینست که m عددی اول باشد و h_2 را به نحوی طراحی کنیم که همیشه یک عدد صحیح مثبت کوچکتر از m را برگرداند. به عنوان مثال، می‌توانیم m را مقداری اول انتخاب کنیم و قرار دهیم

$$h_1(k) = k \bmod m$$
$$h_2(k) = 1+(k \bmod m')$$

که در آن m' طوری انتخاب شده است که کمتر از m باشد (مثلاً $m-1$). به عنوان مثال، اگر $k = 123456$ و $m = 701$ و $m' = 700$ داریم $h_1(k) = 80$ و $h_2(k) = 257$ بنابراین اولین مکان بررسی، 80 است و سپس هر یک از 257 امین مکان (به پیمانه m) بررسی می‌شود تا اینکه کلید پیدا شود یا تمام مکانها امتحان شوند.

درهم سازی مضاعف بهتر از بررسی خطی و درجه دو است، زیرا به جای $\Theta(m)$، در آن $\Theta(m^2)$ توالی بررسی استفاده می‌شوند. هر جفت ممکن $(h_1(k),h_2(k))$ یک توالی بررسی متفاوت را به ما می‌دهد. به عنوان نتیجه، بنظر می‌رسد کارآیی درهم سازی مضاعف خیلی به کارآیی طرح "ایده‌آل" درهم سازی یکنواخت نزدیک باشد.

تحلیل درهم سازی آدرس باز

تحلیل ما از آدرس دهی باز، مانند تحلیل مان از زنجیره سازی، بر حسب ضریب بار $\alpha = n/m$ جدول

---

1. double hashing

درهم سازی بیان می‌شود، که n و m به بی نهایت میل می‌کنند. البته به کمک آدرس دهی باز، حداکثر یک عنصر در هر مکان داریم و بنابراین $n \leq m$ که موجب می‌شود $\alpha \leq 1$.

فرض می‌کنیم از درهم سازی یکنواخت استفاده شده است. در این طرح ایده آل، توالی بررسی $<h(k,0)\ ,\ h(k,1),\ ...,\ h(k,m-1)>$ که برای درج یا جستجوی کلید k استفاده می‌شود، بطور مساوی احتمال دارد که یک جایگشت از $<0,1,\ ...\ ,\ m-1>$ باشد. البته کلید داده شده یک توالی بررسی ثابت منحصر بفرد مربوط دارد؛ مفهومش اینست که با در نظر گرفتن توزیع احتمال روی فضای کلیدها و اعمال تابع درهم سازی روی کلیدها، احتمال هر توالی بررسی ممکن، مساوی است.

اکنون با فرض درهم سازی یکنواخت تعداد مورد انتظار بررسی‌ها برای درهم سازی بوسیله آدرس دهی باز را تحلیل می‌کنیم، با تحلیل تعداد بررسی‌های انجام شده در یک جستجوی ناموفق شروع می‌کنیم.

قضیه ۱۱.۶

در جدول درهم سازی آدرس باز با ضریب بار $\alpha = n/m < 1$ مفروض با فرض درهم سازی یکنواخت، تعداد مورد انتظار بررسی‌ها در یک جستجوی ناموفق حداکثر $1/(1-\alpha)$ است.

اثبات در یک جستجوی ناموفق، همگی بررسی‌ها غیر از آخرین آنها به یک مکان اشغال شده که حاوی کلید مطلوب نیست، دستیابی پیدا می‌کنند و آخرین مکان بررسی شده خالی است. متغیر تصادفی X را تعداد بررسی هایی که در یک جستجوی ناموفق انجام شده است تعریف می‌کنیم و همچنین برای $i = 1, 2, \ldots$ پیشامد A_i پیشامدی است که در آن یک iامین بررسی وجود دارد و این بررسی برای یک مکان اشغال شده انجام گرفته است. آنگاه پیشامد $\{X \geq i\}$ اشتراک پیشامدهای $A_1 \cap A_2 \cap \ldots \cap A_{i-1}$ است. $Pr\{X \geq i\}$ را با استفاده از محدود کردن $Pr\{A_1 \cap A_2 \cap \ldots \cap A_{i-1}\}$ محدود خواهیم کرد. داریم

$$Pr\ \{A_1 \cap A_2 \cap \ldots \cap A_{i-1}\} = Pr\{A_1\}\ .\ Pr\{A_2|A_1\}.\ Pr\{A_3|A_1 \cap A_2\} \ldots$$
$$Pr\{A_{i-1}\ |\ A_1 \cap A_2 \cap \ldots \cap A_{i-2}\}.$$

از آنجا که n عنصر و m مکان وجود دارد، $Pr\{A_1\} = n/m$ برای $j > 1$، احتمال اینکه j امین بررسی وجود دارد و این بررسی برای یک مکان اشغال شده صورت گرفته است برابر $(n-j+1)/(m-j+1)$ است و بیان می‌دارد که اولین $j-1$ بررسی برای مکانهای اشغال شده انجام گرفته است. این احتمال اثبات می‌شود زیرا یکی از $(n-(j-1))$ عنصر باقیمانده را در یکی از $(m-(j-1))$ مکان بررسی نشده پیدا می‌کنیم و با استفاده از فرض درهم سازی یکنواخت این احتمال برابر نسبت این کمیتها است. با در نظر گرفتن اینکه $n < m$ برای همه j هایی که $0 \leq j < m$ ایجاب می‌کند $(n-j)/(m-j) \leq n/m$، برای تمام i ها که 1

$m \le i \le m$ داریم

$$\begin{aligned}\Pr\{X \ge i\} &= \frac{n}{m}\cdot\frac{n-1}{m-1}\cdot\frac{n-2}{m-2}\cdots\frac{n-i+2}{m-i+2}\\ &\le \left(\frac{n}{m}\right)^{i-1}\\ &= \alpha^{i-1}\end{aligned}$$

حال از تساوی زیر برای محدود کردن تعداد مورد انتظار بررسی‌ها استفاده می‌کنیم:

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{\infty}\Pr\{X \ge i\}\\ &\le \sum_{i=1}^{\infty}\alpha^{i-1}\\ &= \sum_{i=0}^{\infty}\alpha^{i}\\ &= \frac{1}{1-\alpha}\ .\end{aligned}$$

■

حد بالای $1+\alpha+\alpha^2+\alpha^3+\dots$ تفسیر روشنی دارد. همیشه یک بررسی انجام می‌شود. با تقریب احتمال α، اولین بررسی یک مکان اشغال شده را پیدا می‌کند، بنابراین بررسی دوم لازم است. با تقریب احتمال α^2، دو مکان اول اشغال شده‌اند، بنابراین سومین بررسی لازم می‌باشد و به همین ترتیب.

اگر α ثابت باشد، قضیه ۱۱.۶ پیش بینی می‌کند که یک جستجوی ناموفق در زمان $O(1)$ اجرا می‌شود. به عنوان مثال اگر جدول درهم سازی نیمه پر باشد، تعداد متوسط بررسی‌ها در یک جستجوی ناموفق حداکثر $1/(1-0.5) = 2$ می‌باشد. اگر 90 درصد آن پر باشد، تعداد متوسط بررسی‌ها حداکثر $1/(1-0.9) = 10$ می‌شود.

قضیه ۱۱.۶ تقریباً بطور مستقیم کارآیی *HASH-INSERT* را به ما می‌دهد.

قضیه فرعی ۱۱.۷

با فرض در هم سازی یکنواخت، درج یک عنصر در یک جدول درهم سازی آدرس باز با ضریب بار α در حالت میانگین به حداکثر $1/(1-\alpha)$ بررسی نیاز دارد.

اثبات یک عنصر درج می‌شود، تنها اگر مکانی در جدول وجود داشته باشد، و بنابراین $\alpha<1$. درج یک کلید به یک جستجوی ناموفق نیاز دارد که به دنبال آن کلید در اولین مکان خالی پیدا شده قرار داده می‌شود. بنابراین، تعداد مورد انتظار بررسی‌ها حداکثر $1/(1-\alpha)$ است. ■

محاسبه تعداد مورد انتظار بررسی‌ها برای یک جستجوی موفق، به کار بیشتری احتیاج دارد.

قضیه ۱۱.۸

در جدول درهم سازی آدرس باز با ضریب بار $\alpha<1$ و با فرض درهم سازی یکنواخت و این فرض که همه کلیدهای جدول، احتمال جستجو شدن برابری دارند تعداد مورد انتظار بررسی‌ها در یک جستجوی موفق حداکثر برابر است با

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

اثبات جستجو برای کلید k همان توالی بررسی را دنبال می‌کند که وقتی عنصر با کلید k درج می‌شد انجام می‌گرفت. بنا به قضیه فرعی ۱۱.۷ اگر k $(i+1)$ امین کلیدی باشد که در جدول درهم‌سازی درج شده است، تعداد مورد انتظار بررسی‌ها در جستجو برای k حداکثر $1/(1-i/m)=m/(m-i)$ است. با گرفتن میانگین روی همه n کلید در جدول درهم سازی، تعداد میانگین بررسی‌ها در یک جستجوی موفق را بدست می‌آوریم:

$$\begin{aligned} \frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\ &= \frac{1}{\alpha}(H_m - H_{m-n}) , \end{aligned}$$

که در آن $H_i = \sum_{j=1}^{i} 1/j$، iامین عدد هارمونیک است. با بکارگیری تکنیک محدودسازی یک جمع با استفاده از یک انتگرال، داریم

$$\begin{aligned} \frac{1}{\alpha}(H_m - H_{m-n}) &= \frac{1}{\alpha}\sum_{k=m-n+1}^{m} 1/k \\ &\leq \frac{1}{\alpha}\int_{m-n}^{m}(1/x)\,dx \\ &= \frac{1}{\alpha}\ln\frac{m}{m-n} \\ &= \frac{1}{\alpha}\ln\frac{1}{1-\alpha} \end{aligned}$$

که یک حد روی تعداد مورد انتظار بررسی‌ها در یک جستجوی موفق است. ■

اگر جدول درهم سازی نیمه پر باشد، تعداد مورد انتظار بررسی‌ها در جستجوی موفق کمتر از 1.387 است. اگر جدول درهم سازی 90 درصد پر باشد، تعداد مورد انتظار بررسی‌ها کمتر از 2.559 می‌باشد.

تمرین‌ها

۱۱.۴-۱ درج کلیدهای $59,77,17,28,15,4,31,22,10$ را به جدول درهم سازی با طول $m = 11$ با

استفاده از آدرس دهی بازو تابع درهم سازی اولیه $h^{'}(k) = k \ mod \ m$ در نظر بگیرید. نتیجه درج این کلیدها با استفاده از بررسی خطی، با استفاده از بررسی درجه دوم $c_1 = 1$ و $c_2 = 3$ و استفاده از درهم‌سازی مضاعف با $h_2(k) = 1+(k \ mod \ (m\text{-}1))$ را شرح دهید.

۱۱.۴-۲ شبه کدی برای $HASH\text{-}DELETE$ به شکل طرح کلی آن که در متن گفته شد بنویسید و $HASH$ - $INSERT$ را به نحوی تغییر دهید که مقدار مشخص $DELETED$ را مدیریت کند.

۱۱.۴-۳ * فرض کنید که از درهم سازی مضاعف برای حل برخوردها استفاده می‌کنیم ؛ بعبارت دیگر از تابع درهم سازی $h(k,i) = (h_1(k)+ih_2(k)) \ mod \ m$ استفاده می‌کنیم. نشان دهید که اگر m و $h_2(k)$ برای کلید k بزرگترین مقسوم علیه مشترک $d \geq 1$ را داشته باشند، آنگاه یک جستجوی ناموفق برای کلید k، $(1/d)$ ام از جدول درهم سازی را قبل از برگشت به مکان $h_1(k)$ امتحان می‌کند. بنابراین وقتی $d = 1$ که به موجب آن m و $h_2(k)$ نسبت به هم اول می‌شوند، جستجو ممکن است کل جدول درهم سازی را بررسی کند.

۱۱.۴-۴ یک جدول درهم سازی آدرس باز با درهم سازی یکنواخت را در نظر بگیرید. وقتی ضریب بار $3/4$ و نیز وقتی $7/8$ است، یک حد بالا برای تعداد مورد انتظار بررسی‌ها در یک جستجوی ناموفق و در یک جستجوی موفق، ارائه دهید.

۱۱.۴-۵ * یک جدول درهم سازی آدرس باز با ضریب بار α را در نظر بگیرید. مقدار غیر صفر α را پیدا کنید که برای آن تعداد مورد انتظار بررسی‌ها در یک جستجوی ناموفق دو برابر تعداد مورد انتظار بررسی‌ها در یک جستجوی موفق است. برای این تعداد مورد انتظار بررسی‌ها از حد بالای داده شده در قضیه‌های ۱۱.۶ و ۱۱.۸ استفاده کنید.

---

* ۱۱.۵ درهم سازی کامل

اگر چه درهم سازی اغلب برای کارآیی مورد انتظار بسیار خوبش استفاده می‌شود، وقتی مجموعه کلیدها *ایستا*[1] است درهم سازی می‌تواند برای ایجاد کارآیی بدترین حالت بسیار خوب استفاده شود. وقتی کلیدها در جدول ذخیره می‌شوند، مجموعه کلیدها هیچگاه عوض نمی‌شود. برخی کاربردها طبیعتاً مجموعه ایستایی از کلیدها را دارند: مجموعه کلمات رزرو شده در یک زبان برنامه نویسی یا مجموعه نام فایلها در یک $CD\text{-}ROM$ را در نظر بگیرید. اگر تعداد دستیابی‌های مورد نیاز به حافظه برای یک جستجو در بدترین حالت $O(1)$ باشد، تکنیک درهم سازی را *درهم سازی کامل*[2] می‌نامیم.

ایده اصلی برای ایجاد طرح درهم سازی کامل، ساده است. از یک طرح درهم سازی دو سطحی با استفاده از درهم سازی جامع در هر مرحله استفاده می‌کنیم. شکل ۱۱.۶ این روش را شرح می‌دهد.

---

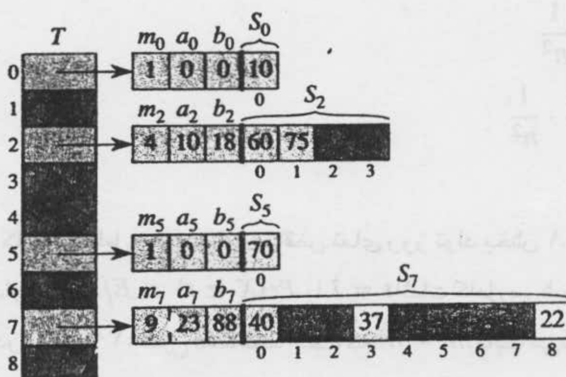
1. static　　2. perfect hashing

اولین سطح ضرورتاً همانند درهم سازی با استفاده از زنجیره سازی است: n کلید با استفاده از تابع درهم سازی h که به دقت از یک خانواده توابع درهم سازی جامع انتخاب شده‌اند به m مکان درهم‌سازی می‌شوند.

اما به جای ساختن یک لیست از کلیدهایی که به مکان j درهم سازی می‌شوند، از یک ***جدول درهم سازی ثانویه***[1] S_j به همراه تابع درهم سازی h_j مربوطه استفاده می‌کنیم. با انتخاب دقیق تابع درهم سازی h_j می‌توانیم تضمین کنیم که هیچ برخوردی در سطح ثانویه وجود نداشته باشد.

اما به منظور تضمین اینکه هیچ برخوردی در سطح دوم وجود نخواهد داشت، احتیاج خواهیم داشت که اندازه m_j جدول S_j را مربع تعداد n_j کلیدهایی که به مکان j در[illegible]ی می‌شوند قرار دهیم. با وجود یک چنین وابستگی درجه دومی بین m_j و n_j ممکن است به نظر بر[illegible] می‌یابد اما نشان خواهیم داد که با انتخاب درست تابع درهم سازی در سطح اول، مقدار مورد انتظار حافظه کل همچنان $O(n)$ باقی می‌ماند.

از توابع درهم سازی که از کلاسهای جامع توابع درهم سازی بخش ۱۱.۳.۳ انتخاب شده‌اند استفاده می‌کنیم. تابع درهم سازی اولین سطح از کلاس $\mathcal{H}_{p,m}$ انتخاب می‌شود، که همانند بخش ۱۱.۳.۳، p یک عدد اول و بزرگتر از مقادیر کلیدها است.



شکل ۱۱.۶ استفاده از یک درهم‌سازی کامل برای ذخیره مجموعه $k=\{10,22,37,40,60,70,75\}$ تابع درهم سازی خارجی $h(k) = ((ak+b) \bmod p) \bmod m$ است، که $a = 3$، $b = 42$، $p = 101$ و $m = 9$. مثلاً $h(75) = 2$، بنابراین کلید 75 به مکان 2 از جدول T، درهم‌سازی می‌شود. یک جدول درهم سازی ثانویه S_j، کلیه کلیدهایی که به مکان j درهم سازی می‌شوند را ذخیره می‌کند. اندازه جدول درهم‌سازی S_j برابر m_j است و تابع درهم سازی مربوطه $h_j(k) = ((a_jk+b_j) \bmod p) \bmod m_j$ است. از آنجا که $h_2(75) = 1$، کلید 75 در مکان 1 از جدول درهم سازی ثانویه S_2 ذخیره می‌شود. هیچ برخوردی در هیچیک از جداول درهم سازی ثانویه وجود ندارد و بنابراین جستجو در بدترین حالت، زمانی ثابت را صرف می‌کند.

---

1. secondary hash table

کلیدهایی که به مکان j درهم سازی می‌شوند، با استفاده از تابع درهم سازی h_j که از کلاس $\mathcal{H}_{p,m_j}$ انتخاب شده است مجدداً به جدول درهم سازی ثانویه S_j با اندازه m_j درهم سازی می‌شود.

در دو مرحله به پیش می‌رویم. اول اینکه مشخص می‌کنیم چگونه اطمینان حاصل کنیم که جداول ثانویه هیچ برخوردی ندارند. دوم اینکه، نشان می‌دهیم که مقدار حافظه مورد انتظار که در کل استفاده شده است - برای جدول درهم سازی اولیه و همگی جداول درهم سازی ثانویه - $O(n)$ است.

قضیه ۱۱.۹

اگر n کلید را در یک جدول درهم سازی با اندازه $m = n^2$ با استفاده از تابع درهم سازی h که تصادفاً از یک کلاس توابع درهم سازی انتخاب شده است، ذخیره کنیم. آنگاه احتمال اینکه برخوردی وجود داشته باشد کمتر از $1/2$ است.

اثبات $\binom{n}{2}$ جفت کلید وجود دارد که ممکن است برخورد داشته باشند؛ اگر h بطور تصادفی از یک خانواده $\mathcal{H}$ از توابع درهم سازی انتخاب شود، هر جفت با احتمال $1/m$ برخورد می‌کند. فرض کنید X یک متغیر تصادفی باشد که تعداد برخوردها را می‌شمارد. وقتی $m = n^2$ تعداد مورد انتظار برخوردها برابر است با

$$\begin{aligned} \mathrm{E}[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\ &= \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \\ &< 1/2 . \end{aligned}$$

(توجه داشته باشید که این تحلیل شبیه تحلیل تناقض نمای روز تولد بخش ۵.۴.۱ است.) با استفاده از نامساوی *Markov* داریم $Pr\{X \geq t\} \leq E[X]/t$، با $t = 1$ اثبات کامل می‌شود. ■

در وضعیتی که در قضیه ۱۱.۹ شرح داده شده است، که $m = n^2$، ثابت می‌شود تابع درهم سازی h که به طور تصادفی از $\mathcal{H}$ انتخاب شده است بیشتر احتمال دارد که هیچ برخوردی نداشته باشد. مجموعه K با n کلید داده شده است. تا درهم سازی شود (به یاد داشته باشید که K ایستا است)، بنابراین پیدا کردن یک تابع درهم سازی بدون برخورد h با تعداد کمی دنباله تصادفی، آسان است.

امّا وقتی n بزرگ است، یک جدول درهم سازی به اندازه $m = n^2$ افراط‌آمیز است. بنابراین، از روش درهم سازی دو سطحی پیروی می‌کنیم و از روش قضیه ۱۱.۹ فقط برای درهم سازی ورودی‌ها در هر مکان استفاده می‌کنیم. تابع درهم سازی h بیرونی، یا سطح اول، برای درهم سازی سازی کلیدها به $m = n$ مکان استفاده می‌شود. سپس اگر n_j کلید به مکان j درهم سازی شوند، از جدول درهم‌سازی ثانویه S_j با اندازه $m_j = n^2_j$ برای ایجاد یک جستجو با زمان ثابت و بدون برخورد استفاده می‌شود.

اکنون به موضوع اطمینان از اینکه حافظه کلی استفاده شده $O(n)$ است برمی گردیم. از آنجا که اندازه m_j از jامین جدول درهم سازی ثانویه بصورت درجه دو با تعداد n_j کلیدهای ذخیره شده رشد می‌کند، این ریسک وجود دارد که مقدار کل حافظه افراط‌آمیز شود.

اگر اندازه جدول اولین سطح $m = n$ باشد، مقدار حافظه استفاده شده برای جدول درهم سازی اولیه ذخیره سازی اندازه‌های m_j جداول درهم سازی ثانویه، و ذخیره سازی پارامترهای a_j و b_j که توابع h_j درهم سازی ثانویه که از کلاس $\mathcal{H}_{p,m_j}$ در بخش ۱۱.۳.۳ بدست آمده‌اند را تعریف می‌کنند (به جز وقتی که $n_j = 1$ و از $a=b=0$ استفاده می‌کنیم) برابر $O(n)$ است. قضیه زیر و یک قضیه فرعی، حدی برای اندازه‌های ترکیب شده مورد انتظار تمامی جداول درهم سازی ثانویه فراهم می‌کنند. قضیه فرعی دوم احتمال اینکه اندازه ترکیب شده تمام جداول درهم سازی ثانویه فرا خطی باشد را محدود می‌کند.

قضیه ۱۱.۱۰

اگر n کلید را با استفاده از تابع درهم سازی h که به طور تصادفی از یک کلاس جامع توابع درهم سازی انتخاب شده است، در یک جدول درهم سازی با اندازه $m = n$ ذخیره کنیم آنگاه

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n ,$$

که در آن n_j تعداد کلیدهایی است که به مکان درهم سازی می‌شود.

اثبات با استفاده از مشخصه زیر که برای هر عدد صحیح غیر منفی a برقرار است شروع می‌کنیم:

$$a^2 = a + 2\binom{a}{2} . \qquad (۱۱.۶)$$

داریم

$$\begin{aligned}
E\left[\sum_{j=0}^{m-1} n_j^2\right] &= E\left[\sum_{j=0}^{m-1}\left(n_j + 2\binom{n_j}{2}\right)\right] && \text{(بنا به معادله (۱۱.۶))}\\
&= E\left[\sum_{j=0}^{m-1} n_j\right] + 2E\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(بنا به خطی بودن انتظار (امید ریاضی))}\\
&= E[n] + 2E\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(بنا به معادله (۱۱.۱))}
\end{aligned}$$

$$= \quad n + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right]$$ (از آنجا که n متغیر تصادفی نیست)

برای بدست آوردن سری $\sum_{j=0}^{m-1}\binom{n_j}{2}$ ، مشاهده می‌کنیم که این عبارت برابر تعداد کل برخوردها می‌باشد. بنا به ویژگی‌های درهم سازی جامع، مقدار مورد انتظار این سری حداکثر برابر است با

$$\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2},$$

زیرا $m = n$. بنابراین

$$\begin{aligned}\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] &\le n + 2\,\frac{n-1}{2} \\ &= 2n-1 \\ &< 2n .\end{aligned}$$

■

قضیه فرعی ۱۱.۱۱

اگر n کلید را در یک جدول درهم سازی با اندازه $m = n$، با استفاده از تابع درهم سازی h که به طور تصادفی از یک کلاس جامع توابع درهم سازی انتخاب شده است ذخیره کنیم و اندازه هر جدول درهم‌سازی ثانویه را برای $j = 0,1,\ldots, m\text{-}1$ برابر $m_j=n_j^2$ قرار دهیم، مقدار حافظه‌ای که انتظار می‌رود برای همه جداول درهم سازی ثانویه به شکل درهم سازی کامل مورد نیاز باشد، کمتر از $2n$ است.

اثبات از آنجا که برای $j = 0, 1, \ldots, m\text{-}1$ داریم $m_j = n_j^2$، قضیه ۱۱.۱۰ بیان می‌دارد که

$$\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right] = \mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n , \qquad (۱۱.۷)$$

که اثبات را کامل می‌کند. ■

قضیه فرعی ۱۱.۱۲

اگر n کلید را در یک جدول درهم سازی با اندازه $m = n$ با استفاده از تابع درهم سازی h که از یک کلاس جامع توابع درهم سازی بطور تصادفی انتخاب شده است ذخیره کنیم و هر جدول درهم سازی ثانویه را برای $j = 0, 1, \ldots, m\text{-}1$ برابر $m_j = n_j^2$ قرار دهیم، احتمال اینکه کل حافظه استفاده شده

برای جدول درهم سازی ثانویه بیشتر از $4n$ باشد کمتر از $1/2$ است.

اثبات مجدداً نامساوی *Markov* را بکار می‌گیریم، $\Pr\{X \geq t\} \leq \mathrm{E}[X]/t$ ، این بار برای نامساوی (۱۱.۷) با $X = \sum_{j=0}^{m-1} m_j$ ، $t = 4n$ داریم:

$$\Pr\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} \leq \frac{\mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right]}{4n} < \frac{2n}{4n} = 1/2\,.$$

■

از قضیه فرعی ۱۱.۱۲، مشاهده می‌کنیم که با امتحان تعدادی از توابع درهم سازی که بطور تصادفی از خانواده جامع انتخاب شده‌اند به تابعی دست پیدا می‌کنیم که از یک مقدار معقول حافظه استفاده می‌کند.

تمرین‌ها

*۱۱.۵.۱ فرض کنید n کلید را با استفاده از آدرس دهی باز و درهم سازی یکنواخت در یک جدول درهم‌سازی درج می‌کنیم. فرض کنید $p(n,m)$ احتمال این باشد که هیچ برخوردی رخ ندهد. نشان دهید که $p(n,m) \leq e^{-n(n-1)/2m}$ (راهنمایی: معادله (۳.۱۱) را ملاحظه کنید.) ثابت کنید وقتی n بیشتر از $\sqrt{m}$ می‌شود احتمال جلوگیری از برخورد، به سرعت به صفر میل می‌کند.

مسائل

۱-۱۱ حد طولانی‌ترین بررسی برای درهم سازی

یک جدول درهم سازی با اندازه m برای ذخیره n داده استفاده می‌شود، با این شرط که $n \leq m/2$ از آدرس دهی باز برای حل برخوردها استفاده می‌شود.

a. با فرض درهم سازی یکنواخت، نشان دهید که برای $i = 1, 2, \ldots, n$، احتمال اینکه iامین درج اکیداً به بیش از k بررسی نیاز داشته باشد حداکثر 2^{-k} است.

b. نشان دهید که برای $i = 1, 2, \ldots, n$، احتمال اینکه iامین درج به بیش از $2\lg n$ بررسی نیاز داشته باشد حداکثر $1/n^2$ است.

فرض کنید متغیر تصادفی X_i به تعداد بررسی‌هایی که برای iامین درج نیاز است دلالت کند. شما در بخش (*b*) نشان داده‌اید که $\Pr\{X_i > 2\lg n\} \leq 1/n^2$ فرض کنید متغیر تصادفی $X = \max_{1 \leq i \leq n} X_i$ به ماکزیمم تعداد بررسی‌هایی که برای هر یک از n عمل درج لازم است دلالت کند.

c. نشان دهید که $Pr\{X > 2\ lgn\} \le 1/n$.

d. نشان دهید که طول مورد انتظار $E[X]$ طولانی‌ترین توالی بررسی‌ها برابر $O(lgn)$ است.

۱۱-۲ حد اندازه مکانی برای زنجیره سازی

فرض کنید یک جدول درهم سازی با n مکان داریم، که برخوردهای آن با استفاده از زنجیره سازی رفع شده است، و فرض کنید n کلید در جدول درج می‌شوند. هر کلید به طور مساوی احتمال دارد که به هر یک از مکان‌ها درهم سازی شود. فرض کنید M ماکزیمم تعداد کلیدها در بین مکان‌ها بعد از درج شدن همه کلیدها باشد. مأموریت شما اینست که ثابت کنید $O(lgn/lglgn)$ یک حد بالا برای $E[M]$، یعنی مقدار مورد انتظار M، است.

a. ثابت کنید احتمال Q_k که دقیقاً k کلید به یک مکان خاص درهم سازی شوند از رابطه زیر محاسبه می‌شود

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

b. فرض کنید P_k احتمال این با شد که $M = k$، بعبارت دیگر احتمال اینکه مکانی که شامل بیشترین تعداد کلیدها است دارای k کلید باشد. نشان دهید که $P_k \le nQ_k$

c. با استفاده از تقریب Stirling معادله (۳.۱۷)، نشان دهید که $Q_k < e^k/k^k$.

d. نشان دهید که یک ثابت $c > 1$ وجود دارد، بطوریکه برای $k_0 = clgn/lglgn$ داریم $Q_{k0} < 1/n^3$. نتیجه بگیرید که برای $k \ge k_0 = clgn/lglgn$ داریم $P_k < 1/n^2$.

e. ثابت کنید که

$$\mathrm{E}[M] \le \Pr\left\{M > \frac{c\lg n}{\lg\lg n}\right\} \cdot n + \Pr\left\{M \le \frac{c\lg n}{\lg\lg n}\right\} \cdot \frac{c\lg n}{\lg\lg n}.$$

نتیجه بگیرید که $E[M] = O(lgn/lglgn)$.

۱۱-۳ بررسی درجه دو

فرض کنید کلید k برای جستجوی آن در یک جدول درهم سازی با موقعیت‌های $0, 1, \ldots, m-1$، داده شده است و فرض کنید یک تابع درهم سازی h داریم که فضای کلید را به مجموعه $\{0, 1, \ldots, m-1\}$ نگاشت می‌کند. طرح جستجو به شکل زیر است.

1. مقدار $i \leftarrow h(k)$ را محاسبه کنید، و قرار دهید $j \leftarrow 0$.

2. مکان i را برای کلید مطلوب k بررسی کنید. اگر آن را پیدا کردید یا اگر این مکان خالی است،

جستجو را خاتمه دهید.

3.قرار دهید $j \leftarrow (j+1) \bmod m$ و $i \leftarrow (i+j) \bmod m$ و به گام 2 باز گردید.

فرض کنید m توانی از 2 است.

a. با نمایش ثابت‌های مناسب c_1 و c_2 برای معادله (۱۱.۵) نشان دهید که این طرح، نمونه‌ای از "طرح بررسی درجه دو" کلی است.

b. ثابت کنید که این الگوریتم در بدترین حالت، هر مکان جدول را بررسی می‌کند.

۴-۱۱ درهم سازی k- جامع و تأیید

فرض کنید $\mathcal{H}=\{h\}$ یک کلاس از توابع درهم سازی است که در آن هر h مجموعه مرجع U از کلیدها را به $\{0,1,..., m-1\}$ نگاشت می‌کند. می‌گوییم $\mathcal{H}$ k- جامع است اگر برای هر توالی ثابت از k کلید متمایز $<x^{(1)}, x^{(2)}, \ldots, x^{(k)}>$ و برای هر h که به صورت تصادفی از H انتخاب می‌شود، توالی $<h(x^{(1)}), h((x^{(2)}), ..., h(x^{(k)})>$ با احتمالی برابر هر یک از m^k توالی با طول k با عناصر انتخاب شده از $\{0,1, ..., m-1\}$ باشد.

a. نشان دهید اگر $\mathcal{H}$، 2- جامع باشد آنگاه جامع است.

b. فرض کنید U، مجموعه‌ای از مقادیر n-گانه باشد که از Z_p انتخاب می‌شوند و فرض کنید $B=Z_p$، که در آن p یک عدد اول است. برای هر n-گانه $a = <a_0 , a_1, ..., a_{n-1}>$ از مقادیر Z_p و برای هر $b \in Z_p$، تابع درهم سازی $h_{a,b} : U \rightarrow B$ روی یک ورودی n-گانه $x = <x_0, x_1, ..., x_{n-1}>$ از U را به صورت زیر تعریف کنید

$$h_{a,b}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b\right) \bmod p$$

و فرض کنید $\mathcal{H} = \{h_{a,b}\}$ ثابت کنید $\mathcal{H}$ 2-جامع است.

c. فرض کنید Alice و Bob مخفیانه روی یک تابع درهم سازی $h_{a,b}$ از یک خانواده 2-جامع $\mathcal{H}$ شامل توابع درهم سازی توافق می‌کنند. بعد از آن، Alice پیغام m را از طریق اینترنت به Bob می‌فرستد، که $m \in U$ او این پیغام را با فرستادن یک بر چسب تأیید $t = h_{a,b}(m)$ برای Bob تأیید می‌کند و Bob چک می‌کند که زوج (m,t) که دریافت می‌کند در $t = h_{a,b}(m)$ صدق می‌کند. فرض کنید یک دشمن در بین راه (m,t) را دریافت کرده و سعی می‌کند Bob را به وسیله جایگزین کردن این زوج با زوج متفاوت (m',t') گمراه کند. ثابت کنید احتمال این که دشمن در گمراه کردن Bob در پذیرفتن (m',t') موفق شود حداکثر $1/p$ است، بدون توجه به میزان توان محاسباتی دشمن.

۱۲ درخت جستجوی دو دویی

درخت‌های جستجو، ساختمان داده‌هایی هستند که بسیاری از اعمال مجموعه‌های پویا شامل جستجو، پیدا کردن مینیمم، ماکزیمم، عنصر ماقبل، عنصر مابعد، درج و حذف را پشتیبانی می‌کنند، بنابراین یک درخت جستجو می‌تواند هم به عنوان یک لغت نامه و هم به عنوان صف اولویت مورد استفاده قرار گیرد.

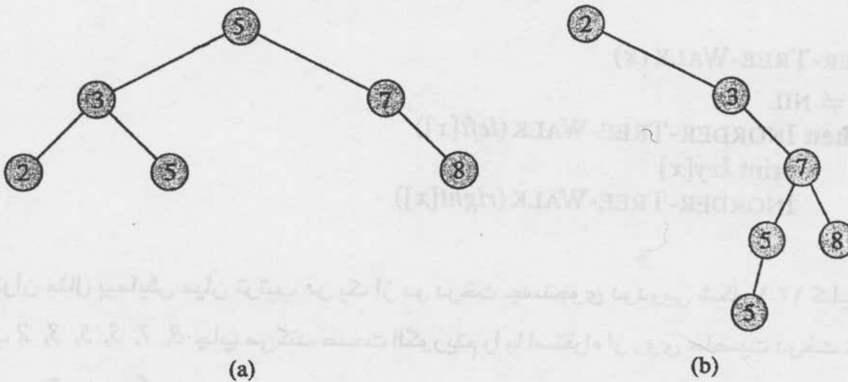
اعمال اصلی روی یک درخت جستجوی دو دویی در زمانی متناسب با ارتفاع درخت اجرا می‌شوند. این اعمال برای یک درخت جستجوی دودویی کامل با n گره، در بدترین حالت در زمان $\Theta(lgn)$ اجرا می‌شوند. بنابراین اگر درخت یک زنجیرهٔ خطی از n گره باشد این اعمال در بدترین حالت دارای زمان $\Theta(n)$ می‌باشند. در بخش ۱۲.۴ خواهیم دید ارتفاع مورد انتظار یک درخت جستجوی دودویی تصادفی ساخته شده برابر $O(lgn)$ است، بنابراین این اعمال اصلی مجموعه‌های پویا روی چنین درختی به طور متوسط در زمان $\Theta(lgn)$ اجرا می‌شوند.

در عمل همیشه نمی‌توانیم تضمین کنیم که درختهای جستجوی دودویی به صورت تصادفی ساخته می‌شوند ولی انواع مختلفی از درخت‌های جستجوی دودویی وجود دارد که می‌توان تضمین کرد که در بدترین حالت، زمان اجرای اعمال اصلی روی آنها مناسب است. فصل ۱۳ یکی از این انواع، بنام درخت‌های قرمز ـ سیاه که دارای ارتفاع $O(lgn)$ می‌باشند را ارائه می‌کند. فصل ۱۸ درخت‌های *B-tree* را معرفی می‌کند که مخصوصاً برای نگهداری پایگاه داده‌ها روی حافظه جانبی (دیسک) با دستیابی تصادفی، مناسب هستند.

بعد از ارائه ویژگی‌های اصلی درخت‌های جستجوی دودویی، این بخش نشان می‌دهد که چگونه درخت جستجو دودویی را برای چاپ تصاویر درخت به صورت مرتب، پیمایش کنیم، چگونه یک مقدار را در درخت جستجو کنیم، چگونه عنصر مینیمم یا ماکزیمم، عنصر ماقبل و عنصر مابعد یک عنصر را پیدا کنیم و چگونه در درخت اعمال درج یا حذف را انجام دهیم.

۱۲.۱ درخت جستجوی دودویی چیست؟

یک درخت جستجوی دودویی، همان طور که نامش نشان می‌دهد، روی یک درخت دودویی ساخته می‌شود مانند آنچه که در شکل ۱۲.۱ نشان داده شده است. چنین درختی می‌تواند توسط یک ساختمان دادهٔ پیوندی که در آن هر گره یک شیء[1] می‌باشد ارائه شود. هر گره علاوه بر یک فیلد کلید و داده‌های وابسته، شامل فیلدهای *left*، *right* و *p* می‌باشد که به ترتیب به گره‌های متناظر با فرزند چپ، فرزند راست و پدر آن گره اشاره می‌کنند. اگر پدر یا فرزندی وجود نداشته باشد فیلد متناظر، مقدار *NIL* را می‌گیرد. گره ریشه تنها گره‌ای است که فیلد پدر آن *NIL* است.



شکل ۱۲.۱ درخت جستجوی دودویی. برای هر گره *x*، کلیدهای زیر درخت چپ *x* حداکثر *key [x]* و کلیدهای زیر درخت راست *x* حداقل *key[x]* می‌باشند. درخت‌های جستجوی دودویی متفاوت می‌توانند یک مجموعهٔ یکسان از مقادیر را ارائه دهند. زمان اجرای اکثر اعمال درخت جستجو در بدترین حالت متناسب با ارتفاع درخت می‌باشد. *(a)* یک درخت جستجوی دودویی روی *6* گره با ارتفاع *2*. *(b)* یک درخت جستجوی دودویی کم کارآمدتر با ارتفاع *4* و شامل همان کلیدها.

در درخت جستجوی دودویی همیشه کلیدها طوری ذخیره می‌شوند که **ویژگی درخت جستجوی دودویی** حفظ شود:

فرض کنید *x* یک گره در درخت جستجوی دودویی باشد. اگر *y* گره‌ای در زیر درخت چپ *x* باشد آنگاه $key[x] \le key[y]$ است. اگر *y* گره‌ای در زیر درخت راست *x* باشد آنگاه $key[y] \le key[x]$

بنابراین در شکل ۱۲.۱*(a)* کلید ریشه 5 است، کلیدهای *2*، *3* و *5* در زیر درخت چپ آن از 5 بزرگتر نیستند و کلیدهای 7 و *8* در زیر درخت راست آن کوچکتر از 5 نمی‌باشند. این خاصیت برای هر گره‌ای

---

1. object

در درخت برقرار است. برای مثال، کلید 3 در شکل ۱۲.۱(a) از کلید 2 در زیر درخت چپ خود کوچکتر و از کلید 5 در زیر درخت راست خود بزرگتر نیست.

ویژگی درخت جستجوی دودویی به ما اجازه می‌دهد تا همهٔ کلیدهای درخت جستجوی دودویی را به صورت مرتب توسط یک الگوریتم بازگشتی ساده که پیمایش میان ترتیب [1] درخت نامیده می‌شود چاپ کنیم. چون کلید ریشهٔ یک زیر درخت بین مقادیر زیر درخت چپ و زیر درخت راست آن چاپ می‌شود، این الگوریتم این نام را گرفته است. (به طور مشابه یک پیمایش پیش ترتیب [2] درخت، ریشه را قبل از مقادیر زیر درخت‌ها چاپ می‌کند و یک پیمایش پس ترتیب، [3] ریشه را بعد از مقادیر زیر درخت‌های آن چاپ می‌کند.) برای استفاده از روال زیر به منظور چاپ تمامی عناصر یک درخت جستجوی دودویی T باید روال را به صورت *INORDER-TREE-WALK (root[T])* فراخوانی کنیم.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```

به عنوان مثال پیمایش میان ترتیب هر یک از دو درخت جستجوی دودویی شکل ۱۲.۱ کلیدها را به ترتیب *2 ,3 ,5 ,5 ,7 ,8* چاپ می‌کند. صحت الگوریتم را با استقراء از روی خاصیت درخت جستجوی دودویی نتیجه می‌گیریم.

پیمایش درخت جستجوی دودویی با n گره، به اندازهٔ $\Theta(n)$ طول می‌کشد، چون بعد از فراخوانی اولیه، روال به صورت بازگشتی دقیقاً دو بار برای هر گره در درخت - یکبار برای فرزند چپ و یکبار برای فرزند راست - فراخوانی می‌شود. قضیه زیر اثبات رسمی‌تری را ارائه می‌دهد که اجرای پیمایش میان ترتیب درخت در زمان خطی صورت می‌گیرد.

قضیه ۱۲.۱

اگر x ریشه یک زیر درخت n گره‌ای باشد، آنگاه فراخوانی *INORDER-TREE-WALK(x)* دارای مرتبه زمانی $\Theta(n)$ می‌باشد.

اثبات فرض کنید $T(n)$ زمانی است که فراخوانی *INORDER-TREE-WALK* برای ریشه یک زیر درخت n گره‌ای طول می‌کشد. روال *INORDER-TREE-WALK* زمان کوچک ثابتی را روی یک زیر

---

1. inorder
2. Preorder tree walk
3. postorder tree walk

درخت خالی صرف می‌کند (برای بررسی $x \neq NIL$) و بنابراین برای مقدار ثابت و مثبت c داریم $T(0) = c$.

برای $n>0$ فرض کنید که *INORDER-TREE-WALK* برای گره x که زیر درخت چپش k گره و زیر درخت راستش $n-k-1$ گره دارد فراخوانی شود. زمان اجرای *INORDER-TREE-WALK(x)* برابر $T(n)=T(k)+T(n-k-1)+d$ می‌باشد که d مقداری ثابت و مثبت است. زمان اجرای *INORDER-TREE-WALK(x)* را بدون در نظر گرفتن زمان صرف شده در فراخوانی‌های بازگشتی منعکس می‌کند.

با استفاده از روش جایگذاری و با اثبات اینکه $T(n)= (c+d)n+c$، نشان می‌دهیم $T(n)= \Theta(n)$ است.

برای $n = 0$ داریم $(c+d) . 0 + c = c = T(0)$ و برای $n>0$ داریم:

$$
\begin{aligned}
T(n) &= T(k) + T(n-k-1) + d \\
&= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\
&= (c+d)n + c - (c+d) + c + d \\
&= (c+d)n + c ,
\end{aligned}
$$

که اثبات را کامل می‌کند. ■

تمرین‌ها

۱- ۱۲.۱ برای مجموعه کلیدهای $\{1, 4, 5, 10, 16, 17, 21\}$ درخت‌های جستجوی دودویی با ارتفاع‌های $2, 3, 4, 5, 6$ رسم کنید.

۲- ۱۲.۱ تفاوت بین ویژگی درخت جستجوی دودویی و ویژگی درخت *min-heap* چیست؟ (به صفحه ۱۴۲ مراجعه کنید.) آیا می‌توان برای چاپ کلیدهای یک درخت n گره‌ای به صورت مرتب با مرتبه زمانی $O(n)$ از ویژگی درخت *min-heap* استفاده کرد؟ توضیح دهید چگونه، یا چرا ممکن نیست.

۳- ۱۲.۱ یک الگوریتم غیر بازگشتی برای پیمایش میان ترتیب درخت ارائه دهید. (راهنمایی: یک راه حل ساده، استفاده از پشته به عنوان یک ساختمان داده کمکی است و یک راه حل پیچیده‌تر ولی دقیق‌تر وجود دارد که از پشته استفاده نمی‌کند، اما در آن فرض می‌کنیم دو اشاره گر می‌توانند برای تساوی تست شوند.)

۴- ۱۲.۱ الگوریتم‌های بازگشتی برای انجام پیمایش‌های پیش ترتیب و پس ترتیب برای یک درخت n گرهی با مرتبه زمانی $\Theta(n)$ بنویسید.

۵- ۱۲.۱ ثابت کنید هر الگوریتم مبتنی بر مقایسه برای ساختن یک درخت جستجوی دودویی از یک لیست n عنصری دلخواه، در بدترین حالت دارای مرتبه زمانی $\Omega(nlgn)$ است، چون مرتب کردن n

عنصر در مدل مقایسه‌ای دارای مرتبه زمانی $\Omega(nlgn)$ است.

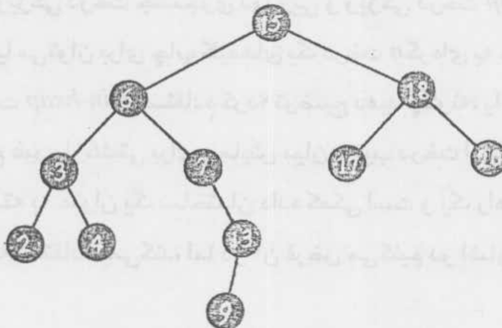
۱۲.۲ پرس و جوی یک درخت جستجوی دودویی

یک عمل متداول که روی درخت جستجوی دودویی انجام می‌شود پیدا کردن یک کلید در درخت است. در کنار عمل *SEARCH* درخت جستجوی دودویی می‌توانند اعمال دیگری از جمله پیدا کردن *MAXIMUM, MINIMUM, PREDECESSOR, SUCCESSOR* را پشتیبانی کند. در این بخش این اعمال را بررسی می‌کنیم و نشان می‌دهیم که هر یک می‌توانند در زمان $O(h)$ روی یک درخت جستجوی دودویی با ارتفاع h انجام شوند.

جستجو

از روال زیر برای یک گره با کلید داده شده در درخت جستجوی دودویی استفاده می‌کنیم. اشاره‌گری به ریشهٔ درخت و کلید k داده شده است. اگر گره‌ای با کلید k وجود داشته باشد *TREE-SEARCH* یک اشاره‌گر به آن را بر می‌گرداند در غیر اینصورت *NIL* را بر می‌گرداند.

```
TREE-SEARCH(x, k)
1  if x = NIL or k = key[x]
2      then return x
3  if k < key[x]
4      then return TREE-SEARCH(left[x], k)
5      else  return TREE-SEARCH(right[x], k)
```



شکل ۱۲.۲ پرس و جوی یک درخت جستجوی دودویی. برای جستجوی کلید 12 در درخت باید مسیر 12→7→6→15 را از ریشه بپیماییم. کلید مینیمم در درخت 2 می‌باشد که می‌توان با پیمایش اشاره گرهای *left* از ریشه آن را پیدا کرد. کلید ماکزیمم 20 با دنبال کردن اشاره گرهای *right* از ریشه پیدا می‌شود. مابعد گره با کلید 15، گره با کلید 17 است. زیرا کوچکترین کلید در زیر درخت راست 15 می‌باشد. گره با کلید 13 زیر درخت راست ندارد بنابراین مابعد آن کوچکترین جد آن می‌باشد که فرزند چپش نیز یک جد 13 می‌باشد. در این حالت گره با کلید 15 گره مابعد آن است.

این روال جستجو را از ریشه شروع می‌کند و همان طور که در شکل ۱۲.۲ نشان داده شده است، مسیری را به طرف پایین در درخت دنبال می‌کند. برای هر گره x آن را ملاقات کرده و کلید k را با $key[x]$ مقایسه می‌کند. اگر دو کلید برابر باشند جستجو پایان می‌یابد. اگر k کوچکتر از $key[x]$ باشد جستجو در زیر درخت چپ x ادامه می‌یابد، زیرا ویژگی درخت جستجوی دودویی بیان می‌کند که k نمی‌تواند در زیر درخت راست ذخیره شده باشد. به طور متقارن اگر k بزرگتر از $key[x]$ باشد جستجو در زیر درخت راست ادامه می‌یابد. گره‌هایی که در طی فرآیند بازگشتی ملاقات می‌شوند یک مسیر از ریشه به سمت پایین تشکیل می‌دهند و بنابراین زمان اجرای *TREE-SEARCH*، $O(h)$ می‌باشد که h ارتفاع درخت است.

همین روال را می‌توان به صورت تکراری با بسط دادن [1] حالت بازگشتی، در یک حلقه *while* نوشت. این مدل روی بیشتر کامپیوترها کارآمدتر است.

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2      do if k < key[x]
3            then x ← left[x]
4            else x ← right[x]
5  return x
```

مینیمم و ماکزیمم

عنصری که کلید آن در درخت جستجوی دودویی مینیمم است همیشه می‌تواند از طریق دنبال کردن اشاره گرهای فرزندان چپ از ریشه تا رسیدن به *NIL* پیدا شود، مانند آنچه که در شکل ۱۲.۲ نشان داده شده است. روال زیر اشاره‌گری به عنصر مینیمم در زیر درخت مشتق شده از x را بر می‌گرداند.

```
TREE-MINIMUM(x)
1  while left[x] ≠ NIL
2      do x ← left[x]
3  return x
```

ویژگی درخت جستجوی دودویی صحت *TREE-MINIMUM* را تضمین می‌کند. اگر گره x هیچ زیر درخت چپی نداشته باشد، آنگاه چون هر کلید در زیر درخت راست x حداقل برابر $key[x]$ است، کلید مینیمم در زیر درخت مشتق شده از x $key[x]$ است. اگر گره x یک زیر درخت چپ داشته باشد آنگاه چون هیچ کلیدی در زیر درخت راست کوچکتر از $key[x]$ نیست و هر کلیدی در زیر درخت چپ بزرگتر از $key[x]$ نیست، کلید مینیمم در زیر درخت مشتق شده از x می‌تواند در زیر درخت مشتق شده از

---
1. unrolling

left[x] پیدا شود.

شبه کد برای *TREE-MAXIMUM* نیز متقارن است.

```
TREE-MAXIMUM(x)
1  while right[x] ≠ NIL
2      do x ← right[x]
3  return x
```

هر دو روال روی یک درخت با ارتفاع *k* در زمان *O(h)* اجرا می‌شوند، زیرا همانند *TREE-SEARCH* ترتیب گره‌های ملاقات شده، مسیری از ریشه به طرف پایین را تشکیل می‌دهند.

گره ماقبل و گره مابعد

یک گره در درخت جستجوی دودویی داده شده است، گاهی مهم است که بتوانیم گره ما بعد آن را در ترتیب مرتب شده توسط پیمایش میان ترتیب پیدا کنیم. اگر همه کلیدها متفاوت باشند گره مابعد گره *x* گره‌ای است که کلید آن کوچکترین کلید بزرگتر از *key[x]* است. ساختار درخت جستجوی دودویی به ما امکان می‌دهد که گره مابعد یک گره را بدون هیچ مقایسه‌ای بین کلیدها مشخص کنیم. روال زیر گره ما بعد گره *x* در درخت جستجوی دودویی را در صورت وجود برمی‌گرداند و اگر *x* بزرگترین کلید در درخت را داشته باشد، *NIL* برمی‌گرداند.

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2      then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5      do x ← y
6         y ← p[y]
7  return y
```

کد روال *TREE-SUCCESSOR* به دو حالت تقسیم می‌شود. اگر زیر درخت راست *x* تهی نباشد آنگاه گره مابعد *x* چپ‌ترین گره در زیر درخت راست می‌باشد که با فراخوانی *TREE-MINIMUM(right[x])* در خط دوم پیدا می‌شود. برای مثال گره مابعد گره با کلید *15*، در شکل ۱۲.۲، گره با کلید *17* است.

از طرف دیگر همان طور که تمرین ۶- ۱۲.۲ از شما می‌خواهد نشان دهید اگر زیر درخت راست *x* تهی باشد و *x* یک گره مابعد بنام *y* داشته باشد، آنگاه *y* کوچک‌ترین جد *x* است که فرزند چپ آن نیز جد *x* می‌باشد. در شکل ۱۲.۲ گره مابعد گره با کلید *13*، گره با کلید *15* می‌باشد. برای پیدا کردن *y* به راحتی از *x* به طرف بالای درخت حرکت می‌کنیم تا زمانی که با گره‌ای مواجه شویم که فرزند چپ پدرش باشد.

این کار توسط خطوط ۷-۳ روال *TREE-SUCCESSOR* انجام می‌شود.

زمان اجرای *TREE-SUCCESSOR* روی درختی با ارتفاع h ، $O(h)$ است، چون مسیری بطرف بالا یا پایین درخت را دنبال می‌کنیم. روال *TREE-PREDECESSOR* که نسبت به *TREE-SUCCESSOR* متقارن است هم در زمان $O(h)$ اجرا می‌شود.

حتی اگر کلیدها متفاوت نباشند گره ماقبل و گره مابعد هر گره x را به عنوان گره‌ای که به ترتیب از فراخوانی *TREE- SUCCESSOR* و *TREE-PREDECESSOR* بر می‌گردد ، تعریف می‌کنیم.

قضیه ۱۲.۲

اعمال روی مجموعه‌های پویا یعنی *MAXIMUM, MINIMUM, SEARCH, PREDECESSOR, SUCCESSOR* می‌توانند در یک درخت جستجوی دودویی با ارتفاع h در زمان $O(h)$ اجرا شود.

تمرین‌ها

۱- ۱۲.۲ فرض کنید که در یک درخت جستجوی دودویی اعداد بین *1* تا *1000* داریم و می‌خواهیم عدد *363* را پیدا کنیم. کدامیک از ترتیب‌های زیر نمی‌تواند توالی گره‌های بررسی شده باشد؟

a. 2,252,401,398,330,344,397,363

b. 924,220,911,244,898,258,362,303

c. 925,202,911,240,912,245,363

d. 2,399,387,219,266,382,381,278,363

e. 935,278,347,621,299,392,358,363

۲- ۱۲.۲ صورت بازگشتی روال‌های *TREE-MINIMUM* و *TREE-MAXIMUM* را بنویسید.

۳- ۱۲.۲ روال *TREE - PREDECESSOR* را بنویسید.

۴- ۱۲.۲ پرفسور *Bunyan* تصور می‌کند که یک ویژگی قابل توجه درخت جستجوی دودویی را کشف کرده است. فرض کنید که‌جستجوی‌درخت جستجوی دودویی برای یافتن کلید k به برگ ختم شود. سه مجموعه را در نظر بگیرید: A، کلیدهای سمت چپ مسیر جستجو، B، کلیدهای مسیر جستجو و C، کلیدهای سمت راست مسیر جستجو. پرفسور *Bunyan* ادعا می‌کند که بین هر سه کلید $a \in A$ و $b \in B$ و $c \in C$ رابطهٔ $a \le b \le c$ باید برقرار باشد. برای ادعای پرفسور، کوچکترین مثال نقض ممکن را ارائه دهید.

۵- ۱۲.۲ نشان دهید که اگر در درخت جستجوی دودویی یک گره، دو فرزند داشته باشد آنگاه گره مابعد آن فرزند چپ، و گرهٔ ما قبل آن فرزند راست ندارد.

۶- ۱۲.۲ یک درخت جستجوی دودویی T را در نظر بگیرید که کلیدهای آن متفاوت باشند. نشان دهید

اگر زیر درخت راست گره x در T تهی باشد و x رأس مابعدی بنام y داشته باشد، آنگاه y پایین‌ترین جد x است که فرزند چپ آن نیز جد x است (توجه کنید که هر گره، جد خودش نیز می‌باشد.)

۷- ۱۲.۲ پیمایش میان ترتیب یک درخت جستجوی دودویی با n گره می‌تواند با پیدا کردن عنصر مینیمم در یک درخت با *TREE-MINIMUM* و سپس n-1 فراخوانی *TREE-SUCCESSOR* پیاده‌سازی شود. ثابت کنید که این الگوریتم در زمان $\Theta(n)$ اجرا می‌شود.

۸- ۱۲.۲ ثابت کنید که بدون توجه به اینکه در یک درخت جستجوی دودویی با ارتفاع h از چه گره‌ای شروع کنیم، k فراخوانی موفقیت‌آمیز *TREE-SUCCESSOR*، زمان $O(k+h)$ را صرف می‌کند.

۹- ۱۲.۲ T را به عنوان یک درخت جستجوی دودویی که کلیدهای آن متفاوت است در نظر بگیرید. x را یک گره برگ و y را پدر آن فرض کنید. نشان دهید که $key[y]$ کوچکترین کلید بزرگتر از $key[x]$ یا بزرگترین کلید کوچکتر از $key[x]$ در T است.

۱۲.۳ درج و حذف

اعمال حذف و درج باعث می‌شوند که مجموعه‌های پویایی که با درخت جستجوی دودویی نمایش داده شده‌اند، تغییر یابند. ساختمان داده باید طوری تغییر کند که این تغییر را منعکس کند ولی به روشی که ویژگی درخت جستجوی دودویی حفظ شود. همان طو که خواهیم دید، تغییر دادن درخت برای درج یک عنصر جدید نسبتاً ساده است ولی کنترل حذف تا حدی پیچیده‌تر می‌باشد.

درج

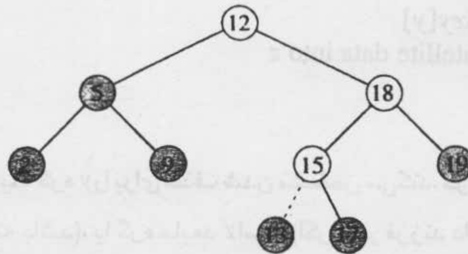
برای درج یک مقدار جدید v در درخت جستجوی دودویی T، از روال *TREE-INSERT* استفاده می‌کنیم. گره z برای روال فرستاده می‌شود و برای z داریم $key[z] = v$, $left[z] = NIL$ و $right[z] = NIL$. روال T، برخی از فیلدهای z را طوری تغییر می‌دهد که z در مکان مناسبی در درخت درج شود.

```
TREE-INSERT(T, z)
1  y ← NIL
2  x ← root[T]
3  while x ≠ NIL
4      do y ← x
5         if key[z] < key[x]
6            then x ← left[x]
7            else x ← right[x]
8  p[z] ← y
9  if y = NIL
```

```
10      then root[T] ← z                        ▷ Tree T was empty
11      else if key[z] < key[y]
12              then left[y] ← z
13              else right[y] ← z
```

شکل ۱۲.۳ نحوه کار *TREE-INSERT* را نشان می‌دهد. *TREE-INSERT* درست مانند روال‌های *TREE-ITERATIVE-SEARCH* و *TREE-SEARCH* از ریشه درخت شروع شده و یک مسیر رو به پایین را دنبال می‌کند. اشاره گر x مسیر را دنبال می‌کند و اشاره گر y، پدر x را نگه می‌دارد. بعد از مقداردهی اولیه، حلقه *while* در خطوط ۷-۳ باعث می‌شود که با توجه به مقایسه *key[z]* با *key[x]* این دو اشاره گر، رو به پایین، چپ یا راست درخت حرکت کنند تا زمانیکه x به *NIL* برسد. این *NIL* مکانی است که داده ورودی z را می‌خواهیم قرار دهیم. خطوط ۱۳-۸ اشاره گرها را برای درج z تنظیم می‌کنند.

روال *TREE-INSERT* همانند عملیات قبلی روی درخت‌های جستجو در زمان $O(h)$ روی درختی با ارتفاع h انجام می‌شود.



شکل ۱۲.۳ درج داده‌ای با کلید *13* در یک درخت جستجوی دودویی. گره‌های سایه زده شده مسیر را از ریشه به طرف پایین به سمت مکانی که گره درج می‌شود نشان می‌دهد. خط تیره در درخت، پیوندی را نشان می‌دهد که برای درج داده، اضافه شده است.

حذف

تابع حذف گره داده شده z از یک درخت جستجوی دودویی را حذف می‌کند، این تابع یک اشاره گر به z را به عنوان آرگومان ورودی می‌پذیرد. تابع سه حالت را که در شکل ۱۲.۴ نشان داده شده است در نظر می‌گیرد. اگر z فرزندی نداشته باشد، پدر آن یعنی *p[z]* را طوری تغییر می‌دهیم که جایگزین z شده و به عنوان فرزند *NIL* را بپذیرد. اگر تنها یک فرزند داشته باشد، z را با ایجاد یک پیوند جدید بین فرزند و پدر گره، حذف می‌کنیم.[1] در نهایت اگر گره دو فرزند داشته باشد عنصر مابعد z بنام y که هیچ فرزند

1.splice out

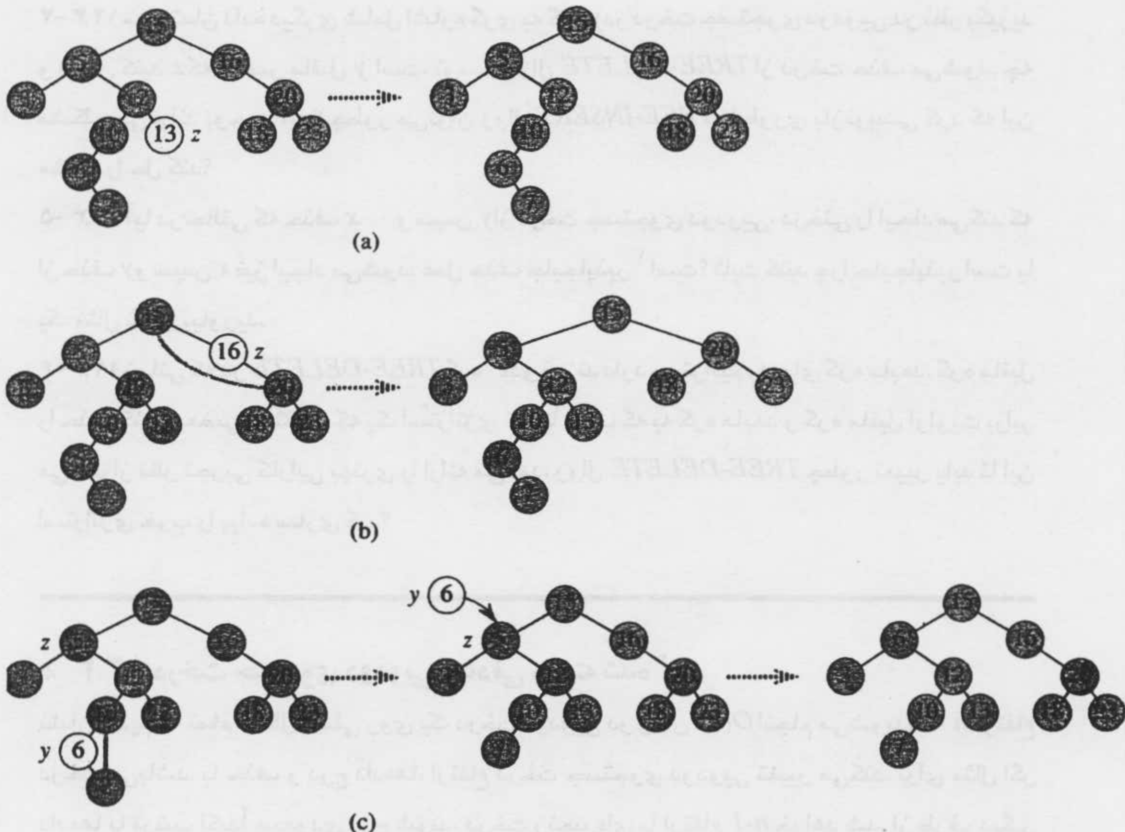
چپی نداشته باشد (تمرین ۵- ۱۲.۲ را ملاحظه کنید) را حذف و سپس کلید و داده‌های وابسته y را جایگزین کلید و داده‌های وابسته z می‌کنیم. کد *TREE-DELETE* این سه حالت را با کمی تفاوت سازماندهی می‌کند.

```
TREE-DELETE(T, z)
 1  if left[z] = NIL or right[z] = NIL
 2      then y ← z
 3      else  y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5      then x ← left[y]
 6      else  x ← right[y]
 7  if x ≠ NIL
 8      then p[x] ← p[y]
 9  if p[y] = NIL
10      then root[T] ← x
11      else if y = left[p[y]]
12              then left[p[y]] ← x
13              else  right[p[y]] ← x
14  if y ≠ z
15      then key[z] ← key[y]
16           copy y's satellite data into z
17  return y
```

در خطوط ۳-۱ الگوریتم یک گره y را برای حذف شدن مشخص می‌کند. گره y یا گره ورودی z است (اگر z حداکثر 1 فرزند داشته باشد)، یا گره مابعد z است (اگر z دو فرزند داشته باشد).سپس در خطوط ۶-۴، x را برابر فرزند غیر NIL گره y می‌کند یا اگر y فرزندی نداشت، x را NIL می‌کند. گره y در خطوط ۱۳-۷ با تغییر اشاره گرهای $p[y]$ و x حذف می‌شود. حذف y زمانیکه کنترل مناسب شرایط مرزی لازم است، تا حدی پیچیده می‌شود و این حالت زمانی رخ می‌دهد که $x=NIL$ باشد یا y به ریشه اشاره کند. در نهایت در خطوط ۱۶-۱۴ اگر رأس مابعد z گره‌ای باشد که حذف شده است، کلید و داده‌های وابسته y به z منتقل شده و روی کلید و داده‌های قبلی نوشته می‌شود. گره y در خط ۱۷ بر گردانده می‌شود. بنابراین با فراخوانی روال می‌توان آن را از طریق یک لیست آزاد شده بازیابی کرد. روال روی درختی با ارتفاع h در زمان $O(h)$ اجرا می‌شود.

قضیه ۱۲.۳

اعمال *DELETE* و *INSERT* مجموعه‌های پویا، می‌توانند در زمان $O(h)$ روی یک درخت جستجوی دودویی با ارتفاع h اجرا شوند.



شکل ۱۲.۴ حذف گره z از درخت جستجوی دودویی. اینکه واقعاً کدام گره حذف می‌شود به تعداد فرزندان z بستگی دارد؛ این گره به صورت سایه زده نشان داده شده است. (a) اگر z فرزندی نداشته باشد فقط آن را حذف می‌کنیم (b) اگر z تنها یک فرزند داشته باشد آنرا نیز حذف می‌کنیم. (c) اگر z دو فرزند داشته باشد، عنصر مابعد آن یعنی y را که حداکثر یک فرزند دارد حذف کرده و سپس کلید و داده‌های وابستهٔ z را با کلید و داده‌های y عوض می‌کنیم.

تمرین‌ها

۱- ۱۲.۳ صورت بازگشتی تابع *TREE-INSERT* را بنویسید.

۲- ۱۲.۳ فرض کنید یک درخت جستجوی دودویی با درج مکرر مقادیر مختلف در درخت ساخته شده است. ثابت کنید تعداد گره‌های تست شده هنگام جستجوی یک مقدار، از تعداد گره‌هایی که هنگامیکه مقدار برای اولین بار در درخت درج می‌شود، یکی بیشتر است.

۳- ۱۲.۳ می‌توانیم مجموعه‌ای از n عدد را ابتدا با ساختن درخت جستجوی دودویی شامل آن اعداد (با استفاده مکرر از روال *TREE-INSERT* برای درج اعداد بصورت یکی یکی) و سپس چاپ اعداد با استفاده از پیمایش میان ترتیب مرتب کنیم. مرتبه زمانی در بدترین حالت برای اجرای این الگوریتم مرتب سازی چیست؟

۴- ۱۲.۳ ساختمان داده دیگری شامل اشاره گری به گره y در درخت جستجوی دودویی در نظر بگیرید و فرض کنید z که عنصر ماقبل y است، توسط روال *TREE-DELETE* از درخت حذف می‌شود. چه مشکلی می‌تواند بوجود آید؟ چطور می‌توان روال *TREE-INSERT* را طوری بازنویسی کرد که این مشکل را حل کند؟

۵- ۱۲.۳ آیا در حالتی که حذف x و سپس y از درخت جستجوی دودویی، درختی را ایجاد می‌کند که از حذف y و سپس x نیز ایجاد می‌شود، عمل حذف جابجاپذیر [۱] است؟ ثابت کنید چرا جابجاپذیر است یا یک مثال نقض بیاورید.

۶- ۱۲.۳ زمانی که در *TREE-DELETE* گره z دو فرزند دارد می‌توانیم به جای گره مابعد، گره ماقبل را حذف کنیم. بعضی معتقدند که یک استراتژی نسبتاً خوب که به گره مابعد و گره ماقبل اولویت برابر می‌دهد از نظر تجربی کارایی بهتری را ارائه می‌دهد. روال *TREE-DELETE* چطور تغییر یابد تا این استراتژی خوب را پیاده سازی کند؟

---

* ۱۲.۴ درخت جستجوی دودویی تصادفی ساخته شده [۲]

نشان دادیم که تمام اعمال اصلی روی یک درخت دودویی در زمان $O(h)$ انجام می‌شوند، که h ارتفاع درخت می‌باشد. با حذف و درج داده‌ها، ارتفاع درخت جستجوی دودویی تغییر می‌کند. برای مثال اگر داده‌ها با ترتیب اکیداً صعودی درج شوند، درخت زنجیره‌ای با ارتفاع $n-1$ خواهد شد. از طرف دیگر تمرین ۴-۱۳.۵ نشان می‌دهد که $h \geq \lfloor lgn \rfloor$ است. بنابراین مانند آنچه که در مورد مرتب سازی سریع [۳] نشان دادیم، رفتار در حالت میانگین به بدترین حالت نزدیکتر است تا بهترین حالت.

متأسفانه در مورد ارتفاع متوسط یک درخت جستجوی دودویی، زمانی که هم درج و هم حذف برای ساخت آن استفاده می‌شود، اطلاعات کمی در دست است. زمانی که درخت تنها با درج ساخته می‌شود تحلیل آن راحت‌تر می‌شود. بنابراین یک درخت جستجوی دودویی تصادفی ساخته شده روی n کلید را به عنوان درختی که از درج کلیدها در یک درخت اولیهٔ خالی با ترتیب تصادفی بوجود می‌آید تعریف کنیم، که هر یک از $n!$ جایگشت کلیدهای ورودی، دارای احتمال برابری است. (تمرین ۳- ۱۲.۴ از شما می‌خواهد نشان دهید این مفهوم با این فرض که هر درخت جستجوی دودویی روی n کلید دارای احتمال برابری است» فرق دارد.) در این بخش نشان خواهیم داد که ارتفاعی که برای درخت جستجوی دودویی تصادفی ساخته شده روی n کلید انتظار می‌رود $O(lgn)$ می‌باشد. فرض می‌کنیم که همه کلیدها متفاوت باشند.

---

1. *commutative*
2. *randomly built binary search tree*
3. *quick sort*

با تعریف سه متغیر تصادفی که به محاسبه ارتفاع درخت جستجوی دودویی تصادفی ساخته شده کمک می‌کند شروع می‌کنیم. ارتفاع درخت جستجوی دودویی تصادفی ساخته شده روی n کلید را X_n و *ارتفاع نمایی* [1] Y_n را به صورت $Y_n = 2^{X_n}$ تعریف می‌کنیم. زمانیکه یک درخت جستجوی دودویی روی n کلید را ساختیم، یک کلید را به عنوان ریشه انتخاب کرده و R_n را برای مشخص کردن متغیر تصادفی که ترتیب کلید را در مجموعه‌ای از n کلید نگه می‌دارد قرار می‌دهیم. مقدار R_n با احتمال یکسانی متناسب با یکی از اعضای مجموعه $\{1, 2, ..., n\}$ می‌باشد. اگر $R_n=i$ باشد آنگاه زیر درخت چپ ریشه، یک درخت جستجوی دودویی تصادفی ساخته شده روی $i\text{-}1$ کلید وزیر درخت راست آن یک درخت جستجوی دودویی تصادفی ساخته شده روی $n\text{-}i$ کلید می‌باشد. چون ارتفاع درخت، یکی بیشتر از بزرگترین ارتفاع بین دو زیردرخت می‌باشد، ارتفاع نمایی درخت دودویی، دو برابر ارتفاعهای نمایی دو زیر درخت ریشه می‌باشد. اگر بدانیم که $R_n=i$ است آنگاه داریم

$Y_n=2.max\ (Y_{i-1}\ ,\ Y_{n-i})$

به عنوان حالت‌های پایه داریم $Y_1=1$، چون ارتفاع نمایی درخت با یک گره $2^0=1$ است و برای راحتی کار $Y_0 = 0$ تعریف می‌کنیم.

سپس متغیرهای تصادفی شاخص $Z_{n,1},\ Z_{n,2}\ \dots\ ,Z_{n,n}$ را تعریف می‌کنیم که $Z_{n,i}= I\{R_n=i\}$.
چون R_n احتمالاً یکی از عناصر $\{1, 2, . . . , n\}$ می‌باشد برای $i=1, 2, . . . n$ داریم $Pr\{R_n=i\}=1/n$
و از این‌رو بنا به لم ۵.۱ برای $i=1, 2, ... n$ داریم

$$E[Z_{n,i}] = 1/n\ , \qquad (12.1)$$

چون دقیقاً یکی از $Z_{n,i}$ برابر 1 و بقیه 0 می‌باشند، داریم

$$Y_n = \sum_{i=1}^{n} Z_{n.i}\ (2 \cdot \max(Y_{i-1}, Y_{n-i}))\ .$$

نشان خواهیم داد که $E[Y_n]$ یک چندجمله‌ای از n می‌باشد که در آخر بیان می‌کند که $E\ [X_n] = O(lgn)$

متغیر تصادفی شاخص $Z_{n,i}\ = I\{R_n= i\}$ مستقل از مقادیر Y_{i-1} است. با انتخاب $R_n= i$ زیر درخت چپ که ارتفاع نمایی آن Y_{i-1} روی $i\text{-}1$ کلید به طور تصادفی ساخته می‌شود که مرتبه آنها کمتر از i می‌باشد. این زیر درخت دقیقاً مانند هر درخت جستجوی دودویی تصادفی ساخته شده روی $i\text{-}1$ کلید دیگر می‌باشد. بجز تعداد کلیدهای درخت، ساختار درخت از انتخاب $R_n=i$ به هیچ عنوان تأثیر نمی‌پذیرد. از اینرو متغیرهای تصادفی Y_{i-1} و $Z_{n,i}$ مستقلند. همچنین زیر درخت راست با ارتفاع نمایی Y_{n-i} روی $n\text{-}i$ کلید که مرتبه آنها بزرگتر از i است به صورت تصادفی ساخته می‌شود. ساختار آن

---

1. exponential height

مستقل از مقدار R_n می‌باشد و بنابراین متغیرهای تصادفی $Z_{n,i}$, Y_{n-i} مستقلند. از اینرو

$$\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\left[\sum_{i=1}^{n} Z_{n,i}\,(2\cdot\max(Y_{i-1},Y_{n-i}))\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}\,(2\cdot\max(Y_{i-1},Y_{n-i}))] && \text{(بنا به خطی بودن انتظار)} \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}]\,\mathrm{E}[2\cdot\max(Y_{i-1},Y_{n-i})] && \text{(بنا به استقلال)} \\
&= \sum_{i=1}^{n} \frac{1}{n}\cdot\mathrm{E}[2\cdot\max(Y_{i-1},Y_{n-i})] && \text{(بنا به معادلهٔ ۱۲.۱)} \\
&= \frac{2}{n}\sum_{i=1}^{n} \mathrm{E}[\max(Y_{i-1},Y_{n-i})] \\
&\le \frac{2}{n}\sum_{i=1}^{n} (\mathrm{E}[Y_{i-1}]+\mathrm{E}[Y_{n-i}])
\end{aligned}$$

در سری آخر هر یک از عبارتهای $E[Y_0]$، $E[Y_1]$ ... و $E[Y_{n-1}]$ دو بار ظاهر می‌شوند، یکبار به عنوان $E[Y_{i-1}]$ و یکبار به عنوان $E[Y_{n-i}]$ و بنابراین معادلهٔ بازگشتی زیر را داریم

$$\mathrm{E}[Y_n] \le \frac{4}{n}\sum_{i=0}^{n-1}\mathrm{E}[Y_i]\ . \tag{۱۲.۲}$$

با استفاده از روش جایگذاری، نشان خواهیم داد که برای همه مقادیر صحیح و مثبت n رابطه بازگشتی (۱۲۰۲) جواب زیر را دارد

$$\mathrm{E}[Y_n] \le \frac{1}{4}\binom{n+3}{3}\ .$$

برای انجام این کار از تساوی

$$\sum_{i=0}^{n-1}\binom{i+3}{3} = \binom{n+3}{4}\ . \tag{۱۲.۳}$$

استفاده می‌کنیم. (تمرین ۱ – ۱۲.۴ از شما می‌خواهد که این تساوی را ثابت کنید.)

برای حالت پایه تأیید می‌کنیم که حد [illegible] برقرار است

$$1 = Y_1 = \mathrm{E}[Y_1] \le \frac{1}{4}\binom{1+3}{3} = 1$$

برای جایگذاری داریم

$$
\begin{aligned}
\mathrm{E}[Y_n] &\le \frac{4}{n}\sum_{i=0}^{n-1}\mathrm{E}[Y_i] \\
&= \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}\binom{i+3}{3} && \text{(بنا به فرض استقراء)} \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} \\
&= \frac{1}{n}\binom{n+3}{4} && \text{(بنا به تساوی (۱۲.۳))} \\
&= \frac{1}{n}\cdot\frac{(n+3)!}{4!\,(n-1)!} \\
&= \frac{1}{4}\cdot\frac{(n+3)!}{3!\,n!} \\
&= \frac{1}{4}\binom{n+3}{3}.
\end{aligned}
$$

$E[Y_n]$ را محدود کرده‌ایم ولی هدف نهایی محدود کردن $E[X_n]$ می‌باشد. همانطور که تمرین ۱۲.۴-۴ از شما می‌خواهد که نشان دهید، تابع $f(x) = 2^x$ محدب است. بنابراین می‌توانیم نا مساوی *Jensen* را بیان کنیم که می‌گوید

$$2^{\mathrm{E}[X_n]} \le \mathrm{E}[2^{X_n}] = \mathrm{E}[Y_n]\ ,$$

تا نتیجه بگیریم که:

$$
\begin{aligned}
2^{\mathrm{E}[X_n]} &\le \frac{1}{4}\binom{n+3}{3} \\
&= \frac{1}{4}\cdot\frac{(n+3)(n+2)(n+1)}{6} \\
&= \frac{n^3+6n^2+11n+6}{24}.
\end{aligned}
$$

با گرفتن لگاریتم از هر دو طرف، نتیجه $E[X_n]=O(lgn)$ بدست می‌آید. بنابراین قضیه زیر را ثابت کردیم:

قضیه ۱۲.۴

ارتفاع مورد انتظار یک درخت جستجوی دودویی تصادفی ساخته شده روی n کلید برابر $O(lgn)$ می‌باشد.

تمرین‌ها

۱- ۱۲.۴ تساوی ۱۲.۳ را ثابت کنید.

۲- ۱۲.۴ یک درخت جستجوی دودویی روی n گره بیان که میانگین عمق هر گره درخت $\Theta(lgn)$ باشد ولی ارتفاع درخت $\omega(lgn)$ می‌باشد. یک حد بالا مجانبی برای ارتفاع یک درخت جستجوی دودویی n گره‌ای که در آن میانگین عمق هر گره $\Theta(lgn)$ می‌باشد ارائه نمایید.

۳- ۱۲.۴ نشان دهید مفهوم درخت جستجوی دودویی تصادفی انتخاب شده روی n کلید که هر درخت جستجوی دودویی با n کلید احتمال برابری برای انتخاب شدن دارد با مفهوم درخت جستجوی دودویی تصادفی ساخته شده که در این بخش ارائه شده است، متفاوت است.(راهنمایی: تمام حالتهای ممکن وقتی $n=3$ است را لیست کنید.)

۴- ۱۲.۴ نشان دهید تابع $f(x) = 2^x$ محدب است.

۵- ۱۲.۴ * فرض کنید *RANDOMIZED-QUICKSORT* روی یک توالی از n عدد ورودی اجرا شود. ثابت کنید که برای هر ثابت $k>0$ همه $n!$ جایگشت‌های ورودی بجز $O(1/n^k)$، زمان اجرای $O(nlgn)$ را دارند.

مسائل

۱-۱۲ درخت جستجوی دودویی با کلیدهای مساوی

کلیدهای مساوی در پیاده سازی درخت‌های جستجوی دودویی، مشکل بوجود می‌آورند.

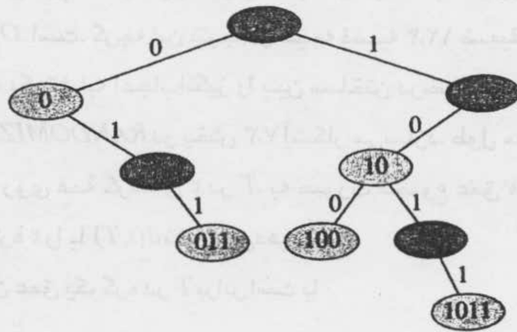
a کارایی مجانبی *TREE-INSERT* زمانیکه برای درج n داده با کلیدهای مشابه در یک درخت جستجوی دودویی بصورت اولیه خالی استفاده می‌شود چیست؟

پیشنهاد می‌کنیم که *TREE-INSERT* را با امتحان اینکه قبل از خط ۵ $Key[z] = key[x]$ و قبل از خط ۱۱ $key[z]=key[y]$ برقرار است یا خیر، بهبود بخشیم. اگر برابری حفظ شود یکی از استراتژیهای زیر را پیاده سازی می‌کنیم. برای هر استراتژی کارایی مجانبی درج n داده با کلیدهای یکسان در درخت جستجوی دودویی اولیهٔ خالی را پیدا کنید. (استراتژیها، برای خط ۵ که کلیدهای z و x را با هم مقایسه می‌کنیم شرح داده شده‌اند. y را با x جابه‌جا کنید تا به استراتژی برای خط ۱۱ برسید.)

b. یک پرچم بولی $b[x]$ برای گره x نگه دارید و x را با توجه به مقدار $b[x]$ با $left[x]$ یا $right[x]$ مقداردهی کنید که مقدار $b[x]$ هر بار که x هنگام درج یک گره با کلید مشابه x ملاقات می‌شود، بین *FALSE* و *TRUE* تغییر می‌کند.

c. یک لیست از گره‌هایی که کلید برابر با x دارند را نگه داشته و z را به لیست اضافه کنید.

d به طور تصادفی x را برابر یکی از مقدارهای $left[x]$ یا $right[x]$ قرار دهید. (کارایی را در بدترین حالت بیان کرده و از روی آن به طور غیر رسمی کارایی در حالت میانگین را نتیجه بگیرید.)



شکل ۱۲.۵ یک درخت مبنا با ذخیرهٔ رشته‌های بیتی *1011 , 10 , 011 , 100 , 0*، کلید هر گره می‌تواند از پیمودن مسیری از ریشه تا آن گره مشخص شود، بنابراین نیازی به ذخیره کلید در گره نمی‌باشد؛ و در اینجا کلیدها تنها برای توضیحات نشان داده شده‌اند. گره‌های تیره، گره‌هایی هستند که کلید متناظر با آنها در درخت وجود ندارد؛ این گره‌ها تنها برای ساختن مسیر به بقیه گره‌ها نشان داده شده‌اند.

۱۲-۲ درخت مبنا [1]

دو رشته $a=a_0a_1 \ldots a_p$ و $b=b_0b_1 \ldots b_q$ داده شده‌اند که در آن هر a_i و b_j در یک مجموعه مرتب از کاراکترها قرار دارد. می‌گوئیم رشته a بطور لغتنامه‌ای [2] از b کوچکتر یا بزرگتر است.

۱. یک مقدار صحیح j بطوریکه $0 \le j \le min(p,q)$ وجود داشته باشد که برای همه مقادیر $i=0, 1, \ldots, j-1$، $a_j<b_j$، $a_i=b_i$، یا

۲. $p<q$ و برای مقادیر $i=0, 1, \ldots, p$ داریم $a_i = b_i$.

برای مثال اگر a و b رشته‌های بیتی باشند آنگاه بنابر قانون ۱، $10100<10110$ است (فرض کنید $j=3$ است.) و بنابر قانون ۲ ، $10100<101000$ است. این کار شبیه مرتب سازی در لغت نامه‌های انگلیسی است.

ساختمان داده *درخت مبنا* که در شکل ۱۲.۵ نشان داده شده است رشته‌های بیتی *0 , 100 , 011 , 10 , 1011* را ذخیره می‌کند زمانیکه یک کلید $a=a_0 \, a_1 \ldots a_p$ را جستجو می‌کنیم، اگر $a_i= 0$ باشد به گره‌ای با عمق i در سمت چپ، و اگر $a_i=1$ باشد به گره‌ای با عمق i در سمت راست می‌رویم. فرض کنید S مجموعه‌ای از رشته‌های دودویی مجزا است که طول آنها در مجموع n می‌باشد. نشان دهید چطور می‌توان با استفاده از درخت مبنا، S را بصورت لغتنامه‌ای در زمان $\Theta(n)$ مرتب کرد. برای مثال شکل ۱۲.۵ خروجی مرتب باید توالی *0 , 011 , 10 , 100 , 1011* باشد.

---
1. radix tree
2. lexicographically less than

۳-۱۲ میانگین عمق گره در یک درخت جستجوی دودویی تصادفی ساخته شده

در این مسئله، ثابت می‌کنیم که میانگین عمق یک گره در درخت جستجوی دودویی تصادفی ساخته شده با n گره، $O(lgn)$ است. گرچه این نتیجه از نتیجه قضیه ۱۲.۴ ضعیف‌تر است، تکنیکی که از آن استفاده خواهیم کرد، یک تشابه اعجاب‌انگیز را بین ساختن درخت جستجوی دودویی و اجرا ی *RANDOMIZED-QUICKSORT* در بخش ۷.۳ آشکار می‌سازد. طول مسیر کلی، یعنی $P(T)$ برای درخت دودویی T را، روی همهٔ گره‌های x در T، به صورت مجموع عمق همه گره‌های x در T تعریف می‌کنیم که عمق هر گره x را با $d(x,T)$ نشان می‌دهیم.

a. نشان دهید میانگین عمق یک گره در T برابر است با

$$\frac{1}{n}\sum_{x\in T} d(x,T) = \frac{1}{n}P(T) .$$

بنابراین، می‌خواهیم نشان دهیم مقدار مورد انتظار $P(T)$ برابر $O(nlgn)$ است.

b. فرض کنید T_L و T_R به ترتیب بر زیردرخت چپ و زیردرخت راست درخت T دلالت می‌کند. ثابت کنید اگر T دارای n گره باشد آنگاه

$$P(T) = P(T_L) + P(T_R) + n - 1 .$$

c. فرض کنید $P(n)$ میانگین طول مسیر کل یک درخت جستجوی دودویی تصادفی ساخته شده با n گره را مشخص می‌کند. نشان دهید

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1) .$$

d. نشان دهید که $P(n)$ می‌تواند به صورت زیر باز نویسی شود

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n) .$$

e. با یاد آوری تحلیل دیگر صورت تصادفی مرتب سازی سریع که در مسئله ۲-۷ ارائه شده است، نتیجه بگیرید که $P(n)= O(n\ lgn)$.

در هر فراخوانی بازگشتی مرتب سازی سریع، یک عنصر محوری تصادفی انتخاب می‌کنیم تا مجموعه عناصر مرتب شده را تقسیم کنیم. هر گره درخت جستجوی دودویی، مجموعه عناصر را طوری تقسیم بندی می‌کند که در زیر درخت مشتق شده از آن گره قرار گیرند.

f. یک پیاده سازی برای مرتب سازی سریع بیان کنید که در آن، مقایسه‌ها برای مرتب کردن مجموعه عناصر، دقیقاً مشابه مقایسه‌هایی باشد که برای درج عناصر در درخت جستجوی دودویی بکار می‌رود (ترتیب مقایسه‌ها ممکن است متفاوت باشد، ولی همان مقایسه‌ها باید انجام شود.)

۴-۱۲ تعداد درخت‌های دودویی متفاوت

فرض کنید b_n تعداد درخت‌های دودویی متفاوت با n گره را نشان می‌دهد. در این مسئله فرمولی برای b_n و همچنین یک تخمین مجانبی خواهید یافت.

a نشان دهید $b_0=1$ و برای $n\le 1$ داریم

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} \ .$$

b.با مراجعه به مسئله ۵-۴ برای تعریف یک تابع مولد، فرض کنید $B(x)$ تابع مولد زیر باشد

$$B(x) = \sum_{n=0}^{\infty} b_n x^n \ .$$

نشان دهید $B(x)=xB(x)^2+1$ و از اینرو یک روش برای تعریف $B(x)$ به فرم بسته زیر است

$$B(x) = \frac{1}{2x}\left(1-\sqrt{1-4x}\right) \ .$$

بسط تیلور[1] $f(x)$ حول نقطه $x=a$ بصورت

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k \ ,$$

است که $f^{(k)}(x)$ مشتق kام f نسبت به x است.

c.نشان دهید که

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(nامین عدد Catalan) از بسط تیلور $\sqrt{1-4x}$ حول نقطهٔ $x=0$ استفاده کنید. (می‌توانید به جای استفاده از بسط تیلور از تعمیم بسط دوجمله‌ای

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

برای توانهای غیرانتگرالی n استفاده کنید، که برای هر عدد حقیقی n و هر عدد صحیح k، $\binom{n}{k}$ را بصورت $n(n-1)\ ...\ (n+k+1)/k!$ اگر $k\ge 0$ و در غیر اینصورت 0 تفسیر می‌کنیم.)

d.نشان دهید که

$$b_n = \frac{4^n}{\sqrt{\pi}\, n^{3/2}}\left(1+O(1/n)\right) \ .$$

1.Taylor expansion

۱۳ درخت‌های قرمز – سیاه[1]

فصل ۱۲ نشان داد که یک درخت جستجوی دودویی با ارتفاع h می‌تواند هر یک از اعمال مجموعه پویا مانند *MAXIMUM*، *SUCCESSOR*، *PREDECESSOR*، *SEARCH*، *DELETE*، *MINIMUM* و *INSERT* را در زمان $O(n)$ پیاده سازی کند. بنابراین اگر ارتفاع درخت جستجو کوچک باشد اعمال مجموعه سریع‌تر انجام می‌شوند؛ ولی اگر ارتفاع آن زیاد باشد کارایی آنها ممکن است بهتر از کارایی لیست پیوندی نباشد. درخت‌های قرمز– سیاه یکی از طرح‌های درخت جستجو می‌باشند که به منظور تضمین اینکه اعمال اصلی مجموعه‌های پویا، زمان اجرای $O(lgn)$ را در بدترین حالت دارند، "موازنه" می‌شوند.

۱۳.۱ ویژگیهای درخت‌های قرمز – سیاه

یک **درخت قرمز– سیاه** یک درخت جستجوی دودویی با یک بیت حافظه اضافی برای هر گره x می‌باشد: رنگ گره‌که می‌تواند *RED* یا *BLACK* باشد. با محدود کردن روشی که گره‌ها در هر مسیر از ریشه تا برگ می‌توانند رنگ شوند، درخت‌های قرمز– سیاه تضمین می‌کنند که هیچ مسیری وجود ندارد که بیشتر از 2 واحد از بقیه مسیرها بزرگتر باشد، بنابراین درخت تقریباً *متوازن*[2] است.

اکنون هر گره درخت شامل فیلدهای *color* ، *key* ، *left* ، *right* و p می‌باشد. اگر فرزند یا پدر یک گره وجود نداشته باشد اشاره‌گر فیلد مربوطه مقدار *NIL* می‌گیرد. بنابراین *NIL*ها را به عنوان اشاره‌گرهایی به گره‌های خارجی (برگ‌های) درخت جستجو دودویی درنظر می‌گیریم و گره‌های معمولی دربردارنده کلیدها را به عنوان گره‌های داخلی درخت در نظر می‌گیریم.

ویژگی‌ها

یک درخت جستجوی دودویی یک درخت قرمز – سیاه است، اگر در ویژگی‌های قرمز– سیاه[3] زیر صدق

---

1. Red-Black
2. balanced
3. red-black properties

کند:

۱. هر گره، قرمز یا سیاه است.

۲. ریشه سیاه است.

۳. هر برگی (*NIL*) سیاه است.

۴. اگر یک گره قرمز باشد آنگاه هر دو فرزندش سیاه هستند.

۵. برای هر گره، همه مسیرها از آن گره به برگهایی که از نواده‌اش هستند شامل تعداد یکسانی گره سیاه هستند.

شکل (*a*)۱۳.۱ یک مثال از درخت قرمز - سیاه را نشان می‌دهد.

به منظور راحتی در رابطه با شرایط مرزی در برنامه، از یک محافظ برای نمایش *NIL* استفاده می‌کنیم (به بخش ۱۰.۲ مراجعه کنید). برای یک درخت قرمز - سیاه *T*، محافظ *nil[T]* یک شیء با فیلدهای مشابه یک گره معمولی در درخت می‌باشد. فیلد *color* آن *BLACK* است و بقیه فیلدها - *right*, *left* , *key* , *color* و *p*- می‌توانند مقادیر اختیاری داشته باشند. همان طور که شکل (*b*)۱۳.۱ نشان می‌دهد، همه اشاره‌گرها که به *NIL* اشاره می‌کنند با اشاره گرهایی که به محافظ *nil[T]* اشاره می‌کنند جایگزین می‌شوند.

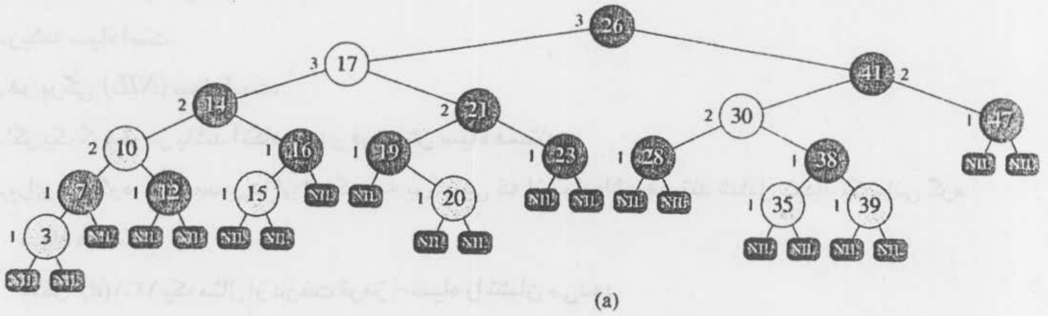
برای اینکه بتوانیم با فرزند *NIL* یک گره x به عنوان گره معمولی رفتار کنیم که پدرش x است، از محافظ استفاده می‌کنیم. گر چه در عوض می‌توانیم یک گره محافظ جدا، برای هر *NIL* در درخت اضافه کنیم تا پدر هر *NIL* خوش تعریف شود که در اینصورت فضای اضافی استفاده می‌شود. بجای آن از یک محافظ *nil[T]* برای نشان دادن *NIL*ها- همه برگ‌ها و پدر ریشه - استفاده می‌کنیم. مقادیر فیلدهای *right* , *left* , *key* , *color* و *p* محافظ بی‌اهمیت هستند. اگر چه برای راحتی کار ممکن است در طول یک دوره از روال آنها را، مقداردهی کنیم.

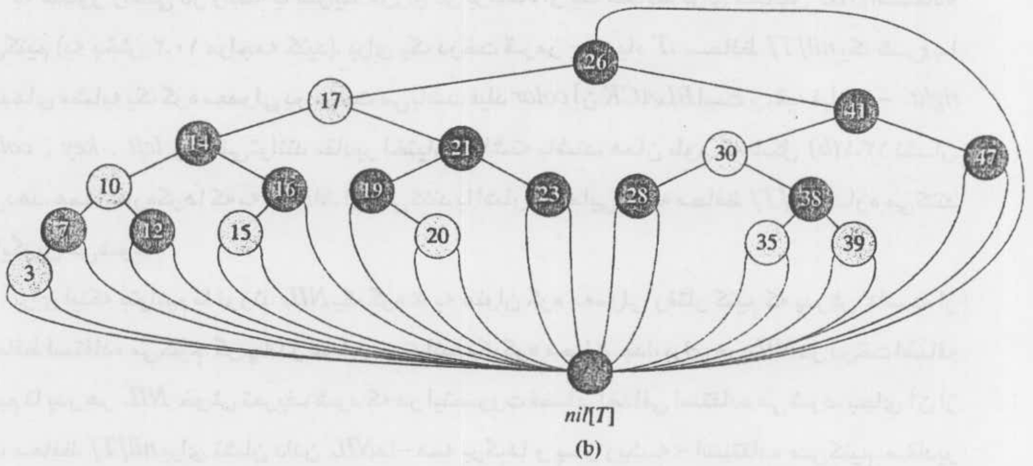
معمولاً توجه خود را معطوف به گره‌های داخلی یک درخت قرمز - سیاه می‌کنیم چون آنها مقادیر کلیدها رانگه می‌دارند. در ادامه این فصل زمانی که درخت‌های قرمز - سیاه را رسم می‌کنیم برگ‌ها را حذف می‌کنیم، همان طور که در شکل (*c*)۱۳.۱ نشان داده شده است.

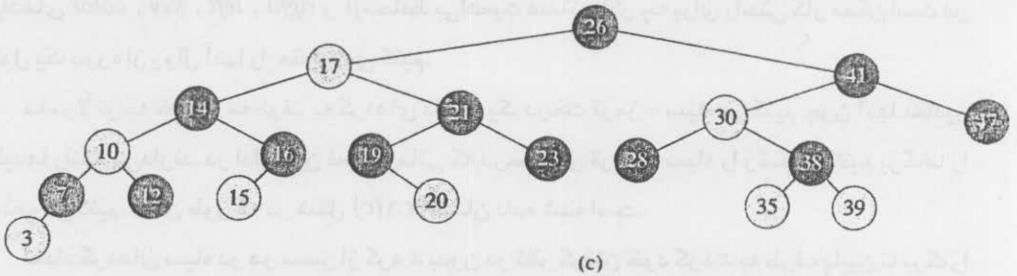
تعداد گره‌های سیاه در هر مسیر از گره x بدون در نظر گرفتن خود گره x به طرف پایین تا برگ را *ارتفاع سیاه*[1] یک گره می‌گوییم، که با $bh(x)$ مشخص می‌کنیم. بنا به ویژگی ۵، مفهوم ارتفاع سیاه خوش تعریف است چون همه میسرهای رو به پایین از یک گره، تعداد یکسانی گره سیاه دارند. ارتفاع سیاه یک درخت قرمز- سیاه را ارتفاع سیاه ریشه آن درخت تعریف می‌کنیم.

لم زیر نشان می‌دهد که چرا درخت‌های قرمز - سیاه درخت‌های جستجوی خوبی را می‌سازند.

1.black-height







شکل ۱۳.۱ یک درخت قرمز - سیاه با گره‌های سیاه تیره شده و گره‌های قرمز سیاه زده شده. هر گره در یک درخت قرمز - سیاه، یا قرمز است یا سیاه، فرزندان هر گره قرمز سیاه می‌باشند و هر مسیر ساده از یک گره به یک برگ نوه شامل تعداد مساوی گره‌های سیاه است. *(a)* هر برگی که به صورت *NIL*ها نشان داده شده سیاه می‌باشد. هر گره غیر *NIL* با ارتفاع سیاهش علامت گذاری شده است؛ *NIL*ها دارای ارتفاع سیاه *0* می‌باشند. *(b)* همان درخت قرمز - سیاه ولی *NIL*ها با یک محافظ *nil[T]* که همیشه سیاه است جایگزین شده‌اند و ارتفاع‌های سیاه حذف شده‌اند. پدر ریشه نیز محافظ می‌باشد. *(c)* همان درخت قرمز - سیاه ولی با حذف کامل برگ‌ها و پدر ریشه، از این روش رسم در ادامه این فصل استفاده خواهیم کرد.

لم ۱۳.۱

یک درخت قرمز – سیاه با n گره داخلی دارای حداکثر ارتفاع $2lg(n+1)$ است.

اثبات با نشان دادن اینکه زیردرخت مشتق شده از هر گره x حداقل شامل $2^{bh(x)}-1$ گره داخلی می‌باشد شروع می‌کنیم. این ادعا را با استقرار روی ارتفاع x اثبات می‌کنیم. اگر ارتفاع x برابر 0 باشد آنگاه x باید یک برگ $(nil[T])$ باشد، و زیردرخت مشتق شده از x در واقع شامل حداقل $2^{bh(x)}-1 = 2^0-1=0$ گره داخلی می‌باشد. برای گام استقراء، گره x را در نظر بگیرید که یک گره داخلی با دو فرزند و ارتفاع مثبت می‌باشد. هر فرزند دارای ارتفاع سیاه $bh(x)$ یا $bh(x)-1$ است، بسته به رنگ آن که به ترتیب قرمز یا سیاه باشد. چون ارتفاع یک فرزند x کوچکتر از ارتفاع خود x می‌باشد می‌توانیم از فرض استقراء نتیجه بگیریم که هر فرزند حداقل $2^{bh(x)}-1$ گره داخلی دارد. بنابراین زیردرخت مشتق شده از x شامل حداقل $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1=2^{bh(x)}-1$ گره داخلی می‌باشد که ادعای ما را ثابت می‌کند.

برای کامل کردن اثبات لم، فرض می‌کنیم h، ارتفاع درخت باشد. با توجه به ویژگی ۴ حداقل نصف گره‌های هر مسیر ساده‌ای از ریشه تا برگ، بدون محاسبه ریشه، باید سیاه باشند. از اینرو ارتفاع سیاه ریشه باید حداقل $h/2$ باشد. بنابراین $n\geq 2^{h/2}-1$.

با انتقال 1 به سمت چپ نامساوی و گرفتن لگاریتم از دو طرف داریم $lg(n+1)\geq h/2$ یا $h\leq 2lg(n+1)$. ■

یک نتیجه‌گیری سریع از این لم این است که عملیات $PREDECESSOR$, $SUCCESSOR$, $MAXIMUM$, $MINIMUM$, $SEARCH$ در مجموعه‌های پویا می‌توانند در زمان $O(lgn)$ روی درخت‌های قرمز - سیاه پیاده‌سازی شوند، چون آنها می‌توانند در زمان $O(h)$ روی درخت‌های جستجو با ارتفاع h اجرا شوند (همانطور که در فصل ۱۲ نشان داده شد) و هر درخت قرمز– سیاه روی n گره، یک درخت جستجو با ارتفاع $O(lgn)$ می‌باشد. (البته ارجاعها به NIL در الگوریتم‌های فصل ۱۲ با $nil[T]$ جابه‌جا شوند.) اگر چه الگوریتم‌های $TREE\text{-}INSERT$ و $TREE\text{-}DELETE$ در فصل ۱۲، با دریافت یک درخت قرمز – سیاه به عنوان ورودی، در زمان $O(lgn)$ اجرا می‌شوند، مستقیماً اعمال $INSERT$ و $DELETE$ مجموعه‌های پویا را پشتیبانی نمی‌کنند چون نمی‌توانند تضمین کنند که درخت جستجوی دودویی تغییر یافته یک درخت قرمز – سیاه خواهد بود. در بخش ۱۳.۵ و ۱۳.۴ خواهیم دید که این دو عمل می‌توانند در واقع در زمان $O(lgn)$ پشتیبانی شوند.

تمرین‌ها

۱-۱۳.۱ به روش شکل ۱۳.۱(a) درخت جستجوی دودویی کامل با ارتفاع 3 روی کلیدهای $\{1, 2, .., 15\}$ را رسم کنید. برگ‌های NIL را اضافه کرده و گره‌ها را به سه روش مختلف رنگ‌آمیزی کنید تا ارتفاع‌های سیاه درخت‌های قرمز – سیاه بدست آمده $3,2$ و 4 باشد.

۱۳.۱-۲ درخت قرمز - سیاهی رسم کنید که پس از فراخوانی *TREE-INSERT* روی درخت شکل ۱۳.۱ با کلید 36 حاصل آید. اگر گره اضافه شده قرمز رنگ شود آیا درخت به دست آمده درخت قرمز - سیاه است؟ اگر سیاه شود چطور؟

۱۳.۱-۳ فرض کنید یک درخت *قرمز - سیاه آرام*[۱] به عنوان یک درخت جستجوی دودویی که ویژگی‌های ۱، ۳، ۴ و ۵ را حفظ می‌کند تعریف می‌کنیم. به بیان دیگر ریشه ممکن است قرمز یا سیاه باشد. یک درخت قرمز - سیاه آرام T در نظر بگیرید که ریشه آن قرمز است.اگر ریشه T را رنگ سیاه بزنیم و هیچ تغییر دیگری در T انجام ندهیم، آیا درخت حاصل یک درخت قرمز - سیاه است؟

۱۳.۱-۴ فرض کنید که هر گره قرمز در درخت قرمز - سیاه را در پدر سیاهش جذب کنیم تا این فرزندان گره قرمز، فرزندان پدر سیاه رنگ شوند. (از اتفاقی که برای کلیدها می‌افتد چشمپوشی کنید.) درجه‌های ممکن برای هر گره سیاه، بعد از اینکه همه فرزندان قرمز رنگش جذب می‌شوند چیست؟ در مورد عمق برگهای درخت حاصل چه می‌توان گفت؟

۱۳.۱-۵ نشان دهید که طولانی‌ترین مسیر ساده از گره x در درخت قرمز - سیاه به برگی که از نواده آن است، دارای طول حداکثر دو برابر کوتاهترین مسیر ساده از گره x به یک برگ از نواده آن می‌باشد.

۱۳.۱-۶ بزرگترین عدد ممکن برای گره‌های داخلی در یک درخت قرمز - سیاه با ارتفاع سیاه k چیست؟ کوچکترین عدد ممکن چیست؟

۱۳.۱-۷ یک درخت قرمز - سیاه با n کلید طوری تعریف کنید که بزرگترین نسبت ممکن گره‌های داخلی قرمز رنگ به گره‌های داخلی سیاه رنگ را داشته باشد. این نسبت چیست؟ چه درختی کوچکترین نسبت را دارد و این نسبت چیست؟

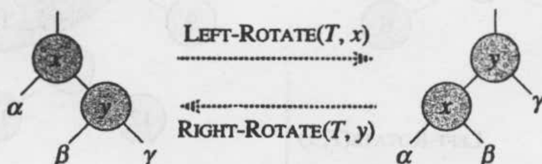
۱۳.۲ چرخش‌ها

زمانی که عملیات *TREE-INSERT* و *TREE-DELETE* روی یک درخت قرمز - سیاه با n کلید اجرا می‌شوند، زمان $O(lgn)$ را صرف می‌کنند. چون این اعمال درخت را تغییر می‌دهند، نتیجه ممکن است از ویژگیهای قرمز - سیاه که در بخش ۱۳.۱ برشمرده شد، تخطی کند. برای بازیابی ویژگی‌ها باید رنگ بعضی گره‌ها در درخت و همچنین ساختار اشاره‌گرها را عوض کنیم.

ساختار اشاره‌گرها را در طول **چرخش**[۲] عوض می‌کنیم که یک عمل محلی در یک درخت جستجو برای حفظ ویژگی درخت جستجوی دودویی می‌باشد. شکل ۱۳.۲ دو نوع چرخش، چرخش به چپ و چرخش به راست را نشان می‌دهد. زمانیکه یک چرخش به چپ روی یک گره x انجام می‌دهیم فرض می‌کنیم که فرزند راست آن y مخالف $nil[T]$ باشد؛ x می‌تواند هر گره‌ای در درخت که فرزند راستش

1.relaxed red - black tree　　2.rotation

مخالف $nil[T]$ است باشد. چرخش به چپ، حول پیوند x به y می‌چرخد. این چرخش، y را ریشه جدید زیر درخت قرار می‌دهد، همراه با x به عنوان فرزند چپ y و فرزند چپ y را به عنوان فرزند راست x.

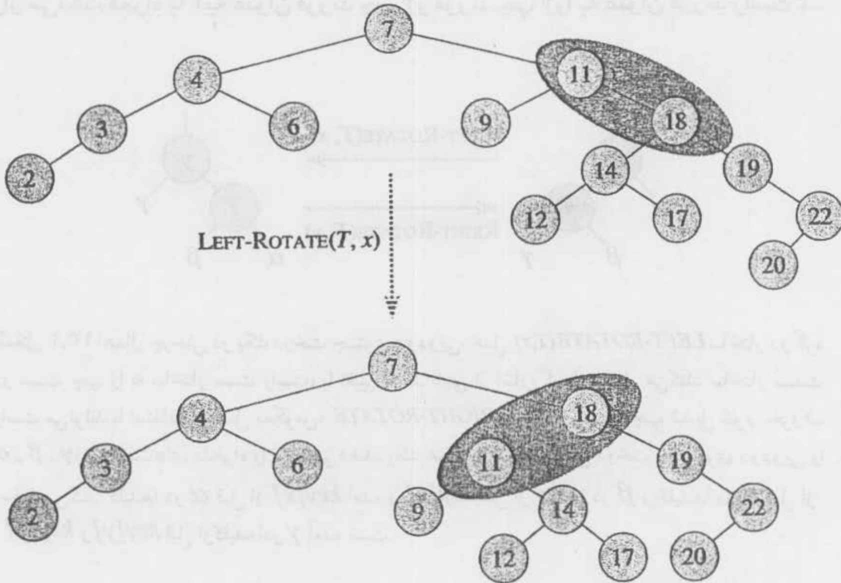


شکل ۱۳.۲ اعمال چرخش در یک درخت جستوی دودویی. عمل $LEFT\text{-}ROTATE(T,x)$ ساختار دو گره در سمت چپ را به ساختار سمت راست، با تغییر تعداد ثابتی از اشاره‌گرها، تبدیل می‌کند. ساختار سمت راست می‌تواند با استفاده از عمل معکوس، $RIGHT\text{-}ROTATE$ به ساختار سمت چپ تبدیل شود. حروف α , β , γ زیردرخت‌های دلخواه را نشان می‌دهند. یک عمل چرخش، ویژگی درخت جستجوی دودویی را حفظ می‌کند: کلیدها در α قبل از $key[x]$ آمده و $key[x]$ قبل از کلیدها در β و کلیدها در β قبل از $key[y]$ و $key[y]$ قبل ازکلیدهادر γ آمده است.

شبه کد $LEFT\text{-}ROTATE$ فرض می‌کند که $right[x] \neq nil[T]$ و پدر ریشه $nil[T]$ می‌باشد.

```
LEFT-ROTATE(T, x)
 1  y ← right[x]                ▷ Set y.
 2  right[x] ← left[y]          ▷ Turn y's left subtree into x's right subtree.
 3  p[left[y]] ← x
 4  p[y] ← p[x]                 ▷ Link x's parent to y.
 5  if p[x] = nil[T]
 6     then root[T] ← y
 7     else if x = left[p[x]]
 8             then left[p[x]] ← y
 9             else right[p[x]] ← y
10  left[y] ← x                 ▷ Put x on y's left.
11  p[x] ← y
```

شکل ۱۳.۳ نشان می‌دهد که $LEFT\text{-}ROTATE$ چطور عمل می‌کند. کد برای $RIGHT\text{-}ROTATE$ متقارن با $LEFT\text{-}ROTATE$ است. هر دو عمل $LEFT\text{-}ROTATE$ و $RIGHT\text{-}ROTATE$ در زمان $O(1)$ انجام می‌شود. تنها اشاره‌گرها با چرخش عوض می‌شوند؛ بقیه فیلدها همگی در یک گره همان‌طور باقی می‌مانند.



شکل ۱۳.۳ مثالی برای این که روال *LEFT-ROTATE(T,x)* یک درخت جستجوی دودویی را تغییر می‌دهد. پیمایش میان ترتیب درخت، هم برای درخت ورودی و هم درخت تغییر یافته، لیست مشابهی از مقادیر کلیدها را ایجاد می‌کند.

تمرین‌ها

۱۳.۲-۱ شبه کدی برای *RIGHT-ROTATE* بنویسید.

۱۳.۲-۲ ثابت کنید که در هر درخت جستجوی دودویی n گره‌ای، دقیقاً $n-1$ چرخش ممکن، وجود دارد.

۱۳.۲-۳ در درخت سمت چپ شکل ۱۳.۲، a ، b و c را به ترتیب در زیردرخت‌های α، β و γ گره‌های اختیاری در نظر بگیرید. زمانی که یک چرخش به چپ روی گره x در شکل انجام شود عمق a، b و c چطور تغییر می‌کند.

۱۳.۲-۴ نشان دهید که هر درخت جستجوی دودویی n گره‌ای اختیاری، می‌تواند با استفاده از $O(n)$ چرخش به یک درخت جستجوی دودویی n گره‌ای اختیاری دیگر تبدیل شود. (راهنمایی: ابتدا نشان دهید که حداکثر $n-1$ چرخش به راست کافی است تا درخت به یک زنجیره راست گرد تبدیل شود.)

* ۱۳.۲-۵ می‌گوییم درخت جستجوی دودویی T_1 قابل تبدیل از راست به درخت جستجوی دودویی T_2 است اگر از طریق یک سری فراخوانی‌های *RIGHT-ROTATE* امکان این که T_1 به T_2 تبدیل شود وجود داشته باشد. مثالی از دو درخت T_1 و T_2 بیان کنید که T_1 قابل تبدیل از راست به T_2 نباشد،

سپس نشان دهید اگر درخت T_1 بتواند از راست به T_2 تبدیل شود، با استفاده از $O(n^2)$ فراخوانی تابع *RIGHT-ROTATE* تبدیل می‌شود.

۱۳.۳ درج

درج یک گره در یک درخت قرمز – سیاه n گره‌ای، می‌تواند در زمان $O(lgn)$ انجام شود. برای درج گره z در درخت T، که یک درخت جستجوی دودویی اختیاری است، از یک نسخه تابع *TREE-INSERT* (بخش ۱۲.۳) با اندکی تغییر استفاده می‌کنیم، و سپس z را به رنگ قرمز درمی‌آوریم. سپس برای تضمین این که ویژگی‌های قرمز– سیاه حفظ شوند، تابع کمکی *RB-INSERT-FIXUP* را برای رنگ‌آمیزی دوباره گره‌ها و انجام چرخش‌ها فراخوانی می‌کنیم. فراخوانی *RB-INSERT(T,Z)* گره z را که فرض می‌کنیم فیلد کلیدش قبلاً پر شده است در درخت قرمز – سیاه T درج می‌کند.

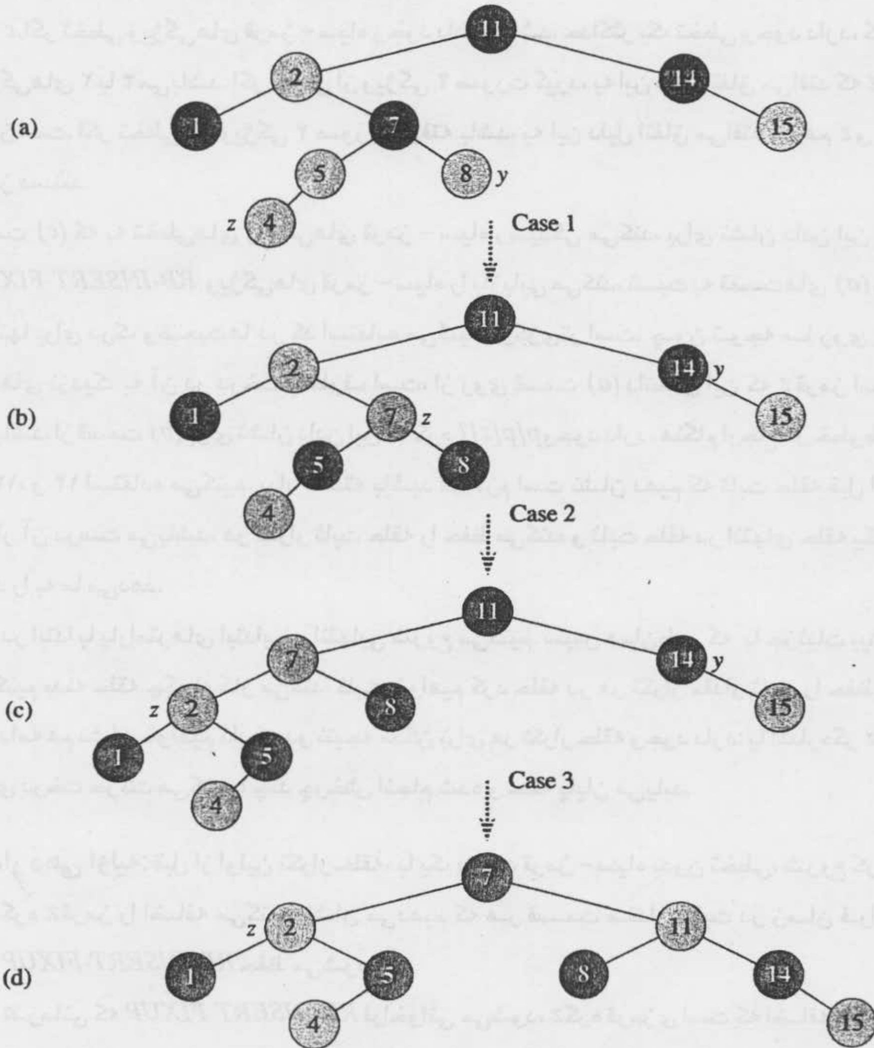
```
RB-INSERT(T, z)
 1  y ← nil[T]
 2  x ← root[T]
 3  while x ≠ nil[T]
 4       do y ← x
 5          if key[z] < key[x]
 6             then x ← left[x]
 7             else  x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then root[T] ← z
11     else  if key[z] < key[y]
12             then left[y] ← z
13             else  right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
17  RB-INSERT-FIXUP(T, z)
```

چهار تفاوت بین روال‌های *TREE-INLERT* و *RB-INSERT* وجود دارد. اول این که همه موارد *NIL* در *TREE-INLERT* با *nil[T]* جابه‌جا می‌شوند. دوم این که برای حفظ ساختار مناسب درخت، در خطوط ۱۵-۱۴ تابع *RB-INSERT*، *left[z]* و *right[z]* را برابر *nil[T]* قرار می‌دهیم. سوم این که در خط ۱۶، z را رنگ قرمز می‌زنیم، و چهارم این که چون رنگ قرمز زدن z ممکن است تخطی در یکی از ویژگی‌های قرمز – سیاه به وجود آورد، در خط ۱۷ روال *RB-INSERT*، *RB-INSERT-FIXUP(T,z)* را برای بازیابی ویژگی‌های قرمز – سیاه فراخوانی می‌کنیم.

```
RB-INSERT-FIXUP(T, z)
 1  while color[p[z]] = RED
 2      do if p[z] = left[p[p[z]]]
 3            then y ← right[p[p[z]]]
 4                 if color[y] = RED
 5                    then color[p[z]] ← BLACK                    ▷ Case 1
 6                         color[y] ← BLACK                       ▷ Case 1
 7                         color[p[p[z]]] ← RED                   ▷ Case 1
 8                         z ← p[p[z]]                            ▷ Case 1
 9                    else if z = right[p[z]]
10                            then z ← p[z]                       ▷ Case 2
11                                 LEFT-ROTATE(T, z)              ▷ Case 2
12                         color[p[z]] ← BLACK                    ▷ Case 3
13                         color[p[p[z]]] ← RED                   ▷ Case 3
14                         RIGHT-ROTATE(T, p[p[z]])               ▷ Case 3
15            else (same as then clause
                         with "right" and "left" exchanged)
16  color[root[T]] ← BLACK
```

برای درک چگونگی عملکرد روال *RB-INSERT-FIXUP*، بررسی برنامه را به سه گام مهم تقسیم می‌کنیم. اول مشخص می‌کنیم هنگامی که گره z درج می‌شود و به رنگ قرمز درمی‌آید، چه تخطی‌هایی از ویژگی‌های قرمز – سیاه در *RB-INSERT* به وجود می‌آید. دوم هدف کلی حلقه *while* را در خطوط ۱۵-۱ بررسی می‌کنیم. در نهایت هر یک از سه حالتی[1] که حلقه *while* شکسته می‌شود را پیدا می‌کنیم و می‌بینیم که چطور آنها به هدف می‌رسند. شکل ۱۳.۴ نشان می‌دهد که چطور *RB-INSERT-FIXUP* روی یک درخت قرمز – سیاه نمونه عمل می‌کند. کدام ویژگی‌های قرمز سیاه بافراخوانی *RB-INSERT-FIXUP* ممکن است مورد تخطی قرار گیرند؟ چون هر دو فرزند گره جدید قرمز اضافه شده، محافظ *nil[T]* می‌باشند مطمئناً ویژگی ۱ و همین طور ویژگی ۳ حفظ می‌شوند، ویژگی ۵ هم، که می‌گوید در هر مسیری از یک گره داده شده تعداد گره‌های سیاه یکسان است، حفظ می‌شود. زیرا گره z محافظ (سیاه) را جابه‌جا می‌کند و گره z دارای دو فرزند محافظ و همچنین رنگ قرمز است. بنابراین تنها ویژگیهایی که ممکن است مورد تخطی قرار گیرند، ویژگی ۲ که ریشه را سیاه می‌کند و ویژگی ۴ که می‌گوید یک گره قرمز نمی‌تواند یک فرزند قرمز داشته باشد هستند. هر دو تخطی ممکن، به رنگ قرمز z مربوط می‌شوند. اگر z ریشه باشد ویژگی ۲ و اگر پدر z قرمز باشد ویژگی ۴ مورد تخطی قرار می‌گیرند. شکل ۱۳.۴(a) یک تخطی ویژگی ۴ بعد از درج گره z را نشان می‌دهد. حلقه *while* در خطوط ۱۵-۱ سه قسمت ثابت زیر را نگه می‌دارد:

۱- حالت دوم روی حالت سوم می‌افتد و بنابراین این دو حالت متقابلاً منحصر نیستند.



شکل ۱۳.۴ عملکرد *RB-INSERT-FIXUP* *(a)* گره z بعد از درج. چون هم z و هم پدرش $p[z]$ قرمز هستند، تخطی ویژگی ۴ اتفاق می‌افتد. چون عموی z یعنی y قرمز است حالت ۱ در برنامه می‌تواند بکار رود. گره‌ها دوباره رنگ‌آمیزی شده و اشاره‌گر z به بالای درخت حرکت می‌کند که درخت نشان داده شده در *(b)* حاصل می‌آید. یکبار دیگر z و پدرش هر دو قرمز هستند، ولی عموی z یعنی y سیاه است. چون z فرزند راست $p[z]$ می‌باشد، حالت ۲ می‌تواند اعمال شود. یک چرخش به چپ انجام شده و درختی که حاصل می‌شود در *(c)* نشان داده شده است. اکنون z فرزند چپ پدرش می‌باشد و حالت ۳ می‌تواند انجام شود. یک چرخش به راست، درخت *(d)* را نتیجه می‌دهد که یک درخت قرمز ـ سیاه مجاز است.

در ابتدای هر تکرار حلقه،

a. گره z قرمز است.

b. اگر $p[z]$ ریشه باشد آنگاه $p[z]$ سیاه است.

c. اگر تخطی ویژگی‌های قرمز – سیاه وجود داشته باشد، حداکثر یک تخطی وجود دارد، که یکی از ویژگی‌های ۲ یا ۴ می‌باشد. اگر تخطی از ویژگی ۲ صورت گیرد، به این دلیل اتفاق می‌افتد که z ریشه و قرمز است. اگر تخطی در ویژگی ۴ صورت گرفته باشد، به این دلیل اتفاق می‌افتد که هم z و هم $p[z]$ قرمز هستند.

قسمت (c) که به تخطی‌های ویژگی‌های قرمز – سیاه رسیدگی می‌کند، برای نشان دادن این که روال *RB-INSERT-FIXUP* ویژگی‌های قرمز – سیاه را بازیابی می‌کند، نسبت به قسمت‌های (a) و (b) که از آنها برای درک وضعیت‌ها در کد استفاده می‌کنیم مرکزی‌تر است. چون توجه ما روی گره z و گره‌های نزدیک به آن در درخت معطوف است، از روی قسمت (a) دانستن این که z قرمز است، مفید می‌باشد. از قسمت (b) برای نشان دادن این که گره $p[p[z]]$ وجود دارد، هنگام ارجاع در خطوط ۲، ۳، ۷، ۸، ۱۳، و ۱۴ استفاده می‌کنیم. بیاد داشته باشید که لازم است نشان دهیم که ثابت حلقه قبل از اولین تکرار آن درست می‌باشد، هر تکرار ثابت حلقه را حفظ می‌کند، و ثابت حلقه در انتهای حلقه یک ویژگی مفید را به ما می‌دهد.

در ابتدا با پارامترهای ابتدایی و انتهایی شروع می‌کنیم. سپس همان‌طور که با جزئیات بیشتر چک می‌کنیم بدنه حلقه چگونه کار می‌کند، ثابت خواهیم کرد حلقه در هر تکرار مقدار ثابت را حفظ می‌کند. در ادامه هم نشان خواهیم داد که دو نتیجه ممکن برای هر تکرار حلقه وجود دارد: یا اشاره‌گر z به طرف بالای درخت حرکت می‌کند یا چند چرخش انجام شده و حلقه پایان می‌یابد.

مقدار دهی اولیه: قبل از اولین تکرار حلقه، با یک درخت قرمز – سیاه بدون تخطی، شروع کرده و یک گره z قرمز را اضافه می‌کنیم. نشان می‌دهیم که هر قسمت مقدار ثابت در زمان فراخوانی *RB-INSERT-FIXUP* حفظ می‌شود.

a. زمانی که *RB-INSERT-FIXUP* فراخوانی می‌شود، z گره قرمزی است که اضافه شده است.

b. اگر $p[z]$ ریشه باشد آنگاه $p[z]$ با رنگ سیاه شروع شده و قبل از فراخوانی *RB-INSERT-FIXUP* تغییر نکرده است.

c. قبلاً دیدیم که ویژگیهای ۱، ۳ و ۵ هنگام فراخوانی *RB-INSERT-FIXUP* حفظ می‌شوند. اگر تخطی از ویژگی ۲ وجود داشته باشد آنگاه ریشه قرمز باید گره z باشد که جدیداً اضافه شده است و تنها گره داخلی در درخت می‌باشد. چون پدر و هر دو فرزند z محافظ سیاه رنگ می‌باشد، تخطی از ویژگی ۴ نیز وجود ندارد. بنابراین این تخطی از ویژگی ۲ تنها تخطی ویژگی‌های قرمز – سیاه در کل درخت می‌باشد.

اگر تخطی ویژگی ۴ وجود داشته باشد آنگاه چون فرزندان گره z محافظ هستند و درخت قبل از اضافه شدن z هیچ تخطی دیگری نداشته است، تخطی باید به دلیل قرمز بودن هم z و هم $p[z]$ باشد. علاوه بر این، تخطی دیگری از ویژگی‌های قرمز - سیاه وجود ندارد.

خاتمه: حلقه به این علت خاتمه می‌یابد که $p[z]$ سیاه است. (اگر z ریشه باشد آنگاه $p[z]$ محافظ $nil[T]$ که سیاه است می‌باشد.) بنابراین در پایان حلقه تخطی ویژگی ۴ وجود ندارد. با توجه به ثابت حلقه تنها ویژگی که ممکن است حفظ نشود ویژگی ۲ است. خط ۱۶ نیز این ویژگی را بازیابی می‌کند تا اینکه وقتی *RB-INSERT-FIXUP* پایان می‌یابد، همه ویژگیهای قرمز - سیاه حفظ شود.

نگهداری: در حقیقت در حلقه *while* ۶ حالت وجود دارد ولی سه حالت آنها با سه حالت دیگر، بسته به این که پدر z یعنی $p[z]$ فرزند چپ یا راست پدربزرگ z یعنی $p[p[z]]$ باشد که در خط ۲ مشخص می‌شود، متقارن هستند. ما تنها کد مربوط به وضعیتی که در آن $p[z]$ یک فرزند چپ می‌باشد را ارائه کرده‌ایم. گره $p[p[z]]$ وجود دارد چون با توجه به قسمت (b) ثابت حلقه، اگر $p[z]$ ریشه باشد آنگاه $p[z]$ سیاه است. از آنجا که تنها زمانی وارد تکرار حلقه می‌شویم که $p[z]$ قرمز است، می‌دانیم که $p[z]$ نمی‌تواند ریشه باشد. از این رو $p[p[z]]$ وجود دارد.

حالت اول از حالت دوم و سوم توسط رنگ همزاد پدر z یا عموی z تشخیص داده می‌شود. خط ۳ باعث می‌شود که y به عموی z $right[p[p[z]]]$ اشاره کند، و در خط ۴ یک تست انجام می‌شود. اگر y قرمز باشد آنگاه حالت ۱ انجام می‌شود، در غیر اینصورت کنترل به حالت‌های ۲ و ۳ منتقل می‌شود. در هر سه حالت، جد z یعنی $p[p[z]]$ سیاه است. چون پدرش $p[z]$ قرمز می‌باشد و ویژگی ۴ تنها بین z و $p[z]$ مورد تخطی قرار می‌گیرد.

حالت ۱: عموی z، یعنی y، قرمز باشد.

شکل ۱۳.۵ وضعیت حالت یک (خطوط ۸-۵) را نشان می‌دهد. حالت اول زمانی انجام می‌شود که هم $p[z]$ و هم y قرمز باشند. چون $p[p[z]]$ سیاه است می‌توانیم هم $p[z]$ و هم y را سیاه رنگ بزنیم، از این طریق مشکل قرمز بودن هم z و هم $p[z]$ را حل کرده و $p[p[z]]$ را قرمز می‌کنیم. بدین وسیله ویژگی ۵ حفظ می‌شود. سپس حلقه *while* را با $p[p[z]]$ به عنوان گره جدید z تکرار می‌کنیم. اشاره گر z دو سطح به بالا در درخت حرکت می‌کند. اکنون نشان می‌دهیم که حالت ۱ در شروع تکرار بعدی، ثابت حلقه را حفظ می‌کند. از z برای مشخص کردن گره z در تکرار جاری و از $z'=p[[z]]$ برای مشخص کردن گره z در تست خط اول تکرار بعدی استفاده می‌کنیم.

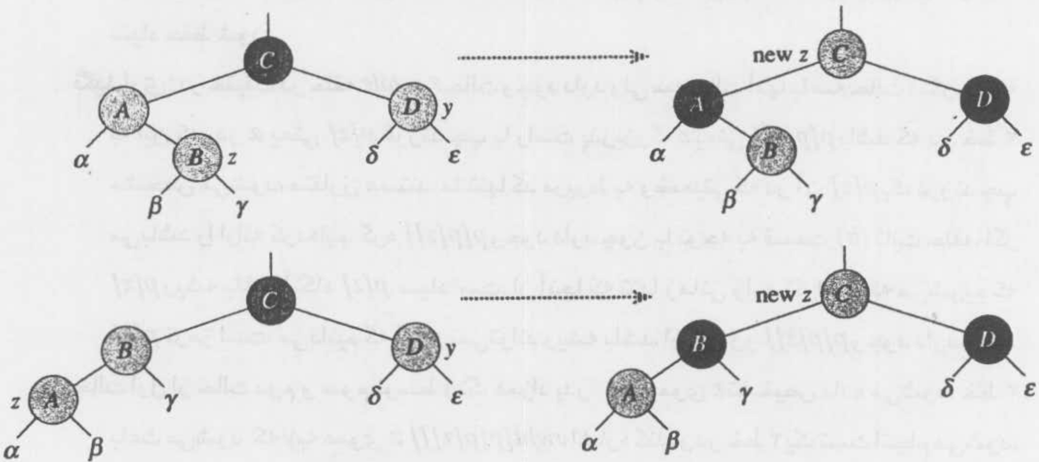
a. چون این تکرار، $p[p[z]]$ را قرمز می‌کنه گره z در شروع تکرار بعدی قرمز است.

b. گره $p[z']$ در این تکرار برابر $p[p[z]]$ می‌باشد و رنگ این گره تغییر نمی‌کند. اگر این گره ریشه

باشد، قبل از این تکرار سیاه بوده و در ابتدای تکرار بعدی سیاه باقی می‌ماند.

c. قبلاً ثابت کردیم که حالت ۱، ویژگی ۵ را حفظ می‌کند و به طور واضح تخطی ویژگی‌های ۱ یا ۳ را مشخص نمی‌کند.

اگر گره z' در شروع تکرار بعدی ریشه باشد آنگاه حالت اول تنها تخطی ویژگی 4 در این تکرار را تصحیح می‌کند. چون z' قرمز است و ریشه نیز می‌باشد، ویژگی ۲ تنها ویژگی است که مورد تخطی قرار می‌گیرد و این تخطی مربوط به z' است.



شکل ۱۳.۵ حالت ۱ روال RB- INSERT چون z و پدرش $p[z]$ هر دو قرمز هستند، ویژگی ۴ مورد تخطی قرار می‌گیرد. چه z (a) یک فرزند راست باشد، یا (b) z یک فرزند چپ باشد، عمل یکسانی انجام می‌گیرد. هر یک از زیردرخت‌های α, β, γ, ε ریشه سیاه و ارتفاع سیاه یکسانی دارند، که حالت ۱، برای حفظ ویژگی ۵، رنگ بعضی از گره‌ها را تغییر می‌دهد: همه مسیرهای رو به پایین از یک گره به یک برگ، تعداد یکسان گره سیاه دارند. حلقه *while* با پدربزرگ گره z، یعنی $p[p[z]]$ به عنوان z جدید ادامه می‌یابد. اکنون تخطی از ویژگی ۴ می‌تواند تنها بین z جدید که قرمز است و پدرش، اگر آن هم قرمز باشد، اتفاق بیفتد.

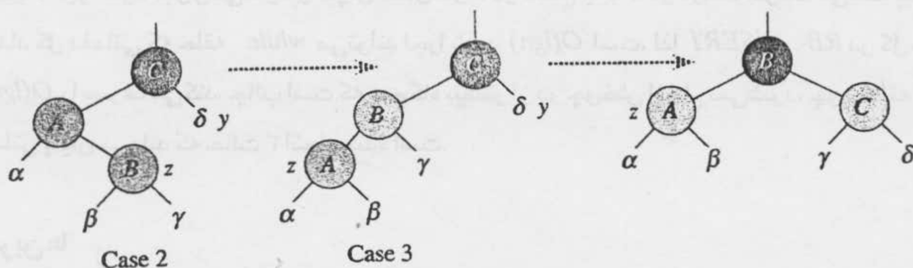
اگر گره z' در ابتدای تکرار بعدی، ریشه نباشد، آنگاه حالت ۱، تخطی از ویژگی ۲ را ایجاد نکرده است. حالت ۱ تنها تخطی از ویژگی ۴ را که در ابتدای این تکرار وجود دارد تصحیح میکند سپس z' را قرمز و $p[z']$ را تنها باقی می‌گذارد. اگر $p[z']$ سیاه بود تخطی از ویژگی ۴ وجود نداشت. اگر $p[z']$ قرمز می‌بود، قرمز کردن z' تخطی از ویژگی ۴ را بین z' و $p[z']$ به وجود می‌آورد.

حالت ۲: عموی z یعنی y سیاه و z یک فرزند راست است.

حالت ۳: عموی z یعنی y سیاه و z یک فرزند چپ است.

در حالت‌های ۳ و ۲، رنگ عموی z یعنی y سیاه است. این دو حالت با توجه به این که z یک فرزند راست

است یا یک فرزند چپ $p[z]$ باشد، از هم متمایز می‌شوند. خطوط ۱۵-۱۱ حالت ۲ را تشکیل می‌دهند که همراه با حالت ۳ در شکل ۱۳.۶ نشان داده شده است. در حالت ۲، گره z فرزند راست پدرش است. بلافاصله از یک چرخش به چپ استفاده می‌کنیم تا وضعیت را به حالت ۳ (خطوط ۱۴-۱۲) که در آن گره z یک فرزند چپ است تبدیل کنیم. چون هم z و هم $p[z]$ قرمز هستند، چرخش نه روی ارتفاع سیاه گره‌ها اثر می‌گذارد و نه روی ویژگی ۵. چه مستقیماً وارد حالت ۳ شویم یا چه از طریق حالت ۲، عموی z یعنی y سیاه است، چون در غیر این صورت حالت ۱ را انجام داده بودیم. و بعلاوه گره $p[p[z]]$ موجود است، زیرا ثابت کرده‌ایم که این گره در زمان اجرا خطوط ۲ و ۳ موجود است و بعد از حرکت یک سطح به بالای z در خط ۱۰ و سپس یک سطح به پایین در خط ۱۱ مقدار $p[p[z]]$ بدون تغییر می‌ماند. در حالت ۳، چندین تغییر رنگ و یک چرخش به راست اجرا می‌کنیم که ویژگی ۵ را حفظ کند و سپس چون دیگر دو گره قرمز در یک ردیف نداریم کار انجام شده است. بدنه حلقه *while* بار دیگر اجرا نمی‌شود چون $p[z]$ اکنون سیاه است.



شکل ۱۳.۶ حالت‌های ۳ و ۲ روال *RB- INSERT* همانند حالت ۱، ویژگی ۴ در هر یک از حالت‌های ۳ و ۲ مورد تخطی قرار می‌گیرد، چون هم z و هم پدرش $p[z]$ قرمز هستند. هر یک از زیردرخت‌های α، β، γ و δ یک ریشه سیاه دارند. (α، β، γ از ویژگی ۴ و δ چون در غیر این صورت در حالت ۱ می‌باشیم)، و هر یک ارتفاع یکسانی دارند. حالت ۲ با یک چرخش به چپ به حالت ۳ تبدیل می‌شود که ویژگی ۵ را حفظ می‌کند: همه مسیرهای رو به پایین از گره به برگ دارای تعداد سیاه یکسانی هستند. حالت ۳، چند تغییر رنگ و یک چرخش به راست انجام می‌دهد که باز هم ویژگی ۵ حفظ می‌شود. چون ویژگی ۴ برقرار است آنگاه حلقه *while* پایان می‌یابد: دیگر هیچ دو گره قرمزی در یک سطر وجود ندارد.

حال نشان می‌دهیم که حالت‌های ۲ و ۳ ثابت حلقه را حفظ می‌کنند. (همان طور که قبلاً ثابت کردیم $p[z]$ قبل از تست بعدی خط ۱ سیاه خواهد بود و بدنه حلقه دوباره اجرا نمی‌شود.)

a. حالت ۲، باعث می‌شود z به $p[z]$ که قرمز است اشاره کند. هیچ تغییر اضافی در z یا رنگش، در حالت ۲ و ۳ اتفاق نمی‌افتد.

b. حالت ۳، باعث می‌شود $p[z]$ سیاه شود، بنابراین اگر $p[z]$ در ابتدای تکرار بعدی ریشه باشد، سیاه است.

c. همانند حالت ۱ در حالت‌های ۲ و ۳ نیز ویژگی‌های ۱، ۳ و ۵ حفظ می‌شوند. چون گره z در حالت‌های ۲ و ۳ ریشه نمی‌باشد می‌دانیم که تخطی از ویژگی ۲ وجود ندارد. حالت‌های ۲ و ۳ تخطی از ویژگی ۲ را مشخص نمی‌کنند، زیرا تنها گره‌ای که قرمز می‌شود توسط چرخش در حالت ۳، فرزند یک گره سیاه می‌شود.

حالت‌های ۲ و ۳ تنها تخطی ویژگی ۴ را تصحیح می‌کنند و تخطی دیگری را مشخص نمی‌کنند. با نشان دادن این که هر تکرار حلقه، ثابت را حفظ می‌کند نشان دادیم که روال *RB-INSERT-FIXUP* به درستی ویژگی قرمز - سیاه را بازیابی می‌کند.

تحلیل

زمان اجرای *RB-INSERT* چیست؟ چون ارتفاع یک درخت قرمز - سیاه روی n گره برابر $O(lgn)$ می‌باشد، خطوط ۱۶-۱ زمان $O(lgn)$ را صرف می‌کنند. در *RB-INSERT-FIXUP* حلقه *while* تنها اگر حالت ۱ اجرا شود تکرار می‌شود و سپس اشاره‌گر z دو سطح به بالا در درخت حرکت می‌کند. بنابراین تعداد کل دفعاتی که حلقه *while* می‌تواند اجرا شود $O(lgn)$ است. لذا *RB- INSERT* در کل، زمان $O(lgn)$ را صرف می‌کند. جالب است که هیچ‌گاه بیشتر از دو چرخش انجام نمی‌شود، چون حلقه *while* زمانی پایان می‌یابد که حالت ۳ انجام شده است.

تمرین‌ها

۱۳.۳-۱ در خط ۱۶ روال *RB-INSERT*، رنگ گره z که جدیداً اضافه شده را قرمز قرار دهید. توجه کنید که اگر رنگ z را سیاه انتخاب کرده بودیم، ویژگی ۴ درخت قرمز - سیاه مورد تخطی قرار نمی‌گرفت. چرا رنگ z را سیاه انتخاب نکرده‌ایم؟

۱۳.۳-۲ درخت‌های قرمز - سیاهی را رسم کنید که از درج متوالی کلیدهای $8,19,12,31,38,41$ در یک درخت قرمز - سیاه بصورت اولیه خالی، حاصل آیند.

۱۳.۳-۳ فرض کنید که ارتفاع سیاه زیردرخت‌های α، β، γ، δ و ε در شکل ۱۳.۵ و ۱۳.۶ ، k می‌باشد. در هر یک از شکل‌ها هر گره را با ارتفاع سیاهش برچسب دهی کنید تا مشخص شود که ویژگی ۵ با انتقال نشان داده شده، حفظ می‌شود.

۱۳.۳-۴ پرفسور *Teach* نگران بود که روال *RB-INSERT-FIXUP* ممکن است *color[nil[T]]* را برابر *RED* قرار دهد که در این حالت وقتی z ریشه است تست خط اول باعث پایان یافتن حلقه نمی‌شود. با اثبات این که تابع *RB-INSERT-FIXUP* هیچ‌گاه *color[nil[T]]* را برابر *RED* قرمز نمی‌دهد، نشان دهید که نگرانی پرفسور بی‌مورد است.

۱۳.۳-۵ یک درخت قرمز - سیاه که با درج n گره توسط روال *RB-INSERT* به وجود آمده را در

نظر بگیرید. ثابت کنیدکه اگر $n>1$ باشد درخت حداقل یک گره قرمز دارد.

۶-۱۳.۳ اگر نمایش درخت قرمز - سیاه شامل حافظه‌ای برای اشاره‌گرهای پدر نباشد، روشی برای پیاده سازی کارآمد تابع *RB-INSERT* پیشنهاد کنید.

۱۳.۴ حذف

همانند بقیه اعمال اصلی روی یک درخت قرمز - سیاه n گره‌ای، حذف یک گره هم زمان $O(lgn)$ را صرف می‌کند .حذف یک گره از درخت قرمز - سیاه تنها کمی از درج یک گره پیچیده‌تر است. روال *RB-DELETE* تغییر یافته روال *TREE-DELETE* (بخش ۱۲.۳) است. بعد از حذف یک گره، روال کمکی *RB-DELETE-FIXUP* را که برای بازگرداندن ویژگیهای قرمز - سیاه، رنگ‌ها را تغییر می‌دهد و چرخش انجام می‌شود، فراخوانی می‌کند .

```
RB-DELETE(T, z)
 1  if left[z] = nil[T] or right[z] = nil[T]
 2     then y ← z
 3     else  y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ nil[T]
 5     then x ← left[y]
 6     else  x ← right[y]
 7  p[x] ← p[y]
 8  if p[y] = nil[T]
 9     then root[T] ← x
10     else if y = left[p[y]]
11             then left[p[y]] ← x
12             else  right[p[y]] ← x
13  if y ≠ z
14     then key[z] ← key[y]
15          copy y's satellite data into z
16  if color[y] = BLACK
17     then RB-DELETE-FIXUP(T, x)
18  return y
```

سه تفاوت بین توابع *TREE-DELETE* و *RB-DELETE* وجود دارد. اول این که همه ارجاعها به *NIL* در *TREE-DELETE* با ارجاع‌هایی به محافظ *nil[T]* در *RB-DELETE* جابه‌جا می‌شوند. دوم تست اینکه آیا x برابر *NIL* است یا خیر، در خط ۷ روال *TREE-DELETE* حذف و انتساب *p[y]* → *p[x]* بدون هیچ شرطی در خط ۷ روال *RB-DELETE* انجام می‌شود. بنابراین اگر x محافظ *nil[T]*

باشد، اشاره‌گر پدرش به پدر گره y حذف شده اشاره می‌کند. سوم اینکه اگر y سیاه باشد، در خطوط ۱۶-۱۷ تابع *RB-DELETE-FIXUP* فراخوانی می‌شود. اگر y قرمز باشد، وقتی حذف می‌شود به دلایل زیر ویژگی‌های قرمز - سیاه حفظ می‌شوند:

- هیچ ارتفاع سیاهی در درخت تغییر نمی‌کند.
- هیچ یک از گره‌های قرمز مجاور هم نمی‌شوند، و
- چون y نمی‌توانست ریشه باشد اگر قرمز می‌بود باز هم ریشه، سیاه باقی می‌ماند.

گره x که به روال *RB-DELETE-FIXUP* فرستاده می‌شود، یکی از این دو گره است: اگر y فرزندی داشته باشد که *nil[T]* نباشد، x تنها فرزند y قبل از حذف آن می‌باشد، و یا اگر y فرزندی نداشته، x محافظ *nil[T]* می‌باشد. در حالت بعدی، انتساب بدون شرط خط ۷ تضمین می‌کند که چه x یک گره داخلی حاوی کلید باشد یا چه محافظ *nil[T]* باشد، گره‌ای است که قبلاً پدر y بوده است.

اکنون بررسی می‌کنیم که چطور *RB-DELETE-FIXUP* ویژگی‌های قرمز - سیاه را در درخت جستجو بازیابی می‌کند.

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ root[T] and color[x] = BLACK
 2      do if x = left[p[x]]
 3            then w ← right[p[x]]
 4                 if color[w] = RED
 5                    then color[w] ← BLACK                          ▷ Case 1
 6                         color[p[x]] ← RED                         ▷ Case 1
 7                         LEFT-ROTATE(T, p[x])                      ▷ Case 1
 8                         w ← right[p[x]]                           ▷ Case 1
 9                 if color[left[w]] = BLACK and color[right[w]] = BLACK
10                    then color[w] ← RED                            ▷ Case 2
11                         x ← p[x]                                  ▷ Case 2
12                    else if color[right[w]] = BLACK
13                            then color[left[w]] ← BLACK            ▷ Case 3
14                                 color[w] ← RED                    ▷ Case 3
15                                 RIGHT-ROTATE(T, w)                ▷ Case 3
16                                 w ← right[p[x]]                   ▷ Case 3
17                         color[w] ← color[p[x]]                    ▷ Case 4
18                         color[p[x]] ← BLACK                       ▷ Case 4
19                         color[right[w]] ← BLACK                   ▷ Case 4
20                         LEFT-ROTATE(T, p[x])                      ▷ Case 4
21                         x ← root[T]                               ▷ Case 4
22            else (same as then clause with "right" and "left" exchanged)
23  color[x] ← BLACK
```

اگر گره حذف شده y در *RB-DELETE* سیاه باشد، سه مشکل ممکن است به وجود آید. اول اگر y ریشه بود و یک فرزند قرمز y، ریشه جدید شود، ویژگی ۲ مورد تخطی قرار می‌گیرد. دوم اگر هم x و هم $p[y]$ (که اکنون برابر $p[x]$ نیز هست) قرمز باشند، آنگاه ویژگی ۴ مورد تخطی قرار می‌گیرد. سوم آنکه حذف y باعث می‌شود هر مسیری که قبلاً شامل y بوده است یک گره سیاه کمتر داشته باشد. بنابراین اکنون ویژگی ۵ توسط هر جد y در درخت مورد تخطی قرار می‌گیرد. می‌توانیم این مشکل را با گفتن اینکه گره x سیاه «اضافی» دارد تصحیح کنیم، این بدان معناست که اگر به تعداد گره‌های سیاه در هر مسیر که شامل x می‌شود یکی اضافه کنیم آنگاه طبق این تعبیر ویژگی ۵ حفظ می‌شود. زمانیکه گره سیاه y را حذف می‌کنیم رنگ سیاهش را به فرزندش منتقل می‌کنیم. مشکل این است که اکنون گره x نه قرمز است و نه سیاه، در نتیجه ویژگی ۱ مورد تخطی قرار می‌گیرد. در عوض، گره x یا "دوبله سیاه"[1] یا "قرمز و سیاه"[2] است و به تعداد گره‌های سیاه در مسیرهایی که شامل x هستند به ترتیب 2 یا 1 اضافه می‌شود. خاصیت *color* گره x هنوز *RED* یا *BLACK* است (اگر x قرمز و سیاه باشد) یا *BLACK* است (اگر x دوبله سیاه باشد) به بیان دیگر، سیاه اضافه برای یک گره در اشاره کردن x به یک گره منعکس می‌شوند نه در خاصیت *color* آن.

روال *RB-DELETE-FIXUP* ویژگی‌های ۱، ۲ و ۴ را بازیابی می‌کند. تمرین‌های ۱- ۱۳.۴ و ۲- ۱۳.۴ از شما می‌خواهند که نشان دهید روال، ویژگی‌های ۲ و ۴ را بازیابی می‌کند و بنابراین در ادامه بخش روی ویژگی ۱ متمرکز می‌شویم. هدف حلقه *while* در خطوط ۲۲-۱، حرکت سیاه اضافی به طرف بالای درخت می‌باشد تا زمانیکه

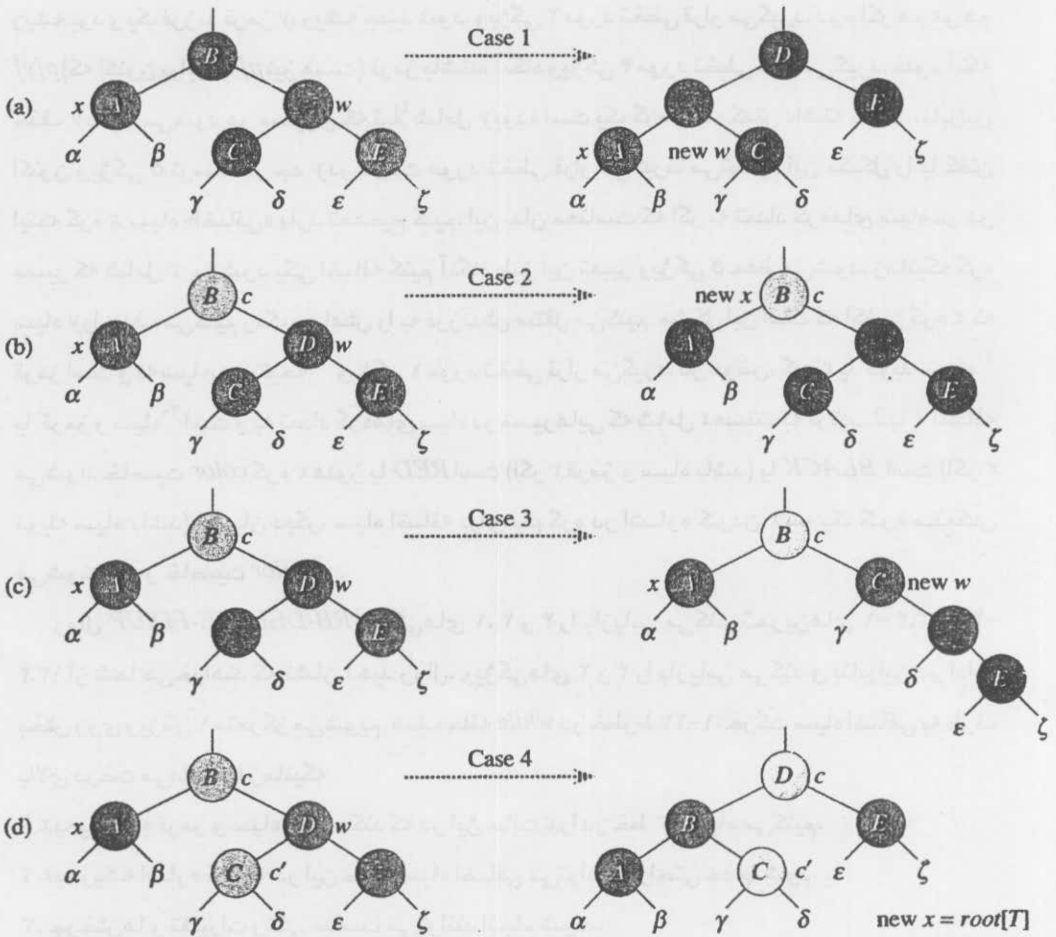
۱. x به یک گره قرمز و سیاه اشاره کند که در این حالت x را در خط ۲۳ سیاه می‌کنیم،

۲. x به ریشه اشاره کند که در این حالت سیاه اضافی می‌تواند به راحتی حذف شود، یا

۳. چرخش‌ها و تغییرات رنگی مناسب می‌توانند انجام شوند.

در حلقه *while*، x همیشه به یک گره دوبله سیاه غیر ریشه اشاره می‌کند. در خط ۲ مشخص می‌کنیم که آیا x فرزند چپ پدرش $p[x]$ است یا فرزند راستش. (کدمربوط به وضعیتی که در آن x فرزند چپ است را ارائه کردیم؛ وضعیتی که در آن x فرزند راست باشد – خط ۲۲– متقارن است.) یک اشاره‌گر w که به همزاد x اشاره می‌کند را نگه می‌داریم. چون گره x دوبله سیاه است، گره w نمی‌تواند $nil[T]$ باشد. در غیر اینصورت تعداد گره‌های سیاه در هر مسیر از $p[x]$ به برگ w (تک سیاه) کوچکتر از تعداد گره‌های سیاه در مسیری از $p[x]$ به x می‌باشد. چهار حالت واقع در کد، در شکل ۱۳.۷ نشان داده شده است. قبل از بررسی مفصل هر قسمت، بیایید به طور کلی‌تر به اینکه چطور می‌توان ثابت کرد که تغییر در هر حالت، ویژگی ۵ را حفظ می‌کند نگاه کنیم. ایده اصلی این است که در هر حالت تعداد گره‌های سیاه (شامل سیاه اضافی x) از ریشه (و شامل ریشه) زیردرختی که برای هر یک از زیردرخت‌های α, β, ..., ζ

1.doubly black   2.red - and - black



شکل ۱۳.۷ حالت‌های موجود در حلقه *while* در روال *RB-DELETE-FIXUP* و ویژگی *color* گره‌های تیره برابر *BLACK* و ویژگی *color* گره‌های سایه پررنگ خورده برابر *RED* است و خاصیت *color* گره‌های سایه روشن خورده توسط c و c' که ممکن است *RED* یا *BLACK* باشند ارائه می‌شود. حروف $\alpha, \beta, \ldots, \zeta$ زیردرخت‌های دلخواه را نمایش می‌دهند. در هر حالت ساختار چپ با تعویض چندین رنگ و / یا انجام یک چرخش به ساختار راست تبدیل می‌شود. هر گره‌ای که x به آن اشاره می‌کند یک سیاه اضافی داشته و یا دوبله سیاه و یا قرمز می‌باشد. تنها حالتی که باعث تکرار حلقه می‌شود، حالت ۲ است. *(a)* حالت ۱ با جابجایی رنگ‌های گره‌های B و D و انجام یک چرخش به چپ به حالت‌های ۲، ۳ و ۴ تبدیل می‌شود. *(b)* در حالت ۲، سیاه اضافی که توسط اشاره‌گر x نمایش داده شده، با قرمز کردن D و تغییر x برای اشاره به B به طرف بالای درخت حرکت می‌کند. اگر از طریق حالت ۱ به حالت ۲ وارد شویم، حلقه *while* به دلیل این که گره جدید x قرمز و سیاه است و بنابراین مقدار c ویژگی *color* آن *RED* است، پایان می‌یابد. *(c)* حالت ۳ با جابه‌جایی رنگ‌های C و D و انجام یک چرخش به راست به حالت ۴ تبدیل می‌شود. *(d)* در حالت ۴، سیاه اضافی که توسط x نمایش داده می‌شود، می‌تواند با تعویض بعضی رنگ‌ها و یک چرخش به چپ از بین برود (بدون تخطی از ویژگیهای قرمز - سیاه) و حلقه پایان می‌یابد.

نشان داده شده است، توسط تبدیلاتی حفظ می‌شود. بنابراین اگر ویژگی ۵ قبل از تبدیلات برقرار باشد، بعد از آن نیز برقرار خواهد بود. برای مثال در شکل (a)۱۳.۷ که حالت ۱ را نشان می‌دهد، تعداد گره‌های سیاه از ریشه به هر یک از زیردرخت‌های α یا β هم قبل و هم بعد از تبدیل برابر 3 می‌باشد. (دوباره خاطرنشان می‌کنیم که گره xیک سیاه اضافی را اضافه می‌کند.) به طور مشابه تعداد گره‌های سیاه از ریشه تا هر یک از $\zeta,\varepsilon,\delta,\gamma$ هم قبل و هم بعد تغییر برابر 2 می‌باشد. در شکل (b)۱۳.۷، شمارش باید شامل مقدار c خاصیت $color$ ریشهٔ زیردرخت نشان داده شده باشد که می‌تواند RED یا $BLACK$ باشد.

اگر $count(RED)=0$ و $count(BLACK)=1$ باشد آنگاه تعداد گره‌های سیاه از ریشه تا α هم قبل و هم بعد از تبدیل برابر $2+count(c)$ می‌باشد. در اینصورت، بعد از تبدیل، گره جدید x دارای ویژگی $color$ یعنی c می‌باشد. ولی این گره در حقیقت یا قرمز و سیاه است (اگر $c=RED$ باشد.) یا دوبله سیاه است (اگر $c=BLACK$ باشد) بقیه حالات هم به طور مشابه می‌توانند بررسی شوند (به تمرین ۵-۱۳.۴ مراجعه کنید).

حالت ۱: همزاد x یعنی w قرمز است.

حالت ۱ (خطوط ۸-۵ روال *RB-DELETE-FIXUP* و شکل (a)۱۳.۷) زمانی اتفاق می‌افتد که w یعنی همزاد x قرمز باشد. چون w باید فرزندان سیاه داشته باشد، می‌توانیم رنگ‌های w و $p[x]$ را عوض کنیم و سپس یک چرخش به روی $p[x]$ بدون تخطی ویژگی‌های قرمز - سیاه انجام دهیم. همزاد جدید x که یکی از فرزندان w قبل از چرخش می‌باشد اکنون سیاه است و بنابراین حالت ۱ را به حالت ۲ و ۳ یا ۴ تبدیل کرده‌ایم. حالت‌های ۲، ۳ و ۴ زمانی اتفاق می‌افتند که گره w سیاه باشد؛ و آنها با رنگ فرزندان w از هم متمایز می‌شوند.

حالت ۲: همزاد x یعنی w سیاه است و هر دو فرزند w سیاه هستند.

در حالت ۲ (خطوط ۱۵-۱۱ روال *RB-DELETE-FIXUP* و شکل (b)۱۳.۷) هر دو فرزند w سیاه هستند. چون w هم سیاه است، یک سیاه از x و w کم می‌کنیم، x را با تنها یک سیاه و w را قرمز رها می‌کنیم. برای جبران حذف یک سیاه از x و w، یک سیاه اضافی به $p[x]$ که در ابتدا قرمز یا سیاه بوده است اضافه می‌کنیم. این کار را با تکرار حلقه *while* با $p[x]$ به عنوان گره جدید x انجام می‌دهیم. می‌بینید که اگر به حالت ۲ از طریق حالت ۱ وارد شویم، گره x جدید، قرمز و سیاه است. چون $p[x]$ اولیه قرمز بوده است. از اینرو مقدار خاصیت $color$ برای گره جدید x برابر RED می‌باشد و زمانی که شرط حلقه تست می‌شود، حلقه پایان می‌یابد، سپس گره جدید x در خط ۲۳، رنگ سیاه (به طور تکی) می‌گیرد.

حالت ۳: همزاد x یعنی w سیاه و فرزند چپ w قرمز و فرزند راست w سیاه می‌باشد.

حالت ۳ (خطوط ۱۶-۳ و شکل (c)۱۳.۷) زمانیکه w سیاه و فرزند چپش قرمز و فرزند راستش سیاه

باشد اتفاق می‌افتد. می‌توانیم رنگ‌های w و فرزند چپش $left[w]$ را عوض کرده و سپس یک چرخش به راست بدون تخطی ویژگی‌های قرمز – سیاه روی w انجام دهیم. همزاد جدید x یعنی w اکنون یک گره سیاه با یک فرزند راست قرمز است و بنابراین حالت ۳ را به حالت ۴ تبدیل کرده‌ایم.

حالت ۴: همزاد x یعنی w سیاه و فرزند راست w قرمز است.

حالت ۴ (خطوط ۲۱–۱۷ و شکل ۱۳.۷(d)) زمانی که همزاد x یعنی w و فرزند راست w قرمز است اتفاق می‌افتد. با چند تغییر رنگ و یک چرخش چپ روی $p[x]$ می‌توانیم سیاه اضافی گره x را حذف کنیم و آن را بدون تخطی ویژگی‌های قرمز – سیاه، تک سیاه کنیم. قرار دادن x به عنوان ریشه باعث می‌شود زمانی که شرط حلقه *while* تست می‌شود، حلقه پایان یابد.

تحلیل

زمان اجرای *RB-DELETE* چیست؟ چون ارتفاع یک درخت قرمز – سیاه با n گره، $O(lgn)$ می‌باشد کل روال بدون فراخوانی *RB-DELETE-FIXUP*، زمان $O(lgn)$ را صرف می‌کند. در *RB-DELETE-FIXUP*، حالت‌های ۱، ۳ و ۴ هر کدام بعد از اجرای تعداد ثابتی تغییر رنگ و حداکثر سه چرخش پایان می‌یابد. حالت ۲ تنها حالتی است که در آن حلقه *while* می‌تواند تکرار شود و سپس اشاره‌گر x حداکثر $O(lgn)$ دفعه به طرف بالای درخت حرکت می‌کند و هیچ چرخش انجام نمی‌شود. بنابراین روال *RB-DELETE-FIXUP* حداکثر سه چرخش انجام می‌دهد و زمان $O(lgn)$ صرف می‌کند، و بنابراین زمان کل برای روال *RB-DELETE* همان $O(lgn)$ است.

تمرین‌ها

۱۳.۴–۱ ثابت کنید که بعد از اجرای روال *RB-DELETE-FIXUP* ریشه درخت باید سیاه باشد.

۱۳.۴–۲ توضیح دهید که اگر *RB-DELETE* هم x و هم $p[y]$ قرمز باشند، آنگاه با فراخوانی *RB-DELETE-FIXUP(T,x)* ویژگی ۴ بازیابی می‌شود.

۱۳.۴–۳ در تمرین ۱۳.۳–۲ درخت قرمز – سیاهی را پیدا کردید که از درج متوالی کلیدهای *8,19,12,13,38,41* در یک درخت خالی اولیه به دست می‌آمد. اکنون درخت‌های قرمز – سیاهی را نشان دهید که از حذف متوالی کلیدها، به ترتیب *41,38,31,19,12,8* به دست می‌آیند.

۱۳.۴–۴ در کدام خطوط کد *RB-DELETE-FIXUP* ممکن است محافظ $nil[T]$ را تغییر داده یا امتحان کنیم؟

۱۳.۴–۵ در هر یک از حالت‌های شکل ۱۳.۷ تعداد گره‌های سیاه از ریشه زیر درخت نشان داده شده تا هر یک از زیردرخت‌های α,β، . . . ، ζ را بیان کرده و تعیین کنید که هر شمارش، بعد از تغییر نیز

یکسان می‌ماند. زمانیکه یک گره دارای خاصیت *color* با مقدار c یا c' است، بصورت نمادین از علامت $count(c)$ یا $count(c')$ برای شمارش استفاده کنید.

۶-۱۳.۴ پرفسور *Skelton* و *Buron* نگرانند که در شروع حالت ۱ روال *RB-DELETE-FIXUP* گره $p[x]$ ممکن است سیاه نباشد. اگر آنها درست بگویند آنگاه خطوط ۶-۵ نادرست است. نشان دهید که $p[x]$ در شروع حالت ۱ باید سیاه باشد، بنابراین جایی برای نگرانی آنها نمی‌باشد.

۷-۱۳.۴ فرض کنید که گره x توسط *RB-DELETE* در یک درخت قرمز - سیاه درج شده و سپس بلافاصله با تابع *RB-DELETE* حذف شود. آیا درخت قرمز - سیاه حاصل با درخت قرمز - سیاه اولیه یکی است؟ جوابتان را توجیه کنید.

مسائل

۱-۱۳ مجموعه‌های پویای پایدار [1]

گاهی در طول یک دورهٔ یک الگوریتم، متوجه می‌شویم که به نگهداری نسخه‌های قبلی یک مجموعه پویا که تغییر کرده است نیاز داریم. چنین مجموعه‌ای، ***پایدار*** [2] نامیده می‌شود. یک روش پیاده‌سازی یک مجموعه پایدار این است که هر زمان که مجموعه تغییر کرد از کل آن یک کپی تهیه کنیم، ولی این روش ممکن است از سرعت برنامه کاسته و همچنین فضای زیادی هدر دهد. گاهی می‌توانیم بهتر عمل کنیم.

یک مجموعه پایدار S را با عملیات درج، حذف و جستجو در نظر بگیرید، که همان طور که در شکل ۱۳.۸(a) نشان داده شده است با استفاده از درخت‌های جستجوی دودویی پیاده‌سازی شده است. برای هر نسخه مجموعه، یک ریشه جدا نگه می‌داریم. برای درج کلید در مجموعه، یک گره جدید با کلید 5 می‌سازیم. چون نمی‌توانیم گره موجود با کلید 7 را تغییر دهیم، این گره، فرزند چپ یک گرهٔ جدید باکلید 7 می‌شود. به طور مشابه گره جدید با کلید 7، فرزند چپ یک گره جدید با کلید 8 می‌شود که فرزند راست آن، گره موجود با کلید 10 است. گره جدید با کلید 8 هم بنوبه خود فرزند راست یک ریشه جدید r' با کلید 4 می‌شود که فرزند چپ آن، گره موجود با کلید 3 است. بنابراین تنها از قسمتی از درخت کپی می‌گیریم و تعدادی از گره‌ها را با درخت اولیه مشترک می‌گیریم، همان طور که در شکل ۱۳.۸(b) نشان داده شده است.

فرض کنید هر گره درخت دارای فیلدهای *key*، *left* و *right* می‌باشد ولی فیلد پدر ندارد. (تمرین ۶-۱۳.۳ را ملاحظه کنید.)

a. در یک درخت جستجوی دودویی پایدار عمومی، گره‌هایی که برای درج کلید k یا حذف گره y لازم است تغییر کنند را مشخص کنید.

---

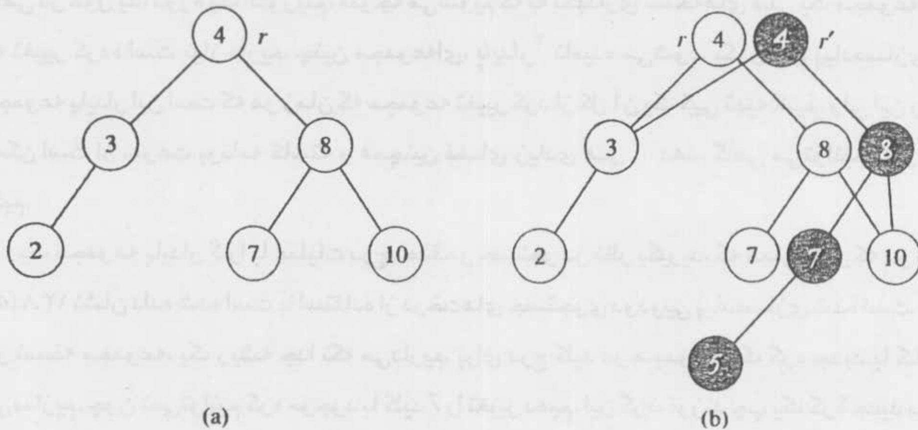
1. persistent dynamic set
2. persistent

b. یک روال *RB-DELETE-FIXUP* بنویسید که یک درخت پایدار T و یک کلید k را برای درج گرفته و یک درخت پایدار T' که از درج k در T به دست می‌آید را برگرداند.

c. اگر ارتفاع درخت جستجوی دودویی پایدار T، h باشد، زمان و فضای مورد نیاز برای پیاده‌سازی *RB-DELETE-FIXUP* شما چیست؟ (زمان مورد نیاز با تعداد گره‌های جدیدی که تخصیص داده می‌شوند متناسب است.)

d. فرض کنید که فیلدی برای پدر و در هر گره اضافه کرده‌ایم. در این حالت روال *RB-DELETE-FIXUP* نیاز به کپی کردن‌های اضافی دارد. ثابت کنید که *RB-DELETE-FIXUP* زمان و فضا $\Omega(n)$ را نیاز دارد که n تعداد گره‌ها در درخت می‌باشد.

e. نشان دهید چطور می‌توان از درخت‌های قرمز – سیاه برای تضمین این که زمان و فضا در بدترین حالت برای هر درج یا حذف $O(lgn)$ می‌باشد استفاده کرد.



شکل ۱۳.۸ (a) یک درخت جستجوی دودویی با کلیدهای 2,3,4,7,8,10. (b) درخت جستجوی دودویی پایداری که از درج کلید 5 به دست می‌آید. جدیدترین نسخه مجموعه شامل گره‌هایی می‌شود که از ریشه r' قابل دستیابی هستند و نسخه قبلی آن شامل گره‌هایی است که از ریشه r قابل دستیابی می‌باشند. گره‌هایی که تیره هستند زمانی اضافه شده‌اند که کلید 5 درج شده است.

۲-۱۳ عمل الحاق[1] در درخت‌های قرمز – سیاه

عمل الحاق، دو مجموعه پویای S_1 و S_2 و یک عنصر x را می‌گیرد که برای هر $x_1 \in S_1$, $x \in S_2$ داریم $key[x_1] \leq key[x] \leq key[x_2]$ این عمل مجموعه $S = S_1 \cup \{x\} \cup S_2$ را برمی‌گرداند. در این مسئله، بررسی می‌کنیم که چطور عمل الحاق را در درخت‌های قرمز – سیاه پیاده‌سازی می‌کنیم.

---
1. join

a. در درخت قرمز – سیاه T داده شده، ارتفاع سیاه آن را در فیلد $bh[T]$ ذخیره می‌کنیم. ثابت کنید که این فیلد می‌تواند توسط روال‌های BR-INSERT و RB-DELETE بدون نیاز به حافظه اضافی در گره‌های درخت و بدون افزایش زمان اجرای مجانبی، حفظ شود. نشان دهید هنگامی که در T به پایین حرکت می‌کنیم، می‌توانیم ارتفاع سیاه هر گره‌ای که در زمان $O(1)$ ملاقات می‌کنیم را در زمان ملاقات آن مشخص کنیم.

می‌خواهیم عمل $RB\text{-}JOIN(T_1,x,T_2)$ را پیاده‌سازی کنیم، که T_1 و T_2 را خراب کرده و یک درخت قرمز – سیاه $T=T_1\cup\{x\}\cup T_2$ را برمی‌گرداند. n را تعدا کل گره‌های T_1 و T_2 قرار دهید.

b. فرض کنید $bh[T_1]\geq bh[T_2]$ است. یک الگوریتم با زمان $O(lgn)$ ارائه دهید که گره سیاه y را در T_1 پیدا کند که در میان گره‌هایی که ارتفاع سیاه آنها $bh[T_2]$ می‌باشد بزرگترین کلید را داشته باشد.

c. T_y را زیردرخت مشتق شده از y قرار دهید. توضیح دهید چطور $T_y\cup\{x\}\cup T_2$ بدون به هم زدن ویژگی درخت جستجوی دودویی در زمان $O(1)$، می‌تواند جایگزین T_y شود.

d. چه رنگی را باید به x داد تا ویژگی‌های قرمز – سیاه ۱، ۳و۵ حفظ شوند؟ توضیح دهید که چطور ویژگی‌های ۲و۴ می‌توانند در زمان $O(lgn)$ اعمال شوند.

e. ثابت کنید با فرض قسمت (b) هیچ کلیتی از بین نمی‌رود. موقعیت متقارنی که وقتی $bh[T_1]\leq bh[T_2]$ است به وجود می‌آید را توضیح دهید.

f. ثابت کنید زمان اجرای RB-JION برابر $O(lgn)$ می‌باشد.

۱۳ـ۳ درخت‌های AVL

درخت AVL، یک درخت جستجوی دودویی است که دارای *ارتفاع متوازن*[1] است: برای هر گره x ارتفاع زیردرخت‌های راست و چپ حداکثر یکی تفاوت دارد. برای پیاده‌سازی یک درخت AVL، یک فیلد اضافی برای هر گره نگه می‌داریم: $h[x]$ ارتفاع گره x است. همینطور برای هر درخت جستجوی دودویی T دیگر، فرض می‌کنیم $root[T]$ بر گره ریشه اشاره می‌کند.

a. ثابت کنید یک درخت AVL با n گره دارای ارتفاع $O(lgn)$ می‌باشد.(راهنمایی: ثابت کنید که در یک درخت AVL با ارتفاع h حداقل F_h گره وجود دارد که F_h، hامین عدد در سری فیبوناچی است.)

b. برای درج در یک درخت AVL، یک گره ابتدا در مکان مناسبش در درخت دودویی قرار می‌گیرد. بعد از این درج، درخت ممکن است دیگر دارای ارتفاع متوازن نباشد. خصوصاً ارتفاع فرزندان چپ و راست بعضی از گره‌ها ممکن است به اندازهٔ 2 واحد با یکدیگر اختلاف داشته باشند. یک روال Balance(x) بیان کنید که یک زیردرخت مشتق شده از x که فرزندان چپ و راستش دارای ارتفاع متوازن هستند و ارتفاع‌ها حداکثر 2 واحد با یکدیگر اختلاف دارند، یعنی

---

1. height balanced

$|h[right[x]]-h[left[x]]| \leq 2$ را گرفته و طوری زیردرخت مشتق شده از x را تغییر دهید که از لحاظ ارتفاع متوازن باشد. (راهنمایی: از چرخش‌ها استفاده کنید.)

c. با استفاده از قسمت (b)، روال بازگشتی *AVL-INSERT* را تعریف کنید که یک گره x در درخت *AVL* و یک گره z جدید ساخته شده (که کلید آن قبلاً پر شده است.) را گرفته و z را به زیردرخت مشتق شده از x اضافه کند، به طوریکه این خاصیت که x ریشه یک درخت *AVL* می‌باشد حفظ شود.

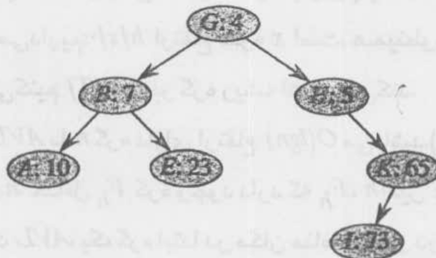
همانند روال *TREE-INSERT* در بخش ۱۲.۳ فرض کنید $key[z]$ قبلاً پر شده است و $right[z]=NIL$, $left[z]=NIL$ می‌باشد. همچنین فرض کنید $h[z]=0$ است. بنابراین برای درج z در درخت *AVL* بنام T، *AVL-INSERT(root[T],z)* را فراخوانی می‌کنیم.

d. مثالی برای یک درخت *AVL* با n گره ارائه دهید که در آن عمل *AVL-INSERT* تعداد $\Omega(lgn)$ چرخش انجام دهد.

۱۳ـ۴ *Treap*ها

اگر یک مجموعه از n داده در یک درخت جستجوی دودویی درج کنیم، درخت حاصل ممکن است به شدت نامتوازن شود، که منجر به زمان جستجوی طولانی می‌شود. همان طور که در بخش ۱۲.۴ مشاهده کردیم، درخت‌های جستجوی دودویی تصادفی ساخته شده، به موازنه بودن متمایل هستند. از این رو یک استراتژی که به طور میانگین، برای یک مجموعهٔ ثابت از داده‌ها یک درخت متوازن می‌سازد، این است که داده‌ها را به طور تصادفی جایگشت کرده و سپس آنها را با آن ترتیب در درخت درج کنیم.

اگر همه داده‌ها را با هم نداشته باشیم چه می‌شود؟ اگر در هر دفعه یکی از داده‌ها را دریافت کنیم، هنوز هم می‌توان به طور تصادفی یک درخت جستجوی دودویی حاصل از آنها را ساخت؟



شکل ۱۳.۹ یک *treap*. هر گره x با *key[x]* برچسب گذاری شده است: *priority[x]*. برای مثال، ریشه دارای کلید G و اولویت 4 است.

یک ساختمان داده را بررسی می‌کنیم که به این پرسش پاسخ دهد. *treap* یک درخت جستجوی دودویی با روش تغییر یافته مرتب کردن گره‌ها می‌باشد. شکل ۱۳.۹ نمونه‌ای از آن را نشان می‌دهد. طبق معمول هر گره x در درخت دارای مقدار کلید *key[x]* می‌باشد. بعلاوه، *priority[x]* را که یک عدد تصادفی است

که به طور مستقل برای هر گره انتخاب می‌شود را اختصاص می‌دهیم. فرض می‌کنیم همه اولویت‌ها مجزا هستند و همه کلیدها نیز مجزا هستند. گره‌های *treap* طوری مرتب می‌شوند که کلیدها از ویژگی درخت جستجوی دودویی و اولویت‌ها از ویژگی *heap* مینیمم پیروی کنند:

- اگر v فرزند چپ u باشد آنگاه $key[v] < key[u]$
- اگر v فرزند راست u باشد آنگاه $key[v] > key[u]$
- اگر v فرزند u باشد آنگاه $priority[v] > priority[u]$

(دلیل نامگذاری "*treap*" همین ترکیب ویژگی‌ها است؛ *treap* ویژگیهای درخت جستجوی دودویی و *heap* را دارد.)

این مطلب کمک می‌کند که *treap* را به شکل زیر در نظر بگیریم. فرض کنید گره‌های $x_1, x_2, \ldots, x_n$ را با کلیدهای مربوطه در یک *treap* درج کنیم. آنگاه *treap* حاصل، درختی است که اگر گره‌ها با ترتیبی که اولویت‌های (تصادفی انتخاب شده) آنها ارائه می‌دهند، در یک درخت جستجوی دودویی معمولی درج شوند، آن درخت بوجود می‌آید به عبارت دیگر $priority[x_i] > priority[x_j]$ بدین معنا است که x_i قبل x_j درج شده است.

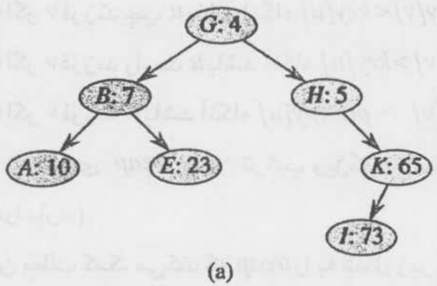
a. نشان دهیدکه برای مجموعه گره‌های داده شده $x_1, x_2, \ldots, x_n$ با کلیدها و اولویت‌های مربوطه (همه مجزا هستند)، تنها یک *treap* منحصر به فرد مرتبط با این گره‌ها وجود دارد.

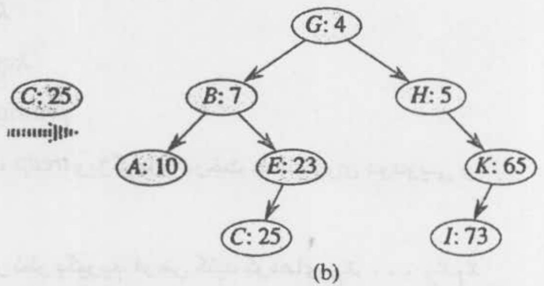
b. نشان دهید ارتفاعی که برای هر *treap* انتظار می‌رود، $\Theta(lgn)$ است و از اینرو زمان جستجوی یک مقدار در *treap* برابر $\Theta(lgn)$ است.

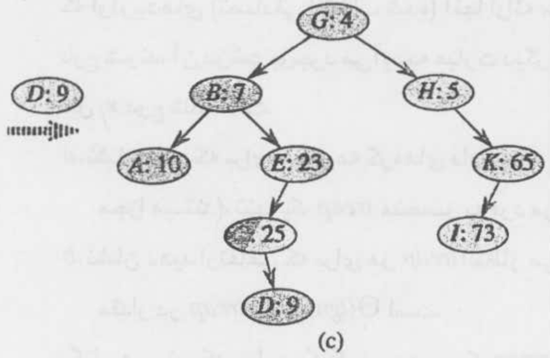
بگذارید ببینیم که چطور یک گره جدید در یک *treap* موجود درج می‌شود. اولین کاری که انجام می‌دهیم انتساب یک اولویت تصادفی به گره جدید می‌باشد. سپس به عنوان الگوریتم درج، روال *TREAP-INSERT* را فراخوانی می‌کنیم که عملکرد آن در شکل ۱۳.۱۰ نشان داده شده است.

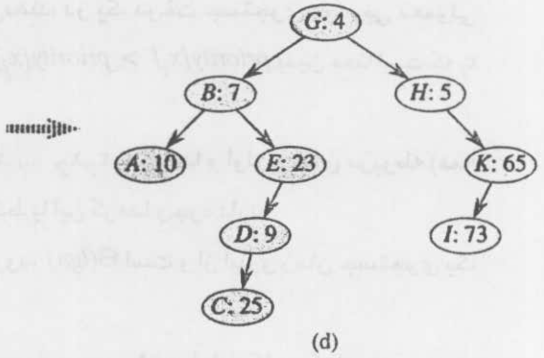
c. طرز کار *TREAP-INSERT* را توضیح دهید. این ایده را به زبان عامیانه توضیح داده و شبه کد آن را بنویسید. (راهنمایی: روال درج معمول درخت جستجوی دودویی را اجرا کرده و سپس برای بازیابی ویژگی *min-heap* چرخش‌هایی انجام دهید.)

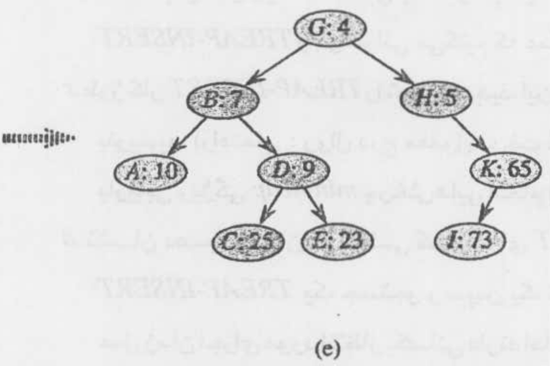
d. نشان دهید زمان اجرائی که برای *TREAP-INSERT* انتظار می‌رود، $\Theta(lgn)$ است. *TREAP-INSERT* یک جستجو و سپس یک توالی از چرخش‌ها را انجام می‌دهد. گر چه این دو عمل زمان اجرای مورد انتظار یکسانی دارند اما در عمل هزینه‌های متفاوتی دارند. یک عمل جستجو بدون تغییر *treap* اطلاعات را از روی آن می‌خواند. در مقابل، یک چرخش در درخت، اشاره‌گرهای پدر و فرزند را تغییر می‌دهد. در بیشتر کامپیوترها، عملیات خواندن خیلی سریع‌تر از عملیات نوشتن است. بنابراین ترجیح می‌دهیم که *TREAP-INSERT* تعداد کمی چرخش انجام دهد. نشان خواهیم داد تعداد چرخش‌های مورد انتظار که انجام می‌شود، با یک مقدار ثابت محدود می‌شود.

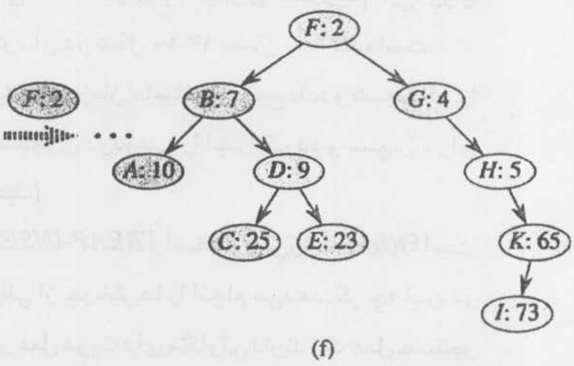












شکل ۱۳.۱۰ عمل *TREAP-INSERT*. *(a)* *treap* اولیه قبل از درج. *(b)* *treap* بعد از درج گره با کلید C و اولویت 25. *(c)*-*(d)* مراحل میانی هنگام درج گره‌ای با کلید D و اولویت 9. *(e)* *treap* بعد از انجام درج در قسمت‌های *(c)* ، *(d)* . *(f)* *treap* بعد از درج یک گره با کلید F و اولویت 2.

برای انجام این کار، به چند تعریف که در شکل ۱۳.۱۱ نشان داده شده نیاز داریم.

spine چپ یک درخت جستجوی دودویی T، مسیری از ریشه به گره‌ای با کوچکترین کلید می‌باشد. به بیان دیگر، *spine* چپ مسیری از ریشه است که تنها شامل یال‌های چپ می‌شود. به طور متقارن *spine* راست درخت T، مسیری از ریشه است که تنها شامل یال‌های راست است. *طول*[1] یک *spine* تعداد گره‌هایی است که دربردارد.

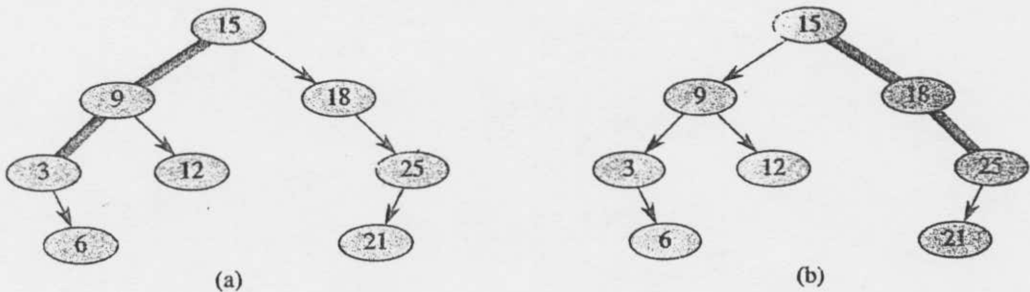
e. T را بعد از درج x با استفاده از *TREAP-INSERT* در نظر بگیرید. C را طول *spine* راست مربوط به زیردرخت چپ x قرار دهید. D را طول *spine* چپ مربوط به زیردرخت راست x در نظر بگیرید. ثابت کنید تعداد کل چرخش‌هایی که در طی مدت درج x انجام می‌شود برابر با $C+D$ است.

اکنون مقادیر مورد انتظار C و D را حساب خواهیم کرد. بدون از دست دادن کلیت فرض می‌کنیم که کلیدها $1,2, ..., n$ هستند، چون آنها را تنها با یکدیگر مقایسه می‌کنیم.

برای گره‌های x و y که $y \neq x$ است، قرار دهید $k=key[x]$ و $i=key[y]$ متغیرهای تصادفی شاخص زیر را تعریف می‌کنیم.

$X_{i,k}=I\{$ y در *spine* راست زیر درخت چپ x (در T) قرار دارد $\}$

f. نشان دهید $X_{i,k}=1$ است اگر و تنها اگر $priority[y] > priority[x]$ و $key[y] < key[x]$، و برای هر z که $key[y] < key[z] < key[x]$ باشد داریم $priority[y] < priority[z]$.



شکل ۱۳.۱۱ *spine*های یک درخت جستجو دودویی. *spine* چپ در *(a)* و *spine* راست در *(b)* تیره شده است.

g. نشان دهید

$$\Pr\{X_{i,k}=1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)} .$$

---

[1] length

h. نشان دهید

$$\mathrm{E}[C]=\sum_{j=1}^{k-1}\frac{1}{j(j+1)}=1-\frac{1}{k}\,.$$

i. با استفاده از اثباتی مشابه نشان دهید که

$$\mathrm{E}[D]=1-\frac{1}{n-k+1}\,.$$

j. نتیجه بگیرید که تعداد چرخش‌های مورد انتظار هنگام درج یک گره در $treap$ کوچکتر از 2 است.

۱۴ بهبود ساختمان داده‌ها

برخی از وضعیت‌های مهندسی به ساختمان داده‌ای بیشتر از آنچه در کتاب درسی آمده نیاز ندارند- ساختمان داده‌هایی همچون لیست‌های پیوندی دو طرفه، جدول درهم سازی، یا درخت جستجوی دودویی -ولی بسیاری دیگر از آنها به کمی خلاقیت نیاز دارند. گر چه تنها در وضعیت‌های اندکی نیاز است که یک نوع ساختمان دادهٔ کاملا جدید بسازید. بیشتر اوقات برای بهبود یک ساختمان داده کتاب درسی، ذخیرهٔ اطلاعات اضافی در آن کفایت می‌کند. آنگاه می‌توانید یک عمل جدید برای آن ساختمان داده برنامه‌نویسی کنید تا کار مورد نظر را پشتیبانی کند. بهبود یک ساختمان داده همیشه ساده نیست، چون اطلاعات اضافه شده باید بروز در آمده و توسط عملیات متداول روی ساختمان داده حفظ شوند. این فصل درمورد دو ساختمان داده‌ای که با بهبود درختهای قرمز- سیاه ساخته می‌شوند بحث می‌کند. بخش ۱۴.۱ ساختمان داده‌ای را شرح می‌دهد که عملیات آماری ترتیبی[1] روی یک مجموعه را پشتیبانی می‌کند. سپس می‌توانیم به سرعت i امین عدد کوچک در مجموعه یا مرتبه یک عنصر داده شده در کل ترتیب یک مجموعه را پیدا کنیم. بخش ۱۴.۲ فرآیند بهبود یک ساختمان داده را تجزیه کرده و یک قضیه بیان می‌کند که می‌تواند بهبود درختهای قرمز- سیاه را ساده کند. بخش۱۴.۳ از این قضیه برای کمک به طراحی ساختمان داده‌ای برای نگهداری یک مجموعه پویا از بازه‌ها مانند بازه‌های زمانی استفاده می‌کند. آنگاه می‌توانیم برای یک بازه پرس‌وجوی داده شده ، به سرعت یک بازه را درمجموعه‌ای که آن را پوشش می‌دهد پیدا کنیم.

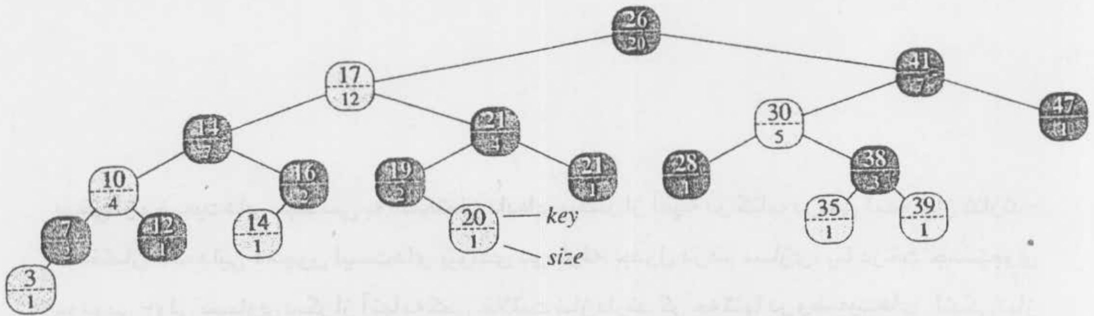
۱۴.۱ آمار ترتیبی پویا[2]

فصل ۹ مفهوم آمار ترتیبی را معرفی کرد. به خصوص، iامین آمار ترتیبی یک مجموعهٔ n عضوی که $i \in \{1,2,...,n\}$ به سادگی عنصری در مجموعه است که iامین کلید کوچکتر را دارد. دیدیم که هر آمار ترتیبی می‌تواند در زمان $O(n)$، از یک مجموعه نا مرتب بازیابی شود. در این بخش خواهیم دید که

1.order statistic

2.Dynamic order statistics

چطور درخت‌های قرمز - سیاه می‌توانند تغییر کنند تا هر آمار ترتیبی بتواند در زمان $O(lgn)$ مشخص شود. همچنین خواهیم دید چطور رتبه [1] یک عنصر - موقعیتش در یک ترتیب خطی مجموعه - نیز می‌توانند در زمان $O(lgn)$ مشخص شود.



شکل ۱۴.۱ درخت آمار ترتیبی که یک درخت قرمز - سیاه بهبود یافته است. گره‌های سایه‌زده قرمز و گره‌های تیره، سیاه هستند. علاوه بر فیلدهای معمول، هر گره x یک فیلد $size[x]$ دارد که تعداد گره‌ها در زیردرخت مشتق شده از x می‌باشد.

ساختمان داده‌ای که می‌تواند اعمال سریع آمار ترتیبی را پشتیبانی کند در شکل ۱۴.۱ نشان داده شده است. *درخت آماری ترتیبی* T [2]، به سادگی یک درخت قرمز - سیاه با اطلاعات اضافی ذخیره شده در هر گره می‌باشد. در کنار فیلدهای معمول درخت قرمز - سیاه، $key[x]$ $color[x]$، $p[x]$، $left[x]$ و $right[x]$ در یک گره x یک فیلد دیگر به نام $size[x]$ داریم. این فیلد شامل تعداد گره‌های (داخلی) در زیردرخت مشتق شده از x(که شامل خود xنیز هست) می‌باشد، به عبارت دیگر $size[nil[T]]$ را برابر 0 قرار می‌دهیم. آنگاه تساوی زیر را داریم:

$$size[x] = size[left[x]] + size[right[x]] + 1 .$$

لازم نیست که در یک درخت آمار ترتیبی، کلیدها متفاوت باشند. (برای مثال درخت شکل ۱۴.۱ دارای دو کلید با مقدار 14 و 21 می‌باشد.) در صورت وجود کلیدهای مساوی، مفهوم بالا در مورد رتبه به خوبی تعریف نشده است. این ابهام را برای درخت آمار ترتیبی توسط تعریف مرتبه یک عنصر به عنوان موقعیتی که در پیمایش میان ترتیب درخت دارد از بین می‌بریم. به عنوان مثال در شکل ۱۴.۱، کلید 14 که در یک گره سیاه ذخیره شده دارای رتبه 5 و کلید 14 که در یک گره قرمز ذخیره شده دارای مرتبه 6 می‌باشد.

1.rank
2.order- statistic-tree

بازیابی یک عنصر با مرتبه داده شده

قبل از اینکه نشان دهیم که چطور اطلاعات اندازه، در حین درج و حذف حفظ می‌شوند، اجازه دهید که پیاده‌سازی دو پرس‌وجوی آماری ترتیبی که از این اطلاعات اضافی استفاده می‌کنند را بررسی کنیم. با عمل بازیابی یک عنصر با رتبه داده شده شروع می‌کنیم. روال *OS-SELECT(x,i)* یک اشاره‌گر به گره‌ای که شامل iامین کلید کوچکتر در زیردرخت مشتق شده از x می‌باشد را برمی‌گرداند. برای پیدا کردن i امین کلید کوچک‌تر در درخت آمار ترتیبی T، *OS-SELECT(root[T])* را فراخوانی می‌کنیم.

```
OS-SELECT(x, i)
1  r ← size[left[x]]+1
2  if i = r
3     then return x
4  elseif i < r
5     then return OS-SELECT(left[x], i)
6  else return OS-SELECT(right[x], i − r)
```

ایده واقع در پس *OS-SELECT* شبیه ایده الگوریتم‌های انتخاب در فصل ۹ می‌باشد. مقدار *size[left[x]]* برابر تعداد گره‌هایی است که در پیمایش میان ترتیب زیردرخت مشتق شده از x قبل از x می‌آیند. بنابراین *size[left[x]]+1* مرتبه x در زیردرخت مشتق شده از x می‌باشد.

در خط اول *OS-SELECT*، مقدار r که رتبه گره x در زیردرخت مشتق شده از x می‌باشد را حساب می‌کنیم. اگر $i=r$ باشد آنگاه گره x iامین عنصر کوچک‌تر است، بنابراین در خط ۳، x را برمی‌گردانیم. اگر $i<r$ باشد آنگاه iامین عنصر کوچکتر در زیردرخت چپ x واقع است .بنابراین روال با مقدار *left[x]* در خط ۵ تکرار می‌شود. اگر $i>r$ باشد آنگاه i امین عنصر کوچکتر عنصر در زیردرخت راست x واقع است. چون r عنصر در زیردرخت مشتق شده از x وجود دارد که قبل از زیردرخت راست x در پیمایش میان ترتیب درخت می‌آیند iامین عنصر کوچک‌تر زیردرخت مشتق شده از x، $(i-r)$امین عنصر کوچک‌تر در زیردرخت مشتق شده از *right[x]* است. این عنصر در خط ۶ به طور بازگشتی مشخص می‌شود.

برای مشاهدهٔ طرز کار *OS-SELECT*، جستجوی *17* امین عنصر مینیمم در درخت آمار ترتیبی شکل ۱۴.۱ را در نظر بگیرید. ابتدا با x به عنوان ریشه با کلید *26* و $i=17$ شروع می‌کنیم. چون اندازه زیردرخت چپ *26* برابر *12* است، مرتبه آن *13* می‌باشد. بنابراین می‌دانیم که گره با مرتبه *17*، $17-13=4$ امین عنصر کوچکتر در زیردرخت راست *26* است. بعد از فراخوانی بازگشتی، x گره‌ای با کلید *41* و $i=4$ است. چون اندازه زیردرخت چپ *41* برابر *5* است، رتبه آن در زیردرختش *6* است. بنابراین می‌دانیم که گره با رتبه *4*، *4* امین عنصر کوچکتر در زیر درخت چپ *41* است. بعد از فراخوانی

بازگشتی، x گره‌ای با کلید 30 است و رتبه آن د ر زیردرختش 2 است. بنابراین، یک بار دیگر بازگشت می‌کنیم تا $4-2=2$ امین عنصر کوچکتر را در زیردرخت مشتق شده از گره با کلید 38 پیدا کنیم. اکنون فهمیدیم که زیردرخت چپ آن دارای اندازه 1 است که به این معنا است که دومین عنصر کوچکتر است. بنابراین یک اشاره‌گر به گره با کلید 38 توسط روال برگردانده می‌شود.

چون هر فراخوانی بازگشتی یک سطح در درخت آمار ترتیبی به پایین می‌رود، زمان کل *OS-SELECT* در بدترین حالت، متناسب با ارتفاع درخت است. چون این درخت یک درخت قرمز – سیاه است، ارتفاع آن گره برابر $O(lgn)$ است که n تعداد گره‌ها است. بنابراین، زمان اجرای *OS-SELECT* برای یک مجموعه پویا n عنصری، $O(lgn)$ است.

تعیین رتبه یک عنصر

روال *OS-RANK* با دریافت اشاره‌گری به گره x در درخت آمار ترتیبی T، موقعیت x را در یک ترتیب خطی که توسط یک پیمایش میان ترتیب درخت T تعیین می‌گردد برمی‌گرداند.

```
OS-RANK(T, x)
1  r ← size[left[x]] + 1
2  y ← x
3  while y ≠ root[T]
4      do if y = right[p[y]]
5           then r ← r + size[left[p[y]]] + 1
6         y ← p[y]
7  return r
```

روال به صورت زیرکار می‌کند. مرتبه x می‌تواند به عنوان تعداد گره‌های مقدم بر x در پیمایش میان ترتیب درخت، به علاوه ۱ برای خود x در نظر گرفته شود. *OS-SELECT* ثابت حلقهٔ زیر را حفظ می‌کند.

در شروع هر تکرار در حلقه *while* در خطوط ۶–۳، r رتبه $key[x]$ در زیردرخت مشتق شده از گره y می‌باشد. ما از این ثابت حلقه برای نشان دادن اینکه *OS-SELECT* درست کار می‌کند به صورت زیر استفاده می‌کنیم.

مقدار دهی اولیه: قبل از اولین تکرار، خط ۱، r را رتبه $key[x]$ در زیردرخت مشتق شده از x قرار می‌دهد. انتساب $y \leftarrow x$ در خط ۲، ثابت را برای اولین باری که تست خط ۳ انجام می‌شود صحیح می‌کند.

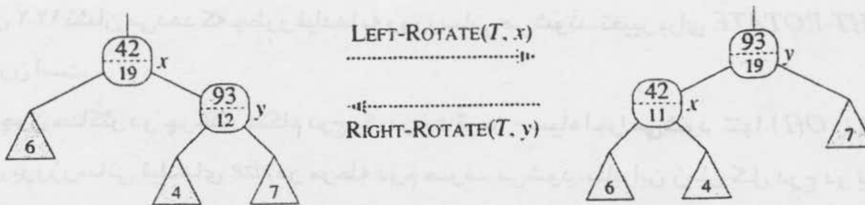
نگهداری: در انتهای هر تکرار حلقهٔ *while*، انتساب $y \leftarrow p[y]$ را انجام می‌دهیم. بنابراین باید نشان دهیم اگر r رتبه $key[x]$ در زیردرخت مشتق شده از y در شروع بدنه حلقه باشد، آنگاه r در پایان بدنه

حلقه رتبه $key[x]$ در زیردرخت مشتق شده از $p[y]$ است. درهر تکرار حلقه *while* زیر درخت مشتق شده از $p[y]$ را در نظر می‌گیریم. قبلاً تعداد گره‌هایی که در زیردرخت مشتق شده از گره y بر x در پیمایش میان ترتیب مقدم هستند را حساب کرده‌ایم. بنابراین باید گره‌هایی که در زیردرخت مشتق شده از همزاد y، در پیمایش میان ترتیب بر x مقدم هستند را به اضافه ۱ برای $p[y]$ اگر آن هم بر x مقدم باشد، اضافه کنیم. اگر y یک فرزند چپ باشد، آنگاه نه $p[y]$ و نه هیچ گره‌ای در زیردرخت راست $p[y]$ بر x مقدم نیستند، بنابراین r را تنها رها می‌کنیم. در غیر اینصورت y یک فرزند راست است و همه گره‌ها در زیردرخت چپ $p[y]$ همچون خود $p[y]$ بر x مقدم هستند. بنابراین در خط ۵، $size[left[p[y]]]+1$ را به مقدار فعلی r اضافه می‌کنیم.

خاتمه: حلقه زمانی پایان می‌یابد که $y=root[T]$ باشد، بنابراین زیردرخت مشتق شده از y، کل درخت می‌باشد. لذا مقدار r، رتبهٔ $key[x]$ در کل درخت می‌باشد.

به عنوان مثال، زمانی که *OS-RANK* را روی درخت آمار ترتیبی شکل ۱۴.۱ برای پیدا کردن رتبه گره‌ای با کلید 38 اجرا می‌کنیم، توالی مقادیر $key[y]$ و r در بالای حلقهٔ *while* را بصورت زیر داریم

iteration	$key[y]$	r
1	38	2
2	30	4
3	41	4
4	26	17



شکل ۱۴.۲ به روزرسانی اندازه زیردرخت‌ها هنگام چرخش. پیوندی که چرخش حول آن انجام می‌شود. روی دو گره‌ای که فیلد *size* آنها نیاز به بروز رسانی دارد، متلاقی است. به روزرسانی‌ها، محلی هستند و تنها نیاز به اطلاعات *size* ذخیره شده در x و y ریشه‌های زیردرخت‌هایی که با مثلث نشان داده شده‌اند دارند.

رتبهٔ 17 برگردانده می‌شود. چون هر تکرار در حلقه *while* زمان $O(1)$ را صرف کرده و y در هر تکرار یک سطح به بالای درخت می‌رود زمان اجرای *OS-RANK* در بدترین حالت متناسب با ارتفاع درخت است: $O(lgn)$ روی یک درخت آمار ترتیبی n گره‌ای.

حفظ اندازه زیردرخت‌ها

با دریافت فیلد *size* برای هر گره، *OS-SELECT* و *OS-RANK* می‌توانند به سرعت اطلاعات آمار ترتیبی را حساب کنند. ولی کار ما بیهوده خواهد بود. مگر اینکه این فیلدها بتوانند به طور مؤثری توسط اعمال تغییر دهنده پایداری روی درخت‌های قرمز - سیاه حفظ شوند. اکنون نشان خواهیم داد اندازه زیردرخت‌ها هم برای درج و هم برای حذف، بدون تأثیر روی زمان اجرای مجانبی هر یک، می‌تواند حفظ شود.

در بخش ۱۳.۳ ذکر کردیم که درج یک درخت قرمز - سیاه شامل دو مرحله است. مرحلهٔ اول از ریشه به طرف پایین درخت حرکت کرده، رنگ‌ها را تغییر می‌دهد و در نهایت چرخش‌هایی برای برقرار ساختن ویژگی‌های قرمز - سیاه انجام می‌دهد.

برای حفظ اندازه زیردرخت‌ها در مرحله اول، به راحتی *size[x]* را برای هر گره *x* در مسیری که از ریشه تا برگ‌ها می‌پیماید افزایش می‌دهیم. گره جدیدی که اضافه می‌شود، اندازهٔ *1* را برمی‌گیرد. چون *O(lgn)* گره در مسیر پیموده شده وجود دارد، هزینه اضافی حفظ فیلدهای *size* برابر *O(lgn)* می‌باشد.

در مرحله دوم، تنها تغییرات ساختاری درخت قرمز - سیاه اصلی، توسط چرخش‌های که حداکثر دو واحد هستند به وجود می‌آید. بیشتر اوقات چرخش، عملی محلی است: تنها فیلد *size* دو گره نامعتبر می‌شود. پیوندی که چرخش حول آن انجام شده روی گره متلاقی است. با مراجعه به کد *LEFT-ROTATE(T,X)* در بخش ۱۳.۲ خطوط زیر را به آن اضافه می‌کنیم.

```
12  size[y] ← size[x]
13  size[x] ← size[left[x]] + size[right[x]] + 1
```

شکل ۱۴.۲ نشان می‌دهد که چطور فیلدها به روزرسانی می‌شوند. تغییر برای *RIGHT-ROTATE* نیز متقارن است.

چون حداکثر دو چرخش هنگام درج یک درخت قرمز - سیاه اجرا می‌شود تنها *O(1)* زمان اضافی برای بروزرسانی فیلدهای *size* در مرحله دوم صرف می‌شود .بنابراین زمان کل درج در یک درخت آمار ترتیبی با *n* گره، *O(lgn)* است که به طور مجانبی با زمان درخت قرمز - سیاه معمولی یکسان است. حذف از یک درخت قرمز - سیاه نیز شامل دو مرحله است: اول روی درخت جستجوی اصلی عمل کرده و دوم حداکثر سه چرخش انجام می‌دهد. و در غیر اینصورت هیچ تغییر ساختاری ایجاد نمی‌کند. (بخش ۱۳.۴ را ملاحظه کنید.) مرحله اول، یک گره *y* را حذف می‌کند. برای بروزرسانی اندازه‌های زیردرخت‌ها به راحتی یک مسیر از گره *y* رو به بالا به طرف ریشه را پیموده و فیلد *size* هر گره در مسیر را کاهش می‌دهیم. چون این مسیر در یک درخت قرمز - سیاه با *n* گره، طول *O(lgn)* دارد. *O(1)* چرخش در مرحله دوم حذف می‌تواند مانند حالتی مشابه با درج، انجام گیرد.بنابراین هم درج و هم

حذف که شامل حفظ فیلدهای *size* می‌باشند برای یک درخت آمار ترتیبی با n گره زمان $O(lgn)$ صرف می‌کنند.

تمرین‌ها

۱- ۱۴.۱ نشان دهید چگونه $OS\text{-}SELECT(T,10)$ روی درخت قرمز - سیاه T در شکل ۱۴.۱ عمل می‌کند.

۲- ۱۴.۱ نشان دهید چگونه $OS\text{-}RANK(T,X)$ روی درخت قرمز - سیاه T در شکل ۱۴.۱ و گره x با $key[x]=35$ عمل می‌کند.

۳- ۱۴.۱ صورت غیر بازگشتی *OS-SELECT* را بنویسید.

۴- ۱۴.۱ یک روال بازگشتی $OS\text{-}KEY\text{-}RANK(T,k)$ بنویسید که به عنوان ورودی، درخت آماری ترتیبی T و یک کلید k را گرفته و مرتبه k ارائه شده در مجموعه پویا توسط T را برگرداند. فرض کنید کلیدهای T متفاوت هستند.

۵- ۱۴.۱ با دریافت یک عنصر x در درخت آمار ترتیبی با n گره و عدد طبیعی i چطور می‌توان i امین عنصر ما بعد گره x را در ترتیب خطی درخت در زمان $O(lgn)$ مشخص کرد.

۶- ۱۴.۱ مشاهده می‌گردد هنگامیکه به فیلد *size* یک گره، در هر یک از روالهای *OS-SELECT* یا *OS-RANK* مراجعه شده است، تنها از آن برای محاسبه رتبه یک گره در زیردرخت مشتق شده از همان گره استفاده شده است. از اینرو، فرض کنید در هر گره رتبه آن گره در زیردرختی که از آن مشتق می‌شود را ذخیره کنیم. نشان دهید چگونه این اطلاعات هنگام درج و حذف می‌توانند حفظ شوند. (به خاطر داشته باشید که این دو عمل ممکن است باعث چرخش شوند.)

۷- ۱۴.۱ نشان دهید که چطور می‌توان برای شمردن تعداد **وارونگی‌ها**[1] در یک آرایه با اندازهٔ n در زمان $O(nlgn)$ از یک درخت آمار ترتیبی استفاده کرد. (مسئله ۴-۲ را ملاحظه کنید.)

*۸- ۱۴.۱ n وتر روی یک دایره در نظر بگیرید، هر یک با نقاط پایانی اش تعریف می‌شود. یک الگوریتم با زمان $O(nlgn)$ بنویسید که تعداد جفت‌های وترهایی را که داخل دایره با یکدیگر برخورد دارند مشخص کند. (برای مثال، اگر n وتر همگی قطر دایره باشند که در مرکز متلاقی‌اند آنگاه جواب درست $\binom{n}{2}$ می‌باشد.) فرض کنید هیچ دو وتری در یک نقطه پایانی مشترک نیستند.

۱۴.۲ چطور یک ساختمان داده را بهبود بخشیم

فرآیند بهبود یک ساختمان داده اصلی برای پشتیبانی کارهای اضافی، غالباً در طرح الگوریتم اتفاق

---

[1] inversions

می‌افتد. در بخش بعد نیز از آن برای طراحی ساختمان داده‌ای که اعمال روی بازه‌ها را پشتیبانی می‌کند استفاده می‌شود. در این بخش مراحلی که در چنین بهبودی قرار دارند را بررسی خواهیم کرد. یک قضیه را نیز اثبات خواهیم کرد که به ما اجازه می‌دهد که درخت‌های قرمز - سیاه را در بسیاری از زمینه‌ها به راحتی بهبود بخشیم.

بهبود یک ساختمان داده می‌تواند به چهار مرحله تقسیم شود:

۱. انتخاب یک ساختمان داده اصلی

۲. مشخص کردن اطلاعات اضافی برای نگهداری در ساختمان داده اصلی،

۳. مشخص کردن این اطلاعات اضافی می‌توانند برای اعمال تغییر دهنده روی ساختمان داده اصلی نگهداری شوند و

۴. توسعه اعمال جدید

مانند یک روش طراحی قاعده‌مند، نباید به طور کورکورانه از مرحله‌هایی که به ترتیب داده شده‌اند پیروی کنید. بیشتر کار طراحی شامل یک عنصر امتحان و خطامی‌باشد. و پیشروی در همه مرحله‌ها معمولا به طور موازی انجام می‌شود. به عنوان مثال اگر نتوانیم اطلاعات اضافی را به طور کارآمدی نگه داریم، مشخص کردن اطلاعات اضافی و توسعه اعمال جدید (مرحله‌های ۲ و ۴) هیچ فایده‌ای ندارد. با این وجود این روش چهار مرحله‌ای، یک کانون خوب برای تلاشمان در بهبود یک ساختمان داده فراهم می‌کند، و این روش یک روش خوب برای سازماندهی مستندات یک ساختمان داده بهبود یافته است.

این مراحل را در بخش ۱۴.۱ برای طراحی درخت‌های آمار ترتیبی دنبال کردیم. برای مرحله ۱، درخت‌های قرمز - سیاه را به عنوان ساختمان داده اصلی انتخاب کردیم. یک نشانه خوب برای متناسب بودن درخت‌های قرمز - سیاه، از پشتیبانی موثر آنها از اعمال دیگر مجموعه‌های پویا روی یک ترتیب کلی مانند *PREDECESSOR, MINIMUM*، *SUCCESSOR* ،*MAXIMUM* بوجود می‌آید.

برای مرحله ۲، فیلد *size* را ایجاد کردیم که در آن هر گره x اندازه زیردرخت مشتق شده از x را ذخیره می‌کند. معمولاً اطلاعات اضافی اعمال را کارآمدتر می‌کنند. برای مثال می‌توانستیم *OS-SELECT* و *OS-RANK* را تنها با استفاده از کلیدهای ذخیره شده در درخت پیاده‌سازی کنیم، ولی در زمان $O(lgn)$ نمی‌توانستند اجرا شوند. بعضی اوقات اطلاعات اضافی همانطور که در تمرین ۱-۴.۲ آمده است به جای اینکه داده‌ها باشند، اطلاعات اشاره‌ای هستند.

برای مرحله ۳، مطمئن می‌شویم که درج و حذف در حالی که هنوز در زمان $O(lgn)$ اجرا می‌شوند، می‌توانند فیلدهای *size* را حفظ کنند. به طور ایده آل، تنها یک تعداد کمی از عناصر ساختمان داده باید برای حفظ اطلاعات اضافی، بروزرسانی شوند. برای مثال، اگر به سادگی در هر گره رتبه آن در درخت را ذخیره می‌کردیم، روال‌های *OS-SELECT* و *OS-RANK* سریع اجرا می‌شدند. ولی درج یک عنصر

مینیمم جدید باعث تغییر این اطلاعات در همه گره‌های درخت می‌شد. زمانیکه به جای آن اندازه زیردرخت را ذخیره می‌کنیم درج یک عنصر جدید باعث تغییر اطلاعات در تنها $O(lgn)$ گره می‌شود.

برای مرحله ۴، عملیات *OS-RANK, OS-SELECT* را توسعه دادیم. بعد همه اینها نیاز برای عملیات جدید به دلیل این است که برای بهبود یک ساختمان داده در اولین مکان به دردسر می‌افتیم. گاهی اوقات، از اطلاعات اضافی به جای توسعه اعمال جدید، برای بالا بردن سرعت اعمال موجود استفاده می‌کنیم، همان طور که در تمرین ۱-۱۴.۲ آمده است.

بهبود درخت‌های قرمز - سیاه

وقتی درخت‌های قرمز -سیاه پایه و اساس یک ساختمان داده بهبود یافته را تشکیل می‌دهند، می‌توانیم اثبات کنیم که انواع اصلی اطلاعات اضافی می‌توانند همیشه به طور مؤثری توسط درج و حذف حفظ شوند، در نتیجه مرحله ۳ خیلی آسان می‌شود. اثبات قضیه زیر، مشابه بحث بخش ۱۴.۱ در مورد این که فیلد *size* می‌تواند برای درخت‌های آمار ترتیبی حفظ شود، می‌باشد.

قضیه ۱۴.۱ (بهبود یک درخت قرمز - سیاه)

f را یک فیلدی قرار دهید که یک درخت قرمز - سیاه T با n گره را بهبود بخشد، و فرض کنید محتوی f برای یک گره x می‌توانند تنها با استفاده از اطلاعات گره x، $left[x]$ و $right[x]$، شامل $f[right+[x]]$ و $f[left[x]]$ محاسبه شوند. سپس می‌توانیم مقادیر f را در همه گره‌های T در طول درج و حذف حفظ کنیم. بدون این که تأثیر مجانبی به روی اجرا $O(lgn)$ این اعمال داشته باشد.

اثبات ایده اصلی اثبات این است که تغییر در یک فیلد f یک گره تنها در اجداد x در درخت منتشر می‌شود. به عبارت دیگر تغییر $f[x]$ ممکن است باعث بروز رسانی $f[p[x]]$ شوند نه چیز دیگری. بروزرسانی $f[p[x]]$ ممکن است باعث بروزرسانی $f[p[p[x]]]$ شود، نه چیز دیگری و همینطور تا بالای درخت زمانی که $f[root[T]]$ بروز می‌شود. هیچ گره دیگری به مقدار جدید بستگی ندارد، بنابراین فرآیند پایان می‌یابد. چون ارتفاع درخت قرمز - سیاه، برابر $O(lgn)$ است، تغییر فیلد f در یک گره، زمان $O(lgn)$ برای به روزرسانی گره‌هایی که به این تغییر وابسته‌اند را صرف می‌کند.

درج گره x در T، شامل دو مرحله است. (بخش ۱۳.۳ را ملاحظه کنید) در طول مرحله اول، x به عنوان یک فرزند گره موجود $p[x]$ درج می‌شود. مقدار $f[x]$ می‌تواند در زمان $O(1)$ محاسبه شود، چون بنا به فرض تنها بستگی به فیلدهای دیگر خود x و اطلاعات فرزندان x دارد، ولی فرزندان x هر دو محافظ $nil[T]$ می‌باشند. وقتی $f[x]$ محاسبه می‌شود تغییر در بالای درخت منتشر می‌شود. بنابراین زمان کل مرحله اول درج ، $O(lgn)$ است. در طول مرحله دوم، تنها تغییرات ساختاری درخت توسط چرخش‌ها بوجود می‌آیند. چون تنها دو گره در یک چرخش تغییر می‌کنند، زمان کل بروز رسانی فیلد f برای هر

چرخش $O(lgn)$ است. چون تعداد چرخش‌ها در هنگام درج حداکثر دو می‌باشد، زمان کل عمل درج برابر $O(lgn)$ است.

مانند درج، حذف هم دو مرحله دارد. (بخش ۱۳.۴ را ملاحظه کنید.) در مرحله اول اگر گره حذف شده با مابعد خود جایگزین شود و زمانیکه گره یا مابعد آن حذف شود اجرای بروزرسانی‌های f توسط این تغییرات ایجاد می‌شود. و از آنجایی که تغییرات، درخت را به طور محلی تغییر می‌دهند حداکثر هزینه $O(lgn)$ را برای انتشار بروزرسانی‌های f نیاز دارد. بنابراین، مانند درج، زمان کل حذف نیز $O(lgn)$ است.

در بسیاری از موارد، همچون نگهداری فیلدهای *size* در درخت آمار ترتیبی، هزینه به روزرسانی‌ها بعد از یک چرخش $O(1)$ می‌باشد نه $O(lgn)$، که از اثبات قضیه ۱۴.۱ نتیجه گرفته می‌شود. تمرین ۴-۱۴.۲ نمونه‌ای را بیان می‌کند.

تمرین‌ها

۱۴.۲-۱ نشان دهید چطور عملیات *SUCCESSOR* ،*MAXIMUM* ، *MINIMUM* و *PREDECESSOR* مجموعه پویا، هر یک می‌توانند در بدترین حالت در زمان $O(1)$ روی یک درخت آمار ترتیبی بهبود یافته پشتیبانی شوند. کارآیی مجانبی اعمال دیگر روی درخت‌های آمار ترتیبی نباید تأثیر بپذیرند. (راهنمایی: به گره‌ها اشاره‌گر اضافه کنید.)

۱۴.۲-۲ آیا ارتفاع سیاه گره‌ها در یک درخت قرمز - سیاه می‌تواند به عنوان فیلد در گره‌های درخت نگهداری شود بدون این که روی کارآیی مجانبی عمل روی درخت قرمز - سیاه تأثیری بگذارد؟ نشان دهید چگونه و یا دلیل عدم آن را توضیح دهید.

۱۴.۲-۳ آیا ارتفاع گره‌ها در یک درخت قرمز - سیاه می‌تواند به عنوان فیلد در گره‌های درخت بصورت کارآمدی نگهداری شود؟ نشان دهید چگونه، یا اگر پاسخ منفی است آن را ثابت کنید.

۱۴.۲-۴ علامت $\otimes$ را یک عملگر شرکت‌پذیری دودویی قرار داده و a را یک فیلد که در هر گره یک درخت قرمز - سیاه نگهداری می‌شود قرار دهید. فرض کنید می‌خواهیم در هر گره x یک فیلد اضافی f اضافه کنیم. به طوریکه فرمول که $x_1, x_2, \ldots, x_n$ لیست میان ترتیب از گره‌های زیر درخت مشتق شده از x می‌باشد. نشان دهید که فیلدهای f می‌توانند به طور مجانبی در زمان $O(1)$ بعد از یک چرخش بروزرسانی شوند. ایده خود را کمی تغییر دهید که نشان دهید فیلدهای *size* در درخت آمار ترتیبی می‌توانند در زمان $O(1)$ برای هر چرخش نگهداری شوند.

۱۴.۲-۵ می‌خواهیم درخت‌های قرمز - سیاه را با یک عمل *RB-ENUMERATE(x,a,b)* بهبود بخشیم که این عمل همه کلیدهای k که $a \le k \le b$ را در یک درخت قرمز - سیاه مشتق شده از x به عنوان خروجی برمی‌گرداند. توضیح دهید وقتی x تعداد کلیدهای خروجی و n تعداد گره‌های داخلی درخت

است چطور *RB-ENUMERATE* می‌تواند در زمان $\Theta(m+lgn)$ پیاده‌سازی شود. (راهنمایی: هیچ نیازی به اضافه کردن فیلدهای جدید به درخت قرمز - سیاه نیست.)

۱۴.۳ درخت‌های بازه‌ای

در این بخش درخت‌های قرمز - سیاه را برای پشتیبانی اعمال روال روی مجموعه‌های پویایی از بازه‌ها بهبود می‌بخشیم. یک **بازه بسته**، یک جفت عدد حقیقی مرتب شده $[t_1,t_2]$ که $t_1 \le t_2$ است. بازه $[t_1,t_2]$ مجموعه $\{t \in IR: t_1 \le t \le t_2\}$ را نشان می‌دهد. بازه‌های باز و نیمه باز به ترتیب هر دو یا یکی از نقاط پایانی مجموعه را حذف می‌کنند. در این بخش فرض می‌کنیم که بازه‌ها بسته هستند؛ و تعمیم برای بازه‌های **باز و نیمه باز** به طور ذهنی آسان می‌باشد.

بازه‌ها برای نمایش رویدادهایی که هر یک، یک دوره زمانی پیوسته را اشغال می‌کنند مناسب هستند. برای مثال ممکن است بخواهیم پایگاه دادهٔ بازه‌های زمانی را برای پیدا کردن اینکه در طول یک بازه داده شده چه رویدادهایی رخ داده است، جستجو کنیم. ساختمان داده این بخش، یک مفهوم موثر برای حفظ چنین پایگاه داده بازه‌ای را فراهم می‌کند.

می‌توانیم بازهٔ $[t_1,t_2]$ را به عنوان شی i با فیلدهای $low[i]=t_1$ **(حد پایین)** و $hight[i]=t_2$ **(حد بالا)** نمایش دهیم. می‌گوئیم بازه‌های i و i' با یکدیگر تداخل دارند. اگر $i \cap i' \neq \phi$ و به عبارت دیگر اگر $low[i] \le high[i']$ و $low[i'] \le high[i]$ باشند. هر دو بازه i و i' در سه **قسمتی بازه‌ای**[1] صدق می‌کنند. به عبارت دیگر دقیقاً یکی از ویژگی‌های زیر برقرار است:

a. i و i' با هم همپوشانی داشته باشند.

b. i در چپ i' قرار داشته باشد(یعنی $high[i] < low[i']$).

c. i در راست i' قرار داشته باشد (یعنی $high[i'] < low[i]$).

درخت بازه‌ای[2] یک درخت قرمز - سیاه است که یک مجموعه پویا از عناصر را با هر عنصر x شامل یک بازه $int[x]$ نگه می‌دارد. درخت‌های بازه‌ای عملیات زیر را پشتیبانی می‌کنند.

INTERVAL-INSERT(T,x) عنصر x که فیلد *int* آن شامل یک بازه است را به درخت بازه‌ای T اضافه می‌کند و *INTERVAL-DELETE(T,x)* عنصر x را از درخت بازه‌ای T حذف می‌کند. *INTERVAL-SEARCH(T,i)* یک اشاره‌گر به عنصر x را در درخت بازه‌ای T بر می‌گرداند که $int[x]$ با بازهٔ i تداخل دارد یا اگر هیچ عنصری در مجموعه نیست $nil[T]$ را برمی‌گرداند.

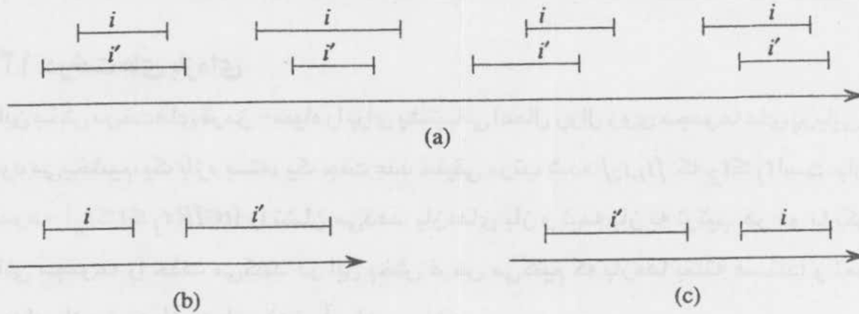
شکل ۱۴.۴ نشان می‌دهد که چطور یک درخت بازه، یک مجموعه از بازه‌ها را نشان می‌دهد. همان‌طور که در بخش ۱۴.۲ طراحی یک درخت بازه‌ای و اعمالی که روی آن اجرا می‌شوند را مرور

---

1. interval trichotomy

2. tree interval

می‌کنیم. روش چهار مرحله‌ای در بخش ۱۴.۲ را پیگیری می‌کنیم.



شکل ۱۴.۳ سه قسمتی بازه‌ای برای دو بازه بسته i و i'.
a.اگر i و i' همپوشانی داشته باشند، چهار وضعیت وجود دارد؛ در هر یک $low[i] \leq high[i']$ و $low[i'] \leq high[i]$. *b*.بازه‌ها با هم همپوشانی ندارند و $high[i] < low[i']$. *c*.بازه‌ها همپوشانی نمی‌کنند و $high[i'] < low[i]$.

مرحله ۱: ساختمان داده اصلی

درخت قرمز - سیاه را انتخاب می‌کنیم که در آن گره x شامل یک بازه $int[x]$ است وکلید x حد پایین بازه یعنی $low[int[x]]$ است. بنابراین، یک پیمایش میان ترتیب درخت این ساختمان داده، بازه‌ها را به صورت مرتب بر اساس حد پایین لیست می‌کند.

مرحله ۲: اطلاعات اضافی

به اضافه خود بازه‌ها، هر گره x شامل یک مقدار $max[x]$ است که ماکزیمم مقدار نقطه پایانی که در زیردرخت مشتق شده از x ذخیره شده است می‌باشد.

مرحله ۳: نگهداری اطلاعات

باید مشخص کنیم که حذف و درج می‌توانند در زمان $O(lgn)$ روی یک درخت بازه‌ای n گره‌ای انجام شوند. می‌توانیم $max[x]$ را با دریافت بازه $int[x]$ و مقدار فیلد max فرزندان گره x تعیین کنیم.

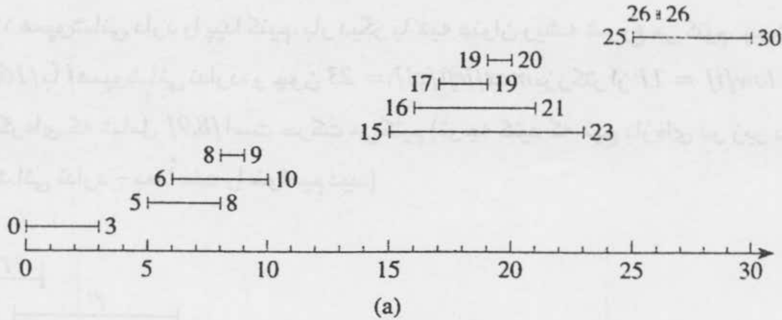
$$max[x] = \max(high[int[x]], max[left[x]], max[right[x]]) .$$

بنابراین، با توجه به قضیه ۱۴.۱، حذف و درج در زمان $O(lgn)$ اجرا می‌شوند. در حقیقت، همان طور که در تمرین‌های ۱- ۱۴.۳ و ۴- ۱۴.۲ نشان داده شده، به روزرسانی فیلدهای max بعد از یک چرخش می‌تواند در زمان $O(1)$ انجام شود.

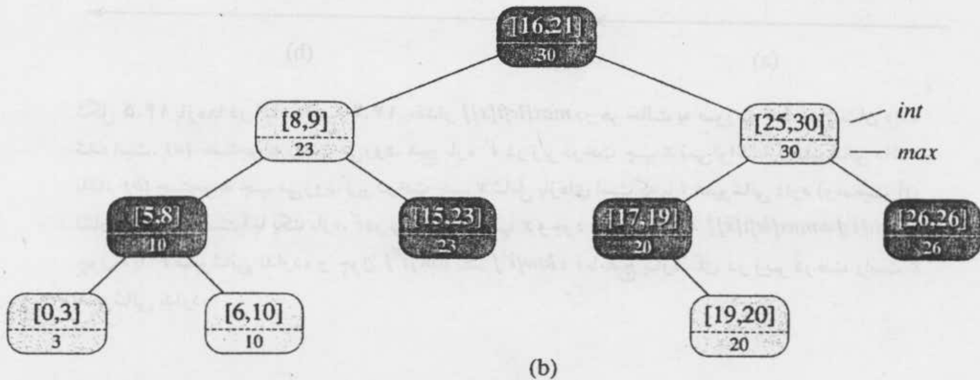
مرحله ۴: توسعه اعمال جدید

تنها عمل جدیدی که نیاز داریم *INTERVAL-SEARCH(T,i)* است که یک گره در درخت T که بازه‌اش با بازهٔ i همپوشانی دارد پیدا می‌کند. اگر هیچ بازه‌ای در درخت وجود نداشته باشد که با i همپوشانی کند، یک اشاره گر محافظ *nil[T]* برگردانده می‌شود.

```
INTERVAL-SEARCH(T, i)
1  x ← root[T]
2  while x ≠ nil[T] and i does not overlap int[x]
3      do if left[x] ≠ nil[T] and max[left[x]] ≥ low[i]
4            then x ← left[x]
5            else x ← right[x]
6  return x
```



(a)



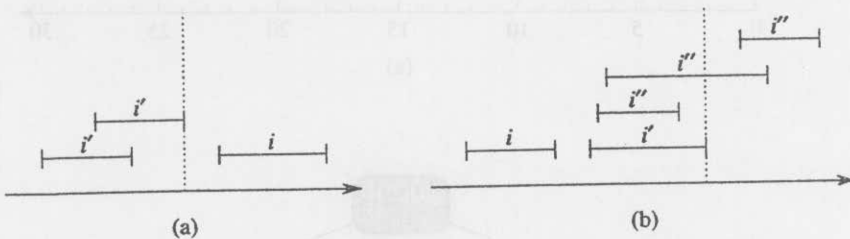
(b)

شکل ۱۴.۴ یک درخت بازه. *(a)* یک مجموعه با *10* بازه نشان داده شده که با توجه به حد پایین، از پایین به بالا ذخیره شده‌اند. *(b)* درخت بازه‌ای که آنها را نشان می‌دهد. یک پیمایش میان ترتیب درخت، گره‌ها را به ترتیب حد پایین آنها لیست می‌کند.

جستجو برای بازه‌ای که با i همپوشانی دارد با x به عنوان ریشه درخت شروع شده و به طرف پائین پیش می‌رود. زمانی که یک فاصله همپوشان پیدا شود یا x به محافظ $nil[T]$ اشاره کند جستجو پایان می‌یابد. چون هر تکرار حلقه اصلی زمان $O(1)$ را صرف می‌کند و چون ارتفاع یک درخت قرمز-سیاه با n گره، $O(lgn)$ است، روال *INTERVAL-SEARCH* زمان $O(lgn)$ را صرف می‌کند.

قبل از اینکه ببینیم چرا *INTERVAL-SEARCH* درست است، بیایید طرز کار آن روی درخت بازه‌ای شکل ۱۴.۴ را امتحان کنیم. فرض کنید می‌خواهیم بازه‌ای را پیدا کنیم که با بازهٔ $i= [22,25]$ همپوشانی دارد. ابتدا با x به عنوان ریشه شروع می‌کنیم که x شامل $[16,21]$ بوده و با i همپوشانی ندارد چون $max[left[x]] = 23$ بزرگتر از $low[i] = 22$ است، حلقه با x به عنوان فرزند چپ ریشه ادامه می‌یابد - که این گره شامل $[8,9]$ بوده و با i نیز همپوشانی ندارد -. این بار $max[left[x]] = 10$ کوچکتر از $low[i] = 22$ است بنابراین حلقه با فرزند راست x به عنوان x جدید ادامه می‌یابد. بازه $[15,23]$ که در این گره ذخیره شده است با i همپوشانی دارد، بنابراین این گره را بر می‌گرداند.

به عنوان نمونه‌ای از جستجوی ناموفق، فرض کنید می‌خواهیم بازه‌ای که با $i=[11,14]$ در درخت بازه‌ای شکل ۱۴.۴ همپوشانی دارد را پیدا کنیم. بار دیگر با x به عنوان ریشه شروع می‌کنیم. چون بازه ریشه یعنی $[16,21]$ با i همپوشانی ندارد، و چون $max[left[x]] = 23$ بزرگتر از $low[i] = 11$ است، به طرف چپ به گره‌ای که شامل $[8,9]$ است حرکت می‌کنیم (توجه کنید که هیچ بازه‌ای در زیر درخت راست با i همپوشانی ندارد - بعداً علت را خواهیم دید.)



شکل ۱۴.۵ بازه‌ها در اثبات قضیه ۱۴.۲. مقدار $max[left[x]]$ در هر حالت به صورت خط چین نشان داده شده است. (a) جستجو به راست می‌رود. هیچ بازه i' در زیر درخت چپ x نمی‌تواند با i همپوشانی داشته باشد. (b) جستجو به چپ می‌رود. زیر درخت چپ x شامل بازه‌ای است که با i همپوشانی دارد (وضعیت آن نشان داده نشده است) یا یک بازه i' در زیر درخت چپ x وجود دارد بطوریکه $high[i']=max[left[x]]$. چون i با i' همپوشانی ندارد، و چون $low[i'] \le low[i'']$، i با هیچ بازهٔ i'' در زیر درخت راست x همپوشانی ندارد.

بازه $[8,9]$ با i همپوشان نداشته و $max [left [x]] = 10$ کوچکتر از $low[i] = 11$ است، بنابراین به سمت راست می‌رویم. (توجه کنید که هیچ بازه‌ای در زیر درخت چپ با i همپوشان ندارد.) بازه $[15,23]$ با i همپوشانی ندارد و فرزند چپش $nil[T]$ است، بنابراین به راست می‌رویم، حلقه پایان یافته

و $nil[T]$ برگردانده می‌شود.

برای اینکه نشان دهیم چرا *INTERVAL-SEARCH* صحیح است، باید بفهمیم چرا بررسی تنها یک مسیر از ریشه کافی است. ایده اصلی این است که اگر بر روی هر گره x $int[x]$ با i همپوشانی نداشته باشد، جستجو همیشه در یک جهت پیش می‌رود: اگر یک بازه همپوشانی وجود داشته باشد، به طور قطعی پیدا خواهد شد. قضیه زیر این ویژگی را صحیح‌تر بیان می‌کند.

قضیه ۱۴.۲

هر اجرای $INTERVAL\text{-}SEARCH(T,i)$ با یک گره که بازه‌اش با i همپوشانی دارد را بر می‌گرداند یا $nil[T]$ را بر می‌گرداند که در این حالت T شامل گره‌ای که با i همپوشانی داشته باشد نیست.

اثبات حلقه *while* در خطوط ۵-۲ زمانی پایان می‌یابد که یا $x = nil[T]$ باشد و یا i با $int[x]$ همپوشانی داشته باشد. در حالت دوم، مطمئناً برگرداندن x صحیح است. بنابراین روی حالت اول که حلقه *while* بدلیل $x = nil[T]$ پایان می‌یابد، متمرکز می‌شویم.

از ثابت زیر برای حلقه *while* در خطوط ۵-۲ استفاده می‌کنیم:

اگر درخت T شامل بازه‌ای باشد که با i همپوشانی دارد آنگاه چنین بازه‌ای در زیر درخت مشتق شده از x وجود دارد.

از این ثابت به صورت زیر استفاده می‌کنیم.

مقدار دهی اولیه: قبل از اولین تکرار، خط اول x را ریشهٔ T قرار می‌دهد بنابراین ثابت برقرار می‌شود.

نگهداری: در هر تکرار حلقه یکی از خطوط ۴ یا ۵ اجرا می‌شوند. نشان خواهیم داد که ثابت حلقه در هر دو حالت حفظ می‌شود.

اگر خط ۵ اجرا شود، آنگاه بدلیل شرط انشعاب در خط ۳ داریم $left[x] = nil[T]$ یا $max\ [left[x]] < low[i]$. اگر $left[x] = nil[T]$ زیر درخت مشتق شده از $left[x]$ شامل بازه‌ای که با i همپوشانی داشته باشد نیست، و لذا قرار دادن x برابر $right[x]$ ثابت را حفظ می‌کند. بنابراین فرض کنید که $left[x] \neq nil[T]$ و $max\ [left[x]] < low\ [i]$ برقرار است. همان طور که شکل $14.5(a)$ نشان می‌دهد برای هر بازه i' در زیر درخت چپ x داریم

$$\begin{aligned} high[i'] &\le max[left[x]] \\ &< low[i]\,. \end{aligned}$$

بنابراین با توجه به سه قسمتی بازه‌ای، i و i' با یکدیگر همپوشانی ندارند. لذا زیر درخت چپ x شامل گره‌ای که با i همپوشانی داشته باشد نیست، از اینرو قرار دادن x برابر $right[x]$ ثابت را حفظ می‌کند.

از طرف دیگر اگر خط ۴ اجرا شود آنگاه نشان خواهیم داد که عکس نقیض ثابت حلقه حفظ می‌شود.

به عبارت دیگر اگر بازه‌ای همپوشان با i در زیر درخت مشتق شده از $left[x]$ وجود نداشته باشد آنگاه هیچ بازه همپوشانی با i در هیچ جای درخت وجود ندارد. از آنجایی که خط ۴ اجرا می‌شود، بدلیل شرط منشعب شده در خط ۳ داریم $max[left[x]] \leq low[i]$ بنا به تعریف فیلد max باید بازه i' در زیر درخت چپ x باشد بطوریکه

$$\begin{aligned} high[i'] &= max[left[x]] \\ &\geq low[i] . \end{aligned}$$

(شکل $14.5(b)$ این وضعیت را نشان می‌دهد.) چون i و i' با یکدیگر همپوشانی نمی‌کنند و چون $high[i'] < low[i']$ درست نیست، با استفاده از سه قسمتی بازه‌ای، $high[i] < low[i]$ نتیجه می‌شود. درخت‌های بازه‌ای روی حد پایین بازه‌ها کلید گذاری می‌شوند و بنابراین ویژگی درخت جستجو بیان می‌کند که برای هر بازه i'' در زیر درخت راست x داریم:

$$\begin{aligned} high[i] &< low[i'] \\ &\leq low[i''] . \end{aligned}$$

بنا به سه قسمتی بازه‌ای، i و i'' با یکدیگر همپوشانی ندارند پس نتیجه می‌گیریم که چه یک بازه در زیر درخت چپ x با i همپوشانی کند یا نکنده قرار دادن x برابر $left[x]$ ثابت را حفظ می‌کند.

خاتمه: اگر زمانی که $x = nil[T]$ است حلقه پایان یابد، هیچ بازه‌ای در زیر درخت مشتق شده از x وجود ندارد که با i همپوشانی کند. عکس نقیض ثابت حلقه بیان می‌کند که T شامل بازه‌ای که با i همپوشانی داشته باشد نیست. از اینرو برگرداندن $x = nil[T]$ صحیح است.

بنابراین روال *INTERVAL-SEARCH* به درستی کار می‌کند.

تمرین‌ها

۱۴.۳-۱ یک شبه کد برای *LEFT-ROTATE* بنویسید که روی گره‌های یک درخت بازه‌ای عمل کند و در زمان $O(1)$ فیلدهای max را به روز رسانی کند.

۱۴.۳-۲ کد *INTERVAL-SEARCH* را طوری بازنویسی کنید که وقتی همه بازه‌ها باز فرض می‌شوند درست کار کند.

۱۴.۳-۳ یک الگوریتم کارآمد بیان کنید که با دریافت بازهٔ i، یک بازه همپوشانی با i را برمی‌گرداند.

۱۴.۳-۴ یک درخت بازه‌ای T و یک بازهٔ i داده شده، توضیح دهید چطور همه بازه‌ها در T که با i همپوشانی دارند می‌توانند در زمان $O(min(n,k lg n))$ لیست شوند، اگر k تعداد بازه در لیست خروجی باشد. (اختیاری: جوابی را پیدا کنید که درخت را تغییر ندهد.)

۱۴.۳-۵ تغییراتی برای روال‌های درخت بازه‌ای پیشنهاد کنید که عمل جدید

$INTERVAL\text{-}SEARCH\text{-}EXACTLY(T,i)$ را پشتیبانی کند، که یک اشاره گر به گره x در درخت بازه‌ای T را بر می‌گرداند بطوریکه $low[int[x]] = low[i]$ و $high[int[x]] = high[i]$، یا اگر T چنین گره‌ای نداشت، $nil[T]$ را برگرداند. همهٔ اعمال شامل $INTERVAL\text{-}SEARCH\text{-}EXACTLY$ باید در زمان $O(lgn)$ روی یک درخت با n گره انجام شوند.

۱۴.۳-۶ نشان دهید چطور می‌توان یک مجموعه پویای Q از اعداد را طوری نگهداری کرد که عمل $MIN\text{-}GAP$ را پشتیبانی کنند، که این عمل میزان تفاوت بین نزدیکترین دو عدد در Q را می‌دهد. برای مثال، اگر $Q = \{1,5,9,15,18,22\}$، آنگاه $MIN\text{-}GAP(Q)$، $18 - 15 = 3$ را بر می‌گرداند چون 15 و 18 نزدیکترین دو عدد در Q می‌باشند. اعمال $INSERT$، $DELETE$، $SEARCH$ و $MIN\text{-}GAP$ را به مؤثرترین روش ممکن تبدیل کنید و زمان اجرای آنها را تحلیل کنید.

*۱۴.۳-۷ پایگاه داده‌های $VLSI$، عموماً یک مدار مجتمع را به صورت لیستی در مستطیل‌ها ارائه می‌دهند فرض کنید هر مستطیل به صورت خطهای راست جهت داده شده است (لبه‌ها، موازی محور x و y هستند.) بنابراین نمایش یک مستطیل شامل مینیمم و ماکزیمم مختصات x و y آن می‌باشد. یک الگوریتم با زمان $O(nlgn)$ ارائه دهید که معین کند آیا مجموعهٔ مستطیل‌های ارائه شده شامل دو مستطیل همپوشان هست یا خیر. الگوریتم شما لازم نیست همهٔ زوج‌های متقاطع را گزارش دهد ولی اگر دو مستطیل کاملاً همدیگر را بپوشانند حتی اگر خطوط مرزی متقاطع نباشند، باید گزارش دهد که یک همپوشانی وجود دارد. (راهنمایی: یک خط حرکت کننده[1] را در راستای مجموعهٔ مستطیل‌ها انتقال دهید.)

مسائل

۱۴-۱ نقطه ماکزیمم همپوشانی [2]

فرض کنید می‌خواهیم نقطه ماکزیمم همپوشانی را در یک مجموعه از بازه‌ها پیدا کنیم - نقطه‌ای که بزرگترین عدد بازه‌ها را در پایگاه داده همپوشانی با آن دارد.

a. نشان دهید همیشه یک نقطه ماکزیمم همپوشانی وجود دارد که حد بالای یکی از بخش‌ها می‌باشد.

b. ساختمان داده‌ای طراحی کنید که به طور مؤثری عملیات $INTERVAL\text{-}INSERT$، $INTERVAL\text{-}DELETE$ و $FIND\text{-}POM$ که یک اشاره‌گر به ماکزیمم همپوشانی را بر می‌گرداند، پشتیبانی کند. (راهنمایی: یک درخت قرمز - سیاه از تمام نقاط پایانی نگه دارید. مقدار $+1$ را برای حد پایین و مقدار -1 را برای حد بالا قرار دهید. هرگره درخت را برای نگهداری ماکزیمم همپوشانی با اطلاعات اضافی بهبود بخشید.)

---

1. sweep Line
2. point of maximum overlap

۲-۱۴ جایگشت Josephus

مسئله *Josephus* به شکل زیر تعریف می‌شود.فرض کنید که n فرد در یک دایره قرار گرفته‌اند و یک مقدار صحیح مثبت $n \geq m$ داده شده است. با شروع از اولین فرد تعیین شده، دور دایره پیش می‌رویم و هر بار mامین فرد را حذف می‌کنیم. بعد از اینکه هر فرد حذف شد، شمارش دور دایره برای افراد باقیمانده ادامه می‌یابد. تا زمانی که همهٔ n فرد حذف نشده‌اند، این فرآیند ادامه می‌یابد. ترتیبی که افراد از دایره حذف می‌شوند به عنوان **جایگشت** *(n,m)-Josephus* برای اعداد صحیح n ,... ,2 , 1 تعریف می‌شود. برای مثال جایگشت *(7,3)-Josephus*، <3, 6, 2, 7, 5, 1, 4> است.

a.فرض کنید m یک مقدار ثابت است. الگوریتمی را توضیح دهید که در زمان $O(n\lg n)$، اعداد صحیح n و m را گرفته و جایگشت *(n,m)-Josephus* را به عنوان خروجی بدهد.

b.فرض کنید m مقدار ثابت نیست. الگوریتمی با زمان $O(n\lg n)$ توضیح دهید که مقادیر صحیح n و m را گرفته و به عنوان خروجی جایگشت *(n,m)-Josephus* را بدهد.

IV

طراحی و تکنیک‌های تحلیل پیشرفته

مقدمه

این قسمت، سه تکنیک مهم برای طراحی و تحلیل الگوریتم‌های کارآمد را در بر دارد: برنامه سازی پویا (فصل ۱۵)، الگوریتم‌های حریصانه (فصل ۱۶) و تحلیل سرشکن شده (فصل ۱۷). قسمت‌های قبلی، تکنیک‌هایی که از لحاظ کاربردی گسترده بودند ارائه دادند، مانند تقسیم و حل، تصادفی سازی و راه حل‌های بازگشتی. تکنیک‌های جدید تا حدی پیچیده‌تر هستند، ولی برای حل بسیاری از مسائل محاسباتی مفید هستند. موضوع‌هایی که در این قسمت معرفی شدند بعداً تکرار خواهند شد.

برنامه سازی پویا به طور معمول برای بهینه سازی مسائلی مورد استفاده قرار می‌گیرند که در آن‌ها یک مجموعه از انتخاب‌ها برای رسیدن به نتیجه بهینه انتخاب می‌شوند. زمانی که انتخاب‌ها انجام شد، زیر مسئله‌هایی به همان شکل بوجود می‌آید. برنامه سازی پویا، وقتی یک زیر مسئله داده شده از بیش از یک مجموعهٔ جزئی از انتخاب‌ها بوجود می‌آید مؤثر است. تکنیک اصلی این است که جواب را برای هر یک از چنین زیر مسئله‌هایی که دوباره پدیدار می‌شوند، ذخیره کنیم. فصل ۱۵ نشان می‌دهد که چطور این ایده ساده می‌تواند گاهی الگوریتم‌های با زمان نمایی را به الگوریتم‌های با زمان چند جمله‌ای تبدیل کند.

همانند الگوریتم‌های برنامه سازی پویا، الگوریتم‌های حریصانه هم به طور معمول برای مسائل بهینه‌سازی مورد استفاده قرار می‌گیرند که در آن برای رسیدن به نتیجه بهینه، یک مجموعه از انتخاب‌ها ایجاد می‌شود. ایده یک الگوریتم حریصانه این است که هر انتخاب در حالت بهینه محلی انجام می‌شود. یک مثال ساده، خرد کردن پول است: برای مینیمم کردن تعداد سکه‌های مورد نیاز جهت خرد کردن یک مقدار داده شده، کافیست به طور متوالی بزرگترین واحد سکه را که از مقدار باقیمانده بزرگتر نیست انتخاب کنیم. بسیاری از چنین مسائلی وجود دارند که روش حریصانه، نتیجهٔ بهینه را سریعتر از روش برنامه سازی پویا بدست می‌آورد. بنابراین همیشه گفتن اینکه آیا روش حریصانه مؤثرتر است آسان نیست. فصل ۱۶ تئوری *motroid* را بررسی می‌کند که اغلب می‌تواند برای ایجاد چنین تشخیص مفید باشد.

تحلیل سرشکن شده وسیله‌ای برای تحلیل الگوریتم‌هایی که یک توالی از عملیات مشابه را انجام

می‌دهند می‌باشد. بجای محدود کردن هزینه یک توالی از عملیات توسط محدود کردن هزینه واقعی هر عمل به طور جداگانه، یک تحلیل سرشکن شده می‌تواند برای ایجاد محدودیت روی هزینه واقعی همه توالی مورد استفاده قرار گیرد. یکی از دلایلی که این ایده می‌تواند مؤثر باشد این است که در یک توالی از اعمال، اجرای همه اعمال خاص در زمان شناخته شده محدود، در بدترین حالت ممکن است غیر ممکن باشد. در جایی که بعضی از اعمال هزینه زیادی صرف می‌کنند (گران هستند)، بسیاری دیگر ممکن است ارزان باشند. بنابراین تحلیل سرشکن شده فقط یک وسیله تحلیل نیست: یک روش برای تفکر در مورد طراحی الگوریتم‌ها نیز می‌باشد، چون طراحی و تحلیل زمان اجرا الگوریتم‌ها به طور نزدیکی به هم مرتبطند. فصل ۱۷ سه روش برای انجام یک تحلیل سرشکن شده روی یک الگوریتم را معرفی می‌کند.

۱۵ برنامه‌سازی پویا

برنامه سازی پویا مانند روش تقسیم و حل، مسئله را با ترکیب نتایج زیر مسئله‌ها حل می‌کند. (برنامه سازی در این زمینه به یک روش جدولی اشاره می‌کند نه به نوشتن برنامهٔ کامپیوتری.) همان طور که در فصل ۲ دیدیم، الگوریتم‌های تقسیم و حل مسئله را به زیر مسئله‌های مستقل تقسیم کرده، به طور بازگشتی زیر مسئله‌ها را حل کرده و سپس نتایج آنها را برای حل مسئله اصلی با هم ترکیب می‌کند. در مقابل، زمانی که زیر مسئله‌ها مستقل نیستند، برنامه سازی پویا کاربرد بیشتری دارد، به عبارت دیگر زمانی که زیر مسئله‌ها در زیر مسئله‌های کوچکتر با هم مشترک باشند. در این مفهوم، الگوریتم تقسیم و حل بیشتر از نیاز کار می‌کند. یعنی زیر مسئله‌های کوچکتر رایج را چندین بار حل می‌کند. یک الگوریتم به روش برنامه سازی پویا هر زیر مسئله را تنها یکبار حل می‌کند و سپس جوابها را در یک جدول نگهداری می‌کند و به این وسیله هر بار که با یک زیر مسئله مواجه میشود از محاسبهٔ دوباره جواب جلوگیری می‌کند.

برنامه سازی پویا نوعاً برای *مسائل بهینه سازی* [1] بکار می‌رود. در این نوع مسئله‌ها تعداد زیادی نتیجه ممکن می‌تواند وجود داشته باشد. هر جواب مقداری دارد و می‌خواهیم جوابی را پیدا کنیم که دارای مقدار بهینه (مینیمم یا ماکزیمم) است. به چنین جوابی **یک** جواب بهینه مسئله می‌گوئیم که با جواب بهینه فرق دارد، چون ممکن است چندین جواب به مقدار بهینه دست یابند.

توسعه یک الگوریتم به روش برنامه سازی پویا می‌تواند به ۴ مرحله تقسیم شود.

۱. مشخص کردن ساختار جواب بهینه

۲. تعریف بازگشتی مقدار یک جواب بهینه

۳. محاسبهٔ مقدار یک جواب بهینه به روش از پایین به بالا.

۴. ساختن یک جواب بهینه از روی اطلاعات محاسبه شده.

مراحل ۱ تا ۳ پایهٔ یک جواب برنامه سازی پویای مسئله را شکل می‌دهند. اگر تنها مقدار یک جواب بهینه مورد نیاز باشد، مرحله ۴ می‌تواند حذف شود. هنگامی که مرحله ۴ را انجام می‌دهیم، گاهی در طی

1. *optimization problems*

محاسبات در مرحله ۳، برای آسان کردن ساخت یک جواب بهینه اطلاعات اضافی را نگهداری می‌کنیم.

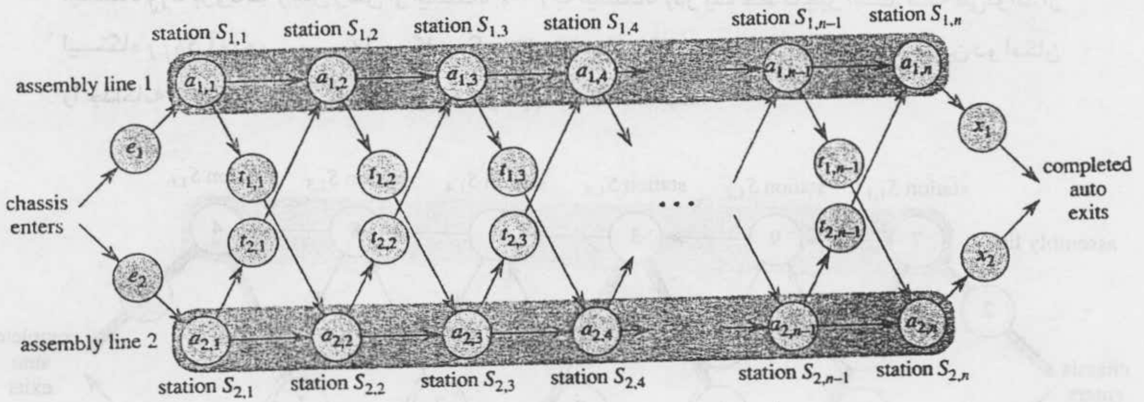
بخش‌های زیر از روش برنامه سازی پویا برای حل تعدادی مسئله بهینه سازی استفاده می‌کنند. بخش ۱۵.۱ مسئله‌ای در ارتباط با دو خط مونتاژ اتومبیل را بررسی می‌کند که بعد از هر ایستگاه، اتومبیل در حال ساخت، می‌تواند در همان خط باقیمانده یا به خط دیگر برود. بخش ۱۵.۲ بیان می‌کند که چطور می‌توانیم یک زنجیره از ماتریسها را طوری ضرب کنیم که کمترین تعداد ضرب عددی انجام شود. با ارائه این مثالها از برنامه سازی پویا، بخش ۱۵.۳ در مورد دو ویژگی کلیدی که یک مسئله باید داشته باشد تا برنامه سازی پویا یک تکنیک مناسب برای جواب آن باشد، بحث می‌کند سپس بخش ۱۵.۴ نشان می‌دهد که چطور می‌توان بزرگترین زیر رشتهٔ مشترک دو رشته را پیدا کرد. در پایان، بخش ۱۵.۵ برای ساختن درخت‌های جستجوی دودویی بهینه، با یک توزیع معلوم از کلیدها برای جستجو، از برنامه سازی پویا استفاده می‌کند.

۱۵.۱ زمان بندی خطوط مونتاژ

اولین مثال برنامه سازی پویا، یک مسئله تولیدی را حل می‌کند. شرکت *Colonel Motors* اتومبیل‌ها را در کارخانه‌ای تولید می‌کند که دارای دو خط مونتاژ می‌باشد که در شکل ۱۵.۱ نشان داده شده است. بدنه اتومبیلی که وارد هر خط مونتاژ می‌شود، دارای قسمت‌هایی است که در تعدادی ایستگاه به آن اضافه می‌شود و یک ماشین تکمیل شده در پایان خط خارج می‌شود. هر خط مونتاژ، n ایستگاه دارد که به صورت $j = 1, 2, ..., n$ شماره گذاری می‌شوند. jامین ایستگاه در خط i را (که i، 1 یا 2است.) با $S_{i,j}$ مشخص می‌کنیم. jامین ایستگاه در خط 1 $(S_{1,j})$ عملی مشابه عمل jامین ایستگاه در خط 2 $(S_{2,j})$ انجام می‌دهد. ایستگاهها در زمانها و با تکنولوژیهای متفاوتی ساخته شده‌اند، بنابراین زمان لازم برای هر ایستگاه متفاوت است، حتی بین ایستگاههایی که در یک موقعیت مشابه روی دو خط مختلف قرار دارند. زمان مونتاژ لازم برای ایستگاه $S_{i,j}$ را با $a_{i,j}$ نمایش می‌دهیم. همان طور که شکل ۱۵.۱ نشان می‌دهد یک بدنه به ایستگاه اول یکی از خطوط مونتاژ وارد شده و از هر ایستگاه به ایستگاه بعدی پیشرفت می‌کند. همچنین زمان ورودی e_i برای اینکه بدنه به خط i وارد شود و زمان خروجی x_i برای اینکه ماشین کامل شده از خط i خارج شود وجود دارد.

به طور طبیعی، وقتی بدنه به یک خط مونتاژ وارد می‌شود تنها از همان خط عبور می‌کند. زمان رفتن از یک ایستگاه به ایستگاه بعدی در یک خط مونتاژ ناچیز است. گاهی اوقات، یک سفارش مخصوص فوری داده می‌شود و مشتری می‌خواهد که اتومبیل در سریعترین زمان ممکن ساخته شود. بر روی سفارشهای فوری، بدنه همچنان از n ایستگاه به ترتیب عبور می‌کند ولی مدیر کارخانه ممکن است اتومبیل ناقص را از یک خط مونتاژ به خط دیگر بعد از هر ایستگاه منتقل کند. زمان منتقل کردن بدنه از یک خط مونتاژ i بعد از گذر از ایستگاه $S_{i,j}$، برابر $t_{i,j}$ می‌باشد که $i = 1, 2$ و $j = 1, 2, ..., n - 1$

است (چون بعد از n امین ایستگاه، مونتاژ تکمیل می‌شود.) مسئله این است که مشخص کنیم چه ایستگاههایی در خط 1 و چه ایستگاههایی در خط 2 باید انتخاب شوند تا زمان کل برای مونتاژ یک ماشین در این کارخانه مینیمم شود. در مثال شکل (a)۱۵.۲ سریعترین زمان کل با انتخاب ایستگاههای 1 و 3 و 6 از خط 1 و انتخاب ایستگاههای 2، 4 و 5 از خط 2 بدست می‌آید. وقتی تعداد ایستگاهها زیاد باشد روش واضح و "قدرتمندانه"[1] برای مینیمم کردن زمان در کارخانه غیر ممکن است.



شکل ۱۵.۱ یک مسئله تولیدی برای پیدا کردن سریعترین مسیر در یک کارخانه. در خط مونتاژ وجود دارد که هر یک n ایستگاه دارد. j امین ایستگاه در خط i با $S_{i,j}$ مشخص می‌شود و زمان مونتاژ در آن ایستگاه، $a_{i,j}$ است. بدنه اتومبیل، وارد کارخانه شده و با صرف زمان e_i وارد خط i (که $i = 1$ یا 2 است) می‌شود. بعد از گذر از jامین ایستگاه در یک خط، به بدنه $(j + 1)$ امین ایستگاه در خط دیگر می‌رود. اگر بدنه درهمان خط بماند، هیچ هزینه انتقالی وجود ندارد ولی انتقال به خط دیگر بعد از ایستگاه $S_{i,j}$ به اندازه $t_{i,j}$ به [illegible] می‌انجامد. بعد از خروج از ایستگاه nام در یک خط، زمان x_i صرف خروج [illegible] از کارخانه می‌شود. مسئله مشخص کردن ایستگاههایی از خط 1 و ایستگاههایی که از 2 بایستی انتخاب شوند است، بطوریکه زمان کل برای مونتاژ یک ماشین در کارخانه مینیمم شود.

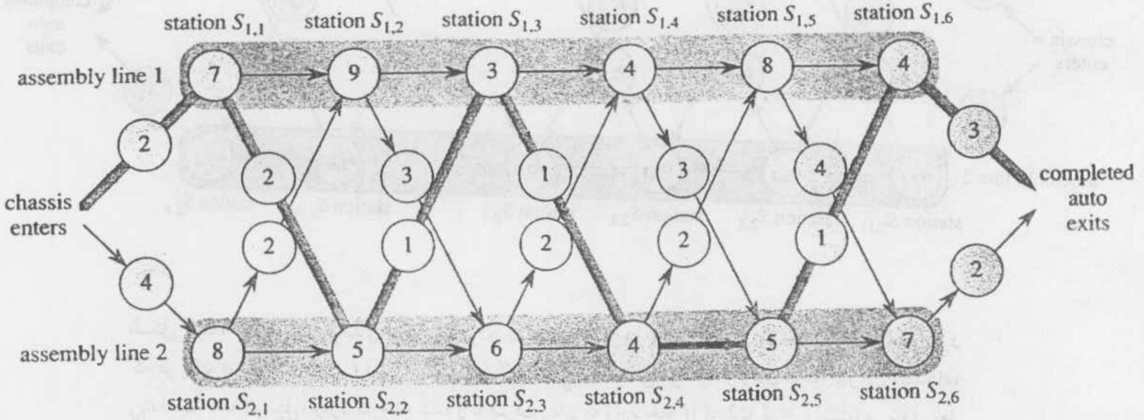
اگر لیستی از اینکه کدام ایستگاهها در خط 1 و کدام ایستگاهها در خط 2 استفاده می‌شوند، به ما داده شود، محاسبه زمان گذر یک بدنه از کارخانه در زمان $\Theta(n)$ آسان است. متأسفانه 2^n روش ممکن برای انتخاب ایستگاهها وجود دارد که آنها را با در نظر گرفتن مجموعه ایستگاههای استفاده شده در خط 1 به عنوان زیر مجموعه‌ای از $\{1,2,...,n\}$ و مشخص کردن آنکه 2^n تا از این زیر مجموعه‌ها وجود دارد، تعیین می‌کنیم. بنابراین مشخص کردن سریعترین زمان در کارخانه با شمردن همهٔ راههای ممکن و محاسبه زمان آنها، نیاز به زمان $\Omega(2^n)$ دارد که وقتی n بزرگ باشد غیر ممکن است.

---

1. brute force

مرحله ۱: ساختار سریعترین مسیر در کارخانه

اولین مرحله الگوی برنامه سازی پویا، مشخص کردن ساختار جواب بهینه است. برای مسئله زمان‌بندی خطوط مونتاژ، می‌توانیم این مرحله را به صورت زیر انجام دهیم. فرض کنید سریعترین محاسبه ممکن برای یک بدنه از ایستگاه $S_{i,j}$ شروع می‌شود. اگر $j = 1$ باشد، تنها یک مسیر وجود دارد که بدنه می‌تواند به آن رفته باشد و بنابراین مشخص کردن زمان گذر از ایستگاه $S_{1,j}$ آسان است. اما برای $j = 2, 3, ..., n$ دو انتخاب وجود دارد: بدنه می‌تواند از ایستگاه $S_{1,j-1}$ آمده باشد و سپس به ایستگاه $S_{1,j}$ برود که زمان رفتن از ایستگاه $j - 1$ به ایستگاه j در یک خط ناچیز است. بدنه می‌تواند از ایستگاه $S_{2,j-1}$ آمده و سپس به ایستگاه $S_{1,j}$ منتقل شده باشد که زمان انتقال $t_{2,j-1}$ است. ما این دو امکان را جداگانه در نظر خواهیم گرفت. اگر چه خواهیم دید که مشابهند.



j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

شکل ۱۵.۲ (a) نمونه‌ای از مسئله خطوط مونتاژ با هزینه‌های مشخص شده e_i، $a_{i,j}$، $t_{i,j}$ و x_i. مسیر پر رنگ سریعترین مسیر در کارخانه را نشان می‌دهد. (b) مقادیر $f_i[j]$، f^*، $l_i[j]$ و l^* برای قسمت (a).

ابتدا اجازه دهید فرض کنیم سریعترین مسیر تا ایستگاه $S_{i,j}$ که سریعترین مسیر تا ایستگاه $S_{1,j-1}$ می‌باشد. ایده اصلی آن است که بدنه باید سریعترین راه را از نقطه شروع تا ایستگاه $S_{1,j-1}$ پیموده باشد. چرا؟ اگر مسیر سریعتری برای گذر از $S_{1,j-1}$ وجود داشته باشد، می‌توانیم این مسیر سریعتر را جایگزین کنیم تا یک مسیر سریعتر تا ایستگاه $S_{1,j}$ حاصل آید که این یک تناقض می‌باشد.

به طور مشابه، اکنون فرض می‌کنیم که سریعترین مسیر تا ایستگاه $S_{1,j}$که سریعترین مسیر تا ایستگاه $S_{2,j-1}$است. اکنون می‌بینیم که بدنه باید سریعترین مسیر را از نقطه شروع تا ایستگاه $S_{2,j-1}$طی کرده باشد استدلال به همان شکل است: اگر یک مسیر سریعتر برای گذر از $S_{2,j-1}$ وجود داشته باشد می‌توانیم این مسیر سریعتر را جایگزین کنیم تا یک مسیر سریعتر تا ایستگاه $S_{1,j}$ حاصل آید که یک تناقض می‌باشد.

به طور کلی‌تر می‌توانیم بگوئیم برای زمان بندی خط مونتاژ، یک جواب بهینه برای مسئله (پیدا کردن سریعترین مسیر تا ایستگاه $S_{i,j}$) شامل یک جواب بهینه برای زیر مسئله‌ها است (پیدا کردن سریعترین مسیر تا $S_{1,j-1}$یا $S_{2,j-1}$) به این ویژگی به عنوان ***ساختار بهینه***[1] اشاره می‌کنیم، و این یکی از نشانه‌های قابلیت برنامه سازی پویا است، همان طور که در بخش ۱۵.۳ خواهیم دید.

از زیر ساختار بهینه استفاده می‌کنیم تا نشان دهیم، می‌توانیم از جوابهای بهینه زیر مسئله‌ها، حل بهینهٔ یک مسئله را بسازیم. برای زمان بندی خط مونتاژ به صورت زیر استدلال می‌کنیم. اگر به سریعترین مسیر تا ایستگاه $S_{1,j}$نگاه کنیم، باید از ایستگاه $j-1$ در یکی از خطوط 1 یا 2 گذر کرده باشد بنابراین سریعترین مسیر تا ایستگاه $S_{1,j}$یکی از موارد زیر است.

- سریعترین مسیر تا ایستگاه $S_{1,j-1}$و سپس مستقیماً به ایستگاه $S_{1,j}$ یا
- سریعترین مسیر تا ایستگاه $S_{2,j-1}$که یک انتقال از خط 2 به خط 1، سپس به ایستگاه $S_{1,j}$ با استفاده از استدلال متقارن سریعترین مسیر تا ایستگاه $S_{2,j}$یکی از موارد زیر است:
- سریعترین مسیر تا ایستگاه $S_{2,j-1}$و سپس مستقیماً به ایستگاه $S_{2,j}$ یا
- سریعترین مسیر تا ایستگاه $S_{1,j-1}$که یک انتقال از خط 1 به خط 2 و سپس به ایستگاه $S_{2,j}$ برای حل مسئله پیدا کردن سریعترین مسیر تا ایستگاه jاز یکی از خطها، زیر مسئله‌های پیدا کردن سریعترین مسیر تا ایستگاه $j-1$در هر دو خط را حل می‌کنیم.

بنابراین، می‌توانیم با تولید نتایج بهینه برای زیر مسئله‌ها، یک حل بهینه برای نمونه مسئله زمان‌بندی خط مونتاژ بسازیم.

مرحله ۲: حل بازگشتی

مرحله دوم الگوی برنامه سازی پویا، تعریف بازگشتی مقدار یک جواب بهینه با توجه به جوابهای بهینه زیر مسئله‌ها است. برای مسئله زمان بندی خط مونتاژ، به عنوان زیر مسئله‌ها، مسائلی را برای پیدا کردن سریعترین مسیر تا ایستگاه jدر هر دو خط که $j=1,2,\ldots , n$است انتخاب می‌کنیم، $f_i[j]$را سریعترین زمان ممکن برای عبور بدنه از نقطه شروع تا ایستگاه $S_{i,j}$ قرار می‌دهیم.

هدف نهایی، مشخص کردن سریعترین زمان برای عبور بدنه از همه مسیر تا خروج از کارخانه

---

1. optimal structure

است که با f^* نمایش می‌دهیم. بدنه باید همه مسیر تا n ایستگاه را روی یکی از خطوط 1 یا 2 طی کرده و سپس از کارخانه خارج شود. چون سریعترین این مسیرها، سریعترین مسیر در کل کارخانه است داریم:

$$f^* = min\ (f_1\ [n] + x_1\ ,\ f_2[n] + x_2) \qquad (۱۵.۱)$$

استدلال در مورد $f_1[1]$ و $f_2[1]$ نیز آسان است. برای گذشتن از ایستگاه 1 در یکی از خطوط، بدنه فقط مستقیماً به آن ایستگاه می‌رود. بنابراین

$$f_1[1] = e_1 + a_{1,1} \qquad (۱۵.۲)$$

$$f_2[1] = e_2 + a_{2,1} \qquad (۱۵.۳)$$

حال اجازه دهید چگونگی محاسبه $f_i[j]$ برای $(i = 1,2)\ ,\ j = 2,3,...,n$ را در نظر بگیریم. با تمرکز روی $f_i[j]$ دوباره خاطر نشان می‌کنیم که سریعترین مسیر تا ایستگاه $S_{i,j}$ یا سریعترین مسیر تا ایستگاه $S_{1,j-1}$ و سپس مستقیماً به ایستگاه $S_{1,j}$ است، یا سریعترین مسیر تا ایستگاه $S_{2,j-1}$ که یک انتقال از خط 2 به خط 1 و سپس به ایستگاه $S_{1,j}$ است. در حالت اول داریم $f_1[j] = f_1[j-1] + a_{1,j}$ و در حالت بعدی داریم $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$ بنابراین برای $j = 2,3,...,n$ داریم

$$f_1[j] = min\ (f_1\ [j-1] + a_{1,j}\ ,\ f_2[j-1] + t_{2,j-1} + a_{1,j}) \qquad (۱۵.۴)$$

متقابلاً برای $j = 2, 3, ..., n$ داریم.

$$f_2[j] = min\ (f_2\ [j-1] + a_{1,j}\ ,\ f_1[j-1] + t_{1,j-1} + a_{2,j}) \qquad (۱۵.۵)$$

با ترکیب معادلات (۱۵.۵) - (۱۵.۲) به معادلات بازگشتی زیر می‌رسیم.

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1\ , \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases} \qquad (۱۵.۶)$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1\ , \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases} \qquad (۱۵.۷)$$

شکل (b)۱۵.۲ مقادیر $f_i[j]$ را برای مثال قسمت (a) نشان می‌دهد که توسط معادلات (۱۵.۶) و (۱۵.۷) به همراه مقدار f^* محاسبه شده است.

مقادیر $f_i[j]$ مقادیر جوابهای بهینه برای زیر مسئله‌ها را ارائه می‌کنند. برای کمک به دنبال کردن چگونگی ساخت یک جواب بهینه، $l_i[j]$ را تعریف می‌کنیم که شماره خطی است که ایستگاه $j-1$ آن در سریعترین مسیر تا ایستگاه $S_{i,j}$ استفاده شده است که ممکن است 1 یا 2 باشد. در اینجا $i = 1, 2$ و $j = 2, 3, ...,n$ است (از تعریف $l_i[j]$ صرف نظر می‌کنیم چون هیچ ایستگاهی در هر یک از خطها قبل از ایستگاه اول قرار ندارد.) l^* را نیز به عنوان خطی که ایستگاه n آن در سریعترین مسیر در کل کارخانه

استفاده شده است، تعریف می‌کنیم. مقادیر $l_i[j]$ به ما در دنبال نمودن سریعترین راه کمک می‌کنند. با استفاده از مقادیر l^* و $l_i[j]$ که در شکل ۱۵.۲(b) نشان داده شده‌اند می‌توانیم سریعترین مسیر در کارخانه که در قسمت (a) نشان داده شده است را به صورت زیر دنبال کنیم. با $l^* = 1$ شروع کرده و لذا ایستگاه $S_{1,6}$ استفاده می‌کنیم. اکنون به $l_1[6]$ که برابر 2 است می‌نگریم و لذا از ایستگاه $S_{2,5}$ استفاده می‌کنیم. در ادامه به $l_2[5] = 2$ (با استفاده از ایستگاه $S_{2,4}$)، $l_2[4] = 1$ (ایستگاه $S_{1,3}$)، $l_1[3] = 2$ (ایستگاه $S_{2,2}$) و $l_2[2] = 1$ (ایستگاه $S_{1,1}$) می‌نگریم.

مرحله ۳: محاسبه سریعترین زمان‌ها

در این مرحله، محاسبه سریعترین راه در کارخانه با نوشتن یک الگوریتم بازگشتی بر پایه معادله ۱۵.۱ و رابطه‌های بازگشتی (۱۵.۶) و (۱۵.۷) کار آسانی است. یک مسئله با چنین الگوریتم بازگشتی وجود دارد: زمان اجرای آن بصورت نمایی در n است. برای فهمیدن دلیل این مطلب، $r_i(j)$ را تعداد ارجاعها به $f_i[j]$ در الگوریتم بازگشتی قرار دهید. از معادله (۱۵.۱) داریم:

$$r_1(n) = r_2(n) = 1 \qquad (۱۵.۸)$$

با توجه به بازگشت‌های (۱۵.۶) و (۱۵.۷) برای $j = 1,2,...,n - 1$ داریم

$$r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1) \qquad (۱۵.۹)$$

به همان شکل تمرین ۲-۱۵.۱ از شما می‌خواهد تا نشان دهید $r_i(j) = 2^{n-j}$ بنابراین $f_1[1]$ به تنهایی، 2^{n-1} بار مورد مراجعه قرار می‌گیرد! به همان شکل تمرین ۳-۱۵.۱ از شما می‌خواهد تا نشان دهید تعداد کل ارجاعها به تمام مقادیر $f_i[j]$ برابر $\Theta(2^n)$ است.

اگر مقادیر $f_i[j]$ را در یک ترتیب متفاوت با روش بازگشتی محاسبه کنیم می‌توانیم بهتر عمل کنیم. ملاحظه کنید که برای $j \geq 2$ هر مقدار $f_i[j]$ تنها بستگی به مقادیر $f_1[j-1]$ و $f_2[j-1]$ دارد. با محاسبه مقادیر $f_i[j]$ در ترتیب صعودی از شماره ایستگاهها، یعنی j- چپ به راست در شکل ۱۵.۲ (b) - می‌توانیم سریعترین مسیر در کارخانه و زمانی که صرف می‌کند را در زمان $\Theta(n)$ محاسبه کنیم. روال $FASTEST\text{-}WAY$ به عنوان ورودی مقادیر $a_{i,j}$، $t_{i,j}$، e_i و x_i و تعداد ایستگاهها در هر خط مونتاژ، یعنی n را می‌گیرد.

```
FASTEST-WAY(a, t, e, x, n)
1  f1[1] ← e1 + a1,1
2  f2[1] ← e2 + a2,1
3  for j ← 2 to n
4      do if f1[j − 1] + a1,j ≤ f2[j − 1] + t2,j−1 + a1,j
5            then f1[j] ← f1[j − 1] + a1,j
6                 l1[j] ← 1
```

```
7            else  f1[j] ← f2[j − 1] + t2,j−1 + a1,j
8                  l1[j] ← 2
9         if f2[j − 1] + a2,j ≤ f1[j − 1] + t1,j−1 + a2,j
10           then f2[j] ← f2[j − 1] + a2,j
11                l2[j] ← 2
12           else  f2[j] ← f1[j − 1] + t1,j−1 + a2,j
13                 l2[j] ← 1
14  if f1[n] + x1 ≤ f2[n] + x2
15     then f* = f1[n] + x1
16          l* = 1
17     else f* = f2[n] + x2
18          l* = 2
```

FASTEST-WAY به این صورت عمل می‌کند. خطوط ۲-۱، $f_1[1]$ و $f_2[1]$ را با استفاده از معادلات (۱۵.۲) و (۱۵.۵) محاسبه می‌کنند. سپس حلقهٔ *for* در خطوط ۱۳-۳، $f_i[j]$ و $l_i[j]$ را برای $i = 1,2$ و $j = 2,3,...,n$ محاسبه می‌کنند. خطوط ۸-۴، $f_1[j]$ و $l_1[j]$ را با استفاده از معادلهٔ ۱۵.۴ و خطوط ۱۳-۹، $f_2[j]$ و $l_2[j]$ را با استفاده از معادله (۱۵.۵) محاسبه می‌کنند. در پایان خطوط ۱۸-۱۴، f^* و l^* را با استفاده از معادله ۱۵.۱ محاسبه می‌نمایند. چون خطوط ۲-۱ و ۱۸-۱۴ و هر یک از $n - 1$ تکرار حلقهٔ *for* در خطوط ۱۳-۳، زمان ثابتی صرف می‌کنند زمان کل روال $\Theta(n)$ می‌باشد. یک روش برای مشاهده روند محاسبه مقادیر $f_i[j]$ و $l_i[j]$ این است که یک جدول از ورودیها را پر کنیم. با مراجعه به شکل (b) ۱۵.۲، یک جدول شامل مقادیر $f_i[j]$ و $l_i[j]$ را از چپ به راست (و بالا به پایین در هر ستون) پر می‌کنیم. برای پر کردن $f_i[j]$ ورودی، به مقادیر $f_1[j-1]$ و $f_2[j-1]$ نیاز داریم، و با دانستن اینکه آنها را قبلاً محاسبه و ذخیره کرده‌ایم با مشاهدهٔ مقادیر آنها در جدول، به راحتی آنها را تعیین می‌کنیم.

مرحله ۴: ایجاد سریعترین مسیر در کارخانه

پس از محاسبهٔ مقادیر $f_i[j]$ f^* $l_i[j]$ و l^* لازم است یک توالی شامل ایستگاههایی که در سریعترین مسیر در کارخانه استفاده می‌شوند را تولید کنیم. قبلاً در مثال شکل ۱۵.۲، در مورد چگونگی این کار بحث کردیم. روال زیر، ایستگاههای به کار رفته در ترتیب نزولی از شماره ایستگاهها را چاپ می‌کند. تمرین ۱- ۱۵.۱ از شما می‌خواهد تا این تابع را طوری تغییر دهید که آنها را در یک ترتیب صعودی از شماره ایستگاهها چاپ کند.

```
PRINT-STATIONS(l, n)
1  i ← l*
2  print "line " i ", station " n
3  for j ← n downto 2
4       do i ← l_i[j]
5           print "line " i ", station " j − 1
```

در مثال شکل ۱۵.۲، روال *PRINT-STATIONS* خروجی زیر را تولید می‌کند.

```
line 1, station 6
line 2, station 5
line 2, station 4
line 1, station 3
line 2, station 2
line 1, station 1
```

تمرین‌ها

۱۵.۱-۱ نشان دهید چطور می‌توان تابع *PRINT-STATIONS* را طوری تغییر داد که ایستگاهها را در یک ترتیب صعودی از شمارهٔ ایستگاهها چاپ کند (راهنمایی: از بازگشت استفاده کنید.)

۱۵.۱-۲ برای نشان دادن اینکه $r_i(j)$ یعنی تعداد ارجاعها به $f_i[j]$ در یک الگوریتم بازگشتی، برابر 2^{n-j} است از معادلات (۱۵.۸) و (۱۵.۹) و روش جایگذاری استفاده کنید.

۱۵.۱-۳ با استفاده از نتیجه تمرین ۱۵.۱-۲ نشان دهید که تعداد کل ارجاعها به همهٔ مقادیر $f_i[j]$ یا به عبارتی $\sum_{i=1}^{2}\sum_{j=1}^{n} r_i(j)$ دقیقاً برابر $2^{n+1} - 2$ است.

۱۵.۱-۴ جداول شامل مقادیر $f_i[j]$ و $l_i[j]$ با هم در مجموع شامل $4n - 2$ ورودی اند. نشان دهید چطور می‌توان هنگامی که همچنان در حال محاسبه f^* و قادر به چاپ همهٔ ایستگاهها در سریعترین مسیر در کارخانه هستیم، فضای لازم برای کل $2n + 2$ ورودی را کاهش داد.

۱۵.۱-۵ پرفسور *Canty* گام می‌کند که مقادیری برای $a_{i,j}$ $t_{i,j}$ و e_i وجود دارد که روال *FASTEST-WAY* برای آنها مقادیر $l_i[j]$ را تولید می‌کند بطوریکه برای ایستگاه شمارهٔ j داریم $l_1[j] = 2$ و $l_2[j] = 1$ با فرض نامخفی بودن همهٔ هزینه‌های انتقال $t_{i,j}$ نشان دهید که پرفسور اشتباه می‌کند.

۱۵.۲ ضرب زنجیره‌ای ماتریس‌ها[1]

مثال بعدی برنامه سازی پویا، الگوریتمی است که مسئله ضرب زنجیره‌ای ماتریسها را حل می‌کند. یک توالی (زنجیره) $<A_1, A_2, \ldots, A_n>$ از n ماتریس برای ضرب به ما داده شده و می‌خواهیم ضرب زیر را محاسبه نماییم.

$$A_1, A_2, \ldots, A_n \qquad (۱۵.۱۰)$$

می‌توانیم عبارت (۱۵.۱۰) را با استفاده از الگوریتم استاندارد ضرب دو ماتریس، به عنوان یک زیر روال محاسبه کنیم بطوریکه برای رفع ابهامات در چگونگی ضرب ماتریس‌ها با یکدیگر آن را پرانتزگذاری می‌کنیم. ضرب ماتریس‌ها *کاملاً پرانتزگذاری شده*[2] است، اگر، یا فقط یک ماتریس باشد یا حاصل ضرب دو ماتریس کاملاً پرانتزگذاری شده باشد که توسط پرانتزها محدود شده‌اند. ضرب ماتریس شرکت‌پذیر است و بنابراین همه پرانتزگذاریها نتیجهٔ یکسانی حاصل می‌کنند. برای مثال اگر زنجیرهٔ ماتریس‌ها، $<A_1, A_2, A_3, A_4>$ باشد، ضرب A_1, A_2, A_3, A_4 می‌تواند به ۵ روش متفاوت کاملاً پرانتزگذاری شود.

$$(A_1(A_2(A_3A_4))),$$
$$(A_1((A_2A_3)A_4)),$$
$$((A_1A_2)(A_3A_4)),$$
$$((A_1(A_2A_3))A_4),$$
$$(((A_1A_2)A_3)A_4).$$

روشی که یک زنجیره از ماتریس‌ها را پرانتزگذاری می‌کنیم، می‌تواند اثر چشمگیری روی هزینه محاسبه حاصلضرب داشته باشد. ابتدا، هزینه ضرب دو ماتریس را در نظر بگیرید. الگوریتم استاندارد توسط شبه کد زیر ارائه شده است. مشخصه‌های *rows* و *columns* تعداد سطرها و ستونها در ماتریس می‌باشند.

```
MATRIX-MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2    then error "incompatible dimensions"
3    else  for i ← 1 to rows[A]
4              do for j ← 1 to columns[B]
5                     do C[i, j] ← 0
6                        for k ← 1 to columns[A]
7                            do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8          return C
```

---

1. Matrix - chain Multiplication
2. fully parenthesized

تنها اگر دو ماتریس A و B *سازگار*[1] باشند، می‌توانیم آنها را ضرب کنیم: تعداد ستونهای A باید با تعداد سطرهای B برابر باشد. اگر A یک ماتریس $p \times q$ و B یک ماتریس $q \times r$ باشد، ماتریس حاصل C یک ماتریس $p \times r$ است. زمان محاسبهٔ C، توسط تعداد ضرب‌های عددی در خط ۷ که pqr است تحت الشعاع قرار می‌گیرد. در زیر، هزینه‌ها را از نظر تعداد ضربهای عددی نشان می‌دهیم.

برای نشان دادن هزینه‌های مختلفی که از پرانتزگذاریهای متفاوت حاصلضرب ماتریس‌ها بوجود می‌آید، مسئله زنجیره سه ماتریس $< A_1, A_2, A_3 >$ را در نظر بگیرید. فرض کنید ابعاد ماتریس‌ها به ترتیب 10×100، 100×5 و 5×50 باشند. اگر مطابق با پرانتزگذاری $((A_1\ A_2)\ A_3)$، ضرب کنیم $10 . 100 . 5 = 5000$ ضرب عددی برای محاسبه ماتریس 10×5 حاصلضرب $A_1\ A_2$، به اضافهٔ $10 . 5 . 50 = 2500$ ضرب عددی دیگر برای ضرب این ماتریس در A_3، و در مجموع 7500 ضرب عددی انجام می‌دهیم. اگر به جای آن مطابق با پرانتزگذاری $(A_1\ (A_2\ A_3))$ ضرب کنیم، $100 . 5 . 50 = 25000$ ضرب عددی برای محاسبه ماتریس 100×50 حاصلضرب A_2A_3، به اضافه $10 . 100 . 50 = 50000$ ضرب عددی دیگر برای ضرب این ماتریس در A_1 و در مجموع 75000 ضرب عددی انجام می‌دهیم. بنابراین محاسبه حاصلضرب، مطابق با پرانتزگذاری اول، 10 برابر سریعتر است.

مسئله ضرب زنجیره ماتریس‌ها می‌تواند به صورت زیر بیان شود: یک زنجیره از n ماتریس $< A_1, A_2, ..., A_n >$ داده شده که برای $i = 1, 2, ..., n$، ماتریس A_i دارای ابعاد $p_{i-1} \times p_i$ است، حاصلضرب $A_1\ A_2\ ...\ A_n$ را به روش پرانتزگذاری حل کنید که تعداد ضربهای عددی مینیمم شود.

توجه کنید که در مسئله ضرب زنجیره‌ای ماتریس‌ها، در واقع ماتریس‌ها را ضرب نمی‌کنیم. هدف ما تنها مشخص کردن ترتیبی برای ضرب ماتریس‌ها است که کمترین هزینه را داشته باشد. نوعاً زمانی که صرف مشخص کردن این ترتیب بهینه می‌شود، بیشتر از زمانی است که بعداً هنگامی که واقعاً ضرب ماتریس‌ها را بصورت بهینه انجام می‌دهیم، صرف می‌شود. (مانند انجام تنها 7500 ضرب عددی بجای $75,000$ ضرب عددی)

محاسبه تعداد پرانتزگذاری‌ها

قبل از حل مسئله ضرب زنجیره‌ای ماتریس‌ها توسط برنامه سازی پویا، اجازه دهید خود را متقاعد کنیم که امتحان همهٔ پرانتزگذاریهای ممکن به طور کامل الگوریتم کارآمدی حاصل نمی‌کند. تعداد پرانتزگذاریهای متناوب برای یک توالی از n ماتریس را با $P(n)$ نشان می‌دهیم. زمانی که $n = 1$ است، تنها یک ماتریس و بنابراین تنها یک راه برای پرانتزگذاری کامل حاصلضرب ماتریس، وجود دارد. زمانی که $n \geq 2$ است، ماتریس حاصلضرب پرانتزگذاری شده، حاصلضرب دو ماتریس است که هر

---

1. compatible

یک از آنها، ماتریس زیر حاصلضرب پرانتزگذاری شده می‌باشد و شکاف بین دو زیر حاصلضرب، ممکن است برای هر $k = 1, 2, ..., n - 1$ بین kامین و $(k + 1)$ امین ماتریس بوجود آید. بنابراین به رابطهٔ بازگشتی زیر می‌رسیم

$$P(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2\,. \end{cases} \qquad (15.11)$$

مسئله ۴-۱۲ از شما می‌خواهد که نشان دهید حل یک رابطه بازگشتی مشابه یک توالی از اعداد Catalan[1] است که بصورت $\Omega(4^n/n^{3/2})$ رشد می‌کند. یک تمرین ساده‌تر (تمرین ۳- ۱۵.۲ را ملاحظه کنید.) این است که نشان دهید حل رابطهٔ بازگشتی (۱۵.۱۱)، $\Omega(2^n)$ است. بنابراین تعداد راه حل‌ها بصورت نمایی در nاست و از اینرو روش قدرتمندانه جستجوی کامل، تدبیر ضعیفی برای مشخص کردن پرانتزگذاری بهینه در یک زنجیرهٔ ماتریس می‌باشد.

مرحله ۱: ساختار یک پرانتزگذاری بهینه

اولین مرحله در الگوی برنامه سازی پویا، پیدا کردن زیر ساختار بهینه و سپس استفاده از آن برای ساخت یک جواب بهینه از روی جوابهای بهینهٔ زیر مسئله‌های تشکیل دهنده مسئله می‌باشد. می‌توانیم برای مسئله ضرب زنجیره‌ای ماتریس‌ها، این مرحله را به صورت زیر انجام دهیم. برای راحتی کار، علامت $A_{i,j}$را که $i \leq j$است، بعنوان ماتریس که از محاسبه حاصلضرب $A_i A_{i+1} ... A_j$ بدست می‌آید قرار می‌دهیم. می‌بینید که اگر مسئله غیر بدیهی باشد یعنی $i < j$ باشد، آنگاه هر پرانتزگذاری حاصلضرب $A_i A_{i+1} ... A_j$ باید حاصلضرب را بین A_kو A_{k+1} برای مقادیر صحیح k بین بازهٔ $i \leq k < j$ تقسیم کند. به عبارت دیگر برای یک مقدار k ابتدا ماتریسهای $A_{i..k}$و $A_{k+1..j}$را محاسبه کرده و سپس برای ایجاد حاصلضرب نهایی $A_{i..j}$، آنها را در هم ضرب می‌کنیم. بنابراین هزینهٔ این پرانتزگذاری، هزینه محاسبه ماتریس $A_{i..k}$، بعلاوه هزینه محاسبه $A_{k+1..j}$ بعلاوهٔ هزینهٔ ضرب این دو در یکدیگر است.

زیر ساختار بهینه این مسئله به صورت زیر است. فرض کنید که یک پرانتزگذاری بهینه $A_i A_{i+1} ... A_j$، حاصلضرب را بین A_kو A_{k+1} بشکند. آنگاه پرانتزگذاری زیر زنجیرهٔ "پیشوندی[2]" $A_i A_{i+1} ... A_k$در این پرانتزگذاری بهینه $A_i A_{i+1} ... A_j$، باید یک پرانتزگذاری بهینهٔ $A_i A_{i+1} ... A_k$ باشد. چرا؟ اگر روش کم هزینه‌تری برای پرانتزگذاری $A_i A_{i+1} ... A_k$ وجود داشته باشد با جایگزینی آن پرانتزگذاری، پرانتزگذاری دیگری برای $A_i A_{i+1} ... A_j$بدست می‌آید که

---

1. Catalan numbers

2. prefix

هزینه آن کمتر از مقدار بهینه است، که این یک تناقض است. دیدگاه مشابهی برای پرانتزگذاری زیر زنجیره $A_{k+1} A_{k+2} ... A_j$ در پرانتزگذاری بهینهٔ $A_i A_{i+1} ... A_j$ برقرار است: این پرانتزگذاری باید یک پرانتزگذاری بهینه برای $A_{k+1} A_{k+2} ... A_j$ باشد.

اکنون از زیر ساختار بهینه خود استفاده می‌کنیم تا نشان دهیم که می‌توانیم یک جواب بهینه برای مسئله، از روی جوابهای بهینه زیر مسئله‌ها بسازیم. دیدیم که هر جوابی برای نمونه غیر بدیهی مسئله ضرب زنجیره‌ای ماتریس‌ها، ما را به شکستن حاصلضرب ملزم می‌کند و دیدیم که هر جواب بهینه، شامل جواب‌های بهینه نمونه زیر مسئله‌هاست. بنابراین، می‌توانیم با تقسیم مسئله به دو زیر مسئله (پرانتزگذاری بهینه $A_i A_{i+1} ... A_k$ و $A_{k+1} A_{k+2} ... A_j$) و پیدا کردن جوابهای بهینه نمونه زیر مسئله‌ها و سپس ترکیب این جوابهای بهینه، یک جواب بهینه برای نمونه‌ای از مسئله ضرب زنجیره‌ای ماتریسها بسازیم. زمانی که یک مکان صحیح برای شکستن ضرب را جستجو می‌کنیم، باید مطمئن شویم که همهٔ مکانهای ممکن را در نظر گرفته‌ایم و از اینرو مطمئنیم که مکان بهینه را بررسی کرده‌ایم.

مرحله ۲: حل بازگشتی

سپس، هزینه جواب بهینه را به صورت بازگشتی با توجه به جوابهای بهینهٔ زیر مسئله‌ها تعریف می‌کنیم. برای مسئله ضرب زنجیره‌ای ماتریس‌ها، مسائل مشخص کردن هزینهٔ مینیمم پرانتزگذاری $A_i A_{i+1} ... A_j$ برای $1 \leq i \leq j \leq n$ را به عنوان زیر مسئله‌ها انتخاب می‌کنیم. فرض کنید $m[i,j]$ مینیمم تعداد ضربهای عددی مورد نیاز برای محاسبه ماتریس $A_{i..j}$ باشند؛ بنابراین برای کل مسئله، هزینهٔ ارزانترین روش برای محاسبه $A_{1..n}$، $m[1,n]$ می‌باشد.

می‌توانیم $m[i,j]$ را بطور بازگشتی به صورت زیر تعریف کنیم. اگر $i = j$ باشد، مسئله بدیهی است؛ زنجیره فقط شامل یک ماتریس $A_{i...j} = A_i$ است، لذا هیچ ضرب عددی برای محاسبه حاصلضرب نیاز نمی‌باشد. بنابراین برای $i=1, 2, ... , n$، $m[i,j] = 0$ است. برای محاسبه $m[i,j]$ وقتی $i < j$ است، از ساختار جواب بهینه در مرحله ۱ استفاده می‌کنیم. فرض می‌کنیم پرانتزگذاری بهینه، حاصلضرب $A_i A_{i+1} .. A_j$ را بین A_k و A_{k+1} می‌شکند که $i \leq k < j$ است. آنگاه $m[i,j]$ برابر است با هزینهٔ مینیمم محاسبه زیر حاصلضربهای $A_{i..k}$ و $A_{k+1..j}$، به اضافه هزینهٔ ضرب این دو ماتریس در یکدیگر. با یادآوری اینکه هر ماتریس A_i، $p_{i-1} \times p_i$ است، می‌بینیم که محاسبه حاصلضرب $A_{i..k}$ $A_{k+1..j}$، $p_{i-1} p_k p_j$ ضرب عددی نیاز دارد. بنابراین به تساوی زیر می‌رسیم.

$$m[i,j] = m[i,k] + m[k+1, j] + p_{i-1} p_k p_j$$

این معادلهٔ بازگشتی فرض می‌کند که مقدار k را می‌دانیم، در حالیکه مقدار k مشخص نیست. اما تنها $j - i$ مقدار ممکن برای k وجود دارد، یعنی $k = i, i+1, ..., j-1$ است. از آنجا که پرانتزگذاری بهینه باید از یکی از این مقادیر k استفاده کند، تنها لازم است همه اینها را برای پیدا کردن بهترین،

بررسی کنیم. بنابراین تعریف بازگشتی هزینه مینیمم پرانتزگذاری روی حاصلضرب $A_i A_{i+1} \dots A_j$ به صورت زیر در می‌آید:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j\ . \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j\ . \end{cases} \quad (۱۵.۱۲)$$

مقادیر $m[i,j]$ هزینه‌های جواب‌های بهینه زیر مسئله‌ها را می‌دهند. برای کمک به دنبال کردن چگونگی ساخت جواب بهینه، $s[i,j]$ را تعریف می‌کنیم که مقداری از k است که در آن می‌توانیم حاصلضرب $A_i A_{i+1} \dots A_j$ را برای دستیابی به پرانتزگذاری بهینه، بشکنیم. به عبارت دیگر $s[i,j]$ با مقدار k برابر است بطوریکه $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$

مرحله ۳: محاسبه هزینه‌های بهینه

در این مرحله، نوشتن یک الگوریتم بازگشتی بر پایه رابطه بازگشتی (۱۵.۱۲) برای محاسبه هزینه مینیمم $m[1,n]$ جهت انجام ضرب $A_1 A_2 \dots A_n$ کار ساده ایست. اما همانطور که در بخش ۱۵.۳ خواهیم دید، این الگوریتم دارای زمانی نمایی است که بهتر از روش قدرتمندانه چک کردن همه راههای پرانتزگذاری حاصلضرب، نمی‌باشد.

نکته مهمی که ممکن است در اینجا بدست آوریم، این است که تعداد نسبتاً کمتری زیر مسئله داریم: یک مسئله برای هر یک از انتخابهای i و j با برقراری $1 \le i \le j \le n$ یا در کل $\binom{n}{2} + n = \Theta(n^2)$ مسئله. ممکن است الگوریتم بازگشتی، چندین بار با هر زیر مسئله، در شاخه‌های مختلف درخت بازگشتیش مواجه شود. این ویژگی همپوشانی زیر مسئله‌ها، دومین نشانه قابلیت برنامه سازی پویا است. (اولین نشانه زیر ساختار بهینه است.)

به جای محاسبه بازگشتی جواب معادله بازگشتی (۱۵.۱۲)، سومین مرحله الگوی برنامه سازی پویا را اجرا کرده و هزینه بهینه را با استفاده از روش جدولی پایین به بالا محاسبه می‌کنیم. شبه کد زیر فرض می‌کند که برای $i = 1, 2, \dots, n$ ماتریس A_i دارای ابعاد $p_{i-1} \times p_i$ است. ورودی، توالی $p = \langle p_0, p_1, \dots, p_n \rangle$ است که $length[p] = n + 1$ می‌باشد. روال، از جدول کمکی $m[1..r\ ;\ 1..n]$ برای ذخیره هزینه‌های $m[i,j]$ و از جدول کمکی $s[1..n\ ,\ 1..n]$ که ثبت می‌کند کدام شاخص k در محاسبه $m[i,j]$ و از جدول کمکی $s[1..n,1..n]$ که ثبت می‌کند کدام شاخص k در محاسبه $m[i,j]$ به هزینه بهینه رسیده است، استفاده می‌نماید. از جدول s برای ساخت جواب بهینه استفاده خواهیم کرد.

برای پیاده سازی صحیح روش پایین به بالا، باید مشخص کنیم، کدام ورودیهای جدول، در محاسبه $m[i,j]$ مورد استفاده قرار می‌گیرند. معادله (۱۵.۱۲) نشان می‌دهد که هزینه $m[i,j]$ برای محاسبه حاصلضرب زنجیره‌ای ماتریسها با $j - i + 1$ ماتریس، تنها به هزینه‌های محاسبه

حاصلضربهای زنجیره‌ای ماتریسهایی با کمتر از $j - i + 1$ ماتریس بستگی دارد. به عبارت دیگر برای $k = i, i + 1, ..., j$ ماتریس $A_{i..k}$، حاصلضرب $j-i + 1 > k-i+1$ ماتریس، و ماتریس $A_{k+1..j}$، حاصلضرب $j - i + 1 > j - k$ ماتریس است. بنابراین الگوریتم، باید طوری جدول m را پر کند که با حل مسئله پرانتزگذاری زنجیره‌های ماتریسی با طول صعودی، مطابقت داشته باشد.

```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3      do m[i, i] ← 0
 4  for l ← 2 to n              ▷ l is the chain length.
 5      do for i ← 1 to n − l + 1
 6             do j ← i + l − 1
 7                m[i, j] ← ∞
 8                for k ← i to j − 1
 9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                       if q < m[i, j]
11                          then m[i, j] ← q
12                               s[i, j] ← k
13  return m and s
```

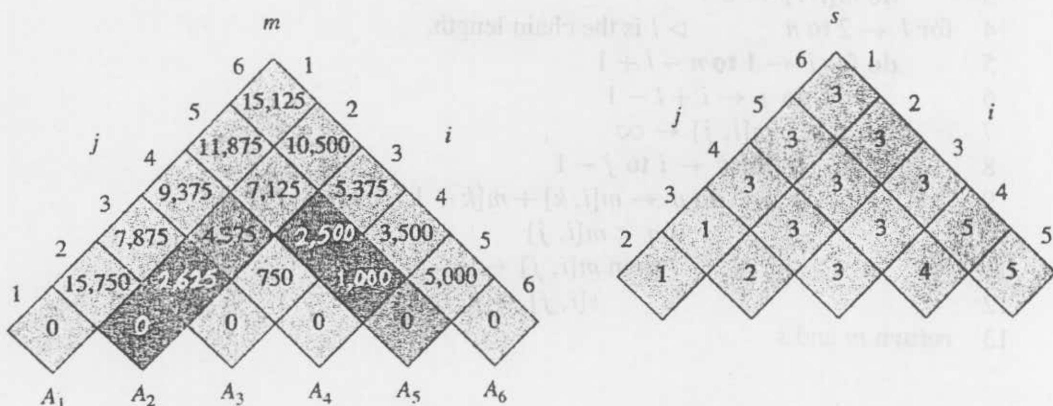
ابتدا الگوریتم در خطوط ۳-۲، برای $i = 1,2,...,n$، $m[i,i] \leftarrow 0$ (مینیمم هزینه برای زنجیره‌هایی با طول l) را محاسبه می‌کند. سپس با استفاده از رابطه بازگشتی (۱۵.۱۲) در طول اولین اجرای حلقه در خطوط ۱۲-۴، برای $i = 1, 2, ..., n-1$، $m[i, i + 1]$ (مینیمم هزینه برای زنجیره‌هایی با طول $l = 2$) را محاسبه می‌کند. دومین بار در حلقه، برای $i=1,2,...,n-2$، $m[i, i + 2]$ (مینیمم هزینه برای زنجیره‌هایی با طول $l = 3$) را محاسبه می‌کند، و به همین ترتیب در هر مرحله، هزینه $m[i,j]$ محاسبه شده در خطوط ۱۲-۹، فقط بستگی به ورودیهای قبلاً محاسبه شده جدول، یعنی $m[i,k]$ و $m[k+1 , j]$ دارد.

شکل ۱۵.۳، این روال را با یک زنجیره $n = 6$ ماتریس، توضیح می‌دهد. چون $m[i,j]$ را تنها برای $i \leq j$ تعریف کرده‌ایم، فقط بخشی از جدول m که دقیقاً بالای قطر اصلی قرار دارد مورد استفاده قرار می‌گیرد. شکل نشان می‌دهد که جدولی که برای ساخت قطر اصلی چرخانده شده، به صورت افقی اجرا می‌شود. زنجیره ماتریس‌ها در زیر آن لیست شده است. در این طرح، مینیمم هزینه $m[i,j]$ برای ضرب زیر زنجیره $A_i A_{i+1} ... A_j$ از ماتریسها، می‌تواند در نقطه برخورد خطوط اجرا شده از شمال شرقی A_i و شمال غربی A_j پیدا شود. هر ردیف افقی در جدول شامل ورودیهایی برای زنجیره‌های ماتریس است که طول یکسانی دارند.

$MATRIX\text{-}CHAIN\text{-}ORDER$ ردیف‌ها را از پایین به بالا و در هر ردیف از چپ به راست محاسبه می‌کند. ورودی $m[i,j]$ با استفاده از حاصلضرب $p_{i-1}p_kp_j$ برای $k = i, i + 1, ..., j - 1$ و همه

ورودیهای جنوب غرب و جنوب شرقی $m[i,j]$ محاسبه می‌شود.

یک بررسی ساده حلقه‌های تو در توی *MATRIX-CHAIN-ORDER*، نشان می‌دهد که زمان اجرای الگوریتم، $O(n^3)$ است. سه حلقه تودرتو وجود دارد که شاخص هر حلقه (l، i و k) حداکثر $n-1$ مقدار می‌گیرد. تمرین ۴-۱۵.۲ از شما می‌خواهد که نشان دهید، زمان اجرای این الگوریتم در حقیقت $\Omega(n^3)$ نیز می‌باشد. الگوریتم به فضای $\Theta(n^2)$ برای ذخیره جدولهای m و s نیاز دارد. بنابراین، *MATRIX-CHAIN-ORDER* خیلی مؤثرتر از روشی است که با زمانی نمایی، همه پرانتزگذاریهای ممکن را بر شمرده و هر یک را بررسی می‌کند.



شکل ۱۵.۳ جدولهای m و s که توسط *MATRIX-CHAIN-ORDER* برای $n = 6$ و با ماتریسهایی با ابعاد زیر محاسبه شده‌اند:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

جدولها، به منظور اینکه قطر اصلی به صورت افقی اجرا شود چرخانده می‌شوند. فقط قطر اصلی و مثلث بالای آن برای جدول m مورد استفاده قرار می‌گیرد و برای جدول s تنها مثلث بالا مورد استفاده قرار می‌گیرد. تعداد مینیمم ضربهای عددی برای ضرب 6 ماتریس $m[1,6] = 15\ 125$ است. زمانی که عبارت زیر را محاسبه می‌کنیم، در خط ۹ با ورودیهای تیره‌تر، جفت‌هایی را که یکرنگ هستند با هم در نظر می‌گیریم.

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125\,.$$

مرحله ۴: ساخت جواب بهینه

گرچه *MATRIX-CHAIN-ORDER* تعداد بهینه ضرب‌های عددی مورد نیاز برای محاسبه ضرب زنجیره‌ای ماتریسها را مشخص می‌کند، دقیقاً چگونگی ضرب ماتریسها را نشان نمی‌دهد. ساخت جواب بهینه از روی اطلاعاتی که محاسبه شده و در جدول $S[1..n, 1..n]$ ذخیره شده‌اند، مشکل نیست. هر ورودی $s[i,j]$ مقدار k را ثبت می‌کند بطوریکه پرانتزگذاری بهینه $A_i A_{i+1} \dots A_j$، حاصلضرب را بین A_k و A_{k+1} می‌شکند. بنابراین، می‌دانیم که ضرب ماتریس نهایی در محاسبه بهینه $A_{1...s[1,n]}$ $A_{1..n}$ $A_{s[l,n]+1..n}$ است. ضربهای قبلی ماتریس می‌توانند به صورت بازگشتی محاسبه شوند، چون $5[1,s[1,n]]$ آخرین ضرب ماتریس در محاسبه $A_{1...s[1,n]}$ و $s[s[1,n]+1,n]$ آخرین ضرب ماتریس در محاسبه $A_{s[1,n]+1..n}$ را مشخص می‌کنند. روال بازگشتی زیر، با گرفتن جدول s محاسبه شده توسط *MATRIX-CHAIN-ORDER* و اندیس‌های i,j، پرانتزگذاری بهینه $<A_i A_{i+1} \dots A_j>$ را چاپ می‌کند. فراخوانی ابتدایی *PRINT-OPTIMAL-PARENS(s,1,n)* پرانتزگذاری بهینه $<A_1 A_2 .. A_n>$ را چاپ می‌کند.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i = j
2     then print "A"_i
3     else print "("
4          PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5          PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6          print ")"
```

در مثال شکل ۱۵.۵، فراخوانی *PRINT-OPTIMAL-PARENS(s,1,6)*، پرانتزگذاری $((A_1(A_2A_3))((A_4A_5)A_6))$ را چاپ می‌کند.

تمرین‌ها

۱-۱۵.۲ پرانتزگذاری بهینه یک ضرب زنجیره‌ای ماتریس‌ها را پیدا کنید بطوریکه توالی ابعاد آن $<5, 10, 3, 12, 5, 50, 6>$ باشد.

۲-۱۵.۲ الگوریتم بازگشتی *MATRIX-CHAIN-MULTIPLY(A,s,i,j)* را طوری بنویسید که با گرفتن توالی ماتریسهای $<A_1 A_2 .. A_n>$، جدول s محاسبه شده با *MATRIX-CHAIN-ORDER* و اندیس‌های i و j واقعاً ضرب بهینه زنجیره‌ای ماتریسها را انجام دهد. (فراخوانی اولیه به صورت *MATRIX-CHAIN-MULTIPLY(A,S,1,n)* می‌باشد.)

۳-۱۵.۲ با استفاده از روش جایگذاری، نشان دهید که جواب معادله بازگشتی (۱۵.۱۱)، $\Omega(2^n)$ است.

۴-۱۵.۲ فرض کنید $R(i,j)$ تعداد دفعاتی است که در یک فراخوانی *MATRIX-CHAIN-ORDER* به ورودی $m[i,j]$ جدول، هنگامی محاسبه ورودیهای دیگر مراجعه می‌کنیم. نشان دهید که تعداد کل ارجاعها برای کل جدول برابر است با

$$\sum_{i=1}^{n}\sum_{j=i}^{n} R(i, j) = \frac{n^3 - n}{3} .$$

(راهنمایی: ممکن است سری $\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$ مفید باشد.)

۵-۱۵.۲ نشان دهید که پرانتزگذاری کامل یک عبارت n عنصری، دقیقاً $n-1$ جفت پرانتز دارد.

۱۵.۳ عناصر برنامه سازی پویا

گرچه تنها دو مثال از روش برنامه سازی پویا را بررسی کرده‌ایم، ممکن است هنوز سر در گم باشید که دقیقاً چه زمانی این روش بکار می‌رود. از نقطه نظر تخصصی، چه زمانی باید برای یک مسئله، به دنبال راه حل برنامه سازی پویا برویم؟ در این بخش دو عامل کلیدی که یک مسئله بهینه سازی باید داشته باشد تا برنامه سازی پویا برای آن قابل اعمال باشد، را بررسی می‌کنیم: زیر ساختار بهینه و زیر مسئله‌های متداخل. همچنین روشی دیگر بنام *memoization*[1] را برای بهره گرفتن از ویژگی زیر مسئله‌های متداخل، بررسی می‌کنیم.

زیر ساختار بهینه

اولین مرحله در حل یک مسئله بهینه سازی توسط برنامه سازی پویا، مشخص کردن ساختار جواب بهینه است. یادآوری می‌کنیم که اگر جواب بهینه مسئله شامل جوابهای بهینه زیر مسئله‌ها باشد، مسئله *زیرساختار بهینه*[2] را ارائه می‌کند. زمانی که مسئله، زیر ساختار بهینه را ارائه می‌دهد، نشانه خوبیست از اینکه برنامه سازی پویا ممکن است بکار رود. (هر چند ممکن است به این معنا نیز باشد که روش حریصانه بکار رود. به فصل ۱۶ مراجعه کنید.) در برنامه سازی پویا، جواب بهینه مسئله را از روی جواب‌های بهینه زیر مسئله‌ها می‌سازیم. در نتیجه باید مطمئن شویم بازه‌ای که برای زیر مسئله‌ها در نظر گرفته‌ایم، شامل آنهایی که در جواب بهینه مورد استفاده قرار می‌گیرند، باشد.

زیر ساختار بهینه هر دو مسئله‌ای که تا اینجا در این فصل بررسی کرده‌ایم را مشخص نمودیم. در بخش ۱۵.۱، دیدیم که سریعترین مسیر تا ایستگاه j در یکی از خطوط، شامل سریعترین مسیر تا ایستگاه $j-1$ در یک خط است. در بخش ۱۵.۲ دیدیم که پرانتزگذاری بهینه $A_i A_{i+1} \dots A_j$ که ضرب

۱- این غلط املایی نیست کلمه واقعاً *memoization* است نه *memorization*. *memoization* از *memo* می‌آید چون تکنیک آن شامل ثبت یک مقدار است به این منظور که بعداً به آن مراجعه کنیم.

2. optimal substructure

را بین A_k و A_{k+1} می‌شکند، شامل جواب‌های بهینه مسئله‌های پرانتزگذاری شده $A_i A_{i+1} \dots A_k$ و $A_{k+1} A_{k+2} \dots A_k$ است.

شما برای پیدا کردن زیر ساختار بهینه از الگوی رایج زیر پیروی می‌کنید:

۱. نشان می‌دهید که جواب مسئله شامل انجام یک انتخاب است، مانند انتخاب ایستگاه بعدی خط مونتاژ یا انتخاب اندیسی که در آن زنجیره ماتریسها بشکند. انجام این انتخاب، یک یا چند زیر مسئله را برای حل شدن باقی می‌گذارد.

۲. فرض می‌کنید که برای یک مسئله مفروض، انتخابی که منجر به جواب بهینه می‌شود، داده شده است شما هنوز نگران چگونگی مشخص کردن این انتخاب نیستید. فقط فرض می‌کنید که این انتخاب به شما داده شده است.

۳. با دریافت این انتخاب، مشخص می‌کنید که چه زیر مسئله‌هایی رخ داده‌اند و چطور به بهترین صورت، فضای نتیجه شده از زیر مسئله‌ها را مشخص کنیم.

۴. نشان می‌دهید که جواب زیر مسئله‌هایی که در جواب بهینه مسئله مورد استفاده قرار گرفته‌اند، خودشان باید استفاده از تکنیک "برش و الصاق"[1] بهینه شوند. این کار را فرض اینکه هر یک از جوابهای زیر مسئله بهینه نیست و آنگاه رسیدن به تناقض، انجام می‌دهید. به خصوص با "برش"[2] جواب غیر بهینه زیر مسئله و "الصاق"[3] جواب بهینه، نشان می‌دهید که می‌توانید به جواب بهتری برای مسئله اولیه برسید، بنابراین با این فرض شما که دانستن یک جواب بهینه از قبل بود، تناقض دارد. اگر بیش از یک زیرمسئله وجود داشته باشند، معمولاً بسیار مشابه هم هستند، بطوریکه اثبات برش و الصاق برای یکی از آنها می‌تواند با اندکی تلاش برای سایر زیر مسئله‌ها تغییر یابد.

برای مشخص کردن فضای زیر مسئله‌ها، یک حساب سر انگشتی خوب این است که سعی کنیم به ساده‌ترین روش ممکن، فضا را نگهداری کنیم و سپس هنگام نیاز آن را توسعه دهیم. برای مثال، فضای زیر مسئله‌هایی که برای زمان بندی خط مونتاژ فرض کردیم، سریعترین راه از ورود به کارخانه تا ایستگاه $S_{1,j}$ و $S_{2,j}$ بود. این فضای زیر مسئله به خوبی عمل کرد و نیازی به فضای دائمی بیشتری برای زیر مسئله‌ها نبود.

برعکس فرض کنید سعی کرده‌ایم فضای زیر مسئله خود را برای ضرب زنجیره‌ای ماتریس‌ها، به ضرب ماتریس‌ها به شکل $A_1 A_2 \dots A_j$ محدود کنیم. همانند قبل، پرانتزگذاری بهینه باید این ضرب را بین A_k و A_{k+1}، برای $1 \le k \le j$ بشکند. در می‌یابیم که زیر مسئله‌هایی به شکل $A_1 A_2 \dots A_k$ و $A_{k+1} A_{k+2} \dots A_j$ داریم و اینکه زیر مسئله دوم به شکل $A_1 A_2 \dots A_j$ نیست مگر آنکه تضمین کنیم k همیشه با $j-1$ برابر است. برای این مسئله، لازم است اجازه دهیم زیر مسئله‌ها در نقاط پایانی متفاوت

---

1. *cut-and-paste*
2. *cutting out*
3. *pasting in*

باشند به عبارت دیگر، اجازه دهیم هم i و هم j در زیر مسئله $A_i A_{i+1} \dots A_j$ تغییر نمایند.

زیر ساختار بهینه‌ای که در این مسئله تغییر نمی‌کند به دو روش محدود می‌شود:

۱. چه تعداد زیر مسئله‌هایی در جواب برای مسئله اصلی استفاده می‌شوند و

۲. چه تعداد انتخاب، برای تعیین اینکه کدام زیر مسئله (ها) در جواب بهینه استفاده شوند داریم.

در زمان‌بندی خط مونتاژ، جواب بهینه فقط از یک زیر مسئله استفاده می‌کند، ولی ما دو انتخاب را برای تعیین جواب بهینه در نظر می‌گیریم. برای پیدا کردن سریعترین مسیر تا ایستگاه $S_{i,j}$ از سریعترین راه تا $S_{1,j}$ یا از سریعترین راه تا $S_{2,j-1}$ استفاده می‌کنیم. هر کدام را که استفاده کنیم، یک زیر مسئله را ارائه می‌دهد که باید بصورت بهینه آن را حل کنیم. ضرب زنجیره‌ای ماتریس‌ها برای زیر زنجیره $A_i A_{i+1} \dots A_j$ به عنوان مثالی با دو زیر مسئله و $j-1$ انتخاب اجرا می‌شود. برای ماتریس داده شده A_k که در آن ضرب را می‌شکنیم، دو زیر مسئله وجود دارد - پرانتزگذاری $A_i A_{i+1} \dots A_k$ و پرانتزگذاری $A_{k+1} A_{k+2} \dots A_k$ - و باید هر دو را به صورت بهینه حل نمائیم. هنگامی که جوابهای بهینه زیر مسئله‌ها را تعیین می‌کنیم، اندیس k را از بین $j - i$ کاندیدا انتخاب می‌کنیم.

به طور غیر رسمی، زمان اجرای یک الگوریتم برنامه سازی پویا، به حاصلضرب دو عامل بستگی دارد: تعداد زیر مسئله‌ها در مجموع و تعداد انتخابهایی که برای هر زیر مسئله در نظر می‌گیریم. در زمان بندی خط مونتاژ، در مجموع $\Theta(n)$ زیر مسئله و برای بررسی هر یک تنها 2 انتخاب داشتیم که زمان اجرای $\Theta(n)$ را نتیجه می‌دهد. برای ضرب زنجیره‌ای ماتریسها، در کل $\Theta(n^2)$ زیر مسئله وجود داشت و در هر یک حداکثر $n-1$ انتخاب داشتیم که زمان اجرای $O(n^3)$ را نتیجه می‌دهد.

برنامه سازی پویا از زیر ساختار بهینه، به روش پایین به بالا استفاده می‌کند. به عبارت دیگر ابتدا جواب بهینه زیر مسئله‌ها را پیدا می‌کنیم و با استفاده از حل زیر مسئله‌ها، جواب بهینه مسئله را پیدا می‌کنیم. یافتن یک جواب بهینه برای مسئله، مستلزم انتخابی از بین زیر مسئله‌ها می‌باشد که در حل مسئله از آنها استفاده خواهیم کرد. هزینه جواب مسئله معمولاً برابر هزینه زیر مسئله‌ها، بعلاوه هزینه ایست که مستقیماً به خود انتخاب نسبت داده می‌شود. برای مثال در زمان بندی خط مونتاژ، ابتدا زیر مسئله‌های یافتن سریعترین مسیر تا ایستگاه $S_{1,j-1}$ و $S_{2,j-1}$ را حل کردیم و سپس یکی از ایستگاهها را به عنوان ایستگاه قبل از $S_{1,j}$ انتخاب نمودیم. هزینه ناشی از خود انتخاب، به اینکه آیا خطوط را بین ایستگاههای $j-1$ و j عوض می‌کنیم بستگی دارد؛ اگر در همان خط بمانیم هزینه برابر $a_{i,j}$ و اگر خط را عوض کنیم برابر $t_{i,j-1} + a_{i,j}$ است که $i' \neq j$ می‌باشد. در ضرب زنجیره‌ای ماتریس‌ها، پرانتزگذاری بهینه زیر زنجیره‌های $A_i A_{i+1} \dots A_j$ را مشخص کرده و سپس ماتریس A_k که در آن شکسته می‌شود را انتخاب می‌کنیم. هزینه ناشی از خود انتخاب عبارت $P_{i-1} P_k P_j$ است.

در فصل ۱۶، الگوریتم‌های حریصانه را بررسی خواهیم کرد که بسیار شبیه به برنامه سازی پویا هستند. خصوصاً مسائلی که برای آنها الگوریتم‌های حریصانه کاربرد دارد، دارای زیر ساختار بهینه

هستند. یک تفاوت برجسته بین الگوریتم‌های حریصانه و برنامه سازی پویا، این است که در الگوریتم‌های حریصانه از زیر ساختار بهینه به روش بالا به پایین استفاده می‌کنیم. به جای اینکه ابتدا جواب بهینه زیر مسئله‌ها را پیدا کنیم و سپس انتخاب را انجام دهیم، الگوریتم‌های حریصانه ابتدا انتخاب می‌کنند - انتخابی که در آن زمان بهترین به نظر آید - و سپس زیر مسئله حاصل را حل می‌کنند.

نکات ظریف

فرد باید مواظب باشد که وقتی زیر ساختار بهینه کاربرد ندارد آن را در نظر نگیرد. دو مسئله زیر را در نظر بگیرید که در آنها گراف جهتدار $G = (V,E)$ در رئوس $v,u \in V$ داده شده است.

کوتاهترین مسیر بی‌وزن[1,2]: پیدا کردن مسیری از u به v که شامل یالهای کمتری باشد. چنین مسیری باید ساده باشد. چون حذف دور از مسیر، مسیری با یال‌های کمتر ایجاد می‌کند.

طولانی‌ترین مسیر ساده بی‌وزن[3]: یافتن مسیر ساده از u به v که شامل بیشترین یالها است. باید لزوم ساده بودن را لحاظ نماییم چون در غیر اینصورت می‌توانیم یک دور آنقدر بپیماییم تا مسیرهایی با تعداد یال به دلخواه بزرگ بسازیم.

مسئله کوتاهترین مسیر بی‌وزن زیر ساختار بهینه‌ای به صورت زیر ارائه می‌دهد. فرض کنید $u \neq v$ است، بنابراین مسئله بدیهی نیست. آنگاه هر مسیر p از u به v باید شامل یک رأس میانی w باشد. (توجه کنید که w ممکن است u یا v باشد.) بنابراین می‌توانیم مسیر $u \overset{p}{\leadsto} v$ را به زیر مسیرهای $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$ تجزیه کنیم. واضح است که تعداد یالها در p با مجموع یالها در p_1 و p_2 برابر است. ادعا می‌کنیم اگر p مسیری بهینه از u به v (یعنی کوتاهترین) باشد، آنگاه p_1 باید کوتاهترین مسیر از u به w باشد. چرا؟ از بحث "برش و الصاق" استفاده می‌کنیم: اگر مسیر دیگری بنام p'_1 از u به w با یال‌های کمتری نسبت به p_1 وجود داشته باشد آن گاه می‌توانیم p_1 را برش زده و p'_1 را به جای آن الصاق نمائیم تا مسیر $u \overset{p'_1}{\leadsto} w \overset{p_2}{\leadsto} v$ را با یالهای کمتری نسبت به p بسازیم که این با بهینه بودن p تناقض دارد. به طور متقارن، p_2 هم باید کوتاهترین مسیر از w به v باشد. بنابراین، با در نظر گرفتن همه راسهای میانی w، پیدا کردن کوتاهترین مسیر از u به w و کوتاهترین مسیر از w به v و انتخاب رأس میانی w، که کوتاهترین مسیر نهایی را نتیجه می‌دهد، می‌توانیم کوتاهترین مسیر از u به v را پیدا کنیم. در بخش ۲۵.۲، از یک دیدگاه متفاوت برای زیر ساختار بهینه استفاده می‌کنیم تا کوتاهترین مسیر بین هر دو رأس در یک گراف جهتدار وزن دار را پیدا کنیم. تصور اینکه مسئله پیدا کردن طولانی‌ترین

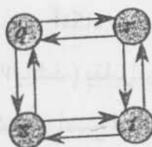
---

۱ - از اصطلاح «بی وزن» برای متمایز کردن این مسئله، از مسئله پیدا کردن کوتاهترین مسیر با یالهای وزن دار که در فصلهای ۲۴ و ۲۵ خواهیم دید، استفاده می‌کنیم. می‌توانیم از روش جستجو به ترتیب سطح در فصل ۲۲، برای حل مسئله مسیر بی‌وزن استفاده کنیم.

2. unweighted shortest path *3. unweighted longest simple path*

مسیر ساده بی‌وزن، باز هم زیر ساختار بهینه را ارائه می‌دهد، وسوسه‌انگیز است. با تمام اینها، اگر طولانی‌ترین مسیر ساده $u \overset{p}{\leadsto} v$ را به زیر مسیرهای $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$ تجزیه کنیم آیا p_1 نباید طولانی‌ترین مسیر ساده از u به w و p_2 طولانی‌ترین مسیر ساده از w به v باشد؟ جواب منفی است! شکل ۱۵.۴ یک نمونه‌ای را ارائه می‌دهد. مسیر $q \to r \to t$ را در نظر بگیرید که طولانی‌ترین مسیر ساده از q به t است. آیا $q \to r$ طولانی‌ترین مسیر ساده از q به r است؟ خیر، برای این مسئله $q \to s \to t \to r$ یک مسیر ساده ایست که طولانی‌تر می‌باشد. آیا $r \to t$ طولانی‌ترین مسیر ساده از r به t است؟ باز هم خیر، در این مورد اخیر $r \to q \to s \to t$ یک مسیر ساده است که طولانی‌تر می‌باشد. این مثال نشان می‌دهد که برای طولانی‌ترین مسیرهای ساده، نه تنها زیر ساختار بهینه وجود ندارد، بلکه نمی‌توانیم لزوماً یک جواب مجاز[1] مسئله را از جواب زیر مسئله‌ها بدست آوریم. اگر طولانی‌ترین مسیرهای ساده $q \to s \to t \to r$ و $r \to q \to s \to t$ را ترکیب کنیم، مسیر $q \to s \to t \to r \to q \to s \to t$ بدست می‌آید که یک مسیر ساده نیست.

در واقع به نظر نمی‌آید که مسئله پیدا کردن طولانی‌ترین مسیر ساده بی‌وزن، هیچ ترتیبی برای زیر ساختار بهینه داشته باشد. هیچ الگوریتم برنامه سازی پویای کار آمد، تاکنون برای این مسئله پیدا نشده است. در حقیقت این مسئله NP کامل است و به این معنا ست که بعید است بتواند در زمان چندجمله‌ای حل شود.



شکل ۱۵.۴ یک گراف جهتدار که نشان می‌دهد مسئله پیدا کردن طولانی‌ترین مسیر ساده در گراف جهتدار بی‌وزن فاقد زیر ساختار بهینه است. مسیر $q \to r \to t$ طولانی‌ترین مسیر ساده از q به t است ولی زیر مسیر $q \to r$ طولانی‌ترین مسیر ساده از q به r نیست یا زیر مسیر $r \to t$ هم طولانی‌ترین مسیر ساده از r به t نیست.

در مورد زیر ساختار طولانی‌ترین مسیر ساده که با زیر ساختار کوتاهترین مسیر بسیار متفاوت است چطور؟ گرچه دو زیر مسئله، در جواب هر دو مسئله طولانی‌ترین و کوتاهترین مسیرها استفاده شده‌اند، اما در پیدا کردن طولانی‌ترین مسیر ساده، دو زیر مسئله مستقل[2] نمی‌باشند، در حالیکه در یافتن کوتاهترین مسیرهای ساده مستقل می‌باشند. منظور از مستقل بودن زیر مسئله‌ها چیست؟ منظور این است که در یک مسئله جواب یکی از زیر مسئله‌ها به جواب زیر مسئله دیگر در همان مسئله تأثیر نمی‌گذارد. در مثال شکل ۱۵.۴، مسئله پیدا کردن طولانی‌ترین مسیر ساده از q به t با دو زیر مسئله را داریم: پیدا کردن طولانی‌ترین مسیر ساده از q به r و از r به t برای زیر مسئله اول مسیر

1. legal
2. independent

$q \to s \to t \to r$ را انتخاب می‌کنیم و بنابراین از رئوس s و t نیز استفاده کرده‌ایم. از این رئوس دیگر نمی‌توانیم در زیر مسئله دوم استفاده کنیم، چون ترکیب جواب‌های دو زیر مسئله یک مسیر ساده را نتیجه نمی‌دهد. اگر نتوانیم از رأس t در مسئله دوم استفاده کنیم، هرگز نمی‌توانیم آن را حل نماییم، چون لازم است t در مسیری که پیدا کردیم قرار داشته باشد و t رأسی نیست که در آن جواب‌های دو زیر مسئله را به هم وصل کنیم (آن رأس r می‌باشد.) استفاده از رئوس s و t در یک زیر مسئله مانع از این می‌شود که آنها در جواب زیر مسئله دیگر مورد استفاده قرار گیرند. باید حداقل از یکی از آنها برای حل زیر مسئله دیگر استفاده کنیم، و از هر دوی آنها برای حل بهینه آن استفاده نماییم. بنابراین می‌گوییم که این زیر مسئله‌ها مستقل نیستند. به نظر به روشی دیگر، استفاده از منابع (این منابع، رأسها می‌باشند) در حل یک زیر مسئله موجب می‌شود که آنها برای زیر مسئله دیگر غیر قابل استفاده شوند.

پس چرا زیر مسئله‌ها برای پیدا کردن کوتاهترین مسیر، مستقل هستند؟ جواب این است که زیر مسئله‌ها ذاتاً در منابع مشترک نیستند. ادعا می‌کنیم که اگر رأس w در کوتاهترین مسیر p از u به v قرار داشته باشد، آنگاه می‌توانیم هر کوتاهترین مسیر $u \overset{p_1}{\leadsto} w$ و هر کوتاهترین مسیر $w \overset{p_2}{\leadsto} v$ را برای تولید کوتاهترین مسیر از u به v به هم متصل کنیم. مطمئن هستیم که به جز w، هیچ رأس دیگری در هر دو مسیر p_1 و p_2 ظاهر نمی‌شود. چرا؟ فرض کنید رأس $x \neq w$ در هر دو مسیر p_1 و p_2 ظاهر شده است بنابراین می‌توانیم p_1 را به $u \overset{p_{ux}}{\leadsto} x \leadsto w$ و P_2 را به $w \leadsto x \overset{p_{xv}}{\leadsto} v$ تجزیه کنیم. بنابه زیر ساختار بهینه این مسئله، مسیر p به اندازه مجموع یالهای p_1 و p_2 یال دارد: فرض کنید می‌گوییم p e یال دارد. اکنون اجازه دهید مسیر $u \overset{p_{ux}}{\leadsto} x \overset{p_{xv}}{\leadsto} v$ را از u به v بسازیم. این مسئله حداکثر $e - 2$ یال دارد که با فرض اینکه p کوتاهترین مسیر است در تناقض است. بنابراین مطمئنیم که برای مسئله کوتاهترین مسیر، زیر مسئله‌ها مستقل هستند.

هر دو مسئله بررسی شده در بخش‌های ۱۵.۱ و ۱۵.۲، زیر مسئله‌ای مستقل دارند. در ضرب زنجیره‌ای ماتریس‌ها زیر مسئله‌ها، ضرب زیر زنجیره‌های $A_i A_{i+1} \cdots A_k$ و $A_{k+1} A_{k+2} \cdots A_j$ می‌باشند. این زیر زنجیره‌ها مجزا هستند، بنابراین هیچ ماتریس نمی‌تواند در هر دو آنها قرار داشته باشد. در زمان بندی خط مونتاژ، برای مشخص کردن سریعترین مسیر تا ایستگاه $S_{i,j}$ به سریعترین راه تا ایستگاه $S_{1,j-1}$ و $S_{2,j-1}$ می‌نگریم. چون جواب سریعترین راه تا ایستگاه $S_{i,j}$ فقط شامل یکی از این جواب‌های زیر مسئله‌هاست، آن زیر مسئله به طور خودکار از بقیه که در جواب مورد استفاده قرار گرفته‌اند، مستقل است.

زیر مسئله‌های متداخل

دومین عاملی که یک مسئله بهینه سازی باید داشته باشد تا برنامه سازی پویا قابل اعمال باشد، این است که فضای زیر مسئله‌ها باید کوچک باشند تا حدی که الگوریتم بازگشتی برای مسئله، به جای

اینکه همیشه زیر مسئله‌های جدید تولید کند همان زیر مسئله‌ها را به طور متوالی حل کند. نوعاً، تعداد کل زیر مسئله‌های مجزا یک چند جمله‌ای بر حسب اندازه ورودی است. زمانی که الگوریتم بازگشتی به یک مسئله چندین بار به طور پی در پی برخورد کند، می‌گوییم که مسئله بهینه سازی دارای *زیرمسئله‌های متداخل*[1,2] است. در مقابل مسئله‌ای که روش تقسیم و حل بر آن مناسب است معمولاً در هر مرحله بازگشت مسئله‌های کاملاً جدید تولید می‌کند. الگوریتم‌های برنامه سازی پویا نوعاً، با یکبار حل هر زیر مسئله و سپس ذخیره جواب در جدولی که بتوان هنگام نیاز، با استفاده از زمان ثابتی در هر مراجعه به جدول آن را جستجو کرد، از زیر مسئله‌های متداخل بهره می‌برند.

در بخش ۱۵.۱ به طور منحصر بررسی کردیم که چطور جواب بازگشتی برای زمانبندی خط مونتاژ برای $j = 1, 2, ..., n$ ، 2^{n-j} ارجاع به $f_i[j]$ انجام می‌دهد. جواب جدولی زمان نمایی الگوریتم بازگشتی را به زمان خطی کاهش می‌دهد. برای توضیح ویژگی زیر مسئله‌های متدخل با جزئیات بیشتر اجازه دهید مسئله ضرب زنجیره‌ای ماتریس‌ها را دوباره بررسی کنیم.

شکل ۱۵.۵ درخت بازگشت برای محاسبه *RECURSIVE-MATRIX-CHAIN(p,1,4)* هر گره شامل پارامترهای i و jاست. محاسباتی که در زیردرخت سایه زده انجام شده با جدولی که در *MEMOIZED-MATRIX-CHAIN(p,1,4)* به آن مراجعه می‌شود جایگزین می‌شود.

با مراجعه مجدد به شکل ۱۵.۵ می‌بینیم که *MATRIX-CHAIN-ORDER* هنگام حل زیر مسئله‌ها در سطوح بالاتر، به طور مکرر به جواب زیر مسئله‌ها در سطوح پایین‌تر مراجعه می‌کند. برای مثال به ورودی $m[3,4]$ ، 4 بار مراجعه می‌شود: هنگام $m[2,4]$ ، $m[1,4]$ ، $m[3,5]$ و $m[3,6]$ اگر تنها به جای

---

۱ - ممکن است عجیب به نظر آید که برنامه سازی پویا متکی بر زیر مسئله‌هایی باشد که هم مستقل و هم همپوشان‌اند. گرچه ممکن است این الزامات متناقض به نظر آیند، دو نظر متفاوت را به جای دو نقطه نظر هم محور توصیف می‌کنند. اگر دو زیر مسئله یک مسئله در منابع مشتکر نباشد، مستقل هستند. اگر دو زیر مسئله واقعاً یک زیر مسئله باشند که به عنوان زیر مسائل مختلف اتفاق بیفتند، با یکدیگر تدخل دارند.

2. *overlapping subproblems*

مراجعه به $m[3,4]$ هر بار محاسبه می‌شد، افزایش چشمگیری در زمان اجرا بوجود می‌آمد. برای درک این مطلب، روال بازگشتی زیر (غیر کار آمد) را در نظر بگیرید که $m[i,j]$ یعنی مینیمم تعداد ضربهای مورد نیاز برای محاسبه حاصلضرب زنجیره‌ای ماتریس‌های $A_{i,j} = A_i A_{i+1} \ldots A_j$ را مشخص می‌کند. روال دقیقاً بر پایه معادله بازگشتی (۱۵.۱۲) بنا شده است.

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i = j
2     then return 0
3  m[i, j] ← ∞
4  for k ← i to j − 1
5       do q ← RECURSIVE-MATRIX-CHAIN(p, i, k)
                + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
                + p_{i−1} p_k p_j
6          if q < m[i, j]
7             then m[i, j] ← q
8  return m[i, j]
```

شکل ۱۵.۵ نشان می‌دهد که درخت بازگشت، با فراخوانی $RECURSIVE\text{-}MATRIX\text{-}CHAIN(p,1,4)$ تولید می‌شود. هر گره با مقادیر پارامترهای i و j برچسب گذاری شده است. می‌بینید که بعضی از جفت مقدارها چندین بار اتفاق می‌افتند.

در حقیقت، می‌توانیم نشان دهیم که زمان محاسبه $m[1,n]$ توسط این روال بازگشتی، حداقل بصورت نمایی در n است. فرض کنید $T(n)$ زمان صرف شده توسط $RECURSIVE\text{-}MATRIX\text{-}CHAIN$ برای محاسبه پرانتزگذاری بهینه یک زنجیره از n ماتریس را نشان می‌دهد. اگر فرض کنیم که اجرای خطوط ۲-۱ و اجرای خطوط ۷-۶، هر یک حداقل یک واحد زمانی طول می‌کشد، آنگاه بررسی روال، رابطه بازگشتی زیر را نتیجه می‌دهد:

$$T(1) \geq 1 ,$$
$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{for } n > 1 .$$

با توجه به اینکه برای $i = 1, 2, \ldots, n-1$ هر عبارت $T(i)$، یکبار به صورت $T(k)$ و یکبار به صورت $T(n-k)$ ظاهر می‌شود و با جمع کردن $n-1$ عدد 1 با هم و با عدد 1 خارج از سیگما، می‌توانیم رابطه بازگشتی را به صورت زیر بازنویسی کنیم.

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n . \qquad (۱۵.۱۳)$$

با استفاده از روش جایگذاری ثابت خواهیم کرد که $T(n) = \Omega(2^n)$ است. به خصوص نشان

می‌دهیم که برای همه $n \geq 1$ داریم $T(n) \geq 2^{n-1}$. حالت پایه آسان است چون $T(1) \geq 1 = 2^0$ با استقراء برای $n \geq 2$ داریم:

$$\begin{aligned} T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^{i} + n \\ &= 2(2^{n-1} - 1) + n \\ &= (2^{n} - 2) + n \\ &\geq 2^{n-1}\,. \end{aligned}$$

که اثبات را کامل می‌کند. بنابراین تعداد کل کاری که با فراخوانی *RECURSIVE-MATRIX-CHAIN* انجام می‌شود، حداقل به صورت نمایی در n است. این الگوریتم بازگشتی بالا به پایین را با الگوریتم پایین به بالای برنامه سازی پویا، مقایسه کنید. دومین الگوریتم کارآمدتر است، چون از ویژگی زیر مسئله‌های متداخل بهره می‌برد. فقط $\Theta(n^2)$ زیر مسئله مختلف وجود دارد که الگوریتم برنامه سازی پویا هر یک را دقیقاً یکبار حل می‌کند. از طرف دیگر، الگوریتم بازگشتی باید هر زیر مسئله را هر بار که در درخت بازگشت ظاهر شد، دوباره حل کند. هر زمان که برای جواب بازگشتی طبیعی مسئله، درخت بازگشت شامل زیر مسئله‌های تکراری باشد و تعداد کل زیر مسئله‌های متفاوت کم باشد، یک ایده خوب آن است که ببنیم آیا برنامه سازی پویا می‌تواند انجام شود یا خیر.

بازسازی جواب بهینه

به عنوان یک مورد عملی، اغلب انتخابی که در هر زیر مسئله انجام می‌دهیم را در یک جدول ذخیره می‌کنیم تا مجبور نباشیم این اطلاعات را از هزینه‌هایی که ذخیره کرده‌ایم دوباره بسازیم. زمان بندی خطوط مونتاژ، در سریعترین مسیر تا $S_{i,j}$ که ایستگاه قبل از $S_{i,j}$ را در $l_i[j]$ ذخیره می‌کنیم. متناوباً با پر کردن کل جدول $f_i[j]$ می‌توانیم با کمی کار اضافی، ایستگاه قبل از $S_{i,j}$ را در سریعتر مسیر تا $S_{i,j}$ مشخص کنیم. اگر $f_1[j] = f_1[j-1] + a_{1,j}$ باشد آنگاه در سریعترین مسیر تا $S_{i,j}$ که ایستگاه $S_{1,j-1}$ قبل از $S_{1,j}$ قرار می‌گیرد. در غیر اینصورت باید حالتی باشد که $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$ و بنابراین $S_{2,j-1}$ قبل از $S_{1,j}$ واقع می‌شود. برای زمان بندی خطوط مونتاژ، ساخت مجدد ایستگاههای ماقبل حتی بدون جدول $l_i[j]$ برای هر ایستگاه فقط زمان $O(1)$ را صرف می‌کند.

اما برای ضرب زنجیره‌ای ماتریس‌ها، هنگام ساخت مجدد جواب بهینه، جدول $S[i,j]$ مقدار قابل توجهی از کار مان را کاهش می‌دهد. فرض کنید جدول $S[i,j]$ را نگه داشته‌ایم تنها جدول $m[i,j]$ که شامل هزینه زیر مسئله‌های بهینه است را پر کرده‌ایم. برای مشخص کردن زیر مسئله‌هایی که در

جواب بهینه پرانتزگذاری $A_i A_{i+1} \ldots A_j$ استفاده می‌شوند، $j - i$ انتخاب وجود دارد و $j - i$ یک عدد ثابت نیست. بنابراین برای تولید زیر مسئله‌هایی که برای جواب مسئله داده شده انتخاب کرده‌ایم، زمان $\Theta(j\text{-}i) = \omega(1)$ صرف می‌شود. می‌توانیم با ذخیره اندیس ماتریس در $S[i,j]$ که در آن اندیس، ضرب $A_i A_{i+1} \ldots A_j$ را می‌شکنیم، هر انتخاب را در زمان $O(1)$ بازسازی کنیم.

Memoization

یک نوع برنامه سازی پویا وجود دارد که معمولاً هنگامی که تدبیر بالا به پایین را حفظ‌کنیم، قابلیت روش برنامه سازی پویای معمولی را بر روز می‌دهد. این ایده، *memoize* کردن یک الگوریتم ذاتاً بازگشتی ولی غیر کار آمد است. همچون برنامه سازی پویای معمولی، جدولی با جواب‌های زیر مسئله‌ها نگه می‌داریم ولی ساختار کنترلی پر کردن جدول بسیار شبیه الگوریتم بازگشتی است.

الگوریتم بازگشتی *memoize* شده، یک ورودی را برای حل هر زیر مسئله در جدول نگه می‌دارد. هر ورودی جدول، در ابتدا شامل یک مقدار خاص می‌باشد که نشان می‌دهد ورودی هنوز باید پر شود. در طول اجرای الگوریتم بازگشتی، هنگامی که برای اولین بار با یک زیر مسئله مواجه می‌شویم. جواب آن محاسبه شده و سپس در جدول ذخیره می‌شود. هر زمان دیگری که به زیر مسئله مراجعه شود، به راحتی مقدار ذخیره شده در جدول مشاهده و برگردانده می‌شود.[1]

در اینجا صورت *memoize* شده روال *RECURSIVE-MATRIX-CHAIN* آمده است:

```
MEMOIZED-MATRIX-CHAIN(p)
1  n ← length[p] − 1
2  for i ← 1 to n
3      do for j ← i to n
4             do m[i, j] ← ∞
5  return LOOKUP-CHAIN(p, 1, n)
```

```
LOOKUP-CHAIN(p, i, j)
1  if m[i, j] < ∞
2     then return m[i, j]
3  if i = j
4     then m[i, j] ← 0
5     else for k ← i to j − 1
6              do q ← LOOKUP-CHAIN(p, i, k)
                        + LOOKUP-CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
7                 if q < m[i, j]
8                    then m[i, j] ← q
9  return m[i, j]
```

---

۱ - این روش از قبل فرض می‌کند که مجموعه همه پارامترهای زیر مسئله‌های ممکن، شناخته شده است بین مکانهای جدول و زیر مسئله‌ها، رابطه بر قرار است. روش دیگر *memoize* کردن با استفاده از درهم سازی پارامتر زیر مسئله‌ها به عنوان کلیدها می‌باشد.

MEMOIZED-MATRIX-CHAIN مانند *MATRIX-CHAIN-ORDER*، جدول *m[1..n,1..n]* را نگه می‌دارد که شامل مقادیر محاسبه شده *m[i,j]*، یعنی مینیمم تعداد ضربهای مورد نیاز برای محاسبه ماتریس $A_{i..j}$ است. هر ورودی جدول در ابتدا دارای مقدار ∞ است که نشان می‌دهد ورودی هنوز باید پر شود. وقتی فراخوانی *LOOKUP-CHAIN(p,i,j)* انجام می‌شود. اگر در خط ۱، $m[i,j] < \infty$ باشد، روال به راحتی مقدار *m[i,j]* که قبلاً محاسبه شده است را بر می‌گرداند (خط ۲) در غیر اینصورت این مقدار همانند *RECURSIVE-MATRIX-CHAIN* محاسبه شده، در *m[i,j]* ذخیره شده و برگردانده می‌شود. (استفاده از مقدار ∞ به عنوان ورودی پر نشده جدول مناسب است چون مقداری است که در خط ۳، *RECURSIVE-MATRIX-CHAIN*، برای مقدار دهی اولیه *m[i,j]* مورد استفاده قرار می‌گیرد). بنابراین *LOOKUP-CHAIN* همیشه مقدار *m[i,j]* را بر می‌گرداند، ولی تنها وقتی آن را محاسبه می‌کند که *LOOKUP-CHAIN* برای اولین بار با پارامترهای *i* و *j* فراخوانی شده باشد.

شکل ۱۵.۵ نشان می‌دهد چطور *MEMOIZED-MATRIX-CHAIN* در مقایسه با *RECURSIVE-MATRIX-CHAIN* در زمان صرفه‌جویی می‌کند زیردرختهای سایه زده شده مقادیری را نشان می‌دهند که به جای محاسبه، جستجو شده‌اند.

همانند الگوریتم برنامه سازی پویای *MATRIX-CHAIN-ORDER*، روال *MEMOIZED-MATRIX-CHAIN* در زمان $O(n^3)$ اجرا می‌شود. هر یک از $\Theta(n^2)$ ورودی جدول، تنها یکبار در خط ۴، *MEMOIZED-MATRIX-CHAIN* مقداردهی می‌شود. می‌توانیم فراخوانی‌های *LOOK UP-CHAIN* را به دو دسته تقسیم کنیم:

۱. فراخوانی‌هایی که در آنها $m[i,j] = \infty$ است. بنابراین خطوط ۹-۳ اجرا می‌شوند و

۲. فراخوانی‌هایی که در آنها $m[i,j] < \infty$ است. بنابراین *LOOKUP-CHAIN* به سادگی در خط ۲ آن را بر می‌گرداند.

$\Theta(n)$ فراخوانی از نوع اول وجود دارد، یک فراخوانی برای هر ورودی جدول، همه فراخوانی‌های نوع دوم بصورت فراخوانی‌های بازگشتی توسط فراخوانی‌های نوع اول انجام می‌شوند. هر زمان که یک فراخوانی مورد نظر *LOOKUP-CHAIN* فراخوانی‌های بازگشتی انجام دهد، $O(n)$ تا از آنها فراخوانی می‌کند. بنابراین، در کل $O(n^3)$ فراخوانی از نوع دوم وجود دارد. هر فراخوانی نوع دوم زمان $O(1)$ را صرف می‌کند و هر فراخوانی نوع اول زمان $O(n)$ به اضافه زمان سپری شده در فراخوانی‌های بازگشتی‌اش را صرف می‌کند. از اینرو زمان کل $O(n^3)$ است. بنابراین *Memoization* الگوریتمی با زمان $\Omega(2^n)$ را به الگوریتمی با زمان $O(n^3)$ تبدیل می‌کند.

به طور خلاصه، مسئله ضرب زنجیره‌ای ماتریس‌ها می‌تواند در زمان $O(n^3)$ توسط یکی از الگوریتم‌های بالا به پائین *memoize* شده یا پایین به بالای برنامه سازی پویا حل شود. هر دو روش از

ویژگی زیر مسئله‌های متداخل استفاده می‌کنند. در کل فقط $\Theta(n^2)$ زیر مسئله متفاوت وجود دارد و هر یک از روشها تنها یکبار جواب هر زیر مسئله را محاسبه می‌کند. بدون *memoization* الگوریتم ذاتاً بازگشتی در زمان نمایی اجرا می‌شود، چون زیر مسئله‌های حل شده، مکرراً حل می‌شوند. به طور کلی در عمل اگر زیر مسئله‌ها باید حداقل یکبار حل شوند، الگوریتم برنامه سازی پویای پایین به بالا، معمولاً با یک ضریب ثابت، الگوریتم *memoize* شده بالا به پایین را اجرا می‌کند، چون هیچ محدودیتی برای بازگشت و محدودیت کمتری برای حفظ جدول وجود ندارد. بعلاوه مسئله‌هایی وجود دارند که برای آنها یک الگوی منظم از دستیابی‌های جدول در الگوریتم برنامه سازی پویا می‌تواند حتی برای کاهش بیشتر فضا و زمان مورد نیاز، استفاده شود. متناوباً اگر بعضی از زیر مسئله‌ها در فضای زیر مسئله به هیچ وجه نیاز به حل شدن نداشته باشند، راه حل *memoize* شده تنها از حل زیر مسئله‌هایی که قطعی مورد نیاز هستند بهره می‌برد.

تمرین‌ها

۱۵.۳-۱ کدامیک از راههای زیر برای مشخص کردن تعداد بهینه ضرب‌ها در مسئله ضرب زنجیره‌ای ماتریس‌ها کارآمدتر است: برشمردن همه راههای پرانتزگذاری ضرب و محاسبه تعداد ضربها برای هر یک، یا اجرای *RECURSIVE-MATRIX-CHAIN*؟ جواب خود را توجیه کنید.

۱۵.۳-۲ درخت بازگشت روال *MERGE-SORT* از بخش ۲.۳.۱ را روی یک آرایه *16* عنصری رسم کنید. توضیح دهید چرا *memoization* برای تسریع یک الگوریتم تقسیم و حل مانند *MERGE-SORT* موثر نیست.

۱۵.۳-۳ یک مسئله متفاوت ضرب زنجیره‌ای ماتریسها را در نظر بگیرید که در آن هدف، پرانتزگذاری توالی ماتریس‌ها به منظور ماکزیمم کردن تعداد ضربها به جای مینیمم کردن آنها باشد. آیا این مسئله زیر ساختار بهینه ارائه می‌کند؟

۱۵.۳-۴ توضیح دهید چرا زمان بندی خطوط مونتاژ دارای زیر مسئله‌های متداخل است.

۱۵.۳-۵ همان طور که ذکر شد، در برنامه سازی پویا، ابتدا زیر مسئله‌ها را حل کرده و سپس انتخاب می‌کنیم کدامیک از آنها در حل بهینه مسئله استفاده شود. پرفسور *Capulet* ادعا می‌کند که همیشه لازم نیست برای پیدا کردن جواب بهینه، همه زیر مسئله‌ها را حل کنیم. او اظهار می‌دارد که همیشه با انتخاب ماتریس A_k که در آن زیر حاصلضرب $A_i A_{i+1} \dots A_j$ شکسته می‌شود (با انتخاب k برای مینیمم کردن کمیت $p_{i-1} p_k p_j$) قبل از حل زیر مسئله‌ها، جواب بهینه مسئله ضرب زنجیره‌ای ماتریس‌ها می‌تواند پیدا شود. نمونه‌ای برای مسئله ضرب زنجیره‌ای ماتریس‌ها پیدا کنید که برای آن این روش حریصانه، یک جواب زیر بهینه تولید کند.

۱۵.۴ طولانی‌ترین زیر رشته مشترک

در کاربردهای زیست شناسی، اغلب می‌خواهیم DNAی دو (یا چند) موجوده زنده مختلف را مقایسه کنیم. یک رشته DNA شامل رشته‌ای از مولکولها است که به آن پایه‌ها[1] می‌گویند که پایه‌های ممکن عبارتند از *adenine* , *guanine* , *cytosine* و *thymine*، با نمایش هر یک از پایه‌ها توسط حروف ابتدائی آنها، یک رشته DNA می‌تواند به عنوان رشته‌ای از مجموعه محدود $\{A , C , G , T\}$ تعریف شود. برای مثال ممکن است DNA یک موجود زنده $S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA$ و DNA موجود زنده دیگری $S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA$ باشد. یکی از هدفهای مقایسه دو رشته DNA این است که مشخص کنیم دو رشته چقدر "مشابهند" تا میزان تناسب بین دو موجود زنده تعیین شود. تشابه می‌تواند وجود داشته باشد و به روشهای متفاوتی تعریف می‌شود. برای مثال اگر یکی از دو رشته DNA زیر رشته دیگری باشد می‌توانیم بگوییم دو رشته DNA مشابهند. در مثال ما، هیچ یک از رشته‌های S_1 و S_2 زیر رشته دیگری نمی‌باشند. متناوباً اگر تعداد تغییرات لازم برای تبدیل یک رشته به رشته دیگر کم باشد، می‌توانیم بگوییم دو رشته مشابهند. (مسئله ۲-۱۵ به این ایده اشاره می‌کند.) اما روشی دیگر برای اندازه گیری تشابه دو رشته S_1 و S_2 که تعریف رشته سوم و S_3 است که پایه‌های آن در هر دو رشته S_1 و S_2 وجود داشته باشند. این پایه‌ها باید در ترتیب یکسانی ظاهر شوند اما نه الزاماً به صورت متوالی، طولانی‌تر بودن رشته S_3ای که می‌توانیم پیدا کنیم، تشابه بیشتر S_1 و S_2 را نشان می‌دهد. در مثال بالا، طولانی‌ترین رشته S_3، $GTCGTCGGAAGCCGGCCGAA$ است. این ایده آخر تشابه را به عنوان مسئله طولانی‌ترین زیر رشته مشترک مطرح می‌کنیم.

زیر رشته یک رشته داده شده دقیقاً همان رشته داده شده است که صفر یا چند عنصر از آن حذف شده است. به طور رسمی، در رشته $X = <x_1, x_2 , \dots x_m>$، رشته $Z = <z_1 , z_2 , \dots ,z_k>$ **زیر رشته‌ای**[2] از X است، اگر توالی اکیداً صعودی $<i_1, i_2, \dots , i_k>$ از اندیس‌های X وجود داشته باشد بطوریکه برای همه $j=1,2,\dots,k$ داشته باشیم $x_{i_j} = z_j$ برای مثال، $Z = <B,C,D,B>$ بنا به توالی اندیس $<2,3, 5,7>$، زیر رشته $X = <A,B,C,B,D,A,B>$ است.

دو رشته X و Y داده شده‌اند، می‌گوییم Z زیر رشته مشترک X و Y است، اگر Z هم زیر رشته X و هم زیر رشته Y باشد. برای مثال اگر $X = <A,B,C,B,D,A,B>$ و $Y = <B,D,C,A,B,A>$ باشد: رشته $<B,C,A>$ زیر رشته مشترکی از X و Y است. رشته $<B,C,A>$ بلندترین زیر رشته مشترک *(LCS)* رشته‌های X و Y نیست، چون طول آن 3 است و طول رشته $<B,C,B,A>$ که آنهم بین X و Y مشترک است 4 است. رشته $<B,C,B,A>$ همانند $<B,D,A,B>$ یک LCS برای X و Y است. چون هیچ زیر رشته

---

1. bases 2. subsequence

مشترکی با طول 5 یا بیشتر وجود ندارد.

در *مسئله طولانی‌ترین زیر رشته مشترک*، دو رشته $X = <x_1,x_2, ... , x_m>$ و $Y = <y_1, y_2, ... , y_n>$ داده شده‌اند و می‌خواهیم زیر رشته مشترکی از X و Y با طول ماکزیمم را پیدا کنیم. این بخش نشان می‌دهد که مسئله LCS می‌تواند به طور موثری با برنامه سازی پویا حل شود.

مرحله ۱: مشخص کردن طولانی ترین زیر رشته مشترک

یک روش قدرتمندانه برای حل مسئله LCS این است که همه زیر رشته‌های X را بر شمرده و هر زیر رشته را بررسی کنیم تا ببنیم آیا زیر رشته Y نیز هست یا نه و به دنبال طولانی‌ترین زیر رشته پیدا شده باشیم. هر زیر رشته X با زیر مجموعه‌ای از اندیس‌های $\{1, 2, ... , m\}$ از X متناظر است. 2^m زیر رشته از X وجود دارد بنابراین این روش نیاز به زمان نمایی دارد که آن را برای رشته‌های طولانی غیر عملی می‌سازد. همانطور که قضیه زیر نشان می‌دهد، مسئله LCS دارای خاصیت زیر ساختار بهینه است. همان طور که خواهیم دید، کلاسهای طبیعی زیر مسئله‌ها با جفت پیشوندی دو رشته ورودی متناظر است.

جهت دقت بیشتر، برای رشته داده شده $X = <x_1, x_2, ... , x_n>$، i امین **پیشوند** x را برای $i = 0, 1, ... , m$ به صورت $X_i = <x_1, x_2, ... , x_i>$ تعریف می‌کنیم. برای مثال اگر $X = <A,B,C,B,D,A,B>$ باشد. $X_4 = <A,B,C,B>$ و X_0 رشته تهی است.

قضیه ۱۵.۱ (زیر ساختار بهینه یک LCS)

فرض کنید $X = <x_1,x_2, ... , x_m>$ و $Y = <y_1, y_2, ... , y_n>$ دو رشته هستند و $Z = <z_1, z_2, ... , z_k>$ یک LCS از X , Y است.

۱. اگر $x_m = y_n$ باشد آنگاه $z_k = x_m = y_n$ و Z_{k-1} یک LCS از X_{m-1} و Y_{n-1} است.

۲. اگر $x_m \neq y_n$ باشد آنگاه $z_k \neq x_m$ نشان می‌دهد که Z یک LCS از X_{m-1} و Y است.

۳. اگر $x_m \neq y_n$ باشد آنگاه $z_k \neq y_n$ نشان می‌دهد که Z یک LCS از X و Y_{n-1} است.

اثبات (۱) اگر $z_k \neq x_m$ باشد آنگاه می‌توانیم $x_m = y_n$ را به Z اضافه کنیم تا به زیر رشته مشترکی از X و Y با طول $k + 1$ برسیم که با فرض اینکه Z طولانی‌ترین زیر رشته مشترک X و Y است تناقض دارد. بنابراین باید داشته باشیم $z_k = x_m = y_n$

اکنون پیشوند Z_{k-1} زیر رشته مشترک X_{m-1} و Y_{n-1} با طول $(k - 1)$ است. می‌خواهیم نشان دهیم که این زیر رشته LCS است. برای ایجاد تناقض، فرض کنید که زیر رشته مشترک W برای رشته‌های X_{m-1} و Y_{n-1} با طول بیشتر از $k - 1$ وجود دارد. سپس عمل اضافه کردن $x_m = y_n$ به W، زیر رشته مشترک X و Y را تولید می‌کند که طول آن بیشتر از k است که این یک تناقض است.

(۲) اگر $z_k \neq x_m$ باشد آنگاه Z زیر رشته مشترکی از Y و X_{m-1} است. اگر زیر رشته مشترک W برای رشته‌های X_{m-1} و Y با طول بیشتر از k وجود داشته باشد آنگاه W زیر رشته مشترک X_m و Y نیز هست، که با فرض اینکه Z LCS مربوط به X و Y است تناقض دارد.

(۳) اثبات، متقارن با قسمت (۲) است.

ویژگی قضیه ۱۵.۱ نشان می‌دهد که LCS دو رشته، داخل خود شامل LCS پیشوندهای دو رشته است. بنابراین مسئله LCS دارای خاصیت زیر ساختار بهینه است. همان طور که خواهیم دید، حل بازگشتی نیز دارای ویژگی زیر مسئله‌ها متداخل است.

مرحله ۲: حل بازگشتی

قضیه ۱۵.۱ بیان می‌کند که هنگام پیدا کردن LCS مربوط به $X = <x_1, x_2, ..., x_m>$ و $Y = <y_1, y_2, ..., y_n>$ یک یا دو زیر مسئله برای بررسی وجود دارد. اگر $x_m = y_n$ باشد، باید LCS مربوط به X_{m-1} و Y_{n-1} را پیدا کنیم. اضافه کردن $x_m = y_n$ به این LCS، LCS مربوط به X و Y را حاصل می‌کند. اگر $x_m \neq y_n$ باشد، آنگاه باید دو زیر مسئله را حل کنیم: پیدا کردن LCS مربوط به X_{m-1} و Y و پیدا کردن LCS مربوط به X و Y_{n-1} هر کدام از این LCSها که طولانی‌تر باشد، LCS مربوط به X و Y است. چون این حالتها، همه حالتهای ممکن را در بر می‌گیرند، می‌دانیم که یکی از راه حل‌های بهینه زیر مسئله‌ها، باید در LCS مربوط به X و Y مورد استفاده قرار گیرند.

به سادگی ویژگی زیر مسئله‌های متداخل را در مسئله LCS مشاهده می‌کنیم. برای پیدا کردن LCS مربوط به X و Y، ممکن است نیاز داشته باشیم که LCS مربوط به X و Y_{n-1} و LCS مربوط به X_{m-1} و Y را پیدا کنیم. اما هر یک از این زیر مسئله‌ها شامل زیر مسئله یافتن LCS مربوط به X_{m-1} و Y_{n-1} می‌باشند. بسیاری از زیر مسئله‌های دیگر در این زیر مسئله مشترکند.

همانند مسئله ضرب زنجیره‌ای ماتریس‌ها، حل بازگشتی مسئله LCS شامل برقراری یک رابطه بازگشتی برای مقدار جواب بهینه است. $c[i,j]$ را طول LCS مربوط به رشته‌های X_i و Y_j تعریف می‌کنیم. اگر $i = 0$ یا $j = 0$ باشد، طول یکی از رشته‌ها صفر است، بنابراین طول LCS صفر است. زیر ساختار بهینه مسئله LCS فرمول بازگشتی زیر را حاصل می‌کند.

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases} \qquad (۱۵.۱۴)$$

می‌بینید که در این فرمول بازگشتی یک شرط در مسئله محدود می‌کند که کدام زیر مسئله ها را ممکن است در نظر بگیریم. زمانی که $x_i = y_j$ باشد می‌توانیم و باید زیر مسئله پیدا کردن LCS مربوط به X_{i-1} و Y_{j-1} را در نظر بگیریم. در غیر اینصورت دو زیر مسئله پیدا کردن LCS مربوط به X_i و

Y_{j-1} و LCS مربوط به X_{i-1} و Y_j را در نظر می‌گیریم. در الگوریتم‌های قبلی برنامه سازی پویا - برای زمان‌بندی خط مونتاژ و ضرب زنجیره‌ای ماتریس‌ها - هیچ زیر مسئله‌ای با توجه به شرایط مسئله رد نمی‌شد. پیدا کردن LCS تنها الگوریتم برنامه سازی پویا نیست که زیر مسئله‌ها را بر اساس شرایط مسئله رد می‌کند. برای مثال، مسئله ویرایش مسافت (مسئله ۳-۱۵ را ملاحظه نمایید.) دارای این ویژگی است.

مرحله ۳: محاسبه طول LCS

بنا به معادله (۱۵.۱۴) می‌توانیم به راحتی یک الگوریتم بازگشتی با زمان نمایی برای محاسبه طول LCS دو رشته بنویسیم چون فقط $\Theta(mn)$ زیر مسئله متفاوت وجود دارد، بنابراین می‌توانیم برای محاسبه جوابها از پایین به بالا، از برنامه سازی پویا استفاده کنیم.

روال $LCS\text{-}LENGTH$ دو رشته $X = <x_1, x_2, \ldots, x_m>$ و $Y = <y_1, y_2, \ldots, y_n>$ را به عنوان ورودی می‌گیرد. مقادیر $c[i,j]$ را در جدول $c[0..m, 0..n]$ که ورودیهای آن به ترتیب سطری محاسبه می‌شوند ذخیره می‌کند. (به عبارت دیگر، اولین سطر c از چپ به راست پر می‌شود و سپس دومین سطر و به همین ترتیب.) همچنین جدول $b[1..m, 1..n]$ را برای ساده سازی ساخت جواب بهینه نگه می‌دارد. به طور شهودی، $b[i,j]$ به ورودی از جدول اشاره می‌کند که با جواب بهینه زیر مسئله‌ای که هنگام محاسبه $c[i,j]$ انتخاب شده است، متناظر می‌باشد. این روال، جدولهای c و b را بر می‌گرداند؛ $c[m,n]$ دارای طول LCS مربوط به X و Y است.

```
LCS-LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4       do c[i, 0] ← 0
 5  for j ← 0 to n
 6       do c[0, j] ← 0
 7  for i ← 1 to m
 8       do for j ← 1 to n
 9              do if x_i = y_j
10                    then c[i, j] ← c[i − 1, j − 1] + 1
11                         b[i, j] ← "↖"
12                    else if c[i − 1, j] ≥ c[i, j − 1]
13                            then c[i, j] ← c[i − 1, j]
14                                 b[i, j] ← "↑"
15                            else c[i, j] ← c[i, j − 1]
16                                 b[i, j] ← "←"
17  return c and b
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

شکل ۱۵.۶ جدولهای c و b که توسط *LCS-LENGTH* روی رشته‌های $X= <A,B,C,B,D,A,B>$ و $Y = <B,D,C,A,B,A>$ محاسبه شده‌اند. مربع واقع در سطر i و ستون j دارای مقدار $c[i,j]$ و پیکان مناسب بـرای مـقدار $b[i,j]$ است. ورودی 4 در $c[7,6]$ - گـوشه پـایین و سـمت راست جـدول - طـول LCS مربوط به X و Y است. برای $i,j > 0$ ورودی $c[i,j]$ تنها به اینکه آیا $x_i = y_j$ است یا نه و مقدار ورودیهای $c[i-1,j]$، $c[i,j-1]$ و $c[i-1,j-1]$ که قبل از $c[i,j]$ محاسبه شده‌اند، بستگی دارد. برای بازسازی عناصر LCS، پیکانهای $b[i,j]$ را از گوشه پایین سمت راست دنبال کنید؛ مسیر سایه زده شده است. هر "↖" در مسیر، متناظر با ورودیهای (سایه روشن خورده) است که برای آن $x_i = y_j$ عضوی از LCS است.

شکل ۱۵.۶ جـدول‌هایی کـه تـوسط *LCS-LENGTH* روی رشـته‌های $X= <x_1, x_2, ..., x_m>$ و $Y = <y_1, y_2, ..., y_n>$ ایجاد شده‌اند را نشان می‌دهد. زمان اجرای ایـن روال، $O(mn)$ است زیـرا هر ورودی جدول برای محاسبه زمان $O(1)$ را صرف می‌کند.

مرحله ۴: ساخت LCS

جدول b که توسط *LCS-LENGTH* برگردانده می‌شود، می‌تواند برای ساخت سریع LCS مربوط به $X= <x_1, x_2, ..., x_m>$ و $Y = <y_1, y_2, ..., y_n>$ استفاده شود. به سادگی $b[m,n]$ شروع کرده و با دنبال کردن پیکانها در جدول ادامه می‌دهد. هر زمان که در ورودی $b[i,j]$ با "↖" مواجه می‌شویم، بیان می‌کند که $x_i = y_i$ یک عنصر LCS است. به این روش عناصر LCS با ترتیب معکوس ملاقات می‌شوند. روال بازگشتی زیر، LCS مربوط به X و Y را به یک ترتیب مستقیم مناسب چاپ می‌کند. فراخـوانـی ابتدایی *PRINT-LCS(b,X,length[X] , length[Y])* است.

```
PRINT-LCS(b, X, i, j)
1  if i = 0 or j = 0
2      then return
3  if b[i, j] = "↖"
4      then PRINT-LCS(b, X, i − 1, j − 1)
5           print x_i
6  elseif b[i, j] = "↑"
7      then PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

این روال برای جدول b در شکل ۱۵.۶، "$B\ C\ B\ A$" را چاپ می‌کند. روال زمان $O(m + n)$ را صرف می‌کند چون حداقل یک واحد از i یا j در هر مرحله بازگشت کم می‌شود.

بهبود بخشیدن کد برنامه

زمانی که یک الگوریتم را تولید کرده‌اید، اغلب در می‌یابید که می‌توانید زمان یا فضایی کـه اسـتفاده می‌کند را بهبود ببخشید.

این مطلب مخصوصاً در مورد الگوریتم‌های برنامه سازی پویا کاملاً درست است. بعضی تغییرات می‌توانند کد برنامه را ساده کنند و ضرایب ثابت را بهبود بخشند، اما با این وجود هیچ بهبود مجانبی در اجرا حاصل نمی‌شود. سایر تغییرات ممکن است منجر به ذخیره مجانبی قابل توجهی در زمان و فضا شوند. برای مثال می‌توانیم جدول b را به کلی حذف کنیم.

هر ورودی $c[i,j]$ تنها به سه ورودی دیگر جدول c بستگی دارد: $c[i - 1 , j - 1]$ و $c[i - 1,j]$ و $c[i , j - 1]$ با دریافت مقدار $c[i,j]$ می‌توانیم بدون بررسی جدول b در زمان $O(1)$ مشخص کنیم که کدامیک از این سه مقدار برای محاسبه $c[i,j]$ استفاده شده‌اند. بنابراین می‌توانیم با استفاده از روالی مشـابه $PRINT\text{-}LCS$، LCS را در زمان $O(m + n)$ بازسازی کنیم. (تمرین ۲-۱۵.۴ از شما می‌خواهد که شبه کد آن را ارائه دهید.) گرچه با این روش $\Theta(mn)$ فضا صرفه جویی می‌شود، فضای کمکی مورد نیاز برای محاسبه LCS به صورت مجانبی کاهش نمی‌یابد، چون بهر حال $\Theta(mn)$ فضا برای جدول c لازم داریم.

اما می‌توانیم فضای مجانبی مورد نیاز برای $LCS\text{-}LENGTH$ را کاهش دهیم، زیرا در هر زمان تنها به دو سطر از جدول c نیاز دارد: سطری که محاسبه می‌شود و سطر قبلی. (در حقیقت می‌توانیم فقط کمی بیشتر از فضای یک سطر از c را برای محاسبه طول LCS استفاده کنیم. تمرین ۴-۱۵.۴ را ملاحظه نمایید.) اگر فقط طول LCS را لازم داشته باشیم، این بهبود سازی کار می‌کند؛ اگر لازم باشد عناصر LCS را بازسازی کنیم، جدول کوچکتر، اطلاعات کافی برای ردیابی مراحل در زمان $O(m + n)$ را نگه نمی‌دارد.

تمرین‌ها

۱۵.۴-۱ LCS مربوط به $<1,0,0,1,0,1,0,1>$ و $<0,1,0,1,1,0,1,1,0>$ را تعیین کنید.

۱۵.۴-۲ نشان دهید چطور می‌توان یک LCS از روی جدول کامل شده c و رشته‌های اولیه $X= <x_1, x_2, ..., x_m>$ و $Y = <y_1, y_2, ..., y_n>$ را در زمان $O(m+n)$ و بدون استفاده از جدول b بازسازی نمود.

۱۵.۴-۳ صورت *memoize* شده *LCS-LENGTH* را طوری بنویسید که در زمان $O(mn)$ اجرا شود.

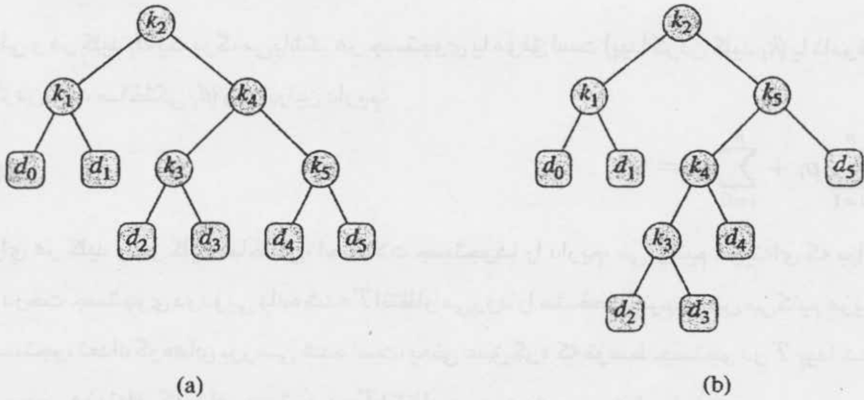
۱۵.۴-۴ نشان دهید چطور می‌توان طول یک LCS را تنها با استفاده از $2. min(m,n)$ ورودی در جدول c به اضافه $O(1)$ فضای اضافی محاسبه کرد. سپس نشان دهید چطور با استفاده از $min(m,n)$ ورودی به اضافه $O(1)$ فضای اضافی، این کار انجام می‌شود.

۱۵.۴-۵ الگوریتمی با زمان $O(n^2)$ ارائه دهید که طولانی‌ترین زیر رشته یکنواخت صعودی یک رشته n عددی را پیدا کند.

۱۵.۴-۶ الگوریتمی با زمان $O(nlgn)$ برای پیدا کردن طولانی‌ترین زیر رشته یکنواخت صعودی یک رشته n عددی، ارائه دهید. (راهنمایی: می‌بینید که آخرین عنصر زیر رشته کاندید شده با طول i حداقل به بزرگی آخرین عنصر زیر رشته کاندید شده با طول $i - 1$ است. زیر رشته‌هایی کاندید شده را با متصل کردن آنها به رشته ورودی، نگه دارید.)

۱۵.۵ درخت‌های جستجوی دودویی بهینه

فرض کنید برنامه‌ای را طراحی می‌کنیم که متن را از انگلیسی به فرانسوی ترجمه می‌کند. برای هر مورد کلمه انگلیسی در متن، لازم است معادل فرانسوی آن را جستجو کنیم. یک راه برای انجام این عملیات جستجو ساخت یک درخت جستجوی دودویی با n کلمه انگلیسی به عنوان کلیدها و معادلهای فرانسوی به عنوان داده‌های وابسته است. چون برای هر کلمه در متن، بصورت مجزا درخت را جستجو می‌کنیم، می‌خواهیم زمان کل صرف شده برای جستجو تا حد امکان کم باشد. با استفاده از درخت قرمز - سیاه یا هر درخت جستجوی دودویی موازنه شده دیگری، می‌توانیم برای هر جستجو، زمان $O(lgn)$ را تضمین نماییم. کلمات با تعداد تکرارهای متفاوتی ظاهر می‌شوند، و ممکن است حالتی رخ دهد که کلمه‌ای که زیاد استفاده می‌شود مانند *"the"* دور از ریشه باشد در حالیکه کلمه‌ای که بندرت استفاده می‌شود مانند *"mycophagist"* نزدیک ریشه باشد. چنین سازماندهی، سرعت ترجمه را کاهش می‌دهد، چون تعداد گره‌هایی که هنگام جستجوی یک کلید در درخت دودویی ملاقات می‌شوند، یکی بیشتر از عمق گره حاوی آن کلید است.



شکل ۱۵.۷ دو درخت جستجوی دودویی برای مجموعه‌ای از $n = 5$ کلید با احتمالات زیر؛

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(a) یک درخت جستجوی دودویی با هزینه جستجوی مورد انتظار *2.80*. *(b)* یک درخت جستجوی دودویی با هزینه جستجوی مورد انتظار *2.75*. این درخت بهینه است.

می‌خواهیم کلماتی که زیاد در متن بکار رفته‌اند نزدیکتر به ریشه قرار گیرند[۱]. بعلاوه ممکن است کلماتی در متن باشند که برای آنها معادل فرانسوی وجود نداشته باشد و چنین کلماتی ممکن است اصلاً در درخت جستجو دودویی نباشند. با دانستن اینکه هر کلمه چند وقت یکبار تکرار می‌شود، چطور می‌توان یک درخت جستجوی دودویی را طوری سازماندهی کرد که تعداد گره‌های ملاقات شده در همه جستجوها، مینیمم شوند؟

چیزی که نیاز داریم به عنوان درخت **جستجوی دودویی بهینه** شناخته می‌شود. به طور رسمی، رشته $K = <k_1, k_2, \dots, k_n>$ از n کلید مختلف به صورت مرتب شده (بطوریکه $k_1 < k_2 < \dots < k_n$) به ما داده می‌شود و می‌خواهیم با این کلیدها، درخت جستجوی دودویی را بسازیم. برای هر کلید k_i احتمال p_i را داریم که احتمال جستجوی کلید k_i خواهد بود. بعضی جستجوها ممکن است برای مقادیری باشد که در k نیستند و بنابراین $n + 1$ "کلید ساختگی" $d_0, d_1, d_2, \dots d_n$ نیز برای نشان دادن مقادیری که در k نیستند، داریم. خصوصاً، d_0 همه مقادیر کمتر از k_1 را نشان می‌دهد، d_n همه مقادیر بیشتر از k_n را نشان می‌دهد و برای $i = 1, 2, \dots, n - 1$ کلید ساختگی d_i همه مقادیر بین k_i و k_{i+1} را نشان می‌دهد. برای هر کلید ساختگی d_i احتمال q_i را داریم احتمال جستجوی متناظر با d_i است. شکل ۱۵.۷ دو درخت جستجوی دودویی برای مجموعه‌ای از $n = 5$ کلید را نشان می‌دهد. هر کلید k_i

---

۱ - اگر موضوع متن قارچهای خوارکی باشد، ممکن است بخواهیم "*mycophogist*" (قارچ شناس) نزدیک ریشه قرار گیرد.

یک گره داخلی و هر کلید d_i یک برگ می‌باشد. هر جستجوی یا موفق است (پیدا کردن کلید k_i) یا ناموفق است (پیدا کردن کلید ساختگی d_i) و بنابراین داریم:

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 . \qquad (۱۵.۱۵)$$

چون برای هر کلید و هر کلید ساختگی، احتمالات جستجوها را داریم، می‌توانیم هزینه‌ای که برای جستجو در درخت جستجوی دودویی داده شده T انتظار می‌رود را مشخص کنیم. فرض می‌کنیم هزینه واقعی هر جستجو، تعداد گره‌های بررسی شده است، یعنی عمق گره که توسط جستجو در T پیدا شده به اضافه 1. سپس هزینه‌ای که برای جستجو در T انتظار می‌رود به صورت زیر است

$$\begin{aligned} \text{E}[\text{search cost in } T] &= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i , \end{aligned} \qquad (۱۵.۱۶)$$

که $depth_T$ عمق گره در درخت T را مشخص می‌کند. معادله آخر از معادله (۱۵.۱۵) حاصل می‌شود. در شکل ۱۵.۷(a) می‌توانیم گره به گره، هزینه جستجویی که انتظار می‌رود را محاسبه کنیم:

گره	عمق	احتمال	سهم
k1	*1*	*0.15*	*0.30*
k2	*0*	*0.10*	*0.10*
k3	*2*	*0.05*	*0.15*
k4	*1*	*0.10.*	*0.20*
k5	*2*	*0.20*	*0.60*
d0	*2*	*0.05*	*0.15*
d1	*2*	*0.10*	*0.30*
d2	*3*	*0.05*	*0.20*
d3	*3*	*0.05*	*0.20*
d4	*3*	*0.05*	*0.20*
d5	*3*	*0.10*	*0.40*
مجموع			*2.80*

برای مجموعه‌ای از احتمالات داده شده، هدف ما ساخت یک درخت جستجوی دودویی است که هزینه جستجوی مورد انتظار آن کمترین باشد. به چنین درختی، **درخت جستجوی دودویی بهینه** می‌گوییم.

شکل ۱۵.۷ (b) یک درخت جستجوی دودویی بهینه را برای احتمالات داده شده در شرح شکل، نشان می‌دهد؛ هزینه مورد انتظار آن *2.75* است. این مثال نشان می‌دهد که لازم نیست درخت جستجوی دودویی بهینه، درختی باشد که ارتفاع کل آن کمترین است. و لازم نیست برای ساخت درخت

جستجوی دودویی بهینه، همیشه کلیدی که بزرگترین احتمال را دارد، در ریشه قرار دهیم. در اینجا، کلید k_5 بزرگترین احتمال جستجو را بین کلیدها دارد، ولی ریشه درخت جستجوی دودویی بهینه نشان داده شده، k_2 است. (کمترین هزینه‌ای که برای هر درخت جستجوی دودویی با ریشه k_5 انتظار می‌رود، 2.85 است.)

همانند ضرب زنجیره‌ای ماتریس‌ها، بررسی کامل همه حالت‌های ممکن، یک الگوریتم کار آمد را نتیجه نمی‌دهد. می‌توانیم برای ساخت درخت جستجوی دودویی گره‌های یک درخت جستجوی n گره‌ای را با کلیدهای $k_1, k_2, \ldots, k_n$ برچسب دهی کنیم و سپس کلیدهای ساختگی را به عنوان برگها به آن اضافه کنیم. در مسئله ۴-۱۲ دیدیم که تعداد درختهای دودویی با n گره، $\Omega(4^n / n^{3/2})$ است بنابراین در یک جستجوی کامل، تعدادی نمایی درخت جستجوی دودویی داریم که باید بررسی کنیم. جای تعجب نیست که این مسئله را با برنامه سازی پویا حل خواهیم کرد.

مرحله ۱: ساختار درخت جستجوی دودویی بهینه

برای مشخص کردن زیر ساختار بهینه درختهای جستجوی دودویی بهینه، با نگاهی به زیر درختها شروع می‌کنیم یک زیر درخت از درخت جستجوی دودویی را در نظر بگیرید. این درخت باید شامل کلیدهایی در بازه پیوسته $k_i, \ldots, k_j$ برای $1 \le i \le j \le n$ باشد. بعلاوه زیر درختی که شامل کلیدهای $k_i, \ldots, k_j$ است. باید کلیدهای ساختگی $d_{i-1}, \ldots, d_j$ را نیز به عنوان برگهایش داشته باشد.

اکنون می‌توانیم زیر ساختار بهینه را بیان کنیم: اگر درخت جستجوی دودویی بهینه T، یک زیر درخت T' شامل کلیدهای $k_i, \ldots, k_j$ داشته باشد، آنگاه این زیر درخت T' باید برای زیر مسئله‌ای با کلیدهای $k_i, \ldots, k_j$ و کلیدهای ساختگی $d_{i-1}, \ldots, d_j$ هم بهینه باشد. بحث معمول "برش و الصاق" بکار می‌رود. اگر زیر درخت T'' وجود می‌داشت که هزینه مورد انتظار برای آن کمتر از T' می‌بود، آنگاه می‌توانستیم T' را از T برش زده و بجای آن T'' را الصاق کنیم که یک درخت جستجوی دودویی با هزینه کمتر از T را نتیجه می‌دهد. بنابراین بهینه بودن T نقض می‌شود.

برای اینکه نشان دهیم می‌توانیم جواب بهینه مسئله را از جوابهای بهینه زیر مسئله‌ها بسازیم، نیاز داریم که از زیر ساختار بهینه استفاده کنیم. یکی از کلیدهای داده شده $k_i, \ldots, k_j$ به نام k_r $(i \le r \le j)$ ریشه زیر درخت بهینه‌ای خواهد بود که شامل این کلیدها می‌باشد. زیر درخت چپ ریشه k_r شامل کلیدهای $k_i, \ldots, k_{r-1}$ (و کلیدهای ساختگی $d_{i-1}, \ldots, d_{r-1}$) و زیر درخت راست شامل کلیدهای $k_{r+1}, \ldots, k_j$ (و کلیدهای ساختگی $d_r, \ldots, d_j$) خواهد بود، مادامیکه همه ریشه‌های کاندید شده k_r را که $i \le r \le j$ بررسی می‌کنیم و همه درختهای جستجوی دودویی بهینه شامل $k_i, \ldots, k_{r-1}$ و همه درختهای جستجوی دودویی بهینه شامل $k_{r+1}, \ldots, k_j$ را تعیین می‌کنیم، مطمئن می‌شویم که درخت جستجوی دودویی بهینه را پیدا خواهیم کرد.

مسئله جزئی مهم دیگری در مورد زیر درختهای "تهی" وجود دارد. فرض کنید در زیر درختی با کلیدهای k_i , ... k_j را به عنوان ریشه انتخاب می‌کنیم. با توجه به بحث فوق زیر درخت چپ k_i شامل کلیدهای k_i , ... k_{i-1} است. طبیعی است که این رشته را طوری تفسیر کنیم که شامل هیچ کلیدی نیست. اما توجه داشته باشید که زیر درختها شامل کلیدهای ساختگی نیز هستند. قرار دادی که بیان می‌کند زیر درختی که شامل کلیدهای k_i , ... k_{i-1} است، هیچ کلید حقیقی ندارد ولی شامل یک کلید ساختگی d_{i-1} است را می‌پذیریم.

به طور متقارن اگر k_j را به عنوان ریشه انتخاب کنیم، آنگاه زیر درخت راست k_j شامل کلیدهای k_{j+1}, .. , k_j است؛ این زیر درخت راست شامل هیچ کلید حقیقی نیست ولی شامل کلید ساختگی d_j می‌باشد.

مرحله ۲: حل بازگشتی

اکنون آماده‌ایم که مقدار جواب بهینه را به صورت بازگشتی تعریف کنیم. محدوده زیر مسئله خود را پیدا کردن درخت جستجوی دودویی شامل کلیدهای k_i, ..., k_j که $i \geq 1$ و $j \leq n$ و $j \geq i - 1$ است انتخاب می‌کنیم. (این زمانیست که $j = i - 1$ باشد و لذا هیچ کلید حقیقی وجود ندارد؛ تنها، کلید ساختگی d_{i-1} را داریم.) $e[i,j]$ را به عنوان هزینه‌ای که جستجوی دودویی بهینه‌ای شامل کلیدهای k_i, ..., k_j انتظار می‌رود تعریف می‌کنیم. در نهایت می‌خواهیم $e[1,n]$ را محاسبه کنیم.

زمانی که $j = i - 1$ است حالت ساده‌ای رخ می‌دهد. آنگاه فقط کلید ساختگی d_{i-1} را داریم. هزینه جستجوی مورد انتظار، $e[i, i - 1] = q_{i-1}$ است.

زمانی که $j \geq i$ است، لازم است ریشه k_r را از بین k_i , ... k_j انتخاب کنیم و سپس یک درخت جستجوی دودویی بهینه با کلیدهای k_i ... k_{r-1} به عنوان زیر درخت چپ آن و یک درخت جستجوی دودویی بهینه با کلیدهای k_{r+1}, ..., k_j به عنوان زیر درخت راست آن بسازیم. زمانی که یک زیر درخت، زیر درخت یک گره می‌شود، برای هزینه جستجوی مورد انتظار آن چه اتفاقی می‌افتد؟ عمق هر گره در زیر درخت، به اندازه یک واحد افزایش می‌یابد. با توجه به معادله (۱۵.۱۶)، هزینه جستجوی مورد انتظار این زیر درخت، به اندازه مجموع همه احتمالات در زیر درخت افزایش می‌یابد. برای زیر درختی با کلیدهای k_i, ..., k_j مجموع احتمالات را به صورت زیر مشخص می‌کنیم.

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \cdot \qquad (15.17)$$

بنابراین، اگر k_r ریشه زیر درخت بهینه‌ای که شامل کلیدهای k_i, ..., k_j است باشد داریم:

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j)) .$$

با توجه به اینکه

$$w(i, j) = w(i, r-1) + p_r + w(r+1, j) ,$$

$e[i,j]$ را به صورت زیر بازنویسی می‌کنیم:

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j) \quad (۱۵.۱۸)$$

معادله بازگشتی (۱۵.۱۸) فرض می‌کند که می‌دانیم کدام گره k_r را به عنوان ریشه استفاده کنیم. ریشه‌ای را انتخاب می‌کنیم که پایین‌ترین هزینه جستجوی مورد انتظار را حاصل کند و فرمول بازگشتی نهایی زیر را به ما بدهد.

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i-1 , \\ \min\limits_{i \le r \le j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \le j . \end{cases} \quad (۱۵.۱۹)$$

مقادیر $e[i,j]$ هزینه‌های جستجوی مورد انتظار درخت‌های جستجوی دودویی بهینه را نشان می‌دهند. برای کمک به برقراری ساختار درخت جستجوی بهینه، برای $1 \le i \le j \le n$، $root\ [i,j]$ را به عنوان اندیس r تعریف می‌کنیم که برای آن، k_r ریشه درخت جستجوی دودویی بهینه شامل کلیدهای $k_i, \ ..., \ k_j$ است. گرچه چگونگی محاسبه مقادیر $root[i,j]$ را خواهیم دید، ساخت درخت جستجوی دودویی بهینه از این مقادیر را به عنوان تمرین ۱-۱۵.۵ واگذار می‌کنیم.

مرحله ۳: محاسبه هزینه جستجوی مورد انتظار درخت جستجوی دودویی بهینه

در این مرحله ممکن است به شباهت‌هایی بین ویژگیهای درخت‌های جستجوی دودویی بهینه و ضرب زنجیره‌ای ماتریس‌ها پی برده باشید. برای محدوده هر دو مسئله، زیر مسئله‌هایمان از زیر بازه‌های اندیس پیوسته تشکیل می‌شوند. پیاده سازی مستقیم و بازگشتی معادله (۱۵.۱۹) به کار آمدی الگوریتم مستقیم و بازگشتی ضرب زنجیره‌ای ماتریس‌ها می‌باشد. در عوض، مقادیر $e[i,j]$ را در جدول $e[1\ .\ .\ n+1, 0\ .\ .\ n]$ ذخیره می‌کنیم. اولین اندیس باید به جای n تا $n+1$ اجرا شود، چون به منظور داشتن زیر درختی که تنها شامل کلید ساختگی d_n است، نیاز است که $e[n+1, n]$ را محاسبه و ذخیره کنیم. اندیس دوم باید از صفر شروع شد و چون به منظور داشتن زیر درختی که تنها شامل کلید ساختگی d_0 است، ملزم است که $e[1,0]$ را محاسبه ذخیره کنیم. تنها از ورودیهای $e[i,j]$ که $j \ge i-1$ است استفاده خواهیم کرد. همچنین از جدول $root[i,j]$ برای نگهداری ریشه زیر درخت‌هایی که شامل کلیدهای $k_i, \ ..., \ k_j$ هستند استفاده می‌کنیم. این جدول تنها از ورودیهایی استفاده می‌کند که برای آنها $1 \le i \le j \le n$ باشد.

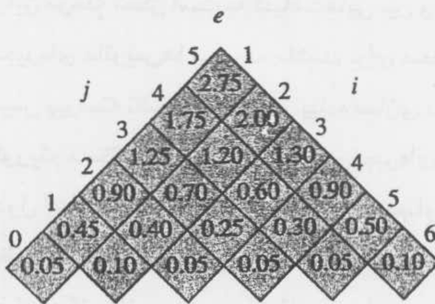
برای کارآیی بیشتر از جدول دیگری نیاز خواهیم داشت. به جای اینکه هر دفعه که $e[i,j]$ را محاسبه می‌کنیم، مقدار $w[i,j]$ را در چک نویس حساب کنیم - که موجب انجام $\Theta(j-i)$ جمع می‌شود - این مقادیر را در جدول $w[1\ .\ .\ n+1, 0\ .\ .\ n]$ ذخیره می‌کنیم. برای حالت پایه $w[i,i-1] = q_{i-1}$ را برای $1 \le i \le n$ محاسبه می‌کنیم. برای $j \ge i$

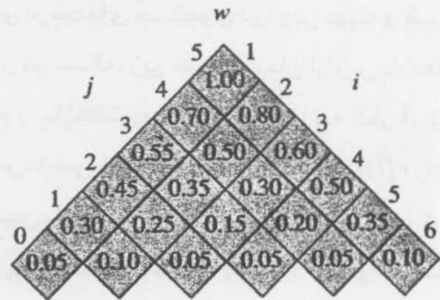
$$w[i, j] = w[i, j-1] + p_j + q_j \qquad (۱۵.۲۰)$$

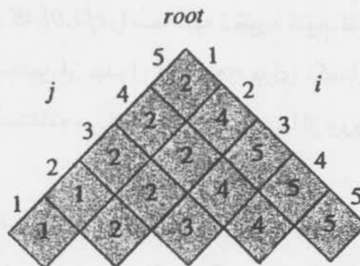
را محاسبه می‌کنیم. بنابراین هر یک از $\Theta(n^2)$ مقدار $w[i,j]$ را در زمان $O(1)$ محاسبه می‌کنیم.

شبه کد زیر، به عنوان ورودی، احتمالات $p_1, \ldots, p_n$ و $q_0, \ldots, q_n$ و اندازه n را گرفته و جدول e و $root$ را برمی‌گرداند.

```
OPTIMAL-BST(p, q, n)
 1  for i ← 1 to n + 1
 2       do e[i, i − 1] ← q_{i−1}
 3          w[i, i − 1] ← q_{i−1}
 4  for l ← 1 to n
 5       do for i ← 1 to n − l + 1
 6              do j ← i + l − 1
 7                 e[i, j] ← ∞
 8                 w[i, j] ← w[i, j − 1] + p_j + q_j
 9                 for r ← i to j
10                     do t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
11                        if t < e[i, j]
12                           then e[i, j] ← t
13                                root[i, j] ← r
14  return e and root
```







شکل ۱۵.۸ جدول‌های $e[i,j]$ و $w[i,j]$ و $root[i,j]$ که توسط OPTIMAL-BST روی توزیع کلیدها که در شکل ۱۵.۷ نشان داده شده است، محاسبه شده‌اند. جدولها چرخانده می‌شوند تا قطرها به صورت افقی اجرا شوند.

با توجه به توضیح فوق و تشابه با روال *MATRIX-CHAIN-ORDER* در بخش ۱۵.۲، عملکرد این روال باید کاملاً روشن باشد. حلقه *for* در خطوط ۳-۱ مقادیر *e[i,i-1]* و *w[i,i - 1]* را مقدار دهی اولیه می‌کند. سپس حلقه *for* در خطوط ۱۳-۴ با استفاده از رابطه‌های بازگشتی (۱۵.۱۹) و (۱۵.۲۰)، *e[i,j]* و *w[i,j]* را برای همه مقادیر $1 \le i \le j \le n$ محاسبه می‌کند. در اولین تکرار، زمانیکه $l=1$ است، حلقه، *e[i,j]* و *w[i,i]* را برای $i = 1,2,..., n$ محاسبه می‌کند. تکرار دوم با $l = 2$، *e[i, i + 1]* و *w[i,i+ 1]* را برای $i = 1, 2, ...,n - 1$ محاسبه می‌کند و به همین ترتیب درونی‌ترین حلقه *for* در خطوط ۱۳-۹ هر اندیس *r* کاندید شده را امتحان می‌کند تا مشخص کند از کدام کلید k_r به عنوان ریشه درخت جستجوی دودویی بهینه شامل کلیدهای $k_i, \dots, k_j$ استفاده شود. این حلقه *for* هر زمان که کلید بهتری برای ریشه پیدا کند مقدار جاری اندیس *r* را در *root[i,j]* ذخیره می‌کند.

شکل ۱۵.۸ جدولهای *e[i ,j]*، *w[i ,j]* و *root[i ,j]* را نشان می‌دهد که توسط روال *OPTIMAL-BST* روی توزیع کلیدها در شکل ۱۵.۷ محاسبه شده‌اند. همانند مثال ضرب زنجیره‌ای ماتریسها، جدولها چرخانده می‌شوند تا قطرها به صورت افقی اجرا شوند. *OPTIMAL-BST*، سطرها را از پایین به بالا و هر سطر را از چپ به راست محاسبه می‌کند.

روال *OPTIMAL-BST* دقیقاً مانند *MATRIX-CHAIN-ORDER* زمان $\Theta(n^3)$ را صرف می‌کند. درک اینکه زمان اجرا $O(n^3)$ است ساده می‌باشد چون این روال سه حلقه *for* تو در تو دارد و هر اندیس حلقه حداکثر *n* مقدار را در بر می‌گیرد. بازه اندیس‌های حلقه در *OPTIMAL-BST* دقیقاً همانند *MATRIX-CHAIN-ORDER* نمی‌باشد، اما در همه جهت‌ها، حداکثر درون یک بازه قرار دارند. بنابراین همانند *MATRIX-CHAIN-ORDER*، روال *OPTIMAL-BST*، زمان $\Omega(n^3)$ را صرف می‌کند.

تمرین‌ها

۱۵.۵-۱ شبه کدی برای روال *CONSTRUCT-OPTIMAL-BST(root)* بنویسید که با دریافت جدول *root* ساختار درخت جستجوی دودویی بهینه را به عنوان خروجی برگرداند. برای مثال شکل ۱۵.۸، شبه کد شما باید ساختار زیر را چاپ کند.

k_2 *is the root*

k_1 *is the left child of* k_2

d_0 *is the left child of* k_1

d_1 *is the right child of* k_1

k_5 *is the right child of* k_2

k_4 *is the left child of* k_5

k_3 *is the left child of* k_4

d_2 *is the left child of* k_3

d_3 *is the right child of* k_3

d_4 *is the right child of* k_4

d_5 *is the right child of* k_5

که با درخت جستجوی دودویی بهینه‌ایی که در شکل ۱۵.۷*(b)* نشان داده شده است متناظر می‌باشد.

۱۵.۵-۲ هزینه و ساختار درخت جستجوی دودویی بهینه‌ای برای مجموعه‌ای از $n = 7$ کلید با احتمالات زیر را تعیین کنید.

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

۱۵.۵-۳ فرض کنید به جای نگهداری جدول *w[i,j]*، مقدار *w(i,j)* را مستقیماً از معادله (۱۵.۱۷) در خط ۸ روال *OPTIMAL-BST* محاسبه کرده‌ایم و در خط ۱۰ از این مقدار محاسبه شده استفاده نموده‌ایم. این تغییر چگونه روی زمان اجرای مجانبی *OPTIMAL-BST* اثر می‌گذارد؟

۱۵.۵-۴* *Knuth* نشان داده است که همیشه ریشه‌های زیر درختهای بهینه وجود دارند، به طوری که برای همه مقادیر $1 \le i < j \le n$، $root[i,j-1] \le root[i,j] \le root[i+1,j]$ با استفاده از این مطلب *OPTIMAL-BST* را به گونه‌ای تغییر دهید که در زمان $\Theta(n^2)$ اجرا شود.

مسائل

۱۵-۱ مسئله اقلیدسی *Bitonic*[۱] فروشنده دوره گرد

مسئله اقلیدسی فروشنده دوره گرد، مسئله مشخص کردن کوتاهترین مسیر بسته می‌باشد که در یک صفحه، مجموعه‌ای از n نقطه داده شده را بهم متصل می‌کند. شکل ۱۵.۹*(a)* حل مسئله‌ای با ۷ نقطه را نشان می‌دهد. مسئله کلی، *NP* کامل است و بنابراین حل آن نیاز به زمانی بیشتر از زمان نمایی دارد.

J. L. Bentley پیشنهاد کرده است که برای ساده سازی مسئله، توجهمان را به *مسیرهای دوآهنگه*[۲] معطوف کنیم، به عبارت دیگر، مسیرهایی که از سمت چپ‌ترین نقطه شروع می‌شوند، اکیداً از چپ به

---

۱ - دو آهنگه

2. bitonic paths

راست به سمت راست‌ترین نقطه رفته و سپس اکیداً از راست به چپ به نقطه شروع باز می‌گردند. شکل ۱۵.۹ *(b)* کوتاهترین مسیر *bitonic* را برای همان ۷ نقطه نشان می‌دهد در این حالت الگوریتمی با زمان چند جمله‌ای قابل اعمال است.

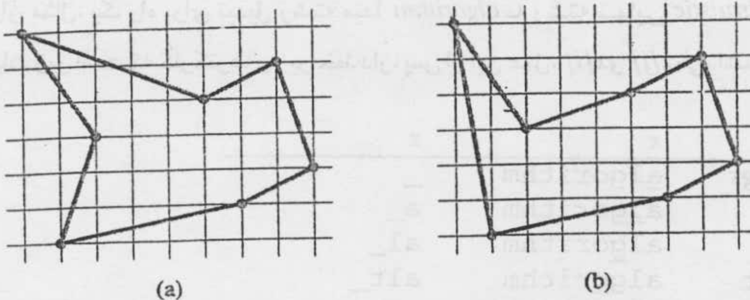
الگوریتمی با زمان $O(n^2)$ برای تعیین مسیر *bitonic* بهینه شرح دهید، فرض کنید هیچ دو نقطه‌ای دارای مختصات x مشابه نیستند. (راهنمایی: با پیمایش از چپ به راست راههای ممکن بهینه برای دو قسمت مسیر را نگهداری کنید.)

۱۵-۲ چاپ منظم

مسئله چاپ منظم یک پاراگراف در یک چاپگر را در نظر بگیرید. متن ورودی، رشته‌ای از n کلمه با طولهای $l_1, \ldots, l_n$ بر حسب تعداد کارکترها است. می‌خواهیم طوری این پاراگراف را به طور منظم در تعدادی خط چاپ کنیم که هر یک حداکثر M کارکتر را در برگیرد. معیار منظم به صورت زیر است. اگر یک خط داده شده وقتی $i \leq j$ است شامل کلمات i تا j باشد و بین کلمات یک فاصله داشته باشیم، تعداد کارکترهای فاصله اضافی در آخر خط،

$M - j + i - \sum_{k=i}^{j} l_k,$

است که باید نامنفی باشد تا کلمات در خط جای گیرند. می‌خواهیم مجموع مکعبات تعداد کارکترهای فاصله، برای همه خطوط به جز خط آخر مینیمم شود. الگوریتمی به روش برنامه سازی پویا ارائه دهید که پاراگرافی با n کلمه را به طور منظم روی یک چاپگر چاپ کند. زمان اجرا و فضای مورد نیاز الگوریتمتان را تحلیل کنید.



شکل ۱۵.۹ هفت نقطه در یک صفحه که روی یک شبکه واحد نشان داده شده‌اند. *(a)* کوتاهترین مسیر بسته با طول تقریباً *24.89* این مسیر *bitonic* نمی‌باشد. *(b)* کوتاهترین مسیر *bitonic* برای همان مجموعه از نقاط. طول آن تقریباً برابر *25.58* است.

۱۵-۳ ویرایش مسافت

به منظور تغییر رشته مبدا $x[1..m]$ از متن، به رشته مورد نظر $y[1..n]$ می‌توانیم تبدیلات مختلفی را انجام دهیم. x و y داده شده‌اند، هدف ما ایجاد دنباله‌ای از تبدیلات است که x را به y تغییر دهد. از آرایه

z - فرض کنید برای نگهداری همه کاراکترهایی که نیاز خواهد داشت، به اندازه کافی بزرگ است - برای نگهداری نتایج میانی استفاده می‌کنیم. در ابتدا z خالی است و در انتها باید برای $j = 1, 2, \dots, n$ داشته باشیم $z[j] = y[j]$ اندیس‌های جاری i , j را به ترتیب در x,z نگه می‌داریم و عملیات برای تغییر z و این اندیس‌ها مجاز می‌باشد. در ابتدا $i = j = 1$ است. ملزم هستیم که هر کاراکتر در x را ضمن تبدیل، بررسی کنیم، به این معنا که در انتهای توالی عملیات تبدیل، باید داشته باشیم $i = m + 1$ شش عمل تبدیل وجود دارد:

کپی کردن یک کارکتر از x به z با انتساب $z[j] \leftarrow x[i]$ و سپس افزایش i,j. این عمل $x[i]$ را بررسی می‌کند.

جایگزین کردن یک کارکتر از x با کارکتر دیگر c، توسط انتساب $z[j] \leftarrow c$ و سپس افزایش i و j. این عمل $x[i]$ را بررسی می‌کند.

حذف یک کارکتر از x با افزایش i اما بدون تغییر j. این عمل $x[i]$ را بررسی می‌کند.

درج کارکتر c در z با انتساب $z[j] \leftarrow c$ و سپس افزایش j اما بدون تغییر i. این عمل هیچ کارکتری از x را بررسی نمی‌کند.

معاوضه یعنی مبادله دو کارکتر بعدی با کپی کردن آنها از x به z اما با ترتیب معکوس، با انتساب $z[j] \leftarrow x[i+1]$ و $z[j+1] \leftarrow x[i]$ و سپس $i \leftarrow i + 2$ و $j \leftarrow j + 2$ این کار را انجام می‌دهیم. این عمل $x[i]$ و $x[i+1]$ را بررسی می‌کند.

پاک کردن باقیمانده x بوسیله انتساب $i \leftarrow m + 1$ این عمل همه کارکترهای x را که هنوز بررسی نشده‌اند بررسی می‌کند. اگر این عمل انجام شود، باید آخرین عمل باشد.

به عنوان مثال، یک راه برای تبدیل رشته مبدأ *algorithm* به رشته نهایی *altruistic*، استفاده از توالی عملیات زیر است که کارکترهای زیر خط دار، پس از این عمل، $x[i]$ و $z[j]$ می‌باشند:

Operation	x	z
initial strings	algorithm	_
copy	algorithm	a_
copy	algorithm	al_
replace by t	algorithm	alt_
delete	algorithm	alt_
copy	algorithm	altr_
insert u	algorithm	altru_
insert i	algorithm	altrui_
insert s	algorithm	altruis_
twiddle	algorithm	altruisti_
insert c	algorithm	altruistic_
kill	algorithm_	altruistic_

توجه کنید که چندین توالی عملیات تبدیل دیگر وجود دارند که *algorithm* را به *altruistic* تبدیل می‌کنند.

هر یک از این عملیات تبدیل دارای هزینه مرتبطی می‌باشند. هزینه یک عمل به کاربرد خاص آن بستگی دارد ولی فرض می‌کنیم که هزینه هر عمل، مقدار ثابتی است که برای ما مشخص است. همچنین فرض می‌کنیم که هزینه‌های عملیات کپی و جایگزینی به تنهایی کمتر از هزینه‌های ترکیب شده عملیات حذف و درج است؛ در غیر اینصورت عملیات جایگزینی و کپی مورد استفاده قرار نمی‌گرفتند. هزینه یک توالی داده شده از عملیات تبدیل، مجموع هزینه‌های عملیات انفرادی در توالی می‌باشد. برای توالی فوق، هزینه تبدیل *algorithm* به *altruistic* به صورت زیر است:

3.(هزینه (کپی)) + (هزینه (جایگزین) + هزینه (حذف) + *4*).(هزینه (درج)

\+ هزینه (پاک کردن) + هزینه (معاوضه) +

a. دو رشته $x[1 .. m]$ و $y[1 .. n]$ و مجموعه‌ای از هزینه‌های عملیات تبدیل داده شده‌اند، **ویرایش مسافت** از x به y ارزانترین هزینه توالی عملیات برای تبدیل x به y است. الگوریتم برنامه سازی پویایی شرح دهید که ویرایش مسافت از $x[1 .. m]$ به $y[1 .. n]$ را پیدا کرده و توالی عملیات بهینه را چاپ کند. زمان اجرا و فضای مورد نیاز برای الگوریتمتان را تحلیل کنید.

مسئله ویرایش مسافت، تعمیم مسئله همتراز کردن دو رشته *DNA* است. چندین روش برای سنجش تشابه دو رشته *DNA* توسط همتراز کردن آنها وجود دارد. یک چنین روشی برای همترازی دو رشته x و y از درج فاصله‌هایی در مکانهای دلخواه در دو رشته (حتی در انتهای آنها) تشکیل شده است تا رشته‌های حاصل x' و y' طول یکسانی داشته باشند، ولی فاصله‌ای در مکان مشابه نداشته باشند. (یعنی برای هیچ موقعیت j هم $x'[j]$ و هم $y'[j]$ کارکتر فاصله نیستند.) سپس یک امتیاز به هر موقعیت نسبت می‌دهیم. موقعیت j امتیازی به صورت زیر دریافت می‌کند:

- *1* + اگر $x'[j] = y'[j]$ و هیچ یک کاراکتر فاصله نباشند.
- *1* - اگر $x'[j] \neq y'[j]$ و هیچ یک کاراکتر فاصله نباشند.
- *2* - اگر $x'[j]$ یا $y'[j]$ فاصله باشند.

امتیاز همترازی، مجموع امتیازهای تک‌تک موقعیت‌ها است. برای مثال، یک همترازی برای رشته‌های $x = GATCGGCAT$ و $y = CAATGTGAATC$ به صورت زیر است:

```
G ATCG GCAT
CAAT GTGAATC
-*++*+*+-++*
```

علامت + در زیر یک موقعیت امتیاز *1* + را برای آن موقعیت مشخص می‌کند. علامت - امتیاز *1* - و علامت * امتیاز *2* - را مشخص می‌کند. بنابراین این همترازی دارای امتیاز کلی $6.1 - 2.1 - 4.2 = -4$

است.

b. توضیح دهید چطور می‌توان مسئله پیدا کردن همترازی بهینه را با استفاده از زیر مجموعه‌ای از عملیات تبدیل شامل کپی، جایگزینی، حذف، درج، معاوضه و پاک کردن به مسئله ویرایش مسافت تبدیل کرد.

۱۵-۴ برنامه ریزی مهمانی یک شرکت

پرفسور *Stewart* با رئیس یک اتحادیه که در حال برنامه ریزی مهمانی یک شرکت است مشاوره می‌کند. شرکت، ساختاری تسلسلی دارد؛ به عبارت دیگر رابطه نظارت، درختی را تشکیل می‌دهد که از رئیس مشتق می‌شود. اداره کارکنان، هر کارمند را با میزان خوش برخورد بودن که یک عدد حقیقی است رتبه بندی می‌کند. برای اینکه به همه حاضران خوش بگذرد، رئیس نمی‌خواهد هم یک کارمند و هم ناظر بی‌واسطه او با هم در مهمانی شرکت کنند. پرفسور *Stewart* درختی را دریافت می‌کند که ساختار شرکت را با استفاده از نمایش فرزند چپ، همزاد راست که در بخش ۱۰.۴ توضیح داده شد، توصیف می‌کند. هر گره در درخت علاوه بر اشاره گرها، نام کارمند و رتبه خوش برخورد بودن او را نگه می‌دارد. الگوریتمی شرح دهید که طوری لیست مهمانان را بسازد که مجموع میزان خوش گذشتن به میهمانان، ماکزیمم شود. زمان اجرای الگوریتم خود را تحلیل کنید.

۱۵-۵ الگوریتم *Viterbi*

برای درک کلام، می‌توانیم از برنامه سازی پویا روی یک گراف جهتدار $G = (V,E)$ استفاده کنیم. هر یال $(u,v) \in E$ با یک صدای $\sigma(u,v)$ از مجموعه محدود $\sum$ از صداها، برچسب گذاری شده است. گراف بر چسب گذاری شده، مدل رسمی صحبت یک فرد به یک زمان محدود شده می‌باشد. هر مسیر در گراف از رأس متمایز $v_0 \in V$ شروع می‌شود که با توالی ممکن از صداهای تولید شده توسط مدل، متناظر است. بر چسب یک مسیر جهتدار، به عنوان الصاق بر چسب‌های یالهای روی آن مسیر تعریف می‌شود.

a. الگوریتم کارآمدی را شرح دهید که با دریافت گراف G با یالهای بر چسب گذاری شده همراه با رأس متمایز v_0 و توالی $S = \langle\sigma_1, \sigma_2, \ldots, \sigma_k\rangle$ از کارکترهای $\sum$، مسیری در G برگرداند که از v_0 شروع می‌شود و اگر چنین مسیری وجود داشت، S را به عنوان بر چسبش داشته باشد. در غیر اینصورت، الگوریتم باید *NO-SUCH-PATH* را برگرداند. زمان اجرای الگوریتم خود را تحلیل کنید (راهنمایی: ممکن است ایده‌های فصل ۲۲ مفید باشند.)

اکنون فرض کنید که برای هر یال $(u,v) \in E$، احتمال نامنفی $p(u,v)$ برای پیمایش یال (u,v) از رأس u داده شده است و بنابراین صدای متناظر را تولید می‌کند. مجموع احتمالات یالهایی که یک رأس را ترک می‌کنند، برابر 1 است. احتمال هر مسیر به عنوان حاصل ضرب احتمالات یالهای آن تعریف می‌شود.

می‌توانیم احتمال مسیری که از v_0 شروع می‌شود را به عنوان احتمالی در نظر بگیریم که یک پیمایش تصادفی شروع شده از v_0، مسیر خاصی را دنبال کند، که انتخاب اینکه کدام یال رأس x در نظر گرفته شود، به طور احتمالی با توجه به احتمالات یالهای موجودی که u را ترک می‌کنند صورت می‌پذیرد.

b. جوابتان برای قسمت (a) را طوری بسط دهید تا اگر مسیری برگردانده شد، محتمل‌ترین مسیری باشد که از v_0 شروع شده و دارای بر چسب S است. زمان اجرای الگوریتم خود را تحلیل کنید.

۱۵-۶ حرکت روی یک صفحه بازرسی

فرض کنید یک صفحه بازرسی $n \times n$ و یک بازرس به شما داده شده است. شما باید با توجه به قوانین زیر، بازرس را از لبه پایین صفحه به لبه بالای صفحه حرکت دهید. در هر مرحله می‌توانید بازرس را به یکی از سه مربع زیر حرکت دهید:

۱. مربع بالایی بلافاصله

۲. مربعی که یکی بالا و یکی چپ است (اما تنها اگر بازرس قبلاً در چپ‌ترین ستون نبوده باشد)

۳. مربعی که یکی بالا و یکی راست است (اما تنها اگر بازرس قبلاً در سمت راست‌ترین ستون نبوده باشد.)

هر بار با حرکت از مربع x به مربع y، $p(x,y)$ دلار دریافت می‌کنید. برای همه زوجهای (x,y) که حرکت از x به y مجاز است، $p(x,y)$ به شما داده می‌شود. فرض نکنید که $p(x,y)$ مثبت است.

الگوریتمی ارائه دهید که مجموعه‌ای از حرکتها را پیدا کند که بازرس را طوری از جایی در لبه پایین به جایی در لبه بالا حرکت دهد که بیشترین دلارهای ممکن را دریافت کنید. الگوریتم شما آزاد است که در لبه پایین، هر مربعی را به عنوان نقطه شروع و هر مربعی در لبه بالا را به عنوان مقصد، انتخاب کند تا تعداد دلارهای دریافت شده در مسیر، ماکزیمم شود. زمان اجرای الگوریتمتان چیست؟

۱۵-۷ زمان بندی برای ماکزیمم کردن سود

فرض کنید یک ماشین و مجموعه‌ای از n کار $a_1, a_2, \ldots, a_n$ را برای پردازش روی آن ماشین دارید. هر کار a_j دارای زمان پردازش t_j و سود p_j و مهلت d_j است. ماشین می‌تواند در هر زمان تنها یک کار را پردازش کند و کار a_j باید بدون وقفه برای t_j واحد زمانی متوالی، اجرا شود. اگر کار a_j در مهلت d_j خود کامل شود، سود p_j را دریافت می‌کنید، ولی اگر بعد از مهلتش کامل شود، سود 0 را دریافت می‌کنید. الگوریتمی ارائه دهید که جدولی را پیدا کند که با فرض اینکه همه زمانهای پردازش، اعداد صحیح بین 1 و n هستند، ماکزیمم مقدار سود را بدست آورد، زمان اجرای الگوریتمتان چیست؟

۱۶ الگوریتم‌های حریصانه

الگوریتم‌های حریصانه، الگوریتم‌هایی هستند که جهت بهینه سازی مسائل بکار می‌روند، و معمولاً یک توالی از گام‌ها و مراحل را با یک مجموعه از انتخابها در هر مرحله طی می‌کنند. برای بسیاری از مسائل بهینه سازی، استفاده از روش برنامه‌سازی پویا جهت تعیین بهترین انتخابها، عملی افراط‌آمیز است؛ به بیان ساده‌تر، الگوریتم‌های کارآمدتری این کار را انجام خواهند داد. یک ***الگوریتم حریصانه***[1]، همواره انتخابی را انجام می‌دهد که در همان لحظه، بهترین انتخاب به نظر می‌رسد. به عبارت دیگر این الگوریتم یک انتخاب بهینه محلی انجام می‌دهد، به این امید که این انتخاب به یک جواب بهینه کلی منجر خواهد شد. این فصل مسائل بهینه‌سازی را که با الگوریتم‌های حریصانه قابل حل می‌باشند، بررسی می‌کند. قبل از مطالعه این فصل، شما باید مبحث برنامه سازی پویا در فصل ۱۵، بویژه بخش ۱۵.۳ را مطالعه نمایید.

الگوریتم‌های حریصانه همواره جواب بهینه را حاصل نمی‌کنند، اما در بسیاری از مسائل، جواب بهینه را حاصل می‌کنند. ابتدا در بخش ۱۶.۱ یک مسئله ساده اما با اهمیت را بررسی خواهیم کرد، مسئله انتخاب فعالیت، که الگوریتم حریصانه برای آن جواب را به طور مؤثر و کارآمدی محاسبه می‌کند. ابتدا با در نظر گرفتن یک راه حل برنامه سازی پویا و سپس نمایش اینکه همواره می‌توانیم انتخابهای حریصانه داشته باشیم تا به یک جواب بهینه دست یابیم، به الگوریتم حریصانه خواهیم رسید. بخش ۱۶.۲ عناصر اصلی روش حریصانه را بیان می‌کند، که برای اثبات درستی الگوریتم‌های حریصانه، روش مستقیم‌تری را نسبت به فرآیند مبتنی بر برنامه‌سازی پویا در بخش ۱۶.۱، ارائه می‌کند. بخش ۱۶.۳، یک کاربرد مهم تکنیکهای حریصانه را نشان می‌دهد: طراحی کدهای فشرده سازی داده‌ها (کد *huffman*). در بخش ۱۶.۴، تعدادی از تئوریهایی که پایه ساختارهای ترکیبی هستند را بررسی می‌کنیم که این ساختارهای ترکیبی، "*matroid*" نامیده می‌شوند و الگوریتم حریصانه، همواره جواب بهینه را برای آنها تولید می‌کند. سرانجام بخش ۱۶.۵، کاربرد *matroid*ها را با استفاده از مسئله زمانبندی کارهایی با طول یک واحد زمانی همراه با مهلت‌ها و جریمه‌ها، توضیح می‌دهد.

---

1. greedy algorithm

روش حریصانه یک روش کاملاً قدرتمند است و در گستره وسیعی از مسائل بخوبی عمل می‌کند. فصل‌هایِ بعدی، بسیاری از الگوریتم‌هایی را بیان خواهند کرد که می‌توانند به عنوان کاربردهای روش حریصانه در نظر گرفته شوند، شامل الگوریتم‌های درخت پوشای مینیمم (فصل ۲۳) و الگوریتم *Dijkstra* برای کوتاهترین مسیرها از یک مبدأ واحد (فصل ۲۴). الگوریتم‌های درخت پوشای مینیمم، نمونه‌ای کلاسیک از روش حریصانه هستند. اگر چه این فصل و فصل ۲۳ می‌توانند به صورت مستقل و مجزا از هم مطالعه شوند، اما مطالعه این دو فصل با هم را مفید می‌یابید.

۱۶.۱ مسئله انتخاب فعالیت

اولین مثال، مسئلهٔ زمان بندی چندین فعالیت در حال رقابت است که نیاز به استفاده منحصر بفرد از یک منبع مشترک دارند، با هدف انتخاب مجموعه‌ای با ماکزیمم اندازه از فعالیت‌هایی که متقابلاً با یکدیگر سازگارند. فرض کنید یک مجموعه $S = \{a_1, a_2, \ldots, a_n\}$ از n ***فعالیت***[1] پیشنهادی داریم که می‌خواهند از یک منبع استفاده نمایند، مانند یک تالار سخنرانی که در یک زمان، می‌تواند تنها مورد استفاده یک فعالیت قرار گیرد. هر فعالیت a_i، دارای ***زمان شروع***[2] s_i و ***زمان خاتمه***[3] f_i است، بطوریکه $0 \le s_i < f_i < \infty$. اگر فعالیت a_i انتخاب شود، می‌تواند در طول بازهٔ زمانی نیمه باز $[s_i, f_i)$ رخ دهد. فعالیت‌های a_i و a_j ***سازگار***[4] هستند، اگر بازه‌های $[s_i, f_i)$ و $[s_j, f_j)$ همپوشانی نداشته باشند (به عبارت دیگر، a_i و a_j سازگارند، اگر $s_i \ge f_j$ و $s_j \ge f_i$). ***مسئله انتخاب فعالیت***[5]، عبارتست از انتخاب یک زیرمجموعه با ماکزیمم اندازه از فعالیت‌های متقابلاً سازگار. برای مثال، مجموعهٔ S زیر از فعالیت‌ها را در نظر بگیرید، که آنها را به ترتیب صعودی یکنواخت از زمانهای خاتمه مرتب کرده‌ایم:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

(بزودی خواهیم دید که چرا در نظر گرفتن فعالیت‌ها به صورت مرتب، با مزیت است.)

برای این مثال، زیرمجموعه $\{a_3, a_9, a_{11}\}$ شامل فعالیت‌های متقابلاً سازگار است.

اگر چه این زیرمجموعه یک مجموعه ماکزیمم نیست، چون زیرمجموعه $\{a_1, a_4, a_9, a_{11}\}$ بزرگتر از آن است. در حقیقت، $\{a_1, a_4, a_9, a_{11}\}$ بزرگترین زیرمجموعه از فعالیت‌های متقابلاً سازگار است.

بزرگترین زیرمجموعه دیگر، زیرمجموعه $\{a_2, a_4, a_9, a_{11}\}$ می‌باشد.

---

1. *activity*
2. *start time*
3. *finish time*
4. *compatible*
5. *activity-selection problem*

این مسئله را طی چندین مرحله حل خواهیم کرد. ابتدا با فرموله کردن یک حل برنامه سازی پویا برای این مسئله شروع می‌کنیم که در آن جوابهای بهینه دو زیرمسئله را با هم ترکیب کرده تا یک جواب بهینه برای مسئله اصلی ایجاد کنیم. در هنگام تعیین اینکه کدام زیرمسائل در حل بهینه استفاده گردند، انتخابهای متعددی را در نظر می‌گیریم. پس از آن خواهیم دید که لازم است تنها یک انتخاب - انتخاب حریصانه - را در نظر بگیریم و اینکه زمانیکه این انتخاب حریصانه را انجام می‌دهیم، یکی از زیر مسائل تضمین می‌شود که تهی باشد، لذا تنها یک زیرمسئله ناتهی، باقی می‌ماند. بر اساس این مشاهدات، یک الگوریتم حریصانه بازگشتی برای حل مسئله زمان‌بندی فعالیتها، ارائه خواهیم نمود. فرآیند توسعه راه حل را با تبدیل الگوریتم بازگشتی به الگوریتم تکراری کامل خواهیم کرد. اگر چه مراحلی را که در این بخش طی می‌کنیم از روال معمول الگوریتم‌های حریصانه بسیار پیچیده‌ترند، اما رابطه الگوریتم‌های حریصانه و برنامه سازی پویا را بیان می‌کنند.

زیرساختار بهینه مسئله انتخاب فعالیت

همانطور که در بالا اشاره شد، با ارائه یک راه حل برنامه سازی پویا برای مسئله انتخاب فعالیت شروع می‌کنیم. همانند فصل ۱۵، اولین گام، پیدا کردن زیرساختار بهینه و سپس استفاده از آن برای ساخت جواب بهینه مسئله از روی جوابهای بهینه زیرمسائل است.

در فصل ۱۵، مشاهده کردیم که نیازی به تعریف یک فضای مناسب از زیرمسائل نداریم. با تعریف مجموعه‌های

$$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\} \ ,$$

شروع می‌کنیم، بطوریکه S_{ij} زیرمجموعه‌ای از فعالیت‌ها در S است که می‌توانند بعد از خاتمه فعالیت a_i شروع شده و قبل از شروع فعالیت a_j خاتمه یابند. در حقیقت، S_{ij} شامل همه فعالیت‌هایی است که با a_i و a_j و همچنین با همه فعالیت‌هایی که بعد از خاتمه a_i خاتمه نیافته و همه فعالیت‌هایی که قبل از شروع a_j شروع نمی‌شوند، سازگارند. به منظور بیان کل مسئله، فعالیت‌های ساختگی a_0 و a_{n+1} را اضافه کرده و توافق می‌کنیم که $f_0=0$ و $s_{n+1}=\infty$ پس $S = S_{0,n+1}$ و محدوده تغییر i و j بصورت $0 \le i, j \le n+1$ خواهد بود.

دامنه تغییر i و j را بصورت زیر می‌توانیم بیشتر محدود کنیم. فرض می‌کنیم که فعالیت‌ها در یک ترتیب یکنواخت صعودی از زمانهای خاتمه مرتب شده‌اند:

$$f_0 \le f_1 \le f_2 \le \ldots \le f_n < f_{n+1} \qquad (۱۶.۱)$$

ادعا می‌کنیم که $S_{ij} = \phi$ هرگاه $i \ge j$ باشد. چرا؟ فرض کنید یک فعالیت $a_k \in S_{ij}$ برای $i \ge j$ وجود دارد، به طوری که a_i بعد از a_j در ترتیب مرتب قرار دارد. پس خواهیم داشت $f_i \le s_k < f_k \le s_j < f_j$. بنابراین $f_i < f_j$ که با این فرض که a_i بعد از a_j در ترتیب مرتب قرار دارد در تناقض است. می‌توانیم

نتیجه بگیریم با این فرض که فعالیت‌ها را در یک ترتیب صعودی یکنواخت از زمانهای خاتمه مرتب کرده‌ایم، فضای زیرمسائل انتخاب زیرمجموعه با ماکزیمم اندازه شامل فعالیت‌های متقابلاً سازگار از S_{ij}است، که $0 \le i < j \le n+1$ با آگاهی از این مطلب که تمام S_{ij}های دیگر تهی هستند.

برای مشاهده زیرساختار مسئله انتخاب فعالیت، تعدادی زیرمسئله غیر تهی S_{ij} را در نظر بگیرید،[1] و فرض کنید که یک جواب برای S_{ij} شامل تعدادی فعالیت a_k است، بطوریکه $f_i \le s_k < f_k \le s_j$. استفاده از فعالیت a_k سبب ایجاد دو زیرمسئله می‌شود، S_{ik} (فعالیت‌هایی که پس از خاتمه فعالیت a_i شروع شده و قبل از شروع فعالیت a_k خاتمه می‌یابند) و S_{kj} (فعالیت‌هایی که پس از خاتمه a_k شروع شده و قبل از شروع a_j خاتمه می‌یابند)، که هر کدام از یک زیرمجموعه شامل فعالیت‌های داخل S_{ij} تشکیل شده‌اند. جواب S_{ij} اجتماع جواب‌های مربوط به S_{ik} و S_{kj} است، همراه با فعالیت a_k بنابراین تعداد فعالیت‌ها در جواب S_{ij} برابر اندازه جواب S_{ik} به علاوه اندازه جواب S_{kj} به علاوه یک (برای a_k) می‌باشد.

زیرساختار بهینه این مسئله به شکل زیر است. اکنون فرض کنید جواب A_{ij} برای S_{ij} شامل فعالیت a_k می‌باشد. پس جواب‌های A_{ik} برای S_{ik} و A_{kj} برای S_{kj} که در این جواب بهینه برای S_{ij} استفاده می‌شوند نیز باید بهینه باشند. بحث برش - و - الصاق معمول در این مورد بکار می‌رود. اگر یک جواب A'_{ik} برای S_{ik} می‌داشتیم که شامل فعالیت‌های بیشتری از A_{ik} می‌بود، می‌توانستیم A_{ik} را از داخل A_{ij} برش داده و به داخل A'_{ik} الصاق نماییم، بنابراین یک جواب دیگر برای S_{ij} با تعداد فعالیت‌های بیشتری از A_{ij} تولید می‌شد. از آنجا که فرض کردیم A_{ij} یک جواب بهینه است، به یک تناقض رسیده‌ایم. به طور مشابه اگر جواب A'_{kj} برای S_{kj} را با فعالیت‌های بیشتر از A_{kj} می‌داشتیم، می‌توانستیم A_{kj} را با A'_{kj} جایگزین کنیم تا یک جواب با فعالیت‌های بیشتری از A_{ij} تولید نماییم.

اکنون از زیرساختار بهینه خود استفاده کرده تا نشان دهیم که می‌توانیم یک جواب بهینه برای مسئله از روی جواب‌های بهینه زیرمسائل بسازیم. مشاهده کردیم که هر جواب برای یک زیرمسئله غیر تهی S_{ij} شامل فعالیت a_k است و آنکه هر جواب بهینه در درون خود شامل جواب‌های بهینه نمونه زیرمسئله‌های S_{ik} و S_{kj} می‌باشد. بنابراین، می‌توانیم یک زیرمجموعه با ماکزیمم اندازه از فعالیت‌های متقابلاً سازگار در S_{ij} بسازیم، با تقسیم مسئله به دو زیرمسئله (یافتن زیرمجموعه‌های با ماکزیمم اندازه از فعالیت‌های متقابلاً سازگار در S_{ik} و S_{kj}) و پیدا کردن زیرمجموعه‌های با ماکزیمم اندازه A_{ik} و A_{kj} از فعالیت‌های متقابلاً سازگار برای این زیرمسائل و سپس تشکیل زیرمجموعه A_{ij} با ماکزیمم اندازه شامل فعالیت‌های متقابلاً سازگار بصورت

---

۱ - گاهی اوقات از مجموعه‌های S_{ij} به عنوان زیر مسائل صحبت می‌کنیم، نه مجموعه‌هایی از فعالیت‌ها. همواره با توجه به متن واضح خواهد بود که چه موقع به S_{ij} به عنوان یک مجموعه از فعالیت‌ها و چه موقع به عنوان زیرمسئله‌ای که ورودی آن، این مجموعه است، اشاره می‌کنیم.

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \quad (۱۶.۲)$$

یک جواب بهینه برای کل مسئله، یک جواب برای $S_{0,\ n+1}$ است.

راه حل بازگشتی

گام دوم در ادامه روند یک حل برنامه سازی پویا آن است که مقدار جواب بهینه را بصورت بازگشتی تعریف کنیم. برای مسئله انتخاب فعالیت، $c[i,\ j]$ را تعداد فعالیت‌ها در زیرمجموعه با ماکزیمم اندازه شامل فعالیت‌های متقابلاً سازگار در S_{ij} قرار می‌دهیم. هرگاه $S_{ij} = \phi$، داریم $c[i,\ j] = 0$ ؛ خصوصاً برای $i \geq j$ $c[i,\ j] = 0$ است.

اکنون زیرمجموعه غیر تهی S_{ij} را در نظر بگیرید. همانطور که مشاهده کردیم، اگر a_k در زیرمجموعه با ماکزیمم اندازه از فعالیت‌های متقابلاً سازگار در S_{ij} استفاده شود، از زیرمجموعه های با ماکزیمم اندازه از فعالیت‌های متقابلاً سازگار برای زیرمسائل S_{ik} و S_{kj} استفاده می‌کنیم.

با استفاده از تساوی (۱۶.۲)، رابطه بازگشتی زیر را داریم

$$c[\ i,\ j] \ = \ c[i,\ k\ ] \ + \ c[k,\ j] \ + \ 1$$

در این معادله بازگشتی فرض می‌شود که مقدار k را می‌دانیم، که اینگونه نیست. $j-i-1$ مقدار ممکن برای k وجود دارد، به عبارت دیگر $k = i+1,\ .\ .\ .\ ,\ j-1$ از آنجا که زیرمجموعه با ماکزیمم اندازه از S_{ij} باید یکی از این مقادیر k را استفاده نماید، تمام آنها را امتحان می‌کنیم تا بهترین را پیدا نماییم. بنابراین تعریف بازگشتی کامل $c[i,\ j]$ بصورت زیر خواهد بود

$$c[i,\ j] = \begin{cases} 0 & \text{اگر } S_{ij} = \phi \\ \max\limits_{i<k<j} \{c[i,k] + c[k,j] + 1\} & \text{اگر } S_{ij} = \phi \end{cases} \quad (۱۶.۳)$$

تبدیل یک راه حل برنامه سازی پویا به یک راه حل حریصانه

در این مرحله، نوشتن یک الگوریتم جدولی و پایین به بالای برنامه سازی پویا، بر مبنای رابطه بازگشتی (۱۶.۳)، تمرین ساده‌ای خواهد بود. در حقیقت، تمرین ۱ - ۱۶.۱ از شما می‌خواهد تا این کار را انجام دهید. گرچه دو مشاهده کلیدی وجود دارند که با ما اجازه می‌دهند تا راه حل خود را ساده و آسان کنیم.

قضیه ۱۶.۱

زیرمسئله غیر تهی S_{ij} را در نظر بگیرید و a_m را فعالیتی در S_{ij} با زودترین زمان خاتمه فرض کنید:

$$f_m = min\{f_k : a_k \in S_{ij}\}$$

آنگاه

۱. فعالیت a_m در یک زیرمجموعه با ماکزیمم اندازه از فعالیت‌های متقابلاً سازگار از S_{ij} استفاده می‌شود.

۲. زیرمسئله S_{im} تهی است، لذا انتخاب a_m زیرمسئله S_{mj} را به عنوان تنها زیرمجموعه که ممکن است غیر تهی باشد، باقی می‌گذارد.

اثبات ابتدا قسمت دوم را اثبات می‌کنیم، زیرا کمی ساده‌تر است. فرض کنید S_{im} غیر تهی باشد لذا فعالیت a_k وجود دارد، بطوریکه $f_i \leq s_k < f_k \leq s_m < f_m$. پس a_k نیز در S_{ij} می‌باشد و زمان خاتمه زودتر از a_m دارد که انتخاب a_m را نقض می‌کند. نتیجه می‌گیریم که S_{im} تهی است.

برای اثبات قسمت اول، فرض می‌کنیم A_{ij} یک زیرمجموعه با ماکزیمم اندازه از فعالیت‌های متقابلاً سازگار از S_{ij} است، و فعالیت‌ها در A_{ij} را در یک ترتیب صعودی یکنواخت از زمانهای خاتمه مرتب می‌کنیم. a_k را اولین فعالیت در A_{ij} قرار دهید. اگر $a_k = a_m$ اثبات انجام می‌شود، زیرا نشان داده‌ایم که a_m در یک زیرمجموعه با ماکزیمم اندازه از فعالیت‌های متقابلاً سازگار از S_{ij} استفاده شده است. اگر $a_k \neq a_m$ زیرمجموعه $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$ را ایجاد می‌کنیم. فعالیت‌های داخل A'_{ij} جدا از هم (مجزا) می‌باشند، چون فعالیت‌های داخل A_{ij} جدا از هم هستند، a_k اولین فعالیت در A_{ij} است که خاتمه می‌یابد، و $f_m \leq f_k$ باتوجه با اینکه A'_{ij} همان تعداد فعالیت‌های مجموعه A_{ij} را دارا می‌باشد، می‌بینیم که A'_{ij} یک زیرمجموعه با ماکزیمم اندازه از فعالیت‌های متقابلاً سازگار از S_{ij} است که شامل a_m می‌باشد. ■

چرا قضیه ۱۶.۱ بسیار ارزشمند است؟ از بخش ۱۵.۳ بیاد آورید که زیرساختار بهینه بر اساس تعداد زیرمسئله‌هایی که در یک جواب بهینه برای مسئله اصلی استفاده می‌شوند و تعداد انتخابهایی که انجام می‌دهیم تا تعیین کنیم کدام زیرمسئله‌ها را بکار ببریم، تغییر می‌کند. در راه حل برنامه‌سازی پویا، دو زیرمسئله در جواب بهینه استفاده شده‌اند و $j-i-1$ انتخاب در هنگام حل زیرمسائل S_{ij} وجود دارد. قضیه ۱۶.۱ هر دوی این کمیت‌ها را به طور چشمگیری کاهش می‌دهد: تنها یک زیرمسئله در جواب بهینه استفاده می‌شود (تضمین می‌شود که زیرمسئله دیگر تهی باشد) و در هنگام حل زیرمسئله S_{ij} لازم است تنها یک انتخاب را در نظر بگیریم: انتخابی که دارای زودترین زمان خاتمه در S_{ij} است. خوشبختانه، می‌توانیم به آسانی مشخص کنیم که کدام فعالیت این چنین است.

علاوه بر کاهش تعداد زیرمسائل و تعداد انتخابها، قضیه ۱۶.۱ یک مزیت دیگر نیز دارد: می‌توانیم هر زیرمسئله را به شکل بالا به پایین حل کنیم، بجای روش پایین به بالا که معمولاً در برنامه‌سازی پویا استفاده می‌شود. برای حل زیرمسئله S_{ij} که فعالیت a_m در S_{ij} بازودترین زمان خاتمه را انتخاب می‌کنیم و مجموعه‌ای از فعالیت‌ها که در جواب بهینه زیرمسئله S_{mj} استفاده شده‌اند را به این جواب اضافه می‌کنیم. از آنجا که می‌دانیم با انتخاب a_m حتماً از یک جواب برای S_{mj} در جواب بهینه خود برای S_{ij} استفاده خواهیم کرد، لذا نیازی به حل S_{mj} قبل از S_{ij} نیست. برای حل S_{ij} که می‌توانیم ابتدا a_m را به عنوان

فعالیتی در S_{ij} بازودترین زمان خاتمه انتخاب کنیم و سپس S_{mj} را حل نماییم.

همچنین توجه کنید که یک الگو برای زیرمسائلی که حل می‌کنیم وجود دارد. مسئله اولیه (اصلی)، $S=S_{0,n+1}$ است. فرض کنید که a_{m1} را به عنوان فعالیتی در $S_{0,n+1}$ با زودترین زمان خاتمه انتخاب کنیم. (از آنجا که فعالیتها را به‌صورت صعودی یکنواخت بر مبنای زمانهای خاتمه مرتب کرده‌ایم و $f_0=0$ باید داشته باشیم $m1 = 1$) زیرمسئله بعدی، $s_{m_1,n+1}$ می‌باشد. حال فرض کنید که a_{m2} را به عنوان فعالیتی بازودترین زمان خاتمه از $S_{m_1,n+1}$ انتخاب کنیم. (لزومی ندارد که $m_2 = 2$ باشد.) زیرمسئله بعدی، $S_{m2,n+1}$ است. با ادامه این کار، می‌بینیم که هر زیرمسئله برای یک فعالیت با شماره m_i به شکل $S_{mi,n+1}$ خواهد بود. به بیان دیگر، هر زیرمسئله از آخرین فعالیت‌هایی که خاتمه می‌یابند و تعدادی از فعالیت‌هایی که از یک زیرمسئله به زیرمسئله دیگر تغییر می‌کنند، تشکیل شده است.

الگویی نیز برای فعالیت‌هایی که انتخاب می‌کنیم وجود دارد. چون همواره فعالیتی بازودترین زمان خاتمه در $S_{mi,n+1}$ را انتخاب می‌کنیم، زمانهای خاتمه فعالیت‌های انتخاب شده تمام زیرمسائل، در طول زمان اکیداً افزایش می‌یابند. علاوه بر این، می‌توانیم هر فعالیت را فقط یکبار در ترتیب صعودی یکنواخت از زمانهای خاتمه در نظر بگیریم.

فعالیت a_m که در هنگام حل یک زیرمسئله انتخاب می‌کنیم، همواره همان فعالیت بازودترین زمان خاتمه است که می‌تواند به طور مجاز زمان بندی شود. بنابراین فعالیتی که انتخاب می‌شود، یک انتخاب "حریصانه" می‌باشد، از این نظر که به طور شهودی، بیشترین فرصت ممکن را به فعالیت‌های باقیمانده می‌دهد تا زمان بندی شوند. به عبارت دیگر، انتخاب حریصانه انتخابی است که زمان باقیمانده زمان بندی نشده را ماکزیمم کند.

الگوریتم حریصانه بازگشتی

اکنون که دیدیم چگونه راه حل برنامه سازی پویای خود را کارآمدتر کنیم و چگونه به عنوان یک روش بالا به پایین با آن برخورد نماییم، آماده‌ایم تا یک الگوریتم را که به روش بالا به پایین کاملاً حریصانه عمل می‌کند مشاهده کنیم. یک راه حل بازگشتی مستقیم و ساده با عنوان روال *RECURSIVE-ACTIVITY-SELECTOR* را ارائه می‌کنیم. این روال، زمان‌های شروع و خاتمه فعالیت‌ها را، که بصورت آرایه‌های s و f نشان داده می‌شوند، بعلاوه اندیسهای شروع i و j از زیرمسئله S_{ij} را که باید حل کند، به عنوان ورودی می‌گیرد. این روال یک مجموعه با ماکزیمم اندازه از فعالیت‌های متقابلاً سازگار در S_{ij} را بر می‌گرداند. فرض می‌کنیم که n فعالیت ورودی بر مبنای زمانهای خاتمه مطابق با معادله (۱۶.۱) بصورت صعودی یکنواخت مرتب شده‌اند. در غیر اینصورت می‌توانیم با شکستن پیوندها به صورت اختیاری، آنها را با مرتبه زمانی $O(nlgn)$ در این ترتیب مرتب کنیم. فراخوانی اولیه روال‌بصورت *RECURSIVE-ACTIVITY-SELECTOR (s, f, 0, n+1)*

است.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, i, j)
1  m ← i + 1
2  while m < j and s_m < f_i      ▷ Find the first activity in S_ij.
3     do m ← m + 1
4  if m < j
5     then return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, j)
6     else return Ø
```

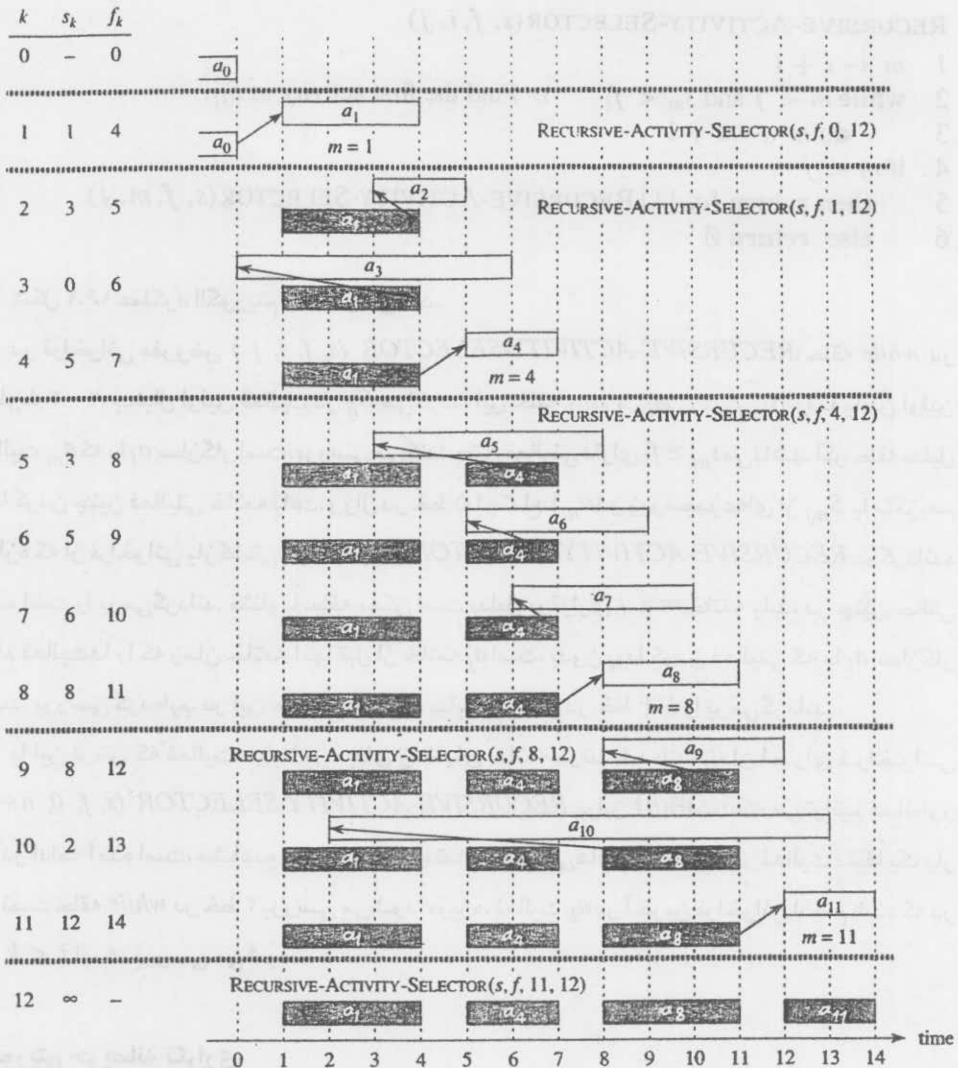
شکل ۱۶.۱ عملکرد الگوریتم را نشان می‌دهد.

در فراخوانی مفروض *RECURSIVE-ACTIVITY-SELECTOR (s, f, i, j)*، حلقه *while* در خطوط ۳ - ۲، بدنبال اولین فعالیت در S_{ij} می‌گردد. این حلقه $a_{i+1}, \dots a_{i+2}\ a_{j-1}$ را تا یافتن اولین فعالیت a_m که با a_i سازگار است، بررسی می‌کند؛ چنین فعالیتی دارای $s_m \geq f_i$ می‌باشد. اگر حلقه بدلیل پیدا کردن چنین فعالیتی خاتمه یافت، روال در خط ۵، اجتماع $\{a_m\}$ و زیرمجموعه‌ای از S_{mj} با ماکزیمم اندازه که از فراخوانی بازگشتی *RECURSIVE-ACTIVITY-SELECTOR (s, f, m, j)* برگردانده شده است را برمی‌گرداند. متناوباً حلقه ممکن است بدلیل برقراری $m \geq j$ خاتمه یابد، در چنین حالتی تمام فعالیت‌ها را که زمان خاتمه آنها قبل از خاتمه a_j است، بدون پیدا کردن فعالیتی که با a_i سازگار باشد، بررسی کرده‌ایم. در این حالت، $S_{ij} = \phi$، بنابراین روال در خط ۶، ϕ را برمی‌گرداند.

با این فرض که فعالیت‌ها قبلاً بر مبنای زمانهای خاتمه مرتب شده‌اند، زمان اجرای فراخوانی *RECURSIVE-ACTIVITY-SELECTOR (s, f, 0, n+1)* برابر $\Theta(n)$ است که می‌توانیم همانطور که در ادامه آمده است، مشاهده نماییم. در طول تمام فراخوانی‌های بازگشتی، هر فعالیت دقیقاً یک بار در تست حلقه *while* در خط ۲ بررسی می‌شود. بویژه، فعالیت a_k در آخرین فراخوانی انجام شده که در آن $i < k$ است، بررسی می‌شود.

الگوریتم حریصانه تکراری

براحتی می‌توانیم روال بازگشتی خود را به یک روال تکراری تبدیل نماییم. روال *RECURSIVE-ACTIVITY-SELECTOR* تقریباً "در انتها بازگشتی" است (مسئله ۴ - ۷ را ملاحظه نمایید): با یک فراخوانی بازگشتی از خودش که با یک عمل اجتماع همراه است، خاتمه می‌یابد. تبدیل یک روال در انتها - بازگشتی به شکل تکراری، معمولاً کار ساده‌ای است؛ در حقیقت برخی از کامپایلرهای زبانهای برنامه نویسی مشخص این کار را به طور خودکار انجام می‌دهند. همانطور که نوشته شده است، *RECURSIVE-ACTIVITY-SELECTOR* برای هر زیرمسئله S_{ij} کار می‌کند، اما مشاهده کردیم که لازم است تنها زیرمسائلی را در نظر بگیریم که برای آنها $j = n+1$، یعنی زیر مسائلی که از آخرین فعالیت‌هایی که خاتمه می‌یابند، تشکیل شده‌اند.



شکل ۱۶.۱ عملکرد *RECURSIVE-ACTIVITY-SELECTOR* بر روی *11* فعالیت داده شده بصورت فوق. فعالیت‌هایی که در هر فراخوانی بازگشتی در نظر گرفته می‌شوند، بین خطوط افقی ظاهر می‌گردند. فعالیت ساختگی a_0 در زمان *0* خاتمه می‌یابد، و در فراخوانی اولیهٔ *RECURSIVE-ACTIVITY-SELECTOR (s, f, 0, 12)*، فعالیت a_1 انتخاب می‌شود. در هر فراخوانی بازگشتی، فعالیت‌هایی که قبلاً انتخاب شده‌اند، سیاه شده‌اند و فعالیتی که بصورت سفید نشان داده شده است، در نظر گرفته می‌شود. اگر زمان شروع یک فعالیت قبل از زمان خاتمه جدیدترین فعالیت اضافه شده، باشد (پیکان بین آنها به چپ اشاره می‌کند)، این فعالیت رد می‌شود. در غیر اینصورت (پیکان مستقیماً به بالا یا به راست اشاره می‌کند)، این فعالیت انتخاب می‌شود. آخرین فراخوانی بازگشتی، یعنی *RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 12)*، ϕ را برمی‌گرداند. مجموعه حاصل از فعالیت‌های انتخاب شده، $\{a_1, a_4, a_8, a_{11}\}$ است.

روال *GREEDY-ACTIVITY-SELECTOR* نسخه تکراری روال *RECURSIVE-ACTIVITY-SELECTOR* می‌باشد. همچنین در این روال، فرض می‌شود که فعالیت‌های ورودی بر مبنای زمانهای خاتمه بصورت صعودی یکنواخت مرتب شده‌اند. این روال، فعالیت‌های انتخاب شده را در مجموعه A گردآوری کرده و در پایان، این مجموعه را بر می‌گرداند.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n ← length[s]
2  A ← {a_1}
3  i ← 1
4  for m ← 2 to n
5       do if s_m ≥ f_i
6             then A ← A ∪ {a_m}
7                  i ← m
8  return A
```

روال بصورت زیر کار می‌کند. متغیر i نشان دهنده آخرین فعالیت اضافه شده به مجموعه A، مطابق با فعالیت a_i در نسخه تکراری، می‌باشد. از آنجا که فعالیت‌ها بصورت صعودی یکنواخت بر مبنای زمانهای خاتمه در نظر گرفته می‌شوند، f_i همواره ماکزیمم زمان خاتمه هر فعالیت در A می‌باشد. به عبارت دیگر

$$f_i = max\{f_k : a_k \in A\} \quad (۱۶.۴)$$

در خطوط ۳ – ۲، فعالیت a_1 انتخاب شده و A مقدار دهی اولیه می‌شود، بطوریکه فقط شامل این فعالیت باشد، و i با اندیس این فعالیت، مقدار دهی اولیه می‌شود. حلقه *for* در خطوط ۷ – ۴، نخستین فعالیتی را که در $S_{i,n+1}$ خاتمه می‌یابد، پیدا می‌کند. این حلقه هر فعالیت a_m را به نوبت در نظر می‌گیرد و a_m را چنانچه با تمام فعالیت‌های انتخاب شده قبلی سازگار باشد، به A اضافه می‌کند؛ چنین فعالیتی، زودترین فعالیتی است که در $S_{i,n+1}$ خاتمه می‌یابد. برای آنکه ببینیم آیا فعالیت a_m با هر فعالیت فعلی در A سازگار است، کافی است بوسیله معادله (۱۶.۴)، بررسی کنیم (خط ۵) که زمان شروع آن، s_m قبل از زمان پایان آخرین فعالیت اضافه شده به A، یعنی f_i نباشد. اگر فعالیت a_m سازگار باشد، خطوط ۷ – ۶، فعالیت a_m را به A اضافه می‌کنند و i را با m مقدار دهی می‌کنند. مجموعه A که با فراخوانی *GREEDY-ACTIVITY-SELECTOR(s, f)* برگردانده می‌شود، دقیقاً همان مجموعه برگردانده شده با فراخوانی *RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n+1)* است.

همانند نسخه بازگشتی، *GREEDY-ACTIVITY-SELECTOR* یک مجموعه با n فعالیت را در زمان $\Theta(n)$ زمانبندی می‌کند، با این فرض که فعالیت‌ها قبلاً بر مبنای زمانهای خاتمه خود مرتب شده‌اند.

تمرین‌ها

۱ - ۱۶.۱ یک الگوریتم برنامه سازی پویا برای مسئله انتخاب فعالیت بر مبنای رابطه بازگشتی (۱۶.۳) ارائه نمایید. الگوریتم شما باید اندازه $c[i, j]$ را همانطور که در بالا تعریف شده است، محاسبه نماید و همچنین زیرمجموعه A با ماکزیمم اندازه از فعالیت‌ها را تولید کند. فرض کنید که ورودیها بر مبنای معادله (۱۶.۱) مرتب شده‌اند. زمان اجرای راه حل خود را با زمان اجرای *GREEDY-ACTIVITY-SELECTOR* مقایسه کنید.

۲ - ۱۶.۱ فرض کنید بجای آنکه همواره اولین فعالیت که خاتمه می‌یابد را انتخاب کنیم، آخرین فعالیتی را که شروع می‌شود و با تمام فعالیت‌های انتخاب شده قبلی سازگار است، انتخاب کنیم. توضیح دهید که چگونه این روش، یک الگوریتم حریصانه است و ثابت کنید که یک جواب بهینه را حاصل می‌کند.

۳ - ۱۶.۱ فرض کنید یک مجموعه از فعالیت‌ها برای زمان بندی از میان تعداد زیادی تالارهای سخنرانی داریم. می‌خواهیم تمام فعالیت‌ها را با استفاده از کمترین تالارهای سخنرانی ممکن زمان‌بندی کنیم. یک الگوریتم حریصانه کارآمد ارائه نمایید تا تعیین کند کدام فعالیت باید کدام تالار سخنرانی را استفاده کند.

(این مسئله به عنوان ***مسئله رنگ‌آمیزی گراف بازه‌ای***[1] نیز شناخته می‌شود. می‌توانیم یک گراف بازه‌ای ایجاد کنیم که رئوس آن، فعالیت‌های داده شده می‌باشند و یالهای آن، فعالیت‌های ناسازگار را به هم وصل می‌کنند. کمترین تعداد رنگهای مورد نیاز جهت رنگ کردن تمام رأسها، به نحوی که هیچ دو رأس مجاور یکرنگ نباشند، مطابق است با پیدا کردن کمترین تالارهای سخنرانی لازم جهت زمان بندی تمام فعالیت‌ها.)

۴ - ۱۶.۱ هر روش حریصانه برای مسئله انتخاب فعالیت، دقیقاً یک مجموعه با ماکزیمم اندازه از فعالیت‌های متقابلاً سازگار را تولید نمی‌کند. مثالی ارائه نمایید تا نشان دهد روش انتخاب فعالیت با کمترین طول زمانی از میان آنهایی که با فعالیت‌های انتخاب شده قبلی سازگار هستند، بدرستی عمل نخواهد کرد. به طور مشابه برای روش انتخاب فعالیت سازگاری که با کمترین فعالیت‌های باقی مانده دیگر همپوشانی دارد و روش انتخاب فعالیت باقی مانده سازگار با نخستین زمان شروع، مثالی ارائه نمایید.

۱۶.۲ عناصر تدبیر حریصانه

الگوریتم حریصانه، جواب بهینه مسئله را با انجام یک توالی از انتخابها بدست می‌آورد. برای هر مرحله

---

1. intreval-graph coloring problem

تصمیم‌گیری در الگوریتم، انتخابی که در همان لحظه بهترین به نظر می‌رسد انجام می‌شود. این تدبیر مکاشفه‌ای همواره جواب بهینه را تولید نمی‌کند، اما همانطور که در مسئله انتخاب فعالیت دیدیم، در بعضی موارد موجب تولید جواب بهینه می‌گردد. این بخش در مورد برخی از ویژگی‌های عمومی روش حریصانه بحث می‌کند.

فرآیندی را که در بخش ۱۶.۱ جهت توسعه یک الگوریتم حریصانه دنبال کردیم، از روش معمول آن قدری پیچیده‌تر بود. گام‌های زیر را طی کردیم:

۱. تعیین زیرساختار بهینه مسئله.
۲. توسعه یک راه حل بازگشتی.
۳. اثبات آنکه در هر مرحله بازگشت، یکی از انتخابهای بهینه، انتخاب حریصانه است. بنابراین همواره مناسب است که انتخاب حریصانه انجام دهیم.
۴. نشان دادن آنکه همه زیرمسائل بجز یکی از آنها با انجام انتخاب حریصانه کاهش می‌یابند.
۵. توسعه یک الگوریتم بازگشتی که تدبیر حریصانه را پیاده سازی می‌کند.
۶. تبدیل الگوریتم بازگشتی به الگوریتم تکراری.

در طی این گام‌ها، بنیان برنامه سازی پویای یک الگوریتم حریصانه را با جزئیات مشاهده کردیم. هر چند در عمل، معمولاً گام‌های فوق را در هنگام طراحی یک الگوریتم حریصانه به طور کارآمد بکار می‌بریم. زیرساختار خود را با توجه به انتخاب حریصانه‌ای که تنها یک زیرمسئله را برای حل بهینه باقی می‌گذارد، توسعه می‌دهیم. برای مثال، در مسئله انتخاب فعالیت، ابتدا زیرمسئله S_{ij} را تعریف کردیم، به طوری که هر دو مقدار i و j تغییر می‌کردند. سپس متوجه شدیم که اگر همواره انتخاب حریصانه داشته باشیم، می‌توانیم زیرمسائل را به شکل $S_{i,n+1}$ محدود کنیم.

متناوباً می‌توانستیم زیرساختار بهینه خود را با یک انتخاب حریصانه در ذهن شکل دهیم. به عبارت دیگر، می‌توانستیم زیرنویس دوم را حذف کرده و زیر مسائل را به شکل $S_i = \{a_k \in S: f_i \leq s_k\}$ تعریف کنیم. سپس می‌توانستیم ثابت کنیم انتخاب حریصانه (اولین فعالیت a_m در S_i که خاتمه می‌یابد) که با جواب بهینه مجموعه S_m باقیمانده از فعالیت‌های سازگار ترکیب شده بود، یک جواب بهینه برای S_i به ما می‌دهد. به طور کلی‌تر، الگوریتم حریصانه را بر طبق سلسله مراحل زیر طراحی می‌کنیم:

۱. تبدیل مسئله بهینه سازی بنحوی که در آن یک انتخاب انجام می‌دهیم و آنرا بهمراه یک زیرمسئله برای حل باقی می‌گذاریم.
۲. اثبات آنکه همواره یک جواب بهینه برای مسئله اصلی وجود دارد که انتخاب حریصانه انجام می‌دهد، بنابراین انتخاب حریصانه همواره مناسب است.
۳. اثبات آنکه با انجام انتخاب حریصانه آنچه باقی می‌ماند، یک زیرمسئله با این ویژگی است که اگر جواب بهینه زیرمسئله را با یک انتخاب حریصانه که انجام داده‌ایم ترکیب کنیم، به یک جواب بهینه

برای مسئله اصلی می‌رسیم.

در بخش‌های بعدی این فصل این فرآیند مستقیم‌تر را بکار خواهیم برد. با این وجود در پس هر الگوریتم حریصانه، تقریباً همواره یک راه حل برنامه سازی پویای پر زحمت‌تر وجود دارد.

چگونه می‌توان گفت که یک الگوریتم حریصانه، مسئله بهینه سازی خاصی را حل خواهد کرد؟ به‌طور کل هیچ راهی وجود ندارد، اما ویژگی انتخاب حریصانه و زیرساختار بهینه، دو عنصر کلیدی هستند. اگر بتوانیم ثابت کنیم که مسئله این دو ویژگی را دارا می‌باشد، توسعه یک الگوریتم حریصانه برای آن بخوبی انجام می‌شود.

ویژگی انتخاب حریصانه

اولین عنصر کلیدی، *ویژگی انتخاب حریصانه*[1] است: یک جواب بهینه کلی بوسیله انتخاب بهینه محلی (حریصانه) می‌تواند به دست آید. به بیان دیگر وقتی در حال بررسی این هستیم که کدام انتخاب را انجام دهیم، انتخابی را انجام می‌دهیم که در مسئله جاری بهترین به نظر می‌رسد، بدون در نظر گرفتن نتایج زیرمسائل.

تفاوت الگوریتم‌های حریصانه با برنامه سازی پویا در اینجاست. در برنامه سازی پویا در هر مرحله یک انتخاب می‌کنیم، اما این انتخاب معمولاً به جواب‌های زیرمسائل بستگی دارد. در نتیجه، معمولاً مسائل برنامه سازی پویا را به روش پایین به بالا حل می‌کنیم، با پیشروی از زیرمسائل کوچکتر به زیرمسائل بزرگتر. در یک الگوریتم حریصانه، انتخابی را انجام می‌دهیم که در همان لحظه بهترین بنظر می‌رسد و سپس زیرمسئله‌ای را که پس از انتخاب بوجود آمده است، حل می‌کنیم. انتخابی که با الگوریتم حریصانه انجام می‌شود، ممکن است به انتخابهای قبلی بستگی داشته باشد، اما نمی‌تواند به انتخابهای بعدی یا جوابهای زیرمسائل بستگی داشته باشد. بنابراین بر خلاف برنامه‌سازی پویا، که زیرمسائل را از پایین به بالا حل می‌کند، تدبیر حریصانه معمولاً در یک روند بالا به پایین حرکت می‌کند، با یک انتخاب حریصانه پس از انتخاب دیگر و تبدیل هر نمونه مسئله داده شده به یک مسئله کوچکتر.

البته باید ثابت کنیم که انتخاب حریصانه در هر مرحله، به یک جواب بهینه کلی می‌انجامد، و این همان جایی است که ممکن است نیاز به هوشیاری داشته باشد. معمولاً همانطور که در قضیه ۱۶.۱ دیدیم این اثبات راه حل بهینه کلی برای یک زیرمسئله را بررسی می‌کند. سپس نشان می‌دهد که راه حل می‌تواند تغییر کند تا از انتخاب حریصانه استفاده نماید که منجر به یک زیرمسئله مشابه اما کوچکتر می‌شود.

ویژگی انتخاب حریصانه اغلب سبب کارآیی در انجام انتخاب در یک زیرمسئله می‌شود. برای مثال،

---

1. greedy-choice property

در مسئله انتخاب فعالیت، با این فرض که قبلاً فعالیت‌ها را در یک ترتیب صعودی یکنواخت برمبنای زمانهای خاتمه مرتب کرده‌ایم، لازم بود هر فعالیت را تنها یکبار بررسی کنیم. بیشتر اوقات با پیش‌پردازش ورودی یا با استفاده از یک ساختمان داده مناسب (اغلب یک صف اولویت)، می‌توانیم انتخابهای حریصانه سریع داشته باشیم، بنابراین به یک الگوریتم مؤثر و کارآمد خواهیم رسید.

زیرساختار بهینه

یک مسئله *زیرساختار بهینه*[1] دارد، اگر جواب بهینه مسئله در درون خود شامل جوابهای بهینه زیرمسائل باشد. این ویژگی یک عنصر کلیدی در ارزیابی اِعمال پذیری برنامه سازی پویا و همچنین الگوریتم‌های حریصانه است. به عنوان مثالی از زیرساختار بهینه، به خاطر آورید که چگونه در بخش ۱۶.۱ اثبات کردیم که اگر جواب بهینه زیرمسئله S_{ij} شامل فعالیت a_k باشد، باید شامل جواب‌های بهینه زیرمسائل S_{ik} و S_{kj} نیز باشد. با ارائه این زیر ساختار بهینه استدلال کردیم که اگر می‌دانستیم کدام فعالیت به عنوان a_k استفاده شود، می‌توانستیم یک جواب بهینه برای S_{ij} با انتخاب a_k از میان تمام فعالیت‌ها در جوابهای بهینه زیرمسائل S_{ik} و S_{kj} بسازیم. بر اساس این مشاهدات از زیرساختار بهینه، قادر شدیم تا رابطه بازگشتی (۱۶.۳) را که مقدار یک جواب بهینه را تعریف می‌کند، ابداع کنیم.

معمولاً وقتی زیرساختار بهینه را برای الگوریتم‌های حریصانه بکار می‌بریم، از یک روش مستقیم‌تر مربوط به آن استفاده می‌کنیم. همانطور که در بالا اشاره شد، از این مزیت برخورداریم که فرض کنیم با انتخاب حریصانه در مسئله اصلی به یک زیرمسئله رسیده‌ایم. تمام کاری که باید انجام دهیم آن است که ثابت کنیم جواب بهینه زیرمسئله که با یک انتخاب حریصانه که قبلاً صورت گرفته ترکیب می‌شود، یک جواب بهینه برای مسئله اصلی حاصل می‌کند. این طرح بطور ضمنی از استقرا در زیرمسائل برای اثبات آنکه انتخاب حریصانه در هر مرحله، جواب بهینه را تولید می‌کند استفاده می‌کند.

روش حریصانه در مقایسه با برنامه سازی پویا

از آنجا که ویژگی زیرساختار بهینه بوسیله هر دو روش حریصانه و برنامه سازی پویا مورد استفاده قرار می‌گیرد، ممکن است راه حل برنامه سازی پویای مسئله را، وقتی که راه حل حریصانه برای آن کافی است، بدست آوریم و یا ممکن است وقتی در حقیقت راه حل برنامه سازی پویا مورد نیاز است، اشتباهاً فکر کنیم که راه حل حریصانه بدرستی عمل می‌کند. برای روشن نمودن ریزه کاریهای بین این دو تکنیک اجازه دهید دو شکل متفاوت از یک مسئله بهینه سازی کلاسیک را بررسی نماییم.

---

1. *optimal substructure*

مسئله کوله پشتی [۱] *0-1* بدین شکل مطرح می‌شود: یک دزد در هنگام دزدی از یک مغازه n شیء پیدا می‌کند؛ iامین شیء، v_i دلار ارزش و w_i پوند وزن دارد، که v_i و w_i اعداد صحیح هستند. او می‌خواهد با ارزش‌ترین بار ممکن را بر دارد، اما بیشترین باری را که می‌تواند در کوله پشتی‌اش حمل کند، W پوند است که W مقداری صحیح است. چه اشیائی را باید بردارد؟ (این مسئله، مسئله کوله پشتی *0-1* نامیده می‌شود زیرا هر شیء تنها می‌تواند برداشته شود و یا رها شده و برداشته نشود؛ دزد نمی‌تواند کسری از یک شیء را بردارد و یا یک شیء را بیش از یک بار بردارد.)

در *مسئله کوله پشتی کسری* [۲]، وضعیت مشابه است، اما دزد می‌تواند کسری از اشیا را بردارد، بجای آنکه مجبور باشد یک انتخاب دودویی (*1–0*) برای هر شیء داشته باشد. شیء مفروض در مسئله کوله پشتی *0-1* می‌تواند شمش طلا و در مسئله کوله پشتی کسری می‌تواند گرد طلا تصور شود.

هر دو مسئله، ویژگی زیرساختار بهینه را از خود بروز می‌دهند. در مسئله *0-1*، با ارزش‌ترین باری را که حداکثر W پوند وزن دارد در نظر بگیرید. اگر شیء j را از این بار برداریم، بار باقیمانده با ارزش‌ترین باری به وزن $W\text{-}w_j$ است که دزد می‌تواند از میان $n\text{-}1$ شیء اولیه به استثنای j بردارد. برای مسئله کسری مشابه، در نظر بگیرید که اگر وزنی برابر w مربوط به شیء j را از بار بهینه برداریم، بار باقیمانده با ارزش‌ترین باری به وزن $W\text{-}w$ است که دزد می‌تواند از میان $n\text{-}1$ شیء اولیه بعلاوه $w_j\text{-} w$ پوند از شیء j بردارد.

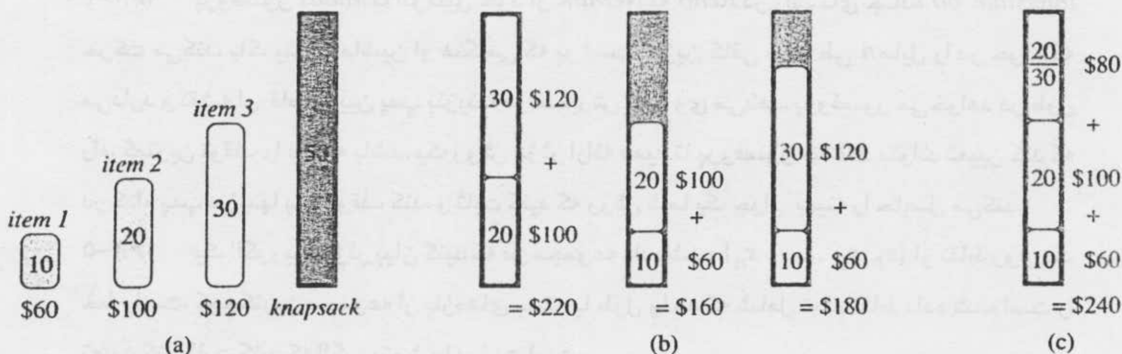
با اینکه مسائل مشابه‌اند، مسئله کوله پشتی کسری با روش حریصانه قابل حل می‌باشد، در حالیکه مسئله *0-1* قابل حل با روش حریصانه نیست. برای حل مسئله کوله پشتی کسری ابتدا ارزش واحد پوند v_i/w_i هر شیء را حساب می‌کنیم. با بکارگیری تدبیر حریصانه، دزد با برداشتن بیشترین مقدار ممکن از شیء با بیشترین ارزش واحد پوند، شروع می‌کند. اگر مقدار این شیء تمام شود و او هنوز بتواند بیشتر از آن حمل کند، بیشترین مقدار ممکن از شیء بعدی با بیشترین ارزش واحد پوند را بر می‌دارد و به همین شکل ادامه می‌دهد تا آنکه نتواند بیشتر از آن حمل کند. بنابراین با مرتب سازی اشیاء براساس ارزش واحد پوند، الگوریتم حریصانه با مرتبه زمانی $O(nlgn)$ اجرا می‌شود. اثبات آنکه مسئله کوله‌پشتی کسری دارای ویژگی انتخاب حریصانه است به عنوان تمرین ۱–۱۶.۲ واگذار می‌شود.

برای مشاهده آنکه روش حریصانه در مسئله کوله پشتی *1–0* قابل اعمال نیست، نمونه مسئله شرح داده شده در شکل ۱۶.۲(a) را در نظر بگیرید. سه شیء وجود دارند و کوله پشتی می‌تواند *50* پوند را در خود نگه دارد. شیء *1*، *10* پوند وزن و *60* دلار ارزش دارد. شیء *2*، *20* پوند وزن و *100* دلار ارزش دارد. شیء *3*، *30* پوند وزن و *120* دلار ارزش دارد. بنابراین ارزش واحد پوند شیء *1*، *6* دلار است که بیشتر از ارزش واحد پوند شیء *2* (*5* دلار) و یا شیء *3* (*4* دلار) است. در نتیجه تدبیر حریصانه ابتدا شیء *1* را برمی‌گزیند. اما همانطور که از تجزیه و تحلیل شکل ۱۶.۲(b) مشاهده می‌شود، جواب

---

1. *0-1 knapsack problem*
2. *fractional knapsack problem*

بهینه اشیاء 2 و 3 را برمی‌گزیند و شیء 1 را باقی می‌گذارد. دو جواب ممکن که شامل شیء 1 هستند، غیربهینه می‌باشند.



شکل ۱۶.۲ تدبیر حریصانه برای مسئله کوله پشتی 0-1 قابل اعمال نیست. (a) دزد باید یک زیرمجموعه از سه شیء نشان داده شده را انتخاب کند، به طوری که وزن آن از 50 پوند تجاوز نکند. (b) زیرمجموعه بهینه شامل اشیاء 2 و 3 است. هر جواب با شیء 1 غیربهینه خواهد بود، حتی با وجود آنکه شیء 1 بیشترین ارزش واحد پوند را دارا می‌باشد. (c) در مسئله کوله پشتی کسری، برداشتن اشیا بترتیب بیشترین ارزش واحد پوند به یک جواب بهینه می‌انجامد.

در مسئله کسری مشابه، تدبیر حریصانه با وجود آنکه شیء 1 را برمی‌گزیند، یک جواب بهینه را، همانطور که در شکل ۱۶.۲(c) نشان داده شده است، حاصل می‌کند. گزینش شیء 1 در مسئله 0-1 عمل نمی‌کند، زیرا دزد نمی‌تواند ظرفیت کوله پشتی‌اش را تکمیل کند و فضای خالی، ارزش واحد پوند مؤثر این بار را کم می‌کند. در مسئله 0-1، هنگامی که یک شیء را برای قرار گرفتن در کوله پشتی درنظر می‌گیریم، قبل از آنکه بتوانیم انتخابی را انجام دهیم، باید جواب زیرمسئله‌ای که در آن شیء وجود دارد را با جواب زیرمسئله‌ای که در آن شیء وجود ندارد مقایسه کنیم. مسئله‌ای که بدین گونه فرموله شود، تعداد زیادی زیرمسائل متداخل بوجود می‌آورد - ویژگی عمده برنامه‌سازی پویا، و در واقع برنامه سازی پویا می‌تواند برای حل مسئله 0-1 استفاده شود. (تمرین ۲ - ۱۶.۲ را ملاحظه نمایید.)

تمرین‌ها

۱-۱۶.۲ ثابت کنید که مسئله کوله پشتی کسری، ویژگی انتخاب حریصانه را دارا می‌باشد.

۲-۱۶.۲ یک راه حل برنامه سازی پویا برای مسئله کوله پشتی 0-1 ارائه دهید که در زمان $O(nW)$ اجرا شود، بطوریکه n تعداد اشیاء و W، ماکزیمم وزن اشیائی است که دزد می‌تواند در کوله‌پشتی‌اش قرار دهد.

۳-۱۶.۲ فرض کنید در مسئله کوله پشتی 0-1، ترتیب اشیاء وقتی که بر اساس افزایش وزن مرتب

شوند همانند زمانی باشد که بر اساس کاهش ارزش مرتب شده‌اند. یک الگوریتم کارآمد برای پیدا کردن جواب بهینه این مسئله ارائه نمایید، و ثابت کنید که الگوریتم شما درست است.

۱۶.۲-۴ پروفسور *Midas* با اتومبیل خود از *Newark* به *Reno* در راستای جاده *Interstate 80* حرکت می‌کند. باک بنزین ماشین او هنگامی که پر است، بنزین کافی برای طی n مایل را در خود نگه می‌دارد و نقشه او، فاصله بین پمپ بنزینها در مسیرش را به وی می‌دهد. پروفسور می‌خواهد در طول راه، کمترین توقف را داشته باشد. یک روش مؤثر ارائه دهید تا پروفسور *Midas* بتواند تعیین کند که در کدام پمپ بنزینها باید توقف کند، و ثابت کنید که روش شما یک جواب بهینه را حاصل می‌کند.

۱۶.۲-۵ یک الگوریتم مؤثر بیان کنید که در مجموعه داده شده $\{x_1, x_2, \ldots, x_n\}$ از نقاط روی یک خط راست، کوچکترین مجموعه از بازه‌های بسته با طول واحد که شامل تمام نقاط داده شده است را تعیین کند. ثابت کنید که الگوریتم شما درست است.

۱۶.۲-۶* نشان دهید که چگونه مسئله کوله پشتی کسری را در زمان $O(n)$ حل کنیم. فرض کنید حل مسئله ۲ - ۹ را دارید.

۱۶.۲-۷ فرض کنید به شما دو مجموعه A و B داده شده است که هر کدام شامل n عدد صحیح مثبت می‌باشند. می‌توانید هر طور که می‌خواهید، اعضاءِ هر مجموعه را مجدداً مرتب کنید. پس از مرتب‌سازی مجدد، a_i را به عنوان iامین عضو مجموعه A و b_i را به عنوان iامین عضو مجموعه B در نظر بگیرید. آنگاه پاداش $\prod_{i=1}^{n} a_i^{b_i}$ را دریافت می‌کنید. الگوریتمی ارائه دهید که این پاداش را ماکزیمم نماید. ثابت کنید که الگوریتم شما، پاداش را ماکزیمم می‌کند و زمان اجرای آنرا تعیین کنید.

۱۶.۳ کد *Huffman*

کد *Huffman* (کد هافمن) یک تکنیک بسیار مؤثر و بسیار پر کاربرد جهت فشرده سازی داده‌ها است؛ که به طور معمول *20* تا *90* درصد، بسته به خصوصیات داده‌هایی که فشرده می‌شوند، سبب صرفه جویی در حافظه می‌شود. داده‌ها را بصورت یک رشته از کاراکترها در نظر می‌گیریم. الگوریتم حریصانه *Huffman* از جدول تعداد تکرار کاراکترها استفاده می‌کند تا یک راه بهینه برای نمایش هر کاراکتر بصورت یک رشته دودویی ایجاد نماید.

فرض کنید یک فایل داده *100,000* کاراکتری داریم که می‌خواهیم آنرا بصورت فشرده ذخیره کنیم. مشاهده می‌کنیم که کاراکترها در فایل، به تعداد تکرار داده شده در شکل ۱۶.۳ تکرار می‌گردند. به عبارت دیگر، تنها شش کاراکتر مختلف وجود دارند و کاراکتر *a*، *45,000* بار تکرار می‌گردد.

راه‌های بسیاری جهت نمایش چنین فایل اطلاعاتی وجود دارد. مسئله طراحی یک کد ***کاراکتری***

دودویی[1] یا به اختصار کد[2] را در نظر می‌گیریم که در آن هر کاراکتر با یک رشته دودویی منحصر بفرد نشان داده می‌شود. اگر از یک *کد با طول ثابت*[3] استفاده کنیم، *3* بیت جهت نشان دادن *6* کاراکتر نیاز داریم، $a = 000$، $b = 001$، ...، $f = 101$. این روش، *300,000* بیت جهت کد کردن تمام فایل نیاز دارد. آیا می‌توانیم بهتر از این عمل کنیم؟

کد با طول متغیر[4] می‌تواند به طور قابل ملاحظه‌ای بهتر از کد با طول ثابت باشد، بدین شکل که به کاراکترهای با تکرار زیاد، کلمهٔ کد کوتاه و به کاراکترهای با تکرار کم، کلمهٔ کد بلند اختصاص می‌دهیم. شکل ۱۶.۳ چنین کدی را نشان می‌دهد؛ رشته *1*-بیتی *0*، نشان دهنده *a* و رشته ۴-بیتی *1100*، نشان دهنده *f* است. این کد به

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4)\ 1{,}000 = 224{,}000$$

بیت جهت نمایش فایل نیاز دارد، که تقریباً *25٪* سبب صرفه جویی می‌شود. در حقیقت همانطور که بزودی خواهیم دید، این کد یک کد کاراکتری بهینه برای این فایل است.

کدهای پیشوندی

در اینجا تنها کدهایی را بررسی می‌کنیم که در آن‌ها، یک کلمه‌کد، پیشوندی برای کلمه‌کد دیگر نیست. چنین کدهایی، *کدهای پیشوندی*[5] نام دارند. می‌توان نشان داد (اگر چه در اینجا این کار را نخواهیم کرد) که فشرده‌سازی بهینه داده‌ها که با کد کاراکتری قابل انجام است، می‌تواند همواره با یک کد پیشوندی انجام شود، بنابراین معطوف کردن توجه به کدهای پیشوندی هیچ ضرری به کلیت موضوع نخواهد زد.

کد گذاری برای هر کد کاراکتری دودویی همواره ساده است؛ تنها کلمه‌کدهایی که هر کاراکتر فایل را نشان می‌دهند، بصورت زنجیره‌ای بهم متصل می‌کنیم. برای مثال، با کد پیشوندی با طول متغیر در شکل ۱۶.۳، فایل *3* کاراکتری *abc* را بصورت *0101100 = 0.101.100* کدگذاری می‌کنیم، که از «.» جهت دلالت بر زنجیره بودن، استفاده می‌کنیم.

کدهای پیشوندی مطلوب هستند، زیرا کد گشایی را آسان می‌کنند. از آنجا که هیچ کلمهٔ کدی، پیشوند کلمه کد دیگر نیست، کلمه‌کدی که یک فایل کدگذاری شده را آغاز می‌کند غیرمبهم و مشخص است. می‌توانیم به سادگی کلمه‌کد اول را مشخص کرده، آنرا به کاراکتر اولیه ترجمه کرده و این روند کد گشایی را برای باقیمانده فایل کدگذاری شده تکرار کنیم. در مثال ما، رشته *001011101* منحصراً بصورت *0.0.101.1011* تجزیه می‌شود که به *aabe* کدگشایی می‌شود.

---

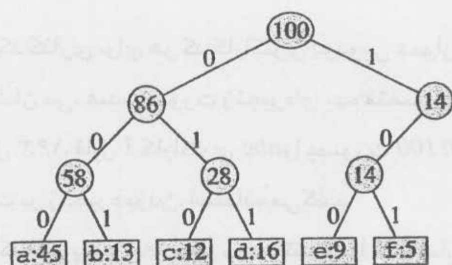
1. *binary character code*
2. *code*
3. *fixed-length code*
4. *variable-length code*

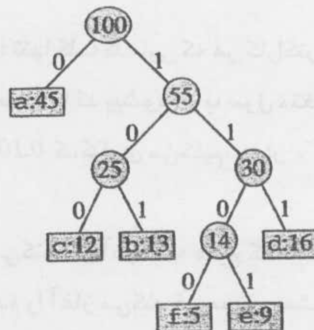
۵ - شاید «کدهای بدون پیشوند» نام بهتری باشد، اما اصطلاح «کدهای پیشوندی» در متون علمی استاندارد است.

	a	*b*	*c*	*d*	*e*	*f*
تعداد تکرار (به هزار)	45	13	12	16	9	5
کلمه‌کد با طول ثابت	000	001	010	011	100	101
کلمه‌کد با طول متغیر	0	101	100	111	1101	1100

شکل ۱۶.۳ مسئله کدگذاری کاراکترها. یک فایل داده *100,000* کاراکتری تنها شامل کاراکترهای *a-f* با تعداد تکرار نشان داده شده در فوق، می‌باشد. اگر به هر کاراکتر یک کلمه کد *3*- بیتی اختصاص داده شود، فایل می‌تواند با *300,000* بیت کدگذاری شود. با استفاده از کد با طول متغیر نشان داده شده، فایل می‌تواند با *224,000* بیت کدگذاری شود.

روند کد گشایی به یک نمایش مناسب برای کد پیشوندی نیاز دارد تا کلمه‌کد اولیه بتواند براحتی بدست آید. یک درخت دودویی که برگهایش کاراکترهای داده شده می‌باشند، چنین نمایشی را فراهم می‌کند. کلمه کد دودویی برای هر کاراکتر را بصورت مسیری از ریشه تا آن کاراکتر تعبیر می‌کنیم، به‌طوری که *0* به معنای "رفتن به فرزند چپ" و *1* به معنای "رفتن به فرزند راست" است. شکل ۱۶.۴ درختهای مربوط به دو کد مثال ما را نشان می‌دهد. توجه کنید که این درخت‌ها، درخت‌های جستجوی دودویی نیستند، زیرا نیازی نیست که برگها در یک ترتیب مرتب شده ظاهر شوند و گره‌های داخلی شامل کلیدهای کاراکتری نیستند.





شکل ۱۶.۴ درخت‌های متناظر با طرحهای کدگذاری در شکل ۱۶.۳. هر برگ با یک کاراکتر و تعداد تکرار آن برچسب‌گذاری شده است. هر گره داخلی با مجموع تعداد تکرارهای برگهای زیردرختش برچسب‌گذاری شده است. *(a)* درخت متناظر با کد با طول ثابت *a = 000*، . . . ، *f = 101*. *(b)* درخت متناظر با کد پیشوندی بهینه *a = 0*، *b = 101*، . . . ، *f=1100*

کد بهینه برای یک فایل همواره با یک درخت دودویی پُر نمایش داده می‌شود که در آن هر گره غیر برگ، دو فرزند دارد (تمرین ۱ - ۱۶.۳ را ملاحظه نمایید). کد با طول ثابت در مثال ما بهینه نیست، زیرا درخت آن که در شکل ۱۶.۴(a) نشان داده شده است، یک درخت دودویی نیست: کلمه‌کدهایی وجود دارند که با... 10 آغاز می‌شوند، اما هیچ کلمه‌کدی با... 11 شروع نمی‌شود. از آنجا که اکنون می‌توانیم توجه خود را معطوف درخت‌های دودویی پر کنیم، می‌توانیم بگوییم اگر C الفبایی باشد که کاراکترها از آن برداشته می‌شوند و تمام تکرارهای کاراکترها مثبت باشند، آنگاه درخت مربوط به کد پیشوندی بهینه دقیقاً $|C|$ برگ دارد، یک برگ برای هر حرف از الفبا و دقیقاً $|C|-1$ گره داخلی دارد.

با دریافت درخت T متناظر با کد پیشوندی، محاسبه تعداد بیت‌های مورد نیاز جهت کدگذاری یک فایل، کار ساده‌ای است. برای هر کاراکتر c در الفبای C، $f(c)$ به تعداد تکرار c در فایل و $d_T(c)$ به عمق برگ c در درخت اشاره می‌کند. توجه کنید که $d_T(c)$ برابر طول کلمه‌کد مربوط به کاراکتر c نیز می‌باشد. بنابراین تعداد بیت‌های مورد نیاز جهت کدگذاری فایل برابر است با

$$B(T) = \sum_{c \in C} f(c) d_T(c) \,, \qquad (۱۶.۵)$$

که آنرا به عنوان هزینه[1] درخت T تعریف می‌کنیم.

ساخت کد *Huffman*

Huffman الگوریتمی حریصانه ابداع کرد که یک کد پیشوندی بهینه به نام کد *Huffman*[2] می‌سازد. با توجه به مشاهدات خود در بخش ۱۶.۲، اثبات درستی این مطلب متکی بر ویژگی انتخاب حریصانه و زیرساختار بهینه است. بجای آنکه ثابت کنیم این ویژگی‌ها بر قرار هستند و سپس شبه کد را توسعه دهیم، ابتدا شبه کد را بیان می‌کنیم. انجام این کار کمک می‌کند تا توضیح دهیم چگونه الگوریتم، انتخابهای حریصانه انجام می‌دهد.

در شبه کدی که در ادامه می‌آید، فرض می‌کنیم C یک مجموعه از n کاراکتر است و هر کاراکتر $c \in C$، یک شیء با تعداد تکرار تعریف شده $f[c]$ است. الگوریتم، درخت T متناظر با کد بهینه را به روش پایین به بالا می‌سازد. این الگوریتم با یک مجموعه از $|C|$ برگ شروع کرده و یک توالی از $|C|-1$ عمل "ادغام" را جهت ایجاد درخت نهایی اجرا می‌کند. صف مینیمم اولویت Q، که با f مقدار دهی شده است، جهت مشخص کردن دو شیء با کمترین تعداد تکرار به منظور ادغام با یکدیگر بکار گرفته می‌شود. نتیجه ادغام دو شیء، یک شیء جدید است که تعداد تکرار آن، مجموع تعداد تکرارهای دو شیئی است که با یکدیگر ادغام شده‌اند.

---
1. cost
2. Huffman code

```
HUFFMAN(C)
1  n ← |C|
2  Q ← C
3  for i ← 1 to n − 1
4      do allocate a new node z
5         left[z] ← x ← EXTRACT-MIN(Q)
6         right[z] ← y ← EXTRACT-MIN(Q)
7         f[z] ← f[x] + f[y]
8         INSERT(Q, z)
9  return EXTRACT-MIN(Q)          ▷ Return the root of the tree.
```

برای مثال ما، الگوریتم *Huffman* بصورت نشان داده شده در شکل ۱۶.۵ پیش می‌رود. از آنجا که *6* حرف در الفبا وجود دارد، اندازه اولیه صف برابر $n = 6$ است و *5* مرحله ادغام جهت ساخت درخت لازم است. درخت نهایی، کد پیشوندی بهینه را نشان می‌دهد. کلمه‌کد مربوط به هر حرف، توالی برچسب‌های یالها در مسیر ریشه به حرف می‌باشد.

در خط ۲، صف مینیمم اولویت Q با کاراکترهای داخل C مقدار دهی اولیه می‌شود. حلقه *for* در خطوط ۸ – ۳، به طور پی در پی دو گره x و y با کمترین تکرار را از صف خارج کرده و آن‌ها را در صف با یک گره جدید z که ادغام شده آنها را نشان می‌دهد، جایگزین می‌کند. تعداد تکرار z بصورت مجموع تعداد تکرارهای x و y در خط ۷ محاسبه می‌شود. گره z، x را به عنوان فرزند چپ و y را به عنوان فرزند راست خود دارا می‌باشد. (این ترتیب اختیاری است؛ جابجایی فرزندهای چپ و راست یک گره به کدی متفاوت با هزینه یکسان منجر خواهد شد.) پس از *n-1* ادغام، تنها گره باقیمانده در صف – ریشه درخت کد – در خط ۹ برگردانده می‌شود.

در تحلیل زمان اجرای الگوریتم *Huffman* فرض می‌شود که Q بصورت یک *min-heap* دودویی پیاده سازی می‌شود (فصل ۶ را ملاحظه نمایید). برای مجموعه C با n کاراکتر، مقدار دهی اولیه Q در خط ۲ می‌تواند با استفاده از روال *BUILD-MIN-HEAP* در بخش ۶.۳ در زمان *O(n)* انجام شود. حلقه *for* در خطوط ۸ – ۳، دقیقا *n-1* مرتبه اجرا می‌شود، و از آنجا که هر عمل روی *heap* زمان *O(lgn)* را نیاز دارد، حلقه سبب اضافه شدن زمان *O(nlgn)* به زمان اجرا می‌شود. بنابراین زمان کل اجرای *HUFFMAN* روی یک مجموعه n کاراکتری برابر *O(nlgn)* است.

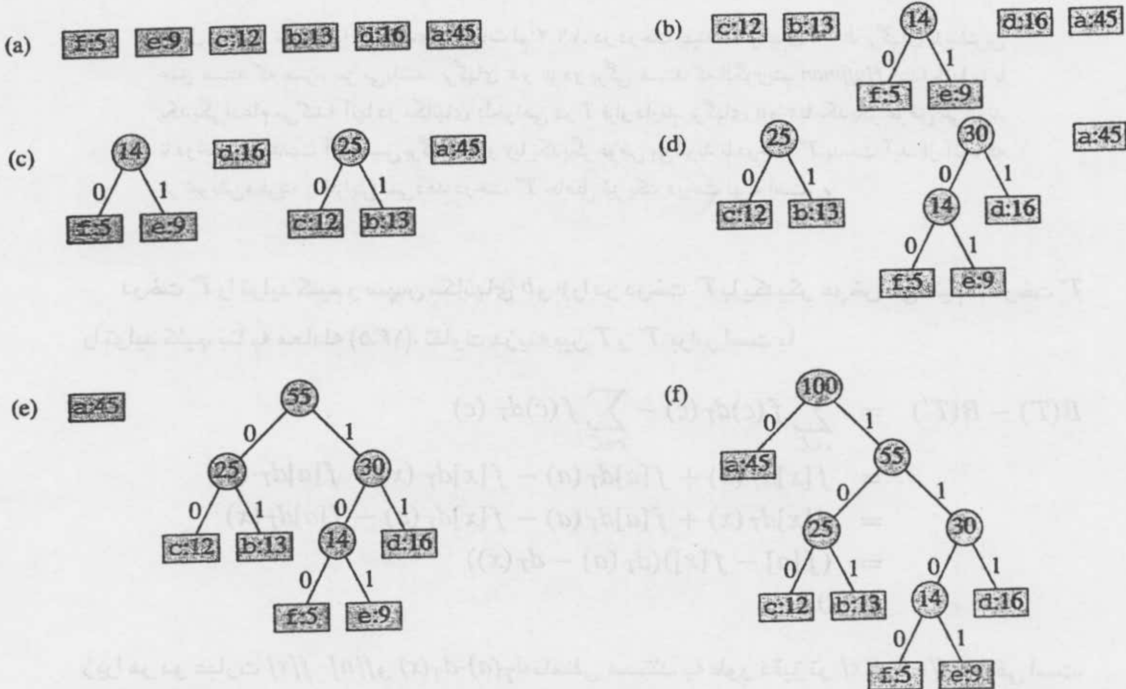
صحت الگوریتم *Huffman*

برای اثبات آنکه الگوریتم حریصانه *HUFFMAN* درست است، نشان می‌دهیم که مسئله تعیین کد پیشوندی بهینه ویژگی‌های انتخاب حریصانه و زیرساختار بهینه را دارا می‌باشد. لم بعد نشان می‌دهد که ویژگی انتخاب حریصانه برقرار است.

لم ۱۶.۲

فرض کنید C یک الفبا باشد که در آن هر کاراکتر $c \in C$ دارای تعداد تکرار $f[c]$ است. x و y را دو کاراکتر با کمترین تعداد تکرار در C در نظر بگیرید. آنگاه یک کد پیشوندی بهینه برای C وجود دارد که در آن کلمه‌کدهای مربوط به x و y، طول یکسانی دارند و تنها در آخرین بیت متفاوت می‌باشند.

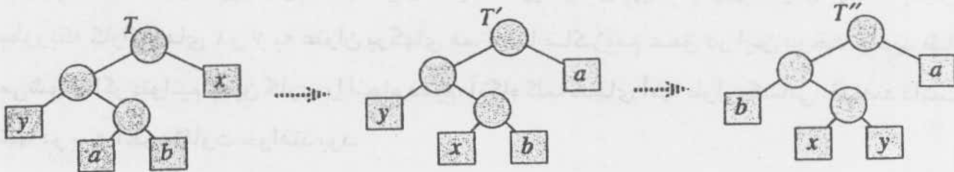
اثبات ایده اثبات آنست که درخت T را به عنوان نمایش دهنده یک کد پیشوندی بهینه دلخواه درنظر بگیریم و آنرا طوری تغییر دهیم تا درختی را که یک کد پیشوندی بهینه دیگر را نمایش می‌دهد بسازیم، بطوریکه کاراکترهای x و y به عنوان برگهای همزاد با ماکزیمم عمق در این درخت جدید ظاهر می‌شوند. اگر بتوانیم چنین کاری را انجام دهیم، آنگاه کلمه‌کدهای آنها طول یکسانی خواهند داشت و تنها در بیت آخر متفاوت خواهند بود.



شکل ۱۶.۵ مراحل الگوریتم *Huffman* برای تعداد تکرارهای داده شده در شکل ۱۶.۳. هر قسمت محتویات صف را که به صورت صعودی بر اساس تعداد تکرارها مرتب شده‌اند، نشان می‌دهد. در هر مرحله، دو درخت با کمترین تعداد تکرار ادغام می‌شوند. برگها بصورت مستطیل‌های حاوی یک کاراکتر و تعداد تکرار آن نمایش داده می‌شوند. گره‌های داخل بصورت دوایری حاوی مجموع تعداد تکرارهای فرزندانشان نشان داده می‌شوند. یالی که یک گره داخلی را به فرزندانش متصل می‌کند، چنانچه یالی به فرزند چپ باشد، با *0* و چنانچه یالی به فرزند راست باشد، با *1* برچسب گذاری می‌شود. کلمه کد برای هر حرف، توالی برچسب‌های یالهایی است که ریشه را به برگ آن حرف متصل می‌کنند. *(a)* مجموعه اولیه با $n = 6$ گره، یک گره برای هر حرف. *(b)-(e)* مراحل میانی. *(f)* درخت نهایی.

a و b را دو کاراکتر که برگهای همزاد با ماکزیمم عمق در T هستند، در نظر بگیرید. بدون از دست دادن کلیت، فرض می‌کنیم که $f[a] \le f[b]$ و $f[x] \le f[y]$ از آنجا که $f[x]$ و $f[y]$ بترتیب کمترین تعداد تکرار برگها هستند، و $f[a]$ و $f[b]$ بترتیب دو تعداد تکرار دلخواه هستند، داریم $f[x] \le f[a]$ و $f[y] \le f[b]$.

همانطور که در شکل ۱۶.۶ نشان داده شده است، مکانهای a و x را در درخت T عوض می‌کنیم تا



شکل ۱۶.۶ تشریح مراحل کلیدی در اثبات لم ۱۶.۲. در درخت بهینه T، برگهای a و b برگهای با بیشترین عمق هستند که همزاد نیز می‌باشند. برگهای x و y دو برگی هستند که الگوریتم *Huffman* ابتدا آنها را با یکدیگر ادغام می‌کند؛ آنها در مکانهای دلخواهی در T قرار دارند. برگهای a و x با یکدیگر عوض می‌شوند تا درخت T' بدست آید. سپس برگهای b و y با یکدیگر عوض می‌شوند تا درخت T'' بدست آید. از آنجا که هر تعویض، هزینه را افزایش نمی‌دهد درخت T'' حاصل نیز یک درخت بهینه است.

درخت T' را تولید کنیم و سپس مکانهای b و y را در درخت T' با یکدیگر عوض می‌کنیم تا درخت T'' را تولید کنیم. بنا به معادله (۱۶.۵)، تفاوت هزینه بین T و T' برابر است با

$$\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a) \\
&= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x) \\
&= (f[a] - f[x])(d_T(a) - d_T(x)) \\
&\ge 0 ,
\end{aligned}$$

زیرا هر دو عبارت $f[a]- f[x]$ و $d_T(a)-d_T(x)$ نامنفی هستند. به طور دقیق‌تر $f[a]- f[x]$ نامنفی است، چون x یک برگ با کمترین تعداد تکرار است و $d_T(a)-d_T(x)$ نامنفی است، چون a برگی با ماکزیمم عمق در T است. به طور مشابه، تعویض y و b هزینه را افزایش نمی‌دهد و بنابراین $B(T')-B(T'')$ نامنفی است. در نتیجه $B(T'') \le B(T)$. و از آنجا که T بهینه است، $B(T) \le B(T'')$ که دلالت می‌کند براینکه $B(T'') = B(T)$. بنابراین T'' یک درخت بهینه است که x و y در آن بصورت برگهای همزاد با ماکزیمم عمق ظاهر می‌شوند، که از این مطلب لم ثابت می‌شود. ■

لم ۱۶.۲ بر این دلالت می‌کند که فرآیند ساخت یک درخت بهینه بوسیله ادغام‌ها، بدون هیچ آسیبی به کلیت موضوع می‌تواند با انتخاب حریصانه ادغام دو کاراکتر با کمترین تعداد تکرار شروع شود. چرا

این انتخاب، حریصانه است؟ می‌توانیم هزینه یک ادغام را بصورت مجموع تعداد تکرارهای دو شیء ادغام شده در نظر بگیریم. تمرین ۳- ۱۶.۳ نشان می‌دهد که هزینه کل درخت ساخته شده برابر مجموع هزینه ادغام‌های آن است. از بین تمام ادغام‌های ممکن در هر مرحله، *HUFFMAN* ادغامی را انتخاب می‌کند که کمترین هزینه را موجب می‌شود.

لم بعد نشان می‌دهد که مسئله ساخت کدهای پیشوندی بهینه، ویژگی زیرساختار بهینه را دارد.

لم ۱۶.۳

C را یک الفبا با تعداد تکرار $f[c]$ برای هر کاراکتر $c \in C$ در نظر بگیرید. فرض کنید x و y دو کاراکتر در C با مینیمم تعداد تکرار باشند. C' را الفبای C که x و y از آن حذف شده و کاراکتر (جدید) z به آن اضافه شده است، در نظر بگیرید، بنابراین $C' = C-\{x, y\} \cup \{z\}$؛ f را برای C' همانند C تعریف کنید، به استثنای آنکه $f[z] = f[x] + f[y]$. T' را درخت نمایش دهنده کد پیشوندی بهینه برای الفبای C' درنظر بگیرید. آنگاه درخت T که از درخت T' با جایگزینی برگ z با یک گره داخلی که x و y فرزندان آن هستند بدست می‌آید، یک کد پیشوندی بهینه را برای الفبای C نمایش می‌دهد.

اثبات ابتدا نشان می‌دهیم که هزینه $B(T)$ درخت T می‌تواند برحسب هزینه $B(T')$ درخت T' با درنظر گرفتن هزینه‌های تشکیل دهنده آن در معادلهٔ (۱۶.۵) بیان گردد. برای هر $c \in C-\{x, y\}$ داریم $d_T(c)=d_{T'}(c)$، و از آنجا $f[c]d_T(c)=f[c]d_{T'}(c)$. از آنجا که $d_T(x) = d_T(y) = d_{T'}(z)+1$، داریم

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= f[z]d_{T'}(z) + (f[x] + f[y]) , \end{aligned}$$

که از آن نتیجه می‌گیریم که

$$B(T) = B(T') + f[x] + f[y]$$

یا به طور معادل

$$B(T') = B(T) - f[x] - f[y]$$

اکنون لم را با استفاده از برهان خلف اثبات می‌کنیم. فرض کنید T یک کد بهینه پیشوندی را برای C نمایش ندهد. پس درخت T'' وجود دارد، به طوری که $B(T'') < B(T)$. بدون هیچ آسیبی به کلیت (بنا به لم ۱۶.۲)، T'' دارای x و y بصورت همزاد است. T''' را همان درخت T'' در نظر بگیرید، بطوریکه پدر مشترک x و y با برگ z که تعداد تکرار آن $f[z] = f[x] + f[y]$ است، جایگزین شده است. پس

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \\ &= B(T') , \end{aligned}$$

که با فرض آنکه T' یک کد پیشوندی بهینه را برای C' نمایش می‌دهد، تناقض دارد. بنابراین T باید یک کد پیشون ، بهینه را ای الفبای C نمایش دهد. ■

نه ۴ ۱۶

وال ... ۲ HUF کد پیشوندی بهینه را تولید می‌کند.

اثبات مستقیماً از لم‌های ۱۶.۲ و ۱۶.۳ اثبات می‌شود. ■

تمرین‌ها

۱ - ۱۶.۳ ثابت کنید درخت دودویی که پر نیست، نمی‌تواند متناظر با یک کد پیشوندی بهینه باشد.

۲ - ۱۶.۳ کد *Huffman* بهینه برای مجموعه تعداد تکرارهای زیر که بر مبنای *8* عدد اول دنباله فیبوناچی می‌باشند، چیست؟

h:21 *g:13* *f:8* *e:5* *d:3* *c:2* *b:1* *a:1*

آیا می‌توانید پاسخ خود را برای یافتن کد بهینه، وقتی تعداد تکرارها n عدد اول دنباله فیبوناچی هستند، تعمیم دهید؟

۳ - ۱۶.۳ ثابت کنید هزینه کل یک درخت برای یک کد می‌تواند بصورت جمع تعداد تکرارهای ترکیب شده از دو فرزند تمام گره‌های داخلی نیز محاسبه شود.

۴ - ۱۶.۳ ثابت کنید اگر کاراکترهای داخل الفبا را طوری مرتب کنیم که تعداد تکرارهای آنها بصورت یکنواخت کاهش یابند، آنگاه یک کد بهینه وجود دارد که طولهای کلمه‌کدهای آن بصورت یکنواخت افزایش می‌یابند.

رض کنید یک کد پیشوندی بهینه روی مجموعه $C=\{0,1,\dots,n-1\}$ از کاراکترها داریم و م با استفاده از کمترین بیت‌های ممکن، این کد را ارسال کنیم. نشان دهید چگونه یک کد دی بهینه روی C را تنها با استفاده از $2n-1+\lceil n\lg n\rceil$ بیت نمایش دهیم. (راهنمایی: از $2n-1$ ت، همانطور که با یک پیمایش درخت فهمیده می‌شود، جهت نشان دادن ساختار درخت استفاده کنید.)

۶ - ۱۶.۳ الگوریتم *Huffman* را برای کلمه‌کدهای سه‌سه‌ای (مبنای *3*) تعمیم دهید (یعنی کلمه‌کدهایی که از نمادهای *0*، *1* و *2* استفاده می‌کنند)، و ثابت کنید یک کد سه‌سه‌ای بهینه را حاصل می‌کند.

۷ - ۱۶.۳ فرض کنید یک فایل داده شامل یک توالی از کاراکترهای *8* - بیتی است، به طوری که تقریباً تمام *256* کاراکتر استفاده شده‌اند: تعداد تکرار کاراکتر با ماکزیمم تکرار کمتر از دو برابر تکرار کاراکتر با مینیمم تکرار است. ثابت کنید کدگذاری *Huffman* در این حالت، کارآمدتر از استفاده از یک کد *8* - بیتی معمول با طول ثابت نیست.

۸ - ۱۶.۳ نشان دهید که از هیچ طرح فشرده سازی نمی‌توان انتظار داشت که یک فایل از کاراکترهای

8- بیتی که بصورت تصادفی انتخاب شده‌اند را حتی به اندازه یک بیت فشرده نماید. (راهنمایی: تعداد فایل‌ها را با تعداد فایل‌های کدگذاری شده ممکن مقایسه کنید.)

---

* ۱۶.۴ پایه‌های نظری روشهای حریصانه

یک تئوری جالب در مورد الگوریتم‌های حریصانه وجود دارد که آنرا بصورت خلاصه در این بخش بیان می‌کنیم. این تئوری در تعیین آنکه چه موقع روش حریصانه به جواب بهینه منجر خواهد شد، مفید است. این تئوری ساختارهای ترکیبی بنام "*matroid*"ها را دربرمی‌گیرد. اگر چه این تئوری تمام حالاتی را که برای آنها روش حریصانه بکار می‌رود، پوشش نمی‌دهد (برای مثال، مسئله انتخاب فعالیت از بخش ۱۶.۱ و یا مسئله کدگذاری *Huffman* از بخش ۱۶.۳ را پوشش نمی‌دهد)، اما بسیاری از حالتهایی را که استفاده عملی دارند، پوشش می‌دهد. علاوه بر آن، این تئوری به سرعت توسعه یافته و جهت پوشش کاربردهای بیشتری گسترش می‌یابد.

*Matroid*ها

matroid یک زوج مرتب $M = (S, \mathcal{I})$ است که در شرایط زیر صدق می‌کند.

۱. S یک مجموعه غیر تهی محدود است.

۲. $\mathcal{I}$ یک خانواده غیر تهی از زیرمجموعه‌های S است که زیرمجموعه‌های *مستقل*[1] S نامیده می‌شوند، به طوری که اگر $B \in \mathcal{I}$ و $A \in B$ آنگاه $A \in \mathcal{I}$. اگر $\mathcal{I}$ این ویژگی را داشته باشد، می‌گوییم *قابل ارث بری*[2] است. توجه کنید که مجموعه ϕ لزوماً یکی از اعضای $\mathcal{I}$ است.

۳. اگر $A \in \mathcal{I}$، $B \in \mathcal{I}$ و $|A| < |B|$، آنگاه عضو $x \in B\text{-}A$ وجود دارد، بطوریکه $A \cup \{x\} \in \mathcal{I}$ می‌گوییم M دارای *ویژگی تبادل*[3] است.

کلمه "*matroid*" بخاطر *Hassler Whitney* است. وی *matroid*های *ماتریسی*[4] را مطالعه می‌کرد که در آنها اعضاءِ S سطرهای یک ماتریس داده شده هستند و یک مجموعه از سطرها مستقل است، اگر سطرها به معنای معمول مستقل خطی باشند. نشان دادن اینکه این ساختار، یک *matroid* را تعریف می‌کند کار ساده‌ای است (تمرین ۲ - ۱۶.۴ را ملاحظه نمایید).

بعنوان نمونه‌ای دیگر از *matroid*ها، *matroid* *گرافی*[5] $M_G = (S_G, \mathcal{I}_G)$ را در نظر بگیرید که برحسب گراف بدون جهت $G = (V, E)$ بصورت زیر تعریف می‌شود.

- مجموعه S_G برابر E که مجموعه یالهای G است، تعریف می‌شود.

---

1. *independent*
2. *hereditary*
3. *exchange property*
4. *matric matroids*
5. *graphic matroid*

- اگر A یک زیرمجموعه از E باشد، آنگاه $A \in \mathcal{I}_G$ اگر و تنها اگر A بدون دور باشد. به عبارت دیگر، مجموعه A از یالها مستقل است، اگر و تنها اگر زیرگراف $G_A = (V, A)$ یک جنگل را تشکیل دهد.

matroid گرافی M_G به شکلی تنگاتنگ به مسئله درخت پوشای مینیمم مربوط است که در فصل ۲۳ با جزئیات بررسی شده است.

قضیه ۱۶.۵

اگر $G = (V, E)$ یک گراف بدون جهت باشد، آنگاه $M_G = (S_G, \mathcal{I}_G)$ یک *matroid* است.

اثبات به وضوح $S_G = E$ یک مجموعه محدود است. از سوی دیگر $\mathcal{I}_G$ قابل ارث بری است، زیرا یک زیرمجموعه از یک جنگل، یک جنگل است. به بیان دیگر، حذف یالها از یک مجموعه از یالهای بدون دور، نمی‌تواند دور ایجاد کند.

بنابراین تنها باقی می‌ماند که نشان دهیم M_G ویژگی تبادل را دارا است. فرض کنید $G_A = (V, A)$ و $G_B = (V, B)$ جنگل‌هایی از G هستند و آنکه $|B| > |A|$. به عبارت دیگر، A و B مجموعه‌های بدون دور از یالها هستند و B یالهای بیشتری از A دارد.

اگر یک جنگل k یال داشته باشد، دقیقاً $|V|-k$ درخت دارد. (برای اثبات این مطلب، با $|V|$ درخت شروع کنید که هر کدام از یک رأس تشکیل شده‌اند و هیچ یالی ندارند. آنگاه هر یالی که به جنگل اضافه می‌شود، تعداد درخت‌ها را یک واحد کاهش می‌دهد.) بنابراین جنگل G_A شامل $|V|-|A|$ درخت و جنگل G_B شامل $|V|-|B|$ درخت است.

از آنجا که جنگل G_B درخت‌های کمتری نسبت به جنگل G_A دارد، جنگل G_B باید شامل یک درخت T باشد که رئوس آن در دو درخت متفاوت در جنگل G_A هستند. علاوه بر این، از آنجا که T همبند است، باید شامل یک یال (u, v) باشد بطوریکه رئوس u و v در درخت‌های متفاوتی در جنگل G_A باشند. از آنجا که یال (u, v) دو رأس در دو درخت متفاوت در جنگل G_A را به هم وصل می‌کند، یال (u, v) می‌تواند بدون ایجاد دور به جنگل G_A اضافه گردد. بنابراین M_G دارای ویژگی تبادل است، که اثبات *matroid* بودن M_G کامل می‌شود. ■

در *matroid* مفروض $M = (S, \mathcal{I})$، عضو $x \notin A$ را یک توسعه[1] از $A \in \mathcal{I}$ گوییم، اگر x بتواند به A اضافه گردد، بطوریکه A استقلال خود را حفظ کند؛ به عبارت دیگر x یک توسعه از A است، اگر $A \cup \{x\} \in \mathcal{I}$. بعنوان مثال، *matroid* گرافی M_G را در نظر بگیرید. اگر A یک مجموعه مستقل از یالها باشد، آنگاه یال e یک توسعه از A است، اگر و تنها اگر e در A نباشد و افزودن e به A سبب تولید دور نگردد.

---

1. *extension*

اگر A یک زیرمجموعه مستقل در *matroid* مفروض M باشد، می‌گوییم A **ماکزیمال**[1] است، اگر هیچ توسعه‌ای نداشته باشد. به عبارت دیگر، A ماکزیمال است اگر در هیچ زیرمجموعه‌ای از M که مستقل و بزرگتر است قرار نگیرد. ویژگی بعد اغلب مفید است.

قضیه ١٦.٦

تمام زیرمجموعه‌های مستقل ماکزیمال در یک *matroid* اندازه یکسانی دارند.

اثبات فرض کنید بر خلاف آنچه در قضیه گفته شده است، A یک زیرمجموعه مستقل ماکزیمال از M باشد و زیرمجموعه مستقل بزرگتر دیگری از M بنام B وجود داشته باشد. آنگاه ویژگی تبادل دلالت می‌کند بر اینکه به ازای یک $x \in B\text{-}A$، A به یک مجموعه مستقل بزرگتر $A \cup \{x\}$ توسعه‌پذیر است، که با فرض ماکزیمال بودن A تناقض دارد. ■

بعنوان توضیح این قضیه، *matroid* گرافی M_G برای یک گراف بدون جهت و همبند را درنظر بگیرید. هر زیرمجموعه مستقل ماکزیمال از M_G باید یک درخت آزاد با دقیقاً $|V|-1$ یال باشد که تمام رأسهای G را به هم وصل می‌کند. چنین درختی، **درخت پوشای**[2] G نامیده می‌شود.

می‌گوییم *matdroid* $M = (S, \mathcal{I})$ **وزن دار**[3] است، اگر یک تابع وزن w مربوطه وجود داشته باشد که یک وزن مثبت و غیر صفر $w(x)$ را به هر عضو $x \in S$ اختصاص می‌دهد. تابع وزن w برای زیرمجموعه‌های S بوسیله سری زیر برای هر $A \subseteq S$ بسط می‌یابد:

$$w(A) = \sum_{x \in A} w(x)$$

برای مثال اگر $w(e)$ به طول یال e در *matroid* گرافی M_G اشاره کند، آنگاه $w(A)$ مجموع طولهای یالهای مجموعه یالی A است.

الگوریتم‌های حریصانه روی یک *matroid* وزن دار

بسیاری از مسائل که روش حریصانه برای آنها جواب بهینه را تولید می‌کند، می‌توانند بعنوان یافتن یک زیرمجموعه مستقل با ماکزیمم وزن در یک *matroid* وزن دار بیان گردند. به عبارت دیگر، *matroid* وزن دار $M = (S, \mathcal{I})$ را داریم و می‌خواهیم مجموعه مستقل $A \in \mathcal{I}$ را پیدا کنیم، بطوریکه $w(A)$ ماکزیمم شده باشد. چنین زیرمجموعه مستقلی که دارای بیشترین وزن ممکن است را یک زیرمجموعه بهینه[4] *matroid* می‌نامیم. از آنجا که وزن $w(x)$ هر عضو $x \in S$ مثبت است، زیرمجموعه بهینه همواره یک زیرمجموعه مستقل ماکزیمال است - این مطلب همواره کمک می‌کند تا A را با بزرگترین اندازه

---

1. maximal
2. spanning tree
3. weighted
4. optimal

ممکن بسازیم.

برای مثال در **مسئله درخت پوشای مینیمم**[1]، گراف بدون جهت و همبند $G = (V, E)$ و تابع طول w، بطوریکه $w(e)$ طول (مثبت) یال e است، داده شده است. (اصطلاح "طول" را در اینجا برای اشاره به وزنهای اصلی یالهای گراف بکار می‌بریم و اصطلاح "وزن" را جهت اشاره به وزنهای $matroid$ مربوطه اختصاص می‌دهیم.) از ما خواسته می‌شود تا زیرمجموعه‌ای از یالها را پیدا کنیم که تمام رأسها را به یکدیگر متصل می‌کند و مینیمم مجموع طول را دارا می‌باشد. به منظور پرداختن به این مسئله به عنوان یافتن یک زیرمجموعه بهینه از یک $matroid$، $matroid$ وزن دار M_G با تابع وزن w' را درنظر بگیرید، بطوریکه $w'(e) = w_0 - w(e)$ و w_0 از طول یالی با ماکزیمم طول بزرگتر است. در این $matroid$ وزن‌دار، تمام وزنها مثبت‌اند و زیرمجموعه بهینه، یک درخت پوشا با مینیمم مجموع طول در گراف اصلی می‌باشد. به صورت دقیق‌تر، هر زیرمجموعه مستقل ماکزیمال A متناظر با یک درخت پوشا است و از آنجا که برای هر زیرمجموعه مستقل ماکزیمال A داریم

$$w'(A) = (|V| - 1)w_0 - w(A)$$

لذا یک زیرمجموعه مستقل که مقدار $w'(A)$ را ماکزیمم می‌کند باید $w(A)$ را مینیمم نماید. بنابراین هر الگوریتم که بتواند زیرمجموعه بهینه A را در یک $matroid$ دلخواه پیدا کند، مسئله درخت پوشای مینیمم را نیز می‌تواند حل کند.

فصل ۲۳ الگوریتم‌هایی را برای مسئله درخت پوشای مینیمم ارائه می‌کند، اما در اینجا یک الگوریتم حریصانه ارائه می‌کنیم که برای هر $matroid$ وزن دار قابل اعمال است. الگوریتم بعنوان ورودی، $matroid$ وزن دار $M = (S, \ell)$ را با یک تابع وزن مثبت مربوطه w گرفته، و زیرمجموعه بهینه A را برمی‌گرداند. در شبه کد، اجزاءِ M را با $S[M]$ و $\ell[M]$ وتابع وزن را با w نشان می‌دهیم. الگوریتم، حریصانه است زیرا هر عضو $x \in S$ را به نوبت در ترتیب نزولی یکنواخت وزنها درنظر می‌گیرد و اگر $A \cup \{x\}$ مستقل باشد، بلافاصله آنرا به مجموعه A اضافه می‌کند.

```
GREEDY(M, w)
1  A ← Ø
2  sort S[M] into monotonically decreasing order by weight w
3  for each x ∈ S[M], taken in monotonically decreasing order by weight w(x)
4       do if A ∪ {x} ∈ ℓ[M]
5            then A ← A ∪ {x}
6  return A
```

اعضاء S بنوبت در یک ترتیب نزولی یکنواخت بر حسب وزنهایشان درنظر گرفته می‌شوند. اگر عضو x

---

1. *minimum-spanning-tree problem*

درنظر گرفته شود، در حالی که استقلال A حفظ گردد می‌تواند به A اضافه شود. درغیراینصورت x کنار گذاشته می‌شود. از آنجا که مجموعه تهی طبق تعریف *matroid* مستقل است و از آنجا که x تنها زمانی که $A\cup\{x\}$ مستقل باشد به A اضافه می‌شود، زیرمجموعه A بنا به استقرا همواره مستقل است. بنابراین *GREEDY* همواره مجموعه مستقل A را برمی‌گرداند. بزودی خواهیم دید که A یک زیرمجموعه با ماکزیمم وزن ممکن است، بنابراین A یک زیرمجموعه بهینه است.

زمان اجرای *GREEDY* به آسانی قابل تحلیل است. n را برابر $|S|$ قرار می‌دهیم. مرحله مرتب‌سازی *GREEDY* زمان $O(nlgn)$ را صرف می‌کند. خط ۴ دقیقاً n مرتبه اجرا می‌شود، یکبار برای هر عضو S. هر اجرای خط ۴ نیاز به یک بررسی دارد که آیا مجموعه $A\cup\{x\}$ مستقل است یا خیر. اگر هر بررسی زمان $O(f(n))$ را صرف کند، زمان اجرای کل الگوریتم برابر $O(nlgn+nf(n))$ است.

اکنون ثابت می‌کنیم که *GREEDY* یک زیرمجموعه بهینه را بر می‌گرداند.

لم ۱۶.۷ (*matroid*ها دارای ویژگی انتخاب حریصانه هستند)

فرض کنید $M = (S, \mathcal{I})$ یک *matroid* وزن دار با تابع وزن w باشد که S بصورت نزولی یکنواخت بر اساس وزن مرتب شده است. x را اولین عضو از S در نظر بگیرید بطوریکه $\{x\}$ مستقل است، اگر چنین x ای وجود داشته باشد. اگر x موجود باشد، آنگاه زیرمجموعه بهینه A از S وجود دارد که شامل x است.

اثبات اگر چنین x ای وجود داشته باشد، آنگاه زیرمجموعه مستقل، مجموعه تهی است و اثبات کامل است. درغیراینصورت B را به عنوان یک زیرمجموعه غیر تهی بهینه درنظر بگیرید. فرض کنید $x \notin B$ در غیر اینصورت قرار می‌دهیم $A=B$ و اثبات کامل می‌شود.

هیچ یک از اعضای B وزنی بیشتر از $w(x)$ ندارند. برای مشاهده این موضوع توجه کنید که $y \in B$ بر این دلالت دارد که $\{y\}$ مستقل است، زیرا $B\in\mathcal{I}$ و $\mathcal{I}$ قابل ارث بری است. بنابراین انتخاب x تضمین می‌کند که برای هر $y \in B$، $w(x) \geq w(y)$.

مجموعه A را بصورت مقابل بسازید. با مجموعه $A = \{x\}$ شروع کنید. با انتخاب x، A مستقل است. با استفاده از ویژگی تبادل، به طور پی در پی یک عضو جدید از B را پیدا کنید که بتواند در حالی که استقلال A حفظ می‌شود به A اضافه شود تا آنکه $|A| = |B|$. آنگاه برای $y\in B$، $A=B-\{y\} \cup \{x\}$ و بنابراین

$$w(A) = w(B) - w(y) + w(x)$$

چون B بهینه است، A نیز باید بهینه باشد و از آنجا که $x \in A$، لم اثبات می‌شود. ■

در ادامه نشان می‌دهیم که اگر یک عضو در ابتدا قابل انتخاب نباشد، آنگاه بعد از آن هم نمی‌تواند قابل انتخاب باشد.

لم ۱۶.۸

$M = (S, \ell)$ را بعنوان یک *matroid* در نظر بگیرید. اگر x که یک توسعه از زیرمجموعه مستقل A از S است، عضوی از S باشد آنگاه x یک توسعه از $\varnothing$ نیز می‌باشد.

اثبات از آنجا که x یک توسعه از A است، $A \cup \{x\}$ مستقل است. از آنجا که ℓ قابل ارث بری است، $\{x\}$ باید مستقل باشد. بنابراین x یک توسعه از $\varnothing$ است. ■

قضیه فرعی ۱۶.۹

$M = (S, \ell)$ را یک *matroid* درنظر بگیرید. اگر x عضوی از S باشد بطوریکه توسعه‌ای از $\varnothing$ نباشد، آنگاه x یک توسعه از زیرمجموعه مستقل A از S نیست.

اثبات این قضیه فرعی بوضوح، عکس نقیض لم ۱۶.۸ است. ■

قضیه فرعی ۱۶.۹ بیان می‌دارد که هر عضو که بلافاصله نتواند مورد استفاده قرار گیرد، هرگز نمی‌تواند استفاده شود. بنابراین *GREEDY* با نادیده گرفتن اعضای اولیه در S که توسعه $\varnothing$ نیستند نمی‌تواند باعث بروز خطا گردد، زیرا آنها هرگز نمی‌توانند مورد استفاده قرار گیرند.

لم ۱۶.۱۰ (*matroid*ها دارای ویژگی زیرساختار بهینه هستند)

x را اولین عضو S که بوسیله *GREEDY* برای *matroid* وزن‌دار $M = (S, \ell)$ انتخاب شده است، درنظر بگیرید. ادامه مسئله که عبارت است از پیدا کردن یک زیرمجموعه مستقل با ماکزیمم وزن شامل x، به پیدا کردن یک زیرمجموعه مستقل با ماکزیمم وزن از *matroid* وزن دار $M' = (S', \ell')$ تبدیل می‌شود، بطوریکه

$$S' = \{y \in S : \{x, y\} \in \ell\} ,$$
$$\ell' = \{B \subseteq S - \{x\} : B \cup \{x\} \in \ell\} ,$$

و تابع وزن برای M' همان تابع وزن برای M است که به S' محدود شده است. (M' را *انقباض*[1] M بوسیله عضو x می‌نامیم.)

اثبات اگر A یک زیرمجموعه مستقل با ماکزیمم وزن از M باشد که شامل x نیز است، آنگاه $A' = A - \{x\}$ یک زیرمجموعه مستقل از M' است. بر عکس، زیرمجموعه مستقل A' از M'، زیرمجموعه مستقل $A = A' \cup \{x\}$ را حاصل می‌کند. از آنجا که در هر دو حالت داریم

$$w(A) = w(A') + w(x),$$

1. contraction

جواب با ماکزیمم وزن در M که شامل x است، جواب با ماکزیمم وزن در M' را حاصل می‌کند و برعکس. ■

قضیه ۱۶.۱۱ (صحت اجرای الگوریتم حریصانه روی ***matroid*** ها)

اگر $M=(S, \mathcal{I})$ یک *matroid* وزن دار با تابع وزن w باشد، آنگاه $GREEDY(M, w)$ یک زیرمجموعه بهینه را برمی‌گرداند.

اثبات بنا به قضیه فرعی ۱۶.۹، هر عضوی که در ابتدا حذف می‌شود، بدلیل آنکه توسعه $\emptyset$ نیست می‌تواند نادیده گرفته شود، زیرا هرگز نمی‌تواند مفید باشد. وقتی اولین عضو x انتخاب می‌شود، لم ۱۶.۷ بر این دلالت می‌کند که روال $GREEDY$ با اضافه نمودن x به A ایجاد خطا نمی‌کند، زیرا یک زیرمجموعه بهینه شامل x وجود دارد. سرانجام لم ۱۶.۱۰ بر این دلالت می‌کند که مسئله باقیمانده، پیدا کردن یک زیرمجموعه بهینه در M' *matroid* است که انقباض M بوسیله x است. پس از آنکه روال $GREEDY$، A را با $\{x\}$ مقداردهی کرد، تمام گام‌های باقیمانده می‌توانند در *matroid* $M'=(S', \mathcal{I}')$ قابل اعمال باشند، زیرا B در M' مستقل است اگر و تنها اگر برای تمام مجموعه‌های $B \in \mathcal{I}'$، $B \cup \{x\}$ در M مستقل باشد. بنابراین عمل بعدی $GREEDY$ یک زیرمجموعه مستقل با ماکزیمم وزن برای M' پیدا خواهد کرد و عمل کلی $GREEDY$ یک زیرمجموعه مستقل با ماکزیمم وزن برای M پیدا خواهد کرد. ■

تمرین‌ها

۱ - ۱۶.۴* نشان دهید $(S, \mathcal{I}_k)$ یک *matroid* است، بطوریکه S یک مجموعه محدود و $\mathcal{I}_k$ مجموعه تمام زیرمجموعه‌های S با اندازه حداکثر k می‌باشد، بطوریکه $k \leq |S|$

۲ - ۱۶.۴* در ماتریس T با ابعاد $m \times n$ و با درایه‌های هم نوع (مانند اعداد حقیقی)، نشان دهید $(S, \mathcal{I})$ یک *matroid* است، بطوریکه S مجموعه ستونهای T می‌باشد و $A \in \mathcal{I}$ است، اگر و تنها اگر ستونهای داخل A به صورت خطی مستقل باشند.

۳ - ۱۶.۴* نشان دهید اگر $(S, \mathcal{I})$ یک *matroid* باشد، آنگاه $(S, \mathcal{I}')$ یک *matroid* است که در آن

$\mathcal{I}' = \{A' :$ شامل مجموعه ماکزیمال $S - A'$ $A \in \mathcal{I}$ است$\}$

به عبارت دیگر، مجموعه‌های مستقل ماکزیمال از $(S, \mathcal{I}')$ فقط مکمل‌های مجموعه‌های مستقل ماکزیمال از $(S, \mathcal{I})$ هستند.

۴ - ۱۶.۴* S را یک مجموعه محدود و $S_1, S_2, \ldots, S_k$ را یک افراز از S به زیرمجموعه‌های جدا از هم (مجزا) غیر تهی در نظر بگیرید. ساختار $(S, \mathcal{I})$ را با این شرط که $\{$برای $i = 1, 2, \ldots, k$ تعریف کنید. $\mathcal{I} = \{A : |A \cap S_i| \leq 1$ نشان دهید که $(S, \mathcal{I})$ یک *matroid* است. به عبارت دیگر،

مجموعه تمام مجموعه‌های A که شامل حداکثر یک عضو در هر بلاک از افراز هستند، مجموعه‌های مستقل یک *matroid* را مشخص می‌کند.

۵ - ۱۶.۴* نشان دهید چگونه تابع وزن یک مسئله *matroid* وزن دار که در آن جواب بهینه مطلوب، یک زیرمجموعه مستقل ماکزیمال با مینیمم وزن است را تغییر دهیم تا به یک مسئله *matroid* وزن دار استاندارد تبدیل شود. به طور دقیق ثابت کنید که تغییر شما صحیح است.

---

* ۱۶.۵ مسئله زمان‌بندی کارها[1]

یکی از مسائل جالب که می‌تواند با استفاده از *matroid*ها حل شود، مسئله زمان بندی بهینه کارهای دارای زمان واحد روی یک پردازنده است، به طوریکه هر کار یک مهلت دارد، بهمراه یک جریمه که اگر مهلت از دست رفت باید پرداخت شود. مسئله پیچیده به نظر می‌رسد، اما می‌تواند با یک روش ساده شگفت‌انگیز که از الگوریتم حریصانه استفاده می‌کند حل شود.

یک *کار دارای زمان واحد*[2]، یک عمل است، مانند اجرای یک برنامه روی کامپیوتر، که دقیقاً یک واحد از زمان جهت کامل شدن نیاز دارد. در مجموعه محدود S از کارهای با زمان واحد، *زمان بندی*[3] برای S عبارتست از یک جایگشت از S که ترتیب کارهایی را که باید انجام شوند مشخص می‌کند. اولین کار در زمان بندی در زمان 0 شروع و در زمان 1 خاتمه می‌یابد، دومین کار در زمان 1 شروع و در زمان 2 خاتمه می‌یابد و به همین ترتیب.

مسئله *زمان بندی کارهای با زمان واحد با مهلتها و جریمه‌ها برای یک تک پردازنده*[4] ورودیهای زیر را دارد:

- مجموعه $S = \{a_1, a_2, \dots, a_n\}$ از n کار با زمان واحد؛
- یک مجموعه از n *مهلت*[5] با مقادیر صحیح $d_1, d_2, \dots, d_n$ به طوری که هر d_i در شرط $1 \le d_i \le n$ صدق کرده و کار a_i در زمان d_i خاتمه می‌یابد؛ و
- یک مجموعه از m وزن یا *جریمه*[6] غیر منفی $w_1, w_2, \dots, w_n$ بطوریکه اگر کار a_i در زمان d_i خاتمه نیابد باعث جریمه w_i می‌شویم و چنانچه یک کار در مهلت تعیین شده‌اش خاتمه یابد، جریمه‌ای درنظر گرفته نمی‌شود.

از ما خواسته می‌شود تا یک زمان بندی برای S پیدا کنیم که مجموع جریمه‌های ایجاد شده بدلیل مهلتهای از دست رفته را مینیمم کند.

---

1. task-scheduling problem
2. unit-time task
3. schedule
4. scheduling unit-time tasks with deadlines and penalties for a single proccessor
5. deadline
6. penalty

یک زمان بندی ارائه شده را درنظر بگیرید. می‌گوییم یک کار در این زمان بندی ***تأخیردار***[1] است، اگر پس از مهلتش خاتمه یابد. در غیر اینصورت کار در زمان بندی ***بدون تأخیر***[2] است. یک زمان بندی دلخواه می‌تواند همواره به ***شکل ابتدا-بدون تأخیر***[3] در آید، که در آن کارهای بدون تأخیر بر کارهای دارای تأخیر مقدم هستند. برای مشاهده این مطلب، توجه کنید که اگر کار بدون تأخیر a_i پس از کار تأخیردار a_j بیاید، می‌توانیم مکان a_i و a_j را عوض کنیم و a_i هنوز هم بدون تأخیر و a_j هنوز تأخیردار خواهد بود.

به طور مشابه ادعا می‌کنیم که یک زمان بندی دلخواه می‌تواند همواره به ***شکل مرسوم***[4] در آید که در آن کارهای بدون تأخیر بر کارهای تأخیر دار مقدم هستند و کارهای بدون تأخیر به صورت صعودی یکنواخت بر اساس مهلتها زمان بندی می‌شوند. برای انجام این کار، زمان بندی را به شکل ابتدا-بدون‌تأخیر در می‌آوریم. آنگاه تا زمانی که دو کار بدون تأخیر a_i و a_j وجود دارند که در زمانهای مربوطه k و $k+1$ در زمان بندی خاتمه می‌یابند بطوریکه $d_j<d_i$ مکانهای a_i و a_j را با هم عوض می‌کنیم. از آنجا که a_j قبل از تعویض بدون تأخیر است، لذا $k+1 \leq d_j$. بنابراین $k+1 < d_i$ و در نتیجه a_i پس از تعویض بازهم بدون تأخیر است. کار a_j در زمان بندی به یک مکان جلوتر منتقل شده است، بنابراین پس از تعویض هنوز هم بدون تأخیر است.

بنابراین جستجو برای یک زمان بندی بهینه تبدیل به پیدا کردن یک مجموعه A از کارها می‌شود که این کارها باید در زمان بندی بهینه بدون تأخیر باشند. هنگامی که A مشخص شد، می‌توانیم زمان بندی واقعی را با لیست کردن اعضاء A در یک ترتیب صعودی یکنواخت بر اساس مهلت، و سپس لیست کردن کارهای تأخیردار (یعنی $S-A$) در ترتیبی که تولید یک ترتیب مرسوم از زمان بندی بهینه نماید، بوجود آوریم.

می‌گوییم مجموعه A شامل کارها ***مستقل***[5] است، اگر یک زمان بندی برای این کارها وجود داشته باشد، بطوریکه هیچ کدام از کارها تأخیردار نباشند. به وضوح مجموعه کارهای بدون تأخیر برای یک زمان‌بندی، یک مجموعه مستقل شامل کارها را شکل می‌دهد. فرض کنید $\mathcal{I}$ به مجموعه تمام مجموعه‌های مستقل شامل کارها اشاره می‌کند.

مسئله تعیین مستقل بودن مجموعه داده شده A از کارها را در نظر بگیرید. برای $t = 0, 1, \ldots, n$ فرض کنید $N_t(A)$ به تعداد کارها در A که مهلتهایشان برابر t یا زودتر از آن است اشاره می‌کند. توجه کنید که برای مجموعه A داریم $N_0(A) = 0$.

---

1. late
2. early
3. early-first form
4. canonical form
5. independent

لم ۱۶.۱۲

برای مجموعه A از کارها، گزاره‌های زیر معادل‌اند.

۱. مجموعه A مستقل است.

۲. برای $t = 0, 1, . . . , n$ داریم $N_t(A) \leq t$.

۳. اگر کارهای A در یک ترتیب صعودی یکنواخت بر حسب مهلت‌ها زمان بندی شده باشند، آنگاه هیچ کاری تأخیردار نمی‌باشد.

اثبات به وضوح اگر برای یک t داشته باشیم $N_t(A) > t$، آنگاه هیچ راهی برای ساخت یک زمان‌بندی برای مجموعه A، طوری که هیچ کاری تأخیردار نباشد وجود ندارد، زیرا بیش از t کار وجود دارند که باید قبل از زمان t خاتمه یابند. بنابراین (۱) بر (۲) دلالت می‌کند. اگر (۲) برقرار باشد، آنگاه (۳) نیز باید بر قرار باشد: هنگامی که کارها را در یک ترتیب صعودی یکنواخت بر اساس مهلت‌ها زمان بندی می‌کنیم، هرگز "گیر" نکرده و همواره به جواب می‌رسیم، زیرا (۲) بر این دلالت می‌کند که بیشترین مهلت iام، حداکثر برابر i است. در نهایت (۳) به طور جزئی بر (۱) دلالت می‌کند. ■

با استفاده از ویژگی ۲ لم ۱۶.۱۲، می‌توانیم به آسانی محاسبه کنیم که آیا یک مجموعه داده شده شامل کارها مستقل است یا خیر (تمرین ۲ - ۱۶.۵ را ملاحظه نمایید).

مسئله مینیمم کردن مجموع جریمه‌های کارهای تأخیردار همانند مسئله ماکزیمم کردن مجموع جریمه‌های کارهای بدون تأخیر است. از این‌رو قضیه زیر تضمین می‌کند که می‌توانیم از الگوریتم حریصانه جهت پیدا کردن یک مجموعه مستقل A شامل کارها با ماکزیمم مجموع جریمه‌ها استفاده کنیم.

قضیه ۱۶.۱۳

اگر S یک مجموعه از کارهای دارای زمان واحد با مهلت‌ها باشد، و ℓ مجموعه تمام مجموعه‌های مستقل شامل کارها باشد، آنگاه ساختار (S,ℓ) نظیر، یک *matroid* است.

Task							
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

شکل ۱۶.۷ نمونه‌ای از مسئله زمان بندی کارهای دارای زمان واحد با مهلت‌ها و جریمه‌ها برای یک پردازنده واحد.

اثبات هر زیرمجموعه یک مجموعه مستقل از کارها، مطمئناً مستقل است. برای اثبات ویژگی تبادل، فرض کنید B و A مجموعه‌های مستقلی از کارها هستند، بطوریکه $|B| > |A|$. k را بزرگترین t درنظر

بگیرید به طوریکه $N_t(B) \le N_t(A)$. (چنین مقداری از t وجود دارد زیرا $N_0(A) = N_0(B) = 0$.) از آنجا که $|B| = N_0\ (B)$ و $N_0(A) = |A|$ اما $|B| > |A|$، لذا باید داشته باشیم $k < n$ و همچنین برای تمام j ها در بازه $k+1 \le j \le n$ باید داشته باشیم $N_j(B) > N_j(A)$ بنابراین B شامل کارهای بیشتری با مهلت $k+1$ نسبت به A است. a_i را یک کار در $B-A$ با مهلت $k+1$ درنظر بگیرید. قرار دهید $A' = A \cup \{a_i\}$.

اکنون نشان می‌دهیم که A' با استفاده از ویژگی ۲ لم ۱۶.۱۲ باید مستقل باشد. برای $0 \le t \le k$ داریم $N_t(A') = N_t(A) \le t$ زیرا A مستقل است. برای $k < t \le n$ داریم $N_t(A') \le N_t(B) \le t$ زیرا B مستقل است. بنابراین A' مستقل است که این مطلب اثبات *matroid* بودن $(S, \mathcal{I})$ را کامل می‌کند. ■

با توجه به قضیه ۱۶.۱۱، می‌توانیم از یک الگوریتم حریصانه جهت پیدا کردن یک مجموعه مستقل A شامل کارها با ماکزیمم وزن استفاده کنیم. سپس می‌توانیم یک زمان بندی بهینه ایجاد کنیم که کارهای داخل A را به عنوان کارهای بدون تأخیرش دارا است. این روش یک الگوریتم کارآمد جهت زمان بندی کارهای دارای زمان واحد با مهلت‌ها و جریمه‌ها برای یک پردازنده واحد می‌باشد. زمان اجرا با استفاده از روال *GREEDY* برابر $O(n^2)$ است، زیرا هر یک از $O(n)$ بررسی به منظور تعیین استقلال که بوسیله این الگوریتم انجام می‌شود، زمان $O(n)$ را صرف می‌کند (تمرین ۲-۱۶.۵ را ملاحظه نمایید).

یک پیاده‌سازی سریعتردر مسئله ٤-١٦ ارائه شده است. شکل ١٦.٧ نمونه‌ای از مسئله زمان‌بندی کارهای دارای زمان واحد با مهلت‌ها و جریمه‌ها برای یک پردازنده واحد را ارائه می‌کند. در این نمونه، الگوریتم حریصانه کارهای a_1، a_2، a_3 و a_4 را انتخاب می‌کند، سپس a_5 و a_6 را رد کرده و a_7 را می‌پذیرد. زمان‌بندی بهینه بشکل $\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle$ باجریمه $w_5 + w_6 = 50$ است.

تمرین‌ها

۱ - ۱۶.۵ نمونه مسئله زمان بندی داده شده در شکل ۱۶.۷ را حل کنید، اما با هر جریمه w_i که بجای آن $80-w_i$ جایگزین شده است.

۲ - ۱۶.۵ نشان دهید چگونه با استفاده از ویژگی ۲ لم ۱۶.۱۲، در زمان $O(|A|)$ مشخص کنیم که آیا مجموعه داده شده A شامل کارها مستقل است یا خیر.

مسائل

۱ - ۱٦ خرد کردن سکّه

مسئله خرد کردن n سنت با استفاده از کمترین تعداد سکّه را در نظر بگیرید. فرض کنید که ارزش هر سکّه، مقداری صحیح است.

a. الگوریتمی حریصانه برای خرد کردن با استفاده از سکّه‌های کوارتر (25 سنتی)، [illegible] 10 سنتی)،

نیکل (5 سنتی) و پنی (1 سنتی) ارائه نمایید. ثابت کنید الگوریتم شما یک جواب بهینه حاصل می‌کند.

b. فرض کنید ارزش سکّه‌های موجود توانهای c است، یعنی ارزشها برای اعداد صحیح $c > 1$ و $K \geq 1$ برابر $c^0, c^1, \ldots, c^k$ هستند. نشان دهید الگوریتم حریصانه همواره به یک جواب بهینه منجر خواهد شد.

c. یک مجموعه از ارزشهای سکّه‌ها ارائه نمایید که الگوریتم حریصانه برای آن، جواب بهینه را تولید نکند. مجموعه شما باید شامل یک سکّه پنی (1 سنتی) باشد تا یک جواب برای هر مقدار n وجود داشته باشد.

d. الگوریتمی با مرتبه زمانی $O(nk)$ ارائه دهید که عمل خرد کردن را برای مجموعه‌ای شامل k سکّه با ارزشهای مختلف انجام دهد، با این فرض که یکی از سکّه‌ها پنی است.

۲ - ۱۶ زمان بندی جهت مینیمم کردن میانگین زمان تکمیل

فرض کنید مجموعه $S = \{a_1, a_2, \ldots, a_n\}$ از کارها داده شده است، به طوریکه کار a_i از زمانی که شروع می‌شود به p_i واحد زمان پردازش جهت تکمیل شدن نیاز دارد. شما یک کامپیوتر برای اجرای این کارها دارید، بنحوی که کامپیوتر می‌تواند تنها یک کار را در یک زمان اجرا کند. c_i را *زمان تکمیل*[1] کار a_i در نظر بگیرید، به عبارت دیگر زمانی که کار a_i پردازش خود را کامل می‌کند. هدف شما مینیمم کردن میانگین زمان تکمیل و یا به عبارت دیگر مینیمم کردن $(1/n) \sum_{i=1}^{n} c_i$ است. برای مثال فرض کنید دو کار a_1 و a_2 با $p_1 = 3$ و $p_2 = 5$ وجود دارند، و زمان بندی را در نظر بگیرید که در آن a_2 ابتدا اجرا می‌شود و پس از آن a_1 اجرا می‌گردد. پس $c_2 = 5$ ، $c_1 = 8$ و میانگین زمان تکمیل برابر $(5 + 8) / 2 = 6.5$ است.

a. الگوریتمی ارائه کنید که کارها را بنحوی زمان بندی کند که میانگین زمان تکمیل مینیمم گردد. هر کار باید بدون توقف اجرا شود، به عبارت دیگر هنگامی که کار a_i شروع شد باید به طور پیوسته در p_i واحد از زمان اجرا شود. ثابت کنید الگوریتم شما میانگین زمان تکمیل را مینیمم می‌کند و زمان اجرای الگوریتم خود را بیان کنید.

b. اکنون فرض کنید کارها همگی در یک زمان در دسترس نمی‌باشند. به عبارت دیگر هر کار یک *زمان رهایی*[2] r_i دارد که قبل از آن برای پردازش در دسترس نیست. همچنین فرض کنید که **توقف**[3] مجاز است، بطوریکه یک کار می‌تواند متوقف شده و دوباره در یک زمان دیگر آغاز شود. برای مثال کار a_i با زمان پردازش $p_i = 6$ ممکن است در زمان 1 شروع شود و در زمان 4 متوقف گردد. سپس می‌تواند در زمان 10 ادامه یافته و در زمان 11 متوقف شود و سرانجام در زمان 13 ادامه یابد و در

1. completion time
2. release time
3. preemption

زمان 15 کامل شود. کار a_i در مجموع در 6 واحد زمانی اجرا شده است، اما زمان اجرای آن به سه قسمت تقسیم شده است. می‌گوییم زمان تکمیل a_i 15 است. الگوریتمی ارائه نمایید که کارها را طوری زمان بندی نماید که میانگین زمان تکمیل در این زمان بندی جدید مینیمم شود. ثابت کنید الگوریتم شما میانگین زمان تکمیل را مینیمم می‌کند و زمان اجرای الگوریتم خود را تعیین کنید.

۳ - ۱۶ زیرگرافهای بدون دور

a. گراف بدون جهت $G = (V, E)$ را درنظر بگیرید. با استفاده از تعریف *matroid* نشان دهید که $(E, \mathcal{I})$ یک *matroid* است که در آن $A \in \mathcal{I}$ اگر و تنها اگر A یک زیرمجموعه بدون دور از E باشد.

b. *ماتریس برخورد*[1] برای گراف بدون جهت $G = (V, E)$، یک ماتریس $|V| \times |E|$ بنام M است بطوریکه اگر یال e با رأس v برخورد داشته باشد، $M_{ve} = 1$ و درغیراینصورت $M_{ve} = 0$ ثابت کنید یک مجموعه از ستونهای M روی فیلدهای صحیح به پیمانه 2 بصورت خطی مستقل است، اگر و تنها اگر مجموعه متناظر از یالها بدون دور باشد. سپس با استفاده از نتیجه تمرین ۲-۱۶.۴، اثبات دیگری ارائه کنید که $(E, \mathcal{I})$ در قسمت (a) یک *matroid* است.

c. فرض کنید وزن نامنفی $w(e)$ به هر یال در گراف بدون جهت $G = (V, E)$ اختصاص داده شده است. الگوریتمی کارآمد برای پیدا کردن یک زیرمجموعه بدون دور از E با ماکزیمم مجموع وزن ارائه کنید.

d. گراف جهت‌دار دلخواه $G = (V, E)$ را درنظر بگیرید و فرض کنید $(E, \mathcal{I})$ طوری تعریف شده باشد که $A \in \mathcal{I}$ اگر و تنها اگر A شامل هیچ دور جهت داری نباشد. نمونه‌ای از یک گراف جهت دار ارائه دهید که ساختار مربوطه $(E, \mathcal{I})$ یک *matroid* نباشد. مشخص کنید کدام شرط از تعریف *matroid* برقرار نیست.

e. *ماتریس برخورد* برای گراف جهت دار $G = (V, E)$، یک ماتریس $|V| \times |E|$ بنام M است که در آن $M_{ve} = -1$ اگر یال e رأس v را ترک کند و $M_{ve} = 1$ اگر یال e به رأس v وارد شود و درغیر اینصورت $M_{ve} = 0$ ثابت کنید اگر یک مجموعه از ستونهای M بصورت خطی مستقل باشد، آنگاه مجموعه متناظر از یالها شامل دور جهت دار نیست.

f. تمرین ۲-۱۶.۴ می‌گوید که مجموعه شامل مجموعه‌های مستقل خطی از ستونهای ماتریس M یک *matroid* را تشکیل می‌دهد. به طور دقیق توضیح دهید چرا نتایج قسمتهای (d) و (e) متناقض نمی‌باشند. چگونه یک تناظر کامل بین مفهوم یک مجموعه از یالها که بدون دور هستند و مفهوم یک مجموعه مربوطه از ستونهای ماتریس برخورد که بصورت خطی مستقل هستند می‌توانند به شکست بینجامد؟

---

1. *incidence matrix*

۴-۱۶ انواع متفاوت زمان بندی

الگوریتم زیر را برای مسئله زمان بندی کارهای دارای زمان واحد با مهلت‌ها و جریمه‌ها در بخش ۱۶.۵ درنظر بگیرید. فرض کنید n مکان زمانی بصورت اولیه خالی باشند که مکان زمانی i یک مکان با طول واحد از زمانی است که در زمان i خاتمه می‌یابد. کارها را در یک ترتیب نزولی یکنواخت از جریمه‌ها درنظر می‌گیریم. هنگامی که کار a_j را درنظر گرفتیم، اگر یک مکان زمانی در مهلت a_j یعنی d_j و یا قبل از آن وجود داشته باشد که هنوز خالی باشد، a_j را به آخرین مکانی که اینگونه است اختصاص داده و آنرا پر می‌کنیم. اگر چنین مکانی وجود نداشته باشد، کار a_j را به آخرین مکانی که هنوز پر نشده و خالی است، اختصاص می‌دهیم.

a. ثابت کنید این الگوریتم همواره یک جواب بهینه را خواهد داد.

b. با استفاده از جنگل مجموعه جدا از هم سریع که در بخش ۲۱.۳ بیان شده است، الگوریتم را بصورت کارآمد پیاده سازی کنید. فرض کنید مجموعه کارهای ورودی قبلاً در یک ترتیب نزولی یکنواخت از جریمه‌ها مرتب شده‌اند. زمان اجرای پیاده سازی خود را تحلیل نمایید.

۱۷ تحلیل سرشکن شده

در *تحلیل سرشکن شده*[1]، زمان مورد نیاز جهت انجام یک سلسله از اعمال ساختمان داده‌ای روی تمام اعمالی که انجام شده‌اند، میانگین گرفته می‌شود. اگر روی یک توالی از اعمال میانگین گرفته شود، هر چند ممکن است یک عمل در داخل آن توالی پر هزینه باشد، اما با استفاده از تحلیل سرشکن شده می‌توان نشان داد که هزینه میانگین عمل کم است. تحلیل سرشکن شده با تحلیل حالت میانگین که در آن احتمال درنظر گرفته نمی‌شود تفاوت دارد؛ تحلیل سرشکن شده میانگین کارآیی هر عمل در بدترین حالت را ضمانت می‌کند.

سه بخش نخست این فصل، سه تکنیک رایج که در تحلیل سرشکن شده استفاده می‌شوند را پوشش می‌دهند. بخش ۱۷.۱ با تحلیل جمعی شروع می‌شود که در آن حد بالای $T(n)$ را روی کل هزینه یک توالی از n عمل مشخص می‌کنیم. آنگاه هزینه میانگین هر عمل برابر $T(n)/n$ است. هزینه میانگین را هزینه سرشکن شده هر عمل درنظر می‌گیریم، بطوریکه تمام اعمال هزینه سرشکن شده یکسانی دارند.

بخش ۱۷.۲ روش حسابداری را پوشش می‌دهد، که در آن هزینه سرشکن شده هر عمل را مشخص می‌کنیم. هنگای که بیش از یک نوع عمل داریم، هر نوع عمل ممکن است هزینه سرشکن شده متفاوتی داشته باشد. روش حسابداری از قبل به تعدادی از اعمال در توالی، هزینه زیادی را نسبت می‌دهد و این هزینه زیاد را به عنوان "موجودی پیش پرداخت شده" روی اشیاء خاصی در ساختمان داده ذخیره می‌کند. این موجودی بعداً در توالی استفاده می‌شود تا برای اعمالی که کمتر از هزینه واقعی خود به آنها هزینه اختصاص داده شده است، پرداخت انجام شود.

بخش ۱۷.۳ در مورد روش پتانسیل بحث می‌کند که همانند روش حسابداری است و در آن هزینه سرشکن شده هر عمل را مشخص می‌کنیم و ممکن است از قبل به اعمال، هزینه‌های زیادی نسبت دهیم تا بعداً برای هزینه‌هایی که کم حساب شده‌اند، جبرانی باشد. روش پتانسیل، موجودی را بجای اختصاص به تک‌تک اشیاء در داخل ساختمان، بعنوان "انرژی پتانسیل" کل ساختمان داده نگهداری می‌کند.

---

1. *amortized analysis*

از دو مثال جهت بررسی این سه روش استفاده خواهیم کرد. یکی از آنها یک پشته با عمل اضافی *MULTIPOP* است که چندین شیء را به یکباره *pop* می‌کند. مثال دیگر، یک شمارنده دودویی است که به طور افزاینده بوسیله عمل *INCREMENT* از *0* شروع به شمردن می‌کند.

در هنگام مطالعه این فصل، به یاد داشته باشید که هزینه‌های نسبت داده شده در طول یک تحلیل سرشکن شده، تنها برای اهداف تحلیلی هستند. نیازی نیست که این هزینه‌ها در کد ظاهر شوند. برای مثال، اگر یک موجودی به شیء x در هنگام استفاده از روش حسابداری اختصاص داده شود نیازی به اختصاص یک مقدار مناسب به خصوصیت *credit[x]* در کُد نیست.

دیدگاهی که با انجام یک تحلیل سرشکن شده نسبت به یک ساختمان داده خاص بدست می‌آید، می‌تواند به بهینه سازی طراحی کمک کند. برای مثال در بخش ۱۷.۴، از روش پتانسیل برای تحلیل انبساط و انقباض جدول بصورت پویا استفاده خواهیم کرد.

۱۷.۱ تحلیل جمعی

در **تحلیل جمعی**[1]، نشان می‌دهیم که به ازای تمام nها، یک توالی از n عمل در مجموع، زمان $T(n)$ را در بدترین حالت صرف می‌کند. بنابراین در بدترین حالت، هزینه میانگین، یا **هزینه سرشکن شده**[2] برای هر عمل برابر $T(n)/n$ است. توجه کنید که این هزینه سرشکن شده برای هر عمل بکار می‌رود، حتی وقتی که چندین نوع عمل در توالی وجود داشته باشند. دو روش دیگر که در این فصل مطالعه خواهیم کرد، روش حسابداری و روش پتانسیل، ممکن است هزینه‌های سرشکن شده متفاوتی به انواع متفاوت عمل‌ها نسبت دهند.

اعمال پشته

در اولین مثال تحلیل جمعی، پشته‌هایی را مورد تحلیل قرار می‌دهیم که یک عمل جدید به آنها افزوده شده است. بخش ۱۰.۱ دو عمل اصلی پشته را ارائه کرد که هر کدام زمان $O(1)$ را صرف می‌کردند:

PUSH(S, x) شیء x را به داخل پشته S، *push* می‌کند.

POP(S) شیء بالای پشته S را *pop* کرده و آنرا برمی‌گرداند.

از آنجا که هر کدام از این اعمال در زمان $O(1)$ اجرا می‌شود، اجازه دهید هزینه هر کدام را برابر *1* درنظر بگیریم. از اینرو هزینه کلی یک توالی از n عمل *PUSH* و *POP* برابر n است و زمان اجرای واقعی n عمل برابر $\Theta(n)$ می‌باشد.

اکنون عمل پشته *MULTIPOP(S, k)* را اضافه می‌کنیم. این عمل k شیء بالای پشته S را حذف

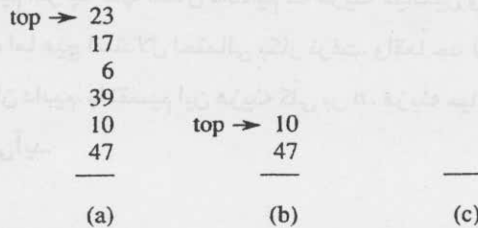
---

1. aggregate analysis
2. amortized cost

می‌کند و یا اگر پشته شامل کمتر از k شیء باشد کلّ پشته را *pop* می‌کند. در شبه کد زیر، عمل *STACK-EMPTY* اگر هیچ شیئی داخل پشته نباشد مقدار *TRUE* و درغیر اینصورت مقدار *FALSE* را برمی‌گرداند.

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k ≠ 0
2      do POP(S)
3         k ← k − 1
```

شکل ۱۷.۱ مثالی از *MULTIPOP* را نشان می‌دهد.



شکل ۱۷.۱ عملکرد *MULTIPOP* روی پشته S که بصورت اولیه در *(a)* نشان داده شده است. *4* شیء بالای پشته با فراخوانی *MULTIPOP(S, 4)*، *pop* می‌شوند که نتیجه آن در *(b)* نشان داده شده است. عمل بعدی، *MULTIPOP(S, 7)* است که پشته را خالی می‌کند - در *(c)* نشان داده شده است - زیرا کمتر از 7 شیء باقی مانده است.

زمان اجرای *MULTIPOP(S, k)* روی یک پشته شامل s شیء چیست؟ زمان اجرای واقعی در تعداد اعمال *POP* که انجام می‌شوند خطی است. بنابراین کافی است *MULTIPOP* را برحسب هزینه‌های مطلق *1* برای هر *PUSH* و *POP* تحلیل کنیم. تعداد تکرار حلقه *while* برابر مقدار *min(s, k)* از اشیائی است که از پشته *pop* می‌شوند. در هر تکرار حلقه، *POP* در خط *2* یکبار فراخوانی می‌شود. بنابراین کل هزینه *MULTIPOP* برابر *min(s, k)* است و زمان اجرای واقعی، یک تابع خطی از این هزینه است. اجازه دهید یک توالی از n عمل *PUSH*، *POP* و *MULTIPOP* را روی یک پشته خالی مورد تحلیل قرار دهیم. هزینه عمل *MULTIPOP* در بدترین حالت در توالی برابر $O(n)$ است، زیرا اندازه پشته حداکثر n است. بنابراین زمان هر عمل پشته در بدترین حالت برابر $O(n)$ است و از آنجا، یک توالی از n عمل دارای هزینه $O(n^2)$ است، زیرا ممکن است $O(n)$ عمل *MULTIPOP* با هزینه $O(n)$ برای هر کدام داشته باشیم. اگر چه این تحلیل درست است اما نتیجه $O(n^2)$ که با درنظر گرفتن هزینه در بدترین حالت برای هر عمل بدست می‌آید، قوی نیست.

با استفاده از تحلیل جمعی می‌توانیم به یک حد بالای بهتر دست یابیم که کل توالی شامل n عمل را

درنظر می‌گیرد. در حقیقت اگر چه یک عمل *MULTIPOP* می‌تواند پرهزینه باشد، اما یک توالی از n عمل *PUSH*، *POP* و *MULTIPOP* روی یک پشته که در ابتدا خالی است حداکثر دارای هزینه $O(n)$ است. چرا؟ هر شیء می‌تواند حداکثر یکبار برای هر بار که *push* می‌شود *pop* شود. بنابراین تعداد دفعاتی که *POP* در یک پشته غیر تهی می‌تواند فراخوانی شود، به همراه فراخوانی‌های داخل *MULTIPOP* حداکثر برابر تعداد اعمال *PUSH* می‌باشد که حداکثر برابر n است. برای هر مقدار n یک توالی از n عمل *PUSH*، *POP* و *MULTIPOP* زمان کل $O(n)$ را صرف می‌کند. هزینه میانگین یک عمل $O(n)/n = O(1)$ است. در تحلیل جمعی، هزینه میانگین هر عمل را به عنوان هزینه سرشکن شده اختصاص می‌دهیم. بنابراین در این مثال، تمام سه عمل پشته هزینه سرشکن شده $O(1)$ را دارند.

دوباره تأکید می‌کنیم اگر چه تنها نشان داده‌ایم که هزینه میانگین و از آنجا زمان اجرای یک عمل پشته برابر $O(1)$ است، اما هیچ استدلال احتمالی بکار نرفت. واقعاً حد $O(n)$ را در بدترین حالت روی یک توالی از n عمل نشان دادیم. با تقسیم این هزینه کلّی بر n، هزینه میانگین هر عمل و یا همان هزینه سرشکن شده بدست می‌آید.

افزایش یک شمارنده دودویی

بعنوان مثالی دیگر از تحلیل جمعی، مسئله پیاده سازی یک شمارنده k-بیتی دودویی را که از 0 به طور افزاینده می‌شمارد، درنظر بگیرید. آرایه $A[0\ .\ .\ k\text{-}1]$ از بیتها را که در آن $length[A] = k$، به عنوان شمارنده استفاده می‌کنیم. عدد دودویی x که در شمارنده ذخیره می‌شود کم ارزش‌ترین بیت خود را در $A[0]$ و با ارزش‌ترین بیت خود را در $A[k\text{-}1]$ قرار می‌دهد، بطوریکه $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. در ابتدا $x = 0$ است و بنابراین برای $i = 0, 1, \ldots, k\text{-}1$ داریم $A[i] = 0$. برای اضافه کردن 1 (در مبنای 2^k) به مقدار داخل شمارنده، از روال زیر استفاده می‌کنیم.

```
INCREMENT(A)
1  i ← 0
2  while i < length[A] and A[i] = 1
3      do A[i] ← 0
4         i ← i + 1
5  if i < length[A]
6     then A[i] ← 1
```

شکل ۱۷.۲ نشان می‌دهد که برای یک شمارنده دودویی هنگامی که با مقدار اولیه 0 شروع به افزایش می‌کند و پس از 16 بار افزایش با مقدار 16 خاتمه می‌یابد، چه اتفاقی روی می‌دهد. در آغاز هر تکرار حلقه *while* در خطوط ۴ – ۲، می‌خواهیم عدد 1 را به مکان i اضافه کنیم. اگر $A[i] = 1$، آنگاه افزودن 1 سبب تغییر آن بیت به 0 در مکان i و تولید یک نقلی با مقدار 1 می‌شود که باید در تکرار بعدی حلقه به

مکان $i+1$ام اضافه گردد. در غیر اینصورت حلقه به پایان می‌رسد و سپس اگر $i < k$، می‌دانیم که $A[i] = 0$، لذا اضافه کردن 1 به مکان i ام سبب تبدیل 0 به 1 می‌شود. این کار در خط ۶ انجام می‌شود. هزینه هر عمل *INCREMENT* در تعداد بیتهای تغییر یافته خطی است.

همانند مثال پشته تحلیل غیر دقیق و شتابزده به یک حد درست می‌انجامد، که البته قوی نیست. یک اجرای *INCREMENT* زمان $\Theta(k)$ را در بدترین حالت صرف می‌کند، که در آن آرایه A شامل تمام 1ها است. بنابراین یک توالی از n عمل *INCREMENT* روی یک شمارنده با مقدار اولیه صفر، در بدترین حالت زمان $O(nk)$ را صرف می‌کند.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

شکل ۱۷.۲ یک شمارنده دودویی 8بیتی که مقدارش بوسیله یک توالی از 16 عمل *INCREMENT* از 0 تا 16 افزایش می‌یابد. بیت‌هایی که تغییر کرده‌اند تا مقدار بعدی بوجود آید سایه زده شده‌اند. هزینه اجرا برای تغییر بیت‌ها در سمت راست نشان داده شده است. توجه کنید که هزینه کل، هرگز بیشتر از دو برابر تعداد کل عمل‌های *INCREMENT* نیست.

می‌توانیم با توجه به آنکه تمام بیت‌ها در هر مرتبه فراخوانی *INCREMENT* دچار تغییر نمی‌شوند، تحلیل خود را قوی‌تر کنیم تا به هزینه $O(n)$ در بدترین حالت برای یک توالی از n عمل *INCREMENT* بینجامد. همانطور که شکل ۱۷.۲ نشان می‌دهد، هر بار که *INCREMENT* فراخوانی می‌شود $A[0]$ تغییر می‌کند. بیت بعدی، $A[1]$، هر دو مرتبه یک بار تغییر می‌کند: یک توالی از n عمل *INCREMENT* روی یک شمارنده با مقدار اولیه صفر، سبب $\lfloor n/2 \rfloor$ مرتبه تغییر $A[1]$ می‌شود. بطور مشابه بیت $A[2]$ هر چهار مرتبه یکبار و یا $\lfloor n/4 \rfloor$ مرتبه در یک توالی از n عمل *INCREMENT* تغییر می‌کند. به طور کل برای $i = 0, 1, \ldots, \lfloor lgn \rfloor$ بیت $A[i]$، $\lfloor n/2^i \rfloor$ مرتبه در یک توالی از n عمل *INCREMENT*

روی یک شمارنده با مقدار اولیه صفر تغییر می‌کند. برای $i > \lfloor lg n \rfloor$، بیت $A[i]$ هرگز تغییر نمی‌کند. بنابراین تعداد کل تغییرات در توالی برابر است با

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n ,$$

بنابراین زمان یک توالی از n عمل *INCREMENT* روی یک شمارنده با مقدار اولیه صفر در بدترین حالت برابر $O(n)$ است. هزینه میانگین هر عمل، و بنابراین هزینه سرشکن شده آن برابر $O(n)/n = O(1)$ است.

تمرین‌ها

۱ - ۱۷.۱ اگر مجموعه اعمال شامل یک عمل *MULTIPUSH* باشد که k شیء را به داخل پشته *push* می‌کند، آیا حد $O(1)$ برای هزینه سرشکن شده اعمال پشته باز هم برقرار خواهد بود.

۲ - ۱۷.۱ نشان دهید که اگر عمل *DECREMENT* در مثال شمارنده k-بیتی منظور شده بود، n عمل می‌توانستند هزینه‌ای به بزرگی $\Theta(nk)$ داشته باشند.

۳ - ۱۷.۱ یک توالی از n عمل روی یک ساختمان داده انجام می‌شود. i امین عمل هزینه i را دارد، اگر i توانی صحیح از 2 باشد و درغیر اینصورت، هزینه آن 1 است. با استفاده از تحلیل جمعی، هزینه سرشکن شده هر عمل را مشخص کنید.

۱۷.۲ روش حسابداری

در روش *حسابداری*[1] از تحلیل سرشکن شده، به عمل‌های متفاوت هزینه‌های متفاوتی اختصاص می‌دهیم، بطوریکه به تعدادی از اعمال هزینه بیشتر و به تعدادی هزینه کمتری نسبت به هزینه واقعی آنها اختصاص داده می‌شود. مقدار هزینه‌ای که به یک عمل می‌دهیم، **هزینه سرشکن شده** آن عمل نامیده می‌شود. هنگامیکه هزینه سرشکن شده یک عمل از هزینه واقعی آن تجاوز کند، اختلاف هزینه بعنوان **موجودی**[2] به اشیائی مشخص در ساختمان داده اختصاص می‌یابد. موجودی بعداً می‌تواند جهت کمک به پرداخت برای اعمالی که هزینه سرشکن شده آنها کمتر از هزینه واقعی شان است مورد استفاده قرار گیرد. بنابراین می‌توان هزینه سرشکن شده یک عمل را مشاهده کرد که بین هزینه واقعی آن و موجودی که پرداخت شده یا استفاده می‌شود، تقسیم شده است. این روش با تحلیل جمعی که در آن تمام اعمال هزینه‌های سرشکن شده یکسانی دارند بسیار متفاوت است.

---

1. accounting method   2. credit

هزینه‌های سرشکن شده اعمال باید به دقت انتخاب شوند. اگر بخواهیم تحلیل را با هزینه‌های سرشکن شده انجام دهیم تا نشان دهیم که در بدترین حالت هزینه میانگین هر عمل کم است، کلّ هزینه سرشکن شده یک توالی از اعمال باید یک حد بالا برای هزینه کل واقعی توالی باشد. علاوه بر این همانند تحلیل جمعی، این رابطه باید برای تمام توالی‌های اعمال برقرار باشد. اگر هزینه واقعی iامین عمل را با c_i و هزینه سرشکن شده iامین عمل را با $\widehat{c_i}$ نشان دهیم، برای تمام توالی‌هایی از n عمل نیاز داریم

$$\sum_{i=1}^{n}\widehat{c_i} \geq \sum_{i=1}^{n} c_i \qquad (17.1)$$

موجودی کل که در ساختمان داده ذخیره شده است برابر تفاوت بین کل هزینه سرشکن شده و کل هزینه واقعی است، یا $\sum_{i=1}^{n}\widehat{c_i} - \sum_{i=1}^{n} c_i$. بنا به نامساوی (۱۷.۱)، کل موجودی مربوط به ساختمان داده باید در همه زمانها نامنفی باشد. اگر کل موجودی می‌توانست منفی باشد (در نتیجه پرداخت ارزش کم به عمل‌های قبلی، با این تعهد که دوباره پس از آن پرداخت صورت گیرد) آنگاه کل هزینه‌های سرشکن شده ایجاد شده در آن هنگام کمتر از کل هزینه‌های واقعی ایجاد شده می‌بود؛ برای توالی عمل‌ها تا آن زمان کل هزینه سرشکن شده یک حد بالا برای کل هزینه واقعی نمی‌بود. بنابراین باید مراقب باشیم تا موجودی کل در ساختمان داده هرگز منفی نشود.

اعمال پشته

برای توضیح روش حسابداری تحلیل سرشکن شده اجازه دهید به مثال پشته برگردیم. بخاطر آورید که هزینه‌های واقعی اعمال عبارت بودند از

PUSH 1 ,
POP 1 ,
MULTIPOP $\min(k, s)$,

که k آرگومانی است که برای *MULTIPOP* استفاده شده است و s اندازه پشته وقتیکه فراخوانی می‌شود است. اجازه دهید هزینه‌های سرشکن شده زیر را اختصاص دهیم:

PUSH 2 ,
POP 0 ,
MULTIPOP 0 .

توجه کنید که هزینه سرشکن شده *MULTIPOP* مقدار ثابت (*0*) است، در حالیکه هزینه واقعی متغیر است. در اینجا تمام سه هزینه سرشکن شده برابر *O(1)* می‌باشند اگر چه در کل، هزینه‌های سرشکن شده اعمال مورد نظر ممکن است به طور مجانبی تغییر کنند.

حال نشان خواهیم داد که می‌توانیم برای یک توالی از اعمال پشته، با پرداخت هزینه‌های

سرشکن‌شده پرداخت داشته باشیم. فرض کنید از یک اسکناس *1* دلاری برای نشان دادن هر واحد از هزینه استفاده می‌کنیم. با یک پشته خالی شروع می‌کنیم. شباهت بین ساختمان داده پشته و پشته‌ای از بشقابها در یک کافه تریا را از بخش ۱۰.۱ به خاطر آورید. وقتی یک بشقاب را روی پشته قرار می‌دهیم، از *1* دلار برای پرداخت هزینه واقعی *push* استفاده می‌کنیم و یک موجودی با *1* دلار (از *2* دلاری که پرداخت شده است) برای ما باقی می‌ماند، که آنرا روی بشقاب می‌گذاریم. در هر زمان یک دلار از موجودی روی هر بشقاب در پشته قرار دارد.

دلاری که در بالای بشقاب ذخیره شده است پیش پرداختی برای هزینه *pop* آن بشقاب از پشته می‌باشد. هنگامی که یک عمل *POP* را اجرا می‌کنیم، هیچ هزینه‌ای برای عمل پرداخت نمی‌کنیم و هزینه واقعی آنرا با استفاده از موجودی ذخیره شده در پشته پرداخت می‌کنیم. برای *pop* کردن یک بشقاب، یک دلار موجودی را از روی بشقاب برداشته و جهت پرداخت هزینه واقعی عمل استفاده می‌کنیم. بنابراین با پرداخت هزینه بیشتر برای عمل *PUSH* نیازی به پرداخت هزینه برای عمل *POP* نداریم.

علاوه بر این نیازی به پرداخت هزینه برای عمل *MULTIPOP* نیز نداریم. برای *pop* کردن اولین بشقاب، یک دلار موجودی را از بشقاب برداشته و جهت پرداخت هزینه واقعی یک عمل *pop* استفاده می‌کنیم. برای *pop* کردن بشقاب دوم، دوباره یک دلار از موجودی روی بشقاب جهت پرداخت برای عمل *POP* داریم و به همین ترتیب. بنابراین همواره از پیش جهت پرداخت برای اعمال *MULTIPOP* به مقدار کافی هزینه پرداخت کرده‌ایم. به عبارت دیگر از آنجا که هر بشقاب در پشته *1* دلار موجودی دارد و پشته همواره تعداد نامنفی بشقاب دارد تضمین کرده‌ایم که مقدار موجودی همواره نامنفی است. بنابراین برای هر توالی از n عمل *PUSH*، *POP* و *MULTIPOP*، هزینه سرشکن شده کل یک حد بالا برای هزینه واقعی کل است. از آنجا که هزینه سرشکن شده کل برابر $O(n)$ است، لذا هزینه واقعی کل نیز $O(n)$ می‌باشد.

افزایش یک شمارنده دودویی

به عنوان توضیحی دیگر از روش حسابداری، عمل *INCREMENT* روی یک شمارنده دودویی که از صفر شروع می‌شود را تحلیل می‌کنیم. همانطور که قبلاً مشاهده کردیم زمان اجرای این عمل متناسب با تعداد بیت‌های تغییر یافته است که از آن به عنوان هزینه خود برای این مثال استفاده خواهیم کرد. اجازه دهید یکبار دیگر از اسکناس یک دلاری برای نمایش هر واحد از هزینه (تغییر هر بیت در این مثال) استفاده کنیم.

برای تحلیل سرشکن شده، اجازه دهید یک هزینه سرشکن شده *2* دلاری را برای *1* کردن یک بیت پرداخت کنیم. هنگامیکه بیت *1* می‌شود، از *1* دلار (از *2* دلاری که پرداخت شده‌اند) برای پرداخت هزینه واقعی *1* کردن آن استفاده می‌کنیم و دلار دیگر را روی بیت به عنوان موجودی که بعداً وقتی بیت *0*

می‌شود استفاده می‌گردد، قرار می‌دهیم. در هر زمان هر *1* در شمارنده، یک دلار موجودی روی خود دارد، و بنابراین نیازی به پرداخت هزینه برای *0* کردن بیت نداریم؛ برای *0* کردن تنها از اسکناس *1* دلاری روی بیت جهت پرداخت استفاده می‌کنیم.

اکنون هزینه سرشکن شده *INCREMENT* می‌تواند تعیین گردد. هزینه *0* کردن بیت‌ها در داخل حلقه *while* با دلارهای روی بیت‌هایی که *0* می‌شوند پرداخت می‌گردد. در خط ۶ از *INCREMENT* حداکثر یک بیت *1* می‌شود و بنابراین هزینه سرشکن شده یک عمل *INCREMENT* حداکثر *2* دلار است. تعداد *1*ها در شمارنده هرگز منفی نیست و بنابراین مقدار موجودی همواره نامنفی است. بنابراین برای *n* عمل *INCREMENT*، هزینه سرشکن شده کل برابر $O(n)$ است که هزینه واقعی کل را محدود می‌کند.

تمرین‌ها

۱ - ۱۷.۲ یک توالی از اعمال پشته روی یک پشته که اندازه آن از k تجاوز نمی‌کند انجام می‌شود. بعد از اتمام هر k عمل، یک کپی پشتیبان از تمام پشته تهیه می‌شود. نشان دهید که هزینه n عمل پشته بهمراه کپی پشته، با اختصاص هزینه سرشکن شده مناسب به اعمال گوناگون پشته برابر $O(n)$ است.

۲ - ۱۷.۲ تمرین ۳ - ۱۷.۱ را با استفاده از روش حسابداری تحلیل سرشکن شده انجام دهید.

۳ - ۱۷.۲ فرض کنید نمی‌خواهیم تنها یک شمارنده را افزایش دهیم بلکه علاوه بر آن می‌خواهیم آن را *0* کنیم (یعنی تمام بیت‌های داخل آنرا *0* کنیم). نشان دهید چگونه یک شمارنده را بصورت آرایه‌ای از بیت‌ها پیاده سازی کنیم، بنحوی که هر توالی از n عمل *INCREMENT* و *RESET* روی یک شمارنده با مقدار اولیه صفر، زمان $O(n)$ را صرف کند. (راهنمایی: یک اشاره گر به با ارزش‌ترین *1* نگهداری کنید.)

۱۷.۳ روش پتانسیل

بجای نمایش کار از پیش پرداخت شده بصورت موجودی که با اشیائی مشخص در ساختمان داده ذخیره می‌شود، **روش پتانسیل**[1] از تحلیل سرشکن شده، کار از پیش پرداخت شده را بصورت "انرژی پتانسیل" و یا تنها "پتانسیل" نمایش می‌دهد که می‌تواند جهت پرداخت برای اعمال آینده آزاد شود. پتانسیل بجای آنکه به اشیائی خاص در داخل ساختمان داده اختصاص داده شود، به کل ساختمان داده اختصاص داده می‌شود.

روش پتانسیل بصورت زیر کار می‌کند. با یک ساختمان داده اولیه D_0 شروع می‌کنیم که روی آن n

---

1. potential method

عمل انجام می‌شود. برای هر $i = 1, 2, . . . , n$، c_i را هزینه واقعی iامین عمل قرار می‌دهیم و D_i را ساختمان داده‌ای که پس از بکارگیری iامین عمل روی ساختمان داده D_{i-1} حاصل می‌شود، قرار می‌دهیم. **تابع پتانسیل**[1] Φ هر ساختمان داده D_i را به یک عدد حقیقی $\Phi(D_i)$ نگاشت می‌کند. این عدد، **پتانسیل**[2] مربوط به ساختمان داده D_i است. **هزینه سرشکن شده** $\widehat{c_i}$ مربوط به iامین عمل با توجه به تابع پتانسیل Φ بصورت زیر تعریف می‌شود

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \quad (۱۷.۲)$$

بنابراین هزینه سرشکن شده هر عمل برابر هزینه واقعی آن بعلاوه افزایش پتانسیل حاصل از عمل می‌باشد. بنا به معادله (۱۷.۲)، هزینه سرشکن شده کل n عمل برابر است با

$$\begin{aligned} \sum_{i=1}^{n} \widehat{c_i} &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) . \end{aligned} \quad (۱۷.۳)$$

اگر بتوانیم یک تابع پتانسیل Φ تعریف کنیم بطوریکه $\Phi(D_n) \geq \Phi(D_0)$، آنگاه هزینه سرشکن‌شده کل $\sum_{i=1}^{n} \widehat{c_i}$ یک حد بالا برای هزینه واقعی کل $\sum_{i=1}^{n} c_i$ می‌باشد. در عمل همواره نمی‌دانیم چه تعداد عمل ممکن است انجام شود. بنابراین اگر برای تمام nها تأکید کنیم که $\Phi(D_n) \geq \Phi(D_0)$، آنگاه همانند روش حسابداری تضمین می‌کنیم که از پیش هزینه را پرداخت کرده‌ایم. اغلب مناسب است که $\Phi(D_0)$ را 0 تعریف کنیم و آنگاه نشان دهیم که برای تمام iها، $\Phi(D_i) \geq 0$. (تمرین ۱-۱۷.۳ را در مورد یک روش ساده برای نحوه مدیریت حالتهایی که در آنها $\Phi(D_0) \neq 0$ است، ملاحظه نمایید.)

به طور شهودی اگر اختلاف پتانسیل $\Phi(D_i) - \Phi(D_{i-1})$ از iامین عمل مثبت باشد آنگاه هزینه سرشکن شده $\widehat{c_i}$ یک پرداخت اضافه به i امین عمل را نشان می‌دهد، و پتانسیل ساختمان داده افزایش می‌یابد. اگر اختلاف پتانسیل منفی باشد، آنگاه هزینه سرشکن شده، یک پرداخت کم به iامین عمل را نشان می‌دهد و هزینه واقعی عمل با کاهش پتانسیل پرداخت می‌شود.

هزینه‌های سرشکن شده تعریف شده بوسیله معادله‌های (۱۷.۲) و (۱۷.۳) به انتخاب تابع پتانسیل Φ بستگی دارند. توابع پتانسیل متفاوت ممکن است به هزینه‌های سرشکن شده متفاوت که همچنان حدود بالا برای هزینه‌های واقعی‌اند منجر گردند. اغلب در انتخاب یک تابع پتانسیل مبادلاتی می‌توانند انجام گیرند؛ بهترین تابع پتانسیل جهت استفاده بستگی به حدود زمانی مطلوب دارد.

---

1. potential function
2. potential

اعمال پشته

برای توضیح روش پتانسیل، یکبار دیگر به مثال اعمال پشته *PUSH* ،*POP* و *MULTIPOP* بازمی‌گردیم. تابع پتانسیل Φ روی یک پشته را تعداد اشیاء داخل پشته تعریف می‌کنیم. برای پشته خالی D_0 که با آن شروع می‌کنیم، داریم $\Phi(D_0) = 0$. از آنجا که تعداد اشیاء داخل پشته هرگز منفی نیست، پشته D_i که پس از i امین عمل حاصل می‌گردد، پتانسیل نامنفی دارد و بنابراین

$$\begin{aligned} \Phi(D_i) &\geq 0 \\ &= \Phi(D_0)\,. \end{aligned}$$

لذا هزینه سرشکن شده کل n عمل با توجه به Φ ، یک حد بالا برای هزینه واقعی را نشان می‌دهد. اجازه دهید هزینه‌های سرشکن شده اعمال متفاوت پشته را حساب کنیم. اگر i امین عمل روی یک پشته که شامل s شیء است عمل *PUSH* باشد، آنگاه اختلاف پتانسیل برابر است با

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1\,. \end{aligned}$$

بنا به معادله (۱۷.۲)، هزینه سرشکن شده این عمل *PUSH* برابر است با

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2\,. \end{aligned}$$

فرض کنید i امین عمل روی پشته $MULTIPOP(S, k)$ است و $k' = min(k, s)$ شیء از پشته *pop* می‌شوند. هزینه واقعی عمل برابر k' است و اختلاف پتانسیل برابر است

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'\,.$$

بنابراین هزینه سرشکن شده عمل *MULTIPOP* برابر است با

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0\,. \end{aligned}$$

به طور مشابه هزینه سرشکن شده یک عمل *POP* معمولی برابر *0* است.

هزینه سرشکن شده هر یک از سه عمل برابر $O(1)$ است و بنابراین هزینه سرشکن شده کل یک توالی از n عمل برابر $O(n)$ است. از آنجا که قبلاً ثابت کردیم که $\Phi(D_i) \geq \Phi(D_0)$، هزینه سرشکن شده کل n عمل، یک حد بالا برای هزینه واقعی کل است. لذا هزینه n عمل در بدترین حالت برابر $O(n)$ است.

افزایش یک شمارنده دودویی

بعنوان مثالی دیگر از روش پتانسیل، نگاهی دوباره به افزایش یک شمارنده دودویی می‌اندازیم. این بار پتانسیل شمارنده بعد از iامین عمل *INCREMENT* را برابر b_i تعریف می‌کنیم، که تعداد 1 های داخل شمارنده پس از iامین عمل است.

حال هزینه سرشکن شده یک عمل *INCREMENT* را محاسبه می‌کنیم. فرض کنید که iامین عمل *INCREMENT* t_i بیت را 0 می‌کند. لذا هزینه واقعی عمل حداکثر t_i+1 است، زیرا علاوه بر 0 کردن t_i بیت، حداکثر یک بیت را 1 می‌کند. اگر $b_i=0$، آنگاه iامین عمل تمام k بیت را 0 می‌کند، و بنابراین $b_{i-1} = t_i = K$. اگر $b_i>0$ آنگاه $b_i = b_{i-1}-t_i+1$. در حالت دیگر، $b_i \le b_{i-1} - t_i + 1$ و اختلاف پتانسیل برابر است با

$$\begin{aligned}\Phi(D_i)-\Phi(D_{i-1}) &\le (b_{i-1}-t_i+1)-b_{i-1}\\ &= 1-t_i\ .\end{aligned}$$

بنابراین هزینه سرشکن شده برابر است با

$$\begin{aligned}\widehat{c_i} &= c_i+\Phi(D_i)-\Phi(D_{i-1})\\ &\le (t_i+1)+(1-t_i)\\ &= 2\ .\end{aligned}$$

اگر شمارنده از صفر شروع شود، آنگاه $\Phi(D_0) = 0$. از آنجا که برای تمام iها $\Phi(D_i) \ge 0$، لذا هزینه سرشکن شده کل یک توالی از n عمل *INCREMENT* یک حد بالا برای هزینه واقعی کل می‌باشد، و بنابراین هزینه n عمل *INCREMENT* در بدترین حالت برابر $O(n)$ است.

روش پتانسیل یک راه ساده برای تحلیل شمارنده، حتی زمانیکه از صفر شروع نمی‌شود را به ما می‌دهد. در ابتدا b_0 تا 1 وجود دارند و بعد از n عمل *INCREMENT*، b_n تا 1 وجود دارند، بطوریکه $0 \le b_0$ و $b_n \le k$. (بخاطر آورید که k تعداد بیت‌های داخل شمارنده است.) می‌توانیم معادله (۱۷.۳) را بصورت زیر بازنویسی کنیم

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \widehat{c_i} - \Phi(D_n) + \Phi(D_0)\ . \qquad (۱۷.۴)$$

برای تمام iها که $1 \le i \le n$، داریم $\widehat{c_i} \le 2$. از آنجا که $\Phi(D_0) = b_0$ و $\Phi(D_n) = b_n$، لذا هزینه واقعی کل n عمل *INCREMENT* برابر است با

$$\begin{aligned}\sum_{i=1}^{n} c_i &\le \sum_{i=1}^{n} 2 - b_n + b_0\\ &= 2n - b_n + b_0\ .\end{aligned}$$

بخصوص توجه کنید که چون $b_0 \le k$ تا زمانیکه $k = O(n)$ هزینه واقعی کل برابر $O(n)$ است.

بعبارت دیگر، اگر حداقل $n = \Omega(k)$ عمل *INCREMENT* را اجرا کنیم هزینه واقعی کل برابر $O(n)$ است، بدون توجه به اینکه شمارنده چه مقدار اولیه‌ای داشته باشد.

تمرین‌ها

۱ - ۱۷.۳ فرض کنید تابع پتانسیل Φ را داریم بطوریکه برای تمام iها، $\Phi(D_i) \geq \Phi(D_0)$، اما $\Phi(D_0) \neq 0$. نشان دهید تابع پتانسیل Φ' وجود دارد بطوریکه $\Phi'(D_0)=0$ و برای تمام $i \geq 1$، $\Phi(D_i) \geq 0$، و هزینه‌های سرشکن شده‌ای که از Φ' استفاده می‌کنند همان هزینه‌های سرشکن شده‌ای هستند که از Φ استفاده می‌کنند.

۲ - ۱۷.۳ تمرین ۳ - ۱۷.۱ را با استفاده از روش پتانسیل تحلیل سرشکن شده انجام دهید.

۳- ۱۷.۳ ساختمان داده *min-heap* دودویی معمولی با n عضو را درنظر بگیرید که اعمال *INSERT* و *EXTRACT-MIN* را در بدترین حالت در زمان $O(lgn)$ پشتیبانی می‌کند. یک تابع پتانسیل Φ ارائه کنید بطوریکه هزینه سرشکن شده *INSERT* برابر $O(lgn)$ باشد و هزینه سرشکن‌شده *EXTRACT-MIN* برابر $O(1)$ باشد و نشان دهید که این تابع کار می‌کند.

۴ - ۱۷.۳ هزینه کل اجرای n عمل پشته *PUSH*، *POP* و *MULTIPOP* با این فرض که پشته با s_0 شیء شروع شده و با s_n شیء خاتمه یابد، چیست؟

۵ - ۱۷.۳ فرض کنید یک شمارنده بجای 0 از یک عدد با b تا 1 در نمایش دودویی شروع می‌شود. نشان دهید که هزینه انجام n عمل *INCREMENT* اگر $n = \Omega(b)$ برابر $O(n)$ است. (b را یک ثابت فرض نکنید.)

۶ - ۱۷.۳ نشان دهید چگونه یک صف را با دو پشته معمولی (تمرین ۶-۱۰.۱) پیاده سازی کنیم تا هزینه سرشکن شده هر عمل *ENQUEUE* و هر عمل *DEQUEUE* برابر $O(1)$ باشد.

۷ - ۱۷.۳ ساختمان داده‌ای طراحی کنید که دارای دو عمل زیر برای مجموعه S از اعداد صحیح باشد:

INSERT(S, x) x را به S اضافه می‌کند.

DELETE-LARGER-HALF(S) بزرگترین $\lceil S/2 \rceil$ عضو را از S حذف می‌کند.

توضیح دهید چگونه این ساختمان داده را پیاده سازی کنیم تا هر توالی از m عمل در زمان $O(m)$ اجرا شود.

۱۷.۴ جداول پویا

در برخی از کاربردها از قبل نمی‌دانیم چه تعداد شیء در یک جدول ذخیره خواهند شد. ممکن است فضایی برای جدول اختصاص دهیم و تنها بعد از آن متوجه شویم که این فضا کافی نیست. پس دوباره باید برای جدول یک اندازه بزرگتر اختصاص داده شود، و تمام اشیائی که در جدول اول ذخیره شده‌اند

در جدول جدید بزرگتر کپی شوند. به طور مشابه اگر تعداد زیادی شیء از جدول حذف شده باشند، اختصاص یک اندازه کوچکتر به جدول می‌تواند به صرفه باشد. در این بخش مسئله انبساط و انقباض یک جدول بصورت پویا را مطالعه می‌کنیم. با استفاده از تحلیل سرشکن شده نشان خواهیم داد که هزینه سرشکن شده درج و حذف تنها برابر $O(1)$ است، هر چند هزینه واقعی یک عمل هنگامیکه موجب یک انبساط یا یک انقباض می‌شود زیاد است. بعلاوه خواهیم دید که چگونه تضمین کنیم که فضای غیر قابل استفاده در یک جدول پویا هرگز از یک کسر ثابت از فضای کل تجاوز نکند.

فرض می‌کنیم که جدول پویا دارای اعمال *TABLE-INSERT* و *TABLE-DELETE* است. *TABLE-INSERT* یک عنصر که یک **محل**[1] را اشغال می‌کند در جدول درج می‌کند، به عبارت دیگر یک فضا برای یک عنصر. همچنین *TABLE-DELETE* می‌تواند به عنوان حذف کننده یک عنصر از جدول و لذا آزاد کننده یک محل درنظر گرفته شود. جزئیات روش ساختمان داده‌ای که برای تشکیل جدول استفاده می‌شود مهم نیست؛ ممکن است از یک پشته (بخش ۱۰.۱)، از یک *heap* (فصل ۶) و یا از یک جدول *hash*(فصل ۱۱) استفاده کنیم. همچنین ممکن است همانطور که در بخش ۱۰.۳ عمل کردیم، از یک آرایه و یا مجموعه‌ای از آرایه‌ها برای پیاده سازی ذخیره اشیاء استفاده کنیم.

استفاده از مفهوم معرفی شده در تحلیل درهم سازی (فصل ۱۱) مناسب به نظر خواهد رسید. *ضریب بار*[2] $\alpha(T)$ یک جدول غیر تهی T را تعداد عناصر ذخیره شده در جدول تقسیم بر اندازه (تعداد محل‌های) جدول تعریف می‌کنیم. به یک جدول خالی (جدولی بدون عنصر) اندازه 0 اختصاص می‌دهیم و ضریب بار آن را 1 تعریف می‌کنیم. اگر ضریب بار یک جدول پویا از پایین با یک ثابت محدود شده باشد، فضای غیر قابل استفاده در جدول هرگز بیشتر از کسری ثابت از مقدار فضای کل نیست.

با تحلیل یک جدول پویا که در آن تنها درج انجام می‌شود شروع می‌کنیم. سپس حالت کلی‌تری که در آن هم درج و هم حذف انجام می‌شود را درنظر می‌گیریم.

۱۷.۴.۱ انبساط جدول

فرض کنید یک آرایه از محل‌ها برای ذخیره سازی یک جدول اختصاص داده می‌شود. هنگامیکه تمام محل‌ها استفاده شده باشند جدول پر می‌شود، یا به طور معادل هنگامیکه ضریب بار آن برابر 1 باشد.[3] در برخی از محیط‌های نرم‌افزاری، اگر تلاشی برای درج یک عنصر در یک جدول پر صورت گیرد هیچ چاره‌ای جز توقف با یک خطا وجود ندارد. با این وجود فرض خواهیم کرد که محیط نرم‌افزاری ما همانند بسیاری از محیط‌های پیشرفته و مدرن، دارای یک سیستم مدیریت حافظه است که می‌تواند

---

1. slot   2. load factor

۳ – در بعضی از وضعیت‌ها، همانند یک جدول *hash* آدرس باز، ممکن است بخواهیم جدول را پر درنظر بگیریم، اگر ضریب بار آن برابر ثابتی باشد که اکیداً کوچکتر از 1 است. (تمرین ۱ - ۱۷.۴ را ملاحظه نمایید.)

بلاکهای ذخیره سازی را بر اساس نیاز اختصاص داده و یا آزاد کند. بنابراین هنگامیکه یک عنصر در یک جدول پر درج می‌شود، می‌توانیم با اختصاص یک جدول جدید با محل‌های بیشتری نسبت به جدول قدیم، آنرا **منبسط**[1] کنیم. از آنجا که همواره لازم است جدول را در یک حافظه پیوسته قرار دهیم، لذا باید یک آرایه جدید برای جدول بزرگتر اختصاص دهیم و سپس عناصر را از جدول قدیم به جدول جدید کپی کنیم.

یک روش مکاشفه‌ای معمول آن است که یک جدول جدید که دو برابر جدول قدیم محل دارد را اختصاص دهیم. اگر تنها درج انجام شود ضریب بار جدول همواره حداقل برابر $1/2$ است، بنابراین مقدار فضای هرز شده هرگز از نصف فضای کل در جدول تجاوز نمی‌کند.

در شبه کد زیر، فرض می‌کنیم T شیئی باشد که جدول را نشان می‌دهد. فیلد $table[T]$ شامل یک اشاره‌گر به بلاک حافظه‌ای است که جدول را نشان می‌دهد. فیلد $num[T]$ شامل تعداد عناصر داخل جدول است، و فیلد $size[T]$ برابر تعداد کل محل‌ها در جدول است. در ابتدا جدول خالی است: $num[T] = size[T] = 0$.

```
TABLE-INSERT(T, x)
 1  if size[T] = 0
 2     then allocate table[T] with 1 slot
 3          size[T] ← 1
 4  if num[T] = size[T]
 5     then allocate new-table with 2 · size[T] slots
 6          insert all items in table[T] into new-table
 7          free table[T]
 8          table[T] ← new-table
 9          size[T] ← 2 · size[T]
10  insert x into table[T]
11  num[T] ← num[T] + 1
```

توجه کنید که در اینجا دو روال "درج" داریم: یکی خود روال *TABLE-INSERT* و دیگری **درج اولیه**[2] به داخل یک جدول در خطوط ۶ و ۱۰. می‌توانیم زمان اجرای *TABLE-INSERT* را بر حسب تعداد درج‌های اولیه با اختصاص هزینه 1 به هر درج اولیه تحلیل کنیم. فرض می‌کنیم هزینه اجرای واقعی *TABLE-INSERT* در زمان درج یک عنصر خطی است، به‌طوریکه هزینه اضافی اختصاص جدول اولیه در خط ۲ ثابت است و هزینه اضافی اختصاص دادن و آزاد کردن حافظه در خطوط ۵ و ۷ تحت الشعاع هزینه انتقال عناصر در خط ۶ قرار می‌گیرد. رویدادی که در آن عبارت *then* در خطوط ۹ - ۵

---

1. expand   2. elementary insertion

اجرا می‌شود را یک *انبساط*[1] می‌نامیم.

اکنون یک توالی از n عمل *TABLE-INSERT* را روی یک جدول خالی تحلیل می‌کنیم. هزینه c_i مربوط به iامین عمل چیست؟ اگر در جدول جاری جای کافی وجود داشته باشد (یا اگر این عمل، اولین عمل باشد)، آنگاه $c_i = 1$، زیرا تنها لازم است یک درج اولیه را در خط ۱۰ انجام دهیم. اگر جدول جاری پر باشد و یک انبساط انجام شود، آنگاه $c_i = i$: هزینه درج اولیه در خط ۱۰ برابر 1 است که با هزینه $i-1$ برای عناصری که از جدول قدیم به جدول جدید کپی می‌شوند (در خط ۶) جمع می‌شود. اگر n عمل انجام شوند، هزینه یک عمل در بدترین حالت برابر $O(n)$ است که به حد بالای $O(n^2)$ روی زمان اجرای کل n عمل می‌انجامد.

این حد قوی نیست، زیرا هزینه انبساط جدول اغلب در هنگام n عمل *TABLE-INSERT* بوجود نمی‌آید. بویژه iامین عمل تنها هنگامی سبب یک انبساط می‌شود که $i-1$ توانی صحیح از 2 است. همانطور که می‌توانیم با استفاده از تحلیل جمعی نشان دهیم، هزینه سرشکن شده یک عمل در حقیقت $O(1)$ است. هزینه i امین عمل برابر است با

$$c_i = \begin{cases} i & \text{اگر } i-1 \text{ توانی صحیح از } 2 \text{ باشد} \\ 1 & \text{در غیر اینصورت} \end{cases}$$

بنابراین هزینه کل n عمل *TABLE-INSERT* برابر است با

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n , \end{aligned}$$

زیرا حداکثر n عمل با هزینه 1 وجود دارند و هزینه‌های اعمال باقی‌مانده، یک سری هندسی را تشکیل می‌دهند. از آنجا که هزینه کل n عمل *TABLE-INSERT* برابر $3n$ است، هزینه سرشکن شده یک عمل به تنهایی برابر 3 است.

با استفاده از روش حسابداری پی خواهیم برد که چرا هزینه سرشکن شده یک عمل *TABLE-INSERT* باید 3 باشد. به طور شهودی هر عنصر برای 3 درج اولیه هزینه پرداخت می‌کند: درج خودش در جدول جاری، انتقال خودش هنگامیکه جدول منبسط می‌شود و انتقال عنصر دیگر که قبلاً یکبار هنگامیکه جدول منبسط می‌شود، منتقل گردیده است. برای مثال فرض کنید که اندازه جدول بلافاصله پس از یک انبساط برابر m است. پس تعداد عناصر داخل جدول برابر $m/2$ است و جدول شامل موجودی نیست. 3 دلار برای هر درج هزینه می‌کنیم. درج اولیه‌ای که بلافاصله رخ می‌دهد، 1 دلار

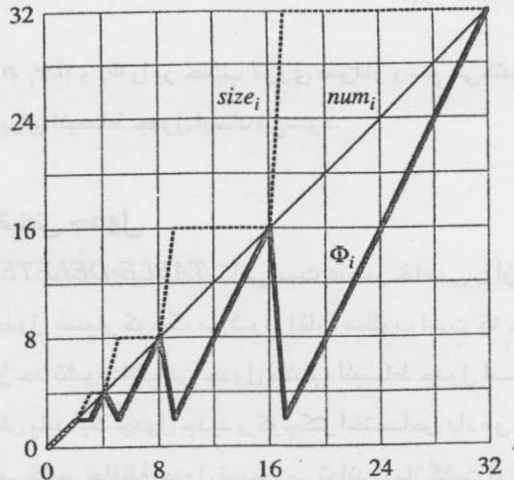
1. expansion

هزینه دارد. دلار بعدی بعنوان موجودی روی عنصری که درج می‌شود قرار می‌گیرد. دلار سوم بعنوان موجودی روی یکی از $m/2$ عنصر که قبلاً در جدول بوده‌اند قرار می‌گیرد. پر کردن جدول نیاز به $m/2-1$ درج اضافه دارد و بنابراین زمانیکه جدول شامل m عنصر و پر است، هر عنصر یک دلار جهت پرداخت برای درج دوباره‌اش در هنگام انبساط دارد.

روش پتانسیل نیز می‌تواند برای تحلیل یک توالی از n عمل *TABLE-INSERT* بکار رود. این روش را در بخش ۱۷.۴.۲ به منظور طراحی یک عمل *TABLE-DELETE* که داری هزینه سرشکن‌شده $O(1)$ است، استفاده خواهیم کرد. با تعریف تابع پتانسیل Φ شروع می‌کنیم که بلافاصله پس از انبساط برابر 0 است، اما هنگامیکه جدول پر است برابر اندازه جدول می‌شود لذا انبساط بعدی می‌تواند با پتانسیل پرداخت شود. تابع

$$\Phi(T) = 2 \cdot num[T]-size[T] \qquad (۱۷.۵)$$

می‌تواند یک تابع مطلوب باشد. بلافاصله پس از یک انبساط، داریم $num[T] = size[T]/2$ و بنابراین همانطور که می‌خواستیم $\Phi(T) = 0$. بلافاصله قبل از یک انبساط، داریم $num[T] = size[T]$، و بنابراین همانطور که می‌خواستیم $\Phi[T] = num\ [T]$. مقدار اولیه پتانسیل برابر 0 است و از آنجا که جدول همواره حداقل نیمه‌پر است، لذا $num\ [T] \geq size[T]/2$ که نشان می‌دهد $\Phi[T]$ همواره نامنفی است. بنابراین مجموع هزینه‌های سرشکن شده n عمل *TABLE-INSERT* یک حد بالا روی مجموع هزینه‌های واقعی است.



شکل ۱۷.۳ تأثیر یک توالی از n عمل *TABLE-INSERT* روی تعداد num_i عنصر در جدول، تعداد $size_i$ محل در جدول، و پتانسیل $\Phi_i = 2 \cdot num_i - size_i$، که هر کدام پس از iامین عمل اندازه‌گیری شده‌اند. خط نازک num_i، خط چین $size_i$ و خط ضخیم Φ_i را نشان می‌دهد. توجه کنید که بلافاصله قبل از یک انبساط، پتانسیل به تعداد عناصر داخل جدول تولید شده است و بنابراین برای انتقال تمام عناصر به جدول جدید می‌تواند پرداخت داشته باشد. در نتیجه پتانسیل به 0 کاهش می‌یابد، اما بلافاصله وقتی عنصری که سبب انبساط می‌شود درج می‌گردد، 2 واحد افزایش می‌یابد.

برای تحلیل هزینه iامین عمل $TABLE\text{-}INSERT$، num_i را تعداد عناصر ذخیره شده در جدول پس از i امین عمل، $size_i$ را اندازه کل جدول پس از i امین عمل، و Φ را پتانسیل پس از i امین عمل قرار می‌دهیم. در ابتدا داریم $num_0 = 0$ ، $size_0 = 0$ و $\Phi_0 = 0$.

اگر iامین عمل $TABLE\text{-}INSERT$ موجب انبساط نشود، آنگاه داریم $size_i = size_{i-1}$ و هزینه سرشکن شده عمل برابر است با

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\
&= 3 .
\end{aligned}$$

اگر iامین عمل موجب انبساط شود، آنگاه داریم $size = 2 . size_{i-1}$ و $size_{i-1} = num_{i-1} = num_i - 1$ ، که نشان می‌دهد $size_i = 2 . (num_i - 1)$. بنابراین هزینه سرشکن شده عمل برابر است با

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - (num_i - 1) \\
&= 3 .
\end{aligned}$$

شکل ۱۷.۳ مقادیر num_i $size_i$ و Φ_i را بر حسب i روی نمودار رسم می‌کند. توجه کنید که چگونه پتانسیل جهت پرداخت برای انبساط جدول ایجاد می‌شود.

۱۷.۴.۲ انبساط و انقباض جدول

برای پیاده سازی عمل $TABLE\text{-}DELETE$، کافی است عنصر خاصی را از جدول حذف کنیم. اگر چه هنگامیکه ضریب بار جدول بسیار کوچک می‌شود، اغلب مطلوب است که جدول را منقبض[1] کنیم تا فضای هدر رفته بیش از حد نشود. انقباض جدول مشابه انبساط جدول است: هنگامیکه تعداد عناصر داخل جدول بسیار کاهش یابد، یک جدول جدید و کوچکتر اختصاص داده و عناصر را از جدول قدیم به داخل جدول جدید کپی می‌کنیم. حافظهٔ جدول قدیمی می‌تواند با بازگشت آن به سیستم مدیریت حافظه آزاد گردد. به طور ایده‌آل تمایل به حفظ دو ویژگی داریم:

- ضریب بار جدول پویا با یک مقدار ثابت از پایین محدود می‌شود، و
- هزینه سرشکن شده یک عمل جدول از بالا با یک مقدار ثابت محدود می‌شود.

---

1. contract

فرض می‌کنیم که هزینه می‌تواند بر حسب درجها و حذف‌های اولیه اندازه‌گیری شود. یک استراتژی معمول برای انبساط و انقباض عبارت است از دو برابر کردن اندازه جدول، هنگام درج یک عنصر در یک جدول پر و نصف کردن اندازه، هنگامی که حذف عنصر سبب می‌شود تا جدول از نیمه پر کمتر شود. این استراتژی ضمانت می‌کند که ضریب بار جدول هرگز کمتر از $1/2$ نمی‌شود. اما متأسفانه این استراتژی می‌تواند سبب افزایش هزینه سرشکن شده یک عمل شود. روند زیر را درنظر بگیرید. n عمل را روی جدول T انجام می‌دهیم، به طوریکه n توانی صحیح از 2 است. اولین $n/2$ عمل، درجها هستند که با توجه به تحلیل قبلی ما دارای هزینه کل $\Theta(n)$ می‌باشند. در پایان این توالی از درجها، $num[T] = size[T] = n/2$ برای $n/2$ عمل دوم، توالی زیر را اجرا می‌کنیم:

$I, D, D, I, I, D, D, I, I, \ldots,$

که در آن I نشانه یک درج و D نشانه یک حذف است. اولین درج سبب انبساط جدول به اندازه n می‌شود. دو حذف بعدی سبب انقباض جدول به اندازه $n/2$ می‌شوند. دو درج بعدی سبب یک انبساط دیگر شده، و... . هزینه هر انبساط و انقباض برابر $\Theta(n)$ است و $\Theta(n)$ تعداد از این اعمال وجود دارد. بنابراین هزینه کل n عمل برابر است با $\Theta(n^2)$، و هزینه سرشکن شده یک عمل برابر است با $\Theta(n)$.

اشکال این استراتژی واضح است: بعد از یک انبساط، جهت پرداخت هزینه برای یک انقباض به تعداد کافی عمل حذف انجام نمی‌دهیم. بعلاوه پس از یک انقباض، به تعداد کافی عمل درج جهت پرداخت هزینه برای یک انبساط انجام نمی‌دهیم.

می‌توانیم این استراتژی را با قبول کاهش ضریب بار به کمتر از $1/2$، بهتر از قبل کنیم. دو برابر کردن اندازه جدول هنگامیکه یک عنصر به داخل جدول پر درج می‌شود را ادامه می‌دهیم اما هنگامیکه یک حذف باعث می‌شود تا جدول از $1/4$ پر کمتر شود، بجای $1/2$ پر مانند قبل، اندازه آن را نصف می‌کنیم. لذا ضریب بار جدول با ثابت $1/4$ از پایین محدود می‌شود. ایده آن است که پس از یک انبساط، ضریب بار جدول برابر $1/2$ است. بنابراین نیمی از عناصر داخل جدول باید قبل از آنکه یک انقباض بتواند رخ دهد حذف گردند، زیرا انقباض رخ نمی‌دهد مگر آنکه ضریب بار به کمتر از $1/4$ کاهش یابد. همچنین پس از یک انقباض، ضریب بار دوباره برابر $1/2$ است. بنابراین تعداد عناصر داخل جدول باید قبل از آنکه یک انبساط بتواند رخ دهد بوسیله درجها دو برابر شوند، زیرا انبساط تنها هنگامی رخ می‌دهد که ضریب بار از 1 تجاوز کند.

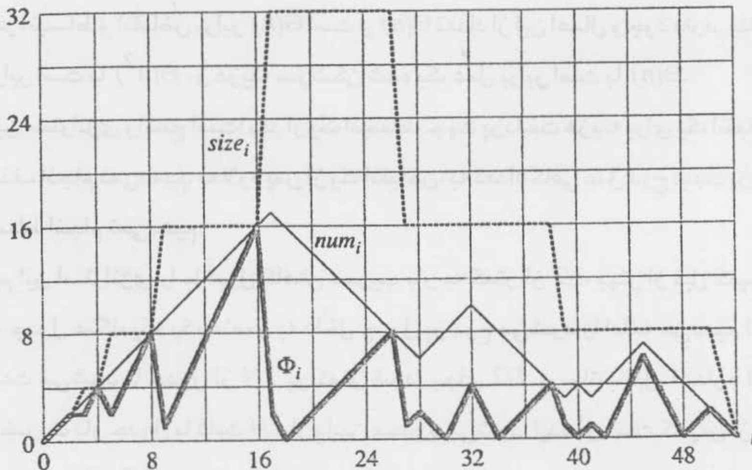
کد مربوط به *TABLE-DELETE* را ذکر نکردیم زیرا مشابه کد *TABLE-INSERT* است. اما به منظور سهولت در تحلیل فرض می‌کنیم که اگر تعداد عناصر داخل جدول به 0 کاهش یابد، حافظه جدول آزاد می‌شود. به عبارت دیگر اگر $num[T] = 0$ آنگاه $size[T] = 0$

اکنون می‌توانیم از روش پتانسیل برای تحلیل هزینه یک توالی از n عمل *TABLE-INSERT* و *TABLE-DELETE* استفاده کنیم. با تعریف تابع پتانسیل Φ شروع می‌کنیم. این تابع پس از یک

انبساط یا انقباض برابر 0 است و به محض آنکه ضریب بار به 1 افزایش یافته و یا به $1/4$ کاهش می‌یابد ایجاد می‌شود. فرض کنید $\alpha(T) = num[T]/size[T]$ به ضریب بار جدول غیر تهی T اشاره می‌کند. از آنجا که برای یک جدول خالی، $num[T] = size[T] = 0$ و $\alpha(T) = 1$، لذا همواره داریم $num[T] = \alpha(T) . size[T]$، خواه جدول خالی باشد یا خالی نباشد. بعنوان تابع پتانسیل از تابع زیر استفاده خواهیم کرد.

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{if } \alpha(T) \geq 1/2 , \\ size[T]/2 - num[T] & \text{if } \alpha(T) < 1/2 . \end{cases} \qquad (۱۷.۶)$$

مشاهده می‌شود که پتانسیل یک جدول خالی برابر 0 است و پتانسیل هرگز منفی نمی‌باشد. بنابراین هزینه سرشکن شده کل یک توالی از اعمال با توجه به Φ، یک حد بالا برای هزینه واقعی توالی است. قبل از ادامه با یک تحلیل دقیق، اندکی درنگ می‌کنیم تا برخی از ویژگی‌های تابع پتانسیل را مشاهده کنیم.



شکل ۱۷.۴ تأثیر یک توالی از n عمل TABLE-INSERT و TABLE-DELETE روی تعداد num_i عنصر در جدول، تعداد $size_i$ محل در جدول، و پتانسیل که هر کدام پس از iامین عمل اندازه گیری شده‌اند.

$$\Phi_i = \begin{cases} 2 \cdot num_i - size_i & \text{if } \alpha_i \geq 1/2 , \\ size_i /2 - num_i & \text{if } \alpha_i < 1/2 , \end{cases}$$

خط نازک num_i، خط چین $size_i$، و خط ضخیم Φ_i را نشان می‌دهد. توجه کنید که بلافاصله قبل از یک انبساط، پتانسیل به تعداد عناصر داخل جدول ساخته شده است، و بنابراین می‌تواند برای انتقال تمام عناصر به جدول جدید پرداخت داشته باشد. به همین شکل بلافاصله قبل از یک انقباض، پتانسیل به تعداد عناصر داخل جدول ساخته شده است.

توجه کنید هنگامیکه ضریب بار برابر $1/2$ است، پتانسیل برابر 0 است. وقتی ضریب بار برابر 1 است، داریم $size[T] = num[T]$ که نشان می‌دهد $\Phi(T) = num[T]$، و بنابراین پتانسیل می‌تواند برای یک انبساط، اگر یک عنصر درج شود، پرداخت گردد. هنگامی که ضریب بار برابر $1/4$ است، داریم

$size[T] = 4 \cdot num[T]$ که نشان می‌دهد $\Phi(T) = num[T]$، و بنابراین پتانسیل می‌تواند برای یک انقباض، اگر یک عنصر حذف شود، پرداخت گردد. شکل ۱۷.۴ چگونگی رفتار پتانسیل برای یک توالی از اعمال را توضیح می‌دهد.

برای تحلیل یک توالی از n عمل TABLE-INSERT و TABLE-DELETE، c_i را هزینه واقعی i امین عمل، $\widehat{c}_i$ را هزینه سرشکن شده آن با توجه به Φ، num را تعداد عناصر ذخیره شده در جدول پس از i امین عمل، $size_i$ را اندازه کل جدول پس از i امین عمل، α_i را ضریب بار جدول پس از i امین عمل، و Φ_i را پتانسیل پس از i امین عمل قرار می‌دهیم. در ابتدا، $num_0 = 0$، $size_0 = 0$، $\alpha_0 = 1$ و $\Phi_0 = 0$.

با حالتی که در آن i امین عمل، TABLE-INSERT است شروع می‌کنیم. اگر $\alpha_{i-1} \geq 1/2$، تحلیل همانند تحلیل انبساط جدول در بخش ۱۷.۴-۱ می‌شود. خواه جدول انبساط پیدا کند یا نکند، هزینه سرشکن شده $\widehat{c}_i$ عمل حداکثر 3 است. اگر $\alpha_{i-1} < 1/2$، جدول نمی‌تواند به عنوان نتیجه عمل انبساط پیدا کند، زیرا انبساط تنها زمانیکه $\alpha_{i-1} = 1$ است رخ می‌دهد. اگر $\alpha_i < 1/2$، آنگاه هزینه سرشکن شده i امین عمل برابر است با

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\
&= 0 .
\end{aligned}$$

اگر $\alpha_{i-1} < 1/2$ اما $\alpha_i \geq 1/2$، آنگاه

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\
&= 3 \cdot num_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3\alpha_{i-1} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&< \frac{3}{2} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3 .
\end{aligned}$$

بنابراین هزینه سرشکن شده عمل TABLE-INSERT حداکثر 3 است.

اکنون به حالتی که در آن i امین عمل، TABLE-DELETE است می‌پردازیم. در این حالت، $num_i = num_{i-1} - 1$. اگر $\alpha_{i-1} < 1/2$ آنگاه باید درنظر بگیریم که آیا عمل موجب انقباض می‌شود یا خیر. اگر موجب انقباض نشود، آنگاه $size_i = size_{i-1}$ و هزینه سرشکن شده عمل برابر است با

$$
\begin{aligned}
\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i /2 - num_i) - (size_{i-1} /2 - num_{i-1}) \\
&= 1 + (size_i /2 - num_i) - (size_i /2 - (num_i +1)) \\
&= 2 .
\end{aligned}
$$

اگر $\alpha_{i-1} < 1/2$ و i امین عمل سبب یک انقباض گردد، آنگاه هزینه واقعی عمل برابر است با $c_i = num_i+1$ زیرا یک عنصر را حذف کرده و num_i عنصر را منتقل می‌کنیم. داریم

$size_i /2 = size_{i-1} /4 = num_{i-1} = num_i+1$

و هزینه سرشکن شده عمل برابر است با

$$
\begin{aligned}
\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i +1) + (size_i /2 - num_i) - (size_{i-1} /2 - num_{i-1}) \\
&= (num_i +1) + ((num_i +1) - num_i) - ((2 \cdot num_i +2) - (num_i +1)) \\
&= 1 .
\end{aligned}
$$

هنگامیکه i امین عمل، *TABLE-DELETE* است و $\alpha_{i-1} \geq 1/2$، هزینه سرشکن شده نیز با یک ثابت از بالا محدود می‌شود. تحلیل به عنوان تمرین ۲ - ۱۷.۴ واگذار می‌شود.

به طور خلاصه از آنجا که هزینه سرشکن شده هر عمل از بالا با یک ثابت محدود می‌شود، زمان واقعی هر توالی از n عمل روی یک جدول پویا $O(n)$ است.

تمرین‌ها

۱ - ۱۷.۴ فرض کنید می‌خواهیم یک جدول *hash* آدرس باز و پویا را پیاده سازی کنیم. هنگامی که ضریب بار جدول به مقدار α ای که اکیداً کوچکتر از 1 است می‌رسد، چرا ممکن است جدول را پر درنظر بگیریم؟ مختصراً توضیح دهید که چگونه درج در یک جدول *hash* آدرس باز پویا را طوری انجام دهیم که مقدار هزینه سرشکن شده پیش بینی شده هر درج برابر $O(1)$ شود. چرا مقدار هزینه واقعی پیش‌بینی شده هر درج برای تمام درجها لزوماً $O(1)$ نیست؟

۲ - ۱۷.۴ نشان دهید اگر $\alpha_{i-1} \geq 1/2$ و iامین عمل روی یک جدول پویا، *TABLE-DELETE* باشد آنگاه هزینه سرشکن شده عمل با توجه به تابع پتانسیل (۱۷.۶) با یک ثابت از بالا محدود می‌شود.

۳ - ۱۷.۴ فرض کنید بجای انقباض جدول با نصف کردن اندازه آن وقتی که ضریب بار جدول به کمتر از $1/4$ کاهش می‌یابد، آنرا با ضرب اندازه‌اش در $2/3$ هنگامیکه ضریب بار آن به کمتر از $1/3$ کاهش می‌یابد منقبض کنیم. با استفاده از تابع پتانسیل

$\Phi(T) = |2 . num[T]\text{-}size[T]|$

نشان دهید که هزینه سرشکن شده یک *TABLE-DELETE* که این استراتژی را استفاده می‌کند، با یک ثابت از بالا محدود می‌شود.

مسائل

۱۷-۱ شمارنده دودویی بیت - وارون

یک الگوریتم مهم بنام *Fast Fourier Transform* یا *FFT* وجود دارد. اولین گام الگوریتم *FFT*، یک **جایگشت بیت - وارونگی**[1] روی آرایه ورودی $A[0 \,.\,.\, n-1]$ که طولش به ازای عدد صحیح نامنفی k برابر $n = 2^k$ است، انجام می‌دهد. این جایگشت، اعضایی را که اندیسهایشان نمایش دودویی دارند و معکوس یکدیگرند، با هم عوض می‌کند.

می‌توانیم هر اندیس را بصورت توالی k - بیتی $<a_{k-1}, a_{k-2}, \ldots, a_0>$ بیان کنیم که

$$a = \sum_{i=0}^{k-1} a_i 2^i$$

تعریف می‌کنیم

$$\text{rev}_k(\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle) = \langle a_0, a_1, \ldots, a_{k-1} \rangle ;$$

بنابراین

$$\text{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i .$$

برای مثال اگر $n = 16$ (یا به طور معادل $k = 4$)، آنگاه $rev_k(3) = 12$، زیرا 4 بیت نمایش دهنده 3 برابر *0011* است که وقتی معکوس می‌شوند، *1100* حاصل می‌شود که نمایش 4 بیتی عدد *12* است.

a. برای تابع مفروض rev_k که در زمان $\Theta(k)$ اجرا می‌شود، الگوریتمی بنویسید تا جایگشت بیت - وارونگی را روی آرایه‌ای با طول $n = 2^k$ در زمان $O(nk)$ انجام دهد.

می‌توانیم از الگوریتمی که بر مبنای تحلیل سرشکن شده است برای بهبود زمان اجرای جایگشت بیت - وارونگی استفاده کنیم. یک "شمارنده بیت - وارون" و روال *BIT-REVERSED-INCREMENT* را که با دریافت مقدار شمارنده بیت - وارون یعنی a، $rev_k(rev_k(a) + 1)$ را تولید می‌کند، درنظر می‌گیریم. اگر برای مثال $k = 4$ و شمارنده بیت - وارون از *0* شروع شود، آنگاه فراخوانی‌های متوالی *BIT-REVERSED-INCREMENT* توالی زیر را تولید می‌کنند

0000, 1000, 0100, 1100, 0010, 1010, . . . = 0, 8, 4, 12, 2, 10, . . .

b. فرض کنید که کلمات در کامپیوتر شما مقادیر k - بیتی ذخیره می‌کنند و اینکه کامپیوتر شما در

---

1. *bit-reversal permutation*

زمان واحد می‌تواند مقادیر دودویی را با اعمالی نظیر شیفت به چپ یا راست به مقدار دلخواه، *AND* بیتی، *OR* بیتی و غیره دستکاری کنند. یک پیاده سازی از روال *BIT-REVERSED-INCREMENT* ارائه نمایید که انجام جایگشت بیت - وارونگی روی یک آرایه n عنصری را در زمان کل $O(n)$ ممکن سازد.

c. فرض کنید می‌توانید یک کلمه را در زمان واحد یک بیت به چپ یا به راست شیفت دهید. آیا هنوز پیاده سازی جایگشت بیت - وارونگی در زمان $O(n)$ ممکن است؟

۱۷-۲ پویاکردن جستجوی دودویی

جستجوی دودویی یک آرایه مرتب شده، زمانی لگاریتمی را صرف می‌کند، اما زمان درج یک عنصر جدید نسبت به اندازه آرایه خطی است. می‌توانیم زمان درج را با نگهداری چندین آرایه مرتب بهبود بخشیم.

بویژه فرض کنید می‌خواهیم *SEARCH* و *INSERT* را روی مجموعه‌ای از n عضو داشته باشیم. قرار دهید $k = \lceil lg(n+1) \rceil$، و نمایش دودویی n را بصورت $<n_{k-2}, n_{k-2}, \ldots, n_0>$ درنظر بگیرید. k آرایه مرتب شده $A_0, A_1, \ldots, A_{k-1}$ داریم که برای $i = 0, 1, \ldots, k-1$ طول آرایه A_i برابر 2^i است. هر آرایه بترتیب بسته به آنکه $n_i = 1$ یا $n_i = 0$، یا پر است یا خالی. بنابراین تعداد کل عناصر موجود در تمام k آرایه برابر است با

$$\sum_{i=0}^{k-1} n_i \, 2^i = n$$

اگر چه هر آرایه به تنهایی مرتب می‌شود اما هیچ ارتباطی بین عناصر در آرایه‌های متفاوت وجود ندارد.

a. توضیح دهید چگونه عمل *SEARCH* را برای این ساختمان داده انجام دهیم. زمان اجرای آنرا در بدترین حالت تحلیل کنید.

b. توضیح دهید چگونه یک عنصر جدید را در این ساختمان داده درج کنیم. زمانهای اجرای سرشکن‌شده و بدترین حالت را برای آن تحلیل کنید.

c. در مورد چگونگی پیاده سازی *DELETE* بحث کنید.

۱۷-۳ درختهای متوازن[1] سرشکن شده

یک درخت جستجوی دودویی معمولی را درنظر بگیرید که با افزودن فیلد $size[x]$ به هر گره x توسعه یافته است. $size[x]$ برابر با تعداد کلیدهای ذخیره شده در زیردرخت با ریشه x است. α را ثابتی در بازه

---

1. *weight-balanced trees*

$1/2 \leq \alpha < 1$ درنظر بگیرید. می‌گوییم گره x، *α - تراز*[1] است، اگر

$size[left[x]] \leq \alpha \,.\, size[x]$

و

$size[right[x]] \leq \alpha \,.\, size[x]$

درخت در مجموع *α - تراز* است، اگر هر گره در درخت α - تراز باشد. روش سرشکن شده زیر برای نگهداری درخت‌های متوازن توسط *G.Varghese* پیشنهاد شده است.

a. یک درخت $1/2$- تراز تا اندازه‌ای که می‌تواند موازنه است. اگر گره x در یک درخت جستجوی دودویی دلخواه داده شود، نشان دهید چگونه زیردرخت با ریشه x را بازسازی کنیم تا $1/2$ - تراز شود. الگوریتم شما باید در زمان $\Theta(size[x])$ اجرا شود و می‌تواند از حافظه کمکی $O(size[x])$ نیز استفاده کند.

b. نشان دهید جستجو در یک درخت جستجوی دودویی α - تراز n گرهی در بدترین حالت زمان $O(lgn)$ را صرف می‌کند.

برای باقی این مسئله فرض کنید α اکیداً بزرگتر از $1/2$ است. فرض کنید *INSERT* و *DELETE* به شکل معمول برای درخت جستجوی n گرهی پیاده سازی شوند، با این تفاوت که پس از هر یک از این اعمال، اگر گرهی در درخت دیگر α - تراز نباشد آنگاه زیردرخت مشتق شده از مرتفع‌ترین گره با این ویژگی در درخت "بازسازی" می‌شود تا $1/2$ - تراز شود.

با استفاده از روش پتانسیل، این طرح بازسازی را تحلیل خواهیم کرد. در درخت جستجوی دودویی T تعریف می‌کنیم

$$\Delta(x) = |size[left[x]] - size[right[x]]| \;,$$

و پتانسیل T را بصورت زیر تعریف می‌کنیم

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x) \;,$$

که در آن c یک ثابت به اندازه کافی بزرگ است.

c. ثابت کنید هر درخت جستجوی دودویی پتانسیل نامنفی دارد و اینکه یک درخت $1/2$ - تراز پتانسیل 0 دارد.

d. فرض کنید m واحد از پتانسیل می‌تواند برای بازسازی یک زیردرخت m گرهی پرداخت شود. c برحسب α به چه میزان باید بزرگ باشد تا زمان سرشکن شده بازسازی زیردرختی که α - تراز نیست، $O(1)$ شود؟

e. نشان دهید هزینه درج یک گره در یک درخت α - تراز n گرهی و یا حذف یک گره از آن، زمان

---

1. *α-balanced*

سرشکن‌شده $O(lgn)$ است.

۱۷-۴ هزینه بازسازی درختهای قرمز - سیاه

چهار عمل اصلی روی درخت‌های قرمز - سیاه وجود دارند که *تغییرات ساختاری*[1] را انجام می‌دهند: درج گره، حذف گره، چرخش و تغییر رنگ. مشاهده کرده‌ایم که *RB-INSERT* و *RB-DELETE* تنها از $O(1)$ چرخش، درج گره و حذف گره برای حفظ ویژگی‌های قرمز - سیاه استفاده می‌کنند، اما ممکن است تغییر رنگ‌های بسیار بیشتری را انجام دهند.

a. یک درخت قرمز - سیاه مجاز با n گره تعریف کنید بطوریکه فراخوانی *RB-INSERT* برای اضافه کردن $(n+1)$ امین گره، سبب $\Omega(lgn)$ تغییر رنگ شود. سپس یک درخت قرمز - سیاه مجاز با n گره تعریف کنید که برای آن فراخوانی *RB-DELETE* روی یک گره خاص، سبب $\Omega(lgn)$ تغییر رنگ شود.

اگر چه تعداد تغییر رنگ‌های هر عمل در بدترین حالت می‌تواند لگاریتمی باشد، ثابت خواهیم کرد که هر توالی از m عمل *RB-INSERT* و *RB-DELETE* روی یک درخت قرمز - سیاه که در ابتدا تهی است، سبب $O(m)$ تغییر ساختار در بدترین حالت می‌شود.

b. برخی از حالتهایی که بوسیلهٔ حلقه اصلی کد *RB-INSERT* و *RB-DELETE* اداره می‌شوند *پایان‌پذیر*[2] هستند: وقتی این حالتها رخ می‌دهند، سبب می‌شوند تا حلقه پس از تعدادی ثابت اعمال اضافه پایان پذیرد. برای هر یک از حالتهای *RB-INSERT-FIXUP* و *RB-DELETE-FIXUP*، مشخص کنید کدامیک پایان‌پذیر و کدامیک پایان‌ناپذیر می‌باشند. (**راهنمایی**: شکل‌های ۱۳.۵، ۱۳.۶ و ۱۳.۷ را مشاهده کنید.)

ابتدا تغییرات ساختاری را، هنگامیکه تنها درجها انجام می‌شوند، تحلیل می‌کنیم. T را درخت قرمز - سیاه درنظر گرفته و $\Phi(T)$ را تعداد گره‌های قرمز در T تعریف کنید. فرض کنید 1 واحد از پتانسیل می‌تواند برای تغییرات ساختاری که توسط هر یک از سه حالت *RB-INSERT-FIXUP* انجام شوند، پرداخت شود.

c. T' را نتیجه بکارگیری حالت ۱ از *RB-INSERT-FIXUP* در T درنظر بگیرید. ثابت کنید $\Phi(T') = \Phi(T)-1$.

d. درج گره در یک درخت قرمز - سیاه با استفاده از *RB-INSERT* می‌تواند به سه قسمت شکسته شود. تغییرات ساختاری و تغییرات پتانسیل که از خطوط ۱۶ - ۱ از *RB-INSERT*، از حالتهای پایان‌ناپذیر *RB-INSERT-FIXUP* و از حالتهای پایان‌پذیر *RB-INSERT-FIXUP* نتیجه می‌شوند را لیست کنید.

---

1. *structural modifications* 2. *terminating*

e. با استفاده از قسمت (d)، ثابت کنید تعداد سرشکن شده تغییرات ساختاری انجام شده با هر فراخوانی *RB-INSERT* برابر $O(1)$ است.

اکنون می‌خواهیم ثابت کنیم هنگامیکه هم درج و هم حذف وجود دارند، $O(m)$ تغییر ساختاری وجود خواهد داشت. برای هر گره x تعریف می‌کنیم

$$w(x) = \begin{cases} 0 & \text{اگر } x \text{ قرمز باشد} \\ 1 & \text{اگر } x \text{ سیاه باشد و فرزند قرمز نداشته باشد} \\ 0 & \text{اگر } x \text{ سیاه باشد و یک فرزند قرمز داشته باشد} \\ 2 & \text{اگر } x \text{ سیاه باشد و دو فرزند قرمز داشته باشد} \end{cases}$$

اکنون پتانسیل درخت قرمز – سیاه T را بصورت زیر، دوباره تعریف می‌کنیم

$$\Phi(T) = \sum_{x \in T} w(x) \ ,$$

و T' را درخت حاصل از بکارگیری هر حالت پایان‌ناپذیر از *RB-INSERT-FIXUP* یا *RB-DELETE-FIXUP* در T درنظر می‌گیریم.

f. نشان دهید برای تمام حالتهای پایان‌ناپذیر *RB-INSERT-FIXUP*، داریم $\Phi(T') = \leq \Phi(T)-1$. ثابت کنید تعداد سرشکن شده تغییرات ساختاری انجام شده با هر فراخوانی *RB-INSERT-FIXUP* برابر $O(1)$است.

g. نشان دهید برای تمام حالتهای پایان‌ناپذیر *RB-INSERT-FIXUP*، داریم $\Phi(T') = \leq \Phi(T)-1$. ثابت کنید تعداد سرشکن شده تغییرات ساختاری انجام شده با هر فراخوانی *RB-DELETE-FIXUP* برابر $O(1)$است.

h. اثبات اینکه هر توالی از m عمل *RB-INSERT* و *RB-DELETE* در بدترین حالت، $O(m)$ تغییر ساختاری انجام می‌دهد را کامل کنید.

V

ساختمان داده‌های پیشرفته

مقدمه

این قسمت به بررسی ساختمان داده‌هایی که اعمال روی مجموعه‌های پویا را پشتیبانی می‌کنند، اما در سطحی پیشرفته‌تر از قسمت III، می‌پردازد. برای مثال دو فصل از این قسمت از تکنیکهای تحلیل سرشکن شده که در فصل ۱۷ مشاهده کردیم، استفاده گسترده‌ای می‌کنند.

فصل ۱۸، *B-tree*ها را معرفی می‌کند. *B-tree*ها درخت‌های جستجوی متوازنی هستند که مخصوص ذخیره سازی روی دیسکهای مغناطیسی طراحی شده‌اند. از آنجا که دیسکهای مغناطیسی بسیار کندتر از حافظه با دستیابی تصادفی *(RAM)* عمل می‌کنند، لذا کارآیی *B-tree*ها را تنها با مقدار زمان محاسبه که اعمال مجموعه‌های پویا تلف می‌کنند اندازه گیری نمی‌کنیم، بلکه با تعداد دستیابی‌ها که به دیسک انجام می‌شوند کارآیی آنها را اندازه گیری می‌کنیم. برای هر عمل *B-tree*، تعداد دستیابی‌ها به دیسک با ارتفاع *B-tree* افزایش می‌یابد. این ارتفاع با اعمال *B-tree* پایین نگه داشته می‌شود.

فصل‌های ۱۹ و ۲۰ پیاده سازی‌های *heap*های ادغام‌پذیر را ارائه می‌کنند. این *heap*ها اعمال *INSERT*، *MINIMUM*، *EXTRACT-MIN* و *UNION* را پشتیبانی می‌کنند.[1] عمل *UNION* دو *heap* را واحدسازی می‌کند. ساختمان داده‌های این فصل اعمال *DELETE* و *DECREASE-KEY* را نیز پشتیبانی می‌کنند.

*heap*های دو جمله‌ای که در فصل ۱۹ ارائه می‌شوند، هر یک از این اعمال را در بدترین حالت در زمان $O(lg n)$ پشتیبانی می‌کنند، که n تعداد کل اعضا در *heap* ورودی است (یا در حالت *UNION* در دو *heap* ورودی). هنگامیکه عمل *UNION* باید پشتیبانی شود، *heap*های دوجمله‌ای از *heap*های دودویی که در فصل ۶ معرفی شدند مناسبترند، زیرا واحدسازی دو *heap* دودویی در بدترین حالت، زمان $\Theta(n)$ را صرف می‌کند.

---

(۱) در مسئله ۲-۱۰، یک *heap* قابل ادغام برای پشتیبانی *MINIMUM* و *EXTRACT-MIN* تعریف کرده‌ایم، و بنابراین می‌توانیم به آن به عنوان یک *min-heap* ادغام رجوع کنیم.
متناوباً اگر این *heap* از *MAXIMUM* و *EXTRACT-MAX* پشتیبانی می‌کرد، یک *max-heap* قابل ادغام بود. چنانچه غیر از این مشخص کنیم، *heap*های قابل ادغام به طور پیش فرض، *min-heap*های قابل ادغام خواهند بود.

*heap*های فیبوناچی در فصل ۲۰، *heap*های دو جمله‌ای را حداقل از لحاظ تئوری بهبود می‌بخشند. از حدود زمانی سرشکن شده برای اندازه گیری کارآیی *heap*های فیبوناچی استفاده می‌کنیم. اعمال *MINIMUM INSERT* و *UNION* روی *heap*های فیبوناچی تنها زمان سرشکن شده و زمان واقعی $O(1)$ را صرف می‌کنند، و اعمال *EXTRACT-MIN* و *DELETE*، زمان سرشکن شده $O(lgn)$ را صرف می‌کنند. اگر چه مهمترین مزیت *heap*های فیبوناچی آنست که *DECREASE-KEY* تنها زمان سرشکن شده $O(1)$ را صرف می‌کند. زمان سرشکن شده کم عمل *DECRE ASE-KEY* باعث شده است تا *heap*های فیبوناچی، اجزای کلیدی برخی از سریعترین الگوریتم‌ها بطور مجانبی برای مسائل گراف تا امروز باشند.

در نهایت، فصل ۲۱ ساختمان داده‌هایی را برای مجموعه‌های جدا از هم معرفی می‌کند. یک مجموعه مرجع از n عضو داریم که در مجموعه‌های پویا گروه بندی شده‌اند. در ابتدا هر عضو متعلق به تک مجموعه خودش است. عمل *UNION* دو مجموعه را واحدسازی می‌کند، و پرس و جوی *FIND-SET* مجموعه‌ای که یک عضو مورد نظر در همان لحظه در آن قرار دارد را مشخص می‌کند. با نمایش هر مجموعه بوسیلهٔ یک درخت مشتق شده ساده به اعمال شگفت انگیزانه سریعی دست می‌یابیم: یک توالی از m عمل در زمان $O(m\,\alpha(n))$ اجرا می‌شود، که $\alpha(n)$ تابعی است که به طور باور نکردنی آهسته رشد می‌کند - $\alpha(n)$ در هر کاربرد قابل تصور حداکثر 4 است. تحلیل سرشکن شده که این حد زمانی را اثبات می‌کند، به همان اندازه که این ساختمان داده ساده است، پیچیده می‌باشد.

عناوینی که در این قسمت پوشش داده می‌شوند، به هیچ وجه تنها نمونه‌های ساختمان داده‌های "پیشرفته" نمی‌باشند. سایر ساختمان داده‌های پیشرفته عبارتند از:

- *درخت‌های پویا*[1]، که بوسیلهٔ *Tarjan* و *Sleator* معرفی شدند و توسط *Tarjan* مورد بحث قرار گرفتند. این درخت‌ها جنگلی از درخت‌های مشتق شده جدا از هم را نگهداری می‌کنند. هر یال در هر درخت، یک هزینه با مقدار حقیقی دارد. درخت‌های پویا پرس و جوهایی برای یافتن پدرها، ریشه‌ها، هزینه‌های یالها و مینیمم هزینه یال در مسیری از یک گره به سمت ریشه را پشتیبانی می‌کنند. درخت‌ها ممکن است با قطع یالها، به روز کردن هزینه‌های تمام یالها در مسیری از یک گره به سمت ریشه، اتصال یک ریشه به درخت دیگر، و ریشه قرار دادن گرهی که در درخت است، دستکاری شوند. یک پیاده سازی درخت‌های پویا، حد زمانی سرشکن شده $O(lgn)$ را برای هر عمل حاصل می‌کند؛ یک پیاده سازی پیچیده‌تر، حدود زمان $O(lgn)$ را در بدترین حالت حاصل می‌کند. درخت‌های پویا در برخی از الگوریتم‌های به طور مجانبی سریع شبکه‌های جریان استفاده می‌شوند.
- *درخت‌های پهن*[2]، که بوسیله *Tarjan* و *Sleator* توسعه یافته و بوسیلهٔ *Tarjan* مورد بحث قرار گرفتند. این درخت‌ها شکلی از درخت جستجوی دودویی می‌باشند که در آنها اعمال جستجوی

---

1. *dynamic trees* 2. *splay trees*

استاندارد درخت در زمان سرشکن شده *O(lgn)* اجرا می‌شوند.

- ساختمان داده‌های *پایدار*[1]، امکان پرس و جو و بعلاوه گاهی بروز رسانی از روی نسخه‌های قبلی یک ساختمان داده را فراهم می‌کنند. *Sleator Sarnak Driscoll* و *Tarjan* تکنیکهای متصل کردن ساختمان داده‌های پایدار را با هزینه فضایی و زمانی کم معرفی کرده‌اند. مسئله ۱ - ۱۳ نمونه‌ای ساده از یک مجموعه پویای پایدار را ارائه می‌کند.
- ساختمان داده‌های متعددی موجب پیاده سازی سریعتر اعمال لغت نامه‌ای (*DELETE، INSERT* و *SEARCH*) برای مجموعه‌ای محدود از کلیدها می‌شوند. با استفاده از مزیت این محدودیت‌های، آنها قادرند تا به زمانهای اجرای مجانبی بهتری نسبت به ساختمان داده‌های مبتنی بر مقایسه در بدترین حالت دست یابند. ساختمان داده‌ای که بوسیلهٔ *Emde Boas* ابداع گردید، از اعمال *INSERT*، *MAXIMUM*، *MINIMUM*، *SUCCESSOR*، *EXTRACT - MAX*، *EXTRACT-MIN*، *SEARCH*، *DELETE* و *PREDECESSOR* در بدترین حالت در زمان *O (lglgn)* پشتیبانی می‌کند، مشروط به این محدودیت که مجموعه مرجع کلیدها، مجموعه *{1, 2, . . . , n}* باشد. *Fredman* و *Willard* *درخت‌های همجوش*[2] را معرفی کردند که اولین ساختمان داده‌ای بود که اعمال لغت نامه‌ای سریعتر را، هنگامیکه مجموعه مرجع به اعداد صحیح محدود می‌شود، ممکن می‌ساخت. آنها نحوه پیاده سازی این اعمال را در زمان *O(lgn/lglgn)* نشان دادند. بسیاری از ساختمان داده‌های بعدی نیز که شامل درخت‌های جستجوی نمایی هستند، حدود بهبود یافته‌ای روی برخی و یا تمام اعمال لغت نامه‌ای ارائه کرده‌اند.
- *ساختمان داده‌های گراف پویا*[3] پرس و جوهای گوناگون را پشتیبانی می‌کنند در حالیکه این امکان را فراهم می‌کنند که ساختار یک گراف بوسیله اعمالی که رأس‌ها یا یالها را درج و یا حذف می‌کنند تغییر کند. نمونه‌هایی از این پرس و جوها که پشتیبانی می‌شوند عبارتند از همبندی رأسی، همبندی یالی، درختهای پوشای مینیمم، همبندی دو طرفه، و بستار تعدی.

---

1. persisstent
2. Fusion trees
3. dynamic graph data structures

۱۸ B-Treeها

B-treeها درخت‌های جستجوی متوازنی هستند که برای کار روی دیسکهای مغناطیس و یا سایر وسایل ذخیره سازی ثانویه با دسترسی مستقیم طراحی شده‌اند. B-treeها شبیه درختهای قرمز-سیاه می‌باشند (فصل ۱۳)، اما در مینیمم کردن اعمال I/O دیسک بهتر عمل می‌کنند. بسیاری از سیستم‌های پایگاه داده برای ذخیره سازی اطلاعات از B-treeها یا انواع مختلف B-treeها استفاده می‌کنند.

B-treeها با درخت‌های قرمز - سیاه از این نظر تفاوت دارند که گره‌های B-tree ممکن است فرزندان بسیاری داشته باشد، از تعدادی کم تا هزاران فرزند. به عبارت دیگر، "ضریب انشعاب" یک B-tree می‌تواند بسیار بزرگ باشد، اگر چه این ضریب معمولاً با توجه به ویژگی‌های واحد دیسک استفاده شده مشخص می‌گردد. B-treeها از این نظر شبیه درخت‌های قرمز - سیاه هستند که هر B-tree با n گره، ارتفاعی برابر $O(lgn)$ دارد، اگر چه ارتفاع یک B-tree می‌تواند به طور قابل ملاحظه‌ای از ارتفاع یک درخت قرمز - سیاه کمتر باشد، زیرا ضریب انشعاب آن می‌تواند بسیار بزرگتر باشد. بنابراین B-treeها نیز می‌توانند برای پیاده سازی بسیاری از اعمال مجموعه‌های پویا در زمان $O(lgn)$ استفاده شوند.

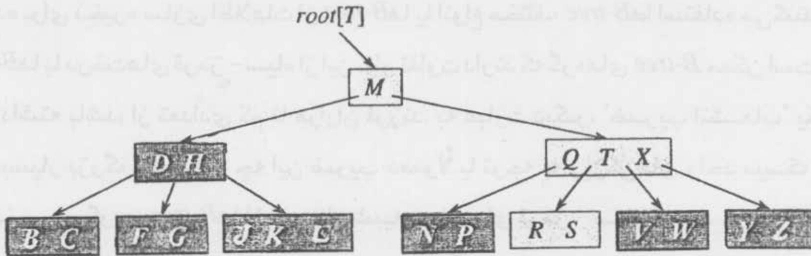
B-treeها درخت‌های جستجوی دودویی را با روش طبیعی تعمیم می‌دهند. شکل ۱۸.۱ یک B-tree ساده را نشان می‌دهد. اگر گره داخل x شامل $n[x]$ کلید باشد، آنگاه x دارای $n[x]+1$ فرزند است. کلیدهای داخل گره x بعنوان نقاط تفکیک کننده استفاده می‌شوند. این کلیدها دامنهٔ کلیدهایی که بوسیلهٔ x اداره می‌شوند را به $n[x]+1$ زیر دامنه تفکیک می‌کنند، بطوریکه هر یک بوسیلهٔ یک فرزند x اداره می‌گردد. در هنگام جستجوی یک کلید در B-tree، یک تعمیم $(n[x]+1)$- انتخابه بر مبنای مقایسات با $n[x]$ کلید ذخیره شده در گره x می‌گیریم. ساختار گره‌های داخلی تفاوت دارد؛ این تفاوت‌ها را در بخش ۱۸.۱ بررسی خواهیم کرد.

بخش ۱۸.۱ تعریفی دقیق از B-treeها ارائه می‌کند و ثابت می‌کند که ارتفاع یک B-tree تنها بصورت لگاریتمی با تعداد گره‌هایی که دارد، رشد می‌کند. بخش ۱۸.۲ چگونگی جستجوی یک کلید و درج یک کلید در B-tree را توضیح می‌دهد، و بخش ۱۸.۳ در مورد حذف بحث می‌کند. اگر چه قبل از هر چیز لازم

است بدانیم که چرا ساختمان داده‌هایی که برای کار روی دیسک مغناطیسی طراحی شده‌اند، نسبت به ساختمان داده‌هایی که برای کار در حافظه با دسترسی تصادفی اصلی طراحی شده‌اند به طور متفاوتی ارزیابی می‌شوند.

ساختمان داده‌ها روی حافظه ثانویه

فن آوریهای متفاوت بسیاری برای تأمین ظرفیت حافظه در یک سیستم کامپیوتری وجود دارد. ***حافظهٔ اولیه***[1] (یا ***حافظهٔ اصلی***[2]) یک سیستم کامپیوتری معمولاً از تراشه‌های سیلیکن حافظه تشکیل شده است. این فن آوری نوعاً به ازای هر بیت ذخیره شده، دو برابر گرانتر از فن آوری حافظهٔ مغناطیسی از قبیل نوارها و دیسکها است. بیشتر سیستم‌های کامپیوتری دارای ***حافظهٔ ثانویه***[3] بر مبنای دیسکهای مغناطیسی نیز هستند؛ مقدار چنین حافظهٔ ثانویه‌ای اغلب از مقدار حافظهٔ اولیه تجاوز می‌کند، حداقل دو برابر بزرگتر.



شکل ۱۸.۱ یک *B-tree* که کلیدهایش حروف بی‌صدای انگلیسی هستند. گره داخل x دارای $n[x]$ کلید و $n[x] + 1$ فرزند است. تمام برگها در درخت، عمق یکسانی دارند. گره‌های سایه روشن خورده در جستجوی حرف R بررسی می‌شوند.

شکل ۱۸.۲ (a) یک دیسک گردان معمول را نشان می‌دهد. دیسک گردان از ***صفحات***[4] متعددی تشکیل شده است که با سرعتی ثابت حول یک ***محور***[5] مشترک می‌چرخند. سطح هر صفحه با ماده ای مغناطیسی پوشیده شده است. هر صفحه بوسیلهٔ یک ***هِد***[6] که در انتهای یک ***بازو***[7] قرار دارد، خوانده شده و یا روی آن نوشته می‌شود. بازوها بصورت فیزیکی به هم متصل شده‌اند، یا به عبارتی با هم "یکی" شده‌اند، و می‌توانند هدهای خود را به سمت محور حرکت دهند یا از محور دور کنند. هنگامی که

---

1. primary
2. main memory
3. secondary storage
4. platter
5. spindle
6. head
7. arm

یک هد ثابت و بی‌حرکت است، سطحی که از زیر آن می‌گذرد *شیار*[1] نامیده می‌شود. هدهای خواندن نوشتن در تمامی اوقات بصورت عمودی تراز می‌شوند و بنابراین مجموعه شیارهای زیر آنها به طور همزمان مورد دستیابی قرار می‌گیرند. شکل (b) ۱۸.۲ چنین مجموعه‌ای از شیارها که *سیلندر*[2] نام دارد را نشان می‌دهد.

اگر چه دیسکها ارزانتر بوده و ظرفیت بیشتری نسبت به حافظهٔ اصلی دارند، اما چون قسمتهای متحرک دارند، بسیار آهسته‌تر می‌باشند. دوجزء حرکت مکانیکی عبارتند از: گردش صفحه و حرکت بازو. به عنوان مثال دیسکهایی که امروزه استفاده می‌شوند با سرعت 15,000 - 5400 دور در دقیقه (RPM) می‌چرخند که 7200 RPM رایج‌ترین است. اگر چه 7200 RPM ممکن است سریع به نظر برسد، اما هر چرخش 8.33 میلی ثانیه طول می‌کشد که تقریباً 5 برابر طولانی‌تر از زمانهای دستیابی 100 نانو ثانیه‌ای است که معمولاً حافظهٔ سیلیکنی دارد. به عبارت دیگر اگر مجبور باشیم برای قرار گرفتن یک عنصر خاص در زیر هد خواندن/ نوشتن منتظر یک چرخش کامل شویم، می‌توانیم در این مدت تقریباً 100,000 بار به حافظهٔ اصلی دستیابی داشته باشیم. به طور میانگین مجبور هستیم تنها نصف دور منتظر بمانیم، اما هنوز اختلاف در زمانهای دستیابی به حافظه سیلیکنی در مقایسه با دیسکها هنگفت است. حرکت بازوها نیز مقداری زمان صرف می‌کند. بعنوان مثال، زمانهای دستیابی میانگین به دیسکهایی که امروزه استفاده می‌شوند در بازه ۳ تا ۹ میلی ثانیه است.

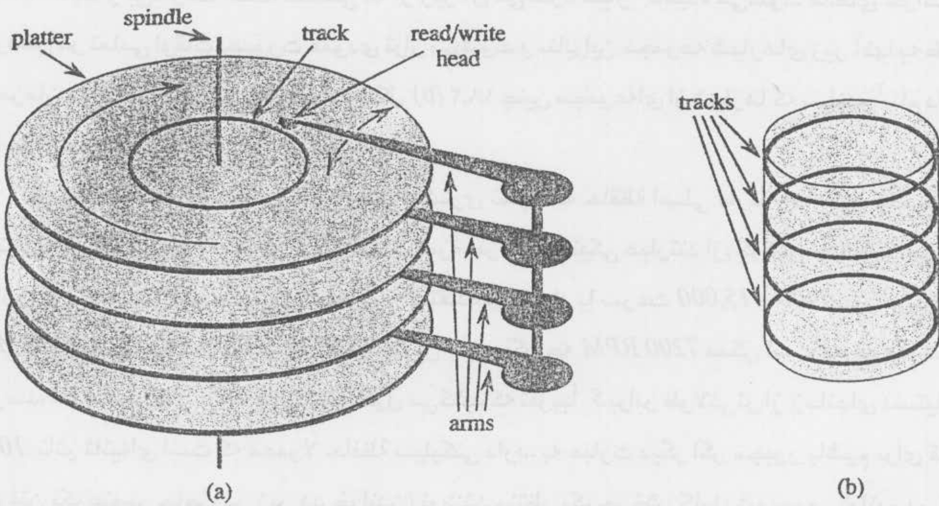
به منظور سرشکن کردن زمان انتظار صرف شده برای حرکات میکانیکی، دیسکها در یک زمان به یک عنصر دستیابی نمی‌کنند، بلکه به چندین عنصر دستیابی می‌کنند. اطلاعات به تعدادی *صفحه*[3] با اندازه برابر شامل بیت‌هایی که به طور پشت سر هم در سیلندرها قرار دارند، تقسیم می‌شوند. هر خواندن از دیسک و نوشتن بر آن از یک یا چند صفحه انجام می‌شود. برای یک دیسک معمول، یک صفحه ممکن است طولی برابر 2^{11} تا 2^{14} بایت داشته باشد. هنگامی که هد خواندن/نوشتن به طور صحیح قرار داده می‌شود و دیسک تا ابتدای صفحه مورد نظر می‌چرخد، خواندن از دیسک مغناطیسی یا نوشتن بر آن تماما الکترونیکی است. (به استثنای چرخش دیسک)، و حجم زیادی از داده‌ها می‌توانند به سرعت نوشته یا خوانده شوند.

اغلب دسترسی به یک صفحه از اطلاعات و خواندن آن از دیسک از بررسی تمام اطلاعات خوانده شده توسط کامپیوتر، زمان بیشتری را صرف می‌کند. به همین دلیل در این فصل به دو بخش اصلی زمان اجرا به طور جداگانه خواهیم پرداخت:

- تعداد دستیابی‌ها به دیسک، و
- زمان (محاسبه) CPU.

---

1. track
2. cylinder
3. page



شکل ۱۸.۲ *(a)* یک دیسک گردان معمول. این دیسک گردان از صفحات متعددی تشکیل شده است که دور یک محور می‌چرخند. هر صفحه با یک هد در انتهای یک بازو، خوانده و نوشته می‌شود. بازوها به هم متصل شده‌اند تا هدهای خود را به طور هماهنگ حرکت دهند. در اینجا بازوها دور یک محور مرکزی مشترک می‌چرخند. شیار، سطحی است که از زیر هد خواندن / نوشتن، وقتی که ثابت است، عبور می‌کند. *(b)* سیلندر از مجموعه‌ای از شیارهای متقابلاً عمود تشکیل شده است.

تعداد دستیابی‌ها به دیسک بر حسب تعداد صفحات اطلاعات که نیاز است از دیسک خوانده شده و یا روی آن نوشته شوند، اندازه گیری می‌شود. توجه داریم که زمان دستیابی به دیسک ثابت نیست -این زمان به فاصلهٔ بین شیار فعلی و شیار مورد نظر و همچنین حالت چرخشی اولیه، دیسک بستگی دارد. با این وجود تعداد صفحاتی که خوانده شده یا نوشته می‌شوند را به عنوان نخستین تقریب زمان کل صرف شده دستیابی استفاده خواهیم کرد.

در یک کاربر معمول *B-tree* حجم داده‌هایی که مدیریت می‌شوند بسیار زیاد است بطوریکه تمام داده‌ها بصورت یکجا در حافظهٔ اصلی گنجانده نمی‌شوند. الگوریتم‌های *B-tree* صفحات انتخاب شده را در صورت نیاز از دیسک به حافظهٔ اصلی کپی می‌کنند و صفحاتی را که تغییر کرده‌اند روی دیسک بازنویسی می‌کنند. الگوریتم‌های *B-tree* طوری طراحی شده‌اند که در هر زمان تنها تعداد ثابتی از صفحات در حافظهٔ اصلی قرار دارند. بنابراین اندازه حافظهٔ اصلی، اندازه *B-tree* که می‌تواند مدیریت گردد را محدود نمی‌کند.

اعمال دیسک را در شبه کد خود بصورت زیر مدل می‌کنیم. x را اشاره گری به یک شیء در نظر بگیرید. اگر شیء در حال حاضر در حافظه اصلی کامپیوتر باشد، آنگاه می‌توانیم به فیلدهای شیء همانند معمول مراجعه کنیم: برای مثال، *key[x]* اما اگر شیئی که x به آن اشاره می‌کند روی دیسک قرار داشته باشد، آنگاه باید عمل *DISK-READ(x)* را برای خواندن شیء x به داخل حافظهٔ اصلی، قبل از

آنکه بتوانیم به فیلدهای آن اشاره کنیم، اجرا نماییم. (فرض می‌کنیم که اگر x از قبل در حافظه اصلی باشد، آنگاه $DISK\text{-}READ(x)$ هیچ دستیابی به دیسک نیاز ندارد، این عمل یک "no-op" است.) به طور مشابه به عمل $DISK\text{-}WRITE(x)$ برای ذخیره هر تغییری که در فیلدهای شیء x انجام می‌شود استفاده می‌گردد. به عبارت دیگر، الگوی معمول کار با یک شیء بصورت زیر است:

$x \leftarrow$ یک اشاره گر به شیء

DISK-READ (x)

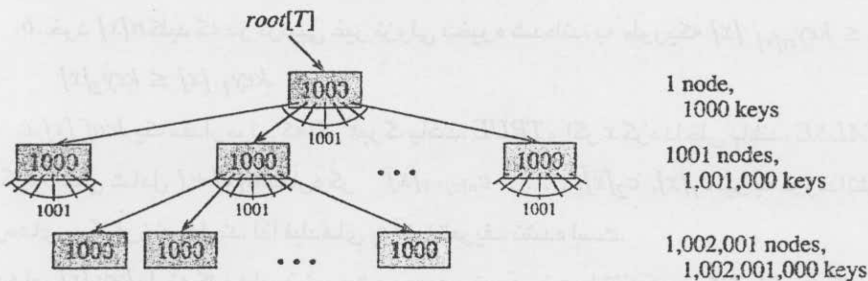
اعمالی که به فیلدهای x دستیابی پیدا کرده و / یا آنها را تغییر می‌دهند

DISK - WRITE (x) ▷ اگر هیچ فیلدی از x تغییر نکرده باشد، حذف می‌گردد

سایر اعمالی که به فیلدهای x دسترسی پیدا کرده، اما آنها را تغییر نمی‌دهند.

سیستم می‌تواند در یک زمان تنها تعداد محدودی از صفحات را در حافظهٔ اصلی نگهداری کند. فرض خواهیم کرد صفحاتی که دیگر استفاده نمی‌شوند، توسط سیستم از حافظهٔ اصلی پاک می‌شوند؛ الگوریتم‌های *B-tree* ما این مطلب را نادیده خواهند گرفت.

از آنجا که در بیشتر سیستم‌ها، زمان اجرای یک الگوریتم *B-tree* به طور عمده با تعداد اعمال *DISK-READ* و *DISK-WRITE* که انجام می‌دهد مشخص می‌گردد، منطقی است از این اعمال به طور مؤثر استفاده کنیم، بدین صورت که بیشترین اطلاعات ممکن را توسط آنها خوانده و یا بنویسیم. بنابراین یک گره *B-tree* معمولاً به بزرگی کل صفحهٔ دیسک است. لذا تعداد فرزندانی یک گره *B-tree* می‌تواند داشته باشد، با اندازه صفحهٔ دیسک محدود می‌شود.



شکل ۱۸.۳ یک *B-tree* با ارتفاع 2 که شامل بیش از یک میلیارد کلید است. هر گره داخلی و برگ شامل 1000 کلید است. 1001 گره در عمق 1 و بیش از یک میلیون برگ در عمق 2 قرار دارند. مقدار داخل هر گره x برابر $n[x]$ یعنی تعداد کلیدهای داخل x است.

برای یک *B-tree* بزرگ ذخیره شده روی دیسک، ضرایب انشعاب بسته به اندازه کلید مربوط به اندازه صفحه، اغلب بین 50 و 2000 استفاده می‌شوند. یک ضریب انشعاب بزرگ، ارتفاع درخت و تعداد دستیابی‌ها به دیسک را که برای پیدا کردن یک کلید مورد نیاز است، به طور شگرفی کاهش می‌دهد.

شکل ۱۸.۳ یک *B-tree* با ضریب انشعاب *1001* و ارتفاع *2* را نشان می‌دهد که می‌تواند بیشتر از یک میلیارد کلید را ذخیره کند. با این وجود از آنجا که گره ریشه می‌تواند به طور دائمی در حافظه اصلی نگهداری شود، لذا تنها حداکثر دو دستیابی به دیسک برای پیدا کردن یک کلید در این درخت نیاز است!

۱۸.۱ تعریف *B-tree*

به منظور سهولت همانطور که برای درخت‌های جستجوی دودویی و درخت‌های قرمز-سیاه داریم، فرض می‌کنیم "اطلاعات وابسته" مربوط به یک کلید در همان گره به عنوان کلید ذخیره می‌شوند. در عمل، یک *B-tree* ممکن است واقعاً با هر کلید، تنها یک اشاره گر به صفحهٔ دیگر دیسک که شامل اطلاعات وابسته برای آن کلید است ذخیره کند. شبه کد در این فصل به طور ضمنی فرض می‌کند که اطلاعات وابسته مربوط به یک کلید یا اشاره گر به چنین اطلاعاتی، هنگام انتقال کلید از گره به گرهی دیگر همراه با آن انتقال می‌یابد. یک نوع متفاوت و رایج از *B-tree* که B^+*-tree* نامیده می‌شود، تمام اطلاعات وابسته را در برگها ذخیره می‌کند و تنها کلیدها و اشاره‌گرهای فرزندان را در گره‌های داخلی ذخیره می‌کند، بنابراین ضریب انشعاب گره‌های داخلی را ماکزیمم می‌کند.

یک *B-tree* بنام T، یک درخت مشتق شده است (که ریشه‌اش *root[T]*) که دارای ویژگی‌های زیر می‌باشد:

۱. هر گره x دارای فیلدهای زیر است:

a. $n[x]$ تعداد کلیدهای فعلی ذخیره شده در گره x

b. خود $n[x]$ کلید که در ترتیبی غیر نزولی ذخیره شده‌اند، به طوریکه $key_1[x] \le key_2[x] \le \ldots \le key_{n[x]}[x]$

c. $leaf[x]$ یک مقدار بولی که اگر x برگ باشد، *TRUE* و اگر x گره داخلی باشد، *FALSE* است.

۲. هر گره داخلی شامل $n[x]+1$ اشاره گر $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ نیز به فرزندانش است. گره‌های برگ فرزند ندارند، لذا فیلدهای c_i آنها تعریف نشده است.

۳. کلیدهای $key_i[x]$ دامنه کلیدهای ذخیره شده در هر زیر درخت را تفکیک می‌کنند: اگر k_i کلید ذخیره شده در زیر درخت با ریشه $c_i[x]$ باشد، آنگاه

$$k_1 \le key_1[x] \le k_2 \le key_2[x] \le \ldots \le key_{n[x]}[x] \le k_{n[x]+1}$$

۴. تمام برگها عمق یکسانی دارند که برابر ارتفاع درخت h است.

۵. حدود پایین و بالا روی تعداد کلیدهایی که یک گره می‌تواند داشته باشد، وجود دارند. این حدود می‌توانند بصورت یک عدد صحیح ثابت $t \ge 2$ که مینیمم درجه[1] *B-tree* نامیده می‌شود بیان گردند.

---

1. *minimum degree*

a. هر گره به غیر از ریشه باید حداقل $t-1$ کلید داشته باشد. بنابراین هر گره داخلی به غیر از ریشه حداقل t فرزند دارد. اگر درخت غیر تهی باشد، ریشه حداقل باید یک کلید داشته باشد.

b. هر گره حداکثر می‌تواند $2t-1$ کلید داشته باشد. بنابراین یک گره داخلی حداکثر می‌تواند $2t$ فرزند داشته باشد. می‌گوییم یک گره پُر [1] است، اگر دقیقاً شامل $2t-1$ کلید باشد.[2]

ساده‌ترین B-tree هنگامیکه $t=2$ است، بوجود می‌آید. هر گره داخلی، 2، 3 یا 4 فرزند دارد و یک درخت 4 - 3 - 2 داریم. اما در عمل معمولاً مقادیر بسیار بزرگتر t استفاده می‌شوند.

ارتفاع یک B-tree

تعداد دستیابی‌های مورد نیاز به دیسک برای اکثر اعمال روی B-tree با ارتفاع B-tree متناسب است. اکنون ارتفاع یک B-tree را در بدترین حالت تحلیل می‌کنیم.

قضیه ۱۸.۱

اگر $n \geq 1$ آنگاه برای B-tree n کلیدی T با ارتفاع h و مینیمم درجهٔ $t \geq 2$ داریم

$$h \leq \log_t \frac{n+1}{2}$$

اثبات اگر یک B-tree دارای ارتفاع h باشد، ریشه حداقل یک کلید دارد و تمام گره‌های دیگر حداقل $t-1$ کلید دارند. بنابراین حداقل 2 گره در عمق 1، $2t$ گره در عمق 2، $2t^2$ گره در عمق 3 و... به همین ترتیب، تا اینکه در عمق h حداقل $2t^{h-1}$ گره وجود دارند. شکل ۱۸.۴ چنین درختی را برای $h=3$ نشان می‌دهد. بنابراین عدد n که برابر تعداد کلیدها است، در نامساوی زیر صدق می‌کند

$$\begin{aligned} n &\geq 1+(t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1+2(t-1)\left(\frac{t^h-1}{t-1}\right) \\ &= 2t^h-1 . \end{aligned}$$

با یک محاسبهٔ جبری ساده داریم $t^h \leq (n+1)/2$ با گرفتن لگاریتم در مبنای t از دو طرف، قضیه ثابت می‌شود. ■

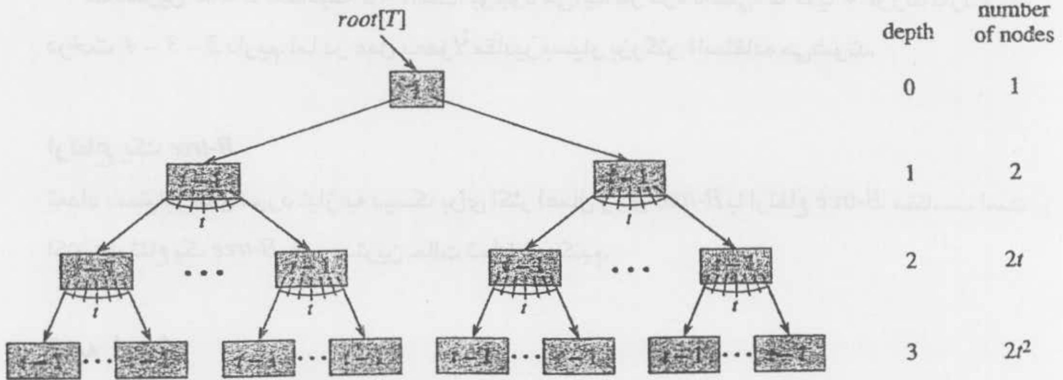
در اینجا قدرت B-treeها را در مقایسه با درخت‌های قرمز - سیاه می‌بینیم. اگر چه ارتفاع درخت بصورت $O(\lg n)$ در هر دو مورد رشد می‌کند (بخاطر آورید که t یک ثابت است)، برای B-treeها، مبنای

---

1. full

۲. نوع رایج دیگری از B-tree، B*-tree نام دارد. در B*-tree لازم است هر گره داخلی حداقل ۲/۳ پر باشد، بجای آنکه همانند یک B-tree حداقل نیمه پر باشد.

لگاریتم می‌تواند چندین مرتبه بزرگتر باشد. بنابراین تعداد گره‌هایی که برای اکثر اعمال درخت در *B-tree*ها بررسی می‌شوند نسبت به درختهای قرمز-سیاه با ضریبی در حدود *lgt* کاهش می‌یابند. از آنجا که بررسی یک گره دلخواه در یک درخت معمولاً به یک دستیابی به دیسک نیاز دارد، لذا تعداد دستیابی‌ها به دیسک به کلی کاهش می‌یابد.



شکل ۱۸.۴ یک *B-tree* با ارتفاع *3* شامل مینیمم تعداد کلید ممکن. مقدار داخل هر گره *x* برابر *n[x]* است.

تمرین‌ها

۱۸.۱ - ۱ چرا مینیمم درجه $t = 1$ را نمی‌پذیریم؟

۱۸.۱ - ۲ درخت شکل ۱۸.۱ به ازای چه مقادیری از *t* یک *B-tree* مجاز است؟

۱۸.۱ - ۳ تمام *B-tree*های مجاز بامینیمم درجهٔ *2* که {*1, 2, 3, 4, 5*} را نمایش می‌دهند، نشان دهید.

۱۸.۱ - ۴ ماکزیمم تعداد کلیدهایی که می‌توانند در یک *B-tree* با ارتفاع *h* ذخیره شوند بصورت تابعی از مینیمم درجهٔ *t* چیست؟

۱۸.۱ - ۵ ساختمان داده‌ای را بیان کنید که هر گره سیاه در یک درخت قرمز-سیاه، فرزندان قرمز خود را جذب کرده و فرزندان این گره‌های قرمز را به خود ملحق می‌کند.

۱۸.۲ اعمال اصلی روی *B-tree*ها

در این بخش جزئیات اعمال *B-TREE-SEARCH* ،*B-TREE-CREATE* و *B- TREE-INSERT* را بیان می‌کنیم. در این روالها دو قرار داد زیر را می‌پذیریم:

- ریشهٔ *B-tree* همواره در حافظهٔ اصلی است، لذا *DISK-READ* هرگز روی ریشه نیاز نمی‌باشد؛ اما هرگاه ریشه تغییر کند، یک *DISK-WRITE* از ریشه مورد نیاز است.
- گره‌هایی که به عنوان پارامتر فرستاده می‌شوند باید از قبل، عمل *DISK-READ* روی آنها انجام

شده باشد.

همه روالهایی که معرفی می‌کنیم الگوریتم‌هایی "یک گذره" هستند که از ریشه درخت به سمت پایین پیش می‌روند و مجبور به بازگشت به سمت بالا نمی‌باشند.

جستجوی یک B-tree

جستجوی یک *B-tree* بسیار شبیه جستجوی یک درخت جستجوی دودویی است، با این تفاوت که بجای اخذ یک تعمیم انشعابی دودویی یا "دو-انتخابه"، یک تعمیم انشعابی چند انتخابه بر طبق تعداد فرزندان گره می‌گیریم. به طور دقیق‌تر در هر گره داخلی x یک تعمیم انشعابی $(n[x]+1)$- انتخابه می‌گیریم.

B-TREE-SEARCH تعمیمی ساده و و مستقیم از روال *TREE-SEARCH* که برای درخت‌های جستجوی دودویی تعریف شده است می‌باشد. *B-TREE-SEARCH* بعنوان یک اشاره گر به گره ریشهٔ x از یک زیر درخت و کلید k که باید در آن زیر درخت جستجو شود را می‌گیرد. بنابراین فراخوانی اولیه به شکل *B-TREE-SEARCH(root[T],k)* است. اگر k در *B-tree* باشد، *B-TREE-SEARCH*- زوج مرتب (y,i) که از گره y و اندیس i تشکیل شده است بطوریکه $key_i[y]=k$، را برمی‌گرداند. در غیر اینصورت مقدار *NIL* برگردانده می‌شود.

```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3       do i ← i + 1
4  if i ≤ n[x] and k = key_i[x]
5     then return (x, i)
6  if leaf[x]
7     then return NIL
8     else  DISK-READ(c_i[x])
9           return B-TREE-SEARCH(c_i[x], k)
```

با استفاده از روال جستجوی خطی، خطوط ۳ - ۱ کوچکترین اندیس i را پیدا می‌کنند بطوریکه $k \le key_i[x]$ یا در غیر اینصورت i را با $n[x]+1$ مقدار دهی می‌کنند. خطوط ۵ - ۴ اگر کلید را پیدا کردیم، (x,i) را بر می‌گردانند. خطوط ۹ - ۶ یا جستجو را بصورت ناموفق خاتمه می‌دهند (اگر x یک برگ باشد) یا پس از انجام *DISK-READ* لازم روی فرزند x، برای جستجوی زیر درخت مناسب x فراخوانی بازگشتی را انجام می‌دهند.

شکل ۱۸.۱ عمل *B-TREE-SEARCH* را توضیح می‌دهد؛ گره‌های سایه روشن خورده در طی جستجو برای کلید R بررسی می‌شوند.

همانند روال *TREE-SEARCH* برای درخت‌های جستجوی دودویی، گره‌هایی که در طی بازگشت ملاقات شده‌اند، یک مسیر را از ریشهٔ درخت به سمت پایین تشکیل می‌دهند. لذا تعداد صفحات دیسک که توسط *B-TREE-SEARCH* مورد دستیابی قرار گرفته‌اند برابر $\Theta(h) = \Theta(\log_t n)$ است که در آن h ارتفاع *B-tree* و n تعداد کلیدها در *B-tree* است. از آنجا که $n[x] < 2t$ زمان صرف شده توسط حلقه *while* خطوط ۳ - ۲ در هر گره برابر $O(t)$ است، و زمان کل *CPU* برابر $O(th) = O(t\log_t n)$ می‌باشد.

ایجاد یک *B-tree* خالی

برای ساختن یک *B-tree* ابتدا از *B-TREE-CREATE* جهت ایجاد یک گره ریشهٔ خالی استفاده کرده و سپس *B-TREE-INSERT* را به منظور اضافه کردن کلیدهای جدید فراخوانی می‌کنیم. هر دوی این روالها از روال کمکی *ALLOCATE-NODE* استفاده می‌کنند. این روال یک صفحه دیسک را برای استفاده بعنوان یک گره جدید در زمان $O(1)$ اختصاص می‌دهد. می‌توانیم فرض کنیم گرهی که بوسیله *ALLOCATE-NODE* ایجاد می‌شود، به *DISK-READ* نیاز ندارد زیرا هنوز اطلاعات مفیدی که روی دیسک برای این گره ذخیره شده باشد وجود ندارد.

```
B-TREE-CREATE(T)
1  x ← ALLOCATE-NODE()
2  leaf[x] ← TRUE
3  n[x] ← 0
4  DISK-WRITE(x)
5  root[T] ← x
```

B-TREE-CREATE به $O(1)$ عمل دیسک و زمان *CPU* برابر $O(1)$ نیاز دارد.

درج[1] یک کلید در *B-tree*

درج یک کلید در *B-tree* به طور چشمگیری پیچیده‌تر از درج یک کلید در درخت جستجوی دودویی است. همانند درخت‌های جستجوی دودویی به دنبال مکان برگ می‌گردیم تا کلید را در آنجا درج کنیم. هر چند در یک *B-tree* به سادگی نمی‌توانیم یک گره برگ جدید ایجاد و آنرا درج کنیم، زیرا درخت حاصل، یک *B-tree* معتبر نخواهد بود. در عوض، کلید جدید را در یک گره برگ موجود درج می‌کنیم. از آنجا که نمی‌توانیم یک کلید را در گره برگی که پر است درج کنیم، عملی را معرفی می‌کنیم که گره پر y را (که دارای $2t-1$ کلید است) حول کلید میانی[2] $key_t[y]$ آن به دو گره که هر کدام $t-1$ کلید دارند

---

1. insert
2. median dey

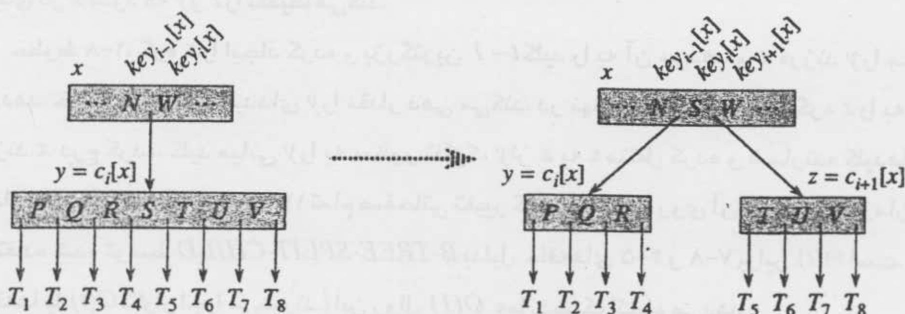
می‌شکنند[1]. کلید میانی به پدر yمنتقل می‌شود تا نقطه تفکیک بین دو درخت جدید را مشخص کند. اما اگر پدر yنیز پر بود باید قبل از آنکه کلید جدید بتواند درج شود، بشکند و بنابراین نیاز به شکستن گره‌های پر می‌تواند در تمام مسیر به سمت بالا در درخت انتشار یابد.

همانند یک درخت جستجوی دودویی می‌توانیم یک کلید را در گذری واحد از ریشه به پایین تا یک برگ در *B-tree* درج کنیم. برای انجام چنین کاری منتظر فهمیدن اینکه آیا واقعاً جهت انجام درج به شکستن یک گره پر نیاز خواهیم داشت یا خیر نمی‌شویم. در عوض همانطور که به سمت پایین درخت جهت جستجو برای مکانی که کلید جدید متعلق به آن است حرکت می‌کنیم، هر گره پری را که در طول راه به آن می‌رسیم (بعلاوه خود برگ) می‌شکنیم. بنابراین هرگاه بخواهیم گره پر yرا بشکنیم، مطمئن هستیم که پدرش پر نیست.

شکستن یک گره در *B-tree*

روال *B-TREE-SPLIT-CHILD* به عنوان ورودی، گره داخلی غیر پر x (فرض می‌شود که در حافظهٔ اصلی باشد)، اندیس i و گره y (فرض می‌شود که در حافظهٔ اصلی باشد) را می‌گیرد، بطوریکه $y = c_i[x]$ یک فرزند پر xاست. سپس روال این فرزند را به دو قسمت می‌شکند و xرا طوری تنظیم می‌کند که دارای یک فرزند اضافی باشد. (برای شکستن ریشه پر، ابتدا ریشه را فرزند یک گره ریشهٔ خالی جدید قرار خواهیم داد تا بتوانیم از روال *B-TREE-SPLIT-CHILD* استفاده کنیم. بنابراین ارتفاع درخت یک واحد رشد می‌کند؛ شکستن، تنها عملی است که سبب رشد درخت می‌شود.)

شکل ۱۸.۵ این فرآیند را نشان می‌دهد. گره پر yحول کلید میانی خود یعنی S شکسته می‌شود، که این کلید به گره xیعنی پدر y منتقل می‌شود. کلیدهای داخل y که از کلید میانی بزرگترند به گره جدید z که فرزند جدید xاست منتقل می‌شوند.



شکل ۱۸.۵ شکستن یک گره با $t = 4$ گره yبه دو گره yوzشکسته می‌شود. و کلیدمیانی آن یعنی S به پدر y منتقل می‌شود.

---
1. split

```
B-TREE-SPLIT-CHILD(x, i, y)
 1  z ← ALLOCATE-NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t − 1
 4  for j ← 1 to t − 1
 5       do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7     then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11       do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14       do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_t[y]
16  n[x] ← n[x] + 1
17  DISK-WRITE(y)
18  DISK-WRITE(z)
19  DISK-WRITE(x)
```

B-TREE-SPLIT-CHILD با "برش و الصاق" مستقیم و ساده کار می‌کند. در اینجا y iامین فرزند x و گرهی است که شکسته می‌شود. گره y در ابتدا $2t$ فرزند دارد ($2t-1$ کلید)، اما بوسیله این عمل به t فرزند ($t-1$ کلید) کاهش می‌یابد. گره z t فرزند ($t-1$ کلید) بزرگتر y را "می‌پذیرد" و فرزند جدید x می‌شود. این گره درست پس از y در ردیف فرزندان x قرار می‌گیرد. کلید میانی y به بالا منتقل شده تا کلیدی در x شود که y و z را تفکیک می‌کند.

خطوط ۸–۱، گره z را ایجاد کرده و بزرگترین $t-1$ کلید را به آن می‌دهد و t فرزند y را مطابقت می‌دهد. خط ۹ شمارنده کلیدهای y را مقدار دهی می‌کند. در نهایت، خطوط ۱۶–۱۰ گره z را به عنوان فرزند x درج کرده، کلید میانی y را به منظور تفکیک y از z به x منتقل کرده و شمارنده کلیدهای x را مقدار دهی می‌کنند. خطوط ۱۹–۱۷ تمام صفحاتی تغییر کرده دیسک را روی آن می‌نویسند. زمان *CPU* استفاده شده توسط *B-TREE-SPLIT-CHILD* بدلیل حلقه‌های ۵–۴ و ۸–۷ برابر $\Theta(t)$ است. (سایر حلقه‌ها با $O(t)$ تکرار اجرا می‌شوند.) این روال $O(1)$ عمل دیسک انجام می‌دهد.

درج یک کلید در *B-tree* در یک گذر واحد به سمت پایین درخت

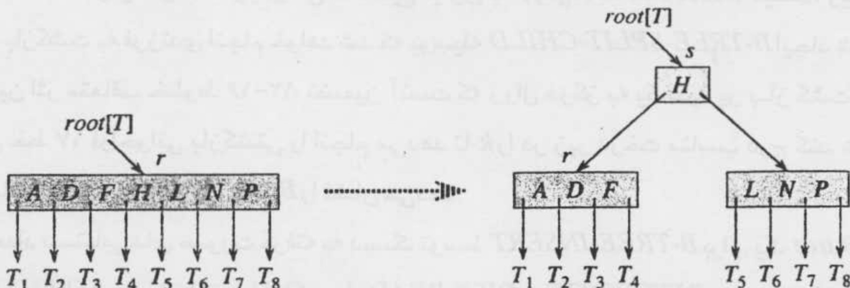
درج یک کلید در *B-tree* با ارتفاع h در یک گذر واحد به سمت پایین درخت نیاز به $O(h)$ دستیابی به

دیسک دارد. زمان *CPU* مورد نیاز برابر $O(th) = O(t \log_t n)$ است. روال *B-TREE-INSERT* از *B-TREE-SPLIT-CHILD* استفاده می‌کند تا تضمین کند که فراخوانی بازگشتی هرگز تا یک گره پر ادامه نمی‌یابد.

```
B-TREE-INSERT(T, k)
 1  r ← root[T]
 2  if n[r] = 2t − 1
 3     then s ← ALLOCATE-NODE()
 4          root[T] ← s
 5          leaf[s] ← FALSE
 6          n[s] ← 0
 7          c1[s] ← r
 8          B-TREE-SPLIT-CHILD(s, 1, r)
 9          B-TREE-INSERT-NONFULL(s, k)
10     else B-TREE-INSERT-NONFULL(r, k)
```

خطوط ۹-۳ حالتی را که در آن گره، ریشه یعنی r پر است، اداره می‌کنند: ریشه شکسته شده و گره جدید s (که دو فرزند دارد) ریشه می‌شود. شکستن ریشه تنها راه افزایش ارتفاع یک *B-tree* است. شکل ۱۸.۶ این حالت را نشان می‌دهد. بر خلاف درخت جستجوی دودویی، ارتفاع *B-tree* بجای آنکه به سمت پایین رشد کند، به سمت بالا رشد می‌کند. روال با فراخوانی *B-TREE-INSERT-NONFULL* برای انجام درج کلید k در درخت مشتق شده از گره ریشهٔ غیر پر، خاتمه می‌یابد. با *B-TREE-INSERT-NONFULL* در هنگام لزوم به پایین درخت بازگشت می‌کند، و در تمام زمانها با فراخوانی *B-TREE-SPLIT-CHILD* در هنگام لزوم، تضمین می‌کند که گرهی که به آن بازگشت می‌کند پر نیست.

روال بازگشتی کمکی *B-TREE-INSERT-NONFULL* کلید k را در گره x درج می‌کند. فرض می‌شود که در هنگام فراخوانی روال، x غیر پر است. عمل *B-TREE-INSERT* و عمل بازگشتی *B-TREE-INSERT-NONFULL* تضمین می‌کنند که این فرض درست است.



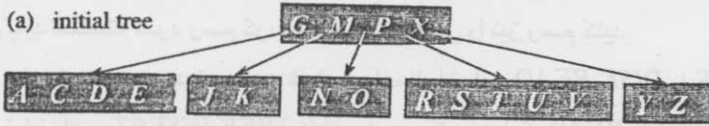
شکل ۱۸.۶ شکستن ریشه با $t = 4$ گره ریشه یعنی r به دو قسمت شکسته می‌شود و گره ریشه جدید یعنی s ایجاد می‌شود. ریشه جدید شامل کلید میانی r است و دو نیمه r را بعنوان فرزندانش دارا می‌باشد. وقتی ریشه شکسته می‌شود، ارتفاع *B-tree* یک واحد افزایش می‌یابد.

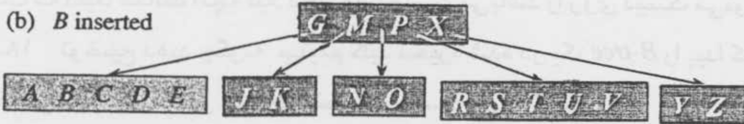
```
B-TREE-INSERT-NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3     then while i ≥ 1 and k < key_i[x]
 4              do key_{i+1}[x] ← key_i[x]
 5                 i ← i − 1
 6           key_{i+1}[x] ← k
 7           n[x] ← n[x] + 1
 8           DISK-WRITE(x)
 9     else  while i ≥ 1 and k < key_i[x]
10              do i ← i − 1
11           i ← i + 1
12           DISK-READ(c_i[x])
13           if n[c_i[x]] = 2t − 1
14              then B-TREE-SPLIT-CHILD(x, i, c_i[x])
15                   if k > key_i[x]
16                      then i ← i + 1
17           B-TREE-INSERT-NONFULL(c_i[x], k)
```

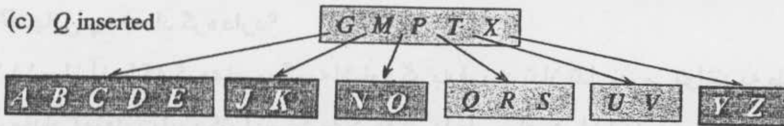
روال *B-TREE-INSERT-NONFULL* به صورت زیر کار می‌کند. خطوط ۸-۳ حالتی که در آن کلید k در x وقتی x برگ است، درج می‌شود را اداره می‌کنند. اگر x گره برگ نباشد، آنگاه باید k را در گره برگ مناسب در زیر درخت مشتق شده از گره داخلی x درج کنیم. در این حالت، خطوط ۱۱-۹ فرزند x که بازگشت به آن انجام می‌شود را مشخص می‌کنند. خط ۱۳ تعیین می‌کند که آیا بازگشت به یک فرزند پر انجام می‌شود یا خیر. اگر بازگشت به یک فرزند پر انجام شود، خط ۱۴ با استفاده از *B-TREE-SPLIT-CHILD* این فرزند را به دو فرزند غیر پر می‌شکند و خطوط ۱۶-۱۵ مشخص می‌کنند که اکنون کدامیک از دو فرزند، فرزند صحیح برای پایین رفتن به آن است. (توجه کنید پس از آنکه خط ۱۶، i را یک واحد افزایش می دهد هیچ نیازی به *DISK-READ($c_i[x]$)* نیست، زیرا در این حالت بازگشت به فرزندی انجام خواهد شد که بوسیله *B-TREE-SPLIT-CHILD* ایجاد شده است.) بنابراین اثر متعاقب خطوط ۱۶-۱۳، تضمین آنست که روال هرگز به یک گره پر باز گشت نمی‌کند. سپس خط ۱۷ فراخوانی بازگشتی را انجام می‌دهد تا k را در زیر درخت مناسب درج کند. شکل ۱۸.۷ حالتهای متفاوت درج در یک *B-tree* را نشان می‌دهد.

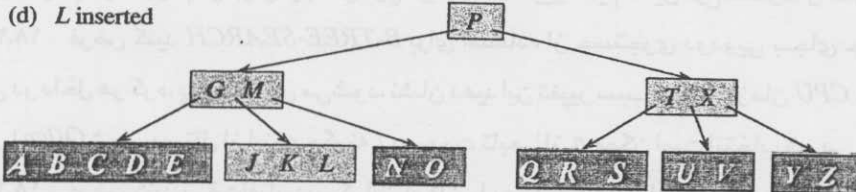
تعداد دستیابی‌های صورت گرفته به دیسک توسط *B-TREE-INSERT* برای یک *B-tree* با ارتفاع h برابر $O(h)$ است، زیرا تنها $O(1)$ عمل *DISK-READ* و *DISK-WRITE* مابین فراخوانی‌های *B-TREE-INSERT-NONFULL* انجام می‌شوند. زمان کل CPU که صرف شده است برابر $O(th)=O(t\log_t n)$ است. از آنجا که *B-TREE-INSERT-NONFULL* یک روال در انتها - بازگشتی

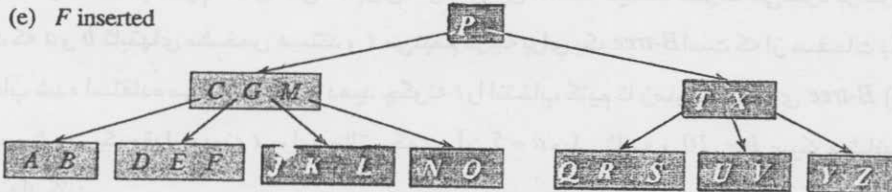
است، می‌تواند متناوباً بصورت یک حلقه *while* پیاده سازی شود، که ثابت می‌کند تعداد صفحاتی که لازم است در حافظه اصلی باشند در هر زمان برابر $O(1)$ است.











شکل ۱۸.۷ درج کلیدها در *B-tree*. مینیمم درجه t برای این *B-tree* برابر 3 است، لذا یک گره می‌تواند حداکثر 5 کلید داشته باشد. گره‌هایی که با فرآیند درج تغییر می‌کنند، سایه روشن خورده‌اند. *(a)* درخت اولیه برای این مثال. *(b)* نتیجه درج B در درخت اولیه؛ این درج، یک درج ساده در گره برگ است. *(c)* نتیجه درج Q در درخت قبلی. گره *RSTUV* به دو گره که شامل *RS* و *UV* هستند شکسته می‌شود، کلید T به ریشه منتقل شده و Q در سمت چپ‌ترین مکان دو نیمه (گره *RS*) درج می‌شود. *(d)* نتیجه درج L در درخت قبلی. چون ریشه پر است بلافاصله شکسته می‌شود و ارتفاع *B-tree* یک واحد افزایش می‌یابد. سپس L در برگی که شامل *JK* است درج می‌شود. *(e)* نتیجه درج F در درخت قبلی. گره *ABCDE* قبل از آنکه F در سمت راست‌ترین مکان دو نیمه (گره *DE*) درج شود، شکسته می‌شود.

تمرین‌ها

۱ - ۱۸.۲ نتایج درج کلیدهای

F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A B, X, Y, D, Z, E

را بترتیب در یک *B-tree* خالی با مینیمم درجه *2* نشان دهید. تنها، وضعیت درخت را درست قبل از آنکه گرهی باید شکسته شود رسم کرده و وضعیت نهایی را نیز رسم کنید.

۲ - ۱۸.۲ توضیح دهید تحت چه شرایطی اعمال اضافی *DISK-READ* یا *DISK-WRITE* در طی انجام فراخوانی *B-TREE-INSERT* انجام می‌شوند. (یک *DISK-READ* اضافی، یک صفحه از اطلاعات که دقیقاً همانند آنچه قبلاً ذخیره شده است می‌باشد را روی دیسک می‌نویسد.)

۳ - ۱۸.۲ توضیح دهید چگونه مینیمم کلید ذخیره شده در یک *B-tree* را پیدا کنیم و چگونه کلید ما قبل یک کلید مفروض که در *B-tree* ذخیره شده است را پیدا کنیم.

۴ - ۱۸.۲ فرض کنید کلیدهای $\{1, 2, \ldots, n\}$ در یک *B-tree* خالی بامینیمم درجه *2* درج می‌شوند. *B-tree* نهایی چه تعداد گره دارد؟

۵ - ۱۸.۲ از آنجا که گره‌های برگ به اشاره گر به فرزند نیاز ندارند، می‌توانند به طور قابل تصوری از یک مقدار t متفاوت (بزرگتر) نسبت به گره‌های داخلی برای همان اندازه صفحه دیسک استفاده کنند. نشان دهید چگونه روالها را برای ایجاد و درج در *B-tree* تغییر دهیم تا این نوع متفاوت را اداره کنند.

۶ - ۱۸.۲ فرض کنید *B-TREE-SEARCH* برای استفاده از جستجوی دودویی بجای جستجوی خطی در داخل هر گره، پیاده سازی می‌شود. نشان دهید این تغییر سبب می‌شود زمان *CPU* مورد نیاز برابر $O(\lg n)$ شود، مستقل از اینکه چگونه t بصورت تابعی از n ممکن است انتخاب شود.

۷ - ۱۸.۲ فرض کنید سخت‌افزار دیسک این امکان را به ما می‌دهد تا اندازه یک صفحه دیسک را به طور دلخواه انتخاب کنیم، اما زمانی که برای خواندن این صفحه دیسک صرف می‌شود برابر $a+bt$ است که a و b ثابتهای مشخص هستند و t می‌نیمم درجه برای یک *B-tree* است که از صفحات با اندازه انتخاب شده استفاده می‌کند. توضیح دهید چگونه t را انتخاب کنیم تا زمان جستجوی *B-tree* (تقریباً) مینیمم شود. یک مقدار بهینه t برای حالتی که در آن $a = 5$ میلی ثانیه و $b = 10$ میکرو ثانیه است پیشنهاد کنید.

۱۸.۳ حذف[1] یک کلید از *B-tree*

حذف از *B-tree* مشابه درج است اما کمی پیچیده‌تر می‌باشد، زیرا یک کلید ممکن است از هر گرهی - نه فقط یک برگ - حذف شود و حذف از یک گره داخلی نیاز به این دارد که فرزندان گره دوباره مرتب

1. delete

شوند. همانند درج باید مراقب حذفی باشیم که سبب تولید یک درخت می‌شود که ساختار آن از ویژگی‌های *B-tree* تخطی می‌کند. تنها همانطور که باید تضمین می‌کردیم یک گره بخاطر درج بسیار بزرگ نشده باشد، باید تضمین کنیم که یک گره در طی حذف بسیار کوچک نشود. (با این تفاوت که ریشه اجازه دارد کمتر از مینیمم تعداد $t-1$ کلید داشته باشد، هر چند اجازه ندارد که بیشتر از ماکزیمم تعداد $2t-1$ کلید داشته باشد). درست همانند الگوریتم درج ساده که اگر گرهی در مسیر که کلید باید در آن درج می‌شد پر بود، باید به عقب بر می‌گشت، یک روش ساده برای حذف نیز اگر یک گره (به غیر از ریشه) در طول مسیر که کلید باید از آن حذف شود مینیمم تعداد کلید را داشته باشد، باید به عقب برگردد.

فرض کنید از روال *B-TREE-DELETE* خواسته می‌شود که کلید k را از زیر درخت مشتق شده از x حذف کند. ساختار این روال برای تضمین آنکه هرگاه *B-TREE-DELETE* روی گره x بصورت بازگشتی فراخوانی می‌شود تعدد کلیدها در x حداقل برابر مینیمم درجه t باشد تغییر می‌کند.

توجه کنید که این شرط نیاز به یک کلید بیشتر از مینیمم تعداد کلیدی دارد که در شرایط معمول *B-tree* مورد نیاز است، لذا گاهی اوقات ممکن است یک کلید مجبور باشد قبل از آنکه بازگشت به یک گره فرزند انجام شود به آن گره منتقل شود. این شرط قوی‌تر امکان آنرا فراهم می‌کند که یک کلید را در یک گذر رو به پایین بدون نیاز به "برگشت" از درخت حذف کنیم (با یک استثنا که توضیح خواهیم داد).

مشخصات زیرا برای حذف از یک *B-tree* باید با این توافق تفسیر شوند که اگر این اتفاق افتاد که گره ریشه x یک گره داخلی شد که هیچ کلیدی نداشت (این وضعیت می‌تواند در حالتهای $2c$ و $3b$ در ذیل رخ دهد)، آنگاه x حذف شده و تنها فرزند x یعنی $c_1[x]$، ریشه جدید درخت می‌شود، ارتفاع درخت یک واحد کاهش می‌یابد و این ویژگی که ریشه درخت شامل حداقل یک کلید است حفظ می‌شود (مگر آنکه درخت خالی باشد).

بجای ارائه شبه کد، نحوه حذف را به طور خلاصه بیان می‌کنیم. شکل ۱۸.۸ حالتهای متفاوت حذف کلیدها از یک *B-tree* را نشان می‌دهد.

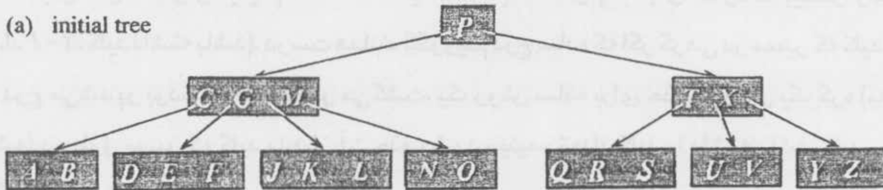
۱. اگر کلید k در گره x باشد و x برگ باشد، k را از x حذف کنید.

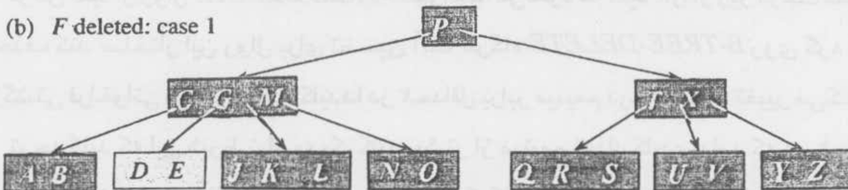
۲. اگر کلید k در گره x باشد و x یک گره داخلی باشد، بصورت زیر عمل کنید.

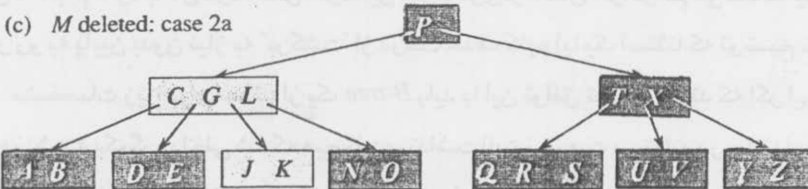
a. اگر y فرزند x که قبل از کلید k در گره x قرار دارد حداقل t کلید داشته باشد، آنگاه کلید ما قبل k یعنی k' را در زیر درخت مشتق شده از y پیدا کنید. به صورت بازگشتی k' را حذف کرده و k را با k' در x جایگزین کنید. (پیدا کردن k' و حذف آن می‌تواند در یک گذر واحد به سمت پایین انجام گیرد.)

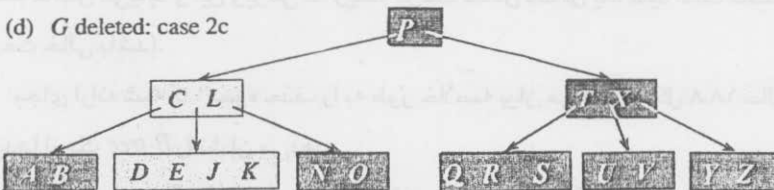
b. به طور متقارن اگر z فرزند x که در گره x پس از k قرار دارد حداقل t کلید داشته باشد، آنگاه کلید ما بعد k یعنی k' را در زیر درخت مشتق شده از z پیدا کنید. k' را بصورت بازگشتی

حذف کرده و k را با k' در x جایگزین کنید. (پیدا کردن k' و حذف آن می‌تواند در یک گذر واحد به سمت پایین انجام گیرد.)

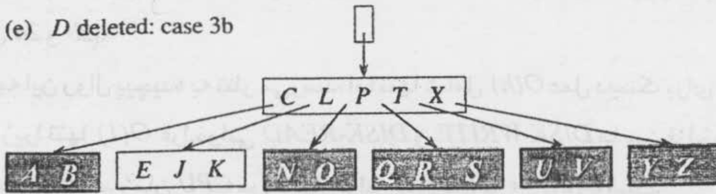


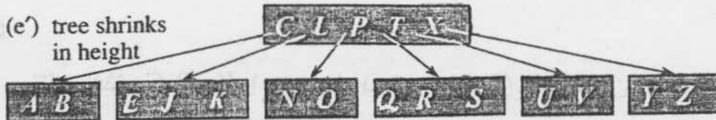


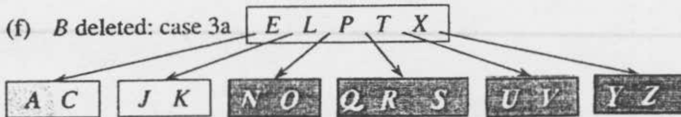




شکل ۱۸.۸ حذف کلیدها از یک *B-tree* مینیمم درجه برای این *B-tree* برابر $t = 3$ است، لذا یک گره (به غیر از ریشه) نمی‌تواند کمتر از 2 کلید داشته باشد. گره‌هایی که تغیر می‌کنند، سایه روشن خورده‌اند. *B-tree* (a) شکل ۱۸.۷ (e). (b) حذف F این حالت 1 است: حذف ساده از یک برگ. (c) حذف M این حالت 2a است: کلید L که قبل از M قرار دارد، برای گرفتن مکان M به بالا منتقل می‌شود. (d) حذف G. این حالت 2c است: G برای ساخت گره $DEGJK$ به پایین رانده می‌شود و سپس G از این برگ حذف می‌شود (حالت 1). (e) حذف D این حالت 3b است: بازگشت به گره CL نمی‌تواند انجام شود، زیرا تنها 2 کلید دارد بنابراین P به پایین رانده می‌شود و با CL و TX برای تشکیل $CLPTX$ ادغام می‌شود؛ سپس D از برگ حذف می‌شود (حالت 1). (e') پس از (d)، ریشه حذف می‌شود و ارتفاع درخت یک واحد کاهش می‌یابد. (f) حذف B این حالت 3a است: C برای پر کردن مکان B منتقل می‌شود و E برای پر کردن مکان C منتقل می‌شود.







c. در غیر اینصورت، اگر y و z هر دو انتها $t-1$ کلید داشته باشند، k و کل z را در y ادغام کنید، لذا x هم k و هم اشاره گر به z را از دست می‌دهد و y اکنون شامل $2t-1$ کلید است. سپس z را آزاد کرده و k را بصورت بازگشتی از y حذف کنید.

۳. اگر کلید k در گره داخلی x نباشد، ریشه $c_i[x]$ زیر درخت مناسبی که اگر k به هر حال داخل درخت باشد باید شامل آن باشد را مشخص کنید. اگر $c_i[x]$ تنها $t-1$ کلید داشته باشد، گام $3a$ و $3b$ را در صورت لزوم برای تضمین آنکه به گرهی که شامل حداقل t کلید است پایین می‌رویم، اجرا کنید. سپس با بازگشت روی فرزند مناسب x به کار خود پایان دهید.

a. اگر $c_i[x]$ تنها $t-1$ کلید داشته باشد اما یک همزاد بلافصل با حداقل t کلید داشته باشد، با انتقال یک کلید از x به پایین به $c_i[x]$ یک کلید اضافی به آن بدهید، یک کلید از همزاد بلافصل چپ یا راست $c_i[x]$ به بالا به x منتقل کرده، و اشاره گر فرزند مناسب را از این همزاد به داخل $c_i[x]$ انتقال دهید.

b. اگر $c_i[x]$ و هر دو همزاد بلافصل $c_i[x]$ دارای $t-1$ کلید باشند، $c_i[x]$ را با یک همزاد ادغام کنید، که مستلزم انتقال یک کلید از x به پایین به داخل گره جدید ادغام شده می‌باشد تا کلید میانی برای آن گره شود.

از آنجا که اکثر کلیدها در B-tree در برگها هستند، ممکن است در عمل انتظار داشته باشیم که اعمال حذف اکثراً برای حذف کلیدها از برگها استفاده می‌شوند. پس روال B-TREE-DELETE در یک گذر رو به پایین در طول درخت عمل می‌کند، بدون آنکه مجبور شود به بالا برگردد. هر چند در هنگام حذف یک کلید در داخل یک گره داخلی، روال یک گذر رو به پایین را در درخت انجام می‌دهد، اما ممکن است

مجبور باشد به گرهی باز گردد که کلید از آن حذف شده است تا با کلید ماقبل یا مابعدش جایگزین شود (حالتهای $2a$ و $2b$).

اگر چه این روال پیچیده به نظر می‌رسد، اما تنها شامل $O(h)$ عمل دیسک برای یک *B-tree* با ارتفاع h است، زیرا تنها $O(1)$ فراخوانی *DISK-READ* و *DISK-WRITE* ما بین فراخوانی‌های بازگشتی روال انجام می‌شود. زمان *CPU* مورد نیاز برابر $O(th) = O(t \log_t n)$ است.

تمرین‌ها

۱ - ۱۸.۳ نتایج حذف C، P و V را بترتیب از درخت شکل (f) ۱۸.۸ نشان دهید.

۲ - ۱۸.۳ شبه کدی برای *B-TREE-DELETE* بنویسید.

مسائل

۱۸-۱ پشته‌ها در حافظه ثانویه

پیاده سازی یک پشته در کامپیوتری را در نظر بگیرید که حافظهٔ اصلی نسبتاً کوچک و حافظه کندتر دیسک نسبتاً بزرگی دارد. اعمال *PUSH* و *POP* روی مقادیر یک کلمه‌ای پشتیبانی می‌شوند. پشته‌ای که می‌خواهیم پشتیبانی کنیم، می‌تواند آنقدر رشد کند تا بسیار بزرگتر از آن شود که در حافظه بگنجد و بنابراین بیشتر آن باید روی دیسک ذخیره شود.

در یک پیاده سازی ساده، ولی نا کار آمد پشته، کل پشته را روی دیسک نگهداری می‌کنیم. یک اشاره گر پشته در حافظه نگه می‌داریم که آدرس عنصر بالای پشته روی دیسک است. اگر مقدار اشاره گر P باشد، عنصر بالا $(p \bmod m)$ امین کلمه در صفحه $\lfloor P/m \rfloor$، دیسک می‌باشد که m تعداد کلمات در هر صفحه است.

برای پیاده سازی عمل *PUSH*، اشاره گر پشته را یک واحد اضافه می‌کنیم، صفحه مناسب را از دیسک به داخل حافظه خوانده، عنصری را که باید در کلمه مناسب در صفحه *push* شود را کپی کرده، و صفحه را روی دیسک بازنویسی می‌کنیم. عمل *POP* نیز به همین شکل است. اشاره‌گر پشته را یک واحد کاهش داده، صفحه مناسب را از دیسک خوانده، و عنصر بالای پشته را بر می‌گردانیم. نیازی به بازنویسی صفحه نداریم زیرا تغییر نکرده است.

چون اعمال دیسک نسبتاً پر هزینه هستند، دو هزینه برای هر پیاده سازی محاسبه می‌کنیم: تعداد کل دستیابی‌ها به دیسک و زمان کل *CPU*. دستیابی به یک صفحهٔ دیسک با m کلمه منجر به هزینه‌های یک دستیابی به دیسک و $\Theta(m)$ زمان *CPU* می‌شود.

a. تعداد دستیابی‌ها به دیسک برای n عمل پشته که این پیاده سازی ساده را استفاده می‌کنند، به طور

مجانبی در بدترین حالت چقدر است؟ زمان CPU برای n عمل پشته چیست؟ (جواب خود را بر حسب m و n برای این قسمت و قسمت‌های بعد بیان کنید.)

اکنون یک پیاده سازی از پشته را در نظر بگیرید که در آن یک صفحه از پشته را در حافظه نگهداری می‌کنیم. (همچنین حجم کوچکی از حافظه را برای اطلاع از آنکه کدام صفحه هم اکنون در حافظه است نگهداری می‌کنیم.) تنها اگر صفحه دیسک مربوطه در حافظه باشد می‌توانیم یک عمل پشته را انجام دهیم. در صورت لزوم صفحه فعلی در حافظه می‌تواند روی دیسک نوشته شده و صفحه جدید از دیسک به حافظه خوانده شود. اگر صفحه دیسک مربوطه از قبل در حافظه باشد آنگاه هیچ دستیابی به دیسک نیاز نخواهد بود.

b. تعداد دستیابی‌های لازم به دیسک برای n عمل *PUSH* در بدترین حالت چیست؟ زمان CPU چقدر است؟

c. تعداد دستیابی‌های لازم به دیسک برای n عمل پشته در بدترین حالت چیست؟ زمان CPU چقدر است؟

اکنون فرض کنید پشته را با نگهداری دو صفحه در حافظه پیاده سازی می‌کنیم (علاوه بر نگهداری تعداد کمی از کلمات برای ثبت).

d. توضیح دهید چگونه صفحه‌های پشته را مدیریت کنیم تا تعداد سرشکن شده دستیابی‌ها به دیسک برای هر عمل پشته برابر $O(1/m)$ و زمان سرشکن شده CPU برای هر عمل پشته برابر $O(1)$ شود.

۲-۱۸ الحاق کردن و شکستن درختهای ۴ - ۳ - ۲

عمل *الحاق*[1]، دو مجموعه پویای S' و S'' و عضو x را می‌گیرد بطوریکه برای هر $x' \in S'$ و $x'' \in S''$ داریم $key[x'] < key[x] < key[x'']$ این عمل، مجموعه $S = S' \cup \{x\} \cup S''$ را بر می‌گرداند. عمل شکست[2] شبیه الحاق "معکوس" است: با دریافت مجموعه پویای S و عضو $x \in S$ این عمل مجموعه S' شامل تمام اعضا در $S\text{-}\{x\}$ که کلیدهایشان کوچکتر از $key[x]$ است و مجموعه S'' شامل تمام اعضا در $S\text{-}\{x\}$ که کلیدهایشان بزرگتر از $key[x]$ است را تولید می‌کند. در این مسئله، بررسی می‌کنیم که چگونه این اعمال را روی درخت‌های $2 - 3 - 4$، پیاده سازی کنیم. به منظور سهولت فرض می‌کنیم که اعضا تنها از کلیدها تشکیل شده‌اند و آنکه تمام مقادیر کلیدها متمایزند.

a. نشان دهید چگونه برای هر گره x از یک درخت $2 - 3 - 4$، ارتفاع زیر درخت مشتق شده از x را به عنوان فیلد $height[x]$ نگهداری کنیم. اطمینان حاصل کنید که پیاده سازی شما تأثیری بر زمانهای اجرای مجانبی جستجو، درج و حذف نمی‌گذارد.

---

1. join

2. split

b. نشان دهید چگونه عمل الحاق را پیاده سازی کنیم. برای دو درخت $4-3-2$ با نام‌های T' و T'' و کلید k الحاق باید در زمان $O(1+|h'-h''|)$ اجرا شود، که h' و h'' بترتیب ارتفاعهای T' و T'' هستند.

c. مسیر p از ریشه یک درخت $4-3-2$ بنام T تا یک کلید k مفروض، مجموعه S' شامل کلیدهای T که کوچکتر از k هستند، و مجموعه S'' شامل کلیدهای T که از k بزرگترند را در نظر بگیرید. نشان دهید p، S' را به یک مجموعه از درختهای $\{T'_0, T'_1, ..., T'_m\}$ و یک مجموعه از کلیدهای $\{k'_1, k'_2, . . . , k'_m\}$ می‌شکنند، که برای $i = 1, 2, . . . , m$ و برای هر کلید $y \in T'_{i-1}$ و $z \in T'_i$ داریم $y < k'_i < z$. چه رابطه‌ای بین ارتفاع T'_{i-1} و T'_i وجود دارد؟ توضیح چگونه p، S'' را به مجموعه های درخت‌ها و کلیدها می‌شکند.

d. نشان دهید چگونه عمل شکست را روی T پیاده سازی کنیم. از عمل الحاق برای گردآوری کلیدهای داخل S' به داخل یک درخت $4-3-2$ بنام T' و گردآوری کلیدهای داخل S'' به داخل یک درخت 2-3-4 بنام T'' استفاده کنید. زمان اجرای عمل شکست باید $O(lgn)$ باشد، که n تعداد کلیدها در T است. (راهنمایی: هزینه‌های الحاق باید تلسکوپی باشند.)

فصل ۱۹ heapهای دوجمله‌ای

این فصل و فصل ۲۰ ساختمان داده‌هایی که به عنوان *heap* های قابل ادغام شناخته شده‌اند را ارائه می‌دهند، که پنج عمل زیر را پشتیبانی می‌کنند.

MAKE-HEAP، *heap* جدیدی که شامل هیچ عنصری نمی‌باشد را ایجاد کرده و برمی‌گرداند.

INSERT(H,x)، گره x را که فیلد *key* آن پر شده است در داخل یک *heap* بنام H اضافه می‌کند.

MINIMUM(H)، اشاره گری به گره‌ای در H که کلید آن مینیمم است را برمی‌گرداند.

EXTRACT-MIN(H)، گره‌ای از H که کلید آن مینیمم است را از H حذف و اشاره‌گری به این گره را برمی‌گرداند.

UNION(H_1,H_2)، *heap* جدیدی که شامل تمام گره‌های *heap* هـای H_1 و H_2 است را ایـجاد و برمی‌گرداند. *heap*های H_1 و H_2 با این عمل از بین می‌روند.

علاوه براین، ساختمان داده‌ها در این فصل‌ها دو عمل زیر را نیز پشتیبانی می‌کنند.

DECREASE-KEY(H,x,k)، به گره x در *heap* بنام H مقدار کلید جدید k را انتساب می‌دهد که فرض شده است از کلید جاری گره بزرگتر نباشد.[1]

DELETE(H,x)، گره x را از *heap* بنام H حذف می‌کند.

همان‌طور که جدول شکل ۱۹.۱ نشان می‌دهد، اگر احتیاج به عمل *UNION* نداشته باشیم *heap* های دودویی معمولی همانطور که در *heapsort* (فصل ۶) استفاده شد به خوبی کار می‌کنند. اعمال دیگر غیر از *UNION* در بدترین حالت زمانی، در $O(lgn)$ (یا بهتر) بر روی یک *heap* دودویی اجرا می‌شوند. اما اگر قرار باشد که عمل *UNION* پشتیبانی شود، *heap* دودویی به طور ضعیف اجرا می‌شود. عمل *UNION* برای ادغام دو *heap* دودویی، با متصل کردن دو آرایه که *heap* دودویی را نگهداری می‌کنند و سپس اجرا کردن *MIN-HEAPIFY* (تمرین ۲-۲-۶ را ملاحظه نمایید) در بدترین حالت، زمان $\Theta(n)$

---

۱. همان‌طور که در مقدمه قسمت V ذکر شد *heap*های قابل ادغام پیش‌فرض، *min-heap* های قابل ادغام هستند و بنابر این اعمال *DECREASE-KEY, EXTRACT-MIN, MINIMUM* به کار می‌روند. متناوباً می‌توانیم *max-heap* قابل ادغام با اعمال *INCREASE-KEY, EXTRACT-MAX, MAXIMUM* را تعریف کنیم.

را صرف می‌کند. در این فصل " *heap*دو جمله‌ای" که حدود بدترین حالت‌های زمانی‌اش در شکل ۱۹.۱ نشان داده شده است را بررسی می‌کنیم. بخصوص عمل *UNION* که تنها زمان *O(lgn)* را برای ادغام دو *heap* دوجمله‌ای با تعداد کل *n* عنصر صرف می‌کند.

در فصل ۲۰، *heap* های فیبوناچی را بررسی خواهیم کرد که زمان بهتری برای برخی از اعمال دارند. توجه کنید که زمان اجرای *heap* های فیبوناچی در شکل ۱۹.۱، حدود زمانی سرشکن شده[1] هستند، نه حدود بدترین حالت‌های زمانی اعمال.

این فصل فرآیند گرفتن گره قبل از درج، و آزاد کردن گره بعد از حذف را نادیده می‌گیرد، فرض می‌کنیم کدهایی که روال‌های *heap* را فراخوانی می‌کنند به این جزئیات رسیدگی می‌کنند.

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

شکل ۱۹.۱ زمان اجرا برای اعمال بر روی سه پیاده سازی *heap*های قابل ادغام. تعداد اقلام در *heap* (ها) در زمان یک عمل به وسیله *n* نشان داده می‌شود.

*heap*های دودویی، *heap*هـای دو جمله‌ای و *heap*هـای فیبوناچی، همگی در پشتیبانی عمل *SEARCH* نا کارآمد هستند؛ و عمل یافتن گره با کلید داده شده مدتی به طول می‌انجامد. به این دلیل اعمالی مانند *DECREASE-KEY*و *DELETE*که به گره داده شده رجوع می‌کنند، به عنوان قسمتی از ورودیشان نیاز به اشاره‌گری به آن گره دارند. مانند بحثی که در مورد صف اولویت در بخش ۶.۵ انجام دادیم وقتی در عمل از *heap* قابل ادغام استفاده می‌کنیم اغلب در هر عنصر *heap* قابل ادغام یک اتصال به شیء کاربردی متناظر ذخیره می‌کنیم. همان‌طور که در هر شیء کاربردی یک اتصال به عنصر متناظر در *heap* قابل ادغام نگه می‌داریم. خاصیت دقیق این اتصال‌ها به کاربرد و پیاده‌سازی آن بستگی دارد.

بخش ۱۹.۱ *heap*های دو جمله‌ای را بعد از تعریف درخت‌های دو جمله‌ای تشکیل دهنده آنها تعریف می‌کند. همچنین نمایش خاصی از *heap* دوجمله‌ای را معرفی می‌کند. بخش ۱۹.۲ نشان مـی‌دهد کـه

1. *Amortized*

چگونه می‌توانیم اعمال heap دوجمله‌ای که در محدوده‌های زمانی شکل ۱۹.۱ نشان داده شده‌اند را پیاده‌سازی کنیم.

۱۹.۱ درخت‌های دو جمله‌ای و heapهای دوجمله‌ای

heap دو جمله‌ای مجموعه‌ای از درخت‌های دو جمله‌ای است، بنابراین این بخش با تعریف درخت‌های دو جمله‌ای و اثبات برخی خواص کلیدی آغاز می‌شود. سپس heapهای دوجمله‌ای را تعریف کرده و نشان می‌دهیم چگونه می‌توانند نمایش داده شوند.

۱۹.۱.۱ درخت‌های دوجمله‌ای

درخت دوجمله‌ای[۱] B_k یک درخت مرتب شده است که به طور بازگشتی تعریف می‌شود. همان‌طور که در شکل (a)۱۹.۲ نشان داده شده است درخت دو جمله‌ای B_0 از یک گره تشکیل شده است. درخت دو جمله‌ای B_k از دو درخت دو جمله‌ای B_{k-1} که به هم متصل شده‌اند تشکیل شده است: ریشه یکی، سمت چپ‌ترین فرزند ریشه دیگر است. شکل (b)۱۹.۲ درخت دو جمله‌ای B_0 تا B_4 را نشان می‌دهد.

برخی ویژگی‌های درخت دو جمله‌ای با لم زیر ارائه شده است.

لم ۱۹.۱ (ویژگی‌های درخت‌های دوجمله‌ای)

برای درخت دو جمله‌ای B_k

۱ـ 2^k گره وجود دارد،

۲ـ ارتفاع درخت برابر با k است،

۳ـ برای $i = 0, 1, ..., k$ در عمق i، دقیقاً $\binom{k}{i}$ گره وجود دارد،

۴ـ ریشه دارای درجه k است که از درجه‌های گره‌های دیگر بزرگتر است؛ بعلاوه اگر فرزندان ریشه از چپ به راست با $k-1,k-2,...,0$ شماره‌گذاری شوند، آنگاه i ریشه زیر درخت B_i است.

اثبات اثبات با استقرا بر روی k صورت می‌گیرد. برای هر ویژگی، درخت دوجمله‌ای B_0 به عنوان پایه است. تعیین صحت این که هر ویژگی برای B_0 برقرار است بدیهی می‌باشد. برای گام استقرایی، فرض می‌کنیم لم برای B_{k-1} برقرار باشد.

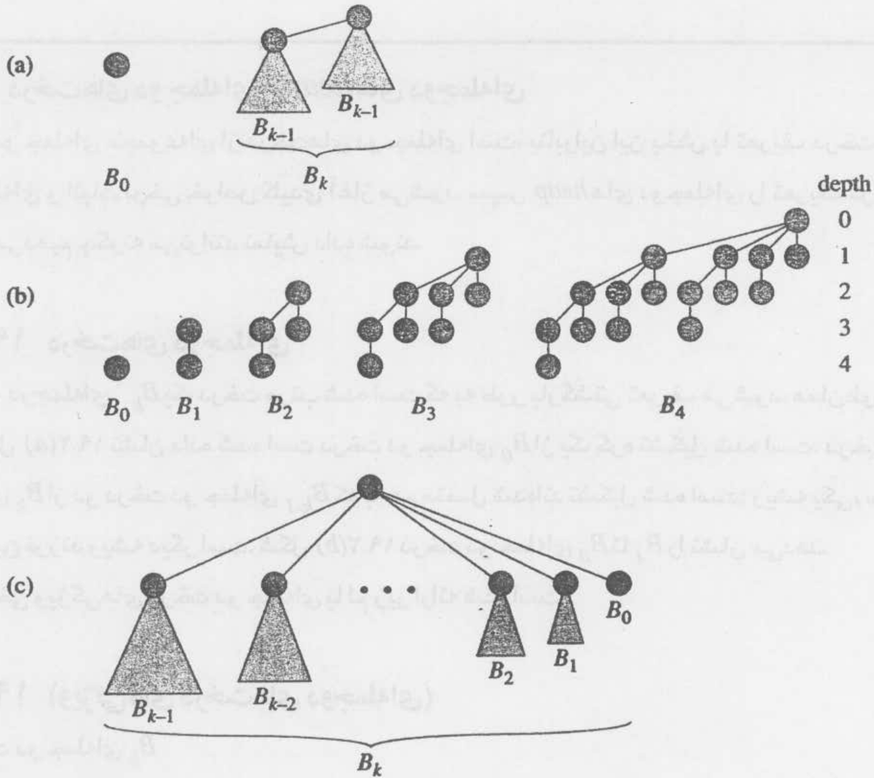
۱ـ درخت دو جمله‌ای B_k از دو کپی B_{k-1} تشکیل شده است و بنابراین B_k شامل $2^{k-1}+2^{k-1}=2^k$ گره می‌باشد.

۲ـ از آنجایی که دو کپی از B_{k-1} برای تشکیل B_k به هم متصل شده‌اند، ماکزیمم عمق یک گره در B_k یک

---

1. binomial tree

واحد از ماکزیمم عمق در B_{k-1} بیشتر است. بنا به فرض استقراء این عمق ماکزیمم برابر با $(k-1)+1=k$ است.



شکل ۱۹.۲ *(a)* تعریف بازگشتی درخت دوجمله‌ای B_k مثلث‌ها زیردرخت‌های مشتق شده را نشان می‌دهند. *(b)* درخت‌های دو جمله‌ای B_0 تا B_4. عمق گره‌ها در B_4 نشان داده شده است. *(c)* روشی دیگر از نمایش درخت دو جمله‌ای B_k.

۳ـ فرض کنید $D(k,i)$ تعداد گره‌ها در عمق i از درخت دو جمله‌ای B_k باشد. از آنجایی که B_k از دو کپی B_{k-1} که به یکدیگر متصل شده‌اند، تشکیل شده است گره در عمق i در B_{k-1} یک بار در عمق i و یک بار در عمق $i+1$ در B_k ظاهر می‌شود. به بیان دیگر تعداد گره‌ها در عمق i در B_k برابر با تعداد گره‌ها در عمق i در B_{k-1} به علاوه تعداد گره‌ها در عمق $i-1$ در B_{k-1} می‌باشد. بنابراین،

$$\begin{aligned} D(k,i) &= D(k-1,i) + D(k-1,i-1) \quad \text{(بنا به فرض استقراء)} \\ &= \binom{k-1}{i} + \binom{k-1}{i-1} \\ &= \binom{k}{i}. \end{aligned}$$

۴ـ تنها گره موجود در B_k با درجه بزرگتر از گره‌های واقع در B_{k-1}، ریشه B_k است که یک فرزند از ریشه B_{k-1} بیشتر دارد. از آنجایی که ریشه B_{k-1} دارای درجه $k-1$ می‌باشد ریشه B_k دارای درجه k است. اکنون بنا به فرض استقراء و همانطور که شکل ۱۹.۲(c) نشان می‌دهد، فرزندان ریشه B_{k-1} از چپ به راست ریشه‌های B_0،... B_{k-3}، B_{k-2} هستند. وقتی B_{k-1} به B_{k-1} متصل می‌گردد فرزندان ریشه حاصل، ریشه‌های B_0، .. B_{k-2}، B_{k-1} هستند. ■

قضیهٔ فرعی ۱۹.۲

ماکزیمم درجه هر گره در درخت دو جمله‌ای با n گره برابر lgn است.

اثبات از ویژگی‌های ۱ و ۴ لم ۱۹.۱ نتیجه می‌شود. ■

اصطلاح «درخت دوجمله‌ای» بدلیل ویژگی ۳ لم ۱۹.۱ است، زیرا عبارت‌های $\binom{k}{i}$ ضرایب دوجمله‌ای هستند. تمرین ۳-۱۹.۱ توجیهات بیشتری برای این عبارت می‌دهد.

۱۹.۱.۲ heap های دوجمله‌ای

heap دو جمله‌ای H[1] مجموعه‌ای از درخت‌های دو جمله‌ای است که *ویژگی‌های heap دوجمله‌ای*[2] که در زیر ذکر شده‌اند را دارا می‌باشد.

۱- هر درخت دو جمله‌ای در H از ویژگی *min-heap* پیروی می‌کند: کلید یک گره، بزرگتر یا مساوی کلید پدرش است. می‌گوییم چنین درختی *min-heap* مرتب شده[3] است.

۲- برای عدد صحیح غیر منفی k حداکثر یک درخت دو جمله‌ای در H که ریشه‌ای با درجه k دارد وجود دارد.

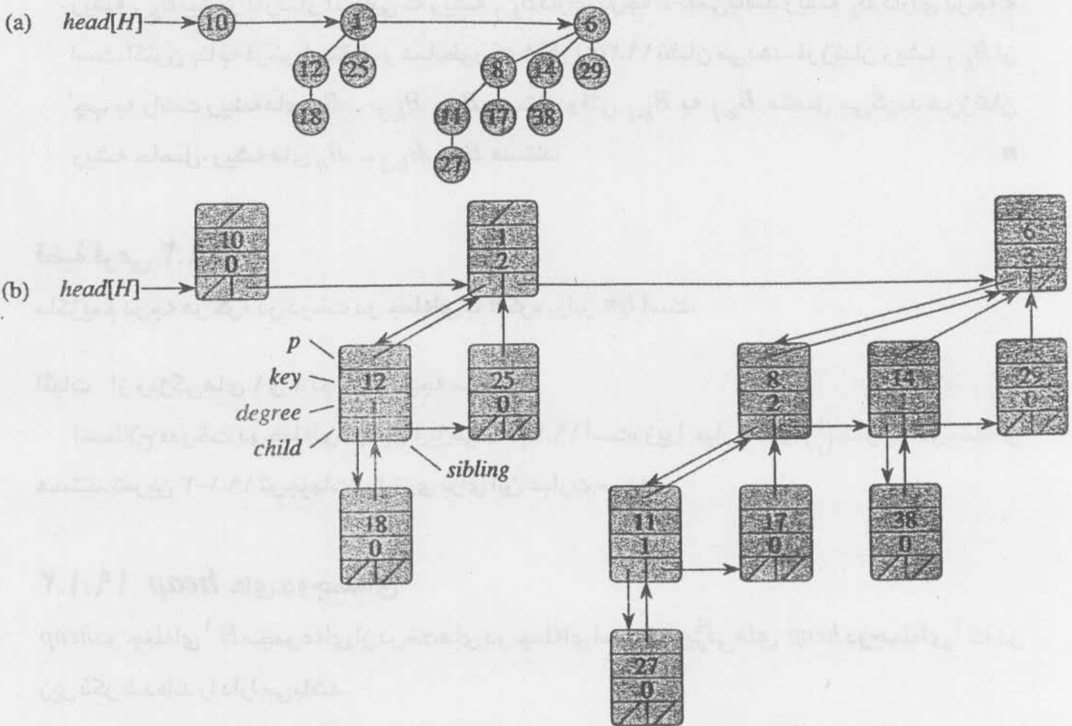
ویژگی اول به ما می‌گوید که ریشه درخت *min-heap* مرتب شده، شامل کوچکترین کلید در درخت است. ویژگی دوم دلالت می‌کند بر این که heap دوجمله‌ای با n گره شامل حداکثر $\lfloor lgn \rfloor + 1$ درخت دوجمله‌ای است. برای فهمیدن دلیل آن، مشاهده می‌کنید که نمایش دودویی عدد n دارای $\lfloor lgn \rfloor + 1$ بیت، $<b_{\lfloor lgn \rfloor}, b_{\lfloor lgn \rfloor -1}, \dots b_0>$ است بطوری که: $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$

لذا بنا به ویژگی ۱ لم ۱۹.۱ درخت دو جمله‌ای B_i در H ظاهر می‌شود، اگر و فقط اگر بیت $b_i = 1$ باشد. بنابراین heap دو جمله‌ای H حداکثر دارای $\lfloor lgn \rfloor + 1$ درخت دو جمله‌ای است.

شکل ۱۹.۳(a) یک heap دو جمله‌ای H با 13 گره را نمایش می‌دهد. نمایش دودویی 13 برابر <1101> است و H شامل درخت‌های دو جمله‌ای *min-heap* مرتب شده B_0، B_2، B_3 که به ترتیب دارای 8، 4، 1 گره برای مجموع کل 13 گره هستند.

---

1. binomial heap
2. binomial heap properties
3. min-heap-ordered



شکل ۱۹.۳ *heap* دو جمله‌ای H با $n=13$ گره. *heap(a)* شامل درخت‌های دو جمله‌ای B_0، B_2 و B_3 است که به ترتیب 1، 4 و 8 گره دارند، و در مجموع دارای $n=13$ گره می‌باشند. چون هر درخت دو جمله‌ای، *min-heap* مرتب شده است کلید هر گره، کوچکتر از کلید پدرش نیست.
همچنین لیست ریشه که یک لیست پیوندی از ریشه‌ها به ترتیب افزایش درجه می‌باشد نشان داده شده است.
(b) نمایش جزئی‌تر *heap* دو جمله‌ای H هر درخت دو جمله‌ای در نمایش فرزند چپ، همزاد راست ذخیره می‌شود و هر گره درجه خود را ذخیره می‌کند.

نمایش *heap* های دو جمله‌ای

همان‌طور که در شکل ۱۹.۳*(b)* نشان داده شده است هر درخت دوجمله‌ای در *heap* دوجمله‌ای با نمایش فرزند چپ، همزاد راست بخش ۱۰.۴ ذخیره می‌شود. هر گره، یک فیلد *key* و دیگر اطلاعات وابسته بنا به کاربرد لازم می‌گردد را شامل می‌شود. علاوه براین هر گره x شامل اشاره‌گر *p[x]* به پدرش، *child[x]* به سمت چپ‌ترین فرزندش، *sibling[x]* به اولین همزاد سمت راستش می‌باشد. اگر گره ریشه باشد آنگاه $p[x]=NIL$ است. اگر گره x فاقد فرزند باشد آنگاه $child[x]=NIL$ و اگر x سمت راست‌ترین فرزند پدرش باشد آنگاه $sibling[x]=NIL$ است. هر گره x شامل فیلد *degree[x]* می‌باشد که برابر با تعداد فرزندان x است.

همان‌طور که در شکل ۱۹.۳ نشان داده شده است ریشه‌های درخت‌های دوجمله‌ای درون *heap*

دوجمله‌ای در یک لیست پیوندی سازماندهی می‌شوند که آن را **لیست ریشه**[1] می‌نامیم. درجه‌های ریشه‌ها همانطور که لیست ریشه را می‌پیماییم به‌طور اکید افزایش می‌یابند. بنا به دومین ویژگی *heap* دو جمله‌ای، در یک *heap* دو جمله‌ای با n گره درجه ریشه‌ها زیرمجموعه‌ای از $\{0, 1, ..., \lfloor Lgn \rfloor\}$ است. فیلد *sibling* برای ریشه‌ها نسبت به گره‌های غیر ریشه دارای معنی متفاوتی است. اگر x ریشه باشد در این صورت $sibling[x]$ به ریشه بعدی در لیست ریشه اشاره می‌کند. (طبق معمول اگر x آخرین ریشه در لیست ریشه باشد آنگاه $sibling[x]=NIL$).

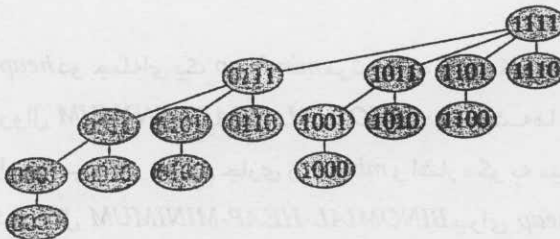
heap دو جمله‌ای داده شده H، بوسیلهٔ فیلد $head[H]$ قابل دستیابی است که اشاره‌گر ساده‌ای به اولین ریشه در لیست ریشه H می‌باشد. اگر *heap* دو جمله‌ای H فاقد عنصر باشد آنگاه $head[H]=NIL$ است.

تمرین‌ها

۱۹.۱-۱ فرض کنید x یک گره در درخت دو جمله‌ای داخل *heap* دو جمله‌ای باشد و فرض کنید $sibling[x] \neq NIL$ اگر x ریشه نباشد چطور $degree[sibling[x]]$ با $degree[x]$ مقایسه می‌شود؟ اگر x ریشه باشد چطور؟

۱۹.۱-۲ اگر x یک گره غیر ریشه در درخت دو جمله‌ای داخل *heap* دو جمله‌ای باشد چگونه $degree[x]$ با $degree[p[x]]$ مقایسه می‌شود؟

۱۹.۱-۳ فرض کنید گره‌های درخت دو جمله‌ای B_k در بصورت دودویی با پیمایش پس ترتیب، همان‌طور که در شکل ۱۹.۴ نشان داده شده است بر چسب دهی کنیم. گره x را در عمق i در نظر بگیرید که با l بر چسب دهی شده است و قرار دهید $j=k-i$ نشان دهید x دارای j تعداد 1 در نمایش دودویی است. چه تعداد k-رشته دودویی وجود دارد که دارای دقیقاً j تعداد 1 است؟ نشان دهید که در نمایش دودویی l درجه گره x برابر با تعداد 1 های سمت راست‌ترین 0 می‌باشد.



شکل ۱۹.۴ درخت دو جمله‌ای B_4 که بوسیله پیمایش پس ترتیب به صورت دوجمله‌ای برچسب دهی شده است.

---

1. root list

۱۹.۲ اعمال بر روی *heap* های دو جمله‌ای

در این بخش نشان می‌دهیم چگونه اعمال بر روی *heap* دوجمله‌ای در حدود زمانی نشان داده شده در شکل ۱۹.۱ اجرا می‌شوند تنها حدود بالا را نشان خواهیم داد؛ حدود پایین بعنوان تمرین ۱۰-۱۹.۲ واگذار شده‌اند.

ایجاد یک *heap* دو جمله‌ای جدید

برای ساختن یک *heap* دو جمله‌ای خالی روال *MAKE-BINOMIAL-HEAP* به سادگی برای شیء *H* حافظه گرفته و آنرا بر می‌گرداند که *head[H]=NIL* است. زمان اجرای آن $\Theta(1)$ است.

پیدا کردن کلید مینیمم

روال *BINOMIAL-HEAP-MIUIMUM* اشاره‌گری به گره با کلید مینیمم در *heap* دوجمله‌ای *H* با *n* گره، را بر می‌گرداند. این پیاده سازی فرض می‌کند که کلیدی با مقدار ∞ وجود ندارد. (تمرین ۵-۱۹.۲ را ملاحظه نمایید).

```
BINOMIAL-HEAP-MINIMUM(H)
1  y ← NIL
2  x ← head[H]
3  min ← ∞
4  while x ≠ NIL
5      do if key[x] < min
6            then min ← key[x]
7                 y ← x
8         x ← sibling[x]
9  return y
```

از آنجایی که *heap* دو جمله‌ای یک *min-heap* مرتب شده است، کلید مینیمم باید در لیست ریشه قرار داشته باشد. روال *BINOMIAL-HEAP-MINIMUM* همهٔ ریشه‌ها که تعداد آنها حداکثر $\lfloor lgn \rfloor +1$ است را چک می‌کند و مینیمم جاری را در *min* و اشاره گر به مینیمم جاری را در *y* ذخیره می‌کند. هنگامی که روال *BINOMIAL-HEAP-MINIMUM* برای *heap* دو جمله‌ای شکل ۱۹.۳ فراخوانی می‌شود اشاره‌گری به گرهی با کلید *1* را بر می‌گرداند.

از آنجا که حداکثر $\lfloor lgn \rfloor +1$ ریشه چک می‌شود زمان اجرای *BINOMIAL-HEAP-MINIMUM* برابر $O(lgn)$ است.

واحدسازی دو heap دوجمله‌ای

عمل واحدسازی دو heap دوجمله‌ای به عنوان یک زیرروال توسط اکثر اعمال بعدی استفاده می‌شود. روال *BINOMIAL-HEAP-UNION* به‌طور پی‌درپی درخت‌های دوجمله‌ای که ریشه‌هایشان دارای درجه یکسانی می‌باشند را به هم متصل می‌کند این روال زیردرخت B_{k-1} با ریشه y متصل می‌کند؛ به عبارت دیگر گره z را پدر گره y قرار می‌دهد. بنابراین گره z ریشه درخت B_k می‌شود.

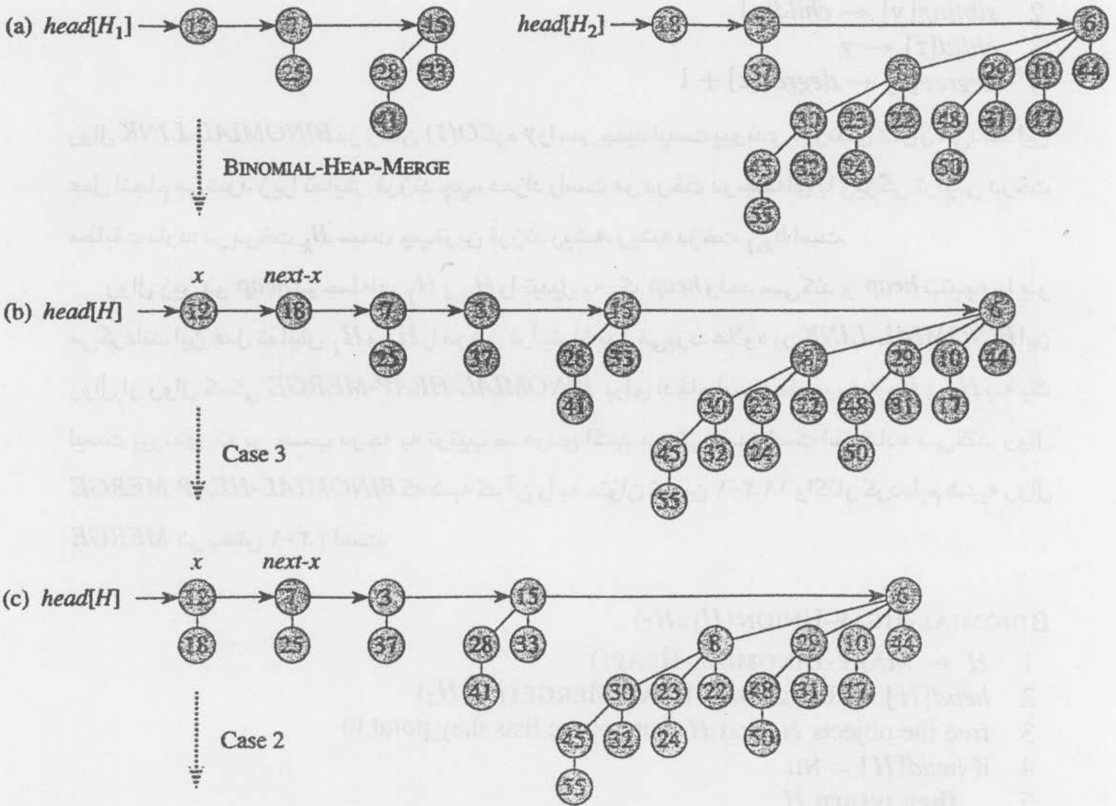
```
BINOMIAL-LINK(y, z)
1  p[y] ← z
2  sibling[y] ← child[z]
3  child[z] ← y
4  degree[z] ← degree[z] + 1
```

روال *BINOMIAL-LINK* در زمان $O(1)$ گره y را سر جدید لیست پیوندی فرزندان z قرار می‌دهد. این عمل انجام می‌شود زیرا نمایش فرزند چپ، همزاد راست هر درخت دوجمله‌ای با ویژگی ترتیبی درخت مطابقت دارد: در درخت B_k سمت چپ‌ترین فرزند ریشه، ریشه درخت B_{k-1} است.

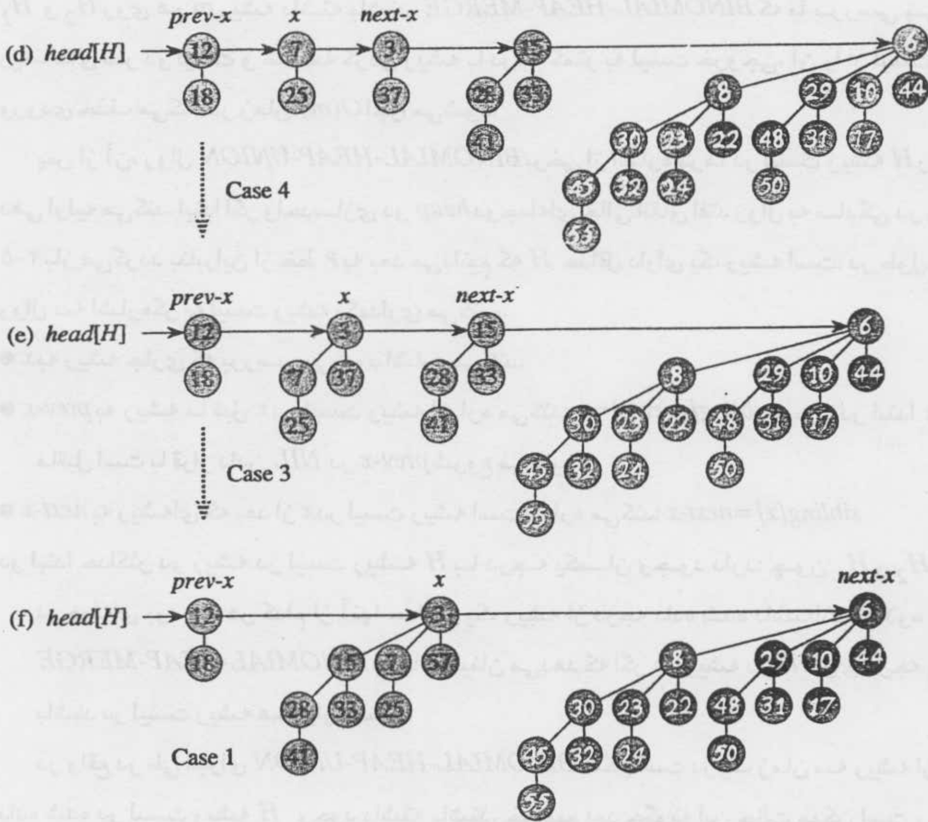
روال زیر دو heap دوجمله‌ای H_1 و H_2 را تبدیل به یک heap واحد می‌کند و heap نتیجه را بر می‌گرداند. این عمل نمایش H_1 و H_2 را در این فرآیند از بین می‌برد. علاوه بر *BINOMIAL-LINK* این روال از روال کمکی *BINOMIAL-HEAP-MERGE* برای ادغام لیست‌های ریشه H_1 و H_2 به یک لیست پیوندی که بر حسب درجه به ترتیب صعودی اکید مرتب شده است استفاده می‌کند. روال *BINOMIAL-HEAP-MERGE* که شبه کد آن را به عنوان تمرین ۱-۱۹.۲ واگذار کرده‌ایم شبیه روال *MERGE* در بخش ۱-۲.۳ است.

```
BINOMIAL-HEAP-UNION(H1, H2)
 1  H ← MAKE-BINOMIAL-HEAP()
 2  head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
 3  free the objects H1 and H2 but not the lists they point to
 4  if head[H] = NIL
 5     then return H
 6  prev-x ← NIL
 7  x ← head[H]
 8  next-x ← sibling[x]
 9  while next-x ≠ NIL
10      do if (degree[x] ≠ degree[next-x]) or
              (sibling[next-x] ≠ NIL and degree[sibling[next-x]] = degree[x])
11            then prev-x ← x                              ▷ Cases 1 and 2
12                 x ← next-x                              ▷ Cases 1 and 2
13            else if key[x] ≤ key[next-x]
```

```
14              then sibling[x] ← sibling[next-x]          ▷ Case 3
15                   BINOMIAL-LINK(next-x, x)              ▷ Case 3
16              else if prev-x = NIL                        ▷ Case 4
17                      then head[H] ← next-x              ▷ Case 4
18                      else sibling[prev-x] ← next-x      ▷ Case 4
19                   BINOMIAL-LINK(x, next-x)              ▷ Case 4
20                   x ← next-x                             ▷ Case 4
21         next-x ← sibling[x]
22    return H
```



شکل ۱۹.۵ اجرای روال *BINOMIAL-HEAP-UNION*. *(a)* heapهای دوجمله‌ای H_1 و H_2. *(b)* heap دو جمله‌ای H خروجی $BINOMIAL-HEAP-MERGE(H_1,H_2)$ است. در آغاز x اولین ریشه در لیست ریشه H است. چون هم x و هم *next-x* درجه صفر دارند و *key[x]<key[next-x]* حالت *3* به کار می‌رود *(c)* بعد از این که اتصال رخ می‌دهد، x اولین ریشه از سه ریشه با درجه مساوی است، بنابراین حالت *2* کار می‌رود. *(d)* بعد از این که همهٔ اشاره گرها یک مکان پایین‌تر در لیست ریشه حرکت می‌کنند، حالت *4* به بکار می‌رود، چون x اولین ریشه از دو ریشه با درجه مساوی است. *(e)* بعد از این که اتصال رخ می‌دهد، حالت *3* به کار می‌رود. *(f)* بعد از اتصال دیگر حالت *1* به کار می‌رود چون x درجه *3* و *next-x* درجه *4* دارد. این تکرار حلقه *while* آخرین تکرار است چون بعد از این که اشاره گرها یک مکان پایین‌تر در لیست ریشه حرکت می‌کنند *next-x=NIL*.



شکل ۱۹.۵ یک مثال از روال *BINOMIAL-HEAP-UNION* که در آن هر *4* حالت ارائه شده در شبه کد رخ می‌دهند را نشان می‌دهد.

روال *BINOMIAL-HEAP-UNION* دارای دو گذر[1] است: گذر اول با فراخوانی *BINOMIAL-HEAP-MERGE* اجرا می‌شود که لیست‌های ریشه heapهای دو جمله‌ای H_1 و H_2 را در یک لیست ریشه H ادغام می‌کند که بر حسب درجه به ترتیب صعودی اکید مرتب شده است. امّا ممکن است دو ریشه (نه بیشتر) از هر درجه وجود داشته باشد، لذا دومین گذر ریشه‌ها با درجه یکسان را به هم متصل می‌کند تا این که حداکثر یک ریشه از هر درجه باقی بماند از آنجا که لیست پیوندی H بر حسب درجه مرتب شده است عمل اتصال را به سرعت اجرا می‌کنیم.

به‌طور دقیق‌تر، روال به شکل زیر کار می‌کند، خطوط ۳-۱ با ادغام لیست‌های ریشه *heap* های دوجمله‌ای H_1 و H_2 به یک لیست ریشه‌ای H شروع می‌شود، لیست‌های ریشه‌ای H_1 و H_2 برحسب درجه به ترتیب صعودی اکید مرتب شده‌اند و روال *BINOMIAL-HEAP-MERGE* لیست ریشه H را که بر حسب درجه به ترتیب صعودی اکید مرتب شده است را بر می‌گرداند. اگر لیست‌های ریشه‌ای

1. phase

H_1 و H_2 روی هم m ریشه داشته باشند، *BINOMIAL-HEAP-MERGE* که با بررسی پی‌درپی ریشه‌های سر دو لیست و ضمیمه کردن ریشه با درجه کمتر به لیست خروجی، آن را از لیست ریشه ورودی حذف می‌کند در زمان $O(m)$ اجرا می‌شود.

پس از آن، روال *BINOMIAL-HEAP-UNION* برخی از اشاره گرها در لیست ریشه H را مقدار دهی اولیه می‌کند. ابتدا اگر واحدسازی دو *heap* دوجمله‌ای خالی اتفاق افتد روال به سادگی در خطوط ۵–۴ باز می‌گردد بنابراین از خط ۶ به بعد می‌دانیم که H حداقل دارای یک ریشه است. در طول اجرای روال سه اشاره‌گر به لیست ریشه نگهداری می‌کنیم.

- x به ریشه جاری که بررسی می‌شود اشاره می‌کند.
- *prev-x* به ریشه ما قبل x در لیست ریشه اشاره می‌کند: *sibling[prev-x]=x* (چون در ابتدا x فاقد ماقبل است با قرار دادن *NIL* در *prev-x* شروع می‌کنیم).
- *next-x* به ریشه‌ای که بعد از x در لیست ریشه است اشاره می‌کند: *sibling[x]=next-x*

در ابتدا حداکثر دو ریشه در لیست ریشه H با درجه یکسان وجود دارد: چون H_1 و H_2، *heap* دوجمله‌ای بوده‌اند هر کدام از آنها حداکثر یک ریشه از درجه داده شده داشته‌اند. علاوه بر آن *BINOMIAL-HEAP-MERGE* به ما اطمینان می‌دهد که اگر دو ریشه در H دارای درجه یکسان باشند در لیست ریشه همجوار هستند.

در واقع در طی اجرای *BINOMIAL-HEAP-UNION* ممکن است در یک زمان سه ریشه از درجه داده شده در لیست ریشه H وجود داشته باشند. خواهیم دید چگونه این حالت ممکن است رخ دهد. بنابراین در هر تکرار حلقه *while* خطوط ۲۱–۹، تصمیم می‌گیریم که آیا گره x و *next-x* را براساس درجه‌هایشان و شاید درجه *sibling[next-x]* به هم متصل کنیم. ثابت حلقه این است که در هر بار که بدنه حلقه را شروع می‌کنیم x و *next-x* غیر *NIL* هستند. (تمرین ۴–۱۹.۲ را برای ثابت دقیق حلقه ملاحظه کنید.)

حالت *1* که در شکل *(a)*۱۹.۶ نشان داده شده است زمانی رخ می‌دهد که *degree[x]≠degree[next-x]* به عبارت دیگر وقتی x ریشه درخت B_k است و برای، بنابراین یک $l>k$ *next-x* ریشه درخت B_l می‌باشد. خطوط ۱۲–۱۱ این حالت را مدیریت می‌کنند. x و *next-x* را به هم متصل نمی‌کنیم بنابراین به سادگی اشاره گر را به یک موقعیت پایین‌تر لیست حرکت می‌دهیم. بروزرسانی *next-x* که باعث می‌شود به گره بعد از گره جدید x اشاره کند در خط ۲۱ صورت می‌گیرد که برای همه حالت‌ها مشترک است.

حالت *2* که در شکل *(b)*۱۹.۶ نشان داده شده است زمانی رخ می‌دهد که x اولین ریشه از سه ریشه با درجه مساوی است، به عبارت دیگر زمانی که

degree[x]=degree[next-x]=degree[sibling[next-x]]

این مورد مشابه مورد 1 اقدام فقط می‌کنیم اشاره‌گر را یک مکان پایین‌تر در لیست حرکت می‌دهیم. تکرار بعدی یکی از دو حالت 3 یا 4 را برای ترکیب دومین و سومین ریشه از سه ریشه با درجه یکسان اجرا خواهد کرد. خط ۱۰ هر دو حالت 1 و 2 را تست می‌کند و خطوط ۱۲-۱۱ برای هر دو حالت اقدام می‌کنند.

حالت‌های 3 و 4 هنگامی رخ می‌دهند که xاولین ریشه از دو ریشه با درجه یکسان باشد به عبارت دیگر وقتی که

$$degree[x]=degree[next\text{-}x]\neq degree[sibling[next\text{-}x]]$$

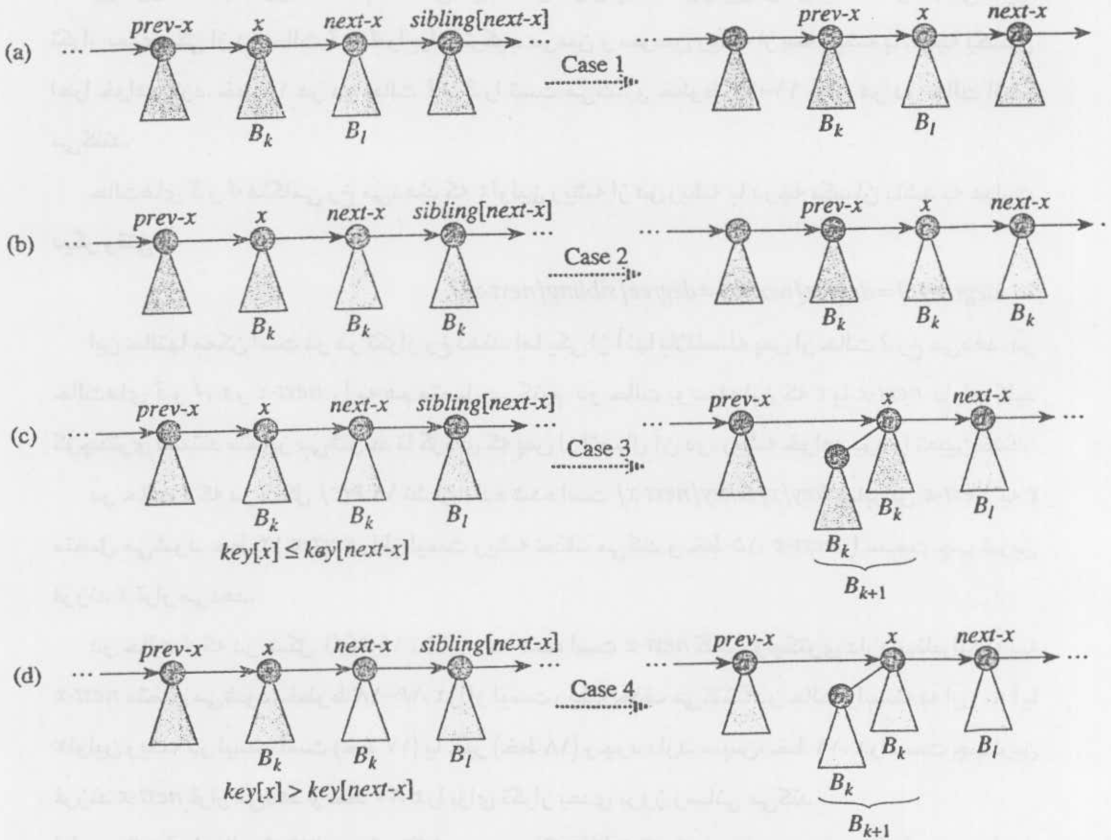
این حالتها ممکن است در هر تکرار رخ دهند، اما یکی از آنها بلافاصله پس از حالت 2 رخ می‌دهد. در حالت‌های 3 و 4 x و $next\text{-}x$ را به هم متصل می‌کنیم. دو حالت بوسیله این که x یا $next\text{-}x$ دارای کلید کوچکتری هستند متمایز می‌شوند، تا گره‌ای که پس از اتصال آن دو، ریشه خواهد بود را تعیین کنند.

در حالت 3 که در شکل ۱۹.۶(c) نشان داده شده است $key[x]\leq key[next\text{-}x]$ بنابراین $next\text{-}x$ به x متصل می‌شود. خط ۱۴ $next\text{-}x$ را از لیست ریشه حذف می‌کند و خط ۱۵، $next\text{-}x$ را سمت چپ ترین فرزند x قرار می‌دهد.

در حالت 4 که در شکل ۱۹.۶(d) نشان داده شده است $next\text{-}x$ کلید کوچکتری دارد، بنابراین x به $next\text{-}x$ متصل می‌شود. خطوط ۱۸-۱۶، x را از لیست ریشه حذف می‌کنند؛ دو حالت وابسته به این که آیا xاولین ریشه در لیست است (خط ۱۷) یا خیر (خط ۱۸) وجود دارد. سپس، خط ۱۹، x را سمت چپ‌ترین فرزند $next\text{-}x$ قرار می‌دهد و خط ۲۰، x را برای تکرار بعدی بروز رسانی می‌کند.

ادامه حالت 3 یا حالت 4 تنظیم برای تکرار بعدی حلقه $while$ یکسان است. دو درخت B_k را به هم متصل کرده‌ایم تا یک درخت B_{k+1} تشکیل دهیم که xاکنون به آن اشاره می‌کند. از قبل صفر، یک یا دو درخت B_{k+1} دیگر که از $BINOMIAL\text{-}HEAP\text{-}MERGE$ نتیجه شده است در لیست ریشه وجود داشت بنابراین xاکنون اولین درخت از یک، دو یا سه درخت B_{k+1} در لیست ریشه است. اگر xتنها درخت باشد آنگاه در تکرار بعدی وارد حالت 1 می‌شویم: $degree[x]\neq degree[next\text{-}x]$ اگر x اولین درخت از دو درخت باشد آن گاه وارد حالت 3 یا حالت 4 در تکرار بعدی می‌شویم. وقتی x اولین درخت از سه درخت باشد، در تکرار بعدی وارد حالت 2 می‌شویم.

زمان اجرای $BINOMIAL\text{-}HEAP\text{-}UNION$ برابر با $O(lgn)$ است که n برابر با تعداد کل گره‌ها در $heap$ های دوجمله‌ای H_1 و H_2 است. می‌توانیم این موضوع را به صورت زیر مشاهده کنیم. فرض کنید H_1 شامل n_1گره و H_2 شامل n_2گره باشد بنابراین $n=n_1+n_2$ لذا H_1حداکثر شامل $\lfloor lgn_1\rfloor+1$ ریشه و H_2 حداکثر دارای $\lfloor lgn_2\rfloor+1$ ریشه است، و بنابراین H بلافاصله پس از فراخوانی $BINOMIAL\text{-}HEAP\text{-}MERGE$ شامل حداکثر $\lfloor lgn_1\rfloor+\lfloor lgn_2\rfloor+2\leq 2\lfloor lgn\rfloor+2$ ریشه است بنابراین زمان اجرای $BINOMIAL\text{-}HEAP\text{-}MERGE$ برابر با $O(lgn)$ است.



شکل ۱۹.۶ چهار حالتی که در *BINOMIAL-HEAP-UNION* رخ می‌دهد. برچسب‌های *a*، *b*، *c* و *d* تنها برای شناسایی ریشه‌های مورد بحث به کار می‌روند؛ آنها درجه‌ها و کلیدهای این ریشه‌ها را مشخص نمی‌کنند. در هر حالت، *x* ریشه یک درخت B_k و $l>k$ است.

(a) حالت 1: *degree[x]≠degree[next-x]*. اشاره گرها به یک مکان پایین‌تر لیست حرکت می‌کنند.

(b) حالت 2: *degree [x]=degree[next-x]=degree[sibling[next-x]]* دوباره اشاره گرها به یک مکان پایین‌تر لیست حرکت می‌کنند و تکرار بعدی یکی از حالت‌های 3 یا 4 را اجرا می‌کند.

(c) حالت 3: *degree[x]=degree[next-x]≠degree[sibling[next-x]]* و *key[x]≤key[next-x]* . *next-x* را از لیست ریشه حذف و به *x* برای ایجاد درخت B_{k+1} متصل می‌کنیم.

(d) حالت 4: *degree[x]=degree[next-x]≠degree[sibling[next-x]]* و *key[next-x]≤key[x]* *x* را از لیست ریشه حذف و به *next-x* برای ایجاد درخت B_{k+1} متصل می‌کنیم.

هر تکرار حلقه *while* زمان $O(1)$ را صرف می‌کند و حداکثر $\lfloor lgn_1 \rfloor + \lfloor lgn_2 \rfloor + 2$ تکرار وجود دارد زیرا هر تکرار یا اشاره گر را یک مکان به سمت پایین لیست ریشه H می‌برد یا یک ریشه را از لیست ریشه حذف می‌کند. بنابراین زمان کل برابر $O(lgn)$ است.

درج یک گره

روال زیر گره xرا در heapدو جمله‌ای Hدرج می‌کند، با فرض این که از قبل برای xحافظه گرفته شده و $key[x]$پر شده است.

```
BINOMIAL-HEAP-INSERT(H, x)
1  H' ← MAKE-BINOMIAL-HEAP()
2  p[x] ← NIL
3  child[x] ← NIL
4  sibling[x] ← NIL
5  degree[x] ← 0
6  head[H'] ← x
7  H ← BINOMIAL-HEAP-UNION(H, H')
```

این روال یک heapدو جمله‌ای H'با یک گره را در زمان $O(1)$ می‌سازد و آن را با یک heapدو جمله‌ای Hبا n گره در زمان $O(lgn)$ به یک واحد تبدیل می‌کند. فراخوانی BINOMIAL-HEAP-UNION، heapدو جمله‌ای موقتی H'را آزاد می‌کند. (پیاده سازی مستقیم که BINOMIAL-HEAP-UNIONرا فراخوانی نمی‌کند، به عنوان تمرین ۸-۱۹.۲ داده شده است)

استخراج[1] گره با کلید مینیمم

روال زیر گره با کلید مینیمم را از heapدو جمله‌ای Hاستخراج و اشاره گر به گره استخراج شده را برمی‌گرداند.

```
BINOMIAL-HEAP-EXTRACT-MIN(H)
1  find the root x with the minimum key in the root list of H,
        and remove x from the root list of H
2  H' ← MAKE-BINOMIAL-HEAP()
3  reverse the order of the linked list of x's children,
        and set head[H'] to point to the head of the resulting list
4  H ← BINOMIAL-HEAP-UNION(H, H')
5  return x
```

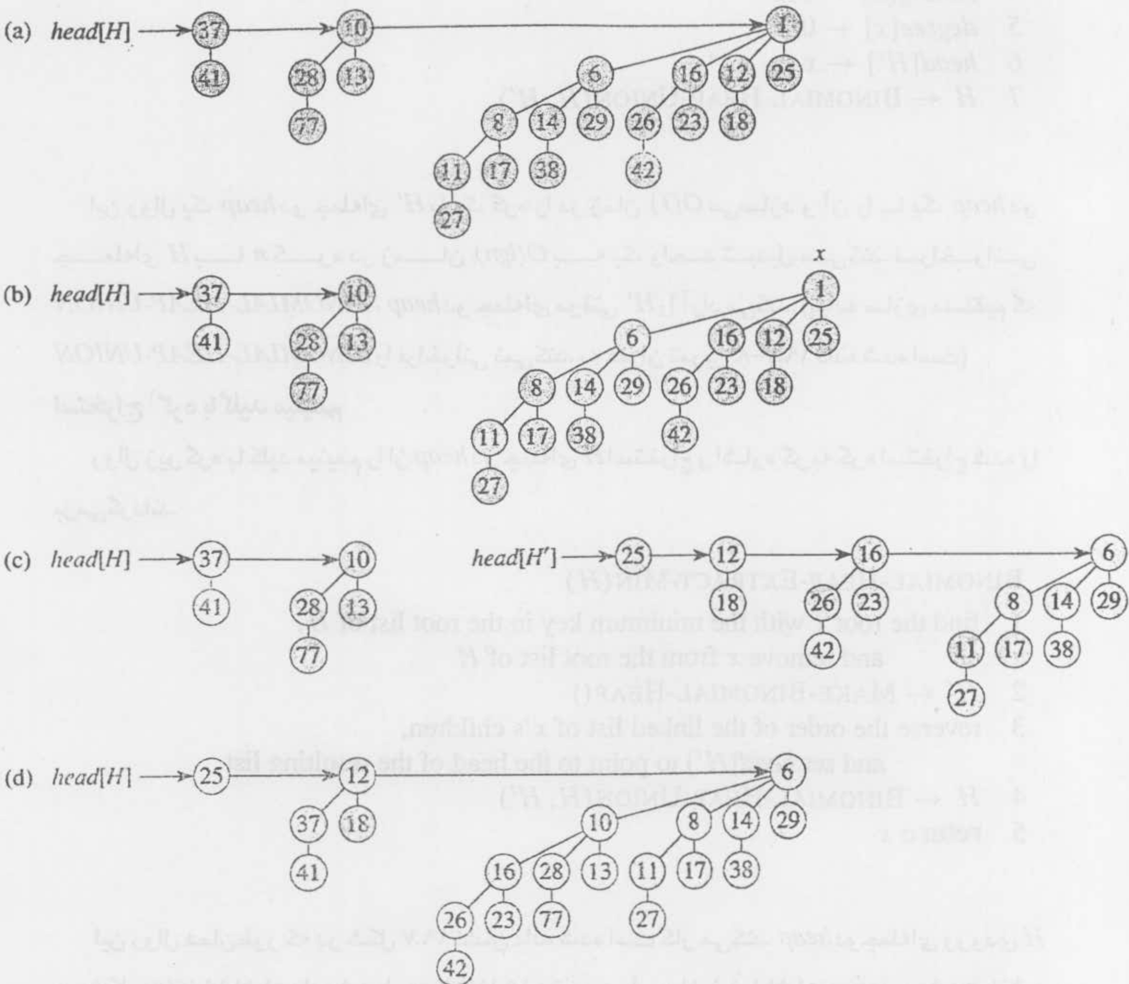
این روال همان‌طور که در شکل ۱۹.۷ نشان داده شده است کار می‌کند. heapدوجمله‌ای ورودی H در شکل (a) ۱۹.۷ نشان داده شده است. (b) ۱۹.۷ وضعیت را بعد از خط ۱ نشان می‌دهد. ریشه xبا کلید مینیمم از لیست ریشه Hحذف شده است. اگر xریشه‌ای از یک درخت B_kباشد آنگاه بنا به ویژگی 4 لم

---

1. Extract

۱۹.۱ فرزندان x از چپ به راست ریشه‌های درخت‌های $B_{k-1}, B_{k-2}, ..., B_0$ هستند شکل (c) ۷–۱۹ نشان می‌دهد که با معکوس کردن لیست فرزندهای x در خط ۳، یک *heap* دو جمله‌ای H' داریم که هر گره در درخت x به جز خود x را شامل می‌شود. چون درخت x از H در خط ۱ حذف شده است، *heap* دو جمله‌ای که از واحد سازی H و H' در خط ۴ نتیجه می‌شود و در شکل (d) ۱۹.۷ نشان داده شده است، شامل همه گره‌های اولیه در H به جز x می‌باشد. سرانجام خط ۵، x را بر می‌گرداند.

از آنجا که اگر H دارای n گره باشد هر یک از خطوط ۴–۱ زمان $(O\lg n)$ را صرف می‌کند *BINOMIAL-HEAP-EXTRACT-MIN* در زمان $O(\lg n)$ اجرا می‌شود.



شکل ۱۹.۷ عمل *BINOMIAL-HEAP-EXTRACT-MIN* (a) *heap* دو جمله‌ای H (b) ریشه x با کلید مینیمم از لیست ریشه H حذف می‌شود. (c) لیست پیوندی فرزندان x معکوس می‌شود، که *heap* دو جمله‌ای دیگر H بدست می‌آید. (d) نتیجه واحد سازی H و H'

کاهش[1] یک کلید

روال زیر مقدار کلید گره x را در heap دو جمله‌ای H به مقدار جدید k کاهش می‌دهد. این روال اگر k بزرگتر از مقدار جاری کلید x باشد یک پیغام خطا می‌دهد.

```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
 1  if k > key[x]
 2     then error "new key is greater than current key"
 3  key[x] ← k
 4  y ← x
 5  z ← p[y]
 6  while z ≠ NIL and key[y] < key[z]
 7       do exchange key[y] ↔ key[z]
 8          ▷ If y and z have satellite fields, exchange them, too.
 9          y ← z
10          z ← p[y]
```

همانطور که در شکل ۱۹.۸ نشان داده شده است این روال کلید را در یک حالت یکسان مانند min-heap دودیی کاهش می‌دهد: با حبابی بالا آمدن کلید در heap بعد از این که مطمئن شدیم کلید جدید در واقع بزرگتر از کلید جاری نیست و کلید جدید را به x انتساب دادیم روال در درخت بالا می‌رود، به همراه y که در ابتدا به گره x اشاره می‌کند. در هر تکرار حلقه *while* خطوط ۱۰-۶، $key[y]$ کلید پدر y یعنی z چک می‌شود. اگر y ریشه باشد و یا $key[y] \geq key[z]$ درخت دوجمله‌ای اکنون min-heap مرتب شده است، در غیر اینصورت گره y از خاصیت مرتب بودن min-heap تخلف کرده و کلید آن با کلید پدرش z همراه با دیگر اطلاعات وابسته عوض می‌شود. روال سپس مقدار z را در y قرار می‌دهد و یک سطح در درخت بالا آمده، و با تکرار بعدی ادامه می‌یابد.

روال *BINOMIAL-HEAP-DECREASE-KEY* زمان $O(\lg n)$ را صرف می‌کند. بنا به ویژگی ۲ لم ۱۹.۱، بیشترین عمق x $\lfloor \lg n \rfloor$ است بنابراین حلقه *while* خطوط ۱۰-۶ حداکثر $\lfloor \lg n \rfloor$ بار تکرار می‌شود.

حذف یک کلید

حذف کردن کلید گره x و اطلاعات وابسته آن از heap دوجمله‌ای در زمان $O(\lg n)$ آسان است. پیاده‌سازی زیر فرض می‌کند که در حال حاضر هیچ گرهی در heap دوجمله‌ای با مقدار کلید $-\infty$ وجود ندارد.

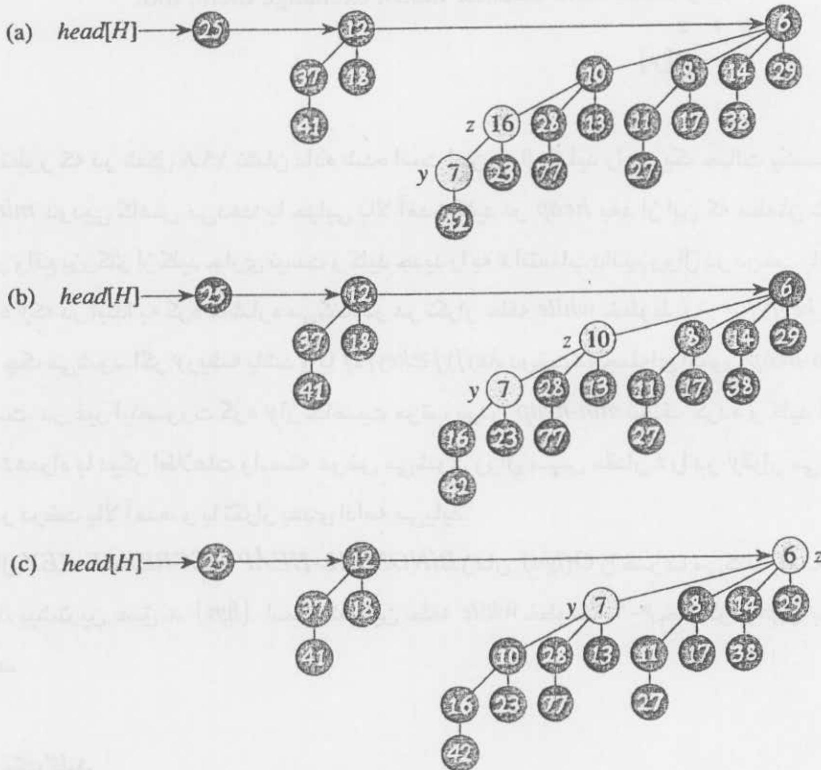
---

1. Decrease

```
BINOMIAL-HEAP-DELETE(H, x)
1  BINOMIAL-HEAP-DECREASE-KEY(H, x, −∞)
2  BINOMIAL-HEAP-EXTRACT-MIN(H)
```

روال *BINOMIAL-HEAP-DELETE* با دادن کلید ∞- به *x* باعث می‌شود گره *x* در کل *heap* دوجمله‌ای کلید مینیمم منحصر بفرد داشته باشد. (تمرین ۶-۱۹.۲ به وضعیتی رسیدگی می‌کند که در آن ∞- حتی به طور موقت هم نمی‌تواند به عنوان کلید مینیمم ظاهر شود.) سپس این کلید و اطلاعات مربوط به آن را به وسیله فراخوانی *BINOMIAL-HEAP-DECREASE-KEY* تا ریشه بالا می‌آورد. آن گاه این ریشه از *H* به وسیله فراخوانی *BINOMIAL-HEAP-EXTRACT-MIN* حذف می‌شود.

روال *BINOMIAL-HEAP-DELETE* زمان *O(lgn)* را صرف می‌کند.



شکل ۱۹.۸ عملکرد *BINOMIAL-HEAP-DECREASE-KEY* *(a)* وضعیت قبل از خط ۶ اولین تکرار حلقه *while*. کلید گره *y* کلیدش به مقدار 7 کاهش یافته است که از کلید پدر *y* یعنی *z* کمتر است. *(b)* کلیدهای دو گره عوض شده‌اند و وضعیت قبل از خط ۶ دومین تکرار نشان داده شده است. اشاره گرهای *y* و *z* به یک سطح بالاتر درخت حرکت کرده‌اند اما هنوز از ویژگی *min-heap* مرتب شده تخطّی می‌شود. *(c)* بعد از یک جابجایی دیگر و حرکت اشاره گرهای *y* و *z* به یک سطح بالاتر در می‌یابیم که ترتیب *min-heap* برقرار می‌شود، بنابراین حلقه *while* خاتمه می‌یابد.

تمرین‌ها

۱۹.۲-۱ یک شبه کد برای *BINOMIAL-HEAP-MERGE* بنویسید.

۱۹.۲-۲ *heap* دوجمله‌ای که با درج کلید *24* در داخل *heap* دوجمله‌ای نشان داده شده در شکل *(d)* ۱۹.۷ نتیجه می‌شود را نشان دهید.

۱۹.۲-۳ *heap* دو جمله‌ای که با حذف کلید *28* از *heap* دوجمله‌ای نشان داده شده در شکل *(c)*۱۹.۸ حاصل می‌شود را نشان دهید.

۱۹.۲-۴ صحت *BINOMIAL-HEAP-UNION* را با استفاده از ثابت حلقه زیر اثبات کنید:

در شروع هر تکرار حلقه *while* خطوط ۲۱-۹، *x* به ریشه‌ای که یکی از موارد زیر است اشاره می‌کند:

- تنها ریشه هم درجه‌اش،
- اولین ریشه از تنها دو ریشه هم درجه‌اش،
- اولین یا دومین ریشه از تنها سه ریشه هم درجه‌اش.

علاوه بر این همه ریشه‌هایی که قبل از ریشه ماقبل *x* در لیست ریشه قرار دارند دارای درجه منحصر بفردی می‌باشند، و اگر ماقبل *x* دارای درجه‌ای متفاوت با درجه *x* باشد درجه آن نیز در لیست ریشه، منحصر بفرد است. سرانجام درجه‌های گره‌ها به‌طور صعودی اکید با پیمایش لیست ریشه افزایش می‌یابد.

۱۹.۲-۵ توضیح دهید چرا روال *BINOMIAL-HEAP-MINMUM* ممکن است اگر کلید با مقدار ∞ موجود باشد به درستی کار نکند. شبه کد آن را چنان بازنویسی کنید که در چنین حالت‌هایی درست کار کند.

۱۹.۲-۶ فرض کنید روشی برای نمایش *k* با مقدار ∞- وجود نداشته باشد. روال *BINOMIAL-HEAP-DELETE* را به صورتی بازنویسی کنید که در این وضعیت به درستی کار کند. این روال همچنان باید در زمان *O(lgn)* اجرا شود.

۱۹.۲-۷ درباره رابطه بین درج در *heap* دوجمله‌ای و افزایش یک عدد دودویی، و رابطه بین واحدسازی دو *heap* دوجمله‌ای و جمع کردن دو عدد دودویی بحث کنید.

۱۹.۲-۸ برای روشن شدن تمرین ۱۹.۲-۷ روال *BINOMIAL-HEAP-INSERT* را جهت درج یک گره به‌طور مستقیم در *heap* دوجمله‌ای بدون فراخوانی *BINOMIAL-HEAP-UNION* بازنویسی کنید.

۱۹.۲-۹ نشان دهید اگر لیست‌های ریشه بر حسب درجه به ترتیب نزولی اکید (به جای ترتیب صعودی اکید) نگهداری شوند هر یک از اعمال *heap* دو جمله‌ای می‌توانند بدون تغییر زمان اجرای مجانبی پیاده سازی شوند.

۱۹.۲-۱۰ ورودی‌هایی را که باعث می‌شوند *BINOMIAL-HEAP-EXTRACT-MIN* *BINOMIAL-HEAP-DECREASE-KEY* و *BINOMIAL-HEAP-DELETE* در زمان *Ω(lgn)*

اجرا شوند را پیدا کنید. توضیح دهید چرا زمان اجرای بدترین حالت *BINOMIAL-HEAP-INSERT* و *BINOMIAL-HEAP-MINIMVN* برابر $\Omega(lgn)$ است نه $\Omega(lgn)$ *BINOMIAL-HEAP-UNION* (مسئله ۵-۳ را ملاحظه نمایید.)

مسائل

۱۹-۱ heapهای *2-3-4*

فصل ۱۸، درخت *2-3-4* را معرفی کرد که در آن هر گره داخلی (شاید غیر از ریشه) دارای دو، سه یا چهار فرزند است و همه برگ‌ها دارای عمق یکسان هستند. در این مسئله *heap*های *2-3-4* را پیاده سازی می‌کنیم که اعمال *heap* قابل ادغام را پشتیبانی می‌کنند.

*heap*های *2-3-4* با درخت‌های *2-3-4* در موارد زیر متفاوت هستند. در *heap*های *2-3-4* فقط برگ‌ها کلیدها را ذخیره می‌کنند و هر برگ x دقیقاً یک کلید را در فیلد *key[x]* ذخیره می‌کند. کلیدها در برگ‌ها دارای نظم مشخصی نمی‌باشند، به عبارت دیگر کلیدها از چپ به راست ممکن است با هر ترتیبی باشند. هر گره داخلی x شامل مقدار *small[x]* است که برابر با کوچکترین کلید ذخیره شده در هر برگ زیردرخت مشتق شده از x می‌باشد. ریشه r شامل فیلد *height[r]* که ارتفاع درخت است. سرانجام *heap* های *2-3-4* برای نگهداری در حافظه اصلی در نظر گرفته می‌شوند و بنابراین خواندن و نوشتن از دیسک نیاز نیست.

اعمال *2-3-4 heap* زیر را پیاده سازی کنید. هر یک از اعمال در قسمت *(a)-(e)* باید در زمان $O(lgn)$ روی یک *2-3-4 heap* با n عنصر اجرا شوند عمل *UNION* در قسمت *(f)* باید در زمان $O(lgn)$ اجرا شود، که n برابر با تعداد عناصر در دو *heap* ورودی است.

a. *MINIMUM* که یک اشاره گر به به برگ با کوچکترین کلید را بر می‌گرداند.

b. *DECREASE-KEY* که مقدار کلید برگ داده شده x را به مقدار $k \le key[x]$ کاهش می‌دهد.

c. *INSERT* که برگ x با کلید k را درج می‌کند.

d. *DELETE* که برگ داده شده x را حذف می‌کند.

e. *EXTRACT-MIN* که برگ با کوچکترین کلید را استخراج می‌کند.

f. *UNION* که دو *heap 2-3-4* را به یک واحد تبدیل کرده و یک تک *2-3-4 heap* را برگردانده و دو *heap* ورودی را نابود می‌سازد.

۱۹-۲ الگوریتم درخت پوشای مینیمم با استفاده از *heap*های دوجمله‌ای

فصل ۲۳ دو الگوریتم برای حل مسئله پیدا کردن درخت پوشای مینیمم یک گراف بدون جهت ارائه

می‌کند. در اینجا خواهیم دید که چگونه heapهای دو جمله‌ای می‌توانند برای پیدا کردن یک الگوریتم درخت پوشای مینیمم متفاوت استفاده شوند.

گراف همبند بدون جهت $G=(V,E)$ با تابع وزن $w:E\rightarrow R$ داده شده است. $w(u,v)$ را وزن یال (u,v) می‌نامیم. می‌خواهیم یک درخت پوشای مینیمم برای G پیدا کنیم: زیرمجموعه بدون دور، $T\subseteq E$ که همهٔ رئوس در V را به هم مرتبط می‌کند و وزن کل آن که برابر است با

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

مینیمم می‌شود.

شبه‌کد زیر که درستی آن می‌تواند با استفاده از بخش ۲۳.۱ اثبات شود، درخت پوشای T را می‌سازد. این شبه کد یک افراز $\{V_i\}$ از رئوس V، و به همراه هر مجموعه V_i یک مجموعه یال $E_i \subseteq \{(u, v) : u \in V_i \text{ or } v \in V_i\}$ از یالهایی که در رأسها متقاطعند را نگهداری می‌کند.

```
MST(G)
 1  T ← ∅
 2  for each vertex v_i ∈ V[G]
 3      do V_i ← {v_i}
 4         E_i ← {(v_i, v) ∈ E[G]}
 5  while there is more than one set V_i
 6      do choose any set V_i
 7         extract the minimum-weight edge (u, v) from E_i
 8         assume without loss of generality that u ∈ V_i and v ∈ V_j
 9         if i ≠ j
10            then T ← T ∪ {(u, v)}
11                 V_i ← V_i ∪ V_j, destroying V_j
12                 E_i ← E_i ∪ E_j
```

توضیح دهید چگونه این الگوریم را با استفاده از heap دوجمله‌ای پیاده‌سازی کنیم تا مجموعه‌های رئوس و یال‌ها را اداره کند. آیا نیاز به تغییری در نمایش heap دوجمله‌ای دارید؟ آیا نیازی به اضافه کردن اعمالی در ماورای اعمال heap قابل ادغام داده شده در شکل ۱۹.۱ دارید؟ زمان اجرای پیاده‌سازی خود را توضیح دهید.

۲۰ heapهای فیبوناچی

در فصل ۱۹ دیدیم که چطور heapهای دوجمله‌ای در بدترین حالت در زمان $O(lgn)$ اعمال heap قابل ادغام INSERT، MINIMUM، EXTRACT-MIN و UNION به علاوه اعمال DECREASE-KEY و DELETE را پشتیبانی می‌کند. در این فصل heapهای فیبوناچی را بررسی می‌کنیم که همان اعمال را پشتیبانی می‌کنند اما دارای این ویژگی هستند که اعمالی که شامل حذف یک عنصر نمی‌باشد در زمان سرشکن شده $O(1)$ اجرا می‌شوند.

از لحاظ تئوری، heap های فیبوناچی به ویژه زمانی که تعداد اعمال EXTRACT-MIN و DELETE نسبت به تعداد اعمال انجام شده دیگر کم است مطلوب می‌باشند. این وضعیت در بسیاری از کاربردها رخ می‌دهد. برای مثال برخی الگوریتم‌ها برای مسائل گراف ممکن است یک بار برای هر یال DECREASE-KEY را فراخوانی کنند. برای گراف‌های چگال که دارای تعداد زیادی یال می‌باشند زمان سرشکن شده $O(1)$ برای هر فراخوانی DECREASE-KEY به معنای بهبود بزرگ در مقابل زمان بدترین حالت $\Theta(lgn)$ مربوط به heap های دودویی و heap دوجمله‌ای می‌باشد. الگوریتم‌های سریع برای مسائلی مانند محاسبه درخت‌های پوشای مینیمم (فصل ۲۳) و یافتن کوتاهترین مسیرها از مبدأ واحد (فصل ۲۴) از heapهای فیبوناچی به‌طور اساسی استفاده می‌کنند.

اما از نقطه نظر عملی، ضرایب ثابت و پیچیدگی برنامه‌ای heapهای فیبوناچی در بیشتر کاربردها آنها را از heapهای دودویی (یا k تایی معمولی) نامطلوب‌تر و کم استفاده‌تر می‌کنند. بنابراین heapهای فیبوناچی از نظر تئوری برجسته هستند. اگر یک ساختمان داده ساده‌تر با حدود زمانی سرشکن شده یکسان با heapهای فیبوناچی توسعه داده می‌شد، در کاربردهای عملی نیز استفاده می‌شد.

مانند heap دوجمله‌ای، heap فیبوناچی مجموعه‌ای از درخت‌ها می‌باشد. heapهای فیبوناچی در حقیقت مبتنی بر heapهای دوجمله‌ای می‌باشند. اگر DECREASE-KEY و DELETE روی یک heap فیبوناچی فراخوانی نشوند، هر درخت در heap مانند یک درخت دوجمله‌ای است. اما heap های فیبوناچی دارای ساختاری ساده‌تر نسبت به heap های دوجمله‌ای هستند که حدود زمانی مجانبی

بهبود یافته‌ای را منظور می‌کنند. کاری که ساختار را حفظ می‌کند می‌تواند تا زمانی که اجرای آن آسان گردد به تأخیر انداخته شود.

مانند جداول پویا در بخش ۱۷.۴، *heap*های فیبوناچی مثال خوبی از یک ساختمان داده طراحی شده با تحلیل سرشکن شده را در ذهن تداعی می‌کنند. شهود و تحلیل اعمال *heap* فیبوناچی در باقی این فصل متکی بر روش پتانسیل بخش ۱۷.۳ است.

در این فصل فرض می‌شود که شما فصل ۱۹ در مورد *heap* های دوجمله‌ای را خوانده‌اید. مشخصات اعمال در آن فصل بوسیلهٔ در جدول شکل ۱۹.۱ نشان داده شده است که حدود زمانی اعمال روی *heap*های دودویی، *heap*های دوجمله‌ای و *heap*های فیبوناچی را بصورت خلاصه بیان می‌کند. نمایش ساختار *heap* های فیبوناچی به ساختار *heap* دوجمله‌ای متکی است و برخی اعمال که روی *heap*های فیبوناچی اجرا می‌شوند شبیه آنهایی هستند که روی *heap* دو جمله‌ای اجرا می‌شوند.

مانند *heap* های دوجمله‌ای، *heap*های فیبوناچی برای دادن پشتیبانی مؤثر به عمل *SEARCH* طراحی نشده‌اند؛ بنابراین اعمالی که به یک گره مفروض مراجعه می‌کنند، به یک اشاره‌گر به آن گره به عنوان قسمتی از ورودیشان نیاز دارند. وقتی از *heap* فیبوناچی در یک کاربرد استفاده می‌کنیم، اغلب یک اتصال به شیء کاربردی متناظر در هر عنصر *heap*فیبوناچی، به علاوه یک اتصال به عنصر *heap* فیبوناچی متناظر در هر شیء کاربردی ذخیره می‌کنیم.

بخش ۲۰.۱، *heap*های فیبوناچی را تعریف و درباره نمایش آنها بحث می‌کند و تابع پتانسیل که برای تحلیل سرشکن شده آن‌ها استفاده می‌شود را ارائه می‌دهد. بخش ۲۰.۲ نشان می‌دهد که چطور اعمال *heap* قابل ادغام را پیاده سازی کرده و به حدود زمانی سرشکن شده نشان داده شده در شکل ۱۹.۱ برسیم. دو عمل باقیمانده *DECREASE-KEY* و *DELETE* در بخش ۲۰.۳ بیان می‌شوند. سرانجام بخش ۲۰.۴، یک قسمت کلیدی تحلیل را تکمیل کرده و همچنین در مورد نام عجیب این ساختمان داد[illegible]

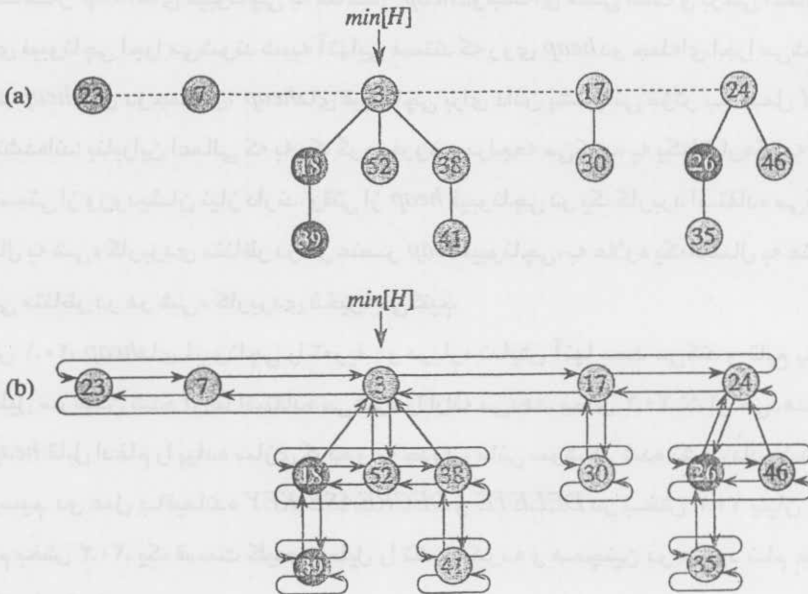
۲۰.۱ ساختار *heap*های فیبوناچی

مانند *heap* دوجمله‌ای، *heap*فیبوناچی یک [illegible]جموعه از درخت‌های *min-heap* [illegible]
درخت‌ها در یک *heap* فیبوناچی لزوماً درخت‌های دو جمله‌ای نیستند. ش[illegible] مثالی از یک [illegible]
فیبوناچی را نشان می‌دهد.

برخلاف درخت‌های داخل *heap*های دوجمله‌ای، که مرتب شده [illegible] درخت‌های داخل *heap*های فیبوناچی، مشتق شده اما غیر مرتب هستند. همانطور که شکل ۲۰.۱(b) نشان می‌دهد، هر گره x شامل یک اشاره‌گر $p[x]$ به پدرش و یک اشاره گر $child[x]$ به هر یک از فرزندانش می‌باشد. فرزندان x به صورت لیست دو پیوندی حلقوی که آن را لیست فرزند گره x می‌نامیم، به هم متصل می‌شوند. هر

فرزند بنام y در لیست فرزند، دارای اشاره‌گرهای *left[y]* و *right[y]* است که به ترتیب به همزادهای چپ و راست y اشاره می‌کنند. اگر گره y تنها فرزند باشد آنگاه داریم $left[y]=right[y]=y$ ترتیب ظاهر شدن همزادها در لیست فرزند اختیاری است.

لیست‌های دو پیوندی حلقوی (بخش ۱۰.۲ را ملاحظه نمایید) دو مزیت برای استفاده در *heap*های فیبوناچی دارند. اول اینکه می‌توانیم یک گره را از لیست دو پیوندی حلقوی در زمان $O(1)$ حذف کنیم. دوم اینکه دو لیست دو پیوندی حلقوی را می‌توانیم در زمان $O(1)$ به هم متصل کنیم (یا آن‌ها را به هم "بچسبانیم"). در تعریف اعمال *heap*های فیبوناچی بطور غیررسمی به این اعمال رجوع خواهیم کرد و جزئیات پیاده سازی آنها را به خواننده واگذار می‌کنیم.



شکل ۲۰.۱ *(a)* یک *heap* فیبوناچی شامل پنج درخت *min-heap* مرتب شده و ۱۴ گره. خط نقطه‌چین، لیست ریشه را مشخص می‌کند. گره مینیمم *heap* گرهی است که شامل کلید 3 می‌باشد. سه گره علامت‌دار سیاهرنگ می‌شوند. پتانسیل این *heap* فیبوناچی خاص برابر $5+2 . 3=11$ است. *(b)* نمایش کاملتر که اشارگرهای *p* (پیکان‌ها به بالا)، *child* (پیکان‌ها به پایین) و *left* و *Right* (پیکان‌ها به چپ و راست) را نشان می‌دهد. این جزئیات در بقیه شکل‌های این فصل حذف می‌شود، چون همهٔ اطلاعاتی که در این جا نشان داده شده است می‌تواند از آنچه در قسمت *(a)* نشان داده شده است تعیین گردد.

دو فیلد دیگر در هر گره استفاده خواهند شد. تعداد فرزندان در لیست فرزند گره x در *degree[x]* ذخیره می‌شود. فیلد *mark[x]* با مقدار بولی نشان می‌دهد آیا گره x از آخرین باری که فرزند گره دیگری شده است فرزندی از دست داده است. گره‌هایی که جدیداً ایجاد می‌شوند بدون علامت هستند و گره x زمانی که فرزند گره دیگری می‌شود بدون علامت می‌گردد. تا زمانی که در بخش ۲۰.۳ عمل

DECREASE-KEYرا ببینیم همه فیلدهای markرا با FALSEمقدار دهی می‌کنیم.

heap فیبوناچی H توسط اشاره‌گر $min[H]$ به ریشه درختی که دارای کلید مینیمم است دستیابی می‌شود؛ این گره، **گره مینیمم**[1] heap فیبوناچی نامیده می‌شود. اگر heap فیبوناچی H خالی باشد آنگاه $min[H]=NIL$.

ریشه‌های همه درخت‌ها در heapفیبوناچی با استفاده از اشاره‌گرهای leftو right در یک لیست دو پیوندی حلقوی که **لیست ریشه**[2] heap فیبوناچی نامیده می‌شود به هم متصل می‌شوند. بنابراین اشاره‌گر $min[H]$ به گرهی در لیست ریشه که کلید آن مینیم است اشاره می‌کند، ترتیب درخت‌ها در داخل لیست ریشه دلخواه است.

به یک خاصیت دیگر heapفیبوناچی Hاتکا می‌کنیم: تعداد گره‌های جاری در Hدر $n[H]$ نگهداری می‌شوند.

تابع پتانسیل

همان‌طور که ذکر شد از روش پتانسیل بخش ۱۷.۳ برای تحلیل کارآیی اعمال heapهای فیبوناچی استفاده خواهیم کرد. برای heap فیبوناچی H، به وسیله $t(H)$به تعداد درخت‌ها در لیست ریشه Hو به وسیله $m(H)$به تعداد گره‌های علامت زده شده در Hاشاره می‌کنیم. پتانسیل heap فیبوناچی Hبه صورت زیر تعریف می‌شود

$$\Phi(H) = t(H) + 2m(H) \qquad (۲۰.۱)$$

(برخی شهود مربوط به این تابع پتانسیل را در بخش ۲۰.۳ بدست خواهیم آورد.) برای مثال، پتانسیل heapفیبوناچی نشان داده شده در شکل ۲۰.۱ برابر است با $5+2\times3=11$

پتانسیل یک مجموعه‌ای از heapهای فیبوناچی برابر با جمع پتانسیل‌های heapهای فیبوناچی تشکیل دهنده آن است. فرض خواهیم کرد که یک واحد پتانسیل برای یک مقدار ثابت کار صرف شود. ثابت برای پوشش هزینه هر یک از قطعات کار با زمان ثابت که ممکن است با آن مواجه شویم به اندازه کافی بزرگ است.

فرض می‌کنیم که کاربرد heapفیبوناچی بدون heapشروع می‌شود. بنابراین پتانسیل اوّلیه برابر 0 است و بنا به معادله (۲۰.۱) در همه زیر مجموعه‌های زمانی، پتانسیل غیر منفی است با توجه به معادله (۱۷.۳) حد بالای هزینه سرشکن شده کل، یک حد بالای هزینه واقعی کل برای توالی اعمال می‌باشد.

---

1. minimum node
2. root list

ماکزیمم درجه

در تحلیل‌های سرشکن شده که در بخش‌های باقیمانده این فصل انجام خواهیم داد فرض می‌شود یک حد بالای شناخته شده $D(n)$ روی ماکزیمم درجه هر گره در یک *heap* فیبوناچی با n گره وجود دارد. تمرین ۳ - ۲۰.۲ نشان می‌دهد که وقتی فقط اعمال *heap* قابل ادغام پشتیبانی می‌شوند $D(n) \leq \lfloor lgn \rfloor$ در بخش ۲۰.۳ نشان خواهیم داد که وقتی اعمال *DECREASE-KEY* و *DELETE* را پشتیبانی می‌کنیم، $D(n)=O(lgn)$

۲۰.۲ اعمال *heap* قابل ادغام

در این بخش، ۲۰.۲ اعمال *heap* قابل ادغام را همانطور که - برای *heap*های فیبوناچی پیاده سازی شدند تعریف و تحلیل می‌کنیم. اگر فقط این اعمال -*MAKE-HEAP*، *INSERT*، *MINIMUM*، *EXTRACT-MIN* و *UNION* - پشتیبانی شوند، هر *heap* فیبوناچی به سادگی مجموعه‌ای از درخت‌های دوجمله‌ای "نامرتب" است. یک **درخت دو جمله‌ای نامرتب**[1] شبیه یک درخت دوجمله‌ای است و آن نیز به صورت بازگشتی تعریف می‌شود. درخت دو جمله‌ای نامرتب U_0، از یک گره تشکیل شده است، و درخت دو جمله‌ای نامرتب U_k از دو درخت دو جمله‌ای نامرتب U_{k-1} تشکیل شده است که برای آن، ریشه یکی فرزند ریشه دیگری قرار داده شده است. لم ۱۹.۱ که ویژگی‌های درخت‌های دو جمله‌ای را ارائه می‌دهد، برای درخت‌های دو جمله‌ای نامرتب نیز برقرار است اما با تفاوت زیر در ویژگی ۴ (تمرین ۲ - ۲۰.۲ را ملاحظه کنید).

'۴: برای درخت دو جمله‌ای نامرتب U_k، ریشه دارای درجه k است، که بزرگتر از درجه هر گره دیگر است. فرزندان ریشه، ریشه‌های زیر درخت‌های $U_0, U_1, ..., U_{k-1}$ هستند.

بنابراین اگر یک *heap* فیبوناچی با n گره مجموعه‌ای از درخت‌های دوجمله‌ای نامرتب باشد آنگاه $D(n)=lgn$.

ایده اصلی در اجرای اعمال *heap* قابل ادغام روی *heap*های فیبوناچی این است که کار را تا حد ممکن به تأخیر اندازیم. در میان پیاده سازی‌های اعمال توازن کارآیی مختلف وجود دارد. اگر تعداد درخت‌ها در *heap* فیبوناچی کم باشد آنگاه در طی عمل *EXTRACT-MIN* می‌توانیم به سرعت تعیین کنیم کدام گره باقیمانده، گره مینیمم جدید می‌شود. امّا همانطور که در *heap* دو جمله‌ای در تمرین ۱۰ - ۱۹.۲ دیدیم برای اطمینان از این که تعداد درخت‌ها کم می‌باشند هزینه‌ای را پرداخت می‌کنیم: زمان $\Omega(lgn)$ برای درج یک گره در *heap* دوجمله‌ای یا متصل کردن دو *heap* دوجمله‌ای صرف می‌شود. همانطور که خواهیم دید وقتی یک گره جدید درج می‌کنیم یا دو *heap* را واحدسازی می‌کنیم، برای

1. *unordered binomial tree*

ترکیب کردن درخت‌ها در heap فیبوناچی کوششی انجام نمی‌دهیم. ترکیب را برای عمل EXTRACT-MIN وقتی نیاز به پیدا کردن گره مینیمم جدید داریم نگه می‌داریم.

ایجاد heap فیبوناچی جدید

برای ساختن یک heap فیبوناچی خالی، روال MAKE-FIB-HEAP شیء heap فیبوناچی H را ایجاد کرده و بر می‌گرداند، بطوریکه $n[H]=0$ و $min[H]=NIL$؛ در H درختی وجود ندارد. چون $t(H)=0$ و $m(H)=0$ پتانسیل heap فیبوناچی خالی $\phi(H)=0$ است. بنابراین هزینه سرشکن شده MAKE-FIB- HEAP برابر هزینه واقعی آن، $O(1)$ است.

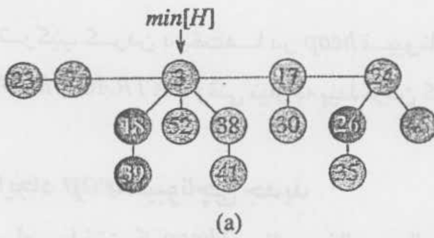
درج یک گره

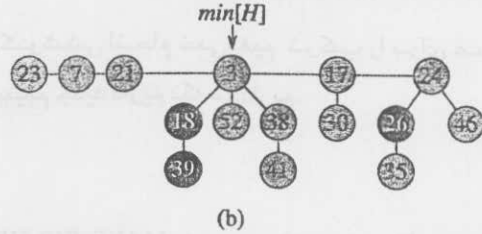
روال زیر گره x را در heap فیبوناچی H درج می‌کند، با این فرض که گره تخصصی یافته و $key[x]$ مقدار دهی شده است.

```
FIB-HEAP-INSERT(H, x)
 1  degree[x] ← 0
 2  p[x] ← NIL
 3  child[x] ← NIL
 4  left[x] ← x
 5  right[x] ← x
 6  mark[x] ← FALSE
 7  concatenate the root list containing x with root list H
 8  if min[H] = NIL or key[x] < key[min[H]]
 9     then min[H] ← x
10  n[H] ← n[H] + 1
```

بعد از خطوط ۶-۱ که فیلدهای ساختاری x مقدار دهی اولیه می‌شوند آن را لیست دوپیوندی حلقوی خودش قرار می‌دهیم، خط ۷، x را به لیست ریشه H در زمان $O(1)$ اضافه می‌کند. بنابراین گره x یک درخت min-heap- مرتب شده تک گرهی و لذا یک درخت دو جمله‌ای مرتب نشده در heap فیبوناچی می‌شود. x فاقد فرزند است و همچنین بدون علامت می‌باشد. آنگاه خطوط ۹ - ۸ در صورت لزوم اشاره‌گر به گره مینیمم heap فیبوناچی H را به روز رسانی می‌کنند. سپس خط ۱۰ مقدار $n[H]$ را یک واحد افزایش می‌دهد تا اضافه کردن گره جدید منعکس شود. شکل ۲۰.۲ اضافه کردن گره با کلید 21 به heap فیبوناچی شکل ۲۰.۱ را نشان می‌دهد.

برخلاف BINOMIAL-HEAP-INSERT، روال FIB-HEAP-INSERTR برای ترکیب درخت‌های داخل heap فیبوناچی کوشش نمی‌کند. اگر k عمل پی‌درپی FIB-HEAP-INSERT رخ دهد، آنگاه k درخت تک گرهی به لیست ریشه اضافه می‌شود.





شکل ۲۰.۲ درج یک گره در داخل *heap* فیبوناچی. *(a)heap* فیبوناچی *H* *heap(b)* فیبوناچی *H* بعد از این که گره با کلید *21* اضافه شده است. گره، درخت *min-heap* مرتب شده خودش می‌شود و سپس به لیست ریشه اضافه می‌شود، و این گره همزاد چپ ریشه می‌گردد.

برای تعیین هزینه سرشکن شده روال *FIB-HEAP-INSERT* فرض کنید *H*، *heap* فیبوناچی ورودی و *H'*، *heap* فیبوناچی حاصل باشد. آنگاه $t(H')=t(H)+1$ و $m(H')=m(H)$ و افزایش در پتانسیل برابر است با

$$((t(H)+1)+2m(H))-(t(H)+2m(H))=1$$

از آنجایی که هزینه واقعی برابر $O(1)$ است هزینه سرشکن شده برابر است با $O(1)+1=O(1)$.

یافتن گره مینیمم

گره مینیمم *heap* فیبوناچی *H* به وسیله اشاره گر *min[H]* داده می‌شود. بنابراین می‌توانیم گره مینیمم را در زمان واقعی $O(1)$ پیدا کنیم. چون پتانسیل *H* عوض نمی‌شود هزینه سرشکن شده این عمل برابر با هزینه واقعی آن یعنی $O(1)$ است.

واحد سازی دو *heap* فیبوناچی

روال زیر دو *heap* فیبوناچی H_1 و H_2 را واحد سازی می‌کند، و در این فرآیند H_1 و H_2 نابود می‌شوند. این روال به سادگی لیست ریشه‌های H_1 و H_2 را به هم متصل و سپس گره مینیمم جدید را تعیین می‌کند.

```
FIB-HEAP-UNION(H1, H2)
1  H ← MAKE-FIB-HEAP()
2  min[H] ← min[H1]
3  concatenate the root list of H2 with the root list of H
4  if (min[H1] = NIL) or (min[H2] ≠ NIL and min[H2] < min[H1])
5      then min[H] ← min[H2]
6  n[H] ← n[H1] + n[H2]
7  free the objects H1 and H2
8  return H
```

خطوط ۳-۱ لیست ریشه‌های H_1 و H_2 را به لیست ریشه جدید H تبدیل می‌کنند. خطوط ۲، ۴ و ۵ گره مینیمم H را تنظیم می‌کنند و خط ۶ مقدار $n[H]$ را برابر تعداد کل گره‌ها قرار می‌دهد. اشیاء فیبوناچی H_1 و H_2 در خط ۷ آزاد می‌شوند و خط ۸ *heap* فیبوناچی H را به عنوان نتیجه بر می‌گرداند. همانند روال *FIB-HEAP-INSERT* ترکیب درختها انجام نمی‌شود.

تغییرات در پتانسیل برابر است با

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) \\ &= (t(H) + 2\,m(H)) - ((t(H_1) + 2\,m(H_1)) + (t(H_2) + 2\,m(H_2))) \\ &= 0\,,\end{aligned}$$

زیرا

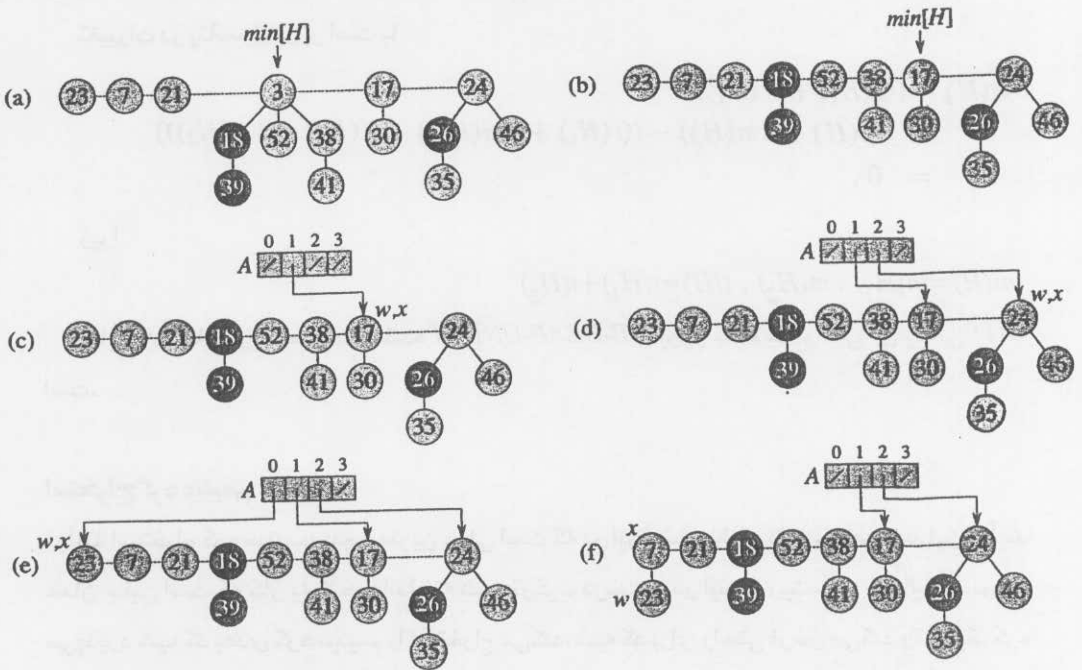
$m(H)=m(H_1)+m(H_2)$, $t(H)=t(H_1)+t(H_2)$.

بنابراین مقدار هزینه سرشکن شده *FIB-HEAP-UNION* برابر با هزینه واقعی آن یعنی $O(1)$ است.

استخراج گره مینیمم

فرآیند استخراج گره مینیمم، پیچیده‌ترین عملی است که در این بخش نشان داده خواهد شد. این فرآیند همان جایی است که کار به تأخیر انداخته شده ترکیب درخت‌ها در لیست ریشه در نهایت صورت می‌پذیرد شبه کد بعدی گره مینیمم را استخراج می‌کند. شبه کد برای راحتی فرض می‌کند وقتی یک گره از لیست پیوندی حذف می‌شود اشاره‌گرهای باقیمانده در لیست بروز رسانی می‌شوند، اما اشاره‌گرها در گره خارج شده بدون تغییر می‌مانند. این روال همچنین از روال کمکی *CONSOLIDATE* که به زودی ارائه خواهد شد استفاده می‌کند.

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z ← min[H]
 2  if z ≠ NIL
 3     then for each child x of z
 4              do add x to the root list of H
 5                 p[x] ← NIL
 6          remove z from the root list of H
 7          if z = right[z]
 8             then min[H] ← NIL
 9             else min[H] ← right[z]
10                  CONSOLIDATE(H)
11          n[H] ← n[H] − 1
12  return z
```



شکل ۲۰.۳ عملکرد *FIB-HEAP-EYTRACT-MIN* فیبوناچی *heap(a).* *H (b)* وضعیت بعد از این که گره مینیمم *z* از لیست ریشه حذف می‌شود و فرزندانش به لیست ریشه اضافه می‌شوند. *(c)-(e)* آرایه *A* و درخت‌ها بعد از هر یک از اولین سه تکرار حلقه *for* خطوط ۱۳-۳ روال *CONSOLIDATE.* لیست ریشه با شروع از گرهی که *min[H]* به آن اشاره می‌کند و دنبال کردن اشاره گرهای *right* پردازش می‌شود. هر قسمت مقادیر *x* و *w* را در انتهای یک تکرار نشان می‌دهد. *(f)-(h)* تکرار بعدی حلقه *for*، با مقادیر *x* و *w* نشان داده شده در انتهای هر تکرار حلقه *while* خطوط ۱۲ - ٦. قسمت *(f)* وضعیت بعد اولین تکرار حلقه *while* را نشان می‌دهد. گره با کلید *23* به گره با کلید *7* متصل شده است، که *x* به آن اشاره می‌کند. در قسمت *(g)* گره با کلید *17* به گره با کلید *7* که *x* همچنان به آن اشاره می‌کند متصل شده است. در قسمت *(h)* گره با کلید *24* به گره با کلید *7* متصل شده است. چون قبلاً به وسیله *A[3]* به هیچ گرهی اشاره نمی‌شد، در انتهای تکرار حلقه *for*، *A[3]* مقدار دهی می‌شود تا به ریشه درخت حاصل اشاره کند. *(i)-(l)* وضعیت بعد از هر یک از چهار تکرار بعدی حلقه *for*. *heap(m)* فیبوناچی *H* بعد از بازسازی لیست ریشه از روی آرایه *A* و تعیین اشاره گر جدید *min[H]*.

(g)

(h)

(i)

(j)

(k)

(l)

(m)

همان‌طور که در شکل ۲۰.۳ نشان داده شده روال *FIB-HEAP-EXTRACT-MIN* ابتدا با قرار دادن فرزندان گره مینیمم به عنوان ریشه و حذف گره مینیمم از لیست ریشه عمل می‌کند. سپس با متصل کردن ریشه‌هایی که دارای درجه یکسانی هستند تا زمانی که از هر درجه حداکثر یک ریشه باقی بماند، عمل ترکیب را انجام می‌دهد.

در خط ۱ با ذخیره کردن اشاره‌گر گره مینیمم در اشاره‌گر z شروع می‌کنیم؛ این اشاره‌گر در انتها برگردانده خواهد شد. اگر $z=NIL$ آنگاه *heap* فیبوناچی H در حال حاضر خالی است و عمل استخراج گره مینیمم انجام شده است. در غیر این صورت مانند روال *BINOMIAL-HEAP-EXTRACT-MIN* به وسیله قرار دادن همه فرزندان z به عنوان ریشه‌های H (با قرار دادن آنها در لیست ریشه) در خطوط

۵ - ۳ و سپس خارج کردن z از لیست ریشه در خط ۶، z را حذف می‌کنیم. اگر بعد از خط ۶، $z=right[z]$ آنگاه z تنها گره در لیست ریشه بوده و هیچ فرزندی نداشته است و تنها چیزی که مانده این است که heap فیبوناچی را در خط ۸ قبل از برگرداندن z خالی کنیم. در غیر این صورت اشاره‌گر $min[H]$ را در لیست ریشه به گره دیگری به غیر از z اشاره می‌دهیم (در این مورد $right[z]$) که لزوماً زمانی که FIB-HEAP- EXTRACT-MIN اجرا می‌شود گره جدید مینیمم نخواهد بود. شکل (b)۲۰.۳ heap فیبوناچی شکل (a)۲۰.۳ بعد از اجرای خط ۹ نشان می‌دهد.

گام بعدی که در آن تعداد درخت‌های heap فیبوناچی را کاهش می‌دهیم **ترکیب**[1] لیست ریشه H می‌باشد؛ که با فراخوانی CONSOLIDATE(H) انجام می‌شود. ترکیب کردن لیست ریشه از اجرای پی‌درپی گام‌های زیر تا وقتی که هر ریشه در لیست ریشه دارای مقدار degree مجزا شود. تشکیل می‌شود.

۱. یافتن ریشه‌های x و y در لیست ریشه با درجه یکسان، که $key[x] \leq key[y]$

۲. اتصال[2] y به x عمل حذف y از لیست ریشه و قرار دادن y به عنوان فرزند x. این عمل توسط روال FIB-HEAP-LINK انجام می‌شود، فیلد $degree[x]$ افزایش داده می‌شود روی y، اگر وجود داشته باشد، پاک می‌شود.

روال CONSOLIDATE از آرایه کمکی $A[0,...,D(n[H])]$ استفاده می‌کند؛ اگر $A[i]= y$، آنگاه y یک ریشه با $degree[y]=i$ است.

```
CONSOLIDATE(H)
 1  for i ← 0 to D(n[H])
 2       do A[i] ← NIL
 3  for each node w in the root list of H
 4       do x ← w
 5          d ← degree[x]
 6          while A[d] ≠ NIL
 7              do y ← A[d]          ▷ Another node with the same degree as x
 8                 if key[x] > key[y]
 9                    then exchange x ↔ y
10                 FIB-HEAP-LINK(H, y, x)
11                 A[d] ← NIL
12                 d ← d + 1
13          A[d] ← x
14  min[H] ← NIL
15  for i ← 0 to D(n[H])
```

1. consolidate
2. Link

```
16          do if A[i] ≠ NIL
17               then add A[i] to the root list of H
18                    if min[H] = NIL or key[A[i]] < key[min[H]]
19                       then min[H] ← A[i]
```

```
FIB-HEAP-LINK(H, y, x)
1  remove y from the root list of H
2  make y a child of x, incrementing degree[x]
3  mark[y] ← FALSE
```

به‌طور جزئی، روال *CONSOLIDATE* مانند زیر کار می‌کند. خطوط ۲-۱، آرایه *A*را با قرار دادن *NIL*در هر یک از عناصر مقدار دهی می‌کند. حلقه *for*خطوط ۱۳ - ۳ هر ریشه *w* در لیست ریشه را پردازش می‌کند. بعد از پردازش هر ریشه *w*، حلقه در یک درخت مشتق شده از یک گره *x*که ممکن است با *w*یکسان باشد یا نباشد، خاتمه می‌یابد. هیچ ریشه پردازش شده‌ای با درجه برابر با ریشه *x* وجود نخواهد داشت، بنابراین ورودی آرایه *A[degree[x]]*را چنان قرار می‌دهیم که به *x*اشاره کند. وقتی که این حلقه *for*خاتمه می‌یابد حداکثر یک ریشه از هر درجه باقیماند و آرایه *A*به هر ریشه باقی مانده اشاره می‌کند.

حلقه *while*خطوط ۱۲-۶ به‌طور پی‌درپی، ریشه *x*درختی که شامل گره *w*است را به درخت دیگری که ریشه‌اش درجه‌ای برابر با *x*دارد متصل می‌کند، تا آنجایی که ریشه دیگری دارای همان درجه نباشد. این حلقه *while*قاعده ثابت زیر را حفظ می‌کند:

در شروع هر تکرار حلقه *while*، *d=degree[x]*

از ثابت حلقه بصورت زیر استفاده می‌کنیم:

مقدار دهی اولیه: خط ۵ اطمینان حاصل می‌کند که وقتی اولین بار وارد حلقه می‌شویم ثابت حلقه برقرار است.

نگهداری: در هر تکرار حلقه *while*، *A[d]*به ریشه *y*اشاره می‌کند. چون *d = degree[x] = degree[y]* می‌خواهیم *x*و *y*را به هم متصل کنیم. به عنوان نتیجه عمل پیوند، هر کدام از *x*و *y*که کلید کوچکتری داشته باشد پدر دیگری می‌شود. و بنابراین در صورت لزوم خطوط ۹ - ۸ اشاره‌گرهای *x*و *y* را تعویض می‌کنند سپس، در خط ۱۰ به وسیله فراخوانی *FIB-HEAP-LINK(H , y , x)*، *y*را به *x* پیوند می‌دهیم. این فراخوانی *degree[x]* را افزایش می‌دهد، و *degree[y]*را در مقدار *d* باقی می‌گذارد. چون گره *y*از ریشه بزرگتر نیست، اشاره‌گر به آن در آرایه در خط ۱۱ حذف می‌شود. چون فراخوانی *FIB-HEAP-LINK* مقدار *degree[x]*را افزایش می‌دهد، خط ۱۲ ثابت *d = degree[x]* را بازیابی می‌کند.

خاتمه: حلقه *while*را تا زمانی که *A[d] = NIL*شود تکرار می‌کنیم، که در این حالت ریشه دیگری با

درجه برابر با درجه x وجود نخواهد داشت.

بعد از پایان حلقه *while*، در خط ۱۳ مقدار x را در $A[d]$ قرار می‌دهیم و تکرار بعدی حلقه *for* را اجرا می‌کنیم.

شکل‌های (e)-(c) ۲۰.۳ آرایه A و درخت نتیجه را بعد از اولین سه تکرار حلقه *for* خطوط ۱۳-۳ نشان می‌دهند. در تکرار بعدی حلقه *for* سه اتصال رخ می‌دهد؛ نتیجه آن در شکلهای (h)-(f) ۲۰.۳ نشان داده شده است. شکلهای (l)-(i) ۲۰.۳، نتیجه چهار تکرار بعدی حلقه *for* را نشان می‌دهند.

تنها چیزی که باقی می‌ماند پاک کردن لیست ریشه است. به محض اینکه حلقه *for* خطوط ۱۳-۳ کامل می‌شود، خط ۱۴ لیست ریشه را خالی می‌کند، و خطوط ۱۹ - ۱۵ آن را از روی آرایه A بازسازی می‌کنند. *heap* فیبوناچی حاصل در شکل (m) ۲۰.۳ نشان داده شده است. بعد از ترکیب کردن لیست ریشه، *FIB-HEAP-EXTRACT-MIN* با کاهش دادن $n[H]$ در خط ۱۱ و برگرداندن اشاره گری به گره حذف شده z در خط ۱۲ خاتمه می‌یابد.

ملاحظه می‌شود قبل از آنکه *FIB-HEAP-EXTRACT-MIN* اجرا شود، اگر همه درخت‌ها در داخل *heap* فیبوناچی درخت‌های دوجمله‌ای نامرتب باشند آنگاه بعد از اجرای آن روال، درخت‌ها همچنان دوجمله‌ای نامرتب هستند. دو راه برای تغییر درخت‌ها وجود دارد. اول اینکه، در خطوط ۵ - ۳ *FIB - HEAP-LINK* هر فرزند x ریشه z یک ریشه می‌شود. بنا به تمرین ۲ - ۲۰.۲ هر درخت جدید، خود یک درخت دوجمله‌ای نامرتب می‌باشد. دوم اینکه، درخت‌ها به وسیله *FIB-HEAP-LINK* به هم متصل می‌شوند تنها اگر دارای درجه برابری باشند. چون همه درخت‌ها قبل از اینکه اتصال رخ دهد درخت‌های دو جمله‌ای نامرتب هستند، دو درخت که ریشه‌های هر کدام k فرزند دارند باید ساختار U_k داشته باشند. بنابراین درخت حاصل ساختار U_{k+1} دارد.

اکنون می‌توانیم نشان دهیم که هزینه سرشکن شده استخراج گره مینیمم از *heap* فیبوناچی با n گره برابر $O(D(n))$ می‌باشد. فرض کنید H، *heap* فیبوناچی را قبل از عمل *FIB-HEAP-EXTRACT-MIN* نشان می‌دهد.

هزینه واقعی استخراج گره مینیمم به شکل زیر محاسبه می‌گردد. $O(D(n))$ از وجود حداکثر $D(n)$ فرزند گره مینیمم که در *FIB-HEAP-EXTRACT-MIN* پردازش می‌شوند و اعمالی که در خطوط ۲-۱ و ۱۹-۱۴ روال *CONSOLIDATE* صورت می‌گیرند حاصل می‌شود. آنچه باقی می‌ماند تحلیل حلقه *for* خطوط ۱۳-۳ است. اندازه لیست ریشه در فراخوانی *CONSOLIDATE* حداکثر برابر $D(n)+t(H)-1$ است، چون لیست ریشه از $t(H)$ گره اصلی لیست ریشه منهای ریشه استخراج شده بعلاوه فرزندان گره استخراج شده که تعداد آنها حداکثر $D(n)$ می‌باشد تشکیل شده است. هر بار در حلقه *while* خطوط ۱۲-۶، یکی از ریشه‌ها به دیگری متصل می‌شود و بنابراین مقدار کل کار انجام شده در حلقه *for* حداکثر متناسب با $D(n)+t(H)$ است. لذا کل کار واقعی در استخراج گره مینیمم برابر

$O(D(n) + t(H))$ است.

پتانسیل قبل از استخراج گره مینیمم برابر $t(H)+2m(H)$ است و پتانسیل بعد از آن حداکثر برابر $(D(n)+1)+2m(H)$ است، چون حداکثر $D(n)+1$ ریشه باقی می‌مانند و در طی عمل هیچ گرهی علامت زده نمی‌شود. بنابراین هزینه سرشکن شده حداکثر برابر است با

$$O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H))$$
$$= O(D(n)) + O(t(H)) - t(H)$$
$$= O(D(n))$$

زیرا می‌توانیم مقیاس واحدهای پتانسیل را بزرگتر کنیم تا تأثیر مقدار ثابت پنهان در $O(t(H))$ را بی‌اهمیت سازد. به‌طور شهودی هزینه انجام هر اتصال با کاهش در پتانسیل به واسطه کاهش تعداد ریشه‌ها به اندازه یکی می‌شود. در بخش ۲۰.۴ خواهیم دید که $D(n) = O(\lg n)$، بنابراین هزینه سرشکن شده استخراج گره مینیمم برابر $O(\lg n)$ است.

تمرین‌ها

۲۰.۲-۱ *heap* فیبوناچی که از فراخوانی *FIB-HEAP-EXTRACT-MIN* روی *heap* فیبوناچی نشان داده شده در شکل ۲۰.۳(*m*) حاصل می‌شود را نشان دهید.

۲۰.۲-۲ اثبات کنید که لم ۱۹.۱ با جایگزینی ویژگی ′۴ به جای ویژگی ۴، برای درخت‌های دو جمله‌ای نامرتب برقرار است.

۲۰.۲-۳ نشان دهید که اگر فقط اعمال *heap* قابل ادغام پشتیبانی شوند، درجه ماکزیمم $D(n)$ در یک *heap* فیبوناچی با n گره حداکثر $\lfloor lgn \rfloor$ می‌باشد.

۲۰.۲-۴ پروفسور *McGee* ساختمان داده جدیدی بر اساس *heap*های فیبوناچی ابداع کرده است. *McGee heap*، دارای ساختاری همانند *heap*فیبوناچی می‌باشد و اعمال *heap* قابل ادغام را پشتیبانی می‌کند. پیاده سازی اعمال این *heap* با اعمال *heap* فیبوناچی یکسان هستند، بجز این که عمل درج و عمل واحد سازی ترکیب را به عنوان آخرین گامشان اجرا می‌کنند. بدترین حالت زمان اجرای اعمال روی *McGee heap* چیست؟ ساختمان داده پروفسور چقدر بدیع است؟

۲۰.۲-۵ ثابت کنید وقتی تنها اعمال بر روی کلیدها، مقایسه دو کلید هستند (همانند حالتی که برای همه پیاده سازی‌های این فصل بر قرار است)، همه اعمال *heap*قابل ادغام نمی‌توانند در زمان سرشکن شده $O(1)$ اجرا شوند.

۲۰.۳ کاهش کلید و حذف یک گره

در این بخش نشان می‌دهیم چگونه کلید یک گره در داخل *heap* فیبوناچی را در زمان سرشکن شده $O(1)$ کاهش می‌دهیم و چگونه هر گره از *heap*فیبوناچی با nگره را در زمان سرشکن شده $O(D(n))$ حذف کنیم. این اعمال، این ویژگی که همه درخت‌های داخل *heap* فیبوناچی درخت‌های دو جمله‌ای نامرتب هستند را حفظ نمی‌کنند. اما آنها به اندازه کافی از نظر زمان اجرایی محدود هستند که می‌توان درجه ماکزیمم $D(n)$ را به وسیله $O(lgn)$ محدود کرد. اثبات این حد که در بخش ۲۰.۴ انجام خواهد شد نشان خواهد داد که *FIB-HEAP-EXTRACT-MIN* و *FIB-HEAP-DELETE* در زمان سرشکن شده $O(lgn)$ اجرا می‌شوند.

کاهش کلید

در شبه کد زیر برای عمل *FIB-HEAP-DECREASE-KEY*، همانند قبل فرض می‌کنیم حذف گره از لیست پیوندی، هیچکدام از فیلدهای ساختاری در گره حذف شده را تغییر نمی‌دهد.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > key[x]
```

```
2      then error "new key is greater than current key"
3   key[x] ← k
4   y ← p[x]
5   if y ≠ NIL and key[x] < key[y]
6      then CUT(H, x, y)
7           CASCADING-CUT(H, y)
8   if key[x] < key[min[H]]
9      then min[H] ← x
```

```
CUT(H, x, y)
1   remove x from the child list of y, decrementing degree[y]
2   add x to the root list of H
3   p[x] ← NIL
4   mark[x] ← FALSE
```

```
CASCADING-CUT(H, y)
1   z ← p[y]
2   if z ≠ NIL
3      then if mark[y] = FALSE
4              then mark[y] ← TRUE
5              else  CUT(H, y, z)
6                    CASCADING-CUT(H, z)
```

روال *FIB-HEAP-DECREAY-KEY* به شکل زیر کار می‌کند. خطوط ۳ - ۱ اطمینان حاصل می‌کنند که کلید جدید بزرگتر از کلید جاری xنیست و کلید جدید را به xانتساب می‌دهند. اگر xریشه باشد یا $key[x] \geq key[y]$ که yپدر xاست، آنگاه نیاز نیست که تغییرات ساختاری رخ دهد چون ترتیب *min-heap* مورد تخطی قرار نگرفته است. خطوط ۵ - ۴ این شرط را چک می‌کنند.

اگر ترتیب *min-heap* مورد تخطی قرار گرفته باشد، تغییرات زیادی ممکن است رخ دهد. **با برش** x در خط ۶ آغاز می‌کنیم. روال *CUT* اتصال بین xو پدرش yرا قطع می‌کند و xرا ریشه قرار می‌دهد.

از فیلدهای *mark* جهت بدست آوردن حدود زمانی مطلوب استفاده می‌کنیم. این فیلدها قسمت کوچکی از پیشینه هر گره را ثبت می‌کنند. فرض کنید رخدادهای زیر برای گره x انجام گرفته باشد:

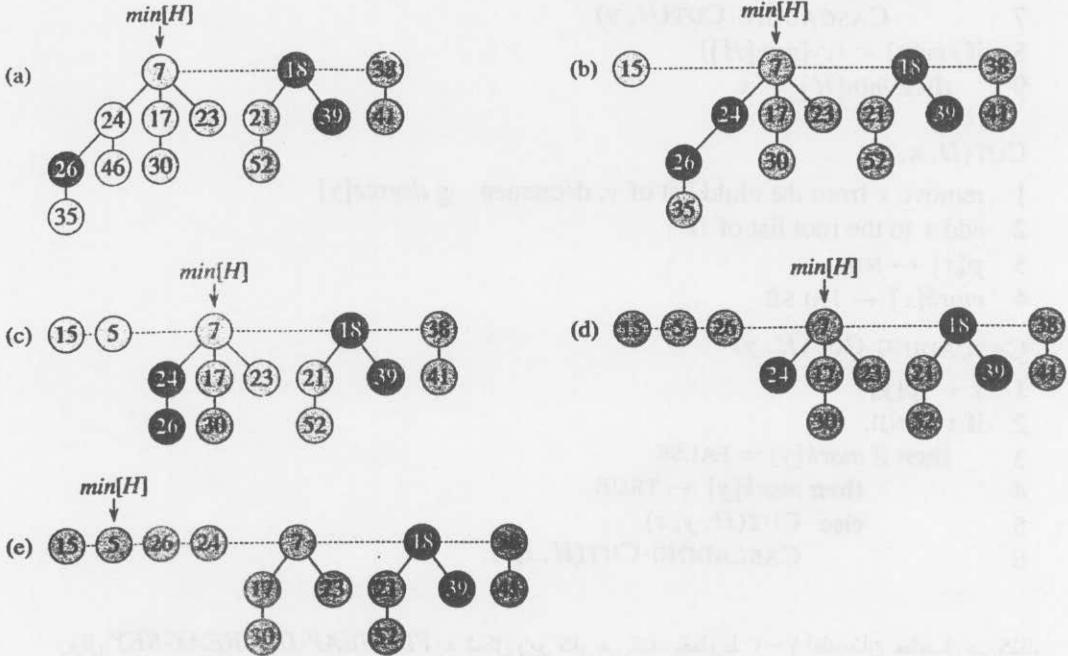
۱. زمانی xریشه بوده است،

۲. سپس xبه یک گره متّصل شده است،

۳. سپس دو فرزند xبه وسیله برش‌ها حذف شده‌اند.

به محض اینکه دومین فرزند جدا شد، xرا از پدرش جدا می‌کنیم و آن را یک ریشه جدید قرار می‌دهیم. فیلد *mark[x]* برابر *TRUE* است اگر گام‌های ۱ و ۲ رخ داده باشند و یک فرزند xجدا شده باشد. بنابراین روالِ *CUT* در خط ۴، *mark[x]* را پاک می‌کند چون این روال گام یک را اجرا می‌کند.

(اکنون می‌توانیم بفهمیم چرا خط ۳ روال *FIB-HEAP-LINK*، *mark[y]* را پاک می‌کند: گره y به گره دیگری متصل شده و بنابراین گام ۲ اجرا می‌شود. دفعه بعد که فرزند y بریده می‌شود، *mark[y]* با، *TRUE* مقداردهی خواهد شد.)



شکل ۲۰.۴ دو فراخوانی روال *FIB-HEAP-DECREASE-KEY*. *heap(a)* فیبوناچی اولیه. *(b)* گره با کلید *46* که مقدار آن به *15* کاهش یافته است. این گره یک ریشه می‌شود و پدرش (با کلید *24*) که قبلاً بدون علامت بود علامت زده می‌شود. *(e)-(c)* گره با کلید *35* که مقدار آن به *5* کاهش یافته است. در قسمت *(c)* این گره با کلید *5* ریشه می‌شود. پدرش با کلید *26* علامت زده می‌شود، بنابراین برش آبشاری رخ می‌دهد. در *(d)* گره با کلید *26* از پدرش بریده و یک ریشه علامت نزده می‌شود. از آنجا که گره با کلید *24* هم علامت زده می‌شود برش آبشاری دیگری رخ می‌دهد. در قسمت *(e)* این گره از پدرش بریده می‌شود و یک ریشه علامت نزده می‌گردد. برش‌های آبشاری در این نقطه متوقف می‌شوند چون گره با کلید *7* یک ریشه است. (حتی اگر این گره ریشه نمی‌بود برش‌های آبشاری متوقف می‌شد چون این گره علامت نزده می‌باشد.) نتیجه عمل *FIB-HEAP-DEREASE-KEY* در قسمت *(e)* نشان داده شده است که *min[H]* به گره مینیمم جدید اشاره می‌کند.

هنوز عمل کاهش کلید را به‌طور کامل انجام نداده‌ایم زیرا ممکن است زمانی که پدر x یعنی y به گره دیگری متصل شده است x دومین فرزند بریده شده از y بوده باشد. بنابراین خط ۷ روال *FIB-HEAP-DECREAS-KEY* عمل **برش آبشاری**[1] را روی y اجرا می‌کند. اگر y ریشه باشد، آنگاه

1. *cascading-cut*

تست خط ۲ روال *CASCADING-CUT* موجب می‌شود روال خاتمه یابد. اگر y علامت زده نباشد، روال آن را در خط ۴ علامت می‌زند، چون اولین فرزندش بریده شده است، و خاتمه می‌یابد. ولی اگر y علامت زده باشد، دومین فرزندش را از دست داده است؛ y در خط ۵ بریده می‌شود و روال *CASCADING-CUT* خودش را به‌طور بازگشتی در خط ۶ برای پدر y یعنی z فراخوانی می‌کند. روال *CASCADING-CUT* خودش را در راستای درخت به سمت بالا به‌طور بازگشتی فراخوانی می‌کند تا زمانیکه یک ریشه و یا یک گره علامت زده نشده را پیدا کند.

هنگامی که همه برش‌های آبشاری رخ دهند، خطوط ۹–۸ روال *FIB-HEAP-DECREAE-KEY* با بروز رسانی $min[H]$ در صورت لزوم، خاتمه می‌یابد. تنها گرهی که کلیدش تغییر کرد، گره x بود که کلیدش کاهش یافت. بنابراین گره مینیمم اولیه یا گره x، گره مینیمم جدید می‌باشد.

شکل ۲۰.۴ اجرای روفراخوانی *FIB-HEAP–DECREASE-KEY*، که با *heap* فیبوناچی نمایش داده شده در شکل ۲۰.۴ *(a)* شروع می‌شود، را نشان می‌دهد. فراخوانی اول که در شکل ۲۰.۴*(b)* نشان داده شده است، فاقد برش‌های آبشاری می‌باشد. دومین فراخوانی که در اشکال ۲۰.۴*(c)-(e)* نشان داده شده است، شامل دو برش آبشاری می‌باشد.

اکنون نشان می‌دهیم که هزینه سرشکن شده روال *FIB-HEAP-DECREASE-KEY* تنها برابر $O(1)$ می‌باشد. با مشخص کردن هزینه واقعی روال شروع می‌کنیم. روال *FIB-HEAP-DECREASE-KEY* زمان $O(1)$ بعلاوه زمان اجرای برش‌های آبشاری را صرف می‌کند. فرض کنید که روال *CASCADING-CUT* با احضار *FIB-HEAP-DECREASE-KEY*، c بار به‌طور باز گشتی فراخوانی می‌شود. هر فراخوانی *CASCADING-CUT*، بدون در نظر گرفتن فراخوانی‌های بازگشتی، زمان $O(1)$ را صرف می‌کند. بنابراین هزینه واقعی *FIB-HEAP-DECREAS-KEY*، که همه فراخوانی‌های بازگشتی را شامل می‌شود، برابر $O(c)$ است.

حال تغییرات در پتانسیل را محاسبه می‌کنیم. فرض کنید H، *heap* فیبوناچی را قبل از عمل *FIB-HEAP-DECREASE-KEY* نشان می‌دهد. هر فراخوانی بازگشتی *CASCADING-CUT*، بجزء برای آخرین فراخوانی، گره علامت زده شده را جدا می‌کند و بیت علامت گره را پاک می‌کند. پس از آن $t(H)+c$ درخت ($t(H)$ درخت اولیه، $c-1$ درخت که با برش آبشاری تولید شده‌اند، و درخت مشتق شده از x) و حداکثر $m(H)-c+2$ گره علامت زده شده ($c-1$ گره بوسیله برش‌های آبشاری بی‌علامت شده‌اند و آخرین فراخوانی *CASCADING-CUT* ممکن است یک گره را علامت زده باشد) وجود دارد. بنابراین تغییرات در پتانسیل حداکثر برابر است با

$$((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H))=4-c$$

بنابراین هزینه سرشکن شده روال *FIB-HEAP-DECREASE-KEY* حداکثر برابر است با

$$O(c)+4-c=O(1)$$

زیرا می‌توانیم مقیاس واحدهای پتانسیل را بزرگتر کنیم تا تأثیر مقدار ثابت پنهان در $O(c)$ را بی اهمیت سازد.

اکنون می‌توانید بفهمید که چرا تابع پتانسیل بصورتی تعریف شده بود تا جمله‌ای را شامل شود که دو برابر تعداد گره‌های علامت زده است. وقتی گره علامت زده شده y بوسیله یک برش آبشاری جدا می‌شود بیت علامتش پاک می‌شود بنابراین پتانسیل آن 2 واحد کاهش می‌یابد. یک واحد پتانسیل برای برش و پاک کردن بیت علامت پرداخته می‌شود و واحد دیگر، افزایش واحد در پتانسیل به واسطه ریشه شدن گره y را جبران می‌کند.

حذف یک گره

حذف یک گره از *heap* فیبوناچی با n گره در زمان سرشکن شده $O(D(n))$ آسان می‌باشد، که این کار توسط شبه کد زیر انجام می‌شود. فرض کنیم هیچ کلیدی مقدار ∞- در حال حاضر در *heap* فیبوناچی وجود ندارد.

```
FIB-HEAP-DELETE(H, x)
1  FIB-HEAP-DECREASE-KEY(H, x, −∞)
2  FIB-HEAP-EXTRACT-MIN(H)
```

FIB-HEAP-DELETE شبیه به *BINOMIAL-HEAP-DELETE* می‌باشد. این روال با دادن کلید مینیمم ∞- به x باعث می‌شود که x گره مینیمم در *heap* فیبوناچی شود. آنگاه گره x بوسیله روال *FIB-HEAP-EXTRACT-MIN* از *heap* فیبوناچی حذف می‌گردد. زمان سرشکن شده روال *FIB-HEAP-DELETE*، برابر با مجموع زمان سرشکن شده، $O(1)$ روال *FIB-HEAP-DECREASE-MIN* و زمان سرشکن شده، $O(D(n))$ روال *FIB-HEAP-EXTRACT-MIN* است. از آنجا که در بخش ۲۰.۴ خواهیم دید که $D(n) = O(lgn)$ زمان سرشکن شده *FIB-HEAP-DELETE* برابر با $O(lgn)$ است.

تمرین‌ها

۲۰.۳ - ۱ فرض کنید که ریشه x در *heap* فیبوناچی علامت زده شده است. توضیح دهید که چگونه x ریشه‌ای علامت زده گردیده است. ثابت کنید اینکه x علامت زده شده باشد در تحلیل اهمیتی ندارد، اگر چه x ریشه‌ای نیست که در ابتدا به گره دیگری متصل شده و سپس یک فرزند از دست داده باشد.

۲۰.۳ - ۲ با استفاده از تحلیل جمعی، زمان سرشکن شده $O(1)$ روال *FIB-HEAP-DECREASE-KEY* را به عنوان هزینه میانگین هر عمل توجیه نمایید.

۲۰.۴ محدود کردن ماکزیمم درجه

برای اثبات اینکه زمان سرشکن شده روال‌های FIB-HEAP-DELETE و FIB-HEAP-EXTRACT-MIN برابر با $O(lgn)$ است، باید نشان دهیم که حد بالای $D(n)$ روی درجه هر گره heap فیبوناچی با n گره برابر با $O(lgn)$ است. بنا به تمرین ۳-۲.۲۰ وقتی که همه درخت‌ها در heap فیبوناچی درخت‌های دوجمله‌ای نامرتب هستند، $D(n)=\lfloor lgn \rfloor$ ولی برش هایی که FIB-HEAP-DEREASE-KEY رخ می‌دهد ممکن است باعث شود که درخت‌ها در heap فیبوناچی از ویژگی‌های درخت‌های دوجمله‌ای نامرتب تخطی کنند. در این بخش نشان خواهیم داد که چون یک گره را به محض آنکه دو فرزندش را از دست می‌دهد از پدرش جدا می‌کنیم، $D(n)$ برابر با $O(lgn)$ است. بویژه نشان خواهیم داد که $\lfloor lg_{\phi}n \rfloor \leq D(n)$ که $\phi=(1+\sqrt{5})/2$ می‌باشد.

اساس تحلیل به شکل زیر است. برای هر گره x در heap فیبوناچی، $size(x)$ را تعداد گره هایی، شامل خود گره x که در زیردرخت مشتق شده از x قرار می‌گیرند تعریف می‌کنیم. (توجه کنید که لازم نیست گره x در لیست ریشه باشد - بلکه هر گرهی می‌تواند باشد.) نشان خواهیم داد که $size[x]$ در $degree[x]$ نمایی است. در نظر داشته باشید که $degree[x]$ همیشه به عنوان یک شمارش دقیق درجه x نگهداری می‌شود.

لم ۲۰.۱

فرض کنید x یک گره در heap فیبوناچی باشد و فرض کنید $degree[x]=k$ همچنین $y_1, y_2, \dots, y_k$ فرزندان گره x را در ترتیبی از اولین تا آخرین به گره x متصل شده‌اند نشان می‌دهد. آنگاه $degree[y_1] \geq 0$ و برای هر i که $i = 2, 3, \dots, k$ داریم

$degree[y_i] \geq i-2$

اثبات واضح است که $degree[y_1] \geq 0$

برای $i \geq 2$ توجه داریم که وقتی y_i به x متصل شده است همهٔ $y_1, y_2, \dots, y_{i-1}$ فرزندان x بوده‌اند، بنابراین باید می‌داشتیم $degree[x]=i-1$ گره y_i وفقط اگر $degree[x]=degree[y_i]$ باشد به گره x متصل می‌شود، بنابراین باید در آن زمان نیز می‌داشتیم $degree\ [y_i]=i-1$. پس گره y_i حداکثر یک فرزند از دست داده است، چون اگر دو فرزند از دست داده بود از x جدا می‌شد. نتیجه می‌گیریم که $degree[y_i] \geq i-2$. ■

در نهایت به قسمتی از تحلیل می‌رسیم که نام «heapهای فیبوناچی» را توضیح می‌دهد. از بخش ۳.۲ به خاطر آورید که برای $k= 0, 1, \dots$، kامین عدد فیبوناچی با رابطه بازگشتی زیر تعریف می‌شود:

$$F_k = \begin{cases} 0 & \text{if } k = 0\,, \\ 1 & \text{if } k = 1\,, \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2\,. \end{cases}$$

لم زیر، راه دیگری برای بیان F_k ارائه می‌دهد.

لم ۲۰.۲

برای همه اعداد صحیح $k \geq 0$

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i\,.$$

اثبات اثبات بوسیله استقرا روی k انجام می‌شود. وقتی $k=0$ است داریم

$$\begin{aligned} 1 + \sum_{i=0}^{0} F_i &= 1 + F_0 \\ &= 1 + 0 \\ &= 1 \\ &= F_2\,. \end{aligned}$$

اکنون فرض استقرا که

$$F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$$

را در نظر می‌گیریم و داریم

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) \\ &= 1 + \sum_{i=0}^{k} F_i\,. \end{aligned}$$

■

لم زیر و قضیه فرعی آن تحلیل را کامل می‌کنند. آنها از نامساوی

$$F_{k+2} \geq \phi^k\,,$$

(در تمرین ۷ – ۳.۲ اثبات شد) استفاده می‌کنند، که ϕ نسبت طلایی [1] است، که در معادله (۳.۲۲) بصورت

$$\phi = (1+\sqrt{5})/2 = 1.61803$$

تعریف شد.

---

1. golden ratio

لم ۲۰.۳

فرض کنید xگرهی در heapفیبوناچی باشد و همچنین $k=degree[x]$ آنگاه $size(x) \geq F_{k+2} \geq \phi^k$ که در آن $\phi = (1+\sqrt{5})/2$

اثبات فرض کنید s_kمینیمم مقدار ممکن $size(z)$برای همه گره‌های z بطوریکه $degree[z]=k$ را نشان می‌دهد. به‌طور جزئی‌تر $s_0=1$، $s_1=2$و $s_2=3$. عدد s_kحداکثر برابر $size(x)$است و به وضوح مقدار s_k به‌طور یکنواخت با k افزایش می‌یابد همانند آنچه در لم ۲۰.۱ بیان شد، فرض کنید $y_1, y_2, ..., y_k$ فرزندان xرا به ترتیبی که به x متصل شده‌اند نشان می‌دهد. برای محاسبه حد پایین روی $size(x)$، یک واحد برای خود xو یک واحد برای اولین فرزند یعنی y_1(که برای آن $size(y) \geq 1$) در نظر می‌گیریم داریم

$$\begin{aligned} size(x) &\geq s_k \\ &= 2+\sum_{i=2}^{k} s_{degree[y_i]} \\ &\geq 2+\sum_{i=2}^{k} s_{i-2}\,, \end{aligned}$$

که خط آخر از لم ۲۰.۱ (بطوریکه $degree[y_i] \geq i-2$) و یکنواختی s_k(بطوریکه $s_{degree[y_i]} \geq s_{i-2}$) اثبات می‌گردد.

اکنون با استقرا روی kنشان می‌دهیم که برای همه اعداد صحیح غیر منفی k، $s_k > f_{k+1}$. پایه‌ها برای $k=0$و $k=1$کم اهمّیت هستند. برای گام استقرایی فرض می‌کنیم $k \geq 2$و $s_i > F_{i+2}$ برای $i = 0,1 ,..., k-1$. داریم

$$\begin{aligned} s_k &\geq 2+\sum_{i=2}^{k} s_{i-2} \\ &\geq 2+\sum_{i=2}^{k} F_i \\ &= 1+\sum_{i=0}^{k} F_i \\ &= F_{k+2} \qquad \text{(بنا به لم ۲۰.۲)} \end{aligned}$$

بنابراین نشان داده‌ایم که $size(x) \geq s_k \geq F_{k+2} \geq \phi^k$. ■

قضیه فرعی ۲۰.۴

ماکزیمم درجه $D(n)$هر گره در داخل heapفیبوناچی با nگره برابر $O(lgn)$است.

اثبات فرض کنید xیک گره در *heap* فیبوناچی با n گره باشد و قرار دهید $k = degree[x]$. بنا به لم ۲۰.۳ داریم $n \ge size(x) \ge \phi^k$. گرفتن لگاریتم در پایه ϕداریم $k \le lgn_\phi$. (در واقع چون kیک عدد صحیح است، $k \le \lfloor lgn_\phi \rfloor$.) لذا ماکزیمم درجه $D(n)$هر گره برابر با $O(lgn)$ است.

■

تمرین‌ها

۱ - ۲۰.۴ پروفسور *Pinocchio* ادعا می‌کند که ارتفاع *heap* فیبوناچی با n گره برابر $O(lgn)$ است. با نمایش اینکه برای هر عدد صحیح مثبت n، یک توالی از اعمال *heap* فیبوناچی که یک *heap* فیبوناچی را ایجاد می‌کند فقط از یک درخت درخت که زنجیره خطی از n گره است تشکیل شده است، نشان دهید که پروفسور اشتباه می‌کند.

۲ - ۲۰.۴ فرض کنید قانون برش آبشاری را برای جدا کردن گره xاز پدرش به محض اینکه kامین فرزندش را به ازای مقدار ثابت صحیح kاز دست می‌دهد تعمیم دهیم. (این قانون در بخش ۲۰.۳ از $k=2$ استفاده می‌کند.) برای چه مقادیر از k، $D(n) = O(lgn)$است.

مسائل

۲۰ - ۱ پیاده‌سازی دیگری از حذف

پروفسور *Pisano* روال *FIB-HEAP-DELETE* متفاوت زیر را پیشنهاد کرده است، و ادعا می‌کند وقتی گرهی که ابتدا حذف شده است گرهی نیست که با *min[H]* بـه آن اشـاره مـی‌شود، ایـن روال سریعتر کار می‌کند.

```
PISANO-DELETE(H, x)
1  if x = min[H]
2     then FIB-HEAP-EXTRACT-MIN(H)
3     else  y ← p[x]
4           if y ≠ NIL
5              then CUT(H, x, y)
6                   CASCADING-CUT(H, y)
7           add x's child list to the root list of H
8           remove x from the root list of H
```

a.ادعای پروفسور که این روال سریعتر اجرا می‌شود تا حدی بر پایه این فرض است که خط ۷ می‌تواند در زمان واقعی $O(1)$ اجرا شود. اشتباه این فرض چیست؟

b.یک حد بالای مناسب برای زمان واقعی *PISANO-DELETE* وقتی که xبرابر $min[H]$ نمی‌باشد ارائه دهید. حد شما باید بر حسب $degree[x]$ و تعداد c فراخوانی روال *CASCADING-CUT* باشد.

c.فرض کنید روال *PISANO-DELETE(H,x)* را فراخوانی می‌کنیم و H'، *heap* فیبوناچی حاصل باشد. با فرض اینکه گره x ریشه نیست، پتانسیل H' را بر حسب $degree[x]$، c، $t(H)$ و $m(H)$ محدود کنید.

d.نتیجه بگیرید که زمان سرشکن شده برای *PISANO-DELETE* به‌طور مجانبی بهتر از زمان سرشکن شده برای *FIB-HEAP-DELETE* نیست، حتی زمانیکه $x \neq min[H]$ باشد.

۲۰-۲ اعمال بیشتر heapفیبوناچی

می‌خواهیم *heap* فیبوناچی H را بهبود بخشیم تا دو عمل جدید را بدون اینکه زمان اجرای سرشکن شده هیچ یک از اعمال *heap* فیبوناچی تغییر کند، پشتیبانی نماید.

a.عمل *FIB-HEAP-CHANGE-KEY(H,x,k)* کلید گره xرا با مقدار kعوض می‌کند. یک پیاده سازی کارآمد از *FIB-HEAP-CHANGE-KEY* را ارائه دهید، زمان اجرای سرشکن شده پیاده سازی‌تان را برای حالت‌های کارآمدی که kبزرگتر، کوچکتر و یا مساوی $key[x]$ است، تحلیل کنید.

b.پیاده سازی کارآمدی از *FIB-HEAP-PRUNE(H,r)* ارائه بدهید، که $min(r,n[H])$ گره را از H حذف می‌کند. این که کدام گره‌ها حذف شوند باید دلخواه باشد. زمان اجرای سرشکن شده پیاده‌سازی خود را تحلیل کنید. (راهنمایی: ممکن است نیاز به تغییر ساختمان داده و تابع پتانسیل داشته باشید.)

فصل ۲۱ ساختمان داده‌ها برای مجموعه‌های جدا از هم[1]

برخی از کاربردها شامل گروه‌بندی n عنصر مجزا به شکل تعدادی مجموعه جدا از هم می‌باشند. بنابراین دو عمل مهم عبارتند از یافتن اینکه عنصر داده شده عضو کدام مجموعه است و واحد سازی دو مجموعه. این فصل روش‌هایی را برای نگهداری ساختمان داده‌ای که این اعمال را پشتیبانی می‌کند بررسی می‌کند.

بخش ۲۱.۱ اعمالی که به وسیله مجموعه‌های جدا از هم پیشتیبانی می‌شوند را بیان می‌کند و کاربردی آسان را ارائه می‌دهد. در بخش ۲۱.۲ به پیاده سازی ساده لیست پیوندی برای مجموعه‌های جدا از هم نگاهی خواهیم داشت. نمایش موثرتری با استفاده از درخت‌های مشتق شده در بخش ۲۱.۳ ارائه می‌شود. برای تمام کاربردهای عملی، زمان اجرای استفاده از نمایش درخت‌ها خطی می‌باشد اما از نظر تئوری فوق خطی می‌باشد. بخش ۲۱.۴ تابعی با رشد بسیار سریع و رشد معکوس بسیار کم، که در زمان اجرای اعمال پیاده سازی بر پایه درخت‌ها ظهور می‌کند را تعریف و مورد بحث قرار می‌دهد و سپس از تحلیل سرشکن شده برای اثبات حد بالای زمان اجرا استفاده می‌کند که به زحمت فوق خطی می‌باشد.

۲۱.۱ اعمال روی مجموعه جدا از هم

ساختمان داده مجموعه جدا از هم، مجموعه $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ از مجموعه‌های پویای جدا از هم را نگه می‌دارد. هر مجموعه توسط **نماینده**[2] آن که عضوی از مجموعه است مشخص می‌شود. در برخی از کاربردها اینکه کدام عضو به عنوان نماینده استفاده شود اهمیتی ندارد؛ فقط باید توجه داشته باشیم که اگر نماینده مجموعه پویا را دو بار بدون تغییر دادن مجموعه در خواست کنیم در هر دو دفعه به یک جواب یکسان دست پیدا کنیم. در کاربردهای دیگر ممکن است قانون از پیش تعیین شده‌ای برای انتخاب نماینده وجود داشته باشد، مانند انتخاب کوچکترین عضو در مجموعه (البته فرض می‌کنیم که عناصر

---

1. *disjoint sets*

2. *representative*

بتوانند مرتب شوند).

مانند آنچه که در دیگر پیاده سازی‌های مجموعه‌های پویا بررسی کرده‌ایم هر عنصر از مجموعه با یک شی نمایش داده می‌شود. فرض می‌کنیم x به یک شیء اشاره می‌کند، می‌خواهیم اعمال زیر را پشتیبانی کنیم:

MAKE-SET(x) یک مجموعه جدید که x تنها عضو (و بنابراین نماینده) آن است را ایجاد می‌کند. از آنجا که مجموعه‌ها جدا از هم هستند لازم است که x عضو مجموعه دیگری نباشد.

UNION(x,y) دو مجموعه پویا شامل x و y که S_x و S_y گفته می‌شود را در یک مجموعه جدید که اجتماع دو مجموعه است قرار می‌دهد. قبل از این عمل فرض شده که دو مجموعه جدا از هم می‌باشند. نماینده مجموعه جدید می‌تواند هر یک از عضوهای $S_x \cup S_y$ باشد، اگر چه بسیاری از پیاده سازی‌های *UNION* به شکل خاصی یکی از نماینده‌های S_x یا S_y را به عنوان نماینده جدید انتخاب می‌کنند. از آنجایی که نیاز داریم مجموعه‌های گردآوری شده جدا از هم باشند مجموعه‌های S_x و S_y را با حذف از مجموعه $\mathcal{S}$ از بین می‌بریم.

FIND-SET(x) اشاره‌گری به نماینده مجموعه (منحصر به فرد) شامل x را بر می‌گرداند.

در طول این فصل زمان اجرای ساختمان داده‌های مجموعه جدا از هم را برحسب دو پارامتر تحلیل خواهیم کرد: n تعداد اعمال *MAKE-SET* و m تعداد کل اعمال *MAKE-SET*، *UNION* و *FIND-SET* از آنجایی که مجموعه‌ها جدا هستند هر عمل *UNION* تعداد مجموعه‌ها را یک واحد کاهش می‌دهد. بنابراین بعد از $n-1$ عمل *UNION* تنها یک مجموعه باقی می‌ماند لذا تعداد اعمال *UNION* حداکثر برابر با $n-1$ عمل است. همچنین توجه داشته باشید که چون اعمال *MAKE-SET* در کل m عمل وجود دارند لذا داریم $m \geq n$. فرض می‌کنیم که n عمل *MAKE-SET*، n عمل اولی است که اجرا می‌شود.

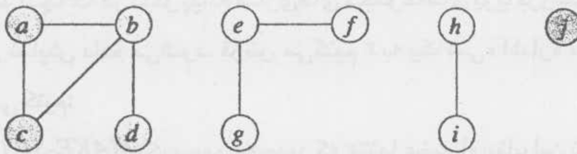
کاربردی از ساختمان داده مجموعه جدا از هم

یکی از مهمترین کاربردهای ساختمان داده مجموعه جدا از هم، از مشخص کردن اجزاء همبند یک گراف بدون جهت نشات می‌گیرد. شکل ۲۱.۱ *(a)* برای مثال گرافی با چهار جزء همبند را نشان می‌دهد.

روال *CONNECTED-COMPONENTS* که در ادامه آمده است از اعمال روی مجموعه جدا از هم برای محاسبه اجزاء همبند گراف استفاده می‌کند. زمانی که *CONNECTED-COMPONENTS* به عنوان یک گام پیش پرداز اجرا شده است، روال *SAME-COMPONENT* به پرس‌وجوها در این مورد که آیا دو رأس در یک جزء همبند هستند یا خیر پاسخ می‌دهد.[1] (مجموعه رئوس گراف G با *V[G]*

۱ـ وقتی یال‌های گراف ایستا هستند - بدون تغییر در طول زمان - اجزای همبند با استفاده از جستجوی اول عمق می‌توانند سریعتر محاسبه شوند. (تمرین ۱۱ - ۲۲.۳ را ملاحظه نمایید) اگر چه برخی اوقات یال‌ها "به‌طور پویا" اضافه می‌شوند و نیاز داریم همانطورکه هر یال اضافه می‌شود، اجزاء همبندی را نگهداری کنیم. در این حالت پیاده سازی داده شده در این جا می‌تواند از اجرای جستجوی اول عمق جدید برای هر یال جدید، موثرتر باشد.

و مجموعه یال‌ها با $E[G]$ نشان داده می‌شود.)



(a)

Edge processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

(b)

شکل ۲۱.۱ *(a)* یک گراف با چهار جزء همبند: $\{a,b,c,d\}$، $\{e,f,g\}$، $\{h,i\}$ و $\{j\}$. *(b)* مجموعه‌ای از مجموعه‌های جدا از هم بعد از اینکه هر یال پردازش می‌شود.

```
CONNECTED-COMPONENTS(G)
1  for each vertex v ∈ V[G]
2      do MAKE-SET(v)
3  for each edge (u, v) ∈ E[G]
4      do if FIND-SET(u) ≠ FIND-SET(v)
5            then UNION(u, v)
```

```
SAME-COMPONENT(u, v)
1  if FIND-SET(u) = FIND-SET(v)
2     then return TRUE
3     else  return FALSE
```

روال *CONNECTED-COMPONENTS* در آغاز هر رأس v را در مجموعه خودش قرار می‌دهد. سپس برای هر یال (u,v)، مجموعه‌هایی که شامل u و v هستند را تبدیل به یک واحد می‌کند. بنا به تمرین ۲ - ۲۱.۱ بعد از این که تمام یال‌ها پردازش می‌شوند، دو رأس در یک جزء همبند قرار دارند اگر و فقط اگر اشیاء متناظرشان در یک مجموعه یکسان باشند. بنابراین *CONNECTED-COMPONENTS* مجموعه‌ها را به طریقی محاسبه می‌کند که روال *SAME-COMPONENT* می‌تواند مشخص کند آیا

دو رأس در یک جزء همبند قرار دارند یا خیر. شکل (b)۲۱.۱ نشان می‌دهد که چگونه مجموعه‌های جدا از هم بوسیله *CONNECTED-COMPONENTS* محاسبه می‌شوند.

در پیاده سازی واقعی این الگوریتم اجزای همبند، نمایش گراف و ساختمان داده مجموعه جدا از هم نیاز به ارجاع به یکدیگر دارند. به عبارت دیگر یکی شیء که رأسی را نشان می‌دهد شامل اشاره‌گری به شیء متناظرش در مجموعه جدا از هم می‌باشد و برعکس. جزئیات برنامه نویسی به زبان پیاده سازی بستگی دارد و بیش از این به آن نمی‌پردازیم.

تمرین‌ها

۱ - ۲۱.۱ فرض کنید *CONNECTED-COMPONENTS* بر روی گراف بدون جهت $G = (V,E)$ اجرا شود که $V = \{a,b,c,d,e,f,g,h,i,j,k\}$ و یال‌های E به ترتیب زیر پردازش می‌شوند: (d,i)، (f,k)، (g,i)، (b,g)، (a,b)، (i,j)، (d,k)، (b,j)، (d,f)، (g,j)، (a,e)، (i,d). رأس‌ها در هر جزء همبند بعد از هر تکرار خطوط ۵ - ۳ را لیست کنید.

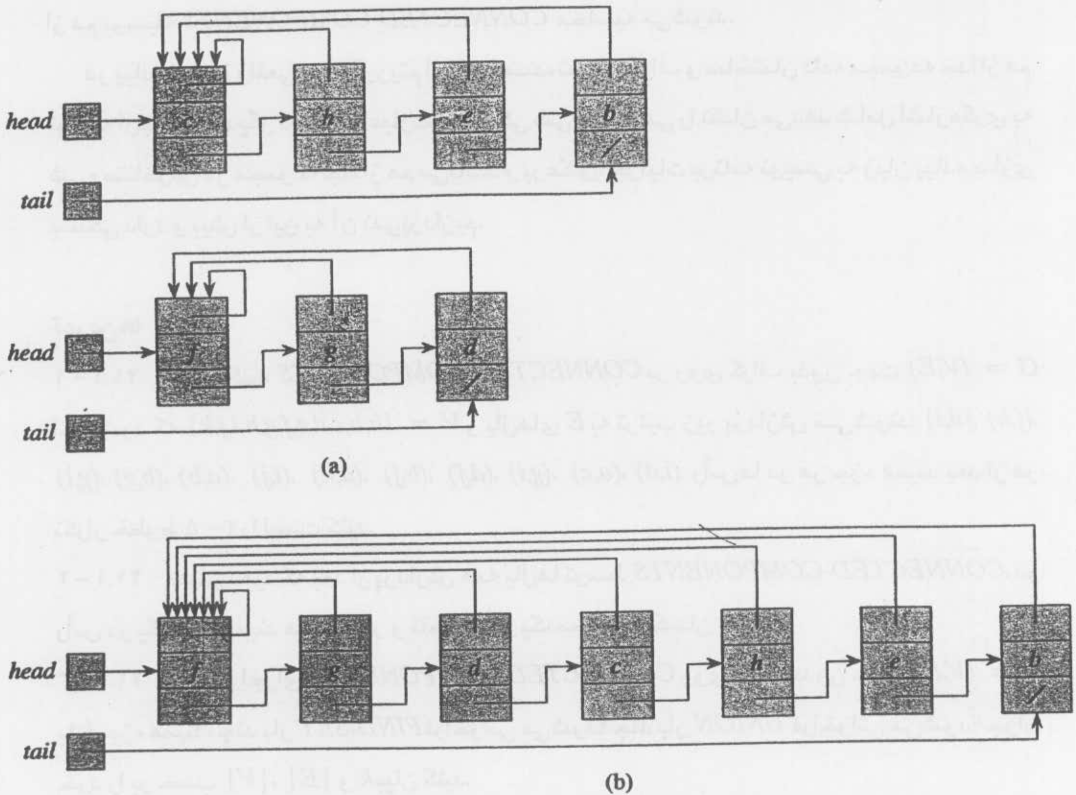
۲ - ۲۱.۱ نشان دهید که بعد از پردازش همه یال‌ها توسط *CONNECTED-COMPONENTS*، دو رأس در یک جزء همبند هستند اگر و فقط اگر در یک مجموعه یکسان باشند.

۳ - ۲۱.۱ در طی اجرای *CONNECTED-COMPONENTS* روی گراف بدون جهت $G = (V,E)$ با k جزء همبند، چند بار *FIND-SET* فراخوانی می‌شود؟ چند بار *UNION* فراخوانی می‌شود؟ جواب خود را بر حسب $|V|$، $|E|$ و k بیان کنید.

۲۱.۲ نمایش لیست پیوندی مجموعه‌های جدا از هم

یک راه ساده برای پیاده سازی ساختمان داده مجموعه جدا از هم، نمایش هر مجموعه با یک لیست پیوندی می‌باشد. اولین شیء در هر لیست پیوندی به عنوان نماینده مجموعه به کار می‌رود. هر شیء در لیست پیوندی شامل یک عضو مجموعه، اشاره‌گری به شیء شامل عضو بعدی مجموعه، و یک اشاره‌گر به نماینده مجموعه می‌باشد. هر لیست، اشاره‌گر *head* به نماینده و *tail* به شیء آخر را نگهداری می‌کند. شکل (a) ۲۱.۲ دو مجموعه را نشان می‌دهد. در داخل هر لیست پیوندی، اشیاء ممکن است با هر ترتیبی ظاهر شوند (با توجه به این فرض که اولین شیء در هر لیست پیوندی نماینده مجموعه است).

با این نمایش لیست پیوندی، هر دو عمل *MAKE-SET* و *FIND-SET* ساده هستند و زمان $O(1)$ را نیاز دارند. برای انجام *MAKE-SET(x)*، یک لیست پیوندی جدید که فقط شامل شیء x است را به وجود می‌آوریم. برای *FIND-SET(x)* فقط اشاره گری که از x به نماینده اشاره می‌کند را بر می‌گردانیم.



شکل ۲۱.۲ *(a)* نمایش لیست پیوندی دو مجموعه. یکی شامل اشیاء *b*، *c*، *e* و *h* که *c* نماینده است و دیگری شامل اشیاء *d*، *f* و *g* که *f* نماینده است. هر شیء در لیست شامل یک عضو مجموعه، اشاره گری به شیء شامل عضو بعدی مجموعه و یک اشاره گر به اولین شیء یعنی نماینده مجموعه می‌باشد. هر لیست اشاره گرهای *head* و *tail* که به ترتیب به اولین شیء و آخرین شیء اشاره می‌کنند را دارا می‌باشد. *(b)* نتیجه *UNION(e,g)*. نماینده مجموعه حاصل *f* است.

یک پیاده سازی ساده از واحد سازی

ساده‌ترین پیاده سازی عمل *UNION* با استفاده از نمایش مجموعه لیست پیوندی، زمان بیشتری را نسبت به *MAKE-SET* یا *FIND-SET* صرف می‌کند. همان‌طور که در شکل ۲۱.۲ *(b)* نشان داده شده است عمل *UNION(x,y)* را با ضمیمه کردن لیست *x* به انتهای لیست *y* اجرا می‌کنیم. از اشاره گر *tail* لیست *y* برای پیدا کردن سریع محل ضمیمه شدن لیست *x* استفاده می‌کنیم. نماینده مجموعه جدید، عنصری است که در ابتدا نماینده مجموعه شامل *y* بوده است. متأسفانه برای هر شیء که در ابتدا در *x* بوده است اشاره گر به نماینده را بروز رسانی می‌کنیم که زمان خطی طول لیست *x* را لازم دارد.

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
⋮	⋮
UNION(x_{n-1}, x_n)	$n-1$

شکل ۲۱.۳ یک توالی از *2n-1* عمل بر روی *n* شیء با استفاده از نمایش لیست پیوندی و پیاده سازی ساده *UNION* زمان $\Theta(n^2)$ یا به‌طور میانگین زمان $\Theta(n)$ را برای هر عمل صرف می‌کند.

در حقیقت انجام یک توالی از *m* عمل بر روی *n* شیء که نیاز به زمان $\Theta(n^2)$ دارد مشکل نیست. فرض کنید اشیاء $x_1, x_2, \ldots, x_n$ را داریم. یک توالی از *n* عمل *MAKE-SET* که به همراه *n-1* عمل *UNION* اجرا می‌کنیم در شکل ۲۱.۳ نشان داده شده است، بنابراین $m = 2n-1$ می‌باشد. زمان $\Theta(n)$ را برای انجام *n* عمل *MAKE-SET* صرف می‌کنیم از آنجایی که *i* امین عمل *UNION*، *i* شیء را به روزرسانی می‌کند، تعداد کل اشیائی که به وسیله همه *n-1* عمل *UNION* بروز رسانی می‌شوند برابر است با

$$\sum_{i=1}^{n-1} i = \Theta(n^2) .$$

تعداد کل اعمال برابر *2n-1* است و بنابراین هر عمل به‌طور میانگین زمان $\Theta(n)$ را نیاز دارد. به عبارت دیگر زمان سرشکن شده یک عمل برابر $\Theta(n)$ است.

روش مکاشفه‌ای واحد وزن دار[1]

در بدترین حالت، پیاده سازی بالا از روال *UNION* نیاز به زمان میانگین $\Theta(n)$ در هر فراخوانی دارد، زیرا ممکن است لیست بزرگتر را به انتهای لیست کوچکتر ضمیمه کنیم؛ بنابراین باید برای هر عضو لیست طولانی‌تر، اشاره‌گر به نماینده را بروز رسانی کنیم. در عوض فرض می‌کنیم هر لیست شامل طول لیست نیز باشد (که به راحتی قابل نگهداری است) و آنکه همیشه لیست کوچکتر را به انتهای لیست بزرگتر ضمیمه کنیم، با شکستن اتصالات بصورت دلخواه. با این روش **مکاشفه‌ای واحد وزن‌دار** ساده، اگر هر دو مجموعه دارای $\Omega(n)$ عضو باشند عمل *UNION* همچنان در زمان $\Omega(n)$ انجام

1. *weighted - union heuristic*

می‌شود، همانطور که قضیه زیر نشان می‌دهد یک توالی از m اعمال *MAKE-SET* ،*UNION* و *FIND-SET* که n عمل از آنها *MAKE-SET* است زمان $O(m+nlgn)$ را صرف می‌کند.

قضیه ۲۱.۱

با استفاده از نمایش لیست پیوندی مجموعه‌های جدا از هم و روش مکاشفه‌ای واحد وزن دار، یک توالی از m عمل *MAKE-SET* ،*UNION* و *FIND-SET* که n عمل از آنها *MAKE-SET* است، زمان $O(m+nlgn)$ را صرف می‌کند.

اثبات برای هر شیء در مجموعه با اندازه n با محاسبه یک حد بالا بر روی تعداد دفعاتی که اشاره‌گر از شیء به نماینده بروز رسانی می‌شود، شروع می‌کنیم. شیء ثابت x را در نظر بگیرید. می‌دانیم در هر بار که اشاره گر نماینده x به روز رسانی شد، x باید از مجموعه‌های کوچکتر شروع می‌شد. بنابراین اولین بار که اشاره‌گر به نماینده x بروزرسانی شده مجموعه حاصل حداقل باید 2 عضو می‌داشت. به طور مشابه بار بعد که اشاره‌گر نماینده x بروز رسانی شد مجموعه حاصل باید حداقل 4 عضو می‌داشت. به همین ترتیب مشاهده می‌شود برای هر $k \leq n$ بعد از اینکه اشاره گر به نماینده x تعداد $\lceil lgk \rceil$ بار بروز رسانی شده است، مجموعه حاصل باید حداقل k عضو داشته باشد. از آنجایی که بزرگترین مجموعه دارای حداکثر n عضو می‌باشد، هر اشاره‌گر به نماینده شیء در بیشترین حالت $\lceil lgn \rceil$ بار در طی مدت عمل *UNION* بروز رسانی شده است. همچنین باید بروز رسانی اشاره‌گرهای *head* و *tail* و طول لیست را نیز به حساب آوریم، که تنها زمان $\Theta(1)$ را برای هر عمل *UNION* صرف می‌کنند. بنابراین کل زمانی که برای بروز رسانی n شیء استفاده می‌شود برابر با $O(nlgn)$ است.

زمان لازم برای کل توالی از m عمل به آسانی محاسبه می‌شود. هر عمل *MAKE-SET* و *FIND-SET* زمان $O(1)$ را صرف می‌کند و $O(m)$ تا از این اعمال وجود دارد بنابراین زمان کل برای این توالی $O(m+nlgn)$ است.

تمرین‌ها

۲۱.۲ – ۱ شبه کدهایی برای *MAKE-SET* ،*FIND-SET* و *UNION* با استفاده از نمایش لیست پیوندی و روش مکاشفه‌ای واحد وزن دار بنویسید. فرض کنید که هر شیء x یک خاصیت $rep[x]$ دارد که به نماینده مجموعه‌ای که شامل x است اشاره می‌کند و هر مجموعه S دارای خواص $head[S]$، $tail[S]$ و $size[S]$ (که برابر با طول لیست است) می‌باشد.

۲۱.۲ – ۲ ساختمان داده‌ای که نتیجه می‌دهد و جواب‌هایی که به وسیله عمل *FIND-SET* در برنامه زیر برگردانده می‌شود را نشان دهید. از نمایش لیست پیوندی به همراه روش مکاشفه‌ای واحد وزن دار استفاده کنید.

```
1  for i ← 1 to 16
2       do MAKE-SET(x_i)
3  for i ← 1 to 15 by 2
4       do UNION(x_i, x_{i+1})
5  for i ← 1 to 13 by 4
6       do UNION(x_i, x_{i+2})
7  UNION(x_1, x_5)
8  UNION(x_11, x_13)
9  UNION(x_1, x_10)
10 FIND-SET(x_2)
11 FIND-SET(x_9)
```

فرض کنید اگر مجموعه‌های شامل x_i و x_j دارای اندازهٔ یکسانی باشند، آنگاه عمل $UNION(x_i,x_j)$ لیست x_j را به لیست x_i ضمیمه می‌کند.

۲۱.۲-۳ اثبات جمعی قضیه ۲۱.۱ را برای بدست آوردن حدهای زمانی سرشکن شده $O(1)$ برای *MAKE-SET* و *FIND-SET* و $O(lgn)$ برای *UNION* با استفاده از نمایش لیست پیوندی و روش مکاشفه‌ای واحد وزن دار، وفق دهید.

۲۱.۲-۴ یک حد مجانبی برای زمان اجرای یک توالی از اعمال در شکل ۲۱.۳ با فرض نمایش لیست پیوندی و روش مکاشفه‌ای واحد وزن دار ارائه دهید.

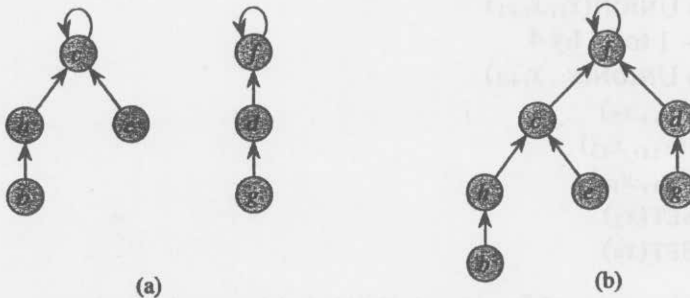
۲۱.۲-۵ تغییر ساده‌ای در روال *UNION* برای نمایش لیست پیوندی پیشنهاد کنید که نیاز به نگهداری اشاره‌گر *tail* به شیء آخر در هر لیست را از بین ببرد. چه از روش مکاشفه‌ای واحد وزن دار استفاده شده یا نشده باشد، تغییرات شما نباید زمان اجرای مجانبی روال *UNION* را تغییر دهید. (راهنمایی: به جای ضمیمه کردن یک لیست به دیگری آنها را به هم بچسبانید.)

۲۱.۳ جنگل‌های مجموعه جدا از هم

در یک پیاده سازی سریعتر مجموعه‌های جدا از هم، مجموعه‌ها را با درخت‌های مشتق شده نشان می‌دهیم که هر گره شامل یک عضو و هر درخت یک مجموعه را نشان می‌دهد. در یک **جنگل مجموعه جدا از هم**[1] که در شکل ۲۱.۴ *(a)* نشان داده شده است، هر عضو تنها به پدر خودش اشاره می‌کند. ریشه هر درخت شامل نماینده است و پدر خودش می‌باشد. همانطور که خواهیم دید اگر چه الگوریتم‌های قابل فهم که از این نمایش استفاده می‌کنند سریعتر از آنهایی که از نمایش لیست پیوندی استفاده می‌کنند، نمی‌باشند، با معرفی دو روش مکاشفه‌ای "واحد سازی بر حسب مرتبه"[2] و "فشرده‌سازی مسیر"[3] می‌توانیم به سریعترین ساختمان داده شناخته شده مجموعه جدا از هم دست

1. disjoint set forest
2. union by rank
3. path compression

یابیم.



شکل ۲۱.۴ یک جنگل مجموعه جدا از هم *(a)* دو درخت که دو مجموعه شکل ۲۱.۲ را نشان می‌دهند. درخت سمت چپ، مجموعه $\{b,c,e,h\}$ با c به عنوان نماینده و درخت سمت راست، مجموعه $\{d,f,g\}$ با f به عنوان نماینده، را نشان می‌دهند. *(b)* نتیجه *UNION(e,g)*

سه عمل مجموعه جدا از هم را به شکل زیر انجام می‌دهیم. عمل *MAKE-SET* به سادگی درختی با تنها یک گره را ایجاد می‌کند. عمل *FIND-SET* را با دنبال کردن اشاره گرهای پدر تا زمانی که ریشه درخت را پیدا کنیم، اجرا می‌کنیم. گره‌های ملاقات شده در این مسیر به سوی ریشه مسیر یافتن [1] را تشکیل می‌دهند. عمل *UNION* که در شکل ۲۱.۴*(b)* نشان داده شده است باعث می‌شود ریشه یک درخت به ریشه درخت دیگر اشاره کند.

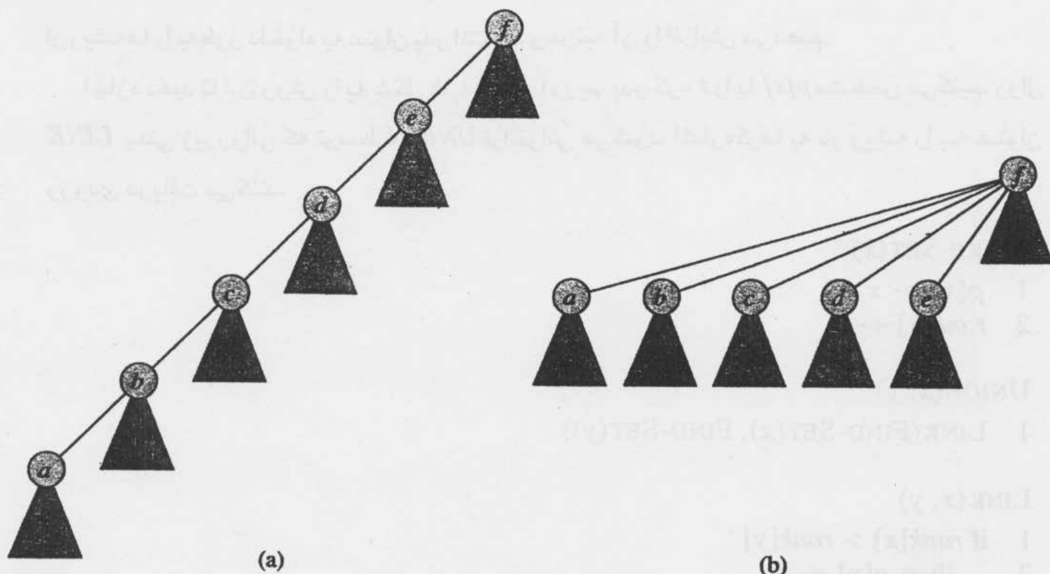
روش‌های مکاشفه‌ای برای بهبود بخشیدن زمان اجرا

تا حالا، پیاده سازی لیست پیوندی را بهبود نداده‌ایم. یک توالی از *n-1* عمل *UNION* ممکن است یک درخت که دقیقاً یک زنجیره خطی از *n* گره است را ایجاد کند. اما با استفاده از دو روش مکاشفه‌ای می‌توانیم به زمان اجرایی که تقریباً بر حسب تعداد کل اعمال یعنی *m* خطی است دست پیدا کنیم.

اولین روش مکاشفه‌ای، **واحدسازی برحسب مرتبه** مشابه روش مکاشفه‌ای واحد وزن‌دار که در نمایش لیست پیوندی استفاده می‌کردیم می‌باشد. ایده این است که ریشه درخت با گره کمتر به ریشه درخت با گره بیشتر اشاره نماید. به جای اینکه صریحاً اندازه زیردرخت مشتق شده در هر گره را دنبال کنیم از یک روش که تحلیل را آسان می‌کند استفاده خواهیم کرد. برای هر گره **مرتبه** [2] که حد بالا روی ارتفاع گره است را نگه می‌داریم. در واحد سازی بر حسب مرتبه در طی عمل *UNION* ، ریشه با مرتبه کوچکتر به ریشه با مرتبه بزرگتر اشاره خواهد کرد.

---

1. find path
2. rank



شکل ۲۱.۵ فشرده سازی مسیر در طی عمل *FIND-SET*. پیکان‌ها و خود حلقه در ریشه‌ها حذف می‌شوند. *(a)* درختی که یک مجموعه را قبل از اجرای *FIND-SET (a)* نمایش می‌دهد. مثلث‌ها زیردرخت‌هایی که ریشه‌های آنها نشان داده شده‌اند را نمایش می‌دهند. هر گره یک اشاره گر به پدرش دارد. *(b)* همان مجموعه بعد از اجرای *FIND-SET (a)* اکنون هر گره در مسیر یافتن به‌طور مستقیم به ریشه اشاره می‌کند.

دومین روش مکاشفه‌ای یعنی فشرده سازی مسیر نیز بسیار ساده و مؤثر است. همان‌طور که در شکل ۲۱.۵ نشان داده شده است در طی عمل *FIND-SET* از روش فشرده سازی مسیر استفاده می‌کنیم تا هر گره در مسیر یافتن به‌طور مستقیم با ریشه اشاره کند. روش فشرده سازی مسیر هیچ مرتبه‌ای را عوض نمی‌کند.

شبه کد برای جنگل‌های مجموعه‌های جدا از هم

برای پیاده سازی جنگل مجموعه جدا از هم با روش مکاشفه‌ای واحد سازی بر حسب مرتبه، باید مرتبه‌ها را دنبال کنیم. با هر گره x مقدار صحیح *rank[x]* که یک حد بالا بر روی ارتفاع x است را نگه می‌داریم (تعداد یال در طولانی‌ترین مسیر بین x و برگ نتیجه). وقتی یک مجموعه منحصر به فرد به وسیله *MAKE-SET* ایجاد می‌شود مرتبه اولیه تک گره در درخت مذکور برابر صفر است. هر عمل *FIND-SET* همه مرتبه‌ها را بدون تغییر باقی می‌گذارد. وقتی که *UNION* را برای دو درخت به کار می‌بریم، بسته به این که آیا ریشه‌های درخت‌ها دارای مرتبه مساوی هستند دو حالت وجود دارد. اگر ریشه‌ها دارای مرتبه نامساوی باشند، ریشه با مرتبه بالاتر را پدر ریشه با مرتبه پایین‌تر قرار می‌دهیم، اما مرتبه‌ها خودشان بدون تغییر باقی می‌مانند. اگر در عوض ریشه‌ها دارای مرتبه مساوی باشند یکی

از ریشه‌ها را به‌طور دلخواه به عنوان پدر انتخاب و مرتبه آن را افزایش می‌دهیم.

اجازه دهید تا این روش را به شکل شبه کد در آوریم. پدر گره *x* را با *p[x]* مشخص می‌کنیم. روال *LINK* یعنی زیرروالی که توسط *UNION* فراخوانی می‌شود، اشاره‌گرها به دو ریشه را به عنوان ورودی دریافت می‌کنند.

```
MAKE-SET(x)
1  p[x] ← x
2  rank[x] ← 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if rank[x] > rank[y]
2     then p[y] ← x
3     else p[x] ← y
4          if rank[x] = rank[y]
5             then rank[y] ← rank[y] + 1
```

روال *FIND-SET* با فشرده سازی مسیر بسیار ساده است.

```
FIND-SET(x)
1  if x ≠ p[x]
2     then p[x] ← FIND-SET(p[x])
3  return p[x]
```

روال *FIND-SET* یک **روش دو گذره**[1] است: روال *FIND-SET* در یک مرحله مسیر یافتن را به سمت بالا پیمایش می‌کند تا ریشه را پیدا کند و در دومین گذر یعنی برگشت به پایین در راستای مسیر یافتن، هر گره را برای اینکه به‌طور مستقیم به ریشه اشاره کند بروز رسانی می‌کند. هر فراخوانی *FIND-SET(x)* مقدار *p[x]* را در خط ۳ بر می‌گرداند. اگر *x* ریشه باشد آنگاه خط ۲ اجرا نمی‌شود و $p[x] = x$ بر گردانده می‌شود. این حالتی است که بازگشت به پایین‌ترین سطح خود می‌رسد. در غیر این صورت خط ۲ اجرا می‌گردد و فراخوانی بازگشتی با پارامتر *p[x]* اشاره‌گری به ریشه را بر می‌گرداند. خط ۲ گره *x* را به روز رسانی می‌کند تا به ریشه به‌طور مستقیم اشاره کند و این اشاره گر در خط ۳ برگردانده می‌شود.

---

1. *two-pass method*

تأثیر روش مکاشفه‌ای بر زمان اجرا

واحد سازی بر حسب مرتبه و فشرده سازی مسیر به‌طور جداگانه زمان اجرای اعمال بر روی جنگل‌های مجموعه جدا از هم را بهبود می‌بخشند و زمانی که دو روش مکاشفه‌ای با هم استفاده شوند این بهبود زمانی افزایش می‌یابد. تنها واحد سازی بر حسب مرتبه زمان اجرای $O(m\lg n)$ را حاصل می‌کند (تمرین ۴ - ۲۱.۴ را ملاحظه نمایید)، این حد قوی است (تمرین ۳ - ۲۱.۳ را ملاحظه نمایید). اگر چه در این جا آن را اثبات نخواهیم کرد، اگر n عمل *MAKE-SET* (از این رو حداکثر $n-1$ عمل *UNION* وجود دارد) و f عمل *FIND-SET* وجود داشته باشد روش مکاشفه‌ای فشرده سازی مسیر به تنهایی در بدترین حالت زمان اجرای $\Theta(n+ f.(1+\log_{2+f/n} n))$ را نتیجه می‌دهد.

وقتی از هر دو روش واحد سازی برحسب مرتبه و فشرده سازی مسیر استفاده کنیم بدترین حالت زمان اجرا $O(m\alpha(n))$ است که $\alpha(n)$ تابعی با رشد بسیار کم می‌باشد که در بخش ۲۱.۴ آن تعریف خواهیم کرد. در هر کاربرد قابل تصور از ساختمان داده مجموعه جدا از هم، $\alpha \leq 4$ است؛ بنابراین در تمامی وضعیت‌های عملی می‌توانیم زمان اجرا را به عنوان تابعی خطی از m در نظر بگیریم. در بخش ۲۱.۴ این حد بالا را اثبات می‌کنیم.

تمرین‌ها

۱ - ۲۱.۳ تمرین ۲ - ۲۱.۲ را با استفاده از جنگل مجموعه جدا از هم با واحد سازی بر حسب مرتبه و فشرده‌سازی مسیر انجام دهید.

۲ - ۲۱.۳ صورت غیر بازگشتی *FIND-SET* با فشرده سازی مسیر را بنویسید.

۳ - ۲۱.۳ یک توالی از m اعمال *MAKE-SET*، *UNION* و *FIND-SET* که n عمل آن *MAKE-SET* است ارائه دهید که وقتی تنها از واحد سازی بر حسب مرتبه استفاده می‌کنیم زمان $\Omega(m\lg n)$ را صرف کند.

۴ - ۲۱.۳ نشان دهید که توالی از m اعمال *MAKE-SET*، *FIND-SET* و *LINK* که همه اعمال *LINK* قبل از هر عمل *FIND-SET* ظاهر می‌شوند، اگر هر دو روش واحد سازی بر حسب مرتبه و فشرده‌سازی مسیر استفاده شوند تنها زمان $O(m)$ را صرف می‌کنند. اگر تنها روش مکاشفه‌ای فشرده‌سازی مسیر استفاده شود در همان وضعیت چه اتفاقی روی می‌دهد.

۲۱.۴* تحلیل واحد سازی بر حسب مرتبه به همراه فشرده‌سازی مسیر

همان‌طور که در بخش ۲۱.۳ ذکر شد زمان اجرای ترکیب روش مکاشفه‌ای واحد سازی بر حسب مرتبه و فشرده سازی مسیر برای m عمل مجموعه جدا از هم بر روی n عنصر $O(m\alpha(n))$ است. در این بخش تابعی α را بررسی می‌کنیم تا ببینیم به چه کندی رشد می‌کند. سپس این زمان اجرا را با استفاده از

روش پتانسیل تحلیل سرشکن شده اثبات می‌کنیم.

یک تابع با رشد سریع و معکوس آن با رشد بسیار آهسته

برای اعداد صحیح $k \geq 0$ و $j \geq 1$ تابع $A_k(j)$ را به شکل زیر تعریف می‌کنیم،

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0\,, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1\,, \end{cases}$$

که عبارت $A_{k-1}^{(j+1)}(j)$ از نماد تکرار تابعی ارائه شده در بخش ۳.۲ استفاده می‌کند. به‌طور خاص برای $i \geq 1$، $A_{k-1}^{(0)}(j) = j$ و $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$. به پارامتر k به عنوان سطح تابع A مراجعه خواهیم کرد.

تابع $A_k(j)$ اکیداً با هر دو پارامتر j و k افزایش می‌یابد. برای اینکه ببینید تابع چقدر سریع رشد می‌کند ابتدا عبارت‌های فرم بسته برای $A_1(j)$ و $A_2(j)$ را بدست می‌آوریم.

لم ۲۱.۲

برای هر عدد صحیح $j \geq 1$ داریم $A_1(j) = 2j + 1$.

اثبات ابتدا از استقرا بر روی i برای این که نشان دهیم $A_0^{(i)}(j) = j + i$، استفاده می‌کنیم. برای حالت پایه، $A_0^{(0)}(j) = j = j + 0$. برای گام استقرایی، فرض کنید که $A_0^{(i-1)}(j) = j + (i-1)$. آنگاه $A_0^{(i)}(j) = A_0(A_0^{(i-1)}(j)) = (j + (i-1)) + 1 = j + i$. در نهایت ملاحظه می‌کنیم که
$A_1(j) = A_0^{(j+1)}(j) = j + (j+1) = 2j + 1$. ■

لم ۲۱.۳

برای هر عدد صحیح $j \geq 1$ داریم $A_2(j) = 2^{j+1}(j+1) - 1$.

اثبات ابتدا از استقراء بر روی i برای اینکه نشان دهیم $A_1^{(i)}(j) = 2^i(j+1) - 1$ استفاده کرده و برای حالت پایه داریم $A_1^{(0)}(j) = j = 2^0(j+1) - 1$. برای گام استقرایی فرض می‌کنیم $A_1^{(i-1)}(j) = 2^{i-1}(j+1) - 1$، آنگاه

$$\begin{aligned} A_1^{(i)}(j) &= A_1(A_1^{(i-1)}(j)) = A_1(2^{i-1}(j+1) - 1) = 2\cdot(2^{i-1}(j+1)-1)+1 \\ &= 2^i(j+1)-2+1 = 2^i(j+1)-1. \end{aligned}$$

در نهایت ملاحظه می‌کنیم که

$$A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j+1) - 1.$$

■

اکنون با بررسی ساده $A_k(1)$ برای سطوح $k=0,1,2,3,4$ می‌توانیم ببینیم که $A_k(j)$ چقدر سریع رشد می‌کند. از تعریف $A_0(k)$ و لم‌های بالا داریم $A_0(1)=1+1=2$ ، $A_1(1)=2\times 1+1=3$ و همچنین داریم $A_2^{(1)} = 2^{1+1}(1+1)-1=7$

$$\begin{aligned} A_3(1) &= A_2^{(2)}(1) \\ &= A_2(A_2(1)) \\ &= A_2(7) \\ &= 2^8 \cdot 8 - 1 \\ &= 2^{11} - 1 \\ &= 2047 \end{aligned}$$

و

$$\begin{aligned} A_4(1) &= A_3^{(2)}(1) \\ &= A_3(A_3(1)) \\ &= A_3(2047) \\ &= A_2^{(2048)}(2047) \\ &\gg A_2(2047) \\ &= 2^{2048} \cdot 2048 - 1 \\ &> 2^{2048} \\ &= (2^4)^{512} \\ &= 16^{512} \\ &\gg 10^{80} , \end{aligned}$$

که برابر با تعداد کل اتم‌های تخمین زده شده در جهان است.

معکوس تابع $A_k(n)$ را برای عدد صحیح $n \geq 0$ به صورت زیر تعریف می‌کنیم

$$\alpha(n) = \min\{k : A_k(1) \geq n\} .$$

به‌طور خلاصه، $\alpha(n)$ کوچکترین سطح k است که برای آن $A_k(1)$ حداقل n است. از مقادیر $A_k(1)$ در فوق می‌بینیم که

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 , \\ 1 & \text{for } n = 3 , \\ 2 & \text{for } 4 \leq n \leq 7 , \\ 3 & \text{for } 8 \leq n \leq 2047 , \\ 4 & \text{for } 2048 \leq n \leq A_4(1) . \end{cases}$$

تنها برای مقادیر تقریباً بزرگ n (بزرگتر از $A_4(1)$ یک عدد بزرگ) $\alpha(n)>4$ می‌باشد و بنابراین برای همهٔ مقاصد عملی $\alpha(n) \leq 4$ است.

خواص مرتبه‌ها

در باقیمانده این بخش، حد $O(m\ \alpha(n))$ روی زمان اجرای اعمال مجموعه جدا از هم با واحد سازی برحسب مرتبه و فشرده سازی مسیر را اثبات می‌کنیم. به منظور اثبات این حد ابتدا برخی ویژگی‌های ساده مرتبه‌ها را اثبات می‌کنیم.

لم ۲۱.۴

برای همه گره‌های با نامساوی اکید $x \neq p[x]$ داریم $rank[x] \leq rank[p[x]]$. مقدار $rank[x]$ در ابتدا صفر است و تا زمانی که $x \neq p[x]$ شود افزایش می‌یابد؛ و از آن لحظه به بعد $rank[x]$ تغییر نمی‌کند. مقدار $rank[p[x]]$ به‌طور یکنواخت با زمان افزایش می‌یابد.

اثبات این اثبات یک استقراء ساده بر روی تعداد اعمالی که در پیاده سازی *MAKE-SET*، *UNION* و *FIND-SET* در بخش ۲۱.۳ ظاهر شده است، می‌باشد. این اثبات را به عنوان تمرین ۱ - ۲۱.۴ واگذار می‌کنیم. ■

قضیه فرعی ۲۱.۵

همان‌طور که مسیر از هر گره به سوی ریشه را دنبال می‌کنیم، مرتبه گره‌ها به‌طور اکید افزایش می‌یابد.

لم ۲۱.۶

هر گره دارای مرتبه حداکثر $n-1$ می‌باشد.

اثبات مرتبه هر گره از 0 شروع می‌شود و تنها در اعمال *LINK* افزایش می‌یابد. از آنجایی که حداکثر $n-1$ عمل *UNION* وجود دارد حداکثر $n-1$ عمل *LINK* نیز وجود دارد. چون هر عمل *LINK* یا همه مرتبه‌ها را بدون تغییر گذاشته و یا مرتبه یک گره را فقط یک واحد افزایش می‌دهد، همه مرتبه‌ها حداکثر $n-1$ می‌باشند. ■

لم ۲۱.۶ یک حد ضعیف بر روی مرتبه‌ها اعمال می‌کند. در واقع مرتبه هر گره حداکثر $\lfloor lgn \rfloor$ می‌باشد (تمرین ۲ - ۲۱.۴ را ملاحظه نمایید). اما حد ضعیف‌تر لم ۲۱.۶ برای هدف ما کافی خواهد بود.

اثبات حد زمانی

برای اثبات حد زمانی $O(m\alpha(n))$ از روش پتانسیل تحلیل سرشکن شده استفاده خواهیم کرد. (بخش ۱۷.۳ را ملاحظه نمایید). در انجام تحلیل سرشکن شده مناسب است فرض کنیم که عمل *LINK* را به جای عمل *UNION* استفاده می‌کنیم. به عبارت دیگر از آنجایی که پارامترهای روال *LINK* اشاره

گره‌هایی به دو ریشه هستند فرض می‌کنیم که اعمال *FIND-SET* مقتضی به‌طور جداگانه انجام می‌شوند. لم زیر نشان می‌دهد حتی اگر تعداد اعمال اضافی *FIND-SET* که بوسیله فراخوانی‌های *UNION* حاصل می‌گردند را محاسبه کنیم زمان اجرای مجانبی بدون تغییر باقی می‌ماند.

لم ۲۱.۷

فرض کنید یک توالی S' از m' اعمال *MAKE-SET*، *UNION* و *FIND-SET* را به یک توالی S از m عمل *MAKE-SET*، *LINK* و *FIND-SET* به وسیله تبدیل هر *UNION* به دو عمل *FIND-SET* به همراه یک *LINK* تبدیل کنیم. آنگاه اگر توالی S در زمان $O(m\alpha(n))$ اجرا شود توالی S' در زمان $O(m'\alpha(n))$ اجرا خواهد شد.

اثبات از آنجایی که هر عمل *UNION* در تولی S' به سه عمل در توالی S تبدیل می‌شود داریم $m' \leq m \leq 3m'$ چون $m = O(m')$ حد زمانی $O(m\alpha(n))$ برای توالی تبدیل شده S به حد زمانی $O(m'\alpha(n))$ برای تولی اولیه S' دلالت دارد. ■

در باقیمانده این بخش فرض خواهیم کرد توالی اولیه شامل m' عمل *MAKE-SET*، *UNION* و *FIND-SET* تبدیل به توالی m عملی *MAKE-SET*، *LINK* و *FIND-SET* شده است. اکنون ثابت می‌کنیم $O(m\alpha\ (n))$ یک حد زمانی برای توالی تبدیل شده می‌باشد و با مراجعه با لم ۲۱.۷ ثابت می‌کنیم که $O(m'\alpha(n))$ زمان اجرای توالی اصلی که شامل m' عمل است می‌باشد.

تابع پتانسیل

تابع پتانسیل که از آن استفاده می‌کنیم به هر گره x در جنگل مجموعه جدا از هم بعد از q عمل، مقدار پتانسیل $\phi_q(x)$ را نسبت می‌دهد. برای بدست آوردن پتانسیل کل جنگل، پتانسیل گره‌ها را با هم جمع می‌کنیم: $\Phi_q = \sum_x \phi(x)$ که Φ_q پتانسیل جنگل را بعد از q عمل نشان می‌دهد. قبل از اولین عمل جنگل خالی است و به‌طور دلخواه قرار می‌دهیم $\Phi_0=0$. پتانسیل Φ_q هیچگاه منفی نمی‌باشد.

مقدار $\phi_q(x)$ به اینکه آیا x ریشه درخت بعد از q امین عمل است یا خیر، بستگی دارد. اگر چنین باشد و یا اگر $rank[x]=0$ باشد، آنگاه $\phi_q(x)\ =\alpha(n).rank[x]$.

اکنون فرض کنید بعد از q امین عمل x ریشه نیست و $rank[x] \geq 1$ است. قبل از آن که بتوانیم $\phi_q(x)$ را تعریف کنیم به تعریف دو تابع کمکی بر وری x احتیاج داریم. ابتدا تعریف می‌کنیم

$$\text{level}(x) = \max\{k : rank[p[x]] \geq A_k(rank[x])\}\ .$$

به عبارت دیگر $level(x)$ بزرگترین سطح k است که برای آن A_k از مرتبه x استفاده کرده است، که بزرگتر از مرتبه پدر x نمی‌باشد.

ادعا می‌کنیم که

(۲۱.۱) $0 \le level(x) < \alpha(n)$

که آن را به صورت زیر مشاهده می‌کنیم‌داریم

$$\begin{aligned} rank[p[x]] &\ge rank[x]+1 && \text{(بنا به لم ۲۱.٤)} \\ &= A_0(rank[x]) && (\text{بنا به تعریف } A_0(j)) \end{aligned}$$

که دلالت دارد بر اینکه $level(x) \ge 0$ و داریم

$$\begin{aligned} A_{\alpha(n)}(rank[x]) &\ge A_{\alpha(n)}(1) && (\text{زیرا } A_k(j) \text{ اکیداً افزایش می‌یابد}) \\ &\ge n && (\text{بنا به تعریف } \alpha(n)) \\ &> rank[p[x]] && \text{(بنا به لم ۲۱.٦)} \end{aligned}$$

که دلالت دارد بر اینکه $level(x) < \alpha(n)$. توجه داشته باشید چون $rank[p[x]]$ در کل زمان به‌طور یکنواخت افزایش می‌یابد $level(x)$ نیز به همین شکل افزایش می‌یابد.

دومین تابع کمکی چنین است:

$$\text{iter}(x) = \max\left\{i : rank[p[x]] \ge A^{(i)}_{\text{level}(x)}(rank[x])\right\} .$$

به عبارت دیگر $iter(x)$ برابر با بیشترین تعداد دفعاتی است که می‌توانیم $A_{level(x)}$ را به‌طور تکراری به کار ببریم، که در ابتدا مرتبه x را به کار می‌برد، قبل از اینکه مقداری بزرگتر از مرتبه پدر x بدست آوریم. ادعا می‌کنیم که

(۲۱.۲) $1 \le iter(x) \le rank[x]$

که آن را به شکل زیر مشاهده می‌کنیم. داریم

$$\begin{aligned} rank[p[x]] &\ge A_{\text{level}(x)}(rank[x]) && (\text{بنا به تعریف } level(x)) \\ &= A^{(1)}_{\text{level}(x)}(rank[x]) && \text{(بنا به تعریف تکرار تابعی)} \end{aligned}$$

که دلالت می‌کند بر اینکه $iter(x) \ge 1$ و داریم

$$\begin{aligned} A^{(rank[x]+1)}_{\text{level}(x)}(rank[x]) &= A_{\text{level}(x)+1}(rank[x]) && (\text{بنا به تعریف } A_k(j)) \\ &> rank[p[x]] && (\text{بنا به تعریف } level(x)) \end{aligned}$$

که دلالت می‌کند بر این که $iter(x) \le rank[x]$. توجه داشته باشید که چون $rank[p[x]]$ در طول زمان به‌طور یکنواخت افزایش می‌یابد، برای آن که $iter(x)$ کاهش یابد، باید $level(x)$ افزایش پیدا کند. تا زمانی که $level(x)$ بدون تغییر می‌ماند، $iter(x)$ باید یا افزایش یابد و یا بدون تغییر بماند.

با این توابع کمکی در این جا آماده هستیم پتانسیل گره x بعد از q عمل را تعریف کنیم:

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot rank[x] & \text{اگر } x \text{ ریشه یا } rank[x] = 0 \text{ باشد،} \\ (\alpha(n) - \text{level}(x)) \cdot rank[x] - \text{iter}(x) & \text{اگر } x \text{ ریشه نیست و } rank[x] \geq 1 \text{.} \end{cases}$$

دو لم بعدی ویژگی‌های مفید پتانسیل گره‌ها را ارائه می‌دهند.

لم ۲۱.۸

برای هر گره x و برای تمام شماره‌های عمل یعنی q داریم

$$0 \leq \phi_q(x) \leq \alpha(n).rank[x]$$

اثبات اگر x ریشه و یا $rank[x]=0$ باشد آنگاه بنا به تعریف داریم $\phi_q(x)=\alpha(n).rank[x]$. اکنون فرض می‌کنیم که گره x ریشه نباشد و $rank[x]\geq 1$ باشد. حد پایین $\phi_q(x)$ را با ماکزیمم کردن $level(x)$ و $iter(x)$ بدست می‌آوریم. بنا به حد (۲۱.۱)، $level(x)\leq \alpha(n)-1$ و بنا به حد (۲۲.۲) داریم $iter(x) \leq rank[x]$ بنابراین

$$\begin{aligned} \phi_q(x) &\geq (\alpha(n) - (\alpha(n) - 1)) \cdot rank[x] - rank[x] \\ &= rank[x] - rank[x] \\ &= 0 . \end{aligned}$$

به‌طور مشابه یک حد بالا بر روی $\phi_q(x)$ به وسیله مینیمم کردن $level(x)$ و $iter(x)$ بدست می‌آوریم. بنا به حد (۲۱.۱)، $level(x) \geq 0$ و بنا به حد (۲۱.۲)، $iter(x) \geq 1$ بنابراین

$$\begin{aligned} \phi_q(x) &\leq (\alpha(n) - 0) \cdot rank[x] - 1 \\ &= \alpha(n) \cdot rank[x] - 1 \\ &< \alpha(n) \cdot rank[x] . \end{aligned}$$

■

تغییرات پتانسیل و هزینه‌های سرشکن شده اعمال

اکنون مهیا هستیم که بررسی کنیم چگونه اعمال مجموعه جدا از هم بر روی پتانسیل گره‌ها اثر می‌گذارد. با درک تغییر در پتانسیل به واسطه هر عمل، می‌توانیم مقدار هزینه سرشکن شده هر عمل را مشخص کنیم.

لم ۲۱.۹

فرض کنید x یک گره غیر ریشه باشد، و فرض کنید که qامین عمل $LINK$ یا $FIND\text{-}SET$ است. آنگاه بعد از qامین عمل، $\phi_q(x) \leq \phi_{q-1}(x)$ می‌باشد. علاوه بر این اگر $rank[x]\geq 1$ یا $level(x)$ و $iter(x)$ به واسطه qامین عمل تغییر کنند آنگاه $\phi_q(x)\leq \phi_{q-1}(x)-1$. به عبارت دیگر پتانسیل x نمی‌تواند افزایش یابد و اگر مرتبه مثبت داشته باشد و $level(x)$ یا $iter(x)$ تغییر یابند، آنگاه پتانسیل x حداقل یک واحد

کاهش می‌یابد.

اثبات از آنجایی که x ریشه نیست، qامین عمل $rank[x]$ را تغییر نمی‌دهد و چون n بعد از n عمل اولیه *MAKE-SET* تغییر نمی‌کند مقدار $\alpha(n)$ هم بدون تغییر می‌ماند. از این رو این اجزاء فرمول برای پتانسیل x بعد از qامین عمل یکسان باقی می‌مانند. اگر $rank[x]=0$ ، آنگاه $\phi_q(x) = \phi_{q-1}(x)=0$ ، اکنون فرض می‌کنید $rank[x] \geq 1$ است.

بیاد داشته باشید که $level(x)$ به‌طور یکنواخت با زمان افزایش می‌یابد اگر qامین عمل $level(x)$ را بدون تغییر باقی گذارد آنگاه $iter(x)$ افزایش می‌یابد یا بدون تغییر باقی می‌ماند. اگر $level(x)$ و $iter(x)$ هر دو بدون تغییر باشند آنگاه $\phi_q(x) = \phi_{q-1}(x)$ اگر $level(x)$ بدون تغییر بماند و $iter(x)$ افزایش یابد آنگاه پتانسیل گره x حداقل یک واحد افزایش می‌یابد و بنابراین

$$\phi_q(x) \leq \phi_{q-1}(x)-1$$

در نهایت اگر qامین عمل $level(x)$ را افزایش دهد، پتانسیل گره x حداقل یک واحد افزایش می‌یابد، بنابراین مقدار عبارت $(\alpha(n)-level(x))\ rank[x]$ حداقل به اندازه $rank[x]$ کاهش می‌یابد. از آنجایی که $level(x)$ افزایش می‌یابد مقدار $iter(x)$ ممکن است کاهش یابد، اما طبق حد (۲۱.۲) حداکثر کاهش $rank[x]-1$ است. بنابراین افزایش پتانسیل به واسطه تغییر $iter(x)$ کمتر از کاهش پتانسیل به واسطه تغییر $level(x)$ است. و نتیجه می‌گیریم که $\phi_q(x) \leq \phi_{q-1}(x) - 1$

■

سه لم آخر نشان می‌دهند که هزینه سرشکن شده هر یک از اعمال *MAKE-SET* ، *LINK* و *FIND-SET* برابر $O(\alpha(n))$ است. از معادله ۱۷.۲ به خاطر بیاورید که هزینه سرشکن شده هر عمل، مقدار هزینه واقعی آن به علاوه افزایش در پتانسیل به واسطه آن عمل می‌باشد.

لم ۲۱.۱۰

هزینه سرشکن شده هر عمل *MAKE-SET* برابر $O(1)$ است.

اثبات فرض کنید qامین عمل *MAKE-SET* است. این عمل گره x با مرتبه 0 را ایجاد می‌کند، بطوریکه $\phi_q(x)=0$، هیچ مرتبه یا پتانسیل دیگر تغییری نمی‌کند و بنابراین $\Phi_q = \Phi_{q-1}$. توجه به این که هزینه واقعی عمل *MAKE-SET* برابر $O(1)$ است اثبات را کامل می‌کند.

■

لم ۲۱.۱۱

هزینه سرشکن شده هر عمل *LINK* برابر $O(\alpha(n))$ است.

اثبات فرض کنید qامین عمل $LINK(x,y)$ است. هزینه واقعی عمل *LINK* برابر $O(1)$ است. بدون از

دست دادن کلیت فرض کنید عمل $LINK$، y را پدر x قرار می‌دهد.

برای مشخص کردن تغییرات در پتانسیل به واسطه عمل $LINK$، توجه می‌کنیم تنها گره‌هایی که ممکن است پتانسیلشان تغییر کند x، y و فرزندان y دقیقاً قبل از این عمل می‌باشند. نشان خواهیم داد که تنها گره‌ای که پتانسیل آن می‌تواند به واسطه عمل $LINK$ افزایش یابد y است و این که افزایش آن حداکثر برابر با $\alpha(n)$ است:

- بنا به لم ۲۱.۹ هر گره که قبل از $LINK$ فرزند y بوده است به واسطه عمل $LINK$ پتانسیلش افزایش نمی‌یابد.

- از تعریف $\phi_q(x)$ می‌بینیم از آنجایی که درست x قبل از qامین عمل ریشه بوده است $\phi_{q-1}(x)=\alpha(n).rank[x]$. اگر چه $rank[x]=0$ باشد آنگاه $\phi_q(x)=\phi_{q-1}(x)=0$. در غیر اینصورت:

$$\phi_q(x)=(\alpha(n)-level(x)).rank[x]-iter(x)$$
$$<\alpha(n).rank[x] \qquad \text{(بنا به نامساوی‌های ۲۱.۱ و ۲۱.۲)}$$

چون آخرین کمیت $\phi_{q-1}(x)$ است می‌بینیم که پتانسیل x کاهش می‌یابد.

- چون y قبل از $LINK$ ریشه است، $\phi_{q-1}(y)=\alpha(n).rank[y]$، عمل $LINK$، y را به عنوان ریشه و مرتبه y را بدون تغییر باقی می‌گذارد و یا آن را یک واحد افزایش می‌دهد. بنابراین $\phi_q(y)=\phi_{q-1}(y)$ ویا $\phi_q(y)=\phi_{q-1}(y)+\alpha(n)$.

بنابراین افزایش در پتانسیل به واسطه عمل $LINK$ حداکثر $\alpha(n)$ است. هزینه سرشکن شده عمل $LINK$ برابر است با $O(1)+\alpha(n)=O(\alpha(n))$. ■

لم ۲۱.۱۲

هزینه سرشکن شده هر عمل $FIND\text{-}SET$ برابر $O(\alpha(n))$ است.

اثبات فرض کنید qامین عمل $FIND\text{-}SET$ است و اینکه مسیر یافتن شامل s گره است. هزینه واقعی عمل $FIND\text{-}SET$ برابر $O(s)$ است. نشان می‌دهیم که پتانسیل هیچ گرهی به واسطه $FIND\text{-}SET$ افزایش نمی‌یابد و آنکه پتانسیل حداقل $max(0,s-(\alpha(n)+2))$ گره در مسیر یافتن، حداقل یک واحد کاهش می‌یابد.

برای درک اینکه پتانسیل هیچ گره‌ای افزایش نمی‌یابد ابتدا به لم ۲۱.۹ برای همه گره‌ها بجز ریشه متوسل می‌شویم. اگر x ریشه باشد آنگاه پتانسیل آن $\alpha(n).rank[x]$ است که تغییر نمی‌کند.

اکنون نشان می‌دهیم که حداقل $max(0,s-(\alpha(n)+2))$ گره وجود دارد که پتانسیل آنها دست کم یک واحد کاهش می‌یابد. فرض کنید x یک گره در مسیر یافتن باشد بطوریکه $rank[x]>0$ و x در مکانی روی مسیر یافتن با گره دیگر y که ریشه نیست دنبال شده است. درست قبل از عمل $FIND\text{-}SET$ داریم $level(y)=level(x)$ (لازم نیست گره y بلافاصله بعد از گره x در مسیر یافتن قرار داشته باشد.) همه اما

حداکثر $\alpha(n)+2$ گره در مسیر یافتن این محدودیت‌ها روی x را ارضاء می‌کنند. گره‌هایی که این محدودیت‌ها را ارضا نمی‌کنند این گره‌ها هستند: اولین گره روی مسیر یافتن (اگر مرتبه 0 داشته باشد) آخرین گره در مسیر (یعنی ریشه) و گره آخر w در مسیر که برای آن به ازای $k=0,1,2,...,\alpha(n)-1$ ، $level(w)=k$.

اجازه دهید چنین گره x را بدست آوریم، و نشان خواهیم داد که پتانسیل x حداقل یک واحد کاهش می‌یابد. قرار دهید $k=level(x)=level(y)$ درست قبل از آن که فشرده سازی مسیر توسط FIND-SET اعمال شود داریم

$$\begin{aligned} rank[p[x]] &\geq A_k^{\text{iter}(x)}(rank[x]) && \text{(بنا به تعریف } iter(x)) \\ rank[p[y]] &\geq A_k(rank[y]) && \text{(بنا به تعریف } level(y)) \\ rank[y] &\geq rank[p[x]] && \text{(بنا به قضیه فرعی ۲۱.۵ و چون } y \text{ به دنبال } x \text{ در مسیر یافتن می‌آید)} \end{aligned}$$

با تلفیق این نامساوی‌ها با هم با قرار دادن مقدار موجود در $iter(x)$ قبل از فشرده سازی مسیر در i داریم

$$\begin{aligned} rank[p[y]] &\geq A_k(rank[y]) && \text{(زیرا } A_k(j) \text{ اکیداً افزایش می‌یابد)} \\ &\geq A_k(rank[p[x]]) \\ &\geq A_k(A_k^{\text{iter}(x)}(rank[x])) \\ &= A_k^{i+1}(rank[x])) . \end{aligned}$$

چون فشرده سازی مسیر باعث می‌شود که x و y پدر یکسان داشته باشند، می‌دانیم که بعد از فشرده سازی مسیر، $rank[p[y]]=rank[p[x]]$ و آنکه فشرده سازی مسیر مقدار $rank[p[y]]$ را کاهش نمی‌دهد. از آنجایی که $rank[x]$ تغییر نمی‌کند بعد از فشرده سازی مسیر داریم $rank[p[x]]\geq A_k^{i+1}(rank(x))$

بنابراین فشرده سازی مسیر باعث می‌شود که $iter(x)$ (حداقل $i+1$ واحد) افزایش یابد یا $level(x)$ افزایش یابد (که در صورتی که $iter(x)$ حداقل $rank[x]+1$ واحد افزایش یابد رخ می‌دهد). در هر حالت بنا به لم ۲۱.۹ داریم $\phi_q(x)\leq \phi_{q-1}(x)-1$ از این رو پتانسیل x حداقل 1 واحد کاهش می‌یابد.

هزینه سرشکن شده عمل FIND-SET برابر با هزینه واقعی به علاوه تغییرات در پتانسیل است، هزینه واقعی $O(s)$ است و نشان داده‌ایم که پتانسیل کل به اندازه حداقل $max(0,s-(\alpha(n)+2))$ کاهش می‌یابد بنابراین هزینه سرشکن شده حداکثر برابر با

$$O(s)-(s-(\alpha(n)+2))=O(s)-s+O(\alpha(n))=O(\alpha(n))$$

چون می‌توانیم مقیاس واحدهای پتانسیل را بزرگ کنیم تا تأثیر مقدار ثابت پنهان در $O(s)$ را بی

اهمیت سازد. ■

قرار دادن لم‌های قبل در کنار هم منجر به قضیه زیر می‌شود.

قضیه ۲۱.۱۳

یک توالی از m اعمال *MAKE-SET*، *UNION* و *FIND-SET* که n عمل آن *MAKE-SET* است می‌تواند روی یک جنگل مجموعه جدا از هم با واحد سازی بر حسب مرتبه و فشرده سازی مسیر، در بدترین حالت زمانی $O(m\alpha(n))$ اجرا شود.

اثبات بلافاصله از لم‌های ۲۱.۷، ۲۱.۱۰، ۲۱.۱۱ و ۲۱.۱۲ ثابت می‌شود. ■

تمرین‌ها

۱ - ۲۱.۴ لم ۲۱.۴ را ثابت کنید.

۲ - ۲۱.۴ ثابت کنید که هر گره دارای مرتبه حداکثر $\lfloor lgn \rfloor$ است.

۳ - ۲۱.۴ با توجه به تمرین ۲ - ۲۱.۴ چند بیت برای ذخیره کردن $rank[x]$ برای هر گره x لازم است؟

۴ - ۲۱.۴ با استفاده از تمرین ۲ - ۲۱.۴، یک اثبات ساده از اینکه اعمال بر روی جنگل مجموعه جدا از هم با واحد سازی مرتبه اما بدون فشرده سازی مسیر، در زمان $O(mlgn)$ اجرا می‌شوند ارائه دهید.

۵ - ۲۱.۴ پرفسور *Dante* استدلال می‌کند که چون مرتبه‌های گره‌ها تنها در مسیر به سمت ریشه اکیداً افزایش می‌یابند. سطح گره‌ها باید به‌طور یکنواخت در طول مسیر افزایش یابد، به عبارت دیگر اگر $rank[x]>0$ و $p[x]$ ریشه نباشد آنگاه $level\ (p[x]) \geq level(x)$ آیا پرفسور درست می‌گوید؟

*۶ - ۲۱.۴ تابع $\alpha'(n)=min\{k:A_k(1)\geq lg(n+1)\}$ را در نظر بگیرید. نشان دهید که برای همه مقادیر عملی n $\alpha'(n)\leq 3$ است، و با استفاده از تمرین ۲ - ۲۱.۴ نشان دهید چگونه آرگومان تابع پتانسیل را تغییر دهیم که یک توالی از m اعمال *MAKE-SET*، *UNION* و *FIND-SET* که n عمل آن *MAKE-SET* است بتواند بر روی یک جنگل مجموعه جدا از هم با واحد سازی بر حسب مرتبه و فشرده سازی مسیر، در بدترین حالت زمانی $O(m\alpha'(n))$ اجرا شود.

مسائل

۱-۲۱ مینیمم برون خطی[1]

مسئله مینیمم برون خطی از ما می‌خواهد مجموعه پویای T از عناصر در دامنه $\{1,2,...,n\}$ را تحت

1. off - line minimum

اعمال *INSERT* و *EXTRACT* نگهداری کنیم. توالی S از n فراخوانی *INSERT* و m فراخوانی *EXTRACT-MIN* داده شده است که هر کلید در $\{1,2,...,n\}$ دقیقاً یک بار درج می‌شود. می‌خواهیم مشخص کنیم کدام کلید با فراخوانی *EXTRACT-MIN* برگردانده می‌شود. به ویژه می‌خواهیم آرایه *extracted[1.m]* را پر کنیم که برای $i=1,2,...,n$ *extracted[i]* کلیدی است که توسط iامین فراخوانی *EXTRACT-MIN* برگردانده می‌شود. مسئله از این جهت برون خطی است که به ما اجازه داده شده است که کل توالی S را قبل از مشخص کردن هر کلید برگردانده شده پردازش کنیم.

a. در نمونه زیر از مسئله مینیمم برون خطی، هر درج به وسیله یک عدد و هر *EXTRACT-MIN* به وسیله حرف E نشان داده شده است:

4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5.

مقادیر درست را در آرایه *extracted* قرار دهید.

جهت توسعه یک الگوریتم برای این مسئله، توالی S را به زیرتوالی‌های همگن می‌شکنیم. به عبارت دیگر S را به شکل زیر نمایش می‌دهیم

$I_1, E, I_2\ E\ , I_3\ ..., I_m, E, I_{m+1}$

که هر E یک فراخوانی *EXTRACT-MIN* و هر I_j یک توالی (ممکن است خالی باشد) از فراخوانی‌های *INSERT* را نشان می‌دهد. برای هر زیرتوالی I_j در آغاز، کلیدهای درج شده به وسیله این اعمال را در مجموعه K_j قرار می‌دهیم. در صورتی که I_j خالی باشد K_j خالی است. آنگاه عمل زیر را انجام می‌دهیم.

```
OFF-LINE-MINIMUM(m, n)
1  for i ← 1 to n
2       do determine j such that i ∈ K_j
3          if j ≠ m + 1
4             then extracted[j] ← i
5                  let l be the smallest value greater than j
                        for which set K_l exists
6                  K_l ← K_j ∪ K_l, destroying K_j
7  return extracted
```

b. ثابت کنید آرایه *extracted* برگردانده شده به وسیله *OFF-LINE-MINIMUM* درست است.

c. توضیح دهید چگونه با ساختمان داده مجموعه جدا از هم می‌توان *OFF-LINE-MINIMUM* را به‌طور مؤثر پیاده سازی کرد. یک حد اکید بر روی بدترین حالت زمان اجرایی پیاده سازیتان ارائه دهید.

۲-۲۱ تعیین عمق[1]

در مسئله تعیین عمق، جنگل $\mathcal{F}=\{T_i\}$ از درخت‌های مشتق شده را تحت سه عمل زیر نگهداری می‌کنیم:

MAKE-TREE(v) درخت که تنها گره آن *v* است را ایجاد می‌کند.

FIND-DEPTH(v) عمق گره *v* درون درخت حاوی *v* را بر می‌گرداند.

GRAFT(r,v) گره *r* را که فرض شده است ریشه باشد را فرزند گره *v* قرار می‌دهد، که فرض شده است *v* در درختی متفاوت از *r* قرار دارد اما ممکن است ریشه باشد یا نباشد.

a. فرض کنید از نمایش درختی شبیه جنگل مجموعه جدا از هم استفاده کنیم: *p[v]* پدر گره *v* است به جزء این که اگر *v* ریشه باشد *p[v]=v*، اگر *GRAFT(r,v)* را به وسیله قرار دادن *p[r]←v* پیاده‌سازی کنیم و *FIND-DEPTH* را به وسیله دنبال کردن مسیر یافتن به سمت ریشه و برگرداندن تعداد گره‌های ملاقات شده به جزء *v* پیاده سازی کنیم نشان دهید که بدترین حالت زمان اجرای یک توالی از *m* اعمال *MAKE-TREE*، *FIND-DEPTH* و *GRAFT* برابر $\Theta(m^2)$ است.

با استفاده از روش‌های مکاشفه‌ای واحد سازی بر حسب مرتبه و فشرده سازی مسیر می‌توانیم زمان اجرای بدترین حالت را کاهش دهیم. از جنگل مجموعه جدا از هم $\mathcal{S}=\{S_i\}$ استفاده می‌کنیم، که هر مجموعه S_i (که خودش یک درخت است) با یک درخت T_i در جنگل $\mathcal{F}$ متناظر است. اما ساختار درخت در داخل مجموعه S_i لزوماً مطابق با ساختار درختی در T_i نمی‌باشد. در واقع پیاده سازی S_i رابطه دقیق پدر – فرزندی را نگهداری نمی‌کند، اما با وجود این به ما اجازه‌می‌دهد تا عمق هر گره در T_i را مشخص کنیم.

ایدهٔ اصلی این است که در هر گره *v*، شبه فاصله *d[v]* را نگهداری کنیم که برای این تعریف شده است که ، مجموع شبه فاصله‌ها در طول مسیر از *v* تا ریشه مجموعه S_i آن برابر با عمق *v* در T_i باشد. به عبارت دیگر اگر مسیر از *v* تا ریشه‌اش در مجموعه S_i برابر $v_0,v_1,\ldots,v_k$ باشد که $v_0=v$ و v_k ریشه S_i است، آنگاه عمق *v* در T_i برابر است با

$$\sum_{j=0}^{k} d[v_j].$$

b. یک پیاده سازی از *MAKE-TREE* ارائه دهید.

c. نشان دهید چگونه جهت پیاده سازی *FIND-DEPTH* باید *FIND-SET* را تغییر دهیم. پیاده سازی‌تان باید فشرده سازی مسیر را اجرا کند و زمان اجرایش باید بر حسب طول مسیر یافتن خطی باشد. اطمینان حاصل کنید که پیاده سازی‌تان به‌طور صحیحی شبه فاصله‌ها را به روز رسانی می‌کند.

---

1. Depth determination

d. نشان دهید چگونه *GRAFT(r,v)* که مجموعه‌های شامل r و v را با تغییر روال‌های *UNION* و *LINK* ترکیب می‌کند پیاده سازی کنیم. اطمینان حاصل کنید که پیاده سازیتان به‌طور صحیحی شبه فاصله‌ها را به روز رسانی می‌کند. توجه کنید که ریشه مجموعه S_i لزوماً ریشه درخت T_i متناظرش نیست.

e. یک حد اکید بر روی بدترین حالت زمان اجرای یک توالی از m اعمال *DEPTH-FIND-MAKE-TREE* و *GRAFT* که n عمل آن *MAKE-TREE* است ارائه دهید.

۳-۲۱ الگوریتم برون خطی نزدیکترین جد مشترک *Tarjan*

نزدیکترین جد مشترک[1] دو گره u و v در درخت مشتق شده T گره w است که یک جد دو گره u و v است و بیشترین عمق در T را دارد. در **مسئله برون خطی نزدیکترین جد مشترک**[2]، درخت مشتق شده T و مجموعه دلخواه $P=\{(u,v)\}$ از جفت‌های گره‌های نامرتب در T داده شده است و می‌خواهیم نزدیکترین جد مشترک هر جفت در P را مشخص کنیم.

برای حل مسئله برون خطی کوچکترین جد مشترک، روال زیر پیمایش درخت T را با فراخوانی اولیه *LCA(root[T])* انجام می‌دهد. فرض شده هر گره قبل از پیمایش سفید رنگ شده است.

```
LCA(u)
 1  MAKE-SET(u)
 2  ancestor[FIND-SET(u)] ← u
 3  for each child v of u in T
 4       do LCA(v)
 5          UNION(u, v)
 6          ancestor[FIND-SET(u)] ← u
 7  color[u] ← BLACK
 8  for each node v such that {u, v} ∈ P
 9      do if color[v] = BLACK
10            then print "The least common ancestor of"
                         u "and" v "is" ancestor[FIND-SET(v)]
```

a. ثابت کنید که خط ۱۰، دقیقاً یک بار برای هر جفت $\{u,v\} \in P$ اجرا می‌شود.

b. ثابت کنید در زمان فراخوانی *LCA(u)* تعداد مجموعه‌ها در ساختمان داده مجموعه جدا از هم برابر با عمق u در T است.

c. اثبات کنید که *LCA* به درستی نزدیکترین جد مشترک u و v را برای هر جفت $(u,v) \in P$ چاپ می‌کند.

d. زمان اجرای *LCA* را تحلیل کنید با این فرض که از پیاده سازی ساختمان داده مجموعه جدا از هم بخش ۲۱.۳ استفاده می‌کنیم.

---

1. least-common-ancestor
2. off-line least-common-ancstor-problem

VI

الگوریتم‌های گراف

مقدمه

گراف‌ها ساختار داده سرایت کننده در علم کامپیوتر هستند، و الگوریتم‌های لازم برای کار کردن با آن‌ها اساس این زمینه هستند. صدها مسئله محاسباتی جالب در زمینه گراف تعریف شده است. در این قسمت تعداد کمی از آنها را بررسی می‌کنیم.

فصل ۲۲ نشان می‌دهد چگونه می‌توانیم یک گراف را در کامپیوتر نمایش دهیم و سپس الگوریتم‌هایی که بر پایهٔ جستجو یک گراف با استفاده جستجوی اول سطح یا اول عمق هستند را توضیح می‌دهد. دو کاربرد از جستجوی اول عمق ارائه می‌شود: مرتب سازی موضعی گراف جهت دار بدون دور و تجزیه گراف جهت دار به اجزای همبند قوی‌اش.

فصل ۲۳ توضیح می‌دهد چگونه درخت پوشا با وزن مینیمم یک گراف را محاسبه کنیم. چنین درختی به عنوان مسیر با وزن مینیمم که رأس‌ها را به هم متصل می‌کند تعریف می‌شود که هر یال یک وزن مربوطه دارد. الگوریتم‌ها برای محاسبه درخت پوشای مینیمم مثال‌های خوبی از الگوریتم‌های حریصانه (فصل ۱۶ را ملاحظه کنید) هستند.

فصل ۲۴ و ۲۵ مسئله محاسبه کوتاهترین مسیر بین رأس‌ها وقتی هر یال یک طول یا "**وزن**"[1] مرتبط دارد را بررسی می‌کند، فصل ۲۴ محاسبه کوتاهترین مسیرها از یک رأس مبدأ به همه رأس‌های دیگر و فصل ۲۵ محاسبه کوتاهترین مسیر بین هر جفت از رئوس را بررسی می‌کند.

در نهایت فصل ۲۶ نشان می‌دهد چگونه ماکزیمم جریان ماده در شبکه (گراف جهت‌دار) با داشتن منبع معین و حفره معین و ظرفیت معین برای مقدار ماده که می‌تواند در هر یال جهت‌دار عبور داده شود، محاسبه می‌گردد. این مسئله کلی در اشکال زیادی رخ می‌دهد و یک الگوریتم خوب برای محاسبه ماکزیمم جریان می‌تواند برای حل مؤثر انواع مسائل مربوطه استفاده شود.

در توصیف زمان اجرای الگوریتم گراف بر روی گراف داده شده $G=(V,E)$ معمولاً اندازه ورودی را برحسب تعداد رئوس $|V|$ و تعداد یال‌ها $|E|$ گراف اندازه‌گیری می‌کنیم. به عبارت دیگر دو پارامتر وجود دارند که مربوط به توصیف اندازهٔ ورودی هستند و نه یک پارامتر. یک قرارداد نماد گذاری معمول برای این پارامترها اتخاذ می‌کنیم. در نمادگذاری مجانبی (مانند نمادگذاری O یا Θ) و تنها در

---

1. weight

چنین نمادگذاری نماد V بر $|V|$ و نماد E بر $|E|$ دلالت می‌کند. برای مثال، ممکن است بگوییم الگوریتم در زمان $O(VE)$ اجرامی‌شود به این معنی که الگوریتم در زمان $O(|V| \ |E|)$ اجرا می‌شود. این قرارداد خواندن فرمول زمان اجرا را ساده‌تر می‌کند بدون اینکه ابهام داشته باشد.

قرارداد دیگری اتخاذ می‌کنیم که در شبه کد ظاهر می‌شود. مجموعه رئوس گراف را با $V[G]$ و مجموعه یال‌ها را با $E[G]$ نمایش می‌دهیم. به عبارت دیگر شبه کد به مجموعه رئوس و یال‌ها به عنوان خصوصیاتی از گراف نگاه می‌کند.

فصل ۲۲ الگوریتم‌های اولیه گراف

این فصل روش‌هایی را برای نمایش و جستجوی یک گراف ارائه می‌دهد. جستجوی گراف به معنی دنبال کردن سیستماتیک یال‌های گراف است تا همه رئوس گراف ملاقات شوند. یک الگوریتم جستجوی گراف بیشتر می‌تواند در مورد ساختار گراف اکتشاف کند. بسیاری از الگوریتم‌ها با جستجوی گراف ورودی برای بدست آوردن این اطلاعات ساختاری آغاز می‌شوند. دیگر الگوریتم‌های گراف به شکل ترکیب ساده‌ای از الگوریتم‌های جستجوی گراف سازماندهی می‌شوند. تکنیک‌های جستجوی یک گراف، قلب الگوریتم‌های گراف می‌باشند.

بخش ۲۲.۱ درباره دو نمایش محاسباتی رایج گراف‌ها بحث می‌کند: لیست همجواری و ماتریس همجواری. بخش ۲۲.۲ یک الگوریتم ساده جستجوی گراف که جستجوی اول سطح نامیده می‌شود را ارائه می‌دهد و نشان می‌دهد چطور درخت اول سطح را ایجاد کنیم. بخش ۲۲.۳ جستجوی اول عمق را ارائه می‌دهد و برخی نتایج استاندارد درباره ترتیبی که جستجوی اول عمق رأس‌ها را ملاقات می‌کند اثبات می‌کنیم. بخش ۲۲.۴ اولین کاربرد واقعی از جستجوی اول عمق را ارائه می‌کند: مرتب سازی موضعی گراف جهت‌دار بدون دور. دومین کاربرد از الگوریتم جستجوی اول عمق، پیدا کردن اجزاء همبند قوی گراف جهت‌دار است که در بخش ۲۲.۵ بیان شده است.

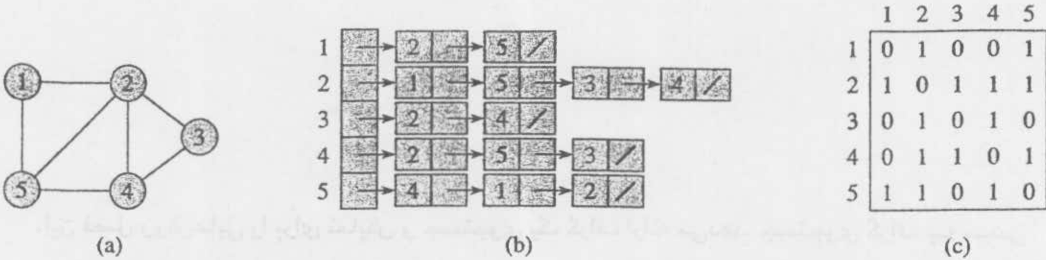
۲۲.۱ نمایش‌های گراف

دو راه استاندارد برای نمایش گراف $G=(V,E)$ وجود دارد: بصورت مجموعه‌ای از لیست‌های همجواری یا یک ماتریس همجواری. هر دو روش برای هر دو گراف جهت‌دار و بدون جهت قابل اعمال است. نمایش لیست همجواری معمولاً ترجیح داده می‌شود زیرا یک راه فشرده برای نمایش گراف‌های پراکنده (خلوت)[1] است گرافهایی که برای آنها $|E|$ بسیار کمتر از $|V|^2$ می‌باشد. بیشتر الگوریتم‌های این کتاب فرض می‌کنند که گراف ورودی به صورت لیست همجواری نمایش داده می‌شود. اما نمایش

---

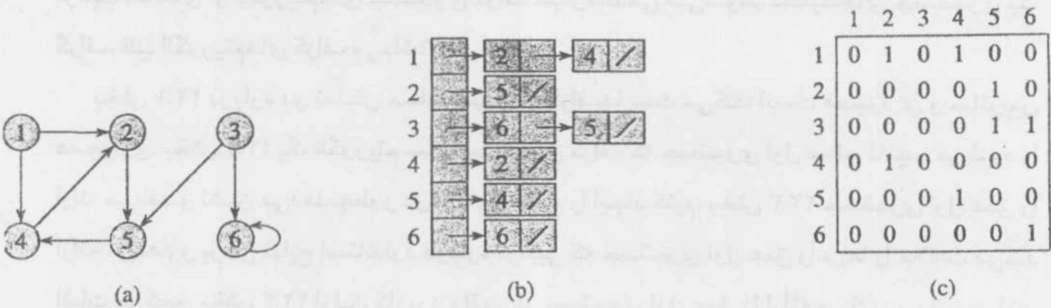
1. *sparse*

ماتریس همجواری ممکن است وقتی گراف **چگال**[1] است ترجیح داده شود - $|E|$ به $|V|^2$ نزدیک است - یا وقتی نیاز داریم به سرعت بیان کنیم آیا یالی وجود دارد که دو رأس را به هم متصل کند یا خیر. برای مثال دو نمونه از الگوریتم‌های کوتاهترین مسیرهای همه جفت‌ها که در فصل ۲۵ ارائه می‌شوند فرض می‌کنند که گراف ورودی با ماتریس همجواری نمایش داده می‌شود.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

شکل ۲۲.۱ دو نمایش یک گراف بدون جهت. *(a)* گراف بدون جهت G که پنج رأس و هفت یال دارد. *(b)* نمایش لیست همجواری گراف G. *(c)* نمایش ماتریس همجواری گراف G.



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

شکل ۲۲.۲ دو نمایش یک گراف جهت‌دار. *(a)* گراف جهت دار G که شش رأس و هشت یال دارد. *(b)* نمایش لیست همجواری گراف G *(c)* نمایش ماتریس همجواری گراف G.

نمایش لیست همجواری[2] گراف $G = (V,E)$ از آرایه Adj از $|V|$ لیست تشکیل شده است یعنی برای هر رأس در V یک لیست. برای هر رأس $u \in V$ لیست همجواری $Adj[u]$ همه رئوس v بطوریکه یال $(u,v) \in E$ وجود دارد را شامل می‌شود. به عبارت دیگر $Adj[u]$ از همه رئوس مجاور u در گراف G تشکیل شده است. (متناوباً ممکن است شامل اشاره‌گرهایی به این رئوس باشد.) رئوس در هر لیست همجواری معمولاً به یک ترتیب دلخواه ذخیره می‌شوند. شکل *(b)* ۲۲.۱ نمایش لیست همجواری گراف بدون جهت شکل *(a)* ۲۲.۱ می‌باشد. به‌طور مشابه، شکل *(b)* ۲۲.۲ نمایش لیست همجواری گراف

---

1. dense
2. adjacency-list representation

جهت‌دار شکل (a) ۲۲.۲ می‌باشد.

اگر G یک گراف جهت‌دار باشد مجموع طول همهٔ لیست‌های همجواری برابر $|E|$ است، چون یال به شکل (u,v) با ظاهر شدن v در $Adj[u]$ نمایش داده می‌شود. اگر G یک گراف بدون جهت باشد مجموع طول همه لیست‌های همجواری برابر $2|E|$ می‌باشد زیرا اگر (u,v) یک یال بدون جهت باشد آن گاه u در لیست همجواری v ظاهر می‌شود و برعکس. برای هر دو گراف جهت دار و بدون جهت، نمایش لیست همجواری دارای این ویژگی مطلوب است که مقدار حافظه مورد نیاز $\Theta(V+E)$ است.

لیست همجواری می‌تواند به آسانی برای نمایش *گراف‌های وزن دار* [1] وفق داده شود، به عبارت دیگر گراف‌هایی که برای هر یال یک *وزن* [2] مرتبط دارد، که این وزن معمولاً با *تابع وزن* [3] $w : E \rightarrow R$ بدست می‌آید. برای مثال فرض کنید $G = (V,E)$ یک گراف وزن دار همراه با تابع وزن w باشد. وزن $w(u,v)$ یال $(u,v) \in E$ به سادگی به همراه رأس v در لیست همجواری u ذخیره می‌شود. نمایش لیست همجواری بسیار قوی است زیرا می‌تواند برای پشتیبانی انواع مختلف گراف تغییر داده شود.

زیان پتانسیلی لیست همجواری این است که جهت تعیین آن که آیا یال (u,v) در گراف وجود دارد، راه سریعتری از این که v را در لیست همجواری $Adj[u]$ جستجو کنیم وجود ندارد. این عیب می‌تواند با نمایش ماتریس همجواری گراف جبران شود که بهای آن استفاده مجانبی بیشتر از حافظه است. (تمرین ۸ - ۲۲.۱ را ملاحظه نمایید که لیست همجواری متفاوتی را ارائه می‌دهد که جستجوی سریعتری را امکان‌پذیر می‌کند.)

برای *نمایش ماتریس همجواری* [4] گراف $G = (V,E)$ فرض می‌کنیم که رئوس از $1,2,..., |V|$ به روشی دلخواه شماره‌گذاری می‌شوند. آن گاه نمایش ماتریس همجواری گراف G تشکیل شده است از یک ماتریس $|V| \times |V|$ به نام $A=(a_{ij})$ بطوریکه

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \,, \\ 0 & \text{otherwise} \,. \end{cases}$$

شکل‌های (c) ۲۲.۱ و (c) ۲۲.۲ به ترتیب ماتریس‌های همجواری گراف‌های بدون جهت و جهت دار شکل‌های (a) ۲۲.۱ و (a) ۲۲.۲ می‌باشند. ماتریس همجواری یک گراف نیاز به $\Theta(V^2)$ حافظه دارد و مستقل از تعداد یال‌ها در گراف می‌باشد.

تقارن در طول قطر اصلی ماتریس همجواری شکل (c) ۲۲.۱ را مشاهده کنید. ترانهاده [5] ماتریس $A = (a_{ij})$ را ماتریس $A^T = (a_{ij}^T)$ تعریف می‌کنیم که از $a^T_{ij} = a_{ji}$ بدست می‌آید. از آنجایی که در گراف بدون جهت، (u,v) و (v,u) یال یکسانی را نمایش می‌دهند، ماتریس همجواری A یک گراف

---

1. weighted graphs
2. weight
3. weight function
4. adjacency-matrix representation
5. transpose

بدون جهت برابر با ترانهاده‌اش است: $A = A^T$ در برخی کاربردها تنها ورودی‌های بالا ورودی قطر ماتریس همجواری ذخیره می‌شوند. بنابراین، حافظه مورد نیاز جهت ذخیره گراف تقریباً نصف می‌شود.

همانند نمایش لیست همجواری گراف، نمایش ماتریس همجواری می‌تواند برای گراف‌های وزن‌دار استفاده شود. برای مثال اگر $G = (V,E)$ یک گراف وزن‌دار با تابع وزن w باشد، وزن $w(u,v)$ یال $(u,v) \in E$ به سادگی به عنوان ورودی در سطر u و ستون v ماتریس همجواری ذخیره می‌شود. اگر یک یال وجود نداشته باشد مقدار *NIL* می‌تواند به عنوان داده متناظر در ماتریس ذخیره شود، اگر چه برای بسیاری از مسائل استفاده از مقادیر 0 یا ∞ مناسب است.

اگر چه نمایش لیست همجواری از لحاظ مجانبی حداقل مانند نمایش ماتریس همجواری مؤثر است اما وقتی گراف‌ها به‌طور منطقی کوچک هستند، سادگی ماتریس همجواری ممکن است باعث ترجیح دادن آن گردد. علاوه بر این اگر گراف بدون وزن باشد یک مزیت اضافی در ذخیره سازی برای نمایش ماتریس همجواری وجود دارد. به جای استفاده از یک کلمه حافظه کامپیوتر برای هر ورودی ماتریس، ماتریس همجواری تنها از یک بیت برای هر ورودی استفاده می‌کند.

تمرین‌ها

۱- ۲۲.۱ نمایش لیست همجواری یک گراف جهت‌دار داده شده است، چه مقدار طول می‌کشد تا درجه خروجی هر رأس را محاسبه کنیم؟ چه مقدار طول می‌کشد تا درجه ورودی هر رأس را محاسبه کنیم؟

۲ - ۲۲.۱ نمایش لیست همجواری برای یک درخت کامل دودویی با 7 رأس را ارائه دهید. نمایش ماتریس همجواری معادل را ارائه دهید. فرض کنید که رئوس از 1 تا 7 همانند *heap* دودویی شماره‌گذاری شده است.

۳ - ۲۲.۱ ترانهاده گراف جهت‌دار $G=(V,E)$، گراف $G^T=(V,E^T)$ است که $E^T=\{(v,u) \in V \times V:(u,v)\in E\}$ بنابراین گراف G^T، گراف G با نمایش معکوس یال‌ها می‌باشد. برای نمایش لیست همجواری و نمایش ماتریس همجواری G الگوریتم‌هایی مؤثر برای محاسبه G^T از روی G ارائه دهید. زمان اجرای الگوریتم‌های خود را تحلیل کنید.

۴- ۲۲.۱ نمایش لیست همجواری گراف چندگانه $G=(V,E)$ داده شده است، الگوریتمی با زمان اجرای $O(V+E)$ برای محاسبه نمایش لیست همجواری گراف بدون جهت «معادل[1]» $G'=(V,E')$ ارائه دهید که در آن E' از یال‌های E به همراه یالهای چندگانه E که با یک تک یال جایگزین شده‌اند تشکیل شده است و همه خود حلقه‌ها حذف گردیده‌اند.

---

1. equivalent

۲۲.۱-۵ مربع[1] گراف جهت‌دار $G=(V,E)$، گراف $G^2=(V,E^2)$ است بطوریکه $(u,w)\in E^2$ اگر و فقط اگر برای یک $v\in V$، هر دو یال $(u,v)\in E$ و $(v,w)\in E$ باشند. به عبارت دیگر G^2 شامل یال بین u و w است هرگاه G شامل یک مسیر با دقیقاً دو یال بین u و w باشد. الگوریتم‌های مؤثری برای محاسبه G^2 از روی G برای هر دو نمایش لیست همجواری و ماتریس همجواری G بدست آورید. زمان اجرای الگوریتم‌های خود را تحلیل کنید.

۲۲.۱-۶ وقتی که نمایش ماتریس همجواری استفاده می‌شود اکثر الگوریتم‌های گراف به زمان $\Omega(V^2)$ نیاز دارند اما برخی استثناءها وجود دارند. نشان دهید تعیین این که آیا گراف جهت‌دار G شامل یک *حفره جامع*[2] است - رأسی با درجه ورودی $|V|-1$ و درجه خروجی 0- می‌تواند با داشتن ماتریس همجواری G در زمان $O(V)$ مشخص گردد.

۲۲.۱-۷ *ماتریس برخورد*[3] گراف جهت‌دار $G=(V,E)$، ماتریس $B=(b_{ij})$ با ابعاد $|V|\times|E|$ است بطوریکه

$$b_{ij}=\begin{cases}-1 & \text{اگر یال } j \text{ رأس } i \text{ را ترک کند}\\ 1 & \text{اگر یال } j \text{ به رأس } i \text{ وارد شود}\\ 0 & \text{در غیر این صورت}\end{cases}$$

توضیح دهید که ورودی‌های ماتریس حاصلضرب BB^T چه چیزی را نمایش می‌دهند، که B^T ترانهاده B است.

۲۲.۱-۸ فرض کنید به جای لیست پیوندی، هر ورودی آرایه $Adj[u]$ یک جدول درهم سازی شامل رئوس v که برای آن‌ها $(u,v)\in E$ است، باشد. اگر همه جستجوهای یالها، امکان برابری داشته باشند زمان مورد انتظار برای مشخص کردن این که آیا یک یال در گراف واقع است یا خیر چقدر است؟ این طرح چه معایبی دارد؟ یک ساختمان داده جایگزین برای هر لیست یال پیشنهاد کنید که این مشکلات را حل کند. آیا این ساختمان دادهٔ جایگزین در مقایسه با جدول درهم سازی دارای معایبی است؟

۲۲.۲ جستجوی اول سطح[4]

جستجوی اول سطح یکی از ساده‌ترین الگوریتم‌ها برای جستجوی گراف و نوعی پایه برای بسیاری از الگوریتم‌های مهم گراف است. الگوریتم درخت پوشای مینیمم *prim* (بخش ۲۳.۲) و الگوریتم کوتاهترین مسیرها از مبدأ[5] واحد *Dijkstra* (بخش ۲۴.۳) از ایده مشابه جستجوی اوّل سطح استفاده می‌کنند.

---

1. *square*
2. *universal sink*
3. *incidence matrix*
4. *Breadth-first search*
5. *source*

گراف $G=(V,E)$ و رأس مبدأ مشخص s داده شده است، جستجوی اول سطح به صورت اصولی یال‌های G را برای پیدا کردن رئوسی که از s قابل دستیابی هستند مرور می‌کند. این الگوریتم فاصله (کمترین تعداد یال‌ها) هر رأس قابل دستیابی از s را محاسبه می‌کند. همچنین "درخت اول سطح"[1] با ریشه s که شامل تمامی رئوس قابل دستیابی می‌باشد را تولید می‌کند. برای هر رأس v قابل دستیابی از s مسیر واقع در درخت جستجوی اول سطح از s به v متناظر با "کوتاهترین مسیر"[2] از s به v در G می‌باشد، به عبارت دیگر مسیری شامل کمترین تعداد یال. الگوریتم بر روی هر دو گراف جهت‌دار و بدون جهت اعمال می‌شود.

علت این که این الگوریتم، جستجوی اول سطح نامیده شده، این است که یک مرز بین رئوس کشف شده و رئوس کشف نشده به‌طور یکنواخت در راستای سطح مرز عبوری توسعه می‌دهد. به عبارت دیگر الگوریتم همهٔ رأس‌های با فاصله k از s را قبل از اینکه رئوس با فاصله $k+1$ را کشف کند کشف می‌کند.

در ادامه، جستجوی اول سطح هر رأس را با سفید، خاکستری یا سیاه رنگ می‌کند. همه رئوس در ابتدا سفید هستند اما ممکن است بعد خاکستری و آن گاه سیاه شوند. یک رأس اولین برای که در طول جستجو ملاقات می‌گردد کشف[3] می‌شود، که در آن زمان غیر سفید می‌شود، بنابراین رئوس سیاه و خاکستری کشف شده‌اند اما جستجوی اول سطح بین آنها تمایز قائل می‌شود تا اطمینان حاصل کند که جستجو به شکل اول سطح پیش می‌رود. اگر $(u,v) \in E$ و رأس u سیاه باشد آن گاه رأس v سیاه یا خاکستری است؛ به عبارت دیگر همه رئوس مجاور رئوس سیاه کشف شده‌اند. رئوس خاکستری ممکن است رأس مجاور به رنگ سفید داشته باشند؛ آنها مرز بین رئوس کشف شده و کشف نشده را نشان می‌دهند.

جستجوی اول سطح یک درخت اول سطح را می‌سازد، که در آغاز تنها شامل ریشه است که رأس منبع s می‌باشد. هر وقت رأس سفید v در مرحله پویش لیست همجواری رأس کشف شده u کشف شود رأس v و یال (u,v) به درخت اضافه می‌شوند. می‌گوییم u *ماقبل*[4] یا پدر[5] v در درخت جستجوی اول سطح است. از آنجایی که هر رأس حداکثر یک بار کشف می‌شود حداکثر یک پدر دارد. رابطه جد و نتیجه (نسل) در درخت جستجوی اول سطح به شکل معمول نسبت به ریشه s تعریف می‌شود: اگر u در مسیری از درخت از s به v واقع باشد آن گاه u جد v و v نتیجه u است.

روال جستجوی اول سطح BFS زیر فرض می‌کند که گراف ورودی $G=(V,E)$ با استفاده از لیست همجواری نمایش داده می‌شود. چندین ساختمان داده اضافی همراه هر رأس در گراف نگهداری

---

1. Breadth-first tree
2. shortest path
3. discovere
4. predecessor
5. parent

می‌شود. رنگ هر رأس $u \in V$ در متغیر $color[u]$ و ماقبل U در متغیر $\pi[u]$ ذخیره می‌شود. اگر u فاقد ماقبل باشد (برای مثال اگر $u = s$ یا u کشف نشده باشد) آن گاه $\pi[u] = NIL$ فاصله از منبع s تا رأس u که توسط الگوریتم محاسبه می‌شود در $d[u]$ ذخیره می‌گردد. الگوریتم همچنین از صف اولین ورودی اولین خروجی Q (بخش ۱۰.۱ را ملاحظه نمایید) برای مدیریت مجموعه رئوس خاکستری استفاده می‌کند.

```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2      do color[u] ← WHITE
 3         d[u] ← ∞
 4         π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← Ø
 9  ENQUEUE(Q, s)
10  while Q ≠ Ø
11      do u ← DEQUEUE(Q)
12         for each v ∈ Adj[u]
13             do if color[v] = WHITE
14                   then color[v] ← GRAY
15                        d[v] ← d[u] + 1
16                        π[v] ← u
17                        ENQUEUE(Q, v)
18         color[u] ← BLACK
```

شکل ۲۲.۳ روند BFS را بر روی یک گراف نمونه نشان می‌دهد.

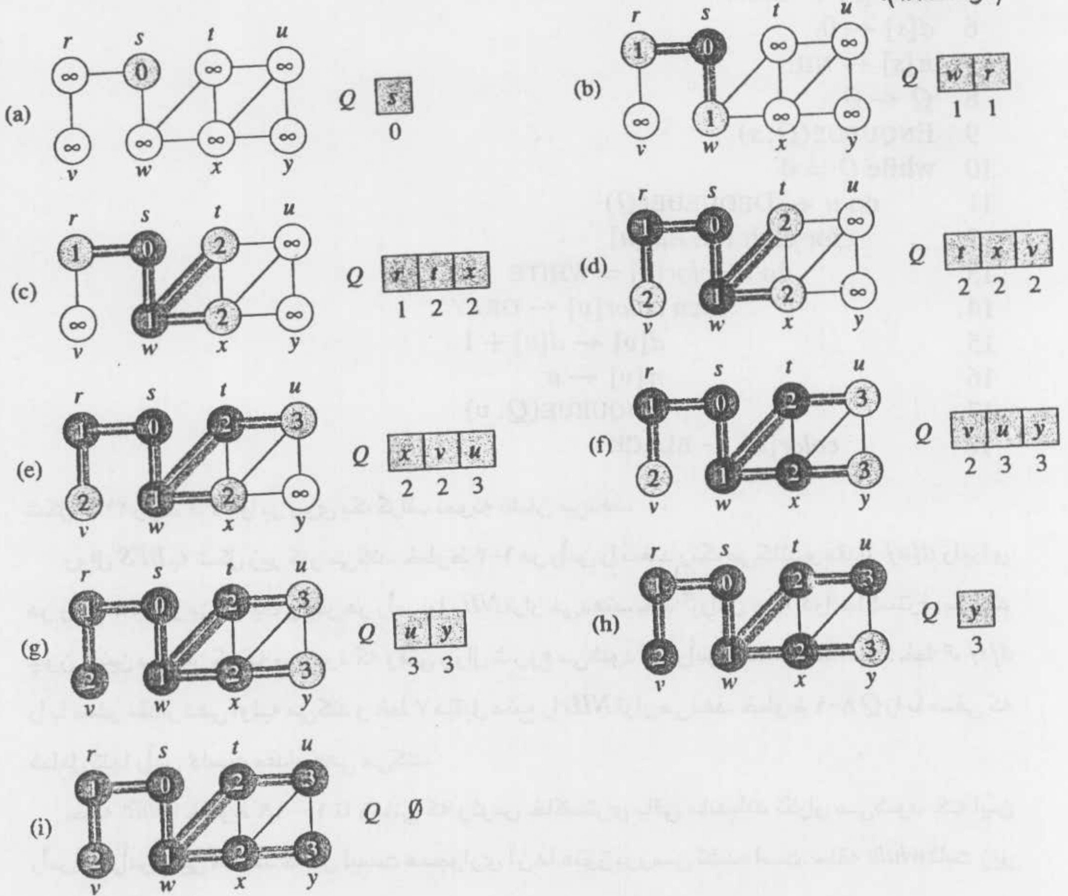
روال BFS به شکل زیر کار می‌کند، خطوط ۴-۱ هر رأس را سفید رنگ می‌کنند و مقدار $d[u]$ را برای هر رأس u مقدار بی نهایت و پدر هر رأس را NIL قرار می‌دهند. خط 5 رأس مبدأ s را خاکستری می‌کند چون چنین در نظر گرفته می‌شود که وقتی روال شروع می‌شود این رأس کشف شده است. خط ۶، $d[s]$ را با صفر مقدار دهی اولیه می‌کند و خط ۷ ماقبل منبع را NIL قرار می‌دهد. خطوط ۹-۸ Q را با صفی که شامل تنها رأس s است مقدار دهی می‌کند.

حلقه $while$ خطوط ۱۸-۱۰ تا زمانی که رئوس خاکستری باقی مانده‌اند تکرار می‌شود، که این رأس‌ها، رأس‌هایی هستند که کل لیست همجواری آن‌ها هنوز بررسی نشده است. حلقه $while$ ثابت زیر را حفظ می‌کند:

در تست خط ۱۰، صف Q از یک مجموعه رئوس خاکستری تشکیل می‌شود.

اگر چه از این ثابت حلقه جهت اثبات درستی الگوریتم استفاده نمی‌کنیم، اما به راحتی مشاهده

می‌شود که این ثابت قبل از تکرار اول برقرار است و این که هر تکرار حلقه ثابت را حفظ می‌کند. قبل از اولین تکرار تنها رأس خاکستری و تنها رأس در صف Q رأس مبدأ s است. خط ۱۱ رأس خاکستری u در ابتدای صف Q را تعیین و آن را از Q حذف می‌کند. حلقه *for* خطوط ۱۷-۱۲ هر رأس v در لیست همجواری u را در نظر می‌گیرد. اگر v سفید باشد هنوز کشف نشده است و الگوریتم آن را با اجرای خطوط ۱۷-۱۶ کشف می‌کند. این رأس ابتدا خاکستری می‌شود و فاصله $d[v]$ برابر $d[u] + 1$ قرار می‌گیرد. سپس u به عنوان پدر آن ذخیره می‌شود و سرانجام در انتهای صف Q قرار می‌گیرد. وقتی همه رئوس در لیست همجواری u بررسی شده باشند u در خطوط ۱۸-۱۱ سیاه رنگ می‌شود. ثابت حلقه حفظ می‌شود زیرا هر زمانی که یک رأس خاکستری می‌شود (در خط ۱۴) به صف نیز اضافه می‌شود (در خط ۱۷)، و زمانی که یک رأس از صف خارج می‌شود (در خط ۱۱)، سیاه رنگ نیز می‌شود (در خط ۱۸).



شکل ۲۲.۳ عملکرد *BFS* بر روی گراف بدون جهت. یال‌های درختی همان‌طور که بوسیله *BFS* تولید می‌شوند به صورت سایه زده نشان داده می‌شوند. درون هر رأس u $d[u]$ نشان داده می‌شود. صف Q در ابتدای هر تکرار حلقه *while* خطوط ۱۸ - ۱۰ نمایش داده می‌شود. فاصله رئوس کنار رئوس در صف نمایش داده می‌شوند.

نتایج جستجوی اول سطح ممکن است به ترتیب ملاقات رئوس همجوار رأس مورد نظر در خط ۱۲ بستگی داشته باشند: درخت اول سطح ممکن است تغییر کند. اما فاصله d محاسبه شده توسط الگوریتم تغییر نمی‌کند. (تمرین ۴ - ۲۲.۲ را ملاحظه کنید.)

تحلیل

قبل از اثبات ویژگی‌های مختلف جستجوی اول سطح، کار به مراتب ساده‌تری یعنی تحلیل زمان اجرای آن روی گراف ورودی $G = (V,E)$ را انجام می‌دهیم. از تحلیل جمعی که در بخش ۱۷.۱ دیدیم استفاده می‌کنیم. بعد از مقدار دهی اولیه هیچ رأسی سفید نشده است و بنابراین تست خط ۱۳ اطمینان حاصل می‌کند که رأس حداکثر یک بار به صف اضافه می‌شود و از این رو حداکثر یک بار از صف خارج می‌شود. اعمال اضافه کردن به صف و خارج کردن از صف زمان $O(1)$ را صرف می‌کنند، بنابراین زمان کل $O(V)$ برای اعمال صف اختصاص می‌یابد. چون لیست همجواری هر رأس تنها زمانی که آن رأس از صف خارج می‌شود، پویش می‌شود هر لیست همجواری حداکثر یک بار پویش می‌شود. از آنجایی که مجموع طول همه لیست‌های همجواری $\Theta(E)$ است زمان کل برای پویش لیست‌های همجواری برابر با $O(E)$ است. کل زمان برای عمل مقدار دهی اولیه $O(V)$ است، و بنابراین کل زمان اجرای BFS برابر $O(V+E)$ است. لذا جستجوی اول سطح در زمان خطی از اندازه نمایش لیست همجواری G اجرا می‌شود.

کوتاهترین مسیرها

در شروع این بخش ادعا کردیم جستجوی اول سطح در گراف $G=(V,E)$ فاصله هر رأس قابل دستیابی از رأس مبدأ مفروض شده $s \in V$ را پیدا می‌کند. *فاصله کوتاهترین مسیر*[1] $\delta(s,v)$ از s به v به عنوان مینیمم تعداد یال‌ها در مسیر s به v تعریف می‌شود؛ اگر مسیری از s به v وجود نداشته باشد آن‌گاه $\delta(s,v) = \infty$. مسیر با طول $\delta(s,v)$ *کوتاهترین مسیر*[2] از s به v است. قبل از این که نشان دهیم جستجوی اول سطح واقعاً فاصله‌های کوتاهترین مسیر را محاسبه می‌کند، یک ویژگی مهم از فاصله‌های کوتاهترین مسیر را در مورد بررسی قرار می‌دهیم.

لم ۲۲.۱

فرض کنید $G = (V,E)$ یک گراف جهت‌دار یا بدون جهت و $s \in V$ یک رأس دلخواه باشد. آن گاه برای هر یال $(u,v) \in E$ داریم

$$\delta(s, v) \le \delta(s, u) + 1 .$$

---

1. shortest-path distance   2. shortest-path

اثبات اگر u یک رأس قابل دستیابی از s باشد آن گاه v نیز از s قابل دستیابی است. در این حالت کوتاهترین مسیر از s به v نمی‌تواند طولانی‌تر از کوتاهترین مسیر از s به u که به دنبال آن یال (u,v) آمده است باشد، و بنابراین نامساوی برقرار است. اگر u قابل دستیابی از s نباشد، آن گاه $\delta(s,u) = \infty$ و نامساوی برقرار است. ■

می‌خواهیم نشان دهیم که BFS برای هر رأس $v \in V$، $d[v]$ را به‌طور صحیح محاسبه می‌کند. ابتدا نشان می‌دهیم که $d[v]$، $\delta(s,v)$ را از بالا محدود می‌کند.

لم ۲۲.۲

فرض کنید $G = (V,E)$ یک گراف جهت دار یا بدون جهت باشد و فرض کنید BFS بر روی G از رأس مبدأ داده شده $s \in V$ اجرا می‌شود. آنگاه در خاتمه، برای هر رأس $v \in V$، مقدار $d[v]$ که توسط BFS محاسبه شده است $d[v] \geq \delta(s,v)$ را برقرار می‌کند.

اثبات از استقرا بر روی تعداد اعمال *ENQUEUE* استفاده می‌کنیم. فرض استقراء این است که برای همهٔ $v \in V$ داریم $d[v] \geq \delta(s,v)$

پایه استقراء وضعیت موجود درست بعد از اضافه شدن s به صف در خط ۹ روال BFS است. فرض استقراء در این جا حفظ می‌شود زیرا برای همه $v \in V - \{s\}$ داریم $d[s] = 0 = \delta(s,s)$ و $d[v] = \infty \geq \delta(s,v)$

برای گام استقراء رأس سفید v که در طی جستجو از رأس u کشف می‌شود را در نظر بگیرید. فرض استقراء دلالت می‌کند بر این که $d[u] \geq \delta(s,u)$. از عمل انتساب که توسط خط ۱۵ اجرا می‌شود و از لم ۲۲.۱ چنین حاصل می‌شود

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq \delta(s,u) + 1 \\ &\geq \delta(s,v) . \end{aligned}$$

سپس رأس v در صف قرار می‌گیرد و دوباره هرگز در صف قرار نخواهد گرفت زیرا خاکستری رنگ می‌شود، و عبارت *then* خطوط ۱۷ - ۱۴ تنها برای رئوس سفید اجرا می‌شود. بنابراین مقدار $d[v]$ دیگر هرگز تغییر نمی‌کند و فرض استقراء حفظ می‌شود. ■

برای اثبات اینکه $d[v] = \delta(s,v)$، ابتدا باید به‌طور دقیقتر نشان دهیم صف Q در طی BFS چگونه عمل می‌کند. لم بعدی نشان می‌دهد که در همه زمانها حداکثر دو مقدار متمایز d در صف قرار دارند.

لم ۲۲.۳

فرض کنید در طی اجرای BFS روی گراف $G = (V,E)$، صف Q شامل رئوس $\langle v_1, v_2, ..., v_r \rangle$ باشد که

در آن v_1 در ابتدای صف و v_r در انتهای صف قرار دارد. آن گاه برای $i=1,2,...,r-1$ داریم $d[v_r] \leq d[v_1]+1$ و $d[v_i] \leq d[v_{i+1}]$

اثبات اثبات به وسیله استقراء بر روی تعداد اعمال صف صورت می‌گیرد. در ابتدا هنگامی که صف فقط شامل s است لم قطعاً برقرار است.

برای گام استقرایی باید ثابت کنیم که لم بعد از اعمال اضافه کردن به صف و حذف از صف برقرار است. اگر سر صف یعنی v_1 از صف خارج شود آنگاه v_2 ابتدای جدید صف می‌شود. (اگر صف خالی شود آن گاه لم مجازاً برقرار است.) بنا به فرض استقراء داریم $d[v_1] \leq d[v_2]$. لذا داریم $d[v_r] \leq d[v_1]+1 \leq d[v_2]+1$ و نامعادلات باقیمانده تأثیری نمی‌پذیرند. بنابراین لم با v_2 به عنوان عنصر ابتدا دنبال می‌شود.

اضافه کردن رأس به صف نیاز به بررسی دقیقتر کد دارد. وقتی رأس v را در خط ۱۷ روال BFS، به صف اضافه می‌کنیم رأس v_{r+1} می‌شود. در آن زمان از قبل رأس u که لیست همجواری آن پویش می‌شود را از صف Q حذف کرده‌ایم و بنا به فرض استقراء v_1 که سر جدید صف است دارای $d[v_1]>d[u]$ است. بنابراین $d[v_{r+1}]=d[v]=d[u]+1\leq d[v_1]+1$. از فرض استقراء همچنین داریم $d[v_r]\leq d[u]+1$ و بنابراین $d[v_r]\leq d[u]+1=d[v]=d[v_{r+1}]$ و نامعادلات باقیمانده تأثیری نمی‌پذیرند. بنابراین لم هنگامی که v به صف اضافه می‌شود ثابت می‌شود. ■

قضیه فرعی زیر نشان می‌دهد در زمانی که رئوس به صف اضافه می‌شوند مقادیر d به‌طور یکنواخت با زمان افزایش می‌یابد.

قضیه فرعی ۲۲.۴

فرض کنید که رئوس v_i و v_j در طی اجرای BFS به صف اضافه می‌شوند و فرض کنید v_i قبل از v_j به صف اضافه می‌شود. آن گاه در زمانی که v_j به صف اضافه می‌شود داریم $d[v_i]\leq d[v_j]$

اثبات مستقیماً از لم ۲۲.۳ و این ویژگی که در طی اجرای BFS هر رأس مقدار محدود d را حداکثر یک‌بار دریافت می‌کند، ثابت می‌شود. ■

اکنون می‌توانیم ثابت کنیم که جستجوی اول سطح مسافتهای کوتاهترین مسیر را به درستی پیدا می‌کند.

قضیه ۲۲.۵ (درستی جستجوی اول سطح)

فرض کنید که $G=(V,E)$ یک گراف جهت دار یا بدون جهت باشد و فرض کنید که BFS روی G از رأس مبدأ $s\in V$ اجرا شود. آن گاه در طی اجرای BFS، این الگوریتم هر رأس $v\in V$ که از s قابل دستیابی است

را کشف می‌کند و در خاتمه برای همه رئوس $v \in V$ داریم $d[v] = \delta(s,v)$ علاوه بر این برای هر رأس $v \neq s$ که از s قابل دستیابی است یکی از کوتاهترین مسیرها از s به v، کوتاهترین مسیر از s به $\pi[v]$ است، که یال $(\pi[v],v)$ به دنبال آن آمده است.

اثبات برای ایجاد تناقض فرض کنید یک رأس یک مقدار d را دریافت می‌کند که مساوی با کوتاهترین مسیرش نیست. فرض کنید v رأس با کمترین مقدار $\delta(s,v)$ باشد که این مقدار نادرست d را دریافت می‌کند؛ روشن است که $v \neq s$. بنا به لم ۲۲.۲ داریم $d[v] \geq \delta(s,v)$ و بنابراین داریم $d[v] > \delta(s,v)$. رأس v باید از s قابل دستیابی باشد وگرنه $\delta(s,v) = \infty \geq d[v]$. فرض کنید u در کوتاهترین مسیر از s به v درست قبل از v قرار داشته باشد، بنابراین $\delta(s,v) = \delta(s,u)+1$ چون $\delta(s,u) < \delta(s,v)$ و به علت چگونگی انتخاب v داریم $d[u] = \delta(s,u)$. با قرار دادن این ویژگی‌ها در کنار هم داریم:

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1. \qquad (۲۲.۱)$$

اکنون زمانی را در نظر بگیرید که BFS در خط ۱۱ رأس u را برای حذف از Q انتخاب می‌کند. در این زمان رأس v سفید، خاکستری یا سیاه است. نشان خواهیم داد که در هر یک از این حالت‌ها، به تناقضی برای نامعادله (۲۲.۱) می‌رسیم. اگر v سفید باشد آن گاه در خط ۱۵ مقداردهی $d[v] = d[u]+1$ انجام می‌شود، که با نامعادله (۲۲.۱) در تناقض است. اگر v سیاه باشد آن گاه قبلاً از صف حذف شده است و بنا به قضیه فرعی ۲۲.۴ داریم $d[v] \leq d[u]$، که باز هم با نامعادله (۲۲.۱) در تناقض قرار دارد. اگر v خاکستری باشد آن گاه در طی حذف رأس w از صف خاکستری شده است که زودتر از u از صف حذف شده است و برای آن $d[v] = d[w]+1$. اما بنا به قضیه فرعی ۲۲.۴، $d[w] \leq d[u]$ و بنابراین داریم $d[v] \leq d[u]+1$. که دوباره نامساوی ۲۲.۱ را نقض می‌کند.

بنابراین نتیجه می‌گیریم برای همه رئوس $v \in V$، $d[v] = \delta(s,v)$ همه رئوس قابل دستیابی از s باید کشف شوند زیرا اگر کشف نمی‌شدند دارای مقادیر بی نهایت d بودند. برای خاتمه اثبات قضیه مشاهده کنید اگر $\pi[v] = u$ آنگاه $d[v] = d[u]+1$ بنابراین با بدست آوردن کوتاهترین مسیر از s به $\pi[v]$ و سپس پیمایش یال $(\pi[v],v)$، می‌توانیم کوتاهترین مسیر از s به v را بدست آوریم. ■

درخت‌های اول سطح

روال BFS همانطور که در شکل ۲۲.۳ نشان داده شده است همزمان با جستجوی گراف یک درخت اول سطح می‌سازد. درخت با فیلد π در هر رأس نمایش داده می‌شود.

به‌طور رسمی‌تر برای گراف $G=(V,E)$ با رأس مبدأ s زیر گراف ماقبل[1] G را به صورت $G_\pi = (V_\pi, E_\pi)$ تعریف می‌کنیم که

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

1. predecessor subgraph

و

$$E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\} \ .$$

زیر گراف ماقبل G_π یک *درخت اول سطح*[1] است اگر V_π از رئوس قابل دستیابی از s تشکیل شده باشد، و برای همه $v \in V_\pi$ یک مسیر ساده منحصر به فرد از s به v در G_π وجود دارد که کوتاهترین مسیر از s به v در G نیز می‌باشد. درخت اول سطح در واقع یک درخت است، زیرا همبند است و $|E_\pi| = |V_\pi| - 1$. یال‌های داخل E_π، *یال‌های درختی*[2] نامیده می‌شوند. بعد از آن که BFS روی گراف G از رأس مبدأ s اجرا شود لم زیر نشان می‌دهد که زیر گراف ماقبل، یک درخت اول سطح است.

لم ۲۲.۶

وقتی روال BFS روی گراف بدون جهت یا جهت‌دار $G=(V,E)$ به کاربرده می‌شود π را طوری می‌سازد که زیر گراف ماقبل $G_\pi=(V_\pi,E_\pi)$ یک درخت اول سطح است.

اثبات خط ۱۶ روال BFS مقدار دهی $\pi[v]=u$ را انجام می‌دهد اگر و فقط اگر $(u,v) \in E$ و $\delta(s,v) < \infty$ – به عبارت دیگر اگر v قابل دسترسی از s باشد – بنابراین V_π از رئوسی در V که قابل دسترسی از s می‌باشند تشکیل شده است. از آنجایی که G_π یک درخت را شکل می‌دهد شامل یک مسیر منحصر به فرد از s به هر رأس در V_π است. با به کارگیری قضیه ۲۲.۵ به صورت استقرایی نتیجه می‌گیریم که هر چنین مسیری، کوتاهترین مسیر است. ∎

روال زیر رئوس روی کوتاهترین مسیر از s به v را چاپ می‌کند، با این فرض که BFS از قبل جهت محاسبه کوتاهترین مسیر درخت اجرا شده است.

```
PRINT-PATH(G, s, v)
1  if v = s
2      then print s
3      else if π[v] = NIL
4              then print "no path from" s "to" v "exists"
5              else PRINT-PATH(G, s, π[v])
6                   print v
```

این روال در زمان خطی از تعداد رئوس چاپ شده در مسیر اجرا می‌شود، چون هر فراخوانی بازگشتی برای یک مسیر با یک رأس کوتاهتر انجام می‌شود.

---

1. breadth-first tree   2. tree edge

تمرین‌ها

۲۲.۲ - ۱ مقادیر d و π که از اجرای جستجوی اول سطح بر روی گراف جهت‌دار شکل (a) ۲۲.۲ حاصل می‌شوند را نشان دهید، از رأس 3 به عنوان رأس مبدأ استفاده کنید.

۲۲.۲ - ۲ مقادیر π و d که از اجرای جستجوی اول سطح بر روی گراف بدون جهت شکل ۲۲.۳ حاصل می‌شوند را نشان دهید از رأس u به عنوان رأس مبدأ استفاده کنید.

۲۲.۲ - ۳ اگر گراف ورودی با یک ماتریس همجواری نمایش داده شود و الگوریتم تغییر کرده تا این شکل از ورودی مدیریت شود زمان اجرای BFS چیست؟

۲۲.۲ - ۴ ثابت کنید که در جستجوی اول سطح، مقدار $d[u]$ اختصاص یافته به رأس u مستقل از ترتیبی است که رئوس در هر لیست همجواری داده شده دارند. با استفاده از شکل ۲۲.۳ به عنوان نمونه، نشان دهید درخت اول سطح محاسبه شده به وسیله BFS می‌تواند به ترتیب درون لیست‌های همجواری بستگی داشته باشد.

۲۲.۲ - ۵ مثالی از گراف جهت‌دار $G = (V,E)$، رأس مبدأ $s \in V$ و مجموعه $E_\pi \subseteq E$ شامل یال‌های درختی ارائه دهید بطوریکه برای هر رأس $v \in V$ مسیر منحصر به فرد در گراف (V,E_π)، از s به v، کوتاهترین مسیر در G باشد. با وجود این مجموعه یال‌های E_π نمی‌توانند به وسیله اجرای BFS روی G تولید شده باشند صرف نظر از این که چطور رئوس در هر لیست همجواری مرتب شده‌اند.

۲۲.۲ - ۶ دو نوع کشتی‌گیر حرفه‌ای وجود دارد: کشتی گیر خوب و کشتی گیر بد بین هر جفت از کشتی گیران حرفه‌ای ممکن است رقابت باشد یا نباشد. فرض کنید n کشتی گیر حرفه‌ای و یک لیست از r جفت از کشتی گیرهایی که بین آنها رقابت وجود دارد داریم. یک الگوریتم با مرتبه زمانی $O(n + r)$ ارائه دهید که مشخص کند آیا امکان دارد برخی از کشتی گیرها را به عنوان کشتی گیر خوب و بقیه را به عنوان کشتی گیر بد تعیین کرد، بطوریکه هر رقابت بین یک کشتی‌گیر خوب و یک کشتی‌گیر بد باشد. اگر امکان اجرای چنین طرحی وجود دارد الگوریتم شما باید این طرح را ایجاد کند.

۲۲.۲ - ۷ قطر[1] درخت $T=(V,E)$ به صورت زیر داده می‌شود

$$\max_{u,v \in V} \delta(u, v) ;$$

به عبارت دیگر قطر، بزرگترین مسافت بین همه مسافتهای کوتاهترین مسیرها در درخت است. یک الگوریتم مؤثر برای محاسبه قطر یک درخت ارائه دهید و زمان اجرای الگوریتم خود را تحلیل کنید.

۲۲.۲ - ۸ فرض کنید $G = (V,E)$ یک گراف بدون جهت و همبند باشد. یک الگوریتم با مرتبه زمانی $O(V+E)$ برای محاسبه یک مسیر در G که هر یال در E را دقیقاً یک بار در هر جهت پیمایش می‌کند ارائه دهید. اگر ذخیره بزرگی از سکه‌های پنی داشته باشید توضیح دهید چگونه راه خود را در یک

---

1. diameter

مسیر پر پیچ و خم پیدا می‌کنید.

۲۲.۳ جستجوی اول عمق

استراتژی که به وسیله جستجوی اول عمق دنبال می‌شود، همان‌طور که نامش اشاره دارد، جستجوی «عمیق‌تر[1]» در گراف تا زمانی که ممکن است، می‌باشد. در جستجوی اول عمق یال‌های خروجی از رأس v که اخیراً کشف شده و هنوز دارای یال‌های کشف نشده می‌باشد مرور می‌شوند. وقتی همه یال‌های v مرور شده باشند، جستجو برای مرور یال‌های خروجی از رأسی که از آن v کشف شده بود "بازگشت به عقب[2]" می‌کند. این فرآیند ادامه می‌یابد تا زمانی که همه یال‌های قابل دسترسی از مبدأ اصلی s را کشف کنیم. اگر رأس‌های کشف نشده‌ای باقی بمانند، آن گاه یکی از آن رئوس به عنوان مبدأ جدید انتخاب می‌گردد و جستجو از آن مبدأ تکرار می‌شود. کل این فرآیند تا زمانی که همه رئوس کشف شوند تکرار می‌گردد.

همانند جستجوی اول سطح، هنگامی که رأس v در طی پویش لیست همجواری رأس u که قبلاً کشف شده، کشف می‌گردد جستجوی اول عمق این رویداد را با قرار دادن u در فیلد ماقبل v یعنی $\pi[v]$ ثبت می‌کند. بر خلاف جستجوی اول سطح که زیر گراف ماقبل آن یک درخت را تشکیل می‌دهد[3]، *زیر گراف ماقبل*[4] تولید شده به وسیله جستجوی اول عمق ممکن است از چندین درخت تشکیل شده باشد، چون جستجو ممکن است از چند مبدأ تکرار شود. بنابراین زیر گراف ماقبل جستجوی اول عمق اندکی متفاوت با زیر گراف ماقبل جستجوی اول سطح تعریف می‌شود: فرض می‌کنیم $G_\pi=(V,E_\pi)$ که

$$E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\} \ .$$

زیر گراف ماقبل جستجوی اول عمق، *جنگل اول عمق*[5] تشکیل شده از چندین *درخت اول عمق*[6] را شکل می‌دهد. یال‌ها در E_π *یال‌های درختی*[7] نامیده می‌شوند.

همانند جستجوی اول سطح، رئوس در طی جستجو جهت مشخص شدن وضعیت شان رنگ می‌شوند. هر رأس در آغاز سفید است، وقتی در جستجو *کشف می‌شود*[8] خاکستری می‌گردد و وقتی

---

1. deeper
2. backtrack

۳ - ممکن است اختیاری به نظر برسد که جستجوی اول سطح تنها به یک مبدأ محدود می‌شود در حالی که جستجوی اول عمق ممکن است از چند مبدأ جستجو کند. اگرچه از لحاظ مفهومی، جستجوی اول سطح می‌تواند از چندین مبدأ پیش برود و جستجوی اول عمق می‌تواند به یک مبدأ محدود شود، روش ما این که چگونه نتایج این جستجوها عموماً استفاده می‌شوند را منعکس می‌کند. روش جستجوی اول سطح معمولاً برای پیدا کردن کوتاهترین مسیرها (و زیر گراف قبل مربوطه) از مبدأ داده شده به کار می‌رود. همان‌طور که بعداً در این فصل خواهیم دید جستجوی اول عمق اغلب به عنوان یک زیر روال برای الگوریتم‌های دیگر به کار می‌رود.

4. predecessor subgraph
5. depth-first-forest
6. depth-first tree
7. tree edge
8. discovered

خاتمه[1] می‌یابد سیاهرنگ می‌شود، به عبارت دیگر وقتی لیست همجواری آن به‌طور کامل بررسی می‌گردد. این تکنیک تضمین می‌کند که هر رأس در دقیقاً یک درخت اول عمق خاتمه می‌یابد که به موجب آن این درخت‌ها متمایز هستند.

علاوه بر ایجاد جنگل اول عمق، جستجوی اول عمق هر رأس را برچسب دهی زمانی[2] نیز می‌کند. هر رأس v دارای دو بر چسب زمانی می‌باشد: وقتی رأس v در ابتدا کشف می‌گردد (خاکستری می‌شود) اولین بر چسب $d[v]$ ثبت می‌شود و دومین بر چسب زمانی $f[v]$زمانی ثبت می‌شود که جستجو بررسی لیست همسایگی vرا خاتمه دهد (و vرا سیاهرنگ کند). این بر چسب‌های زمانی در بسیاری از الگوریتم‌های گراف استفاده می‌شوند و به‌طور کلی در استدلال در مورد رفتار جستجوی اول عمق مفید هستند.

روال DFS زیر، زمان کشف رأس uرا در متغیر $d[u]$و زمان خاتمه رأس uرا در متغیر $f[u]$ ثبت می‌کند. این برچسب‌های زمانی اعداد صحیح بین 1تا $2|V|$هستند، چون برای هر یک از $|V|$ رأس یک رویداد کشف و یک رویداد خاتمه وجود دارد. برای هر رأس u داریم

$$d[u] < f[u] . \qquad (۲۲.۲)$$

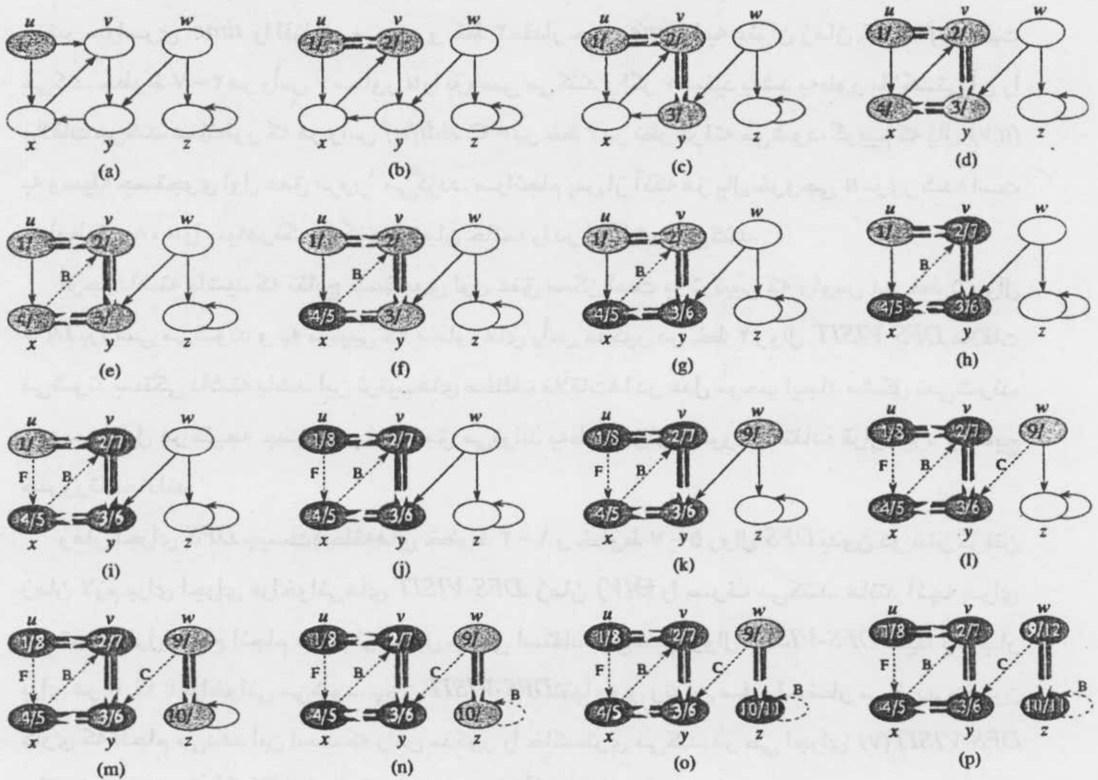
رأس uقبل از زمان $d[u]$، $WHITE$ و بین $d[u]$ و $f[u]$، $GRAY$ و بعد از $f[u]$، $BLACK$ می‌باشد.

شبه کد زیر الگوریتم اصلی جستجوی اول عمق است. گراف ورودی G ممکن است جهت دار یا بدون جهت باشد. متغیر $time$ یک متغیر سراسری است که برای بر چسب دهی زمانی استفاده می‌شود.

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7            then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY              ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]          ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6            then π[v] ← u
7                 DFS-VISIT(v)
8  color[u] ← BLACK             ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

1. finished   2. timestamps



شکل ۲۲.۴ فرآیند اعمال الگوریتم جستجوی اول عمق *DFS* روی یک گراف جهت‌دار. همانطور که یال‌ها توسط الگوریتم مرور می‌شوند، چه سایه زده شده باشند (اگر یال‌های درختی هستند) و چه (در غیر این صورت) به صورت نقطه چنین باشند، نمایش داده شده‌اند. یال‌های غیر درختی بر طبق این که عقب رونده، متقابل، جلو رونده باشند بر چسب‌های *B*، *C* یا *F* را دریافت می‌کنند. رأس‌ها به وسیله زمان کشف / زمان خاتمه بر چسب دهی زمانی شده‌اند.

شکل ۲۲.۴ فرآیند *DFS* را روی گراف نشان داده شده در شکل ۲۲.۲ شرح می‌دهد.

روال *DFS* به شکل زیر کار می‌کند. خطوط ۳ - ۱ همه رئوس را سفید رنگ و فیلدهای π آنها را با *NIL* مقدار دهی اولیه می‌کنند. خط ۴ شمارنده سراسری *time* را صفر می‌کند. خطوط ۷ - ۵ به نوبت هر رأس در V را چک می‌کنند و وقتی یک رأس سفید پیدا می‌شود آن را با استفاده از *DFS-VISIT* ملاقات می‌کنند. هر بار که *DFS-VISIT(u)* در خط ۷ فراخوانی می‌شود، رأس u ریشه یک درخت جدید در جنگل اول عمق می‌شود. وقتی *DFS* بر می‌گردد، به هر رأس u زمان کشف [1] $d[u]$ و زمان خاتمه [2] $f[u]$ اختصاص داده شده است.

---

1. discovery time
2. finishing time

در هر فراخوانی $DFS\text{-}VISIT(u)$ رأس u در ابتدا سفید است. خط ۱، u را خاکستری می‌کند، خط ۲ متغیر سراسری $time$ را افزایش می‌دهد و خط ۳ مقدار جدید $time$ را به عنوان زمان کشف $d[u]$ ثبت می‌کند. خطوط ۷ - ۴ هر رأس v مجاور u را بررسی می‌کنند و اگر v سفید باشد به‌طور بازگشتی آن را ملاقات می‌کند. همانطور که هر رأس $v \in Adj[u]$ در خط ۴ در نظر گرفته می‌شود، گوییم که یال (u,v) به وسیله جستجوی اول عمق مرور[1] می‌گردد. سرانجام پس از آنکه هر یال خروجی u مرور شده است خطوط ۹ - ۸، u را سیاهرنگ می‌کنند و زمان خاتمه را در $f[u]$ ثبت می‌کنند.

توجه داشته باشید که نتایج جستجوی اول عمق ممکن است به ترتیبی که رئوس در خط ۵ روال DFS بررسی می‌شوند و به ترتیبی که مجاورهای رأس مذکور در خط ۴ روال $DFS\text{-}VISIT$ ملاقات می‌شوند بستگی داشته باشد. این ترتیب‌های مختلف ملاقات‌ها در عمل موجب ایجاد مشکل نمی‌شوند، به همین دلیل هر نتیجه جستجوی اول عمق می‌تواند به‌طور مؤثری مورد استفاده قرار گیرد و نتایج ضرورتاً معادلند.

زمان اجرای DFS چیست؟ حلقه‌های خطوط ۳ - ۱ و خطوط ۷ - ۵ روال DFS بدون در نظر گرفتن زمان لازم برای اجرای فراخوانی‌های $DFS\text{-}VISIT$، زمان $\Theta(V)$ را صرف می‌کنند. مانند آنچه برای جستجوی اول سطح انجام دادیم از تحلیل جمعی استفاده می‌کنیم. روال $DFS\text{-}VISIT$ دقیقاً یک بار برای هر $v \in V$ فراخوانی می‌شود، چون $DFS\text{-}VISIT$ تنها روی رئوس سفید احضار می‌گردد و اولین کاری که انجام می‌دهد این است که رأس مذکور را خاکستری می‌کند. در طی اجرای $DFS\text{-}VISIT(v)$ حلقه خطوط ۷ - ۴، $|Adj[v]|$ بار اجرا می‌شود. از آنجایی که

$$\sum_{v \in V} |Adj[v]| = \Theta(E) ,$$

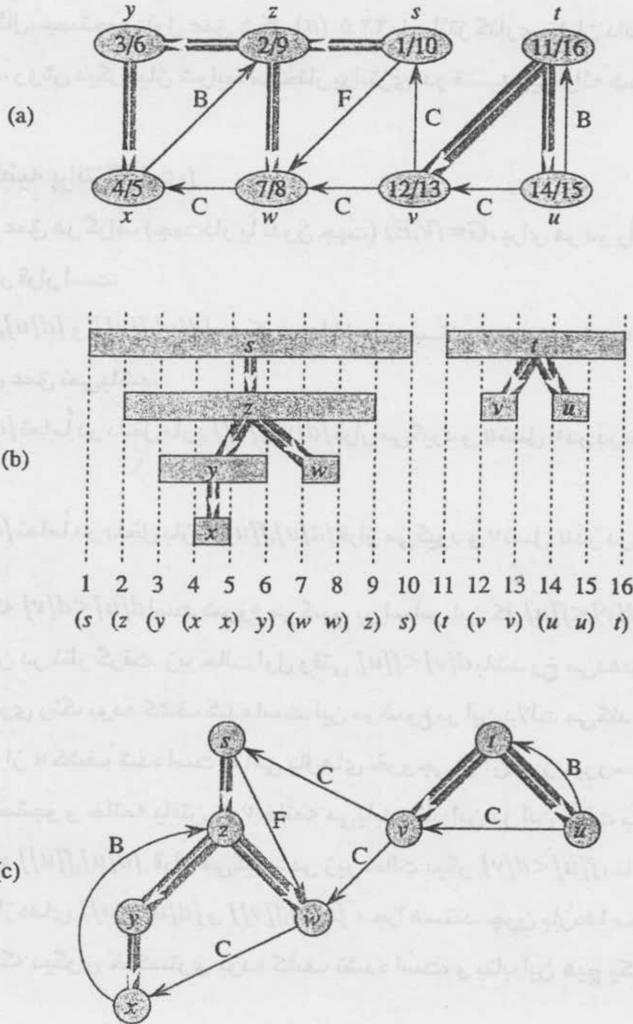
هزینه کل اجرای خطوط ۷ - ۴ روال $DFS\text{-}VISIT$ برابر $\Theta(E)$ می‌باشد. بنابراین زمان اجرای DFS برابر $\Theta(V+E)$ است.

ویژگی‌های جستجوی اول عمق

جستجوی اول عمق باعث می‌شود اطلاعات با ارزشی از ساختار گراف بدست آوریم. شاید اساسی‌ترین ویژگی جستجوی اول عمق این است که زیر گراف ماقبل G_π به راستی یک جنگل از درخت‌ها را شکل می‌دهد، چون ساختار درخت‌های اول عمق، بازتاب دهنده ساختار فراخوانی بازگشتی $DFS\text{-}VISIT$ می‌باشد. به عبارت دیگر $u = \pi[v]$ اگر و فقط اگر $DFS\text{-}VISIT(v)$ در طی جستجوی لیست همجواری u فراخوانی شده باشد. بعلاوه، رأس v نتیجه رأس u در جنگل اول عمق است اگر و فقط اگر v در طی زمانی که u خاکستری است کشف شود.

---

1. explore



شکل ۲۲.۵ ویژگی‌های جستجوی اول عمق. *(a)* نتیجه جستجوی اول عمق روی یک گراف جهت‌دار. رئوس با زمان‌های کشف و خاتمه بر چسب خورده‌اند و انواع یال‌ها همانند شکل ۲۲.۴ مشخص شده‌اند. *(b)* بازه‌های زمانی کشف و خاتمه هر رأس مطابق با پرانتز گذاری نشان داده شده می‌باشند. هر مستطیل بازه حاصل از زمان‌های کشف و خاتمه یک رأس را پوشش می‌دهد. یال‌های درختی نمایش داده شده‌اند. اگر دو بازه تلاقی داشته باشند آن گاه یکی در داخل دیگری قرار دارد و رأس متناظر بازه کوچکتر، درختی نسل رأس متناظر بازه بزرگتر است. *(c)* گراف قسمت *(a)* با همه یال‌های جلو رونده و یال‌های درختی که در داخل درخت اول عمق پایین می‌آیند و همه یال‌های عقب رونده که از نسل به سوی جد بالا می‌آیند.

ویژگی مهم دیگر جستجوی اول عمق این است که زمان‌های کشف و خاتمه *ساختار پرانتزی*[1] دارند. اگر کشف رأس u را با پرانتز چپ کنند «$(u$» و خاتمه را به وسیله پرانتز راست «$u)$» نشان دهیم، آن گاه کشف‌ها و خاتمه‌ها یک عبارت خوش فرم در جمله ایجاد می‌کنند که پرانتزها به‌طور صحیح تو در تو می‌باشند. برای مثال، جستجوی اول عمق شکل (a) ۲۲.۵ با پرانتز گذاری نشان داده شده در شکل (b) ۲۲.۵ متناظر است. روش دیگر بیان شرایط ساختار پرانتزی، در قضیه زیر ارائه شده است.

قضیه ۲۲.۷ (قضیه پرانتزگذاری)

در جستجوی اول عمق هر گراف (جهت‌دار یا بدون جهت) $G=(V,E)$، برای هر دو رأس u و v دقیقاً یکی از سه شرط زیر بر قرار است:

- بازه‌های $[d[u],f[u]]$ و $[d[v],f[v]]$ به کلی جدا از هم هستند، و نه u و نه v فرزند دیگری در جستجوی اول عمق نمی‌باشد،
- بازه $[d[u],f[u]]$ تماماً در داخل بازه $[d[v],f[v]]$ قرار می‌گیرد و u نسل v در درخت اول عمق است و یا
- بازه $[d[v],f[v]]$ تماماً در داخل بازه $[d[u],f[u]]$ قرار می‌گیرد و v نسل u در درخت اول عمق است.

اثبات با حالتی که $d[u]<d[v]$ است شروع می‌کنیم. بر اساس این که $d[v]<f[u]$ باشد یا نباشد دو زیرحالت را می‌توان در نظر گرفت. زیرحالت اول وقتی $d[v]<f[u]$ باشد رخ می‌دهد، بنابراین v زمانی که u هنوز خاکستری رنگ بوده کشف شده است. این موضوع بر این دلالت می‌کند که v نسل u است. بعلاوه چون v بعد از u کشف شده است تمامی یال‌های خروجی از آن مورد بررسی قرار گرفته‌اند، و قبل از بازگشت جستجو و خاتمه یافتن u، v خاتمه می‌یابد. بنابراین در این حالت بازه $[d[v] , f[v]]$ تماماً در داخل بازه $[d[u],f[u]]$ قرار می‌گیرد. در زیر حالت دیگر $f[u]<d[v]$، نامعادله ۲۲.۲ دلالت می‌کند بر این که بازه‌های $[d[u],f[u]]$ و $[d[v],f[v]]$ مجزا هستند. چون بازه‌ها مجزا هستند هیچ یک از دو رأس زمانی که دیگری خاکستری بوده کشف نشده است، و بنابراین هیچ یک از دو رأس نسل دیگری نیست.

حالتی که در آن $d[v]<d[u]$ است مشابه بحث بالا است با این تفاوت که نقش u و v معکوس می‌گردد. ■

قضیه فرعی ۲۲.۸ (تو در تو بودن بازه‌های نسل‌ها)

رأس v یک نسل رأس u در جنگل اول عمق برای گراف G (جهت‌دار یا بدون جهت) است اگر و فقط اگر

---

1. parenthesis structure

$d[u]<d[v]<f[v]<f[u]$

اثبات مستقیماً از قضیه ۲۲.۷ ثابت می‌شود. ■

قضیه بعدی مشخصه مهم دیگری را وقتی یک رأس نسل دیگری در جنگل اول عمق است، ارائه می‌دهد.

قضیه ۲۲.۹ (قضیه مسیر سفید)

در جنگل اول عمق گراف $G=(V,E)$ (جهت دار یا بدون جهت)، رأس v یک نسل رأس u است اگر و فقط اگر در زمان $d[u]$ که عمل جستجو رأس u را کشف می‌کند، رأس v بتواند از رأس u در طول مسیری شامل تماماً رأسهای سفید، قابل دستیابی باشد.

اثبات ⇐: فرض کنید که v نسل u است. فرض کنید w یک رأس در مسیر بین u و v در درخت اول عمق باشد، بنابراین w یک نسل u می‌باشد. بنا به قضیه فرعی ۲۲.۸، $d[u] < d[w]$ و بنابراین w در زمان $d[u]$ سفید رنگ است.

⇒: فرض کنید رأس v از رأس u در زمان $d[u]$ در طول مسیری شامل تماماً رئوس سفید قابل دستیابی باشد، اما v نسل u در درخت اول عمق نمی‌شود. بدون از دست دادن کلیت فرض کنید هر رأس دیگری در مسیر، نسل u شود. (در غیر این صورت، فرض کنید v نزدیکترین رأس به u در مسیر باشد که نسل u نمی‌شود). فرض کنید w رأس ماقبل v در مسیر است، بنابراین w نسل u است (w و u ممکن است در واقع یک رأس باشند) و بنا به قضیه فرعی ۲۲.۸، $f[w] \leq f[u]$ توجه کنید که v باید بعد از کشف u و اماقبل از این که بررسی w خاتمه یابد کشف شود. بنابراین $d[u]<d[v]<f[w]\leq f[u]$ آن گاه قضیه ۲۲.۷ دلالت می‌کند بر این که بازه $[d[v],f[v]]$ تماماً در داخل بازه $[d[u] , f[u]]$ قرار دارد. روی هم رفته بنا به قضیه فرعی ۲۲.۸، v باید نسل u باشد. ■

طبقه‌بندی یال‌ها

ویژگی جالب دیگر جستجوی اول عمق این است که جستجو برای دسته بندی یال‌های گراف ورودی $G = (V,E)$ می‌تواند استفاده شود. این طبقه بندی یال‌ها می‌تواند جهت جمع کردن اطلاعات مهمی درباره گراف استفاده گردد. برای مثال در بخش بعد خواهیم دید که گراف جهت‌دار، بدون دور است اگر و فقط اگر در جستجوی اول عمق یال‌های عقب روند حاصل نگردد (لم ۲۲.۱۱).

بر حسب جنگل اول عمق G_π که به وسیله جستجوی اول عمق بر روی گراف G تولید می‌شود می‌توانیم چهار نوع یال تعریف کنیم.

۱. *یال‌های درختی*[1]، یال‌ها در جنگل اول عمق G_π می‌باشند. یال (u,v) یک یال درختی است اگر v اولین بار با بررسی یال (u,v) کشف شده باشد.

۲. *یال‌های عقب رونده*[2]، یال‌های (u,v) هستند که رأس u را به رأس جد یعنی v در درخت اول عمق متصل می‌کنند. خود حلقه‌ها که ممکن است در گراف‌های جهت‌دار رخ دهند یال‌های عقب رونده در نظر گرفته می‌شوند.

۳. *یال‌های جلو رونده*[3]، یال‌های غیر درختی (u,v) هستند که رأس u را به رأس نسل v در درخت اول عمق متصل می‌کنند.

۴. *یال‌های متقابل*[4] همه یال‌های دیگر هستند. این یالها می‌توانند بین رئوس در یک درخت اول عمق قرار گیرند البته تا زمانی که یک رأس جد دیگری نیست، یا می‌توانند بین رئوس در درخت‌های اول عمق مختلف قرار گیرند.

در شکل ۲۲.۴ و ۲۲.۵ یال‌ها برای مشخص کردن نوعشان برچسب دهی شده‌اند. شکل (c) ۲۲.۵ نشان می‌دهد چطور گراف شکل (a) ۲۲.۵ می‌تواند دوباره رسم شود به‌طوریکه همه یال‌های درختی و یال‌های جلو رونده در درخت اول عمق رو به پایین شوند و همه یال‌های عقب رونده رو به بالا شوند. هر گرافی می‌تواند مجدداً به این سبک رسم شود.

الگوریتم DFS می‌تواند تغییر داده شود تا همان‌طور که با یال‌ها مواجه می‌گردد آنها را دسته بندی کند. ایده کلیدی این است که هر یال (u,v) می‌تواند به وسیله رنگ رأس v طبقه بندی شود که این رأس وقتی یال اولین بار بررسی می‌شود مورد دستیابی قرار می‌گیرد (به جز از این که یال‌ها جلو رونده و متقابل از هم متمایز نمی‌شوند):

۱. $WHITE$ یک یال درختی را مشخص می‌کند،

۲. $GRAY$ یک یال عقب رونده را مشخص می‌کند،و

۳. $BLACK$ یال‌های جلو رونده یا متقابل را مشخص می‌کند.

حالت اول مستقیماً از مشخصه الگوریتم حاصل می‌شود. برای حالت دوم مشاهده می‌شود که رئوس خاکستری همیشه زنجیره‌ای خطی از نسل‌ها را شکل می‌دهند که با پشته احضارهای فعال $DFS\text{-}VISIT$ متناظر است؛ تعداد رئوس خاکستری یکی بیشتر از عمق در جنگل اول عمق رأسی که اخیراً کشف شده می‌باشد. بررسی همیشه از عمیق‌ترین رأس خاکستری پیش می‌رود، بنابراین یک یال که به رأس خاکستری دیگری دسترسی دارد به یک جد دسترسی دارد. سومین حالت، حالت‌های ممکن باقی مانده را در بر می‌گیرد؛ حالت سوم می‌تواند چنین نمایش داده شود که یال (u,v) یال جلو رونده است اگر $d[u]<d[v]$ و یک یال متقابل است اگر $d[u]>d[v]$ (تمرین ۴ – ۲۲.۳ را ملاحظه نمایید.)

---

1. Tree edge
2. back edge
3. forward edge
4. cross edge

در یک گراف بدون جهت ممکن است در نوع طبقه بندی ابهام وجود داشته باشد چون (u,v) و (v,u) در واقع یک یال هستند. در چنین حالتی، یال به عنوان نوع اول در لیست طبقه بندی که به کـار رفتـه دسته‌بندی می‌شود. به‌طور معادل (تمرین ۵ - ۲۲.۳ را ملاحظه نمایید) یال با توجه به این که کدام یک از یال‌های (u,v) یا (v,u) ابتدا در طی اجرای الگوریتم ملاقات شوند دسته بندی می‌گردد. اکنون نشـان می‌دهیم که یال‌های متقابل و جلو رونده هرگز در جستجوی اول عمق گراف بدون جهت رخ نمی‌دهند.

قضیه ۲۲.۱۰

در جستجوی اول عمق گراف بدون جهت G هر یال G یک یال درختی یا یک یال عقب رونده است.

اثبات فرض کنید (u,v) یک یال دلخواه باشد و بدون از دست دادن کلیت فرض کنید که $d[u]<d[v]$، آن گاه v باید کشف و خاتمه یابد قبل از آن که بررسی u را خاتمه دهیم (زمانی که u خاکستری است)، چون v در لیست همجواری u می‌باشد. اگر یال (u,v) ابتدا در جهت u به v بررسی شود، آن گاه v تا آن زمان کشف نشده (سفید) است. برای دیگر موارد این یال در جهت v به u بررسی شده است. بنابراین (u,v) یک یال درختی می‌شود. اگر (u,v) ابتدا از جهت v به سمت u بررسی شود آن گاه (u,v) یک یال عقب رونده است چون u در زمانی که یال اولین بار بررسی می‌شود هنوز خاکستری است. ■

چندین کاربرد این قضایا را در بخش‌های بعدی خواهیم دید.

تمرین‌ها

۱ - ۲۲.۳ یک جدول 3×3 با بر چسب‌های ستون و سطر $WHITE$، $GRAY$ و $BLACK$ بسازید. در هر خانه (i,j) تعیین کنید در هر لحظه در طول جستجوی اول عمق یک گراف جهت‌دار آیا یک یال از رأسی با رنگ i به رأسی با رنگ j وجود دارد یا خیر. برای هر یال ممکن مشخص کنید که این یال چه نوع یالی می‌تواند باشد، یک چنین جدولی برای جستجوی اول عمق یک گراف بدون جهت بسازید.

۲ - ۲۲.۳ نشان دهید جستجوی اول عمق چگونه بر روی گراف شکل ۲۲.۶ عمل می‌کند. فرض کنید که حلقه for خطوط ۷ - ۵ روال DFS رئوس را به ترتیب الفبایی در نظر می‌گیرد و فرض کنید که هر لیست همجواری به صورت الفبایی مرتب شده است. زمان‌های کشف و خاتمه برای هر رأس و طبقه بندی هر یال را نمایش دهید.

۳ - ۲۲.۳ ساختار پرانتزی جستجوی اول عمق نمایش داده شده در شکل ۲۲.۵ را نشان دهید.

۴ - ۲۲.۳ نشان دهید که یال (u,v)

a. یک یال درختی یا یک یال جلو رونده است اگر و فقط اگر $d[u] < d[v] < f[v] < f[u]$

b. یک یال عقب رونده است اگر و فقط اگر $d[v] < d[u] < f[u] < f[v]$ و

c. یک یال متقابل است اگر و فقط اگر $d[v] < f[v] < d[u] < f[u]$

۲۲.۳ - ۵ نشان دهید در یک گراف بدون جهت طبقه بندی کردن یال به عنوان یال درختی یا یال عقب رونده با توجه به این که (u,v) یا (v,u) در طی جستجو اول عمق اول ملاقات شده است با طبقه بندی کردن آن یال با توجه به اولویت نوع‌ها در طرح طبقه بندی معادل است.

۲۲.۳ - ۶ روال DFS را با استفاده از پشته برای حذف بازگشت، بازنویسی نمایید.

۲۲.۳ - ۷ برای این فرض که اگر مسیری از u به v در گراف جهت دار وجود داشته باشد و اگر در جستجوی اول عمق گراف G، $d[u]<d[v]$، آن گاه v نتیجه u در جنگل اول عمق حاصل می‌باشد، مثال نقضی ارائه دهید.

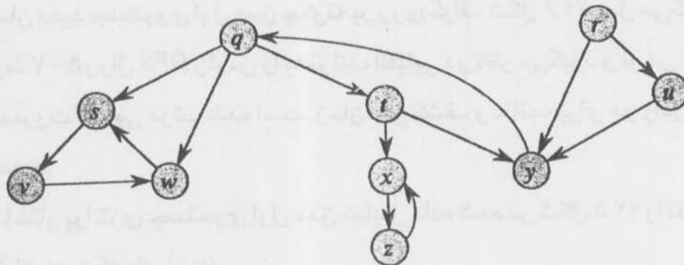
۲۲.۳ - ۸ مثال نقضی ارائه دهید برای این فرض که اگر مسیری از u به v در گراف جهت‌دار G وجود داشته باشد آن گاه هر جستجوی اول عمق باید $d[v] \leq f[u]$ را نتیجه دهد.

۲۲.۳ - ۹ شبه کد جستجوی اول عمق راطوری تغییر دهید تا هر یال در گراف جهت‌دار G را همراه با نوع یال مربوطه چاپ کند. نشان دهید چه تغییراتی در صورتی که G بدون جهت باشد لازم است.

۲۲.۳ - ۱۰ توضیح دهید که چگونه رأس u گراف جهت‌دار می‌تواند در درخت اول عمقی که تنها شامل u است خاتمه یابد، حتی اگر چه رأس u هر دو یال‌های ورودی و خروجی را در G داشته باشد.

۲۲.۳ - ۱۱ نشان دهید که جستجوی اول عمق گراف بدون جهت G می‌تواند جهت مشخص کردن اجزای همبند گراف G استفاده شود، و اینکه جنگل اول عمق شامل تعداد درخت برابر با تعداد اجزای همبند می‌باشد. به‌طور دقیقتر نشان دهید چگونه جستجوی اول عمق را تغییر دهیم تا اینکه به هر رأس v یک بر چسب صحیح $cc[v]$ بین ۱ تا k اختصاص داده شود که k تعداد اجزای همبندی گراف G می‌باشد، بطوریکه $cc[u] = cc[v]$ اگر و فقط اگر u و v در یک جزء همبند یکسان واقع باشند.

۲۲.۳-۱۲* گراف جهت‌دار $G=(V,E)$ همبند یک‌طرفه [1] است اگر $u \rightsquigarrow v$ دلالت کند بر این که حداکثر یک مسیر ساده از u به v برای همه رئوس $v \in V$ و u وجود دارد. یک الگوریتم کارا جهت مشخص کردن این که آیا گراف همبند یک‌طرفه می‌باشد یا خیر ارائه دهید.



شکل ۲۲.۶ گراف جهت‌دار برای استفاده در تمرین‌های ۲ - ۲۲.۳ و ۲ - ۲۲.۵.

---

1. singly connected

۲۲.۴ مرتب‌سازی موضعی

این بخش نشان می‌دهد چگونه جستجوی اول عمق می‌تواند برای اجرای مرتب‌سازی موضعی یک گراف بدون دور جهتدار که برخی اوقات "*dag*" نامیده می‌شود استفاده گردد.

مرتب‌سازی موضعی [1] $G=(V,E)$ که یک *dag* است یک ترتیب خطی از همه رئوس آن می‌باشد بطوریکه اگر G شامل یک یال (u,v) باشد آن گاه u قبل از v در ترتیب ظاهر شود. (اگر گراف بدون دور نباشد آن گاه هیچ ترتیب خطی وجود ندارد)، مرتب‌سازی موضعی گراف می‌تواند به صورت یک ترتیب از رئوس‌اش در راستای یک خط افقی در نظر گرفته شود به طوریکه همهٔ یالهای جهت‌دار از چپ به راست هستند. بنابراین مرتب‌سازی موضعی متفاوت از مرتب‌سازی‌های معمول مطالعه شده در بخش ۲ کتاب است.

گراف‌های بدون دور جهت‌دار در کاربردهای زیادی جهت تعیین کردن تقدم بین رخدادها استفاده می‌شوند، شکل ۲۲.۷ یک مثال ارائه می‌دهد که ناشی می‌شود از هنگامی که پرفسور *Bumstead* در صبح لباس می‌پوشد. پرفسور باید برخی از لباس‌ها را قبل از بقیهٔ لباس‌ها بپوشد (برای مثال، جوراب‌ها را قبل از کفس‌ها بپوشد)، برخی از اقلام و کارها ممکن است در هر ترتیبی انجام شود (به عنوان مثال پوشیدن جوراب‌ها و شلوار). یال جهتدار (u,v) در *dag* شکل ۲۲.۷ *(a)* تعیین می‌کند که لباس u باید قبل از لباس v پوشیده شود بنابراین مرتب‌سازی موضعی این *dag* یک ترتیب برای پوشیدن لباس ارائه می‌دهد. شکل ۲۲.۷ *(b)*، *dag* مرتب شده به صورت موضعی را به صورت ترتیبی از رئوس در راستای یک خط افقی نمایش می‌دهد، بطوریکه همه یال‌های جهت‌دار از چپ به راست می‌باشند.

الگوریتم ساده زیر به طور موضعی *dag* را مرتب می‌کند.

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times f[v] for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

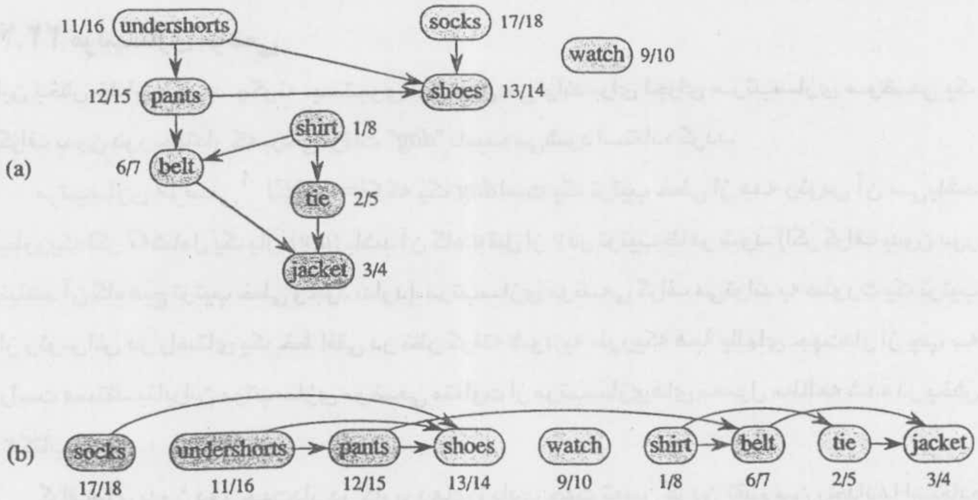
شکل ۲۲.۷ *(b)* نشان می‌دهد چطور رئوس مرتب شده به صورت موضعی در ترتیب معکوس زمان‌های خاتمه‌شان ظاهر می‌گردند.

می‌توانیم مرتب‌سازی موضعی را در زمان $\Theta(V+E)$ اجرا کنیم، چون جستجوی اول عمق زمان $\Theta(V+E)$ را صرف می‌کند و زمان $O(1)$ برای درج هر یک از $|V|$ رأس به جلوی لیست پیوندی صرف می‌شود.

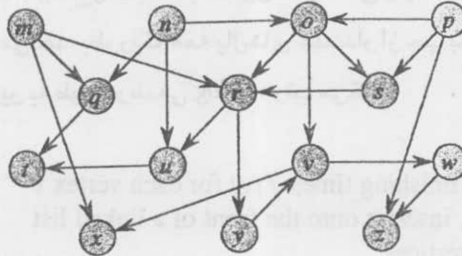
درستی الگوریتم را با استفاده از لم کلیدی زیر اثبات می‌کنیم که این لم گراف‌های بدون دور را توصیف می‌کند.

---

1. *topological sort*



شکل ۲۲.۷ *(a)* پرفسور *Bumstead* به صورت موضعی لباس هایش را هنگامی که آنها را می‌پوشد مرتب می‌کند. هر یال جهتدار *(u,v)* به این معنی است که لباس *u* باید قبل از لباس *v* پوشیده شده باشد زمان‌های خاتمه و کشف که از جستجوی اول عمق حاصل می‌شوند در کنار رأس مذکور قرار گرفته است. *(b)* همان گراف که به صورت موضعی مرتب شده است. رئوس آن از چپ به راست به ترتیب کاهش زمان خاتمه مرتب شده‌اند. توجه کنید که همه یال‌های جهت‌دار از چپ به راست هستند.



شکل ۲۲.۸ یک *dag* برای مرتب‌سازی موضعی

لم ۲۲.۱۱

گراف جهت‌دار G، بدون دور است اگر و فقط اگر از جستجوی اول عمق G هیچ یال عقب رونده حاصل نگردد.

اثبات :⇐فرض کنید یک یال عقب‌رونده (u,v) وجود دارد. آن گاه رأس v یک جد رأس u در جنگل اول عمق است. بنابراین یک مسیر از v به u در G وجود دارد و یال عقب رونده (u,v) دور را کامل می‌کند.

$\Leftarrow$: فرض کنید G شامل دور c باشد. نشان می‌دهیم که جستجوی اول عمق گراف G منجر به ایجاد یک یال عقب رونده می‌گردد. فرض کنید v اولین رأسی باشد که در c کشف شده است و (u,v) یال ماقبل در c باشد. در زمان $d[v]$ رئوس c مسیری از رئوس سفید از v به u تشکیل می‌دهند. بنا به قضیه مسیر سفید رأس u نسل رأس v در جنگل اول عمق می‌شود بنابراین (u,v) یک یال عقب رونده است. ■

قضیه ۲۲.۱۲

$TOPOLOGICAL\text{-}SORT(G)$ مرتب‌سازی موضعی گراف جهتدار بدون دور G را تولید می‌کند.

اثبات فرض کنید که DFS روی dag مفروض $G(V,E)$ اجرا می‌شود تا زمان‌های خاتمه رئوس آن را تعیین کند. کافی است نشان دهیم برای هر جفت رأس مجزا $u,v \in V$ اگر یک یال در G از u به v موجود باشد آن گاه $f[v]<f[u]$. در نظر بگیرید یال (u,v) به وسیله $DFS(E)$ مرور می‌گردد. وقتی این یال مرور می‌شود v نمی‌تواند خاکستری باشد، چون آنگاه v باید یک جد u و (u,v) باید یک یال عقب‌رونده می‌بود که با لم ۲۲.۱۱ در تناقض قرار داد. بنابراین v باید سیاه‌رنگ یا سفیدرنگ باشد. اگر v سفید باشد نسل u می‌شود و بنابراین $f[v]<f\{u]$.

اگر v سیاه باشد آن گاه تا حالا خاتمه یافته است. بنابراین $f[v]$ قبلاً مقداردهی شده است. از آنجا که هنوز در حال مرور u هستیم، اکنون باید برچسب زمان را به $f[u]$ انتساب دهیم و بنابراین زمانی که برچسب‌دهی زمانی صورت گیرد خواهیم داشت $f[v]<f[u]$ لذا برای هر یال (u,v) در dag داریم $f[v]<f[u]$، که قضیه را ثابت می‌کند. ■

تمرین‌ها

۲۲.۴-۱ ترتیب رئوس تولید شده به وسیلهٔ $TOPOLOGICAL\text{-}SORT$ را روی گراف شکل ۲۲.۸ با در نظر گرفتن فرضیات تمرین ۲۲.۳-۲ نشان دهید.

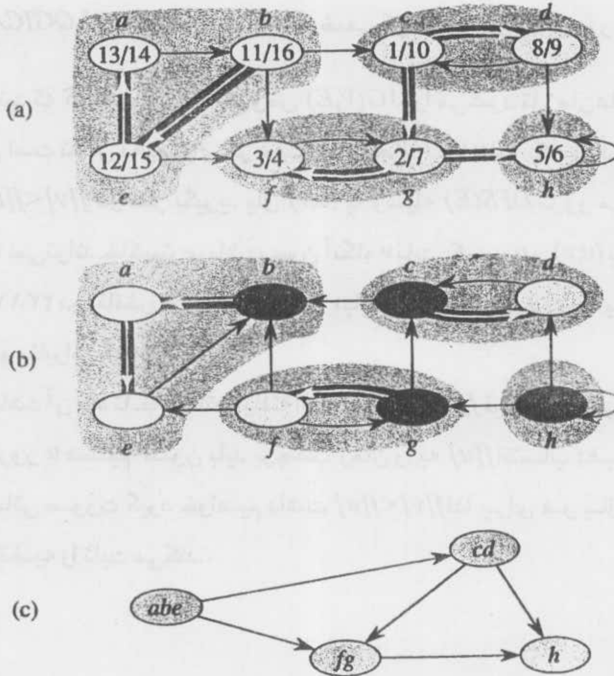
۲۲.۴-۲ یک الگوریتم با زمان خطی ارائه دهید که به عنوان ورودی گراف بدون دور جهت‌دار $G=(V,E)$ و دو رأس s و t را دریافت کند و تعداد مسیرهای موجود از s به t در گراف G را برگرداند. برای مثال در گراف بدون دور جهتدار شکل ۲۲.۸ دقیقاً چهار مسیر از رأس p به رأس v وجود دارد: pov, $poryv$, $posryv$ و $psryv$. (الگوریتم شما تنها لازم است که مسیرها را بشمارد و نیازی به لیست کردن آنها نیست).

۲۲.۴-۳ الگوریتمی ارائه دهید که مشخص کند آیا گراف بدون جهت داده شده $G=(V,E)$ دارای دور می‌باشد یا خیر؟ الگوریتم شما باید مستقل از $|E|$ در زمان $O(V)$ اجرا گردد.

۲۲.۴-۴ اثبات کنید یا رد کنید: اگر گراف جهت‌دار G شامل دور باشد آن گاه $TOPOLOGICAL\text{-}SORT(G)$ ترتیبی از رأس‌ها را ایجاد می‌کند که تعداد یال‌های "bad" که با ترتیب

تولید شده متناقض می‌باشند را مینیمم کند.

۵ـ۲۲.۴ روش دیگر انجام مرتب‌سازی موضعی بر روی گراف بدون دور جهتدار $G=(V,E)$ بدین صورت است که به طور مکرر یک رأس با درجه ورودی 0 را پیدا کرده و آن را خروجی قرار داده و از گراف به همراه یال‌های خروجی‌اش حذف می‌کند. چگونگی پیاده‌سازی این ایده را که در زمان $O(V+E)$ اجرا می‌گردد توضیح دهید. چه اتفاقی برای این الگوریتم روی می‌دهد اگر G دارای دور باشد.



شکل ۲۲.۹ *(a)* گراف جهتدار G و اجزای همبند قوی گراف G که به صورت ناحیه‌های سایه زده شده نشان داده شده است. هر رأس با زمان کشف و خاتمه‌اش برچسب‌دهی می‌شود. یال‌های درختی سایه زده می‌شوند *(b)* گراف GT ترانهاده گراف G، جنگل اول عمق که در خط ۳ روال *STRONGLY-CONNECTED-COMPONENT* محاسبه می‌شود با یال‌های درختی سایه زده شده نشان داده شده است. هر جزء همبند قوی متناظر با یک درخت اول عمق است. رئوس b، c، g و h که به شکل تیره‌تر سایه زده شده‌اند ریشه‌های درخت‌های اول عمق، تولید شده به وسیلهٔ جستجوی اول عمق گراف G^{T} می‌باشند. *(c)* جزء بدون دور گراف G^{SCC} به وسیلهٔ منقبض کردن همهٔ یال‌ها درون هر جزء همبند قوی گراف G بدست می آید در هر جزء تنها یک تک رأس باقی بماند.

۲۲.۵ اجزای همبند قوی

اکنون یک کاربرد کلاسیک جستجوی اول عمق را در نظر می‌گیریم: تجزیه گراف جهتدار به اجزای همبند قوی‌اش. این بخش نشان می‌دهد که چگونه این تجزیه را به وسیله دو جستجوی اول عمق انجام

دهیم. بسیاری از الگوریتم‌ها که با گراف‌های جهتدار کار می‌کنند با چنین تجزیه‌ای آغاز می‌شوند. بعد از تجزیه، الگوریتم به طور جداگانه روی هر جزء همبند قوی اجرا می‌گردد. آن گاه جواب‌ها مطابق با ساختار اتصال بین اجزاء ترکیب می‌گردند. می‌دانیم که جزء همبند قوی گراف جهتدار $G=(V,E)$ یک مجموعه ماکزیمال از رأس‌های $C\subseteq V$ است به طوری که برای هر جفت از رئوس u و v در C، هر دوی $u \leadsto v$ و $v \leadsto u$ را داریم؛ به عبارت دیگر رئوس u و v از یکدیگر قابل دستیابی هستند. شکل ۹. ۲۲ یک نمونه را نشان می‌دهد.

الگوریتم جهت پیدا کردن اجزای همبند قوی گراف جهتدار $G=(V,E)$ از ترانهاده G استفاده می‌کند که در تمرین ۳-۲۲.۱ به صورت گراف $G^T=(V,E^T)$ تعریف شد به طوریکه $E^T=\{(u,v):(v,u)\in E\}$. به عبارت دیگر E^T شامل یال هایی با جهت عکس یال‌های E می‌باشد. با استفاده از نمایش لیست همجواری G زمان ایجاد G^T برابر با $O(V+E)$ است. جالب است که مشاهده می‌شود G و G^T دقیقاً دارای اجزای همبند قوی یکسان هستند: u, v در G از یکدیگر قابل دستیابی هستند اگر و فقط اگر آنها در G^T از همدیگر قابل دستیابی باشند. شکل (b) ۲۲.۹ ترانهاده گراف شکل (a)۲۲.۹ را به همراه اجزاء همبند قوی سایه زده شده را نمایش می‌دهد.

الگوریتم زیر با زمان خطی (به عبارتی $\Theta(V+E)$ ـ زمانی) اجزای همبند قوی گراف جهتدار $G=(V+E)$ را با استفاده از دو جستجوی اول عمق یک‌بار بر روی G و یک بار بر روی G^T محاسبه می‌کند.

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times f[u] for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing f[u] (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

ایده این الگوریتم از ویژگی کلیدی گراف جزئی $G^{scc}=(V^{scc}, E^{scc})$ حاصل می‌شود، که آن را به صورت زیر تعریف می‌کنیم. فرض کنید که G اجزای همبند قوی $C_1, C_2, \dots, C_k$ را دارد. مجموعه رئوس V^{scc} برابر است با $\{v_1,v_2,\dots,v_k\}$ و این مجموعه شامل یک رأس v_i برای هر جزء همبند قوی C_i گراف G می‌باشد. یال $(v_i,v_j)\in E^{scc}$ وجود دارد اگر G شامل یال جهت دار (x,y) برای $x\in C_i$ و $y\in C_j$ باشد. به روش دیگری توجه کنید، به وسیله منقبض کردن همه یال‌ها که رأس‌های ضمنی‌شان در داخل یک جزء همبند قوی یکسان G قرار دارند، گراف حاصل گراف G^{scc} می‌باشد. شکل (c) ۲۲.۹ گراف جزئی گراف شکل (a) ۲۲.۹ را نشان می‌دهد.

ویژگی کلیدی این است که گراف جزئی یک dag است که لم زیر به این ویژگی دلالت دارد.

لم ۲۲.۱۳

فرض کنید C و C' اجزای همبند قوی مجزا در گراف جهت‌دار $G = (V,E)$ باشند و $u,v \in C$ و $u',v' \in C'$

و فرض کنید که مسیر $u \leadsto u'$ در G وجود دارد. آن گاه نمی‌تواند مسیری از $v' \leadsto v$ در G موجود باشد.

اثبات اگر یک مسیر از $v' \leadsto v$ در G وجود داشته باشد آن گاه مسیرهای $u \leadsto u' \leadsto v'$ و $v' \leadsto v \leadsto u$ در G وجود دارند، بنابراین u و v' از یکدیگر قابل دستیابی هستند که به موجب آن فرضی که C و C' اجزای همبند مجزا هستند نقض می‌گردد. ■

خواهیم دید که با در نظر گرفتن رئوس به ترتیب کاهش زمان خاتمه در دومین جستجوی اول عمق، که در اولین جستجوی اول عمق محاسبه شده است، در اصل داریم رئوس گراف جزئی را به ترتیب مرتب شده موضعی ملاقات می‌کنیم (که هر کدام با یک جزء همبند قوی گراف متناظر می‌باشند).

چون *STRONGLY-CONNECTED-COMPONENT* دو بار جستجوی اول عمق را اجرا می‌کند، زمانی که در مورد $d[u]$ یا $f[u]$ بحث می‌کنیم امکان ابهام وجود دارد. در این بخش، این مقادیر همیشه به زمان‌های کشف و خاتمه که به وسیله فراخوانی اول *DFS* در خط *1* محاسبه شده اشاره می‌کنند.

علامت نویسی برای زمان‌های خاتمه و کشف را به مجموعه‌های رئوس بسط می‌دهیم. اگر $U \subseteq V$ باشد آن گاه تعریف می‌کنیم

$$d(U) = min_{u \in U} \{d[u]\} \quad \text{و} \quad f(U) = max_{u \in U} \{f[u]\}$$

به عبارت دیگر $d(U)$ و $f(U)$ زودترین زمان کشف و دیرترین زمان خاتمه برای هر رأس در U می‌باشند.

لم زیر و قضیه فرعی آن یک ویژگی کلیدی مربوط به اجزاء همبند قوی و زمان‌های خاتمه در اولین جستجوی اول عمق را ارائه می‌دهند.

لم ۲۲.۱۴

فرض کنید C و C' اجزاء همبند قوی مجزا در گراف جهت‌دار $G = (V,E)$ باشند. فرض کنید که یال $(u,v) \in E$ که $u \in C$ و $v \in C'$، وجود دارد، آن گاه $f(C) > f(C')$

اثبات دو حالت بسته به اینکه کدامیک از دو جزء همبند قوی C و C' ابتدا در طی جستجو اول عمق کشف شده باشد وجود دارد.

اگر $d(C) < d(C')$ فرض کنید x اولین رأس کشف شده در C باشد. در زمان $d[x]$ همه رئوس در C و C' سفید رنگ هستند. یک مسیر در G از x به هر رأس در C تشکیل شده از همه رئوس سفید وجود

دارد. چون $(u,v) \in E$ می‌باشد، برای هر رأس $w \in C'$ در زمان $d[x]$ یک مسیر از x به w در G تشکیل شده از رئوس سفید وجود دارد: $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$. بنا به قضیه مسیر سفید همه رئوس در C و C' نسل‌های x در درخت اول عمق می‌شوند. بنا به قضیه فرعی ۲۲.۸، $f[x] = f(C) > f(C')$.

اگر در عوض داشته باشیم $d(C) > d(C')$ فرض کنید y اولین رأسی کشف شده در C' باشد در زمان $d[y]$ همه رئوس در C' سفید هستند و یک مسیر در G از y به هر رأس در C' تشکیل شده از تنها رئوس سفید وجود دارد. بنا به قضیه مسیر سفید همه رئوس در C' نسل‌های y در درخت اول عمق می‌شوند و بنا به قضیه فرعی ۲۲.۸، $f[y] = f(C')$ در زمان $d[y]$ همه رئوس در C سفید هستند. از آنجایی که یال (u,v) از C به C' وجود دارد، لم ۲۲.۱۳ دلالت می‌کند بر این که مسیری از C' به C نمی‌تواند وجود داشته باشد، از این رو هیچ رأسی در C از y قابل دستیابی نیست. بنابراین در زمان $f[y]$ همه رئوس در C هنوز سفید هستند. لذا برای هر رأس $w \in C$ داریم $f[w] > f[y]$ که دلالت دارد بر اینکه $f(C) > f(C')$ ■

قضیه فرعی زیر به ما می‌گوید که هر یال در G^T که بین اجزای همبند قوی متفاوت قرار دارد از جزء همبند با زمان خاتمه زودتر به سمت جزء با زمان خاتمه دیرتر (در اولین جستجوی اول عمق) قرار می‌گیرد.

قضیه فرعی ۲۲.۱۵

فرض کنید C و C' اجزای همبند قوی مجزا در گراف جهت‌دار $G = (V,E)$ باشند. فرض کنید که یال $(u,v) \in E^T$ وجود دارد که در آن $u \in C$ و $v \in C'$ است. آن گاه $f(C) < f(C')$

اثبات از آنجایی که $(u,v) \in E^T$ داریم، $(v,u) \in E$. چون اجزای همبند قوی گراف G و گراف G^T یکسان هستند، لم ۲۲.۱۴ دلالت بر این دارد که $f(C) < f(C')$.

قضیه فرعی ۲۲.۱۵ کلید فهم علت کار کردن روال *STRONGLY-CONNECTED-COMPONENTS* را فراهم می‌کند. اجازه دهید بررسی کنیم چه اتفاقی وقتی دومین جستجوی اول عمق را بر روی G^T اجرا می‌کنیم رخ می‌دهد. با جزء همبند قوی C که زمان خاتمه آن $f(C)$ ماکزیمم است شروع می‌کنیم. جستجو از یک رأس $x \in C$ شروع می‌شود و تمامی رئوس در C را ملاقات می‌کند. بنا به قضیه فرعی ۲۲.۱۵، از جزء همبند C به هیچ یک از اجزای همبند دیگر هیچ یالی در G^T وجود ندارد و بنابراین جستجو از x رئوس در دیگر اجزاء همبند را ملاقات نخواهد کرد لذا درخت مشتق شده از x دقیقاً شامل رئوس C می‌باشد. بعد از این که همه رئوس در C ملاقات شدند، جستجو در خط ۳ یک رأس از یک جزء همبند قوی C' را انتخاب می‌کند که زمان خاتمه آن یعنی $f(C')$ در بین همه اجزاء همبند به غیر از جزء همبند C ماکزیمم است. دوباره جستجو همه رئوس در C' را ملاقات خواهد کرد، اما بنا به قضیه فرعی ۲۲.۱۵ یال‌ها در G^T که تا کنون ملاقات کرده‌ایم و از

C' به سمت اجزاء دیگر همبند هستند، باید به C بروند. در کل وقتی جستجوی اول عمق گراف G^T در خط ۳ یک جزء همبند قوی را ملاقات می‌کند هر یال خروجی از آن جزء، باید به سمت اجزایی که تا کنون ملاقات شده‌اند قرار داشته باشد. بنابراین هر درخت اول عمق دقیقاً یک جزء همبند قوی خواهد بود. قضیه زیر این بحث را به صورت رسمی بیان می‌کند.

قضیه ۲۲.۱۶

STRONGLY-CONNECTED-COMPONENTS(G) به درستی اجزاء همبند قوی گراف جهت‌دار G را محاسبه می‌کند.

اثبات اثبات به وسیله استقراء بر روی تعداد درخت‌های اول عمق که در جستجوی اول عمق گراف G^T در خط ۳ پیدا می‌شوند انجام می‌گردد. بدین شکل که رئوس هر درخت یک جزء همبند قوی را شکل می‌دهند. فرض استقراء آن است که اولین k درخت تولید شده در خط ۳ اجزاء همبند قوی هستند. پایه استقراء وقتی $k = 0$ است، بدیهی است. در نظر بگیریم $(k + 1)$ امین درخت تولید شود. فرض کنید ریشه این درخت رأس u باشد و فرض کنید u در جزء همبند قوی C واقع باشد. به خاطر چگونگی انتخاب ریشه‌ها در خط ۳ جستجوی اول عمق برای هر جزء همبند قوی C' به جز C که باید ملاقات شود $f[u]=f(C) > f(C')$ بنا به فرض استقراء در زمانی که جستجو u را ملاقات می‌کند همه رئوس دیگر در C سفید هستند. بنابراین بنا به قضیه مسیر سفید همه رئوس دیگر در C نسل‌های u در درخت اول عمق آن می‌باشند. بعلاوه بنا به فرض استقراء و قضیه فرعی ۲۲.۱۵ هر یال در G^T که از C خارج می‌شود باید به سمت اجزاء همبند قوی که تا کنون ملاقات شده است باشد. بنابراین هیچ رأسی در هر جزء همبند قوی به جز C، نسل u در طی جستجوی اول عمق G^T نخواهد بود. لذا رئوس درخت اول عمق در G^T که از u مشتق شده‌اند دقیقاً یک جزء همبند قوی را شکل می‌دهند که گام استقرا و اثبات را کامل می‌کند. ■

اینک به شیوه‌ای دیگر عملکرد دومین جستجوی اول عمق را بررسی می‌کنیم. گراف جزئی $(G^T)^{scc}$ از گراف G^T را در نظر بگیریم. اگر هر جزء همبند قوی ملاقات شده در دومین جستجوی اول عمق را به یک رأس $(G^T)^{scc}$ نگاشت کنیم، رئوس $(G^T)^{scc}$ در یک ترتیب مرتب شده موضعی معکوس ملاقات می‌شوند. اگر یال‌های گراف $(G^T)^{scc}$ را معکوس کنیم گراف $((G^T)^{scc})^T$ را بدست می‌آوریم. چون $((G^T)^{scc})^T = G^{scc}$ (تمرین ۴ - ۲۲.۵ را ملاحظه کنید)، دومین جستجوی اول عمق رئوس G^{scc} را به ترتیب مرتب شده موضعی ملاقات می‌کند.

تمرین‌ها

۱ - ۲۲.۵ اگر یک یال جدید به گراف اضافه شود تعداد اجزاء همبند قوی گراف چگونه می‌تواند تغییر

کند؟

۲۲.۵-۲ نشان دهید روال *STRONGLY-CONNECTED-COMPONENTS* چگونه بر روی شکل ۲۲.۶ کار می‌کند بخصوص زمان‌های خاتمه محاسبه شده در خط ۱ جنگل ایجاد شده در خط ۳ را نمایش دهید. فرض کنید که حلقه خطوط ۷-۵ روال *DFS* رئوس را به ترتیب الفبا در نظر می‌گیرد و این که لیستهای همجواری به ترتیب الفبایی قرار دارند.

۲۲.۵-۳ پرفسور *Deaver* ادعا می‌کند که الگوریتم برای اجزاء همبند قوی می‌تواند با استفاده از گراف اولیه (به جای ترانهاده) در دومین جستجوی اول عمق و بررسی رئوس به ترتیب افزایش زمان‌های خاتمه ساده‌تر شود، آیا ادعای پرفسور درست می‌باشد؟

۲۲.۵-۴ ثابت کنید برای هر گراف جهت‌دار G داریم $((G^T)^{scc})^T = G^{scc}$. به عبارت دیگر ترانهاده گراف جزئی G^T با گراف جزئی G یکسان است.

۲۲.۵-۵ یک الگوریتم با زمان $O(V+E)$ برای محاسبه گراف جزئی جهت‌دار $G = (V,E)$ ارائه دهید. مطمئن شوید که حداکثر یک یال بین دو رأس در گراف جزئی تولید شده توسط الگوریتم شما وجود دارد.

۲۲.۵-۶ گراف جهت‌دار $G = (V,E)$ داده شده است. توضیح دهید چطور گراف دیگر $G' = (V',E')$ را ایجاد کنیم بطوریکه G' (a) دارای اجزاء همبند قوی یکسانی با G باشد G' (b) دارای گراف جزئی یکسانی با G باشد و E' (c) تا حد ممکن کوچک است. یک الگوریتم سریع برای محاسبه G' ارائه دهید.

۲۲.۵-۷ گراف جهت‌دار $G(v,E)$ «نیمه همبند»[1] گفته می‌شود اگر برای همه جفت رئوس $u,v \in V$ داشته باشیم $u \leadsto v$ یا $v \leadsto u$. یک الگوریتم مؤثر برای تعیین این که آیا G نیمه همبند است یا خیر ارائه دهید. اثبات کنید که الگوریتم شما درست است و زمان اجرای الگوریتم خود را تحلیل کنید.

مسائل

۲۲ - ۱ طبقه بندی یال‌ها به وسیله جستجوی اول سطح

جنگل اول عمق یال‌های گراف را به صورت یال درختی، یال عقب رونده، یال جلو رونده و یال‌های متقابل طبقه بندی می‌کند. درخت اول سطح همچنین می‌تواند جهت طبقه بندی یال‌های قابل دستیابی از مبدأ جستجو به همان چهار رده استفاده شود.

a. اثبات کنید که در جستجوی اول سطح یک گراف بدون جهت ویژگی‌های زیر برقرار می‌گردد:

۱. یال‌های عقب رونده و یال‌های جلو رونده وجود ندارند.

1. *semiconnected*

۲. برای هر یال درختی (u,v)، داریم $d[v] = d[u] + 1$

۳. برای هر یال متقابل (u,v)، داریم $d[v] = d[u]$ یا $d[v] = d[u] + 1$

b. اثبات کنید که در جستجوی اول سطح گراف جهت‌دار ویژگی‌های زیر حفظ می‌گردد:

۱. هیچ یال جلو رونده‌ای وجود ندارد.

۲. برای هر یال درختی (u,v) داریم $d[v] = d[u] + 1$

۳. برای هر یال متقابل (u,v) داریم $d[v] \leq d[u] + 1$

۴. برای هر یال عقب رونده (u,v) داریم $0 \leq d[v] \leq d[u]$.

۲ - ۲۲ نقاط انفصال، پل‌ها و اجزای دو همبندی

فرض کنید $G = (V,E)$ یک گراف همبند بدون جهت باشد. یک *نقطه انفصال* G [1] رأسی از G است که با حذف آن رأس، گراف نا همبند می‌شود. یک پل[2] گراف G یالی است که اگر حذف گردد، گراف ناهمبند می‌گردد. جزء دو همبندی [3] گراف G مجموعه ماکزیممی از یال‌ها است به‌طوریکه هر دو یال از مجموعه، در یک دور ساده مشترک قرار گیرند. شکل ۲۲.۱۰ این تعاریف را نشان می‌دهد. می‌توانیم نقاط انفصال، پل‌ها و اجزاء دو همبندی را با استفاده از جستجوی اول عمق مشخص کنیم. فرض کنید $G_{\pi} = (V,E_{\pi})$ درخت اول عمق گراف G باشد.

a. اثبات کنید که ریشه G_{π} یک نقطه انفصال G است اگر و فقط اگر این ریشه در G_{π} حداقل دو فرزند داشته باشد.

b. فرض کنید v یک رأس غیر ریشه در G_{π} باشد. اثبات کنید که v یک نقطه انفصال G است اگر و فقط اگر v دارای فرزند s باشد به‌طوریکه هیچ یال عقب رونده‌ای از s یا هر یک از نسل‌های s به جدی از v موجود نباشد.

c. فرض کنید

$$low[v] = \min \begin{cases} d[v] , \\ d[w] : \text{(u, w) یک یال عقب رونده برای نسل } u \text{ از رأس } v \text{ است} \end{cases}$$

نشان دهید که چگونه $low[v]$ برای همهٔ رئوس $v \in V$ در زمان $O(E)$ محاسبه می‌شود.

نشان دهید چگونه همه نقاط انفصال در زمان $O(E)$ محاسبه می‌گردند.

اثبات کنید که یک یال از G پل است اگر و فقط اگر در یک دور ساده از G قرار نگیرد.

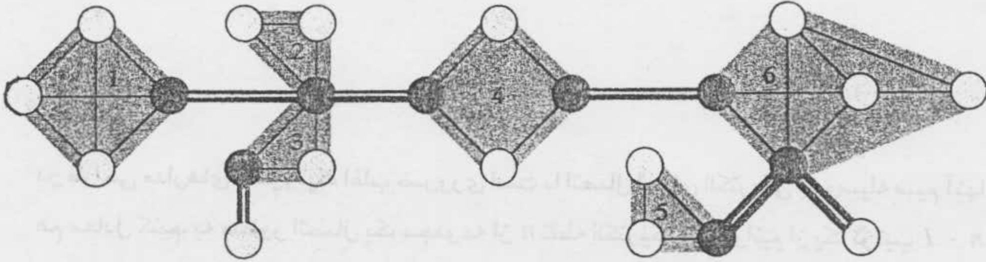
---

1. articulation point
2. bridge
3. biconnected component

نشان دهید چگونه همه پل‌های G در زمان $O(E)$ محاسبه می‌گردند.

اثبات کنید که اجزاء دو همبندی گراف G یال‌های غیر پل G را افراز می‌کنند.

یک الگوریتم با زمان $O(E)$ برای بر چسب دهی هر یال e گراف G با مقدار صحیح مثبت $bcc[e]$ ارائه دهید به‌طوریکه $bcc[e] = bcc[e']$ اگر و فقط اگر e و e' در یک جزء دو همبندی یکسان واقع باشند.



شکل ۲۲.۱۰ نقاط انفصال، پل‌ها و اجزاء دو همبندی گراف همبند بدون دور برای استفاده در مسئله ۲ - ۲۲. نقاط انفصال رئوس سایه زده شده تیره می‌باشند. پل‌ها یال‌های سایه زده تیره هستند. اجزای دو همبندی یال‌های درون ناحیه سایه زده شده با اعداد bcc می‌باشند.

۲۲ - ۳ گردش اویلری[1]

گردش اویلری گراف همبند جهت‌دار $G = (V,E)$ یک دور است که هر یال از G دقیقاً یک بار پیموده شود اگر چه ممکن است رأس بیش از یک بار ملاقات شود.

نشان دهید G دارای گردش اویلری است اگر و فقط اگر برای هر رأس $v \in V$

$in\text{-}degree(v) = out\text{-}degree(v)$

یک الگوریتم با زمان $O(E)$ ارائه دهید که یک گردش اویلری G را در صورت وجود پیدا کند. (راهنمایی: دورهای از لحاظ یال مجزا را ادغام کنید.)

۲۲ - ۴ قابلیت دستیابی[2]

فرض کنید $G = (V,E)$ گراف جهت‌دار باشد به‌طوریکه هر رأس $u \in V$ به فرم زیر بر چسب دهی می‌شود.

$R(u) = \{v \in V : u \rightsquigarrow v\}$ مجموعه رئوسی است که از u قابل دستیابی می‌باشند. $min(u)$ را رأسی در $R(u)$ که بر چسب آن مینیمم است تعریف می‌کنیم. به عبارت دیگر $min(u)$ رأس v است بطوریکه $L(v) = min\ \{L(w): w \in R(u)\}$ یک الگوریتم با زمان $O(V+E)$ ارائه دهید که $min(u)$ را برای همه رأسهای $u \in V$ محاسبه نماید.

1. Euler tour
2. Reachability

فصل ۲۳ درخت پوشای مینیمم

در طراحی مدارهای الکترونیک اغلب ضروری است با اتصال اجزای الکتریکی به وسیله سیم آنها را با هم معادل کنیم. به منظور اتصال یک مجموعه از n نقطه الکتریکی، می‌توانیم از یک ترتیب $n - 1$ سیم استفاده کنیم که هر کدام دو نقطه را به هم متصل کنند. از میان همه این ترتیب‌ها، ترتیبی که از حداقل مقدار سیم استفاده می‌کند مطلوب‌ترین می‌باشد. می‌توانیم این مسئله سیم کشی را با یک گراف همبند بدون جهت $G = (V,E)$ مدل کنیم که V مجموعه نقاط اتصال الکتریکی و E مجموعه اتصال‌های ممکن بین هر جفت نقطه اتصال الکتریکی می‌باشد، و برای هر $(u,v) \in E$ وزن $w(u,v)$ را داریم کـه هـزینه اتصال (مقدار سیم مورد نیاز) u و v را معین می‌کند. آن گاه می‌خواهیم یک زیر مجموعه بدون دور $T \subseteq E$ را بیابیم که همه رئوس را به هم متصل کند و وزن کل آن یعنی

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

مینیمم باشد. از آنجایی که T بدون دور می‌باشد و همهٔ رئوس را به هم متصل می‌کند، باید درختی را که درخت *پوشا*[1] نامیده می‌شود شکل دهد، چون گراف G را پوشش می‌دهد. مسئله مشخص کردن درخت T را مسئله درخت *پوشای مینیمم*[۲] می‌نامیم شکل ۲۳.۱ یک مثال از گراف همبند و درخت پوشای مینیمم آن را نشان می‌دهد.

در این فصل دو الگوریتم برای حل مسئله درخت پوشای مینیمم را بـررسی مـی‌کنیم: الگـوریتم *Kruskal* و الگوریتم *Prim* هر کدام از این دو الگوریتم می‌تواند به سادگی با استفاده از *heap* دودویی معمولی در زمان $O(ElgV)$ اجرا شود. با استفاده از *heap* فیبوناچی الگوریتم *Prim* می‌تواند تسریع یابد و در زمان $O(E+VlgV)$ اجرا شود که اگر $|V|$ خیلی کوچکتر از $|E|$ باشد یک بهبود می‌باشد.

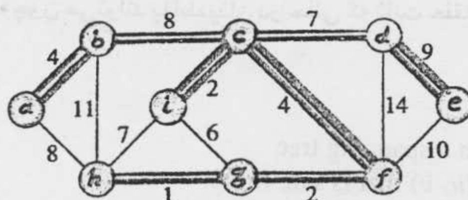
این دو الگوریتم مانند آنچه در فصل ۱۶ تعریف شد الگوریتم‌های حریصانه می‌باشند. در هر گام از

---

1. *spanning tree*

۲. عبارت "درخت پوشای مینیمم" شکل مختصر عبارت "درخت پوشا با وزن مینیمم" می‌باشد، برای مثال تـعداد یـال‌ها در T را مـینیمم نمی‌کنیم چون همه درخت‌های پوشا دقیقاً $|V|-1$ یال دارند.

الگوریتم یکی از چندین انتخاب ممکن باید انتخاب شود. استراتژی حریصانه از بهترین انتخاب در هر لحظه حمایت می‌کند. این چنین استراتژی به‌طور کلی پیدا کردن جواب بهینه برای مسئله را ضمانت نمی‌کند.



شکل ۲۳.۱ درخت پوشای مینیمم برای یک گراف همبند. وزن یال‌ها نشان داده شده و یال‌ها در درخت پوشای مینیمم سایه زده شده‌اند. وزن کل درخت برابر *37* می‌باشد. این درخت پوشای مینیمم منحصر به فرد نیست: حذف یال *(b,c)* و جایگزین کردن آن با یال *(a,h)* منجر به درخت پوشای دیگری با وزن *37* می‌گردد.

اما در مسئله درخت پوشای مینیمم می‌توانیم ثابت کنیم که استراتژی حریصانه مشخص منجر به درخت پوشا با وزن مینیمم می‌گردد. اگر چه فصل جاری می‌تواند مستقل از فصل ۱۶ مطالعه شود، روش‌های حریصانه نشان داده شده در این جا کاربردهای کلاسیک از ایده‌های نظری که در فصل ۱۶ معرفی شده‌اند می‌باشند.

بخش ۲۳.۱ الگوریتم درخت پوشای مینیمم «عمومی»[1] را معرفی می‌کند که درخت پوشا با اضافه کردن یک یال در هر زمان رشد می‌کند. بخش ۲۳.۲ دو روش پیاده سازی الگوریتم عمومی را ارائه می‌دهد. الگوریتم اول *Krukal* شبیه الگوریتم اجزا همبند بخش ۲۱.۱ می‌باشد دومین الگوریتم یعنی *Prim* شبیه الگوریتم کوتاهترین مسیرهای *Dijkstra* است (بخش ۲۴.۳).

۲۳.۱ رشد درخت پوشای مینیمم

فرض کنید که گراف همبند بدون جهت $G = (V,E)$ با تابع وزن $W: E \to R$ را داریم و می‌خواهیم درخت پوشای مینیمم را برای G پیدا کنیم. دو الگوریتم که در این فصل در نظر می‌گیریم از شیوه حریصانه برای مسئله استفاده می‌کنند اگر چه در چگونگی به کار گرفتن این روش تفاوت دارند.

این استراتژی حریصانه به وسیله الگوریتم عمومی زیر به کار برده می‌شود، به‌طوریکه درخت پوشای مینیمم در هر زمان به اندازه یک یال رشد می‌کند. الگوریتم با حفظ ثابت حلقه زیر مجموعه A از یال‌ها را مدیریت می‌کند:

---

1. generic

قبل از هر تکرار، A یک زیر مجموعه از یک درخت پوشای مینیمم است.

در هر گام یال (u,v) را مشخص می‌کنیم که می‌تواند به A بدون تخطی از این ثابت اضافه شود، از این جهت که $A \cup \{(u,v)\}$ نیز زیر مجموعه درخت پوشای مینیمم است. ما این چنین یالی را یک **یال ایمن**[1] برای A می‌نامیم، چون می‌تواند با اطمینان در حالی که ثابت حلقه حفظ می‌شود اضافه شود.

```
GENERIC-MST(G, w)
1  A ← Ø
2  while A does not form a spanning tree
3      do find an edge (u, v) that is safe for A
4          A ← A ∪ {(u, v)}
5  return A
```

از ثابت حلقه به شکل زیر استفاده می‌شود:

مقدار دهی اولیه: بعد از خط ۱، مجموعه A به‌طور بدیهی ثابت حلقه را ارضاء می‌کند.

نگهداری: حلقه در خطوط ۴-۲ ثابت را با اضافه کردن فقط یک یال ایمن حفظ می‌کند.

خاتمه: همه یال‌های اضافه شده به A در درخت پوشای مینیمم واقع هستند و بنابراین مجموعه A که در خط ۵ برگردانده می‌شود باید درخت پوشای مینیمم باشد.

البته قسمت دشوار پیدا کردن یال ایمن در خط ۳ می‌باشد. یک مشکل وجود دارد چون زمانی که خط ۳ اجرا می‌شود، ثابت بیان می‌کند که درخت پوشای T وجود دارد بطوریکه $A \subseteq T$ در داخل بدنه حلقه *while*، A باید زیر مجموعه‌ای از T باشد و بنابراین باید یک یال $(u,v) \in T$ موجود باشد بطوریکه $(u,v) \notin A$ و (u,v) یک یال ایمن برای A باشد.

در باقی مانده این بخش یک قانون (قضیه ۲۳.۱) برای پیدا کردن یال‌های ایمن فراهم می‌کنیم، بخش بعد دو الگوریتم که این قانون را برای پیدا کردن یال‌های ایمن بطور مؤثر به کار می‌برند را توضیح می‌دهد.

در ابتدا به یکسری تعریف نیاز داریم. بریدگی[2] $(S, V-S)$ از گراف بدون جهت $G=(V,E)$ یک افراز از V می‌باشد. شکل ۲۳.۲ این نکته را نشان می‌دهد. می‌گوییم یال $(u,v) \in E$ از بریدگی $(S, V-S)$ عبور می‌کند[3] اگر یکی از نقاط پایانی آن در S و دیگری در $V-S$ باشد. می‌گوییم یک بریدگی با مجموعه A سازگاری[4] دارد اگر هیچ یالی در A با یال‌های این بریدگی تلاقی نکند. یک یال، *یال سبک*[5] عبور کننده از یک بریدگی است اگر وزنش در میان یال‌های عبور کننده از بریدگی مینیمم باشد. توجه داشته باشید که می‌تواند بیش از یک یال عبور کننده از بریدگی وجود داشته باشد. به‌طور کلی‌تر می‌گوییم یک یال،

---

1. *safe edge*
2. *cut*
3. *cross*
4. *respect*
5. *light edge*

یالی سبک است و ویژگی داده شده را ارضاء می‌کند اگر وزنش در بین یال هایی که ویژگی را ارضاء می‌کنند مینیمم باشد.

قانون برای شناختن یال‌های ایمن به وسیله قضیه زیر ارائه می‌شود.

شکل ۲۳.۲ دو شیوه نمایش بریدگی *(S,V-S)* گراف شکل ۲۳.۱. *(a)* رئوس در مجموعه S سیاه‌رنگ و رئوس در مجموعه *(V-S)* سفید رنگ می‌باشند. یال‌های عبور کننده از بریدگی یال‌هایی هستند که رئوس سفید را به رئوس سیاه متصل می‌کنند. یال *(d,c)* یال سبک منحصر به فرد عبور کننده از این بریدگی است. زیر مجموعه A از یال‌ها، سایه زده شده است. توجه داشته باشید که بریدگی *(S,V-S)* با مجموعه A سازگاری دارد. چون هیچ یالی از A از بریدگی عبور نمی‌کند *(b)* همان گراف با رئوس مجموعه S در چپ و رئوس مجموعه V-S در سمت راست. یک یال از بریدگی عبور می‌کند اگر رأس در سمت چپ را به رأس در سمت راست متصل کند.

قضیه ۲۳.۱

فرض کنید $G = (V,E)$ یک گراف همبند بدون جهت با تابع وزن w با مقادیر حقیقی تعریف شده بر روی E باشد، فرض کنید که A زیر مجموعه E باشد که در یک درخت پوشای مینیمم برای G قرار می‌گیرد. فرض کنید *(S,V-S)* یک بریدگی G باشد که با مجموعه A سازگاری دارد و *(u,v)* یال سبک عبور کننده از $(S\ ,\ V\ -\ S)$ باشد آن گاه یال *(u,v)* یال ایمن برای A است.

اثبات فرض کنید T درخت پوشای مینیمم باشد که شامل A است و فرض کنید که T شامل یال سبک *(u,v)* نباشد، چون اگر T شامل یال سبک باشد آن گاه اثبات را انجام داده‌ایم. درخت پوشای مینیمم T' را خواهیم ساخت که $A\cup\{(u,v)\}$ را با استفاده از تکنیک برش و الصاق شامل می‌شود. به وسیله آن نشان می‌دهیم که *(u,v)* یک یال ایمن برای A می‌باشد.

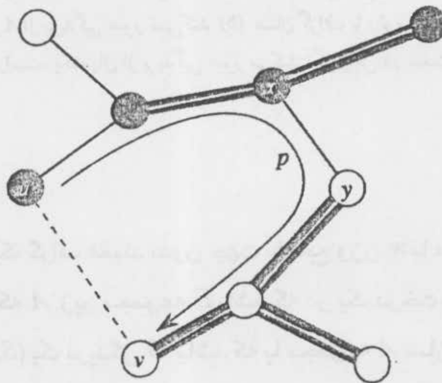
یال (u,v) با یال‌های روی مسیر p از u به v در T یک دور را ایجاد می‌کند، همان‌طور که در شکل ۲۳.۳ نشان داده شده است. از آنجایی که (u,v) در دو طرف مقابل بریدگی $(S,V-S)$ قرار دارند، حداقل یک یال در T روی مسیر p وجود دارد که از این بریدگی نیز عبور می‌کند. فرض کنید (x,y)، یک چنین یالی باشد. یال (x,y) در A نمی‌باشد چون بریدگی با مجموعه A سازگار است. از آنجایی که (x,y) در یک مسیر منحصر به فرد از u به v در T قرار دارد، حذف (x,y) درخت T را به دو جزء می‌شکند. اضافه کردن (u,v) این دو جزء را به هم متصل می‌کند تا درخت پوشای جدید $T'=T-\{(x,y)\}\cup\{(u,v)\}$ را شکل دهد.

نشان می‌دهیم که T' یک درخت پوشای مینیمم است. از آنجایی که (u,v) یک یال سبک عبور کننده از $(S,V-S)$ می‌باشد و (x,y) نیز از این بریدگی عبور می‌کند، $w(u,v) \leq w(x,y)$. بنابراین

$$
\begin{aligned}
w(T') &= w(T) - w(x,y) + w(u,v) \\
&\leq w(T) .
\end{aligned}
$$

اما T یک درخت پوشای مینیمم است به‌طوریکه $w(T) \leq w(T')$؛ بنابراین T' نیز باید درخت پوشای مینیمم باشد.

تنها باقی مانده نشان دهیم که (u,v) یک یال ایمن برای A است. داریم $A \subseteq T'$ چون $A \subseteq T$ و $(x,y) \notin A$؛ بنابراین $A \cup \{(u,v)\} \subseteq T'$ در نتیجه چون T' درخت پوشای مینیمم است، (u,v) یال ایمن برای A است. ■



شکل ۲۳.۳ اثبات قضیه ۲۳.۱. رأس‌ها در مجموعه S سیاه رنگ و رأس‌ها در مجموعه $V-S$ سفید رنگ می‌باشند. یال‌ها در درخت پوشای مینیمم T نمایش داده شده‌اند اما یال‌های گراف G نمایش داده نشده‌اند. یال‌های موجود در A سایه زده شده‌اند و (u,v) یک یال سبک عبور کننده از بریدگی $(S,V-S)$ است. یال (x,y) یک یال روی مسیر منحصر به فرد p در T از u به v است. درخت پوشای مینیمم T' که (u,v) را شامل می‌شود از حذف یال (x,y) از T و اضافه کردن یال (u,v) ایجاد شده است.

قضیه ۲۳.۱ یک درک بهتر از عملکرد الگوریتم *GENERIC-MST* بر روی گراف همبند $G = (V,E)$ را ارائه می‌دهد. همان‌طور که الگوریتم پیش می‌رود مجموعه A همیشه بدون دور است؛ در غیر این صورت درخت پوشای مینیمم شامل A دارای دور می‌باشد که با ویژگی درخت بودن متناقض است. در هر مرحله اجرای الگوریتم، گراف $G_A=(V,A)$ یک جنگل است و هر جزء همبند گراف G_A یک درخت است. (تعدادی از درخت‌ها ممکن است فقط شامل یک رأس باشند. به‌طور مثال وقتی که الگوریتم شروع می‌شود این حالت وجود دارد: A تهی می‌باشد و جنگل شامل $|V|$ درخت، یک درخت برای هر رأس، است.) علاوه بر این هر یال ایمن (u,v) برای A اجزاء مجزاء G_A را به هم متصل می‌کند چون $A \cup \{(u,v)\}$ باید بدون دور باشد.

حلقه خطوط ۴-۲ روال *GENERIC-MST*، تعداد $|V|-1$ بار اجرا می‌شود و هر یک از $|V|-1$ یال درخت پوشای مینیمم متوالیاً مشخص می‌شود. در ابتدا وقتی $A = \phi$ ، $|V|$ درخت در G_A وجود دارد و هر تکرار حلقه تعداد درخت‌ها را به اندازه 1 واحد کاهش می‌دهد. وقتی جنگل تنها شامل یک تک درخت شود الگوریتم خاتمه می‌یابد.

دو الگوریتم بخش ۲۳.۲ از قضیه فرعی زیر که از قضیه ۲۳.۱ حاصل شده است استفاده می‌کنند.

قضیه فرعی ۲۳.۲

فرض کنید $G=(V, E)$ یک گراف همبند بدون جهت با تابع وزن w با مقادیر حقیقی تعریف شده بر روی E باشد. فرض کنید A زیر مجموعه‌ای از E باشد که در یک درخت پوشای مینیمم G قرار گرفته باشد و فرض کنید $C = (V_c , E_c)$ یک جزء همبند (درخت) در جنگل $G_A=(V,A)$ است. اگر (u,v) یک یال سبک باشد که C را به جزء دیگری در G_A متصل می‌کند آن گاه (u,v) برای A یک یال ایمن است.

اثبات بریدگی $(V_c, V-V_c)$ با مجموعه A سازگار می‌باشد و (u,v) یک یال سبک برای این بریدگی است. بنابراین (u,v) برای A یال ایمن است.

تمرین‌ها

۱ - ۲۳.۱ فرض کنید (u,v) یک یال با وزن مینیمم در گراف G باشد. نشان دهید که (u,v) متعلق به یک درخت پوشای مینیمم G است.

۲ - ۲۳.۱ پرفسور *Sabatier* قضیه معکوس زیر را برای قضیه ۲۳.۱ حدس می‌زند. فرض کنید $G=(V,E)$ گراف همبند بدون جهت با تابع وزن w باشد که تابع وزن w با مقادیر حقیقی بر روی E تعریف شده است. فرض کنید A زیر مجموعه‌ای از E است که در یک درخت پوشای مینیمم برای G قرار گرفته است، فرض کنید $(S,V-S)$ یک بریدگی G است که با A سازگار است و فرض کنید (u,v) یک یال ایمن برای A و عبور کننده از $(S,V-S)$ می‌باشد. آن گاه (u,v) یک یال سبک برای این بریدگی است. با

ارائه چند مثال نقض نشان دهید که حدس پرفسور نادرست است.

۲۳.۱ - ۳ نشان دهید که اگر یال (u,v) در درخت پوشای مینیمم باشد آن گاه یک یال سبک می‌باشد که از یک بریدگی گراف عبور می‌کند.

۲۳.۱ - ۴ یک مثال ساده از یک گراف ارائه دهید بطوریکه مجموعه یال‌های

$\{(u,v)$: بریدگی $(S,V\text{-}S)$ وجود دارد بطوریکه (u,v) یال سبک عبور کننده از $(S,V\text{-}S)$ است$\}$

یک درخت پوشای مینیمم را شکل نمی‌دهد.

۲۳.۱ - ۵ فرض کنید e یال با وزن ماکزیمم در یک دور $G=(V,E)$ باشد. ثابت کنید یک درخت پوشای مینیمم از $G'=(V,E\text{-}\{e\})$ وجود دارد که درخت پوشای مینیمم G' نیز می‌باشد، به عبارت دیگر یک درخت پوشای مینیمم از G وجود دارد که شامل e نمی‌باشد.

۲۳.۱ - ۶ نشان دهید که گراف دارای درخت پوشای منحصر به فردی می‌باشد اگر برای هر بریدگی گراف، یال منحصر به فرد سبکی عبور کننده از بریدگی وجود داشته باشد. با ارائه مثال نقض نشان دهید که عکس این مطلب درست نمی‌باشد

۲۳.۱ - ۷ ثابت کنید اگر همه وزن‌های یال‌ها مثبت باشند آن گاه هر زیر مجموعه از یال‌ها که همه رئوس را به هم متصل می‌کند و دارای وزن کلی مینیمم است باید یک درخت باشد. یک مثال ارائه دهید تا نشان دهد که نتیجه‌ای یکسان در زمانی که اجازه می‌دهید برخی وزن‌ها غیر مثبت باشند حاصل نمی‌گردد.

۲۳.۱ - ۸ فرض کنید T درخت پوشای مینیمم گراف G باشد و فرض کنید L لیست مرتب شده وزن یالهای T باشد. نشان دهید که برای هر درخت پوشای مینیمم دیگر T' از G، لیست L نیز یک لیست مرتب شده از وزن‌های یال‌های T' می‌باشد.

۲۳.۱ - ۹ فرض کنید T درخت پوشای مینیمم گراف $G=(V,E)$ و V' زیر مجموعه V باشد. فرض کنید T' زیر گراف T حاصل شده از V' و G' زیر گراف G بدست آمده از V' است. نشان دهید اگر T' همبند باشد آن گاه T' درخت پوشای مینیمم G' است.

۲۳.۱ - ۱۰ گراف G و درخت پوشای مینیمم T داده شده است. فرض کنید که وزن یکی از یال‌ها در T را کاهش می‌دهیم. نشان دهید که T هنوز یک درخت پوشای مینیمم برای G می‌باشد. به‌طور رسمی‌تر فرض کنید T درخت پوشای مینیمم برای G با وزن‌های یال داده شده به وسیله تابع وزن w باشد. یک یال $(x,y) \in T$ و عدد مثبت k را انتخاب کنید و تابع وزن w' را به صورت زیر تعریف کنید.

$$w'(u,v) = \begin{cases} w(u,v) & \text{if } (u,v) \neq (x,y)\,, \\ w(x,y) - k & \text{if } (u,v) = (x,y)\,. \end{cases}$$

نشان دهید که T یک درخت پوشای مینیمم برای G با وزن یال‌های بدست آمده از w' می‌باشد.

۲۳.۱ - ۱۱* گراف G و درخت پوشای مینیمم T داده شده است. فرض کنید که وزن یکی از یال‌ها که

در T نمی‌باشد را کاهش دهیم. یک الگوریتم برای یافتن درخت پوشای مینیمم در گراف تغییر کرده ارائه دهید.

۲۳.۲ الگوریتم‌های *Prim* و *Kruskal*

دو الگوریتم درخت پوشای مینیمم تعریف شده در این بخش ساخته شده از الگوریتم عمومی می‌باشند. هر کدام از آنها قانون خاصی را برای مشخص کردن یال ایمن در خط ۳ روال *GENERIC-MST* استفاده می‌کنند. در الگوریتم *Kruskal* مجموعه A یک جنگل است. یال ایمن اضافه شده به A همیشه یال با وزن حداقل در گراف G است که دو جزء مجزا را به هم متصل می‌کند. در الگوریتم *Prim* مجموعه A یک درخت را شکل می‌دهد. یال ایمن اضافه شده به A همیشه یال با وزن حداقل می‌باشد که درخت را به رأسی که در درخت نمی‌باشد متصل می‌کند.

الگوریتم *Kruskal*

الگوریتم *Kruskal* بر پایه الگوریتم عمومی درخت پوشای مینیمم ارائه شده در بخش ۲۳.۱ می‌باشد. این الگوریتم یال ایمن را برای اضافه کردن به جنگل در حال رشد به وسیله پیدا کردن یال (u,v) با حداقل وزن از میان یال‌هایی که هر دو درخت در جنگل را به هم متصل می‌کنند، می‌یابد.

فرض کنید C_1 و C_2 هر دو درخت که به وسیله (u,v) متصل هستند را نشان می‌دهند. از آنجایی که (u,v) باید یال سبک باشد که C_1 را به درختی دیگر متصل می‌کند، قضیه فرعی ۲۳.۲ دلالت می‌کند بر این که (u,v) یال ایمن برای C_1 می‌باشد. الگوریتم *Kruskal* یک الگوریتم حریصانه است چون در هر مرحله این الگوریتم به جنگل یک یال با کمترین وزن ممکن را اضافه می‌کند.

پیاده سازی الگوریتم *Kruskal* شبیه الگوریتمی که اجزاء همبند گراف را در بخش ۲۱.۱ محاسبه می‌کرد، است. این الگوریتم از ساختمان داده‌های مجموعه‌های جدا از هم برای نگهداری چندین مجموعه جدا از هم از عناصر استفاده می‌کند. هر مجموعه، رئوس یک درخت از جنگل جاری را شامل می‌شود. عمل *FIND-SET(u)* عضو نماینده مجموعه شامل u را بر می‌گرداند. بنابراین می‌توانیم به وسیله بررسی این که آیا *FIND-SET(u)* مساوی *FIND-SET(v)* است مشخص کنیم که آیا رئوس u و v متعلق به درخت یکسانی می‌باشند. ترکیب درخت‌ها توسط روال *UNION* انجام می‌گیرد.

```
MST-KRUSKAL(G, w)
1  A ← Ø
2  for each vertex v ∈ V[G]
3       do MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w
```

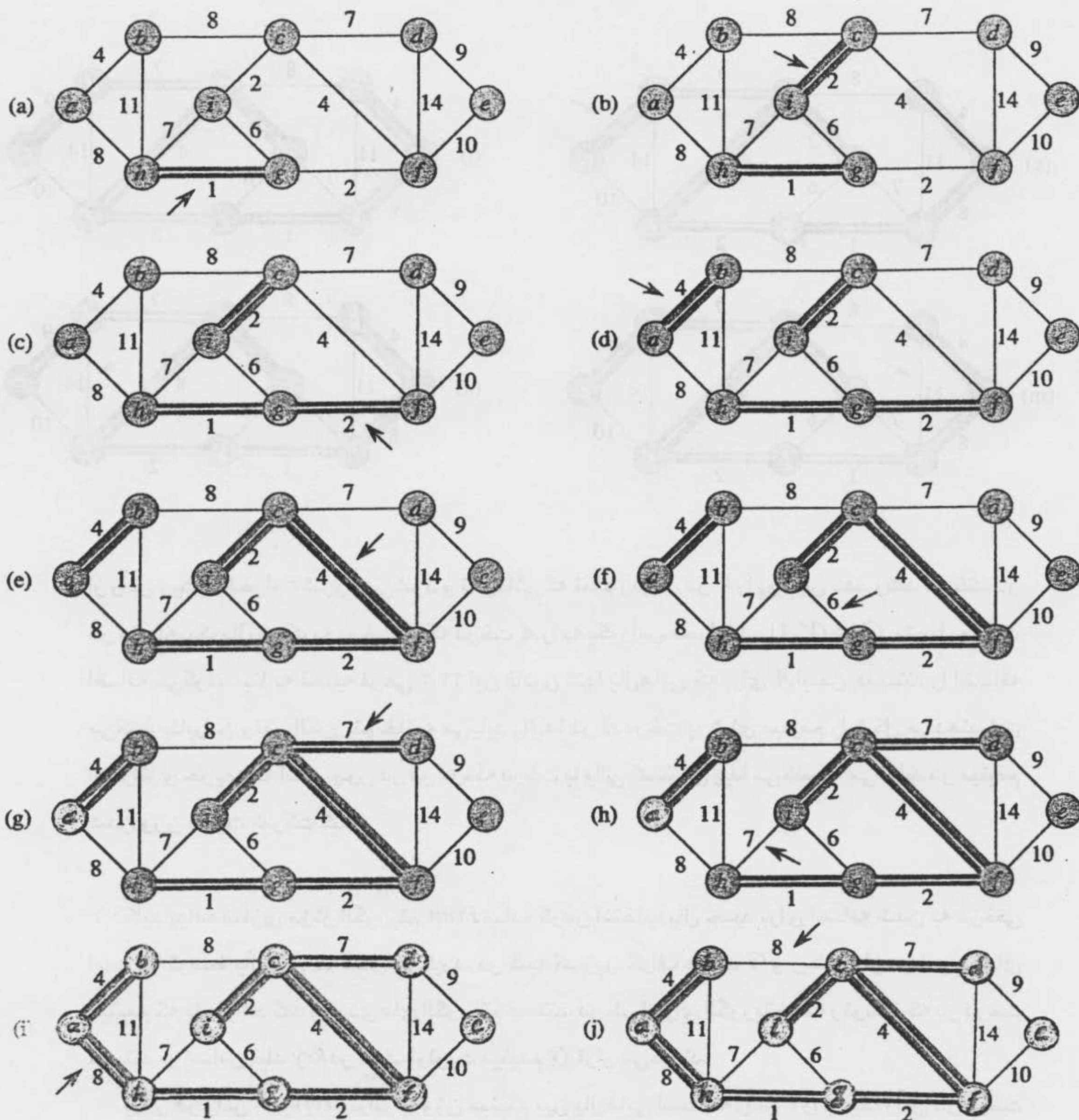
```
5  for each edge (u, v) ∈ E, taken in nondecreasing order by weight
6      do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

الگوریتم *Kruskal* مانند آنچه در شکل ۲۳.۴ نشان داده شده کار می‌کند. خطوط ۳ - ۱ مجموعه A را با یک مجموعه تهی مقدار دهی اولیه می‌کند و $|V|$ درخت که هر کدام شامل یک رأس است را ایجاد می‌کند. در خط ۴ یال‌های E به ترتیب غیر نزولی بر حسب وزن مرتب می‌شوند. حلقه *for* در خطوط ۸ - ۵، برای هر یال (u,v) بررسی می‌کند آیا نقاط انتهایی u و v عضو یک درخت یکسان هستند یا خیر. اگر آنها عضو یک درخت یکسان باشند آن گاه یال (u,v) نمی‌تواند بدون ایجاد دور به جنگل اضافه گردد و از این یال صرف نظر می‌شود. در غیر این صورت دو رأس عضو دو درخت متفاوت هستند. در این حالت یال (u,v) در خط ۷ به A افزوده می‌شود و رئوس دو درخت در خط ۸ ادغام می‌گردند.

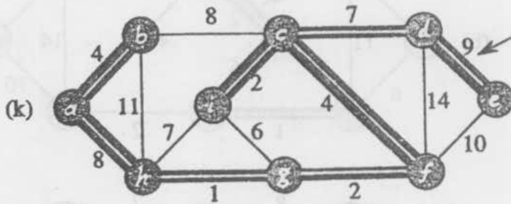
زمان اجرای الگوریتم *Kruskal* برای گراف $G=(V,E)$ بستگی به پیاده سازی ساختمان داده مجموعه جدا از هم دارد. فرض خواهیم کرد که پیاده سازی جنگل مجموعه‌های جدا از هم بخش ۲۱.۳ با دو روش مکاشفه‌ای واحد سازی بر حسب مرتبه و فشرده سازی مسیر باشد، چون به‌طور مجانبی سریعترین پیاده سازی شناخته شده است. مقدار دهی اولیه مجموعه A در خط ۱ زمان $O(1)$ را صرف می‌کند و زمان لازم برای مرتب کردن یال‌ها در خط ۴ برابر $O(E\lg E)$ می‌باشد. (هزینه $|V|$ عمل *MAKE-SET* در حلقه *for* خطوط ۳ - ۲ را به زودی محاسبه خواهیم کرد) حلقه *for* خطوط ۸ - ۵، تعداد $O(E)$ عمل *UNION* و *FIND-SET* را بر روی جنگل مجموعه جدا از هم انجام می‌دهد. به همراه $|V|$ عمل *MAKE-SET*، این اعمال زمان کل $O((V+E)\alpha(V))$ را صرف می‌کنند، که α یک تابع با رشد بسیار کم که در بخش ۲۱.۴ تعریف شده است، می‌باشد. بخاطر این که فرض شده G همبند است داریم $|E| \geq |V|-1$ و بنابراین اعمال مجموعه جدا از هم $O(E\alpha(V))$ را صرف می‌کنند. بعلاوه چون $\alpha(|V|)=O(\lg V)=\alpha(\lg E)$، زمان کل اجرای الگوریتم *Kruskal* برابر $O(E\lg E)$ می‌باشد. با مشاهده این که $|E| < |V|^2$، داریم $\lg|E|=O(\lg V)$ و بنابراین می‌توانیم زمان اجرای الگوریتم *Kruskal* را به صورت $O(E\lg V)$ بیان کنیم.

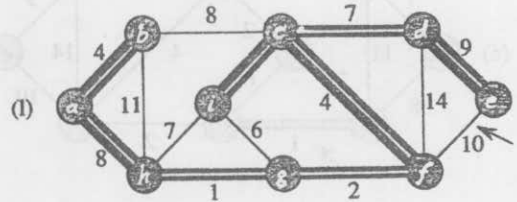
الگوریتم *Prim*

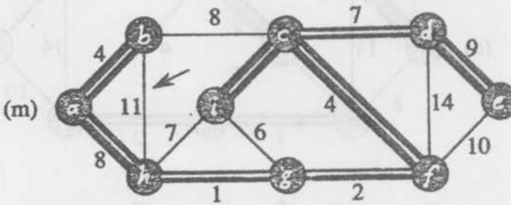
مانند الگوریتم *Kruskal*، الگوریتم *Prim* حالت خاصی از الگوریتم عمومی درخت پوشای مینیمم بخش ۲۳.۱ است. الگوریتم *Prim* برای پیدا کردن کوتاهترین مسیر در گراف شبیه الگوریتم *Dijkstra* که در بخش ۲۴.۳ خواهیم دید عمل می‌کند. الگوریتم *Prim* دارای این ویژگی می‌باشد که یال‌ها در مجموعه A همیشه یک درخت را شکل می‌دهند. همان‌طور که در شکل ۲۳.۵ نشان داده شده است، درخت

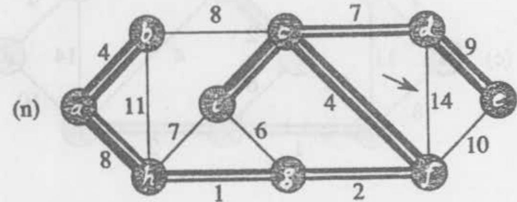


شکل ۲۳.۴ اجرای الگوریتم *Kruskal* روی گراف شکل ۲۳.۱. یال‌های سایه زده شده متعلق به جنگل A در حال رشد می‌باشد. یال‌ها توسط الگوریتم بصورت مرتب شده به ترتیب وزن در نظر گرفته می‌شوند. یک پیکان به یال در نظر گرفته شده در هر مرحله اشاره می‌کند. اگر یال دو درخت مجزا در جنگل را به هم متصل کند درخت حاصل به جنگل اضافه می‌گردد که به موجب آن دو درخت ادغام می‌شوند.









از رأس ریشه دلخواه r شروع می‌شود و تا زمانی که تمام رئوس در V را پوشش دهد رشد می‌کند. در هر مرحله، یک یال سبک به درخت A که درخت A را به یک رأس مجزا شده $G_A=(V,A)$ متصل می‌کند اضافه می‌گردد. بنا به قضیه فرعی ۲۳.۲ این قانون تنها یال‌هایی که برای A ایـمن هسـتند را اضـافه می‌کند؛ بنابراین وقتی الگوریتم خاتمه می‌یابد یال‌ها در A درخت پوشای مینیمم را شکل می‌دهند. این استراتژی حریصانه است چون در هر مرحله درخت با یالی گسترش پیدا می‌کند که می‌تواند در مینیمم شدن وزن درخت شرکت کند.

کلید پیاده سازی مؤثر الگوریتم *Prim* ساده کردن انتخاب یال جدید برای اضافه شدن به درختی است که توسط یال‌های A شکل می‌گیرد. در شبه کد زیر، گراف همبند G و ریشه r از درخت پوشای مینیمم که باید رشد کند ورودی‌های الگوریتم هستند. در طی اجرای الگوریتم همه رئوسی که در درخت نیستند بر اساس فیلد *key* در صف اولویت مینیمم Q قرار می‌گیرند.

برای هر رأس v، $key[v]$ برابر با وزن مینیمم بین یال‌هایی است که رأس v را به یک رأس در درخت متصل می‌کنند؛ بنا به قرار داد $key\ [v]\ =\ \infty$ اگر چنین یالی وجود نداشته باشد. فیلد $\pi[v]$ پدر v در درخت را دارا می‌باشد: در طی الگوریتم، مجموعه A از *GENERIC-MST* بصورت ضمنی به شکل

$$A = \{(v, \pi[v]) : v \in V - \{r\} - Q\} .$$

نگه داشته می‌شود.

وقتی الگوریتم خاتمه می‌یابد، صف اولویت مینیمم Q خالی مـی‌باشد؛ بـنابرایـن درخت پـوشای مینیمم A برای G برابر است با

$$A = \{(v, \pi[v]) : v \in V - \{r\}\} .$$

```
MST-PRIM(G, w, r)
 1  for each u ∈ V[G]
 2      do key[u] ← ∞
 3         π[u] ← NIL
 4  key[r] ← 0
 5  Q ← V[G]
 6  while Q ≠ Ø
 7      do u ← EXTRACT-MIN(Q)
 8         for each v ∈ Adj[u]
 9             do if v ∈ Q and w(u, v) < key[v]
10                   then π[v] ← u
11                        key[v] ← w(u, v)
```

الگوریتم *Prim* همانند شکل ۲۳.۵ کار می‌کند. خطوط ۵-۱ کلید هر رأس را ∞ قرار می‌دهند. (به جز برای ریشه *r* که کلید مقدار صفر قرار می‌گیرد تا اولین رأسی باشد که پردازش خواهد شد)، پدر هر رأس را برابر *NIL* قرار می‌دهد و صف اولویت Q را برای شامل شدن همهٔ رئوس مقدار دهی اولیه می‌کند. الگوریتم ثابت حلقه سه قسمتی زیر را نگه می‌دارد:

قبل از هر تکرار حلقه *while* خطوط ۱۱-۶،

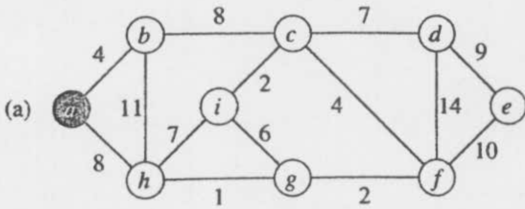
۱. $A = \{(v,\pi[v]): v \in V-\{r\}-Q\}$.

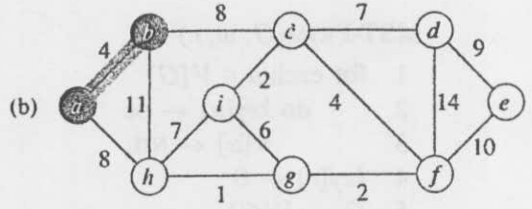
۲. رئوسی که از قبل در درخت پوشای مینیمم واقع شده‌اند رئوس $V-Q$ می‌باشند.

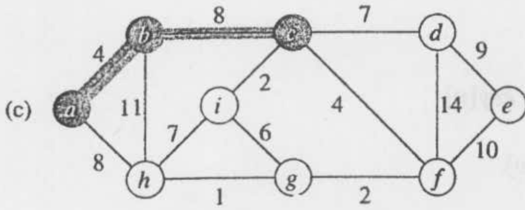
۳. برای همه رئوس $v \in Q$، اگر $\pi[v] \neq NIL$ آن گاه $key[v] < \infty$ و $key[v]$ وزن یال سبک $(v,\pi[v])$ می‌باشد که v را به رأسی که از قبل در درخت پوشای مینیمم واقع شده متصل می‌کند.

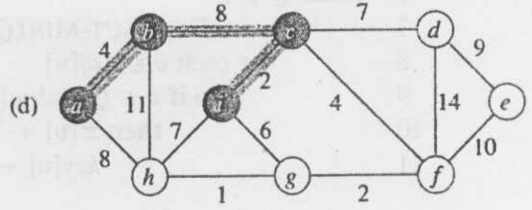
خط ۷ رأس $u \in Q$ بر روی یک یال سبک عبور کننده از بریدگی $(V-Q,Q)$ را مشخص می‌کند (به جز اولین تکرار که به خاطر اجرای خط ۴، $u=r$ می‌شود). حذف u از مجموعه Q آن را به مجموعه رئوس $V-Q$ در درخت اضافه می‌کند. بنابراین $(u,\pi[u])$ به A اضافه می‌شود. حلقه *for* خطوط ۱۱ - ۸، فیلدهای π و *key* برای هر رأس v همجوار u که در درخت نیست را بروز رسانی می‌کند. این بروزرسانی بخش سوم ثابت حلقه را حفظ می‌کند.

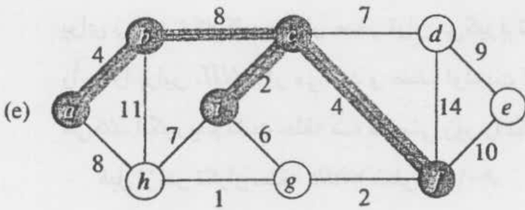
کارایی الگوریتم *prim* به چگونگی پیاده سازی صف اولویت Q بستگی دارد. اگر Q به صورت *min-heap* دودویی پیاده سازی شود. (فصل ۶ را ببینید) می‌توانیم از روال *BUILD-MIN-HEAP* در خطوط ۵ - ۱ برای مقدار دهی اولیه در زمان $O(V)$ استفاده کنیم. بدنه حلقه *while*، $|V|$ بار اجرا می‌شود. و چون هر عمل *EXTRACT-MIN* زمان $O(lgV)$ را صرف می‌کند زمان کل برای همه فراخوانی‌های *EXTRACT-MIN* برابر $O(VlgV)$ است. حلقه *for* در خطوط ۱۱ - ۸ روی هم رفته $O(E)$ بار اجرا می‌شود، چون مجموعه طول همه لیست‌های همجواری برابر $2|E|$ می‌باشد. در داخل

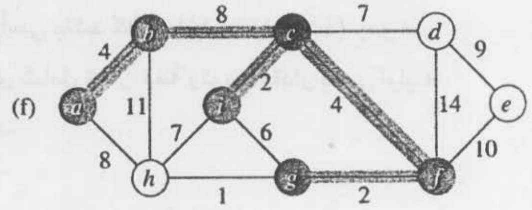


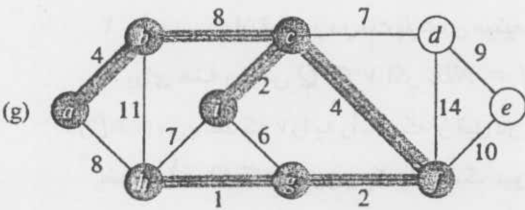


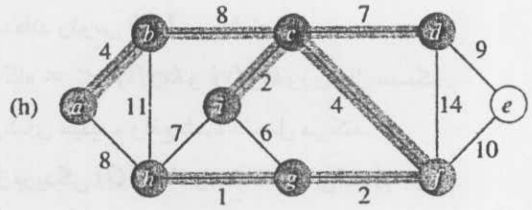


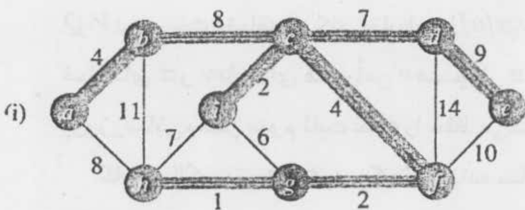












شکل ۲۳.۵ اجرای الگوریتم *Prim* بر روی گراف شکل ۲۳.۱. رأس، ریشه a می‌باشد. یال‌های سایه زده شده در درخت در حال رشد هستند و رئوس در درخت، سیاه می‌باشند. در هر گام الگوریتم رئوس در درخت یک بریدگی گراف را مشخص می‌کنند و یال سبک عبور کننده از بریدگی به درخت اضافه می‌شود. در دومین گام به‌طور مثال، الگوریتم انتخابی بین اضافه کردن یال (b,c) یا یال (a,h) به درخت را دارد، چون هر دو یال سبک و عبور کننده از بریدگی هستند.

حلقه *for* بررسی برای عضویت در Q در خط ۹ می‌تواند در زمان ثابت با نگهداری یک بیت برای هر رأس که بیان می‌کند آیا آن رأس در Q قرار دارد یا خیر، پیاده سازی می‌شود و وقتی کـه رأس از Q حذف شود بروز رسانی بیت انجام می‌گیرید. انتساب در خط ۱۱ شامل یک عمل *DECREASE-KEY* ضمنی بر روی *min-heap* است که می‌تواند در *min-heap* دودویی در زمان $O(lgV)$ پیاده سـازی شود. بنابراین زمان کل برای الگوریتم *Prim* بـرابـر $O(VlgV+ElgV)=O(ElgV)$ است کـه بـه‌طور مجانبی با پیاده سازی الگوریتم *Kruskal* یکسان می‌باشد.

اما زمان مجانبی اجرای الگوریتم *Prim* می‌تواند با استفاده از *heap* فیبوناچی بهبود یابد. فصل ۲۰ نشان می‌دهد که اگر $|V|$ عـنصر در داخـل *heap* فـیبوناچی سـازماندهی شـوند مـی‌توانـیم عـمل *EXTRACT-MIN* را در زمان سرشکن شده، $O(lgV)$ و عمل *DECREASE-KEY* (پیاده سازی خط ۱۱) را در زمان سرشکن شده $O(1)$ اجرا کنیم. بنابراین اگر *heap* فیبوناچی برای پیاده سازی صف اولویت مینیمم Q استفاده شود، زمان اجرای الگوریتم *Prim* به $O(E+VlgV)$ بهبود می‌یابد.

تمرین‌ها

۱ - ۲۳.۲ الگوریتم *Kruskal* می‌تواند درخت‌های پوشای مینیمم مختلفی برای یک گراف G برگرداند، بسته به آن که وقتی یال‌ها به ترتیب مرتب هستند چطور گره‌ها شکسته می‌شوند. نشان دهید برای هر درخت پوشای مینیمم T از G یک راه برای مرتب کردن یال‌های G در الگوریتم *Kruskal* وجود دارد به طوریکه الگوریتم T را بر می‌گرداند.

۲ - ۲۳.۲ فرض کنید که گراف $G = (V,E)$ به صورت ماتریس همجواری نمایش داده می‌شود. پیاده سازی ساده‌ای از الگوریتم *Prim* ارائه دهید که در زمان $O(V^2)$ اجرامی گردد.

۳ - ۲۳.۲ آیا پیاده سازی *heap* فیبوناچی الگوریتم *Prim* به‌طور مجانبی سریعتر از پـیاده سـازی *heap* دودویی برای گراف پراکنده (خلوت)[1] $G = (V,E)$، که در آن $|E| = \Theta(V)$، می‌باشد؟ در مورد گراف فشرده که $|E| = \Theta(V^2)$، چطور؟ $|E|$ و $|V|$ باید چه رابطه‌ای با هم داشته باشند تا پیاده سازی *heap* فیبوناچی به‌طور مجانبی سریعتر از پیاده سازی *heap* دودویی باشد؟

۴ - ۲۳.۲ فرض کنید که وزن همه یال‌ها در گراف اعدادی صحیح در بازه 2 تا $|V|$ باشند. سرعت اجرای الگوریتم *Kruskal* چقدر است؟ اگر برای ثابت W وزن‌های یال‌ها اعدادی صحیح در بازه 1 تا W باشند چطور؟

۵ - ۲۳.۲ فرض کنید که وزن همه یال‌ها در گراف اعدادی صحیح در بازه 1 تا $|V|$ باشند، سرعت اجرای الگوریتم *Prim* چقدر است؟ اگر برای ثابت W وزن‌های یال‌ها اعدادی صحیح دربـازه 1 تـا W باشند چطور؟

---

1. *sparse*

۲۳.۲-۶ فرض کنید که وزن‌های یال‌ها در گراف به‌طور یکنواخت روی بازه نیمه باز $[0,1)$ توزیع شوند. کدام یک از الگوریتم‌های *Krushal* یا *Prim* می‌تواند سریعتر اجرا شود؟

۲۳.۲-۷ فرض کنید گراف G دارای درخت پوشای مینیممی است که از قبل محاسبه شده است. اگر یک رأس جدید و یال‌های متلاقی به G اضافه شوند درخت پوشای مینیمم با چه سرعتی می‌تواند بروز رسانی شود؟

۲۳.۲-۸ پرفسور *Toole* یک الگوریتم تقسیم و حل جدید برای محاسبه درخت پوشای مینیمم طرح کرده است که به صورت زیر است. برای گراف داده شده $G = (V,E)$، مجموعه V از رئوس به دو مجموعه V_1 و V_2 افراز می‌شود بطوریکه $|V_1|$ و $|V_2|$ به اندازهٔ حداکثر یک واحد اختلاف دارند. فرض کنید E_1 مجموعه یال‌هایی که تنها متلاقی با رئوس V_1 و E_2 مجموعه یال‌هایی که تنها متلاقی با رئوس V_2 هستند باشد. به‌طور بازگشتی مسئله درخت پوشای مینیمم را بر روی هر دو زیر گراف $G_1=(V_1,E_1)$ و $G_2=(V_2,E_2)$ حل کنید. در نهایت یال با وزن مینیمم در E که از بریدگی (V_1,V_2) عبور می‌کند را انتخاب کنید. و از این یال برای یکی کردن نتیجه دو درخت پوشای مینیمم به شکل یک درخت پوشا استفاده کنید. ثابت کنید که الگوریتم به درستی درخت پوشای مینیمم G را محاسبه می‌کند یا یک مثال برای شکست الگوریتم ارئه دهید.

مسائل

۲۳ - ۱ دومین بهترین درخت پوشای مینیمم[1]

فرض کنید $G = (V,E)$ یک گراف همبند بدون جهت با تابع وزن $w:E \to R$ باشد و فرض کنید که $|E| \geq |V|$ و همه وزن یال‌ها متمایز هستند. دومین بهترین درخت پوشا به شکل زیر تعریف می‌شود. فرض کنید $\mathcal{T}$ مجموعه‌ای از همه درخت‌های پوشای G و T' درخت پوشای مینیمم G باشد آن گاه *دومین بهترین درخت پوشای مینیمم*، درخت پوشای T است بطوریکه

$$w(T) = \min_{T'' \in \mathcal{T} - \{T'\}} \{w(T'')\}.$$

a. نشان دهید که درخت پوشای مینیمم منحصر به فرد است اما دومین بهترین درخت پوشای نیاز نیست منحصر به فرد باشد.

b. فرض کنید T درخت پوشای مینیمم G باشد اثبات کنید که یال‌های $(u,v) \in T$ و $(x,y) \notin T$ وجود دارند بطوریکه $T - \{(u,v)\} \cup \{(x,y)\}$ دومین بهترین درخت پوشای مینیمم G است.

c. فرض کنید T درخت پوشای G، و برای هر دو رأس $u,v \in V$ فرض کنید $max[u,v]$ یالی با بیشترین

---

1. *second-best-minimum spanning tree*

وزن بر روی مسیر منحصر به فرد u و v در T باشد. یک الگوریتم با مرتبه زمانی $O(V^2)$ تعریف کنید که برای درخت پوشای T داده شده، برای همه رئوس $u,v \in V$ $max[u,v]$ را محاسبه کند.

d. یک الگوریتم مؤثر برای محاسبه دومین بهترین درخت پوشای مینیمم از گراف G، ارائه دهید.

۲ - ۲۳ درخت پوشای مینیمم در گراف پراکنده[1]

برای هر گراف همبند بسیار پراکنده $G = (V,E)$، می‌توانیم زمان اجرای $O(E+VlgV)$ الگوریتم Prim با heapهای فیبوناچی را به وسیله «پیش پردازش»[2] G به منظور کاهش تعداد رئوس قبل از اجرای الگوریتم، بیشتر بهبود ببخشیم. به خصوص برای هر رأس u یال (u,v) با وزن مینیمم متلاقی بر روی u را انتخاب می‌کنیم. و (u,v) را در درخت پوشای مینیمم در حال ساخت قرار می‌دهیم. آن گاه همهٔ یال‌های انتخاب شده را منقبض می‌کنیم، به جای اینکه آنها را یک یال در یک زمان منقبض کنیم، ابتدا مجموعه رئوسی که در همان رأس جدید واحد سازی می‌شوند را مشخص می‌کنیم. آن گاه گرافی را که از منقبض کردن این یال‌ها یک یال در یک زمان حاصل می‌شود ایجاد می‌کنیم، اما این کار را با «تغییر نام»[3] یال‌ها با توجه به مجموعه‌هایی که انتهای یال‌ها در آنها واقع شده، انجام می‌دهیم. ممکن است چندین یال از گراف اولیه مانند هم تغییر نام یابند در چنین حالتی فقط یک یال نتیجه می‌شود و وزنش مینیمم وزن‌های گراف اولیه متناظر است.

در ابتدا درخت پوشای مینیمم T در حال ساخت را تهی قرار می‌دهیم و برای هر یال $(u,v) \in E$، قرار می‌دهیم $orig\ [u,v] = (u,v)$ و $c[u,v] = w(u,v)$ از خاصیت orig برای اشاره به یالی از گراف اولیه که مرتبط با یالی در گراف منقبض شده است استفاده می‌کنیم.

خاصیت c وزن یک یال را نگه می‌دارد، و همان‌طور که یال‌ها منقبض می‌شوند، c با توجه به طرح بالا برای انتخاب وزن‌های یال‌ها، بروز رسانی می‌شود. روال MST-REDUCE ورودی‌های G، $orig$، c و T را دریافت و گراف منقبض شده G' و خاصیت‌های بروز رسانی شده $orig'$ و c' برای گراف G' را بر می‌گرداند. روال همچنین یال‌های G را در داخل درخت پوشای مینیمم T قرار می‌دهد.

```
MST-REDUCE(G, orig, c, T)
1  for each v ∈ V[G]
2      do mark[v] ← FALSE
3         MAKE-SET(v)
4  for each u ∈ V[G]
5      do if mark[u] = FALSE
6            then choose v ∈ Adj[u] such that c[u, v] is minimized
7                 UNION(u, v)
8                 T ← T ∪ {orig[u, v]}
```

---

1. minimum spaning tree in sparse greph
2. preprocessing
3. renaming

```
9               mark[u] ← mark[v] ← TRUE
10  V[G'] ← {FIND-SET(v) : v ∈ V[G]}
11  E[G'] ← Ø
12  for each (x, y) ∈ E[G]
13       do u ← FIND-SET(x)
14          v ← FIND-SET(y)
15          if (u, v) ∉ E[G']
16             then E[G'] ← E[G'] ∪ {(u, v)}
17                  orig'[u, v] ← orig[x, y]
18                  c'[u, v] ← c[x, y]
19             else if c[x, y] < c'[u, v]
20                  then orig'[u, v] ← orig[x, y]
21                       c'[u, v] ← c[x, y]
22  construct adjacency lists Adj for G'
23  return G', orig', c', and T
```

a. فرض کنید T مجموعه‌ای از یال‌های برگردانده شده توسط *MST-REDUCE* باشد، فرض کنید A درخت پوشای مینیمم گراف G' که به وسیله فرخوانی *MST-PRIM(G',c',r)* ایجاد شده است باشد، که در آن r رأسی در $V[G']$ است. اثبات کنید که

$T \cup \{orig[x,y] : (x,y) \in A\}$ درخت پوشای مینیمم G می‌باشد.

b. اثبات کنید که $|V[G']| \leq |V|/2$.

c. نشان دهید چگونه روال *MST-REDUCE* را پیاده سازی کنیم تا در زمان $O(E)$ اجرا شود (راهنمایی: از ساختمان داده‌های ساده استفاده کنید)

d. فرض کنید که k مرحله، *MST-REDUCE* را با استفاده از خروجی‌های G'، $orig'$ و c' تولید شده توسط یک مرحله به عنوان ورودی‌های G، $orig$ و c مرحله بعد و قرار دادن یال‌ها در T اجرا کنیم. ثابت کنید که زمان اجرای کل k مرحله برابر $O(kE)$ می‌باشد.

e. فرض کنید بعد از اجرای k مرحله از *MST-REDUCE*، به شکل قسمت d الگوریتم *Prim* را به وسیله فراخوانی *MST-PRIM (G',c',r)* اجرا کنیم که در آن G' و c' به وسیله آخرین مرحله برگردانده می‌شوند و r یک رأس در $V[G']$ است. نشان دهید چگونه k را انتخاب کنیم تا زمان اجرای کل برابر $O(E\lg\lg V)$ گردد. اثبات کنید که انتخاب k شما، زمان اجرای مجانبی را حداقل می‌کند.

f. برای چه مقادیری از $|E|$ (بر حسب $|V|$) الگوریتم *Prim* با پیش‌پردازی از الگوریتم *Prim* بدون پیش‌پردازی از لحاظ مجانبی بهتر است.

۳-۲۳ درخت پوشای گلوگاه[1] (تنگنا)

یک درخت *پوشای گلوگاه* T گراف بدون جهت G، درخت پوشایی از G است که بزرگترین وزن یال آن در بین درخت‌های پوشای G،مینیمم باشد. می‌گوییم مقدار درخت پوشای گلوگاه برابر با بیشترین وزن یال در T می‌باشد.

a.اثبات کنید که درخت پوشای مینیمم، درخت پوشای گلوگاه است.

بخش a نشان می‌دهد که یافتن درخت پوشای گلوگاه مشکل‌تر از پیدا کردن درخت پوشای مینیمم نیست در قسمت‌های باقی مانده نشان خواهیم داد که درخت پوشای گلوگاه می‌تواند در زمان خطی پیدا شود.

b.الگوریتم با زمان خطی ارائه دهید که با دریافت گراف G و یک عدد صحیح b تعیین کند آیا مقدار درخت پوشای گلوگاه حداکثر برابر bاست یا خیر؟

c.از الگوریتمتان برای قسمت b به عنوان زیر روال در الگوریتم با زمان خطی برای مسئله درخت پوشای گلوگاه استفاده کنید. (راهنمایی: همان‌طور که در روال *MST-REDUCE* در مسئله ۲۳.۲ توضیح داده شده است شما ممکن است بخواهید از زیر روالی که مجموعه‌های یال‌ها را منقبض می‌کند استفاده کنید)

۴ - ۲۳ الگوریتم‌های درخت پوشای مینیمم جایگزین[2]

در این مسئله سه شبه کد برای سه الگوریتم مختلف ارائه می‌دهیم. هر کدام یک گراف را به عنوان ورودی دریافت و مجموعهٔ T از یال‌ها را بر می‌گرداند. برای هر الگوریتم باید ثابت کنید که T درخت پوشای مینیمم است یا ثابت کنید که T درخت پوشای مینیمم نیست. همچنین مؤثرترین پیاده سازی هر الگوریتم‌چه درخت پوشای مینیمم را محاسبه کند و چه محاسبه نکند، را توضیح دهید.

a. MAYBE-MST-A(G, w)

```
1  sort the edges into nonincreasing order of edge weights w
2  T ← E
3  for each edge e, taken in nonincreasing order by weight
4       do if T − {e} is a connected graph
5             then T ← T − e
6  return T
```

---

1. *Bottleneck spanning tree*

2. *Alternative minimum - spanning - tree algorithm*

b. MAYBE-MST-B(G, w)

```
1  T ← Ø
2  for each edge e, taken in arbitrary order
3       do if T ∪ {e} has no cycles
4             then T ← T ∪ e
5  return T
```

c. MAYBE-MST-C(G, w)

```
1  T ← Ø
2  for each edge e, taken in arbitrary order
3       do T ← T ∪ {e}
4            if T has a cycle c
5              then let e′ be the maximum-weight edge on c
6                     T ← T − {e′}
7  return T
```

۲۴ کوتاهترین مسیرها از مبدأ واحد[1]

یک موتور سوار می‌خواهد کوتاهترین مسیر ممکن بین شیکاگو و بوستون را پیدا کند. در یک نقشه از جاده‌های ایالات متحده که روی آن مسیر بین هر دو تقاطع مجاور مشخص شده است، چگونه می‌توانیم کوتاهترین مسیر را تعیین کنیم؟

یک راه ممکن این است که همه مسیرها از شیکاگو به بوستون را محاسبه کرده، روی هر مسیر فاصله‌ها را جمع کرده و کوتاهترین مسیر را انتخاب کنیم. واضح است که حتّی اگر مسیرهایی که دارای دور هستند را هم در نظر نگیریم باز هم میلیونها مسیر ممکن وجود دارد که اکثر آنها دارای ارزش قابل ملاحظه‌ای نیستند. برای مثال، مسیر شیکاگو – هوستون – بوستون بطور واضح یک انتخاب بی‌ارزش است، چون هوستون تقریباً هزار مایل خارج از مسیر است.

در این فصل و فصل ۲۵ نشان می‌دهیم که چگونه چنین مسایلی را به شکل کارآمد حل کنیم. در *مسئله کوتاهترین مسیر*، یک گراف وزن دار و جهت دار $G = (V, E)$ با تابع وزن $w : E \to R$ که یالها را به وزنهایی با مقادیر حقیقی نگاشت می‌دهد به ما داده می‌شود. *وزن* مسیر $p = <v_1, v_2, \ldots, v_k>$ مجموع وزن‌های یالهای تشکیل دهنده آن است:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

وزن کوتاهترین مسیر[2] از u به v را چنین تعریف می‌کنیم:

اگر مسیری از u به v وجود داشته باشد

در غیر اینصورت

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{اگر مسیری از } u \text{ به } v \text{ وجود داشته باشد} \\ \infty & \text{در غیر اینصورت} \end{cases}$$

بنابراین مسیر p با وزن $w(p) = \delta(u,v)$ بعنوان *کوتاهترین مسیر* از رأس u به رأس v تعریف

1. single-source shortest paths

2. shortest-path weight

می‌شود.

در مثال مسیر شیکاگو به بوستون، می‌توانیم نقشه جاده‌ها را با یک گراف مدل کنیم: رأسها محلهای تقاطع، یالها جاده‌های بین تقاطعها و وزن یالها فاصله‌ها را نشان می‌دهند. هدف ما این است که کوتاهترین مسیر از یک تقاطع داده شده در شیکاگو (خیابان *Clark*، و خیابان *Addison*) به تقاطع مورد نظر در بوستون (خیابان *Brooklin* و جاده *Yawkey*) را بیابیم.

وزنهای یالها می‌توانند غیر از فاصله‌ها به عنوان متریک‌ها تفسیر شوند. وزنها اغلب برای نشان دادن زمان، هزینه، جریمه، کمبودها و هر کمیّت دیگر که در طول یک مسیر پیش می‌آید و می‌خواهیم آن را مینیم کنیم استفاده می‌شوند.

الگوریتم جستجوی اوّل سطح در بخش ۲۲.۲، یک الگوریتم کوتاهترین مسیرها است و روی گرافهای بدون وزن عمل می‌کند – بعبارت دیگر گرافهایی که همه یالهای آن دارای وزن واحدی در نظر گرفته شده‌اند –. چون خیلی از ایده‌های جستجو اوّل سطح در مطالعه گرافهای وزن دار استفاده می‌شوند خواننده بهتر است قبل از ادامه دادن این مبحث، بخش ۲۲.۲ را مرور کند.

انواع دیگر

در این فصل روی مسأله کوتاهترین مسیرها از یک مبدأ واحد متمرکز می‌شویم: در گراف $G = (V, E)$ می‌خواهیم کوتاهترین مسیر از رأس مبدأ $s \in V$ به هر رأس $v \in V$ را بیابیم. خیلی از مسایل دیگر شامل موارد زیر، می‌توانند با الگوریتم مسئله مبدأ واحد حل شوند.

مسئله کوتاهترین مسیرها از مقصد واحد[1]: یافتن کوتاهترین مسیر به یک رأس مقصد مانند t از هر رأس دیگر مانند v. با برعکس کردن جهت یالها در گراف می‌توانیم این مسئله را به مسئله یک مبدأ واحد تبدیل کنیم.

مسئله کوتاهترین مسیر بین یک جفت رأس[2]: یافتن کوتاهترین مسیر از u به v برای دو رأس u و v داده شده. اگر مسئله مبدأ واحد را با مبدأ u حل کنیم این مسئله نیز حل شده است. به علاوه، هیچ الگوریتمی برای این مسئله شناخته نشده که بطور مجانبی سریعتر از بهترین الگوریتمهای یک مبدأ واحد در بدترین حالت، اجرا گردد.

مسئله‌کوتاهترین مسیرها بین همه جفتها[3]: یافتن کوتاهترین مسیر از u به v برای هر جفت u و v. اگر چه این مسئله‌می‌تواند با اجرای یک بار الگوریتم مبدأ واحد برای هر رأس حل شود، امّا می‌تواند سریعتر نیز به جواب برسد. به علاوه، ساختار آن در نوع خود جالب است. فصل ۲۵ مسئله‌همه جفتها را همراه با جزئیات بررسی می‌کند.

---

1. *single-destination shortest-paths*
2. *single-pair shortest-path*
3. *all-pairs shortest-paths*

زیر ساختار بهینه یک کوتاهترین مسیر

الگوریتمهای کوتاهترین مسیرها نوعاً اتکا به این خاصیّت می‌کنند که هر کوتاهترین مسیر بین دو رأس، شامل کوتاهترین مسیرهای دیگر در داخلش است. (الگوریتم ماکزیمم جریان *Edmonds-Karp* در فصل ۲۶ نیز مبتنی بر این خاصیّت است.) زیر ساختار بهینه، نشانه‌ای از قابلیت اعمال هر دو روش برنامه‌سازی پویا (فصل ۱۵) و حریصانه (فصل ۱۶) است. الگوریتم *Dijkstra* که در بخش ۲۴.۳ خواهیم دید، یک الگوریتم حریصانه و الگوریتم *Floyd-Warshall* که کوتاهترین مسیرهای بین هر جفت رأس را می‌یابد (فصل ۲۵) یک الگوریتم برنامه سازی پویا است. لم بعد، ویژگی زیرساختار بهینه کوتاهترین مسیرها را دقیقتر بیان می‌کند.

لم ۲۴.۱ (زیر مسیرهای کوتاهترین مسیرها خود، کوتاهترین مسیرها هستند)

در گراف وزن‌دار و جهت‌دار $G = (V, E)$ با تابع وزن $w:E \to R$ فرض کنید $p = <v_1, v_2, \ldots, v_k>$ کوتاهترین مسیر از رأس v_1 به رأس v_k باشد و برای هر i و j که $1 \le i \le j \le k$، $p_{ij} = <v_i, v_{i+1}, \ldots, v_j>$ زیرمسیری از رأس v_i به v_j باشد. آنگاه p_{ij} کوتاهترین مسیر از رأس v_i به رأس v_j است.

اثبات اگر مسیر p را به $v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$ تجزیه کنیم خواهیم داشت:
$w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$

حال فرض کنید مسیر p'_{ij} از v_i به v_j وجود دارد بطوریکه $w(p'_{ij}) < w(p_{ij})$. بنابراین مسیر $v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p'_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$ وجود دارد که وزن آن یعنی $w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ کمتر از $w(p)$ است که این موضوع با این فرض که p کوتاهترین مسیر از v_1 به v_k است در تناقض قرار دارد. ■

یالها با وزن منفی

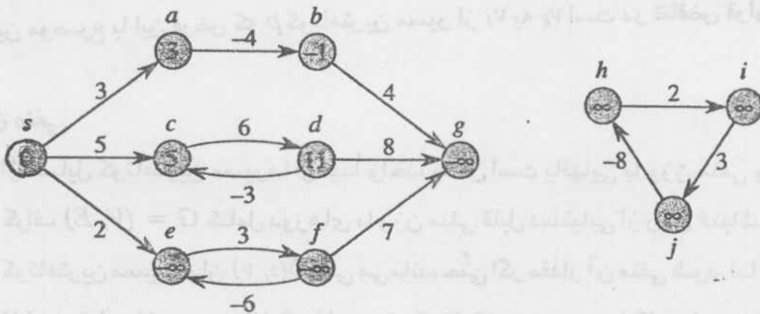
در بعضی از مسایل کوتاهترین مسیرها از مبدأ واحد، ممکن است یالهایی با وزن منفی وجود داشته باشند. اگر گراف $G = (V, E)$ شامل دورهای با وزن منفی قابل دستیابی از رأس s نباشد، برای هر $v \in V$ وزن کوتاهترین مسیر همان $\delta(s, v)$ باقی می‌ماند، حتّی اگر مقدار آن منفی شود. اما اگر دوری با وزن منفی قابل دستیابی از s وجود داشته باشد، وزن کوتاهترین مسیر به شکل مناسبی تعریف نشده است. هیچ مسیری از s به یک رأس موجود در دور، نمی‌تواند به عنوان کوتاهترین مسیر انتخاب شود زیرا همیشه یک مسیر با وزن کمتر یافت می‌شود که از دور با وزن منفی عبور می‌کند. اگر دوری با وزن منفی در مسیر از s به v وجود داشته باشد تعریف می‌کنیم $\delta(s, v) = -\infty$.

شکل ۲۴.۱ تأثیر وزنهای منفی و دورهای با وزن منفی روی وزن کوتاهترین مسیر را شرح می‌دهد. چون فقط یک مسیر از s به a ($<s, a>$) وجود دارد $\delta(s, a) = w(s, a) = 3$ بطور مشابه، تنها یک مسیر از s به b وجود دارد و لذا $\delta(s, b) = w(a, b) = 3+(-4) = -1$. مسیرهای نامحدود زیادی از

s بـــه c وجـــود دارد مـــانند $<s, c>$، $<s, c, d, c>$ و $<s, c, d, c, d, c>$ و... چـون دور $<c, d, c>$ دارای وزن $6+(-3) = 3 > 0$ است کـوتاهترین مسـیر از s به c، $<s, c>$ بـا وزن $\delta = (s, c) = 5$ مـی‌باشد. بـطور مشـابه کـوتاهترین مسـیر از s بـه d، $<s, c, d>$ بـا وزن $\delta(s, d) = w(s, c) = 11$ است. به طور مشابه مسیرهای بسیار زیادی از s به e وجود دارد مانند $<s, e>$، $<s, e, f, e>$، $<s, e, f, e, f, e>$ و... چون دور $<e, f, e>$ دارای وزن $3+(-6) = -3 < 0$ است، کوتاهترین مسیر بین s و e وجود ندارد. با پیمودن دور منفی $<e, f, e>$ به تعداد دفعات زیاد دلخواه، می‌توانیم مسیرهایی از s به e با وزنهای منفی بزرگ دلخواه بیابیم و بنابراین $\delta(s, e) = -\infty$. به طور مشابه $\delta(s, f) = -\infty$ چون g از f قابل دستیابی است می‌توانیم مسیرهایی با وزنهای منفی بزرگ دلخواه از s به g نیز بیابیم، پس $\delta(s, g) = -\infty$. رأسهای h، i و j نیز یک دور با وزن منفی را تشکیل می‌دهند. این رأسها از s قابل دستیابی نیستند و بنابراین $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

برخی از الگوریتمهای کوتاهترین مسیرها مانند الگوریتم *Dijkstra* فرض می‌کنند که همه وزنهای یالها در گراف ورودی غیر منفی هستند، مانند مثال نقشه راه.

دیگر الگوریتمها مانند الگوریتم *Bellman-Ford* یالهای با وزن منفی را در گراف ورودی قبول می‌کنند و جواب درستی را هم تولید می‌کنند، ولی به شرط آنکه هیچ دوری با وزن منفی از مبدأ قابل دستیابی نباشد. نوعاً اگر چنین دوری وجود داشته باشد، این الگوریتمها می‌توانند آن را کشف و وجود آن را گزارش کنند.



شکل ۲۴.۱ وزنهای منفی در گرافهای جهتدار. داخل هر رأس، کوتاهترین مسیر از مبدأ s نشان داده شده است. چون رأسهای e، f یک دور منفی قابل دستیابی از s را تشکیل داده‌اند دارای کوتاهترین مسیر با وزن $-\infty$ هستند. چون رأس g قابل دستیابی از رأسی است که خود دارای کوتاهترین مسیر با مقدار $-\infty$ است، پس وزن کوتاهترین مسیر این رأس هم $-\infty$ است. رأسهای h، i و j قابل دستیابی از s نیستند و بنابراین وزن کوتاهترین مسیر آنها ∞ است، حتّی اگر آنها داخل یک دور منفی باشند.

دورها[1]

آیا کوتاهترین مسیر می‌تواند شامل یک دور باشد؟ همان طور که دیدیم کوتاهترین مسیر نمی‌تواند در داخل خود دور منفی داشته باشد. همچنین نمی‌تواند شامل دور با وزن مثبت هم باشد چون برداشتن این دور از مسیر، مسیری با همان مبدأ و مقصد و با وزن کمتر تولید می‌کند. به عبارت دیگر اگر $p = <v_0, v_1, \ldots, v_k>$ یک مسیر و $c = <v_i, v_{i+1}, \ldots, v_j>$ یک دور مثبت روی این مسیر باشد (که در آن $v_i = v_j$ و $w(c) > 0$)، مسیر $p' = <v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k>$ دارای وزن $w(p') = w(p) - w(c) < w(p)$ است و بنابراین p نمی‌تواند مسیر از v_0 به v_k باشد.

داشتن دورهای با وزن صفر اشکالی ندارد. می‌توانیم دور با وزن صفر را از یک مسیر حذف کنیم تا مسیر دیگری با همان وزن به وجود آید. بنابراین اگر یک کوتاهترین مسیر از رأس مبدأ s به رأس مقصد v شامل دور صفر وجود داشته باشد، یک کوتاهترین مسیر از رأس s به رأس v بدون این دور نیز وجود دارد. تا زمانی که کوتاهترین مسیر شامل دورهای با وزن منفی باشد می‌توانیم مکرراً این دورها را حذف کنیم تا به یک کوتاهترین مسیر بدون دور برسیم. بنابراین بدون از دست دادن کلیّت می‌توانیم فرض کنیم کوتاهترین مسیرهایی که می‌یابیم دور ندارند. چون هر مسیر بدون دور در گراف $G = (V, E)$ شامل حداکثر $|V|$ رأس مجزا است، بنابراین دارای حداکثر $|V|$-1 یال نیز می‌باشد. پس می‌توانیم فقط به مسیرهای با حداکثر $|V|$-1 یال بپردازیم.

نمایش کوتاهترین مسیرها

اغلب می‌خواهیم علاوه بر وزن کوتاهترین مسیر، رأسهای روی این مسیر را نیز محاسبه کنیم. شکلی که برای نشان دادن کوتاهترین مسیرها استفاده می‌کنیم شبیه شکلی است که برای درختهای اول سطح در بخش ۲۲.۲ استفاده کردیم. در گراف $G = (V, E)$ برای هر رأس $v \in V$ یک ماقبل[2] به عنوان $\pi[v]$ نگه می‌داریم که می‌تواند یک رأس دیگر یا NIL باشد. در این فصل الگوریتمهای کوتاهترین مسیرها، خاصیّت π را چنان مقدار دهی می‌کنند که زنجیره ماقبلها که از v سرچشمه می‌گیرند، در کوتاهترین مسیر s به v قبل از رأس v قرار بگیرند. بنابراین برای رأس v با شرط $\pi[v] \neq NIL$، روال $PRINT\text{-}PATH(G, s, v)$ دربخش ۲۲.۲ می‌تواند برای چاپ کوتاهترین مسیر از s به v استفاده شود.

در طول اجرای الگوریتمهای کوتاهترین مسیرها مقادیر π لازم نیست به کوتاهترین مسیرها دلالت کنند. مانند جستجوی اوّل سطح، زیرگراف ماقبل[3] $G_\pi = (V_\pi, E_\pi)$ که از مقادیر π ایجاد شده است مورد نظر ما می‌باشد. اینک دوباره V_π را به عنوان مجموعه‌ای از رأسهای G با ماقبلهای غیر NIL بعلاوه مبدأ s تعریف می‌کنیم:

---

1. cycles
2. predecessor
3. predecessor subgraph

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\} .$$

مجموعه E_π مجموعه‌ای از یالهای جهت دار است که از مقادیر π برای رأسهای مجموعه V_π به دست آمده است:

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\} .$$

ثابت خواهیم کرد که مقادیر π که توسط الگوریتمهای این فصل تولید می‌شوند دارای این ویژگی هستند که در انتها G_π یک درخت کوتاهترین مسیر است - درختی ریشه دار شامل کوتاهترین مسیر، از رأس مبدأ s به هر رأس قابل دستیابی از آن. یک درخت کوتاهترین مسیر مانند درخت اوّل سطح در بخش ۲۲.۲ است امّا شامل کوتاهترین مسیرها از رأس مبدأ از لحاظ وزن یالها می‌باشد نه تعداد یالها. اجازه دهید دقیقتر شویم. گراف $G = (V, E)$ یک گراف جهت دار وزن‌دار با تابع وزن $w : E \rightarrow R$ است و فرض کنید G شامل دور با وزن منفی قابل دستیابی از رأس مبدأ $s \in V$ نیست، همان طور که کوتاهترین مسیرها تعریف شده‌اند. درخت *کوتاهترین مسیرها*[1] با ریشه s، یک زیرگراف جهت دار $G' = (V', E')$ است که $V' \subseteq V$ و $E' \subseteq E$ بطوری که

۱. V' یک مجموعه از رأس‌های قابل دستیابی از s در گراف G است.

۲. G' یک درخت مشتق شده با ریشه s را تشکیل می‌دهد و

۳. برای همه $v \in V'$، تنها مسیر موجود از s به v در G'، کوتاهترین مسیر از s به v در G است.

کوتاهترین مسیرها و درخت‌های کوتاهترین مسیرها لزوماً منحصر به فرد نیستند. برای مثال، شکل ۲۴-۲ یک گراف جهتدار وزن‌دار و دو درخت کوتاهترین مسیرها با ریشه یکسان را نشان می‌دهد.

تکنیک آرام‌سازی[2]

الگوریتمهای این فصل از تکنیک *آرام‌سازی*[3] استفاده می‌کنند. برای هر رأس $v \in V$، خصوصیت $d[v]$ که حد بالای وزن کوتاهترین مسیر از رأس مبدأ s به v است را نگه می‌داریم. $d[v]$ را برآورد کوتاهترین مسیر[4] می‌نامیم. به وسیلهٔ زیرروال زیر که در زمان $\Theta(V)$ اجرا می‌شود مقادیر برآوردهای کوتاهترین مسیر و ماقبلها را مقداردهی اولیه می‌کنیم:

---

1. shortest-paths tree

۲- ممکن است عجیب به نظر برسد که چرا از اصطلاح "آرام سازی" برای عملی که حد بالا را تثبیت می‌کند استفاده شده است. استفاده از این اصطلاح، تاریخی است. نتیجه گام آرام سازی را می‌توان چنین در نظر گرفت: آرام کردن محدودیت $d[v] \leq d[u]+w(u, v)$ که با توجّه به نامساوی مثلث زمانی برقرار است که $d[u] = \delta(s, u)$ و $d[v] = \delta(s, v)$ یعنی اگر $d[v] \leq d[u]+w(u, v)$ هیچ گونه "فشاری" برای ایجاد این محدودیت وجود ندارد و لذا این محدودیت آرام می‌شود.

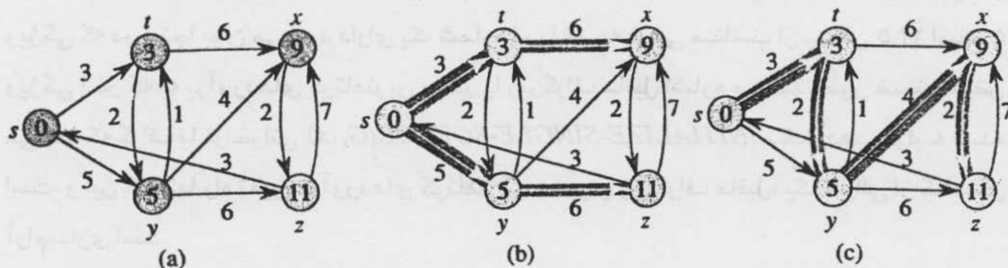
3. relaxation

4. shortest-path estimate

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ V[G]
2       do d[v] ← ∞
3          π[v] ← NIL
4  d[s] ← 0
```

پس از مقداردهی اولیه برای همه $v \in V$ داریم $\pi[v] = NIL$ و برای همه $v \in V-\{s\}$ داریم $d[s] = 0$ و $d[v]=\infty$. فرآیند آرام سازی یال (u, v) تشکیل شده است از تست این که آیا می‌توانیم کوتاهتریم مسیر به v را که تا اینجا پیدا شده، با عبور از u ادامه دهیم و اگر چنین است مقادیر $d[v]$ و $\pi[v]$ را تغییر دهیم. یک گام آرام‌سازی ممکن است مقدار $d[v]$ را کاهش دهد و $\pi[v]$ را تغییر دهد. کد زیر یک گام آرام‌سازی را روی یال (u,v) اجرا می‌کند.

```
RELAX(u, v, w)
1  if d[v] > d[u] + w(u, v)
2     then d[v] ← d[u] + w(u, v)
3          π[v] ← u
```

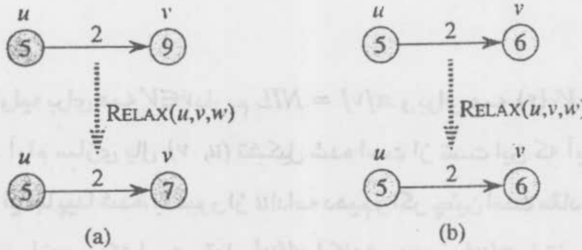


شکل ۲۴.۲ *(a)* یک گراف وزن‌دار جهتدار با وزنهای کوتاهترین مسیر از رأس مبدأ. *(b)* یالهای سایه‌دار، یک درخت کوتاهترین مسیرها با مبدأ s را تشکیل می‌دهند. *(c)* یک درخت کوتاهترین مسیرهای دیگر با همان ریشه.

شکل ۲۴.۳ دو مثال از آرام‌سازی یک یال را نشان می‌دهد. در یکی برآورد کوتاهترین مسیر کاهش می‌یابد و در دیگری تغییر نمی‌کند.

هر الگوریتم در این فصل *INITIALIZE-SINGLE-SOURCE* را فراخوانی می‌کند و مکرراً یالها را آرام‌سازی می‌کند. علاوه بر آن آرام‌سازی تنها وسیله‌ای است که توسط آن، برآورد کوتاهترین

مسیر و ماقبلها تغییر می‌کنند. الگوریتمها در این فصل از نظر تعداد آرام‌سازی هر یال و مرتبه زمانی این عمل با هم متفاوتند. در الگوریتم *Dijkstra* و الگوریتم کوتاهترین مسیرها برای گرافهای جهتدار بدون دور، هر یال دقیقاً یک بار آرام می‌شود. در الگوریتم *Bellman-Ford* هر یال چندین بار آرام می‌شود.



شکل ۲۴.۳ آرام سازی یال (u, v) با وزن $w(u, v) = 2$. برآورد کوتاهترین مسیر هر رأس در داخل آن رأس نشان داده شده است. چون قبل از آرام‌سازی داریم $d[v] > d[u]+w(u, v)$، مقدار $d[v]$ کاهش می‌یابد. *(b)* در اینجا قبل از گام آرام‌سازی داریم $d[v] \leq d[u]+w(u, v)$، بنابراین $d[v]$ بدون تغییر باقی می‌ماند.

ویژگیهای کوتاهترین مسیرها و آرام‌سازی

برای اثبات صحّت الگوریتمهای این فصل، به چند ویژگی کوتاهترین مسیرها و آرام‌سازی متوسل می‌شویم. در اینجا این ویژگیها را بیان می‌کنیم و بخش ۲۴.۵ آنها را ثابت می‌کند. به عنوان مرجع، هر ویژگی که در اینجا بیان می‌شود دارای یک شماره لم یا قضیه فرعی مناسب از بخش ۲۴.۵ است. ۵ ویژگی آخر که به برآوردهای کوتاهترین مسیر یا زیرگراف ماقبل اشاره می‌کنند بطور ضمنی فرض می‌کنند که گراف با فراخوانی *INITIALIZE-SINGLE-SOURCE(G, s)* مقداردهی اولیه شده است، و این که تنها راه تغییر برآوردهای کوتاهترین مسیر و زیرگراف ماقبل، یک توالی از گامهای آرام‌سازی است.

نامساوی مثلث [1] (لم ۲۴.۱۰)

برای هر یال $(u, v) \in E$ داریم $\delta(s, v) \leq \delta(s, u)+w(u, v)$.

ویژگی حدّ بالا [2] (لم ۲۴.۱۱)

برای همه رأسهای $v \in V$ همیشه داریم $d[v] \geq \delta(s, v)$، و زمانی که $d[v]$ برابر $\delta(s,v)$ شد دیگر تغییر

1.Triangle Inequality
2. upper - bound property

نمی‌کند.

ویژگی نبود مسیر[1] (قضیه فرعی ۲۴.۱۲)

اگر مسیری از s به v وجود نداشته باشد همیشه داریم $d[v] = \delta(s, v) = \infty$.

ویژگی همگرایی[2] (لم ۲۴.۱۴)

اگر برای $u, v \in V$ $s \leadsto u \to v$ کوتاهترین مسیر در G باشد و قبل از آرام سازی یال (u,v) داشته باشیم $d[u] = \delta(s, u)$، آنگاه بعد از آرام سازی خواهیم داشت $d[v] = \delta(s, v)$.

ویژگی آرام سازی مسیر(لم ۲۴.۱۵)

اگر $p = <v_0, v_1, \ldots, v_k>$ یک کوتاهترین مسیر از $s = v_0$ به v_k باشد و یالهای p به ترتیب $(v_{k-1}, v_k), \ldots, (v_1, v_2), (v_0, v_1)$ آرام شده باشند آنگاه $d[v_k] = \delta(s, v_k)$. این ویژگی بدون توجه به گامهای آرام سازی دیگر و حتّی آرام سازی یالهای p برقرار است.

ویژگی زیرگراف ماقبل[3] (لم ۲۴.۱۷)

زمانی که برای همه $v \in V$ داشته باشیم $d[v] = \delta(s, v)$ آنگاه زیرگراف ماقبل، درخت کوتاهترین مسیر با ریشه s است.

رئوس مطالب این فصل

بخش ۲۴.۱ الگوریتم *Bellman-Ford* را معرّفی می‌کند که مسئله کوتاهترین مسیر از مبدأ واحد در حالت کلّی که یالها می‌توانند وزن منفی داشته باشند را حل می‌کند. الگوریتم *Bellman-Ford* در عین سادگی، قابل ملاحظه است و علاوه بر آن می‌تواند کشف کند که آیا دوری با وزن منفی از مبدأ قابل دستیابی است. بخش ۲۴.۲ یک الگوریتم با زمان خطی جهت محاسبه کوتاهترین مسیرها از یک مبدأ واحد در یک گراف جهت دار بدون دور را ارائه می‌کند. بخش ۲۴.۳ الگوریتم *Dijkstra* را پوشش می‌دهد که زمان اجرای آن کمتر از *Bellman-Ford* است ولی در آن، وزن یالها باید غیر منفی باشند. بخش ۲۴.۴ نشان می‌دهد چگونه الگوریتم *Bellman-Ford* می‌تواند برای حل یک حالت خاص از "برنامه سازی خطی" استفاده شود. و در نهایت بخش ۲۴.۵ خصوصیّات کوتاهترین مسیرها و آرام سازی که قبلاً ذکر شد را اثبات می‌کند.

به یک سری قرار داد برای انجام اعمال و کار کردن با بی نهایت‌ها احتیاج داریم. فرض خواهیم کرد که برای هر عدد حقیقی $a \neq -\infty$ داریم $a + \infty = \infty + a = \infty$. برای اینکه اثبات‌های ما با وجود دورهای منفی هم درست باشد فرض خواهیم کرد که برای هر عدد حقیقی $a \neq \infty$ داریم $a + (-\infty) = (-\infty) + a = -\infty$.

---

1. no-path property
2. convergence property
3. predecessor-subgraph property

همهٔ الگوریتمهای این فصل فرض می‌کنند که گراف جهت دار G بصورت نمایش لیست مجاورتی نخیره شده و علاوه بر آن با هر یال، وزن آن نیز نخیره شده است که می‌توانیم همراه با پیمایش لیست مجاورتی، وزن را برای هر یال با زمان $O(1)$ تعیین کنیم.

۲۴.۱ الگوریتم *Bellman-Ford*

الگوریتم *Bellman-Ford* مسئله کوتاهترین مسیر از یک مبدأ واحد را در حالت کلّی که وزن یالها ممکن است منفی باشند حل می‌کند. در گراف وزن دار و جهت دار داده شده $G = (V, E)$ با تابع وزن $w : E \rightarrow R$، الگوریتم *Bellman-Ford*، یک مقدار بولی برمی گرداند که نشان می‌دهد آیا دوری با وزن منفی از مبدأ قابل دستیابی است یا خیر. اگر چنین دوری وجود داشته باشد الگوریتم بیان می‌کند که راه حلی برای این مسئله وجود ندارد. اگر چنین دوری وجود نداشته باشد کوتاهترین مسیرها و وزن آنها را تولید می‌کند.

الگوریتم از آرام سازی استفاده می‌کند و به طور پیش رونده برآورد $d[v]$ روی وزن کوتاهترین مسیر از مبدأ را برای هر $v \in V$ کاهش می‌دهد تا وقتی که به مقدار واقعی وزن کوتاهترین مسیر یعنی $\delta(s,v)$ برسد. الگوریتم مقدار *TRUE* را برمی‌گرداند اگر و فقط اگر گراف شامل هیچ دوری با وزن منفی قابل دستیابی از مبدأ نباشد.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to |V[G]| − 1
3      do for each edge (u, v) ∈ E[G]
4             do RELAX(u, v, w)
5  for each edge (u, v) ∈ E[G]
6      do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

شکل ۲۴.۴ اجرای الگوریتم *Bellman-Ford* روی یک گراف ۵ رأسی را نشان می‌دهد. پس از مقدار دهی اولیّه d و π برای همه رأسها در خط ۱، الگوریتم $|V|-1$ گذر روی یالهای گراف انجام می‌دهد. هر گذر، یک تکرار حلقه *for* خطوط ۴-۲ است و تشکیل شده از یک بار آرام سازی هر یال گراف. قسمتهای (b)-(e) شکل ۲۴.۴، وضعیت الگوریتم پس از هر یک از ۴ گذر روی یالها را نشان می‌دهد. پس از $|V|-1$ گذر، خطوط ۸-۵ وجود دور با وزن منفی را چک می‌کنند و مقدار بولی مناسب را برمی گردانند.(اندکی بعد خواهیم دید که چرا این چک صورت می‌گیرد.)

الگوریتم *Bellman-Ford* در زمان $O(VE)$ اجرا می‌شود چون مقدار دهی اولیه در خط ۱، به اندازه

$\Theta(V)$ و هر یک از $|V|-1$ گذر روی یالها در خطوط ۴-۲، به اندازه $\Theta(E)$ و حلقه *for* خطوط ۷-۵ به اندازه $O(E)$ زمان می‌برند.

برای اثبات صحّت الگوریتم *Bellman-Ford* ابتدا نشان می‌دهیم که اگر هیچ دوری با وزن منفی وجود نداشته باشد، الگوریتم وزنهای کوتاهترین مسیر را برای همه رئوس قابل دستیابی از مبدأ به درستی محاسبه می‌کند.

شکل ۲۴.۴ اجرای الگوریتم *Bellman-Ford* مبدأ رأس sاست. مقادیر d داخل رأسها نمایش داده شده‌اند و یالهای سایه دار، مقادیر ماقبل را نشان می‌دهند: اگر یال (u,v) سایه زده شده باشد آنگاه $\pi[v]=u$است. در این مثال خاص، هر گذر یالها را به ترتیب (t, x) ، (t, y) ، (t, z) ، (x, t) ، (y, x) ، (y, z) ، (z, x) ، (z, s) ، (s, t) ، (s, y) . آرام می‌کند. (a) وضعیّت گراف درست قبل از اوّلین گذر روی یالها (b)-(e) وضعیّت گراف بعد از هر گذر متوالی روی یالها. مقادیر π,d در قسمت (e) مقادیر نهایی هستند. الگوریتم *Bellman-Ford* در این مثال، مقدار *TRUE* را برمی‌گرداند.

لم ۲۴.۲

$G=(V,E)$ یک گراف جهت دار وزن دار با مبدأء s و تابع وزن $w:E\rightarrow R$ است. فرض کنید G دوری با وزن منفی قابل دستیابی از s ندارد. بعد از $|V|-1$ تکرار حلقه *for* در خطوط ۴-۲ الگوریتم، برای همه رأسهای v قابل دستیابی از s داریم: $d[v] = \delta(s,v)$

اثبات این لم را به کمک ویژگی آرام سازی مسیر ثابت می‌کنیم. همه رأسهای v از s قابل دستیابی هستند را در نظر بگیرید و فرض کنید $p=<v_0, v_1,...,v_k>$ که در آن $v_0=s$ و $v_k=v$ است یک کوتاهترین

مسیر بدون دور از s به v باشد. مسیر p حداکثر $|V|-1$ یال دارد بنابراین $k \le |V|-1$. هر کدام از $|V|-1$ تکرار حلقه for در خطوط ۴-۲ همه یالهای E را آرام می‌کند. از جمله یالهای آرام شده در i امین تکرار بـرای $i=1,2,...,k$، یـال (v_{i-1}, v_i) است. بـنابرایـن بـنا بـه ویـژگی آرام سـازی مسـیر داریـم

$$d[v]=d[v_k] = \delta(s,v_k) = \delta(s,v)$$ ■

قضیه فرعی ۲۴.۳

فرض کنید $G=(V,E)$ یک گراف وزن دار جهت دار با مبدأ s و تابع وزن $w:E \rightarrow R$ باشد. برای هر رأس $v \in V$ مسیری از s به v وجود دارد اگر و فقط اگر پس از اجرای الگوریتم *Bellman-Ford*، $d[v] < \infty$ باشد.

اثبات اثبات به عنوان تمرین ۲-۲۴.۱ واگذار شده است. ■

قضیه ۲۴.۴ (صحّت الگوریتم *Bellman-Ford*)

فرض کنید الگوریتم *Bellman-Ford* روی گراف وزن دار و جهت دار $G=(V,E)$ با مبدأ s و تابع وزن $w:E \rightarrow R$ اجرا شود. اگر G هیچ دوری با وزن منفی قابل دستیابی از s نداشته باشد الگوریتم مـقدار *TRUE* را برمی‌گرداند و برای همهٔ رئوس $v \in V$ داریم $d[v] = \delta(s,v)$ و زیرگراف مـاقبل G_π یک درخت کوتاهترین مسیر با ریشه s است. اگر G شامل دوری با وزن منفی قابل دستیابی از s باشد، الگوریتم مقدار *FALSE* را برمی‌گرداند.

اثبات فرض کنید گراف G شامل دوری با وزن منفی و قابل دستیابی از s نباشد. در ابتدا این ادّعا ثابت می‌کنیم که در پایان کار الگوریتم برای تمام رئوس $v \in V$، $d[v] = \delta(s,v)$ اگر رأس v قابل دستیابی از s باشد لم ۲۴.۲ این ادّعا را ثابت می‌کند. اگر v از s قابل دستیابی نباشد ادّعای مورد نظر از ویژگی عدم وجود مسیر دنبال می‌شود. لذا این ادّعا ثابت می‌شود. خاصیّت زیرگراف ماقبل همراه با این ادّعا بیان می‌کند که G_π درخت کوتاهترین مسیرها است. حال از این مطلب استفاده می‌کنیم تا نشان دهیم کـه الگوریتم *Bellman-Ford* مقدار *TRUE* را برمی‌گرداند. در انتهای الگوریتم برای همه یالهای $(u,v) \in E$ داریم:

$$\begin{aligned} d[v] &= \delta(s, v) \\ &\le \delta(s, u) + w(u, v) \quad \text{(بنا به نامساوی مثلث)} \\ &= d[u] + w(u, v)\,, \end{aligned}$$

و بنابراین هیچ یک از تستهای خط ششم باعث نمی‌شود که الگوریتم مقدار *FALSE* را برگرداند و بنابراین مقدار *TRUE* برگردانده می‌شود.

بر عکس فرض کنید گراف G دارای دوری با وزن منفی و قابل دستیابی از مبدأ s باشد. قرار دهید $c=<v_0,v_1,...,v_k>$ که $v_0=v_k$ بنابراین

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 . \quad (۲۴.۱)$$

برای ایجاد تناقض فرض کنید الگوریتم *Bellman-Ford* مقدار *TRUE* را برگرداند. بنابراین برای $i=1,2,...,k$ داریم $d[v_i] \le d[v_{i-1}] + w(v_{i-1},v_i)$ با جمع نامساوی‌ها در دور c داریم:

$$\begin{aligned}\sum_{i=1}^{k} d[v_i] &\le \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) .\end{aligned}$$

چون $v_0=v_k$، هر رأس در دور c دقیقاً یک بار در هر یک از جمعهای $\sum_{i=1}^{k} d[v_i]$ و $\sum_{i=1}^{k} d[v_{i-1}]$ ظاهر می‌شود و بنابراین

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}] .$$

علاوه بر این طبق قضیه فرعی ۲۴.۳ برای $i=1,2,...,k$ ، $d[v_i]$ محدود است. بنابراین

$$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i) ,$$

که با نامساوی ۲۴.۱ در تناقض قرار دارد. نتیجه می‌گیریم که الگوریتم *Bellman-Ford* اگر گراف G دور با وزن منفی قابل دستیابی از مبدأ نداشته باشد، مقدار *TRUE* را باز می‌گرداند و در غیر این صورت مقدار *FALSE* را برمی‌گرداند. ■

تمرین‌ها

۲۴.۱-۱ الگوریتم *Bellman-Ford* را روی گراف جهت دار شکل ۲۴.۴ اجرا کنید و از رأس z به عنوان مبدأ استفاده کنید. در هر گذر، یالها را با همان ترتیب ذکر شده آرام سازی کنید و مقادیر d و π را بعد از هر گذر نمایش دهید. اینک وزن یال (z,x) را به 4 تغییر داده و دوباره الگوریتم را با مبدأ s اجرا کنید.

۲۴.۱-۲ قضیه فرعی ۲۴.۳ را ثابت کنید.

۲۴.۱-۳ در گراف جهت دار و وزن دار $G=(V,E)$ که فاقد دورهای منفی است، m را ماکزیمم بین مینیمم تعداد یالها در کوتاهترین مسیر از u به v به ازاء همه جفت رأسهای $u,v\in V$ قرار دهید. (در این جا کوتاهترین مسیر از نظر وزن است و نه از نظر تعداد یالها) تغییر ساده‌ای روی *Bellman-Ford* پیشنهاد کنید تا این الگوریتم در $m+1$ گذر خاتمه یابد.

۲۴.۱-۴ الگوریتم *Bellman-Ford* را چنان تغییر دهید که در همه رأسهای v که برای آنها یک دور با وزن منفی در مسیری با مبدأ v وجود دارد، $d[v]$ مقدار $-\infty$ را بگیرد.

۲۴.۱-۵* فرض کنید $G=(V,E)$ یک گراف وزن‌دار جهتدار با تابع وزن $w:E\rightarrow R$ باشد. یک الگوریتم با مرتبه زمانی $O(VE)$ ارائه دهید که برای هر $v\in V$ مقدار $\delta^*(v)=min_{u\in V}\{\delta(u, v)\}$ را بیابد.

۲۴.۱-۶ فرض کنید گراف وزن‌دار جهتدار $G=(V,E)$ دارای یک دور با وزن منفی باشد. الگوریتم کارآمدی ارائه دهید که رأسهای یک چنین دوری را لیست کند. ثابت کنید که الگوریتم شما، درست است.

۲۴.۲ کوتاهترین مسیر از مبدأ واحد در گرافهای جهتدار بدون دور

با آرام کردن یالهای یک *dag* (گراف جهتدار بدون دور) وزن‌دار $G=(V,E)$ طبق یک مرتب‌سازی موضعی رأسهای آن، می‌توانیم کوتاهترین مسیرها از یک مبدأ واحد را در زمان $\Theta(V+E)$ محاسبه کنیم. کوتاهترین مسیرها همیشه در یک *dag* خوش تعریف هستند چون حتی اگر یالهایی با وزن منفی هم وجود داشته باشند، دوری با وزن منفی نمی‌تواند وجود داشته باشد.

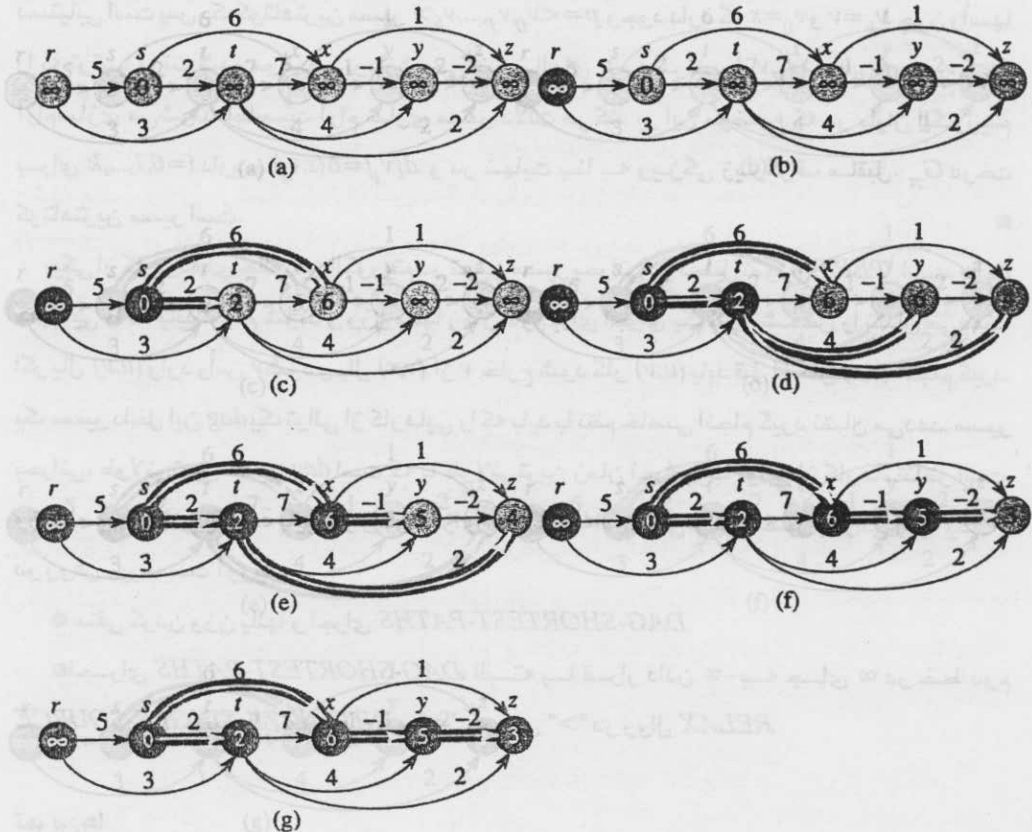
الگوریتم با مرتب سازی موضعی *dag* (بخش ۲۲.۴) آغاز می‌شود تا ترتیب خطی را روی رأسها بوجود آورد. اگر مسیری از u به v وجود دارد در مرتب‌سازی موضعی، u قبل از v قرار می‌گیرد. فقط یک گذر روی رأسهایی که به ترتیب موضعی مرتب شده‌اند انجام می‌دهیم. همانطور که هر رأس را پردازش می‌کنیم، یالی که از آن خارج می‌شود را نیز آرام‌سازی می‌کنیم.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       do for each vertex v ∈ Adj[u]
5              do RELAX(u, v, w)
```

شکل ۲۴.۵ مراحل اجرای این الگوریتم را نشان می‌دهد.

تحلیل زمان اجرای این الگوریتم، ساده است. همان طور که در بخش ۲۲.۴ نشان داده شد مرتب‌سازی موضعی در خط ۱ می‌تواند در زمان $\Theta(V+E)$ اجرا شود. فراخوانی

INITIALIZE-SINGLE-SOURCE در خط ۲ به اندازهٔ $\Theta(V)$ زمان می‌برد. برای هر رأس یک تکرار در حلقه *for* خطوط ۵-۳ وجود دارد. برای هر رأس یالهایی که از آن خارج می‌شوند دقیقاً یک بار آزمایش می‌شوند. بنابراین $|E|$ تکرار در حلقه *for* درونی ۵-۴ وجود دارد. (در اینجا از یک تحلیل جمعی استفاده کرده‌ایم.) چون هر تکرار حلقه *for* به اندازه $\Theta(1)$ زمان می‌برد کل زمان اجرا برابر $\Theta(V+E)$ است که لیست در نمایش مجاورتی گراف، یک زمان خطی از اندازه این لیست است.



شکل ۲۴.۵ اجرای الگوریتم برای یافتن کوتاهترین مسیرها در گراف جهت دار بدون دور. یالها از چپ به راست به شکل موضعی مرتب شده‌اند. مبدأ رأس s است. مقادیر d در داخل رأسها نشان داده شده‌اند و یالهای سایه‌دار، مقادیر π را نشان می‌دهند. *(a)* وضعیّت قبل از اولیّن تکرار حلقه *for* در خطوط ۵-۳. *(b)-(g)* وضعیّت بعد از هر تکرار حلقه *for* خطوط ۵-۳. رأسی که در هر تکرار سایه زده شده، به عنوان u در آن تکرار استفاده شده است. مقادیر نشان داده شده در قسمت *(g)*، مقادیر نهایی هستند.

قضیه زیر نشان می‌دهد که روال *DAG-SHORTEST-PATHS* به درستی کوتاهترین مسیرها را محاسبه می‌کند.

قضیه ۲۴.۵

اگر گراف وزن‌دار جهت‌دار $G=(V,E)$ دارای رأس مبدأ s و بدون دور باشد، در پایان روال *DAG-SHORTEST-PATHS* برای همه رئوس $v \in V$ داریم $d[v]=\delta(s,v)$ و زیر گراف ماقبل G_π، درخت کوتاهترین مسیر است.

اثبات ابتدا نشان می‌دهیم در پایان کار الگوریتم برای همه رئوس $v \in V$ داریم $d[v]=\delta(s,v)$ اگر v از s قابل دستیابی نباشد طبق ویژگی نبود مسیر $d[v]=\delta(s,v)=\infty$ اینک فرض کنید که v از s قابل دستیابی است پس یک کوتاهترین مسیر $p=\langle v_0,v_1,...,v_k \rangle$ وجود دارد که $v_0=s$ و $v_k=v$. چون رأسها را به ترتیب مرتب شده موضعی پردازش می‌کنیم یالهای p به ترتیب $(v_0,v_1),(v_1,v_2),...,(v_{k-1},v_k)$ آرام‌سازی می‌شوند. خاصیّت آرام سازی مسیر دلالت می‌کند بر این موضوع که در پایان الگوریتم برای $i=0,1,...,k$ داریم $d[v_i]=\delta(s,v_i)$ و در نهایت بنا به ویژگی زیرگراف ماقبل، G_π درخت کوتاهترین مسیر است. ■

یکی از کاربردهای جالب این الگوریتم در تعیین مسیر بحرانی در تحلیل نمودار *PERT*[1] است. یالها، کارهایی را که باید انجام شوند و وزن یالها زمان لازم برای اجرای یک کار مشخص را نشان می‌دهند. اگر یال (u,v) وارد رأس v شود و یال (v,x) از v خارج شود کار (u,v) باید قبل از کار (v,x) انجام گیرد. یک مسیر داخل این *dag* یک توالی از کارهایی را که باید با نظم خاصی انجام گیرد نشان می‌دهد. مسیر بحرانی، طولانی‌ترین مسیر *dag* است که با طولانی‌ترین زمان اجرای یک توالی از کارهامتناظر است. وزن مسیر بحرانی یک حدّ پایین از زمان اجرای همه کارها است. می‌توانیم مسیر بحرانی را به یکی از دو روش زیر بدست آوریم:

- منفی کردن وزن یالها و اجرای *DAG-SHORTEST-PATHS*
- اجرای *DAG-SHORTEST-PATHS* البته با قرار دادن $-\infty$ به جای ∞ در خط دوم *INITIALIZE-SINGLE-SOURCE* و ">" به جای "<" در روال *RELAX*.

تمرین‌ها

۱– ۲۴.۲ روال *DAG-SHORTEST-PATHS* را روی گراف جهت‌دار ۲۴.۵ اجرا کنید، از r به عنوان مبدأ استفاده کنید.

۲– ۲۴.۲ فرض کنید خط سوم *DAG-SHORTEST-PATHS* را چنین تغییر دهیم:

3 for the first $|V|-1$ vertices, taken in topologically sorted order

نشان دهید که روال باز هم درست عمل می‌کند.

---

۱ – "PERT" مخفّف "program evaluation and review technique" به معنای " تکنیک ارزش گذاری و بازبینی برنامه" است.

۳-۲۴.۲ ساختار نمودار *PERT* داده شده در فوق، تا اندازه‌ای غیر طبیعی است. معمول‌تر آن بود که رأسها کارها و یالها، محدودیتهای پشت سر هم را نشان دهند: به عبارت دیگر یال (u,v) بیان می‌کرد کار u باید قبل از کار v انجام شود. بنابراین وزنها هم باید به رأسها انتساب داده می‌شدند نه به یالها. روال *DAG-SHORTEST-RATHS* را چنان تغییر دهید که این روال طولانی‌ترین مسیر در گراف جهتدار بدون دور با رأسهای وزن‌دار را با مرتبه زمانی خطی پیدا کند.

۴-۲۴.۲ الگوریتمی کارآمد ارائه دهید که تعداد کل مسیرهای یک گراف جهت‌دار بدون دور را محاسبه کند. الگوریتم خود را تحلیل کنید.

۲۴.۳ الگوریتم *Dijkstra*

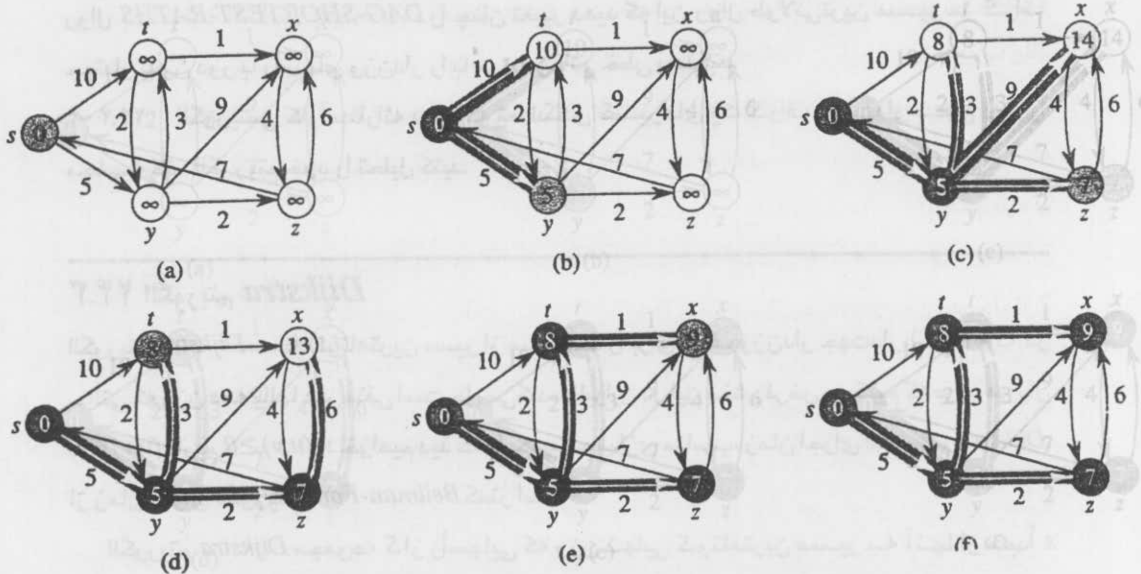
الگوریتم *Dijkstra* مسئله کوتاهترین مسیر از مبدأ واحد را برای گراف وزن‌دار جهت‌دار $G=(V,E)$ در حالتی که وزن همه یالها غیر منفی است حل می‌کند. بنابراین در این بخش فرض می‌کنیم که برای هر یال $(u,v)\in E$ داریم $w(u,v)\geq 0$. خواهیم دید که با یک پیاده سازی مناسب، زمان اجرای الگوریتم *Dijkstra* از زمان اجرای الگوریتم *Bellman-Ford* کمتر است.

الگوریتم *Dijkstra* مجموعه S از رأسهایی که وزن نهایی کوتاهترین مسیر به آنها از مبدأ s مشخص شده را نگه می‌دارد. الگوریتم، مکرراً رأس $u\in V-s$ با مینیمم برآورد کوتاهترین مسیر را انتخاب می‌کند، u را به S اضافه می‌کند و همه یالهایی که از u خارج شده‌اند را آرام سازی می‌کند. در پیاده سازی زیر از صف مینیمم اولویت Q برای رأسها که به وسیله d هایشان مقدار دهی شده‌اند استفاده می‌کنیم.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← Ø
3  Q ← V[G]
4  while Q ≠ Ø
5      do u ← EXTRACT-MIN(Q)
6         S ← S ∪ {u}
7         for each vertex v ∈ Adj[u]
8             do RELAX(u, v, w)
```

الگوریتم *Dijkstra* یالها را طبق شکل ۲۴.۶ آرام سازی می‌کند. خط ۱، مقادیر d و π را مطابق معمول مقدار دهی اولیّه می‌کند و خط ۲، ϕ را به مجموعه S نسبت می‌دهد. در شروع تکرار حلقه *while* خطوط ۴-۸، الگوریتم رابطه ثابت $Q=V-S$ الگوریتم را حفظ می‌کند. خط ۳، صف مینیمم اولویّت Q را چنان مقدار دهی می‌کند که Q شامل همه رأسهای V باشد: چون در آن زمان $S=\phi$ است لذا رابطه ثابت، بعد از

خط ۳ درست می‌باشد. هر بار در حلقه *while* خطوط ۸-۴، یک رأس u از $Q=V-S$ استخراج، و به مجموعه S مینیم اولویّت اضافه می‌شود، به موجب آن رابطه ثابت حفظ می‌شود. (اوّلین بار در حلقه داریم $u=s$).

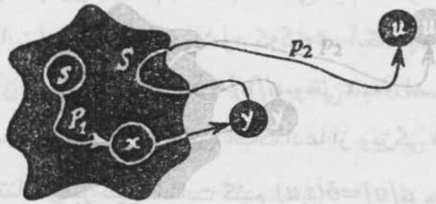


شکل ۲۴.۶ اجرای الگوریتم *Dijkstra* مبدأ s سمت چپ‌ترین رأس است. برآوردهای کوتاهترین مسیر، داخل رأسها نمایش داده شده و یالهای سایه‌دار مقادیر ماقبل را نشان می‌دهند. رأسهای سیاه در مجموعه S و رأسهای سفید در صف مینیم اولویّت $Q=V-S$ قرار دارند. *(a)* وضعیّت گراف درست قبل از اوّلین تکرار حلقه *while* خطوط ۸-۴. رأس سایه‌زده شده، مینیمم d را داراست و به عنوان رأس u در خط ۵ انتخاب شده است. *(b)-(f)* وضعیّت گراف بعد از هر تکرار متوالی حلقه *while* هر رأس سایه‌زده شده در هر قسمت به عنوان رأس u در خط ۵ تکرار بعدی انتخاب می‌شود. مقادیر d و π نشان داده شده در هر قسمت *(f)* مقادیر نهایی هستند.

بنابراین رأس u کوچکترین برآورد کوتاهترین مسیر از هر رأس مجموعه $V-S$ را دارا است. سپس خطوط ۸-۷، هر یال (u,v) که از u خارج می‌شود را آرام سازی می‌کنند و بنابراین مقادیر $d[v]$ و $\pi[v]$ تغییر می‌کنند، اگر کوتاهترین مسیر بتواند از رأس u به رأس v حرکت کند. مشاهده می‌شود که رأسها بعد از خط ۳ دیگر در Q درج نمی‌شوند و هر رأس از Q دقیقاً یک بار استخراج و به S اضافه می‌شود، پس حلقه *while* خطوط ۸-۴ دقیقاً $|V|$ بار تکرار می‌شود.

چون الگوریتم *Dijkstra* همیشه "سبک‌ترین" یا "نزدیکترین" رأس در $V-S$ را انتخاب و به مجموعه S اضافه می‌کند می‌گوییم از تدبیر حریصانه استفاده می‌کند. تدبیر حریصانه با جزئیات در فصل ۱۶ بیان شده، اما لازم نیست برای فهمیدن الگوریتم *Dijkstra* فصل ۱۶ را مطالعه کنید. تدابیر حریصانه همیشه نتایج بهینه در حالت کلی ارائه نمی‌دهند اما قضیه بعد و قضیه فرعی آن نشان می‌دهد که الگوریتم

Dijkstra در حقیقت کوتاهترین مسیر را محاسبه می‌کند. کلید حل این مسأله این است که نشان دهیم هربار که یک رأس u به مجموعه S اضافه می‌شود، $d[u] = \delta(s,u)$ است.



شکل ۲۴.۷ اثبات قضیّه ۲۴.۶ مجموعه S قبل از اضافه شدن رأس u به آن غیر تهی است. کوتاهترین مسیر p از مبدأ s به رأس u می‌تواند به $s \overset{p_1}{\leadsto} x \rightarrow y \overset{p_2}{\leadsto} u,$ تجزیه شود که y اولین رأس در مسیر است که در S نیست و $x \in S$ بلافاصله قبل از y قرار می‌گیرد. رئوس y و x مجزا هستند امّا ممکن است داشته باشیم $s=x$ یا $y=u$. مسیر $p2$ ممکن است دوباره وارد S بشود یا نشود.

قضیّه ۲۴.۶ (صحّت الگوریتم *Dijkstra*)

الگوریتم *Dijkstra* که روی گراف وزن‌دار جهت‌دار $G=(V,E)$ با وزن غیر منفی و تابع وزن w و مبدأ s اجرا می‌شود، با $d[u]=\delta(s,u)$ برای همهٔ رأسها $u \in V$ خاتمه می‌یابد.

اثبات از ثابت حلقه زیر استفاده می‌کنیم:

در شروع هر تکرار حلقه *while* در خطوط ۸-۴ برای هر رأس $v \in S$ داریم $d[v]=\delta(s,v)$ کافی است نشان دهیم برای هر رأس $u \in v$ در زمانی که u به S اضافه می‌شود داریم $d[u]=\delta(s,u)$ یک بار که نشان دهیم $d[u]=\delta(s,u)$ می‌توانیم با اتّکا به ویژگی حدّ بالا نشان دهیم تساوی ذکر شده در تمام زمانهای بعد از آن برقرار است.

مقدار دهی: $S=\phi$ و ثابت مورد نظر $Q \neq \phi$ به طور بدیهی درست است.

نگهداری: می‌خواهیم نشان دهیم که در هر تکرار، برای رأس اضافه شده به مجموعه S، $d[u]=\delta(s.u)$ به منظور ایجاد تناقض فرض کنید u اولیّن رأسی باشد که وقتی به S اضافه می‌شود $d[u]\neq\delta(s,u)$ توجّه خود را معطوف می‌کنیم به وضعیّت الگوریتم در ابتدای تکرار حلقه *while* که در آن u به S اضافه می‌شود و تناقض $d[u]=\delta(s,u)$ در آن هنگام با آزمایش کوتاهترین مسیر از s به u ایجاد می‌شود. باید داشته باشیم $u \neq s$ چون s اولین رأسی است که به مجموعه S اضافه شود و در آن هنگام $d[s]=\delta(s,s)=0$ چون $u \neq s$ دقیقاً قبل از اضافه شدن u به S داریم $S \neq \phi$ باید مسیری از s به u وجود داشته باشد و گرنه بنا به ویژگی نبود مسیر داریم $d[u]=\delta(s,u)=\infty$ که از فرض $d[u]\neq\delta(s,u)$ تخطی می‌کند. چون حداقل یک مسیر وجود دارد، پس یک کوتاهترین مسیر p نیز از s به u وجود دارد. قبل از اضافه شدن u به s مسیر p یک رأس در مجموعه S به نام s را به یک رأس در مجموعه $V\text{-}S$ به نام

u متصل می‌کند. در نظر بگیرید که $y \in V\text{-}S$ اولین رأس در طول مسیر p و $x \in S$ رأس ماقبل y باشد. بنابراین همان طور که در شکل ۲۴.۷ نشان داده شده، مسیر p می‌تواند به $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$

تجزیه شود.(ممکن است p_1 یا p_2 بدون یال هم باشند.)ادعا می‌کنیم که در هنگام اضافه شدن u به S داریم $d[y]=\delta(s,y)$. برای اثبات این ادعا مشاهده می‌شود که $x \in S$. آنگاه چون u بعنوان اولین رأس انتخاب شده،درهنگام اضافه کردن‌آن به S داشتیم $d[u] \neq \delta(s,u)$ ،وقتی x به S اضافه شد داشتیم $d[x]=\delta(s,x)$. یال (x, y) در آن موقع آرام‌سازی شده و بنابراین، اثبات ادعا از ویژگی همگرایی انجام می‌شود.

اکنون می‌توانیم به یک تناقض برسیم تا ثابت کنیم $d[u]=\delta(s,u)$ چون y قبل از u در کوتاهترین مسیر s به u واقع شده و وزن هم یالها غیر منفی هستند (از جمله، یالهای روی مسیر p_2)، داریم $\delta(s,y) \leq \delta(s,u)$ و بنابراین :

$$\begin{aligned} d[y] &= \delta(s, y) \\ &\leq \delta(s, u) \qquad (۲۴.۲) \\ &\leq d[u] \end{aligned}$$

بنا به ویژگی حد بالا

امّا چون هر دو رأس y و u در هنگامی که u در خط ۵ انتخاب شد در مجموعه $V\text{-}S$ قرار داشتند داریم $d[u] \leq d[y]$ بنابراین دو نامساوی بالا در حقیقت دو تساوی هستند.

$$d[y] = \delta(s, y) = \delta(s, u) = d[u] .$$

در نتیجه $d[u]=\delta(s,u)$ که با انتخابمان از u در تناقض قرار دارد. نتیجه می‌گیریم هنگامی که u به S اضافه می‌شود $d[u]=\delta(s,u)$ و این تساوی بعد از این مرحله نیز حفظ می‌شود.

خاتمه: در پایان $Q=\phi$ است که همراه با رابطه ثابت قبل یعنی $Q=V\text{-}S$ دلالت می‌کند بر این که $S=V$ بنابراین برای همه $u \in V$ داریم $d[u]=\delta(s,u)$ ■

قضیّه فرعی ۲۴.۷

اگر الگوریتم *Dijkstra* را روی یک گراف جهت‌دار و وزن‌دار $G=(V,E)$ با تابع وزن غیر منفی w و مبدأ s اجرا کنیم در پایان کار زیرگراف ماقبل G_π یک درخت کوتاهترین مسیر با ریشه s است.

اثبات از قضیّه ۲۴.۶ و خاصیّت زیرگراف ماقبل انجام می‌گیرد.

تحلیل الگوریتم

سرعت الگوریتم *Dijkstra* چقدر است؟ الگوریتم با فراخوانی سه عمل صف اولویّت، صف مینیمم اولویّت Q را حفظ می‌کند: *INSERT* (در خط ۳)، *EXTRACT-MIN* (خط ۵)، و *DECREASE-KEY* (داخل *RELAX* که در خط ۸ آمده). *INSERT* و *EXTRACT-MIN* برای هر رأس یک بار صدا زده می‌شوند. چون هر رأس $v \in V$ دقیقاً یک بار به S اضافه می‌شود، هر یال در لیست مجاورتی *Adj[v]* در

طول اجرای الگوریتم دقیقاً یک بار در حلقه *for* خطوط ۸-۷، آزمایش می‌شود. چون تعداد کل یالها در لیست مجاورتی، $|E|$ است کلاً $|E|$ تکرار در حلقه *for* داریم و بنابراین حداکثر $|E|$ عمل *DECREASE-KEY* وجود دارد. (مجدداً مشاهده می‌شود که از تحلیل جمعی استفاده می‌شود)

زمان اجرای الگوریتم *Dijkstra* به چگونگی پیاده سازی صف مینیمم اولویّت بستگی دارد. ابتدا حالتی را در نظر بگیرید که در آن صف مینیمم اولویّت به وسیله شماره‌گذاری رأسها از 1 تا $|V|$ نگهداری می‌شود. به سادگی $d[v]$ را در vامین ورودی یک آرایه ذخیره می‌کنیم. هر عمل *INSERT* و *DECREASE-KEY* به اندازه $O(1)$ زمان می‌برد و هر عمل *EXTRACT-MIN* به اندازه $O(V)$ به طول می‌انجامد (چون مجبوریم در طول آرایه جستجو کنیم)، و زمان کل برابر است با $O(V^2+E)=O(V^2)$.

اگر گراف به حدّ کافی پراکنده باشد - در حالت خاص، $E=o(V^2/lgV)$ - پیاده سازی صف مینیمم اولویّت با استفاده از *min-heap* دودویی عملی است. (همان طور که در بخش ۶.۵ گفته شد یک نکته مهم پیاده سازی این است که رأسها و عناصر متناظر در *heap* باید یک پیوند به یکدیگر را نگه دارند) هر عمل *EXTRACT-MIN* به اندازهٔ $O(lgV)$ زمان می‌برد. مانند قبل، این عمل $|V|$ بار انجام می‌شود. زمان ساختن *min-heap* دودویی $O(V)$ است. هر عمل *DECREASE-KEY* به اندازهٔ $O(lgV)$ زمان می‌برد و حداکثر $|E|$ بار انجام می‌گیرد. بنابراین کل زمان اجرا، $O((V+E)lgV)$ است که اگر همه رأسها از مبدأ قابل دستیابی باشند برابر است با $O(ElgV)$. این زمان اجرا نسبت به پیاده سازی $O(V^2)$، یک پیشرفت محسوب می‌شود اگر $E=o(V^2/lgV)$. در واقع می‌توانیم با پیاده سازی صف اولویت با *heap* فیبوناچی (فصل ۲۰)، به زمان $O(VlgV+E)$ برسیم. هزینه سرشکن شده هر $|V|$ عمل *EXTRACT-MIN*، $O(lgV)$ و هر فراخوانی *DECREASE-KEY* به تعداد حداکثر $|E|$، برابر $O(1)$ است. پیشرفت *heap*های فیبوناچی از آن جهت حاصل شده است که در الگوریتم *Dijkstra* نوعاً فراخوانی‌های *DECREASE-KEY* خیلی بیشتر از فراخوانی‌های *EXTRACT-MIN*، زمان هر عمل *DECREASE-KEY* را به $o(lgV)$ کاهش دهد پیاده سازی مجانبی سریعتری نسبت به پیاده‌سازی با *heap* دودویی ارائه می‌کند.

الگوریتم *Dijkstra* شباهتهایی با جستجوی اوّل سطح (بخش ۲۲.۲) و الگوریتم *Prim* برای محاسبه درخت‌های پوشای مینیمم (بخش ۲۳.۲) دارد. در مورد شباهت این الگوریتم با جستجوی اول سطح باید گفت مجموعه S دراین الگوریتم با مجموعه رأسهای سیاه در جستجوی اوّل سطح متناظر است؛ همان طور که رأسهای S وزن کوتاهترین مسیر نهایی خود را نگه می‌دارند، رأسهای سیاه در الگوریتم جستجوی اوّل سطح، فاصله خود از نظر سطح را نگه می‌دارند. این الگوریتم با الگوریتم *Prim* ازاین نظر شباهت دارد که در هر دو الگوریتم از یک صف مینیمم اولویّت برای یافتن "سبک‌ترین" رأس، خارج یک مجموعه داده شده (مجموعه S در الگوریتم *Dijkstra* و درخت رشد کرده الگوریتم *Prim*) استفاده

می‌گردد. سپس این رأس به مجموعه اضافه می‌گردد و طبق آن وزن رأسهای باقی مانده خارج از مجموعه، تنظیم می‌شوند.

تمرین‌ها

۱- ۲۴.۳ الگوریتم *Dijkstra* را روی گراف جهت‌دار شکل ۲۴.۲ اجرا کنید، ابتدا از رأس s و سپس از رأس z به عنوان مبدأ استفاده کنید. مانند شکل ۲۴.۶ مقادیر d و π و رأسهای مجموعه S بعد از هر تکرار حلقه *while* را نمایش دهید.

۲- ۲۴.۳ یک مثال ساده از یک گراف جهت‌دار با یالهایی با وزن منفی که برای آن الگوریتم *Dijkstra* جواب نادرستی راتولید می‌کند، ارائه دهید. چرا اثبات قضیهٔ ۲۴.۶ زمانی که یالهای با وزن منفی هم وجود دارند استفاده نمی‌شود؟

۳- ۲۴.۳ فرض کنید که خط ۴ الگوریتم *Dijkstra* را به شکل زیر تغییر دهیم

4 **while** $|Q| > 1$

این تغییر باعث می‌شود که حلقه *while* به جای $|V|$ بار، $|V|-1$ بار اجرا شود. آیا الگوریتم باز هم درست است؟

۴- ۲۴.۳ یک گراف جهت‌دار $G=(V,E)$ که در آن هر یال $(u,v)\in E$ یک مقدار مربوطه $r(u,v)$ دارد داده شده است: $r(u,v)$ یک عدد حقیقی در بازه $0\leq r(u,v)\leq 1$ است که میزان اطمینان از وجود ارتباط بین v و u رانشان می‌دهد. $r(u,v)$ را به عنوان احتمال این که کانال u به v از بین نمی‌رود تفسیر می‌کنیم و فرض می‌کنیم که این احتمالها مستقل هستند. یک الگوریتم کارآمد برای پیدا کردن مطمئن‌ترین مسیر از دو رأس داده شده ارائه دهید.

۵- ۲۴.۳ گراف $G=(V,E)$ را یک گراف وزن‌دار وجهت‌دار با تابع وزن $w:E\rightarrow\{1,2,...,w\}$ برای بعضی مقادیر مثبت w در نظر بگیرید و فرض کنید هیچ دو رأسی دارای وزن مساوی کوتاهترین مسیر از مبدأ s نباشند. اینک فرض کنید گراف بدون وزن جهت‌دار $G(V\cup V',E')$ با جایگزینی هر یال $(u,v)\in E$ با $w(u,v)$ واحد وزن تعریف کنیم. G' چند یال دارد؟ اینک فرض کنید که یک جستجوی اوّل سطح روی G' انجام دهیم. نشان دهید ترتیبی که طبق آن رأسهای V در جستجوی اوّل سطح گراف G' سیاه می‌شوند همان ترتیبی است که رأسهای V در خط ۵ روال *DIJKSTRA* ازصف اولویّت مینیمم در هنگام اجرای آن روی G خارج می‌شوند.

۶- ۲۴.۳ گراف وزن‌دار جهت‌دار $G=(V,E)$ با تابع وزن $w:E\rightarrow\{0,1,..w\}$ برای مقادیر غیر منفی w را در نظر بگیرید. الگوریتم *Dijkstra* را چنان تغییر دهید که کوتاهترین مسیرها از رأس مبدأ s را در زمان $O(WV+E)$ محاسبه کند.

۷- ۲۴.۳ الگوریتمی که در تمرین ۶- ۲۴.۳ ارائه دادید را چنان تغییر دهید که در زمان

اجرا شود. (راهنمایی: چند برآورد کوتاهترین مسیر مجزا می‌تواند در هر زمان در $O((V+E)lgW)$ وجود داشته باشد؟) $V\text{-}S$

۸- ۲۴.۳ فرض کنید گراف وزن‌دار جهت‌دار $G=(V,E)$ داده شده که در آن یالهایی که از رأس s خارج می‌شوند ممکن است مقادیر منفی داشته باشند و وزن بقیه یالها غیر منفی است و هیچ دوری با وزن منفی وجودندارد. در مورد این که الگوریتم *Dijkstra* می‌تواند به درستی کوتاهترین مسیرها از مبدأ s را بیابد بحث کنید.

۲۴.۴ محدودیتهای تفاضلی [1] و کوتاهترین مسیرها

در مسأله برنامه‌سازی خطی، می‌خواهیم یک تابع خطی را به یک مجموعه از نامساویهای خطی بهینه کنیم. دراین بخش به یک حالت خاص از برنامه‌سازی خطی می‌پردازیم که می‌تواند برای یافتن کوتاهترین مسیرها از یک مبدأ واحد استفاده شود. سپس مسئله کوتاهترین مسیر از مبدأ واحدی که حاصل می‌شود می‌تواند با استفاده از الگوریتم *Bellman-Ford* حل شود که بدان وسیله مسأله برنامه‌سازی خطی نیز حل می‌گردد.

برنامه‌سازی خطی [2]

در *مسئله برنامه‌سازی خطی* یک ماتریس $m\times n$ به نام A، یک m- برداری b و یک n- برداری c داده شده است. می‌خواهیم یک بردار x شامل n عنصر پیدا کنیم که مقدار تابع مقصد [3] $\sum_{i=1}^{n} c_i x_i$ را با شرط $Ax\leq b$ ماکزیمم کند.

الگوریتمهای برنامه‌سازی خطی وجود دارند که با مرتبه زمانی چند جمله‌ای اجرا می‌گردند. به چندین دلیل فهمیدن طرح مسایل برنامه‌سازی خطی، مهم است. اول دانستن این موضوع که یک مسئله داده شده می‌تواند به یک مسئله برنامه‌سازی خطی با سایز چند جمله‌ای تبدیل شود بدین معنی است که یک الگوریتم با مرتبه زمانی چند جمله‌ای برای این مسئله وجود دارد. دوّم این که حالتهای خاص مختلفی از برنامه‌سازی خطی وجود دارند که برای آنها الگوریتمهای سریعتری موجود می‌باشند. برای مثال، همان طور که در این بخش نشان داده شده، مسئله کوتاهترین مسیر از یک مبدأ واحد، یک حالت خاص از برنامه‌سازی خطی است. دیگر مسایل شامل مسئله کوتاهترین مسیر از جفت واحد (تمرین ۴- ۲۴.۴) و مسأله ماکزیمم جریان (تمرین ۸- ۲۴.۱) نیز می‌توانند به برنامه‌سازی خطی تبدیل شوند.

---

1. difference constraints
2. linear programming
3. objective function

گاهی اوقات در واقع توجه چندانی به تابع مقصد نداریم بلکه فقط می‌خواهیم یک جواب ممکن [1] را بیابیم، به عبارت دیگر بردار x که شرط $Ax\leq b$ را تحقق می‌بخشد، یا این که تعیین کنیم هیچ راه حل ممکنی وجود ندارد. به یکی از چنین مسایل ممکنی می‌پردازیم.

سیستم‌های محدودیت‌های تفاضلی

در یک سیستم محدودیت‌های *تفاضلی* ، هر ردیف ماتریس برنامه‌سازی خطی A دارای یک 1 و دارای یک -1 است و دیگر ورودیهای A صفر هستند. بنابراین محدودیت‌های حاصل از $Ax\leq b$، یک مجموعه از m محدودیت تفاضلی شامل n مجهول است که در آن هر محدودیت، یک نامساوی خطی ساده به فرم زیر است.

$x_j - x_i \leq b_k$,

که در آن $1\leq i,j\leq n$ و $1\leq k\leq m$ است.

برای مثال، مسئله یافتن ۵- برداری $x=(x_i)$ که شرط زیر را برقرار کند، در نظر بگیرید.

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

این مسأله معادل یافتن مجهولات x_i برای $i=1,2,...,5$ است بطوری که محدودیت تفاضلی زیر برقرار شود:

$$x_1 - x_2 \leq 0 , \qquad (۲۴.۳)$$
$$x_1 - x_5 \leq -1 , \qquad (۲۴.۴)$$
$$x_2 - x_5 \leq 1 , \qquad (۲۴.۵)$$
$$x_3 - x_1 \leq 5 , \qquad (۲۴.۶)$$
$$x_4 - x_1 \leq 4 , \qquad (۲۴.۷)$$
$$x_4 - x_3 \leq -1 , \qquad (۲۴.۸)$$
$$x_5 - x_3 \leq -3 , \qquad (۲۴.۹)$$
$$x_5 - x_4 \leq -3 . \qquad (۲۴.۱۰)$$

1. *feasible solution*

یک جواب این مسئله $x=(-5,-3,0,-1,-4)$ است که درستی آن مستقیماً با چک کردن هر نامساوی ثابت می‌شود. در واقع بیش از یک جواب برای این مسأله وجود دارد. یک جواب دیگر $x'=(0,2,5,4,1)$ است. این دو جواب به هم مربوطند: هر درایه x' به اندازه ۵ واحد بزرگتر از درایه متناظرش در x است. این موضوع تصادفی نیست.

لم ۲۴.۸

فرض کنید $x=(x_1,x_2,...,x_n)$ یک جواب سیستم محدودیت‌های تفاضلی $Ax\leq b$ و d هر ثابت دلخواه باشد. بنابراین $x+d=(x_1+d,x_2+d,...,x_n+d)$ نیز یک جواب $Ax\leq b$ است.

اثبات برای هر x_i و x_j داریم $(x_j+d)-(x_i+d)=x_j-x_i$. بنابراین اگر x در شرط $Ax\leq b$ صدق کند $x+d$ هم صدق می‌کند. ■

سیستم‌های محدودیت‌های تفاضلی در کاربردهای مختلف پیش می‌آیند. برای مثال، مجهولهای x_i ممکن است زمانهایی باشند که در آن زمانها، رویدادهایی اتفاق می‌افتند. هر محدودیت می‌تواند چنین بیان دارد که یک مقدار حداقل زمان و یک مقدار حداکثر زمانی بین دو رویداد قرار دارد. شاید این رویدادها، کارهایی باشند که باید در طول زمان مونتاژ یک محصول انجام گیرند. اگر مثلاً از یک چسبنده در زمان x_1 استفاده می‌کنیم و قرار دادن آن 2 ساعت به طول بینجامد برای نصب یک قطعه در زمان x_2 باید تا قرار گرفتن کامل چسبنده منتظر شویم لذا این شرط را داریم که $x_2\geq x_1+2$ یا به طور معادل $x_1-x_2\leq -2$ از سوی دیگر ممکن است که لازم باشد قطعه پس از این که قرار دادن چسبنده در نیمه راه است، نصب گردد که در این حالت دو شرط $x_2\geq x_1$ و $x_2\leq x_1+1$ را داریم، یا به طور معادل $x_2-x_1\leq 1$ و $x_1-x_2\leq 0$.

گراف‌های محدودیت[1]

خوب است سیستم محدودیت‌های تفاضلی را از نقطه نظر تئوری گراف‌ها تفسیر کنیم. ایده این است که در یک سیستم محدودیت‌های تفاضلی $Ax\leq b$، ماتریس برنامه‌سازی خطی A با ابعاد $m\times n$ می‌تواند به عنوان ترانهاده ماتریس همجواری یک گراف با n رأس و m یال (تمرین ۷-۲۲.۱۰) در نظر گرفته شود. هر رأس v_i در گراف برای $i=1,2,..,n$ متناظر با یکی از n متغیر مجهول x_i است. هر یال جهت‌دار در گراف، متناظر با یکی از m نامساوی که شامل دو مجهولند می‌باشد.

به طور دقیق‌تر در سیستم محدودیت‌های تفاضلی $Ax\leq b$، *گراف محدودیت* متناظر، یک گراف وزن‌دار جهت‌دار $G=(V,E)$ است که

1.constraint graphs

$V = \{v_0, v_1, \dots, v_n\}$

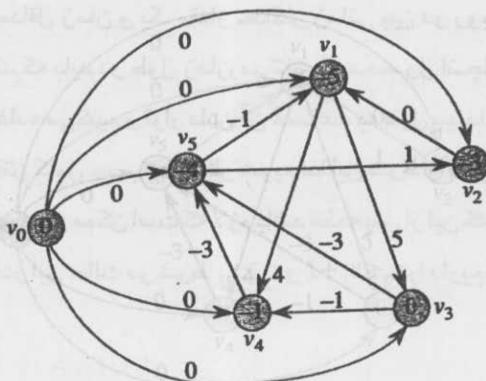
و

$E = \{(v_i, v_j) : x_j - x_i \le b_k$ یک محدودیت است $\}$

$\cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), \dots, (v_0, v_n)\}$

رأس اضافی v_0، ایجاد شده تا همان طور که خواهیم دید تضمین کند که همه رأسهای دیگر از آن قابل دستیابی هستند. بنابراین مجموعه رأس V تشکیل شده است از یک رأس v_i برای هر مجهول x_i و یک رأس اضافه v_0. مجموعه یال E تشکیل شده است از یک یال برای هر محدودیت تفاضلی به علاوه یک یال (v_0, v_i) برای هر مجهول x_i اگر $x_j - x_i \le b_k$ یک محدودیت تفاضلی باشد وزن یال (v_i, v_j) برابر است با $w(v_i, v_j) = b_k$. وزن هر یالی که از v_0 خارج می‌شود 0 است. شکل ۲۴.۸ گراف محدودیت سیستم محدویت‌های تفاضلی (۲۴.۱۰) - (۲۴.۳) را نشان می‌دهد.

قضیّه بعد نشان می‌دهد که می‌توانیم یک جواب برای سیستم محدودیت‌های تفاضلی را به وسیله یافتن وزنهای کوتاهترین مسیر در گراف محدودیت متناظر آن پیدا کنیم.



شکل ۲۴.۸ گراف محدودیت متناظر با سیستم محدودیت‌های تفاضلی (۲۴.۱۰) - (۲۴.۳). مقدار $\delta(v_0, v_i)$ در داخل هر رأس v_i نشان داده شده است. یک جواب ممکن برای این سیستم $x = (-5, -3, 0, -1, -4)$ می‌باشد.

قضیه ۲۴.۹

در سیستم محدودیت‌های تفاضلی $Ax \le b$ فرض کنید $G=(V,E)$ گراف محدودیت متناظر آن باشد. اگر G دور با وزن منفی نداشته باشد آنگاه

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \dots, \delta(v_0, v_n)) \qquad (۲۴.۱۱)$$

یک جواب ممکن برای این سیستم است. اگر G شامل دور با وزن منفی باشد آنگاه هیچ جوابی برای سیستم وجود ندارد.

اثبات ابتدا نشان می‌دهیم که اگر گراف محدودیت، دور با وزن منفی نداشته باشد تساوی (۲۴.۱۱) یک جواب ممکن را ارائه می‌دهد. یال $(v_i,v_j)\in E$ را در نظر بگیرید. با استفاده از نامساوی مثلث داریم $\delta(v_0,v_j)\leq\delta(v_0,v_i)+w(v_i,v_j)$، یا به طور معادل $\delta(v_0,v_j)-\delta(v_0,v_i)\leq w(v_i,v_j)$. بنابراین قرار دادن $x_i=\delta(v_0,v_i)$ و $x_j=\delta(v_0,v_j)$ محدودیت تفاضلی $x_j-x_i\leq w(v_i,v_j)$ که متناظر با یال (v_i,v_j) است را برقرار می‌کند.

اینک نشان می‌دهیم که اگر گراف محدودیت، دارای دور با وزن منفی باشد سیستم محدودیت‌های تفاضلی هیچ جواب ممکنی ندارد و بدون از دست دادن کلیّت فرض کنید $c=<v_1,v_2,...,v_k>$ دوری با وزن منفی باشد که در آن $v_1=v_k$. (رأس v_0 نمی‌تواند در دور c قرار گیرد چون یال ورودی ندارد) دور c متناظر با محدودیت‌های تفاضلی زیر است:

$$
\begin{aligned}
x_2 - x_1 &\leq w(v_1, v_2) , \\
x_3 - x_2 &\leq w(v_2, v_3) , \\
&\vdots \\
x_k - x_{k-1} &\leq w(v_{k-1}, v_k) , \\
x_1 - x_k &\leq w(v_k, v_1) .
\end{aligned}
$$

فرض کنید جوابی برای x وجود دارد که تمام این k نامساوی را برقرار می‌کند. این جواب باید نامساوی که از جمع این k نامساوی حاصل می‌شود را نیز برقرار کند. اگر طرف‌های چپ نامساوی را با هم جمع کنیم هر مجهول x_i یک بار جمع و یک بار کم می‌شود. بنابراین حاصل سمت چپ نامساوی نهایی برابر صفر است. امّا حاصل جمع طرف راست نامساوی‌ها برابر $w(c)$ است، لذا داریم $0\leq w(c)$. امّا چون c دوری با وزن منفی است $w(c)<0$، و به این تناقض می‌رسیم که $0\leq w(c)<0$. ■

حل سیستم محدودیت‌های تفاضلی

قضیّه ۲۴.۹ به ما می‌گوید که می‌توانیم از الگوریتم *Bellman-Ford* برای حل سیستم محدودیت‌های تفاضلی استفاده کنیم. چون در گراف محدودیت یالهایی از v_0 به همه رأسهای دیگر وجود دارد، بنابراین هر دور با وزن منفی در این گراف از v_0 قابل دستیابی است. اگر الگوریتم *Bellman-Ford* مقدار *TRUE* را برگرداند وزن‌های کوتاهترین مسیر، یک جواب ممکن برای سیستم است. برای مثال در شکل ۲۴.۸ وزنهای کوتاهترین مسیر، جواب $x=(-5,-3,0,-1,-4)$ را تولید می‌کنند و با استفاده از لم ۲۴.۸ به ازای هر مقدار ثابت d، $x=(d-5,d-3,d,d-1,d-4)$ هم یک جواب ممکن است. اگر الگوریتم *Bellman-Ford* مقدار *FALSE* را برگرداند هیچ جواب ممکنی برای سیستم محدودیت‌های تفاضلی وجود ندارد.

یک سیستم محدودیت‌های تفاضلی با m محدودیت روی n مجهول، گرافی با $n+1$ رأس و $n+m$ یال تولید می‌کند. لذا با استفاده از الگوریتم *Bellman-Ford* می‌توانیم سیستم را در زمان $O((n+1)(n+m))=O(n^2+nm)$ حل کنیم که تمرین ۵- ۲۴.۴ از شما می‌خواهد الگوریتم را چنان تغییر دهید که در زمان $O(nm)$ اجرا شود حتّی اگر m خیلی کوچکتر از n باشد.

تمرین‌ها

۱- ۲۴.۴ برای سیستم محدودیت‌های تفاضلی زیر، یک جواب ممکن بیابید یا مشخص کنید برای آن جوابی وجود ندارد:

$$\begin{aligned}
x_1 - x_2 &\le 1,\\
x_1 - x_4 &\le -4,\\
x_2 - x_3 &\le 2,\\
x_2 - x_5 &\le 7,\\
x_2 - x_6 &\le 5,\\
x_3 - x_6 &\le 10,\\
x_4 - x_2 &\le 2,\\
x_5 - x_1 &\le -1,\\
x_5 - x_4 &\le 3,\\
x_6 - x_3 &\le -8.
\end{aligned}$$

۲- ۲۴.۴ برای سیستم محدودیت‌های تفاضلی زیر یک جواب ممکن بیابید یا مشخص کنید که برای آن جوابی وجود ندارد:

$$\begin{aligned}
x_1 - x_2 &\le 4,\\
x_1 - x_5 &\le 5,\\
x_2 - x_4 &\le -6,\\
x_3 - x_2 &\le 1,\\
x_4 - x_1 &\le 3,\\
x_4 - x_3 &\le 5,\\
x_4 - x_5 &\le 10,\\
x_5 - x_3 &\le -4,\\
x_5 - x_4 &\le -8.
\end{aligned}$$

۳- ۲۴.۴ آیا وزن هر کوتاهترین مسیر از رأس جدید v_0 در گراف محدودیت می‌تواند مثبت باشد؟ توضیح دهید.

۴- ۲۴.۴ مسئله کوتاهترین مسیر جفت واحد را به شکل یک برنامه خطی بیان کنید.

۲۴.۴-۵ الگوریتم *Bellman-Ford* را اندکی تغییر دهید تا هنگامی که از آن در حل سیستم محدودیت‌های تفاضلی با m نامساوی و n مجهول استفاده شد، در زمان $O(nm)$ اجرا شود.

۲۴.۴-۶ فرض کنید علاوه بر سیستم محدودیت‌های تفاضلی می‌خواهیم محدودیت‌های مساوی به شکل $x_i=x_j+b_k$ داشته باشیم. نشان دهید چگونه الگوریتم *Bellman-Ford* می‌تواند برای حل این سیستم محدودیت متفاوت، وفق داده شود.

۲۴.۴-۷ نشان چگونه یک سیستم محدودیت‌های تفاضلی می‌تواند به وسیله اجرای الگوریتمی مشابه *Bellman-Ford* روی یک گراف محدودیت بدون رأس اضافی v_0 حل شود.

۲۴.۴-۸ * فرض کنید $Ax\leq b$ یک سیستم با m محدودیت تفاضلی و n مجهول باشد. نشان دهید زمانی که الگوریتم *Bellman-Ford* روی گراف محدودیت متناظر آن اجرا می‌شود، مقدار $\sum_{i=1}^{n} x_i$ با شرایط $Ax\leq b$ و $x_i\leq 0$ ها را ماکزیمم می‌کند.

۲۴.۴-۹ * نشان دهید وقتی الگوریتم *Bellman-Ford* روی گراف محدودیت سیستم محدودیت‌های تفاضلی $Ax\leq b$ اجرا می‌شود مقدار $(max\{x_i\}-min\{x_i\})$ مربوط به $Ax\leq b$ را مینیمم می‌کند. توضیح دهید چگونه این واقعیت به دست می‌آید اگر الگوریتم برای زمان‌بندی کارهای ساختاری استفاده شود.

۲۴.۴-۱۰ فرض کنید هر سطر ماتریس A در برنامه خطی $Ax\leq b$، متناظر با یک محدودیت تفاضلی به یکی از این اشکال باشد: محدودیت یک متغیره به شکل $x_i\leq b_k$ یا محدودیت یک متغیره به شکل $-x_i\leq b_k$. چگونگی تطبیق الگوریتم *Bellman-Ford* برای حل این شکل متفاوت سیستم محدودیت را بیان کنید.

۲۴.۴-۱۱ یک الگوریتم کارآمد برای حل سیستم محدودیت‌های تفاضلی $Ax\leq b$ که در آن عناصر b اعداد حقیقی و مجهول‌های x_i اعداد صحیح هستند ارائه دهید.

۲۴.۴-۱۲ * یک الگوریتم کارآمد برای حل سیستم محدودیت‌های تفاضلی $Ax\leq b$ که در آن عناصر b اعداد حقیقی و یک زیر مجموعه معیّن از مجهول‌های x_i و نه لزوماً همه x_iها، اعداد صحیح هستند ارائه دهید.

۲۴.۵ اثبات ویژگی‌های کوتاهترین مسیرها

در طول این فصل صحّت بحث‌های ما متکی بود بر نامساوی مثلث، ویژگی حد بالا، ویژگی عدم وجود مسیر، ویژگی همگرایی، ویژگی آرام‌سازی مسیر و ویژگی زیرگراف ماقبل. این ویژگی‌ها را در ابتدای فصل و بدون اثبات بیان کردیم. در این بخش آنها را اثبات می‌کنیم.

نامساوی مثلث

در بررسی جستجوی اوّل سطح (بخش ۲۲.۲) به عنوان لم ۲۲.۱، یک ویژگی ساده از کوتاهترین فاصله‌ها در گراف‌های بدون وزن را ثابت کردیم. نامساوی مثلث این ویژگی را به گراف‌های وزن‌دار

تعمیم می‌دهد.

لم ۲۴.۱۰ (نامساوی مثلث)

فرض کنید $G=(V,E)$ یک گراف وزن‌دار جهت‌دار با تابع وزن $w:E\to R$ و رأس مبدأ s باشد. برای همه یال‌های $(u,v)\in E$ داریم

$$\delta(s,v) \le \delta(s,u) + w(u,v) .$$

اثبات فرض کنید کوتاهترین مسیر p از مبدأ s به رأس v وجود دارد. بنابراین p دارای وزن کمتر نسبت به بقیه مسیرهای از s به v است. به ویژه، مسیر p وزن کمتری دارد نسبت به مسیری که شامل کوتاهترین مسیر از s به u و یال (u,v) است.

تمرین ۳- ۲۴.۵ از شما می‌خواهد حالتی را بررسی کنید که در آن هیچ کوتاهترین مسیری از s به v وجود ندارد. ■

اثرات آرام‌سازی روی برآوردهای کوتاهترین مسیر

گروه بعدی لم‌ها توضیح می‌دهند چگونه برآوردهای کوتاهترین مسیر از اجرای گامهای آرام‌سازی روی یال‌های گراف جهت‌دار و وزن‌داری که به وسیله روال *INITIALIZE-SINGLE-SOURCE* مقدار دهی اولیّه شده است تأثیر می‌پذیرند.

لم ۲۴.۱۱ (ویژگی حد بالا)

فرض کنید $G=(V,E)$ یک گراف وزن‌دار جهت‌دار با تابع وزن $w:E\to R$ باشد. $s\in V$ رأس مبدأ است و گراف به وسیله به *INITIALIZE-SINGLE-SOURCE(G,s)* مقدار دهی اولیّه شده است. آنگاه برای همه $v\in V$ داریم $d[v]\ge\delta(s,v)$ و این ثابت در طول اجرای یک سری گام‌های آرام‌سازی روی یال‌های G حفظ می‌شود. علاوه بر آن زمانی که $d[v]$ حد پایین خود می‌رسد دیگر هرگز تغییر نمی‌کند.

اثبات به وسیله استقرا روی تعداد گام‌های آرام‌سازی برای همه رأسهای $v\in V$، رابطه ثابت $d[v]\ge\delta(s,v)$ را اثبات می‌کنیم.

برای حالت پایه، $d[v]\ge\delta(s,v)$ پس از مقدار دهی اولیّه مطمئناً درست است، زیرا $d[s]=0\ge\delta(s,s)$ (دقّت کنید که $\delta(s,s)$ برابر ∞- است اگر s روی یک دور منفی قرار داشته باشد) و $d[v]=\infty$ دلالت می‌کند بر این که برای همه رأسهای $v\in V-\{s\}$ داریم $d[v]\ge\delta(s,v)$

برای گام استقرایی، آرام‌سازی یال (u,v) را در نظر بگیرید. از فرض استقرا برای همه رأس‌های $x\in V$ قبل از آرام‌سازی داریم $d[x]\ge\delta(s,x)$ تنها مقداری که ممکن است تغییر کند $d[v]$ است. اگر تغییر

کند داریم:

$$\begin{aligned} d[v] &= d[u] + w(u,v) \\ &\geq \delta(s,u) + w(u,v) \\ &\geq \delta(s,v) \end{aligned}$$

(بنا به فرض استقرا)

(بنا به نامساوی مثلث)

و بنابراین $d[v] \geq \delta(s,v)$ حفظ می‌شود.

برای این که بفهمید $d[v]$ زمانی که $d[v]=\delta(s,v)$ است دیگر هرگز تغییر نمی‌کند، توجّه کنید که $d[v]$ پس از رسیدن به حد پایین نمی‌تواند کاهش پیدا کند چون نشان داده شد که $d[v] \geq \delta(s,v)$ است و افزایش پیدا نمی‌کند چون گام‌های آرام‌سازی مقادیر $d[v]$ را افزایش نمی‌دهند.

قضیه فرعی ۲۴.۱۲ (ویژگی عدم وجود مسیر)

فرض کنید در گراف وزن‌دار جهت‌دار $G=(V,E)$ با تابع وزن $w{:}E \rightarrow R$، هیچ مسیری رأس مبدأ $s \in V$ را به رأس $v \in V$ متّصل نمی‌کند. آنگاه پس از این که گراف به وسیله $INITIALZE\text{-}SINGLE\text{-}SOURCE(G,s)$ مقدار دهی اولیّه می‌شود داریم $d[v]=\delta(s,v)=\infty$ و این تساوی به عنوان یک قاعده ثابت در طول اجرای گام‌های آرام‌سازی یال‌های گراف G حفظ می‌شود.

اثبات: بنا به حد بالا همواره داریم $\infty=\delta(s,v) \leq d[v]$ و بنابراین $d[v]=\infty=\delta(s,v)$ ■

لم ۲۴.۱۳

فرض کنید $G=(V,E)$ یک گراف وزن‌دار و جهت‌دار با تابع وزن $w{:}E \rightarrow R$ و $(u,v) \in E$ باشد. آنگاه، بلافاصله پس از آرام‌سازی یال (u,v) به وسیله اجرای $RELAX(u,v,w)$ داریم $d[v] \leq d[u]+w(u,v)$

اثبات اگر درست قبل از آرام سازی یال (u,v) داشته باشیم $d[v] > d(u)+w(u,v)$ آنگاه پس از آن داریم $d[v]=d[u]+w(u,v)$ امّا اگر در عوض، درست قبل از آرام‌سازی داشته باشیم $d[v] \leq d[u]+w(u,v)$، $d[u]$ و $d[v]$ هیچ کدام تغییر نمی‌کنند و لذا پس از آن، $d[v] \leq d[u]+w(u,v)$ ■

لم ۲۴.۱۴ (ویژگی همگرایی)

فرض کنید گراف $G=(V,E)$، یک گراف وزن‌دار و جهت‌دار با تابع وزن $w{:}E \rightarrow R$ باشد که در آن $s \in V$ رأس مبدأ و به ازای رأس‌های $u,v \in V$ $s \rightsquigarrow u \rightarrow v$ کوتاهترین مسیر در G است. فرض کنید G به وسیله $INITIALIZE\text{-}SINGLE\text{-}SOURCE(G, s)$ مقداردهی اولیه شده است و پس از آن، یک توالی از گام‌های آرام‌سازی که شامل فراخوانی $RELAX(u, v, w)$ هم می‌باشد روی گراف اجرا

می‌شود. اگر در زمانی قبل از فراخوانی $d[u]=\delta(s,u)$ باشد آنگاه در همه زمان‌های بعد از فراخوانی $d[v]=\delta(s,v)$ می‌باشد.

اثبات اگر قبل از آرام‌سازی یال (u,v) داشته باشیم $d[u]=\delta(s,u)$ بنا به ویژگی حد بالا این تساوی پس از آن هم حفظ می‌شود. به ویژه، پس از آرام‌سازی یال (u,v) داریم:

$$
\begin{aligned}
d[v] &\le d[u] + w(u,v) && \text{(بنا به لم ۲۴.۱۳)}\\
&= \delta(s,u) + w(u,v)\\
&= \delta(s,v) && \text{(بنا به لم ۲۴.۱)}
\end{aligned}
$$

بنا به ویژگی حد بالا داریم $d[v]\geq\delta(s,v)$ که از آن نتیجه می‌گیریم که $d[v]=\delta(s,v)$ و این تساوی پس از آن هم حفظ می‌شود. ■

لم ۲۴.۱۵ (ویژگی آرام‌سازی مسیر)

فرض کنید $G=(V,E)$ یک گراف وزن‌دار جهت‌دار با تابع وزن $w:E\rightarrow R$ و رأس مبدأ s باشد. یک کوتاهترین مسیر مانند $p=\langle v_0,v_1,...,v_k\rangle$ از $s=v_0$ به v_k در نظر بگیرید. اگر G به وسیله INITIALIZE-SINGLE-SOURCE(G,s) مقدار دهی اولیّه شود و سپس یک توالی از گام‌های آرام‌سازی که شامل به ترتیب آرام‌سازی یال‌های $(v_0,v_1),(v_1,v_2),...,(v_{k-1},v_k)$ است انجام گیرد، پس از اجرای این آرام‌سازی‌ها و در تمام زمان‌های پس از آن، $d[v_k]=\delta(s,v_k)$ است. این ویژگی اهمیّت نمی‌دهد که چه آرام‌سازی‌های دیگری، شامل آنهایی که با آرام سازی‌های یال‌های p آمیخته شده‌اند صورت می‌گیرند.

اثبات به وسیله استقرا نشان می‌دهیم پس از اینکه iامین یال مسیر p آرام شد داریم $d[v_i]=\delta(s,v_i)$ برای پایه استقرا، $i=0$است و قبل از این که یکی از یال‌های مسیر p آرام شود از مقدار دهی اولیّه داریم $d[v_0]=d[s]=0=\delta(s,s)$ بنا به ویژگی حدّ بالا مقدار $d[s]$ هرگز پس از مقدار دهی اولیّه تغییر نمی‌کند.

برای گام استقرایی فرض می‌کنیم $d[v_{i-1}]=\delta(s,v_{i-1})$ است و آرام‌سازی یال (v_{i-1},v_i) را بررسی می‌کنیم. بنا به ویژگی همگرایی، پس از این آرام‌سازی داریم $d[v_i]=\delta(s,v_i)$ و این تساوی در تمام زمان‌های پس از آن حفظ می‌گردد. ■

آرام‌سازی و درخت‌های کوتاهترین مسیرها

اینک نشان می‌دهیم زمانی که یک توالی از آرام‌سازی‌ها سبب می‌شوند برآوردهای کوتاهترین مسیر به وزنهای کوتاهترین مسیر همگرا شوند، زیرگراف ماقبل G_π که از مقادیر حاصل شده π ایجاد شده، یک درخت کوتاهترین مسیرها برای G است. بحث را آغاز می‌کنیم با لم بعد که نشان می‌دهد زیرگراف

ماقبل، همیشه یک درخت مشتق شده از ریشه مبدأ را تشکیل می‌دهد.

لم ۲۴.۱۶

فرض کنید $G=(V, E)$ یک گراف وزن‌دار جهت‌دار با تابع وزن $w:E\rightarrow R$ و رأس $s\in V$ به عنوان مبدأ باشد و فرض کنید G شامل هیچ دوری با وزن منفی و قابل دستیابی از s نباشد. آن گاه پس از این که گراف به وسیله $INITIALIZE\text{-}SINGLE\text{-}SOURCE(G,s)$ مقدار دهی اولیّه شد، زیرگراف ماقبل G_π یک درخت مشتق شده از ریشه s را تشکیل می‌دهد و یک توالی از گام‌های آرام‌سازی روی یال‌های G این ویژگی را به عنوان یک قاعده ثابت حفظ می‌کند.

اثبات در ابتدا تنها رأس در G_π، رأس مبدأ است و لم به طور بدیهی درست می‌باشد. یک زیرگراف ماقبل G_π را در نظر بگیرید که پس از یک توالی از گام‌های آرام‌سازی ایجاد شده است. درابتدا ثابت خواهیم نمود که G_π بدون دور است. برای ایجاد تناقض فرض کنید یک گام آرام‌سازی دوری در G_π ایجاد می‌کند. دور را $c=<v_0,v_1,...,v_k>$ در نظر بگیرید که در آن $v_k=v_0$ است. بنابراین برای $i=1,2,...,k$ داریم $\pi[v_i]=v_{i-1}$ و بدون از دست دادن کلیّت می‌توانیم فرض کنیم دور در داخل G_π توسط آرام‌سازی یال (v_{k-1},v_k) ایجاد شده است.

ادّعا می‌کنیم که همه رأس‌های روی دور c از رأس مبدأ s قابل دستیابی هستند. چرا؟ هر رأس روی c یک ماقبل غیر NIL دارد و بنابراین با نسبت داده شدن یک مقدار π غیر NIL به هر رأس روی c، یک برآورد با مقدار محدود به عنوان کوتاهترین مسیر به آن نسبت داده شده است. بنا به ویژگی حد بالا، هر رأس روی دور c یک وزن محدود کوتاهترین مسیر دارد که دلالت می‌کند بر این که رأس از s قابل دستیابی است.

برآوردهای کوتاهترین مسیر روی c را دقیقاً قبل از فراخوانی $RELAX(v_{k-1},v_k,w)$ بررسی می‌کنیم و نشان می‌دهیم که c یک دور با وزن منفی است و بدان وسیله با این فرض که G شامل هیچ دوری با وزن منفی قابل دستیابی از s نمی‌باشد تناقض ایجاد می‌شود. درست قبل از فراخوانی، برای $i=1,2,...,k-1$ داریم $\pi[v_i]=v_{i-1}$. بنابراین برای $i=1,2,..,k-1$، آخرین به روز رسانی $d[v_i]$ به وسیله انتساب فرمول انجام گرفته است. اگر $d[v_{i-1}]$ از آن موقع تغییر کرده بود مقدارش کاهش یافته بود. لذا درست قبل از فراخوانی $RELAX(v_{k-1},v_k,w)$ داریم

$$d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i) \qquad \text{(۲۴.۱۲)}$$

برای همه $i = 1, 2, ... , k-1$،

چون $\pi[v_k]$ با این فراخوانی تغییر می‌کند بلافاصله قبل از آن، نامساوی اکید زیر را داریم

$$d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k) .$$

با جمع این نامساوی اکید با $k-1$ نامساوی (۲۴.۱۲)، به مجموع برآوردهای کوتاهترین مسیر حول دور c می‌رسیم:

$$\sum_{i=1}^{k} d[v_i] > \sum_{i=1}^{k}(d[v_{i-1}] + w(v_{i-1}, v_i))$$
$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

امّا

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}] ,$$

چون در هر حاصل جمع، هر رأس در دور c دقیقاً یک بار ظاهر می‌شود. این تساوی دلالت می‌کند بر این که

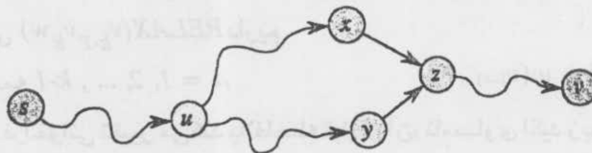
$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

بنابراین مجموع وزن‌ها حول دور c منفی است، که تناقض مورد نظر را تولید می‌کند.

اکنون ثابت کرده‌ایم که G_π یک گراف جهت‌دار بدون دور است. برای این که نشان دهیم G_π یک درخت مشتق شده از ریشه s را تشکیل می‌دهد کافی است ثابت کنیم برای هر رأس $v \in V_\pi$، یک مسیر منحصر به فرد از s به v در G_π وجود دارد.

ابتدا باید نشان دهیم مسیری از s برای هر رأس V_π وجود دارد. رأس‌های V_π، رأس‌های با مقادیر π غیر NIL و رأس s هستند. در اینجا ایده این است که به وسیله استقرا ثابت کنیم که مسیری از s به همه رأس‌های V_π وجود دارد. جزئیات به عنوان تمرین ۶-۲۴.۵ واگذار شده است.

اینک برای کامل کردن اثبات لم، باید نشان دهیم که برای هر رأس $v \in V_\pi$ حداکثر یک مسیر از s به v در گراف G_π وجود دارد. فرض کنید چنین نباشد. به عبارت دیگر فرض کنید دو مسیر ساده از s به یک رأس v وجود داشته باشد: p_1 که می‌تواند به $s \leadsto u \leadsto x \to z \leadsto v$ و p_2 که می‌تواند به $s \leadsto u \leadsto y \to z \leadsto v$ تجزیه شود که در آنها $x \neq y$ (شکل ۹. ۲۴ را ملاحظه کنید) اما $\pi[z] = x$ و $\pi[z] = y$ که تناقض $x = y$ را نشان می‌دهد. نتیجه می‌گیریم که یک مسیر ساده منحصر به فرد در G_π از s به v وجود دارد و بنابراین G_π یک درخت مشتق شده از s را تشکیل می‌دهد. ■



شکل ۹. ۲۴ نمایش این که مسیر واقع در G_π از مبدأ s به رأس v منحصر بوده است. اگر دو مسیر p_1 $(s \leadsto u \leadsto x \to z \leadsto v)$ و p_2 $(s \leadsto u \leadsto y \to z \leadsto v)$ که $x \neq y$ وجود داشته باشد، آنگاه $\pi[z] = y$ و $\pi[z] = y$ است که یک تناقض می‌باشد.

لم ۲۴.۱۷ (ویژگی زیرگراف ماقبل)

$G=(V,E)$ را به عنوان یک گراف وزن‌دار جهت‌دار با تابع وزن $w:E\rightarrow R$ و رأس مبدأ s در نظر بگیرید و فرض کنید G شامل دوری با وزن منفی قابل دستیابی از s نباشد. INITIALIZE-SINGLE-SOURCE(G,s) را فراخوانی و یک توالی از گام‌های آرام‌سازی روی یال‌های G که برای همه $v\in V$، $d[v]=\delta(s,v)$ را تولید می‌کند را اجرا می‌کنیم. آنگاه زیرگراف ماقبل G_π درخت کوتاهترین مسیرها مشتق شده از s است.

اثبات باید ثابت کنیم سه ویژگی درخت‌های کوتاهترین مسیرها که قبلاً بیان شدند برای G_π برقرارند. برای نشان دادن اوّلین ویژگی باید نشان دهیم که V_π مجموعه رأس‌های قابل دستیابی از s است. بنا به تعریف، وزن کوتاهترین مسیر $\delta(s,v)$ یک مقدار متناهی است اگر و فقط اگر v از s قابل دستیابی باشد و بنابراین رأس‌هایی که قابل دستیابی از s هستند دقیقاً آنهایی هستند که دارای مقادیر متناهی d می‌باشند. امّا به هر رأس $v\in V-\{s\}$ یک مقدار متناهی $d[v]$ نسبت داده شده اگر و فقط اگر $\pi[v]\neq NIL$ باشد. بنابراین رأس‌های V_π دقیقاً آنهایی هستند که از s قابل دستیابی‌اند.

ویژگی دوّم مستقیماً از لم ۲۴.۱۶ بدست می‌آید.

بنابراین تنها اثبات آخرین ویژگی درخت‌های کوتاهترین مسیرها باقی می‌ماند: برای هر رأس $v\in V_\pi$، مسیر ساده منحصر بفرد $s\overset{p}{\leadsto}v$ در G_π یک کوتاهترین مسیر از s به v در G است. قرار دهید $p=<v_0,v_1,...,v_k>$، که در آن $v_0=s$ و $v_k=v$ در G است. برای هر $i=1,2,...,k$ داریم $d[v_i]=\delta(s,v_i)$ و $d[v_i]\geq d[v_{i-1}]+w(v_{i-1},v_i)$ که از آنها نتیجه می‌گیریم $w(v_{i-1},v_i)\leq\delta(s,v_i)-\delta(s,v_{i-1})$. با جمع وزن‌ها در مسیر p داریم:

$$\begin{aligned}
w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\
&\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\
&= \delta(s, v_k) - \delta(s, v_0) && \text{(به دلیل ویژگی تلسکوپی)} \\
&= \delta(s, v_k) && (\delta(s,v_0)=\delta(s,s)=0 \text{ زیرا})
\end{aligned}$$

بنابراین $w(p)\leq\delta(s,v_k)$. چون $\delta(s,v_k)$ یک حد پایین و وزن مسیر s به v_k است نتیجه می‌گیریم که $w(p)=\delta(s,v_k)$، و بنابراین p کوتاهترین مسیر از s به $v=v_k$ است. ■

تمرین‌ها

۲۴.۵-۱ دو درخت کوتاهترین مسیرها برای گراف جهت‌دار شکل ۲۴.۲، غیر از درخت نشان داده شده ارائه دهید.

۲۴.۵-۲ یک نمونه گراف وزن‌دار جهت‌دار $G=(V,E)$ با تابع وزن $w:E\rightarrow R$ و رأس مبدأ s ارائه دهید به طوری که G این خصوصیات را داشته باشد: برای هر یال $(u,v)\in E$ یک درخت کوتاهترین مسیرها با ریشه s که شامل (u,v) است و یک درخت کوتاهترین مسیرهای دیگر با ریشه s که شامل (u,v) نمی‌باشد وجود ندارد.

۲۴.۵-۳ اثبات لم ۲۴.۱ را چنان تغییر دهید که حالت هایی که در آنها وزن‌های کوتاهترین مسیر ∞ یا $-\infty$ هستند را هم بررسی کند.

۲۴.۵-۴ فرض کنید $G=(V,E)$ یک گراف وزن‌دار جهت‌دار با رأس مبدأ s باشد و به وسیله *INITIALIZE-SINGLE-SOURCE(G,s)* مقدار دهی اولیّه شده است. ثابت کنید اگر یک توالی از گام‌های آرام‌سازی، یک مقدار غیر *NIL* را به $\pi[s]$ نسبت دهند G شامل یک دور با وزن منفی می‌شود.

۲۴.۵-۵ فرض کنید $G=(V,E)$ یک گراف وزن‌دار و جهت‌دار و بدون یال‌های با وزن منفی باشد. $s\in V$ رأس مبدأ است و فرض کنید $\pi[v]$ مقدار ماقبل v روی کوتاهترین مسیر از مبدأ s به v است اگر $v\in V-\{s\}$ قابل دستیابی از s می‌باشد، و در غیر این صورت *NIL* است. یک نمونه از چنین گرافی و یک انتساب مقادیر π که یک دور در G_π ایجاد می‌کند را ارائه دهید. (بنا به لم ۲۴.۱۶ یک چنین انتسابی نمی‌تواند به وسیله یک توالی از گام‌های آرام‌سازی تولید شود.)

۲۴.۵-۶ فرض کنید $G=(V,E)$ یک گراف وزن‌دار جهت‌دار با تابع وزن $w:E\rightarrow R$ و بدون دوری با وزن منفی باشد. s رأس مبدأ است و G به وسیله *INITIALZE-SINGLE-SOURCE(G,s)* مقدار دهی اولیّه شده است. ثابت کنید برای هر رأس $v\in V_\pi$، مسیری از s به v در G_π وجود دارد، که این ویژگی به عنوان یک قاعده ثابت حین انجام هر توالی از آرام‌سازی‌ها حفظ می‌شود.

۲۴.۵-۷ فرض کنید $G=(V,E)$ یک گراف وزن‌دار جهت‌دار و بدون دور منفی باشد. s رأس مبدأ است و G به وسیله *INITIALIZE-SINGLE-SOURCE(G,s)* مقدار دهی اولیّه شده است. ثابت کنید یک توالی از $|V|-1$ گام آرام‌سازی وجود دارد که برای همه $v\in V$، $d[v]=\delta(s,v)$ را تولید می‌کند.

۲۴.۵-۸ فرض کنید G یک گراف وزن‌دار جهت‌دار دلخواه و شامل یک دور با وزن منفی قابل دستیابی از رأس مبدأ s باشد. نشان دهید همیشه یک توالی نامتناهی از آرام‌سازی‌های یال‌های G همواره می‌تواند انجام شود، به طوری که هر آرام‌سازی باعث تغییر یک برآورد کوتاهترین مسیر می‌شود.

مسائل

۲۴-۱ توسعه الگوریتم *Bellman-Ford* توسط *Yen*

فرض کنید آرام‌سازی یال‌ها در هر گذر الگوریتم *Bellman-Ford* را به شکل زیر مرتب کنیم. قبل از

اوّلین گذر، یک ترتیب خطی دلخواه $v_1, v_2, \ldots, v_{|V|}$ را به رأس‌های گراف ورودی $G=(V,E)$ نسبت می‌دهیم. آنگاه مجموعه E را به $E_f \cup E_b$ افراز می‌کنیم که $E_f=\{(v_i,v_j)\in E: i<j\}$ و $E_b=\{(v_i,v_j)\in E: i>j\}$ (فرض کنید G شامل خود – حلقه نیست و بنابراین هر یال یا در E_f است یا در E_b) $G_f=(V,E_f)$ و $G_b=(V,E_b)$ را تعریف کنید.

a. ثابت کنید G_f با مرتب‌سازی موضعی $<v_1, v_2, \ldots, v_{|V|}>$ و G_b با مرتب سازی موضعی $<v_{|V|}, v_{|V|-1}, \ldots, v_1>$ بدون دور است.

فرض کنید هر گذر از الگوریتم *Bellman-Ford* را به شکل زیر پیاده‌سازی می‌کنیم. هر رأس را به ترتیب $v_1, v_2, \ldots, v_{|V|}$ ملاقات و یال‌های E_f خروجی از آن را آرام می‌کنیم. سپس هر یال را به ترتیب $v_{|V|}, v_{|V|-1}, \ldots, v_1$ ملاقات و یال‌های E_b خروجی از آن را آرام می‌کنیم.

b ثابت کنید که با این طرح، اگر G شامل دوری با وزن منفی قابل دستیابی از رأس مبدأ s نباشد، پس از $\lceil |V|/2 \rceil$ گذر روی یال‌ها، برای همه رأس‌های $v\in V$ خواهیم داشت $d[v]=\delta(s,v)$

c آیا این طرح، زمان اجرای مجانبی الگوریتم *Bellman-Ford* را بهتر می‌کند؟

۲۴-۲ جعبه‌های تودرتو

یک جعبه d- بعدی با ابعاد $(x_1, x_2, \ldots, x_d)$ داخل یک جعبه دیگر با ابعاد $(y_1, y_2, \ldots, y_d)$ قرار می‌گیرد اگر یک جایگشت π روی $\{1,2,\ldots,d\}$ وجود داشته باشد به طوری که $x_{\pi(1)}<y_1, x_{\pi(2)}<y_2, \ldots, x_{\pi}<y_d$.

a ثابت کنید که رابطه تودرتو یک رابطه تعدی است.

b یک روش کارآمد ارائه دهید که تعیین کند آیا یک جعبه d- بعدی داخل جعبه دیگر قرار می‌گیرد یا خیر.

c فرض کنید یک مجموعه شامل n جعبه d- بعدی $\{B_1, B_2, \ldots B_n\}$ داده شده است. یک الگوریتم کارآمد ارائه دهید که بلندترین توالی $<B_{i_1}, B_{i_2}, \ldots, B_{i_k}>$ از جعبه‌ها که در آن برای $j=1,2,\ldots,k-1$ جعبه B_{i_j} داخل جعبه $B_{i_{j+1}}$ قرار می‌گیرد را تعیین کند. زمان اجرای الگوریتم خود را بر حسب d و n بیان کنید.

۲۴-۳ *Arbitrage*

Arbitrage عبارت است از استفاده از اختلافات در نرخ‌های مبادله پول، برای تبدیل یک واحد پول به بیش از یک واحد از همان پول. برای مثال فرض کنید یک دلار آمریکا *46/4* روپیه هند، *1* روپیه هند *2/5* ین ژاپن و *1* ین ژاپن *0/0091* دلار آمریکا است. بنابراین یک تاجر بوسیله تبدیل واحدهای پولی می‌تواند یک دلار را با $46/6 * 2/5 * 0/0091=1/0556$ دلار عوض کند و بنابراین *5/56* درصد سود کسب کند.

فرض کنید n واحد پولی $c_1, c_2, \ldots, c_n$ و یک جدول $n\times n$ از نرخ‌های مبادله به نام R داده شده، بدین

شکل که یک واحد از پول c_i با $R[i,j]$ واحد از پول c_j مبادله می‌شود.

a یک الگوریتم کارآمد ارائه دهید که تعیین کند آیا یک توالی از واحدهای پولی $<c_{i_1}, c_{i_2}, \ldots, c_{i_k}>$ به شرط آنکه فرمول وجود دارد یا نه. زمان اجرای الگوریتم خود را تحلیل کنید.

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1 .$$

b الگوریتم کارآمدی ارائه دهید که اگر چنین توالی وجودداشت آن را نمایش دهد. زمان اجرای الگوریتم خود را تحلیل کنید.

۴-۲۴ الگوریتم مقیاس دهی *Gabow* برای کوتاهترین مسیرها از یک مبدأ واحد

الگوریتم *مقیاس دهی* ابتدا با در نظر گرفتن فقط بالاترین بیت یک مقدار ورودی مناسب (مانند وزن یال‌ها)، مسأله را حل می‌کند. سپس جواب اولیّه را با نگاه کردن به دو بیت اوّل پالایش می‌کند. به همین ترتیب بیت‌های بیشتر و بیشتری را در نظر می‌گیرد و در هر بار جواب را پالایش می‌کند تا این که همه بیت‌ها در نظر گرفته شوند و جواب صحیح محاسبه شود.

در این مسئله، الگوریتم محاسبه کوتاهترین مسیرها از یک مبدأ واحد را به وسیله مقیاس دهی وزن یال‌ها بررسی می‌کنیم. گراف جهت‌دار $G=(V,E)$ با وزن‌های صحیح غیر منفی w داده شده است. فرض کنید $W=\max_{(u,v)\in E}\{w(u,v)\}$. هدف، ایجاد الگوریتمی است که در زمان $O(E\lg W)$ اجرا می‌شود. فرض می‌کنیم که همه رأس‌ها از مبدأ قابل دستیابی‌اند. الگوریتم در نمایش دودویی وزن‌های یال‌ها، هر بار یک بیت را به ترتیب با ارزش‌ترین بیت به کم ارزش‌ترین بیت، وارد حل مسئله می‌کند. به ویژه فرض کنید $k=\lceil \lg(W+1)\rceil$، تعداد بیت‌ها در نمایش دودویی W باشد و برای $i=1, 2, \ldots, k$، $w_i(u,v)=\lfloor w(u,v)/2^{k-i}\rfloor$. به عبارت دیگر $w_i(u,v)$ شکل "کوچک شده" $w(u,v)$ است که به وسیله با ارزش‌ترین i بیت $w(u,v)$ تولید می‌شود. (بنابراین برای همه $(u,v)\in E$ داریم $w_k(u,v)=w(u,v)$). برای مثال اگر $k=5$ و $w(u,v)=25$ با نمایش دودویی $<11001>$ باشد آنگاه $w_3(u,v)=<110>=6$. بعنوان مثالی دیگر اگر $k=5$ و $w(u,v)=<00100>=4$ باشد آنگاه $w_3(u,v)=<001>=1$ است.

$\delta_i(u,v)$ را به عنوان وزن کوتاهترین مسیر از رأس u به رأس v با استفاده از تابع وزن w_i تعریف می‌کنیم. بنابراین برای همه $u,v\in V$ داریم $\delta_k(u,v)=\delta(u,v)$. برای مبدأ s الگوریتم مقیاس دهی ابتدا وزنهای کوتاهترین مسیر $\delta_1(s,v)$ را برای همه $v\in V$ محاسبه می‌کند، سپس برای همه $v\in V$، $\delta_2(s,v)$ را محاسبه می‌کند و به همین ترتیب ادامه می‌دهد تا این که برای همه $v\in V$، $\delta_k(s,v)$ محاسبه شود. فرض می‌کنیم $|E|\geq|V|-1$ است و خواهیم دید که محاسبه δ_i به اندازه $O(E)$، و لذا کل الگوریتم به اندازه $O(kE)=O(E\lg W)$ زمان می‌برد.

a فرض کنید برای همه رأس‌های $v\in V$ داریم $\delta(s,u)\leq|E|$ نشان دهید برای همه $v\in V$ می‌توانیم $\delta(s,v)$ را در زمان $O(E)$ محاسبه کنیم.

b نشان دهید برای همه $v\in V$ می‌توانیم $\delta_1(s,v)$ را در زمان $O(E)$ محاسبه کنیم.

اکنون به محاسبه δ_i از δ_{i-1} می‌پردازیم.

c ثابت کنید برای هر $i=2,3,...,k$ داریم $w_i(u,v) = 2w_{i-1}(u,v)+1$ یا $w_i(u,v) = 2w_{i-1}(u,v)$.

سپس ثابت کنید که برای همه $v\in V$ داریم

$$2\delta_{i-1}(s,v) \le \delta_i(s,v) \le 2\delta_{i-1}(s,v) + |V| - 1$$

d برای $i=2,3,...,k$ و همه $(u,v)\in E$ تعریف زیر را داریم

$$\widehat{w}_i(u,v) = w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v) .$$

ثابت کنید برای $i=2,3,...,k$ و همه $u,v\in V$، $\widehat{w}_i(u,v)$ یال (u,v)، یعنی مقدار "وزن جدید" آن یال، یک عدد صحیح غیر منفی است.

e اینک $\widehat{\delta}_i(s,v)$ را به عنوان وزن کوتاهترین مسیر از s به v با استفاده از تابع وزن $\widehat{w}_i$ تعریف می‌کنیم. ثابت کنید برای $i=2,3,...,k$ و همه $v\in V$ داریم

$$\delta_i(s,v) = \widehat{\delta}_i(s,v) + 2\delta_{i-1}(s,v)$$

و

$$\widehat{\delta}_i(s,v) \le |E|.$$

f نشان دهید چگونه برای همه $v\in V$ می‌توان $\delta_i(s,v)$ را از $\delta_{i-1}(s,v)$ در زمان $O(E)$ به دست آورد و نتیجه بگیرید که برای همه $v\in V$، $\delta(s,v)$ می‌تواند در زمان $O(E\lg W)$ محاسبه گردد.

۲۴-۵ الگوریتم Karp در مورد دور مینیمم از نظر میانگین وزن

فرض کنید $G=(V,E)$ یک گراف جهت‌دار با تابع وزن $w:E\to R$ و $n=|V|$ باشد. وزن میانگین دور $c=\langle e_1,e_2,...,e_k\rangle$ را چنین تعریف می‌کنیم:

$$\mu(c) = \frac{1}{k}\sum_{i=1}^{k} w(e_i) .$$

قرار دهید $\mu^* = \min_c \mu(c)$، که c روی همه دورهای جهتدار G تغییر می‌کند. دور c که برای آن $\mu(c) = \mu^*$، دور مینیمم از نظر میانگین وزن خوانده می‌شود. این مسئله به یک الگوریتم کارآمد برای محاسبه μ^* می‌پردازد.

بدون از دست دادن کلیّت فرض کنید هر رأس $v\in V$ قابل دستیابی از رأس مبدأ $s\in V$ است. $\delta(s,v)$ را به عنوان وزن کوتاهترین مسیر از s به v و $\delta_k(s,v)$ را به عنوان وزن کوتاهترین مسیر از s به v شامل دقیقاً k یال قرار دهید. اگر هیچ مسیری از s به v با دقیقاً k یال وجود نداشته باشد آنگاه $\delta_k(s,v)=\infty$

a نشان دهید که اگر $\mu^*=0$ باشد آنگاه G هیچ دوری با وزن منفی ندارد و برای همه رأس‌های $v\in V$

داریم

$$\delta(s, v) = \min_{0\le k\le n-1} \delta_k(s, v)$$

b. نشان دهید اگر $\mu^*=0$ باشد آنگاه برای همه رأس‌های $v\in V$ داریم

$$\max_{0\le k\le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n-k} \ge 0$$

(راهنمایی: از هر دو ویژگی قسمت (a) استفاده کنید.)

c. فرض کنید c یک دور با وزن 0 و u و v یک جفت رأس روی c باشند. فرض کنید $\mu^*=0$ و وزن مسیر u به v در دور c برابر x است. ثابت کنید $\delta(s,v)=\delta(s,u)+x$ (راهنمایی: وزن مسیر از v به u در این دور برابر $-x$ است.)

d. نشان دهید اگر $\mu^*=0$ باشد آنگاه روی هر دور مینیمم از نظر میانگین وزن، یک رأس v وجود دارد به طوری که

$$\max_{0\le k\le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n-k} = 0 .$$

(راهنمایی: نشان دهید یک کوتاهترین مسیر به هر رأس در دور مینیمم از نظر میانگین وزن، می‌تواند در راستای دور ادامه داده شود تا کوتاهترین مسیر به رأس بعدی روی دور را تولید کند.)

e. نشان دهیداگر $\mu^*=0$ آنگاه

$$\min_{v\in V} \max_{0\le k\le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n-k} = 0 .$$

f. نشان دهید اگر یک مقدار ثابت t به وزن هر یال G اضافه کنیم آنگاه μ^* به اندازه t افزایش می‌یابد. از این مورد استفاده کرده و نشان دهید

$$\mu^* = \min_{v\in V} \max_{0\le k\le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n-k} .$$

g. الگوریتمی با مرتبه زمانی $O(VE)$ برای محاسبه μ^* ارائه دهید.

۶ - ۲۴ کوتاهترین مسیرهای دو آهنگه[1]

یک توالی دو آهنگه است اگر به طور یکنواخت افزایش یابد و سپس به طور یکنواخت کاهش یابد، یا اینکه به طور چرخشی بتواند افزایش یکنواخت و سپس کاهش یکنواخت، تغییر داده شود. برای مثال توالی‌های $<1,4,6,8,3,-2>$، $<9,2,-4,-10,-5>$ و $<1,2,3,4,>$ دو آهنگه هستند ولی توالی

1.bitonic shortest paths

<$1,3,12,4,2,10$> دوآهنگه نیست. (به مسئله ۱- ۱۵ در مورد مسئله فروشنده دوره گرد اقلیدسی دوآهنگه مراجعه کنید.)

فرض کنید گراف جهت‌دار $G=(V,E)$ با تابع وزن $w:E\rightarrow R$ به ما داده شده و می‌خواهیم کوتاهترین مسیرها از مبدأ واحد s را به دست آوریم. یک سری اطلاعات اضافی هم داده شده است: برای هر رأس $v\in V$، وزنهای یالها در طول کوتاهترین مسیر از s به v یک توالی دوآهنگه را تشکیل می‌دهند.

کارآمدترین الگوریتم خود برای حل این مسئله را ارائه نمائید و زمان اجرای آن را تحلیل کنید.

فصل ۲۵ کوتاهترین مسیرها بین همه جفت‌ها[1]

در این فصل به مسئله یافتن کوتاهترین مسیرها بین همه جفت رأس‌های گراف می‌پردازیم. این مسئله ممکن است در ایجاد یک جدول فاصله‌های بین همه جفت شهرها در نقشه جاده، پیش بیاید. مانند فصل ۲۴ یک گراف وزن‌دار جهت‌دار $G=(V,E)$ با تابع وزن $w:E\rightarrow R$ که یال‌ها را به وزن‌هایی با مقادیر حقیقی نگاشت می‌کند داده شده است. می‌خواهیم برای همه جفت رأس‌های $u,v\in V$ کوتاهترین مسیر (کمترین وزن) از u به v را پیدا کنیم که وزن یک مسیر، مجموع وزن‌های یال‌های تشکیل دهنده آن است. معمولاً خروجی را به شکل جدولی می‌خواهیم: ورودی سطر u و ستون v باید وزن کوتاهترین مسیر از u به v باشد.

می‌توانیم مسئله کوتاهترین مسیرهای بین همه جفت‌ها را به وسیله اجرای $|V|$ بار الگوریتم کوتاهترین مسیرها از یک مبدأ واحد، یک بار برای هر رأس به عنوان مبدأ حل کنیم. اگر وزن همه یال‌ها غیر منفی هستند می‌توانیم از الگوریتم *Dijkstra* استفاده کنیم. اگر از پیاده سازی آرایه‌ای خطی صف اولویّت مینیمم استفاده کنیم، زمان اجرا $O(V^3+VE)=O(V^3)$ است. پیاده‌سازی *Min-Heap* دودویی از صف اولویّت مینیمم در زمان $O(VE\lg V)$ اجرا می‌شود که بهبود الگوریتم را نتیجه می‌دهد. به عنوان جایگزینی می‌توانیم صف اولویّت مینیمم را با *heap* فیبوناچی پیاده‌سازی کنیم که زمان اجرای $O(V^2\lg V+VE)$ را حاصل می‌کند.

اگر یال‌های با وزن منفی مجاز باشند الگوریتم *Dijkstra* دیگر نمی‌تواند استفاده شود. به جای آن باید الگوریتم کندتر *Bellman-Ford* را یک بار برای هر رأس اجرا کنیم. زمان اجرای به دست آمده $O(V^2E)$ است که در یک گراف متراکم برابر $O(V^4)$ می‌باشد. در این فصل خواهیم دید که چگونه بهتر عمل کنیم. همچنین ارتباط مسئله کوتاهترین مسیرها بین همه جفت‌ها با ضرب ماتریس و ساختار جبری آن را بررسی می‌کنیم. برخلاف الگوریتم‌های مبدأ واحد که در آنها از نمایش لیست مجاورتی استفاده می‌شد، اکثر الگوریتم‌های این فصل از نمایش ماتریس مجاورتی استفاده می‌کنند. (الگوریتم *Johnson* برای گراف‌های پراکنده از لیست‌های مجاورتی استفاده می‌کند.) برای سادگی فرض می‌کنیم

---

1 *All-pairs shortest paths*

رأس‌ها به شکل $1,2,\ldots,|V|$ شماره گذاری شده‌اند و بنابراین ورودی، یک ماتریس $n\times n$ با مقادیر W است که وزن یال‌های گراف جهت‌دار n رأسی $G=(V,E)$ را نشان می‌دهد. به عبارت دیگر داریم $W=(w_{ij})$، که در آن

$$w_{ij} = \begin{cases} 0 & \text{اگر } i=j \\ (i,j) \text{ وزن یال جهت‌دار} & \text{اگر } i\neq j \text{ و } (i,j)\in E \\ \infty & \text{اگر } i\neq j \text{ و } (i,j)\notin E \end{cases} \qquad (۲۵.۱)$$

یالهای با وزن منفی مجاز هستند، اما فعلاً فرض می‌کنیم گراف ورودی، شامل دور با وزن منفی نیست.

خروجی جدولی الگوریتمهای کوتاهترین مسیرهای بین همهٔ جفتها که در این فصل ارائه شده، ماتریس $n\times n$ $D=(d_{ij})$ است که ورودی d_{ij} شامل وزن کوتاهترین مسیر از رأس i به رأس j می‌باشد. به عبارت دیگر اگر $\delta(i,j)$ به وزن کوتاهترین مسیر از رأس i به j (مانند فصل ۲۴) دلالت کند، در پایان کار الگوریتم، $d_{ij}=\delta(i,j)$ است.

برای حل مسألهٔ کوتاهترین مسیرهای بین همهٔ جفتها روی یک ماتریس مجاورتی ورودی، علاوه بر محاسبه وزن کوتاهترین مسیرها، به *ماتریس ماقبل* $\Pi=(\pi_{ij})$ نیز احتیاج داریم، که در آن π_{ij} برابر NIL است اگر $i=j$ باشد یا این که مسیری از i به j وجود نداشته باشد، و در غیر اینصورت π_{ij}، رأس ماقبل j در کوتاهترین مسیر از i به j است. همانطور که زیرگراف ماقبل G_π فصل ۲۴، درخت کوتاهترین مسیرها برای رأس مبدأ داده شده است، زیرگراف حاصل شده از i امین ردیف ماتریس Π باید یک درخت کوتاهترین مسیرها با ریشه i باشد. برای هر رأس $i\in V$ *زیرگراف ماقبل* G را به صورت $G_{\pi,i}=(V_{\pi,i}, E_{\pi,i})$ تعریف می‌کنیم، که در آن

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$$

و

$$E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} - \{i\}\} .$$

اگر $G_{\pi,i}$ درخت کوتاهترین مسیرها باشد، روال زیر که شکل تغییریافته روال *PRINT-PATH* فصل ۲۲ است، کوتاهترین مسیر از رأس i به رأس j را چاپ می‌کند.

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
1  if i = j
2     then print i
3     else if π_ij = NIL
4             then print "no path from" i "to" j "exists"
5             else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
6                  print j
```

برای برجسته کردن ویژگیهای ضروری الگوریتمهای مربوط به همهٔ جفتها در این فصل، ساخت ماتریسهای ماقبل و ویژگیهای آنها را به گستردگی که زیرگرافهای ماقبل را در فصل ۲۴ بررسی کردیم، پوشش نمی‌دهیم. قواعد و اصول به وسیلهٔ تمرینها پوشش داده می‌شوند.

رئوس مطالب فصل

بخش ۲۵.۱ یک الگوریتم برنامه‌سازی پویا، که بر مبنای ضرب ماتریس طراحی شده است را برای حل مسئله کوتاهترین مسیرهای بین همه جفت‌ها ارائه می‌کند. با استفاده از تکنیک "مجذور کردن مکرر"، این الگوریتم می‌تواند در زمان $\Theta(V^3 lgV)$ اجرا شود. یک الگوریتم دیگر برنامه‌سازی پویا، الگوریتم *Floyd-Warshall* است که در بخش ۲۵.۲ ارائه شده است. الگوریتم *Floyd-Warshall* در زمان $\Theta(V^3)$ اجرا می‌شود. بخش ۲۵.۲ همچنین مسئله یافتن بستار تعدی یک گراف جهت‌دار که مربوط به مسئله کوتاهترین مسیرهای بین همه جفت‌ها است را هم پوشش می‌دهد. و در انتها بخش ۲۵.۳ الگوریتم *Johnson* را ارائه می‌کند. برخلاف دیگر الگوریتم‌های این فصل، الگوریتم *Johnson* از نمایش لیست مجاورتی گراف استفاده می‌کند. این الگوریتم مسئله کوتاهترین مسیرهای بین همه جفت‌ها را در زمان $O(V^2 lgV+VE)$ حل می‌کند که از آن الگوریتمی مناسب برای گراف‌های بزرگ پراکنده می‌سازد.

قبل از ادامه بحث، لازم است یک سری قراردادها برای نمایش ماتریس مجاورتی ارائه دهیم. اوّل اینکه، بطور کلی فرض می‌کنیم که گراف ورودی $G=(V,E)$ دارای n رأس است و بنابراین $n=|V|$ دوّم اینکه از حروف بزرگ مانند W، L یا D برای نشان دادن ماتریس‌ها و از حروف کوچک اندیس‌دار مانند w_{ij}، l_{ij} یا d_{ij} برای نشان دادن عناصر ماتریس‌ها استفاده می‌کنیم. برخی ماتریس‌ها مانند $L^m=(l_{ij}^{(m)})$ یا $D^{(m)}=(d_{ij}^{(m)})$ دارای پرانتز برای نشان دادن تکرار می‌باشند. و در نهایت اینکه برای یک ماتریس A با ابعاد $n\times n$ فرض می‌کنیم که مقدار n در خصوصیّت *rows[A]* ذخیره شده است.

۲۵.۱ کوتاهترین مسیرها و ضرب ماتریس

این بخش یک الگوریتم برنامه‌سازی پویا برای مسئله کوتاهترین مسیرها همه جفت‌های گراف جهت‌دار $G=(V,E)$ ارائه می‌کند. هر حلقه اصلی برنامه پویا، درخواست عملی را انجام می‌دهد که بسیار شبیه ضرب ماتریس‌ها است، به همین جهت الگوریتم شبیه ضرب مکرر ماتریس‌ها به نظر می‌رسد .باایجاد یک الگوریتم برای مسئله کوتاهترین مسیرها بین همه جفت‌ها که دارای مرتبه زمانی $\Theta(V^4)$ است آغاز می‌کنیم و زمان اجرای آن را به $\Theta(V^3 lgV)$ بهبود می‌بخشیم.

قبل از ادامه بحث، اجازه دهید گام‌های پیشبرد الگوریتم برنامه‌سازی پویا که در فصل ۱۵ ارائه شد را به طور خلاصه بررسی می‌کنیم.

۱. توصیف ساختار یک جواب بهینه.

۲. تعریف بازگشتی مقدار یک جواب بهینه.
۳. محاسبه مقدار جواب بهینه با روش از پایین به بالا.
(گام چهارم یعنی تولید یک جواب بهینه از روی اطلاعات محاسبه شده، در تمرین‌ها بررسی می‌شود.)

ساختار کوتاهترین مسیر

با توصیف ساختار یک جواب بهینه آغاز می‌کنیم. برای مسئله کوتاهترین مسیرهای بین همه جفت‌ها در گراف $G=(V,E)$، ثابت کرده‌ایم (لم ۲۴.۱) که همه زیرمسیرهای کوتاهترین مسیر، خود، کوتاهترین مسیر هستند. فرض کنید گراف به وسیله ماتریس مجاورتی $W=(w_{ij})$ نشان داده شده است. کوتاهترین مسیر p از رأس i به رأس j را در نظر بگیرید و فرض کنید که p حداکثر m یال دارد. با فرض این که دوری با وزن منفی وجود ندارد، m مقدار متناهی است. اگر $i=j$ باشد، p دارای وزن صفر و بدون یال است. اگر رأس‌های j و i مجزا باشند، مسیر p را به $i \overset{p'}{\leadsto} k \to j$ تجزیه می‌کنیم که p' حداکثر $m-1$ یال دارد. بنا به لم ۲۴.۱، p' کوتاهترین مسیر از i به k است و بنابراین $\delta(i,j)=\delta(i,k)+w_{kj}$.

یک راه حل بازگشتی برای مسئله کوتاهترین مسیرهای بین همه جفت‌ها

اکنون فرض کنید $l_{ij}^{(m)}$ مینیمم وزن مسیر i به j که شامل حداکثر m یال است، می‌باشد. هنگامی که $m=0$ است کوتاهترین مسیری از i به j وجود دارد که شامل هیچ یالی نیست، اگر و فقط اگر $i=j$ باشد. بنابراین

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{اگر } i=j \\ \infty & \text{اگر } i\neq j \end{cases}$$

برای $m\geq 1$، $l_{ij}^{(m)}$ را به عنوان مینیمم $l_{ij}^{(m-1)}$ (وزن کوتاهترین مسیر از i به j شامل حداکثر $m-1$ یال) محاسبه می‌کنیم و مینیمم وزن هر مسیر i به j شامل حداکثر m یال که به وسیله بررسی همه ماقبل‌های j به نام k به دست می‌آید را نیز محاسبه می‌کنیم. لذا آن را به شکل بازگشتی، چنین تعریف می‌کنیم:

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1\leq k\leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\ &= \min_{1\leq k\leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\} . \end{aligned} \qquad (۲۵.۲)$$

تساوی فوق از آنجا به دست می‌آید که برای همه jها، $w_{jj}=0$ است.

وزن واقعی کوتاهترین مسیر $\delta(i,j)$ چیست؟ اگر گراف شامل دوری با وزن منفی نباشد برای هر

جفت رأس i و j که $\delta(i,j) < \infty$ است، کوتاهترین مسیری از i به j وجود دارد که ساده می‌باشد و بنابراین حداکثر $n-1$ یال دارد. یک مسیر از i به j با بیش از $n-1$ یال نمی‌تواند وزنی کمتر از کوتاهترین مسیر i به j داشته باشد. بنابراین وزن‌های واقعی کوتاهترین مسیر چنین به دست می‌آیند:

$$\delta(i,\, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots \,. \tag{۲۵.۳}$$

محاسبه وزن‌های کوتاهترین مسیر از پایین به بالا

با دریافت ماتریس $W=(w_{ij})$ به عنوان ورودی، اینک می‌توانیم سری ماتریس‌های $L^{(1)},L^{(2)},...,L^{(n-1)}$ را محاسبه کنیم، که برای $m=1,2,...,n-1$ داریم $L^{(m)}=(l_{ij}^{(m)})$ ماتریس نهایی $L^{(n-1)}$ شامل وزن‌های واقعی کوتاهترین مسیر می‌باشد. مشاهده می‌کنید که برای همه رأس‌های $i,j\in V$ $l_{ij}^{(1)}=w_{ij}$ است و لذا $L^{(1)}=W$.

قلب الگوریتم روال زیر می‌باشدکه با گرفتن $L^{(m-1)}$ و W، ماتریس $L^{(m)}$ را برمی‌گرداند. به عبارت دیگر کوتاهترین مسیر محاسبه شده تا این لحظه را به اندازه یک یال بیشتر پیش می‌برد.

```
EXTEND-SHORTEST-PATHS(L, W)
1  n ← rows[L]
2  let L′ = (l′_ij) be an n × n matrix
3  for i ← 1 to n
4      do for j ← 1 to n
5             do l′_ij ← ∞
6                for k ← 1 to n
7                    do l′_ij ← min(l′_ij, l_ik + w_kj)
8  return L′
```

این روال ماتریس $l'=(l'_{ij})$ را محاسبه و در پایان آن را برمی‌گرداند. این عمل را برای هر i و j به وسیله محاسبه تساوی (۲۵.۲) و با استفاده از L برای $L',L^{(m-1)}$ برای $L^{(m)}$ انجام می‌دهد. (ماتریس‌ها بدون اندیس نوشته شده‌اند تا ورودی و خروجی ماتریس‌ها مستقل از m شوند.) زمان اجرای این روال به خاطر وجود سه حلقه تو در تو for برابر $\Theta(n^3)$ است.

اینک می‌توانیم ارتباط آن را با ضرب ماتریس‌ها ببینیم. فرض کنید می‌خواهیم ماتریس $C=A.B$، که حاصل ضرب دو ماتریس $n\times n$ است را به دست آوریم. بنابراین برای i و $j=1,2,...,n$ عبارت زیر را محاسبه می‌کنیم

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \,. \tag{۲۵.۴}$$

مشاهده می‌کنید که اگر جایگزینی‌های زیر را در تساوی (۲۵.۲) انجام دهیم

$$
\begin{aligned}
l^{(m-1)} &\to a\,, \\
w &\to b\,, \\
l^{(m)} &\to c\,, \\
\min &\to +\,, \\
+ &\to \cdot
\end{aligned}
$$

به تساوی (۲۵.۴) می‌رسیم. بنابراین اگر این تغییرات را در *EXTEND-SHORTEST-PATHS* انجام دهیم و همچنین ∞ (همان *min*) را با 0 (همان +) جایگزین کنیم، به روالی با مرتبه زمانی صریح $\Theta(n^3)$ برای ضرب ماتریسی می‌رسیم:

```
MATRIX-MULTIPLY(A, B)
1  n ← rows[A]
2  let C be an n × n matrix
3  for i ← 1 to n
4       do for j ← 1 to n
5              do c_ij ← 0
6                 for k ← 1 to n
7                     do c_ij ← c_ij + a_ik · b_kj
8  return C
```

با بازگشت به مسئله کوتاهترین مسیرهای بین همه جفت‌ها، وزن‌های کوتاهترین مسیر را به وسیله توسعه کوتاهترین مسیرها به شکل یال به یال محاسبه می‌کنیم. با فرض این که $A.B$ حاصل ضرب ماتریس باشد که توسط *EXTEND-SHORTEST-PATHS(A,B)* برگردانده می‌شود، توالی از $n-1$ ماتریس را محاسبه می‌کنیم:

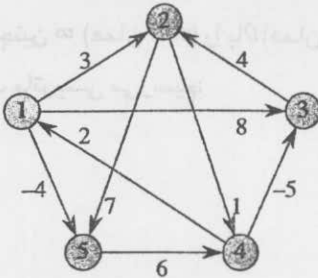
$$
\begin{aligned}
L^{(1)} &= L^{(0)} \cdot W &= W\,, \\
L^{(2)} &= L^{(1)} \cdot W &= W^2\,, \\
L^{(3)} &= L^{(2)} \cdot W &= W^3\,, \\
&\vdots \\
L^{(n-1)} &= L^{(n-2)} \cdot W &= W^{n-1}\,.
\end{aligned}
$$

همان طور که ذکر شد ماتریس $L^{(n-1)}=W^{n-1}$ شامل وزن‌های کوتاهترین مسیر است. روال بعدی، این توالی را در زمان $\Theta(n^4)$ محاسبه می‌کند.

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  n ← rows[W]
2  L^(1) ← W
```

```
3   for m ← 2 to n − 1
4        do L^(m) ← EXTEND-SHORTEST-PATHS (L^(m−1), W)
5   return L^(n−1)
```

شکل ۲۵.۱ یک‌گراف و ماتریس‌های $L^{(m)}$ که به وسیله *SLOW-ALL-PAIRS-SHORTEST-PATHS* محاسبه می‌گردد، را نشان می‌دهد.



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

شکل ۲۵.۱ یک گراف جهت‌دار و توالی ماتریس‌های $L^{(m)}$ که به وسیله *SLOW-ALL-PAIRS SHORTEST-PATHS* محاسبه می‌گردد. خواننده می‌تواند صحت برابری $L^{(5)}=L^{(4)}.W$ با $L^{(4)}$ و در نتیجه برای هر $m\geq4$، $L^{(m)}=L^{(4)}$ را تعیین کند.

بهبود زمان اجرا

هدف ما این نیست که همه $L^{(m)}$ ماتریس را محاسبه کنیم، بلکه تنها به ماتریس $L^{(n-1)}$ می‌پردازیم. به خاطر بیاورید که با نبود دور با وزن منفی، تساوی (۲۵.۳) دلالت می‌کند بر این که برای همه اعداد صحیح $m\geq n-1$ داریم $L^{(m)}=L^{(n-1)}$ درست مانند ضرب ماتریس‌ها به روش قدیمی، ضرب ماتریسی که به وسیله روال *EXTEND-SHORTEST-PATHS* انجام می‌گیرد شرکت‌پذیر است. (تمرین ۲۵.۱-۴ را ملاحظه کنید) بنابراین می‌توانیم $L^{(n-1)}$ را تنها با $\lceil lg(n-1)\rceil$ ضرب ماتریس و به وسیله محاسبه توالی زیر به دست آوریم.

$$
\begin{aligned}
L^{(1)} &= W ,\\
L^{(2)} &= W^2 = W \cdot W ,\\
L^{(4)} &= W^4 = W^2 \cdot W^2\\
L^{(8)} &= W^8 = W^4 \cdot W^4 ,\\
&\vdots\\
L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}} .
\end{aligned}
$$

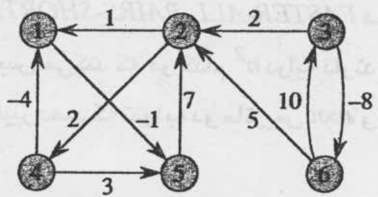
چون $2^{\lceil lg(n-1) \rceil} \geq n-1$ است، حاصل‌ضرب نهایی $L^{(2^{\lceil lg(n-1) \rceil})}$ برابر با $L^{(n-1)}$ است.

روال بعدی توالی ماتریس‌های بالا را به وسیله تکنیک مجذورکردن مکرر محاسبه می‌کند.

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
1  n ← rows[W]
2  L^(1) ← W
3  m ← 1
4  while m < n − 1
5      do L^(2m) ← EXTEND-SHORTEST-PATHS(L^(m), L^(m))
6         m ← 2m
7  return L^(m)
```

در هر تکرار حلقه *while* خطوط ۶-۴، $L^{(2m)}=(L^{(m)})^2$ را با شروع از $m=1$ محاسبه می‌کنیم. در پایان هر تکرار حلقه، مقدار m را دو برابر می‌کنیم. آخرین تکرار حلقه، $L^{(n-1)}$ را به وسیله محاسبه $L^{(2m)}$ برای $n-1 \leq 2m < 2n-2$ به دست می‌آورد. بنا به تساوی (۲۵.۳) داریم $L^{(2m)}=L^{(n-1)}$ دفعه بعد که مقایسه در خط چهارم اجرا می‌شود، m دو برابر شده است و بنابراین $m \geq n-1$ شرط نقض شده است و روال، آخرین ماتریسی که محاسبه کرده است را برمی‌گرداند.

زمان *FASTER-ALL-PAIRS-SHORTEST-PATHS* برابر $\Theta(n^3 lgn)$ است زیرا هر یک از $\lceil lg(n-1) \rceil$ ضرب ماتریس، زمان $\Theta(n^3)$ را صرف می‌کند. مشاهده می‌شود که این روال فشرده است و شامل هیچ ساختمان داده پیچیده‌ای نیست و بنابراین مقدار ضریب ثابت پنهان علامت Θ، مقدار کوچکی است.



شکل ۲۵.۲ گراف وزن‌دار جهت‌دار برای استفاده در تمرین‌های ۱-۲۵.۱ و ۱-۲۵.۲ و ۱-۲۵.۳.

تمرین‌ها

۲۵.۱ -۱ *SLOW-ALL-PAIRS-SHORTEST-PATHS* را روی گراف وزن‌دار جهت‌دار شکل

۲۵.۲ اجرا کنید و ماتریس‌های حاصل از هر تکرار حلقه رانمایش دهید. سپس این کار را برای *FASTER-ALL-PAIRS-SHORTEST-PATHS* هم انجام دهید.

۲۵.۱ -۲ چرا برای همه $1\leq i\leq n$ باید $w_{ii}=0$ باشد؟

۲۵.۱ -۳ ماتریس

$$L^{(0)}=\begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

که در الگوریتم‌های کوتاهترین مسیرها محاسبه می‌شود، با چه چیزی در ضرب ماتریسهای با قاعده متناظر است؟

۲۵.۱ -۴ نشان دهید ضرب ماتریس که به وسیله *EXTEND-SHORTEST-PATHS* تعریف شد، شرکت‌پذیر است.

۲۵.۱ -۵ نشان دهید چگونه می‌توان مسئله کوتاهترین مسیرها از یک مبدأ واحد را به عنوان حاصل ضرب ماتریس‌ها و یک بردار بیان کرد. توضیح دهید چگونه این حاصل ضرب مطابق با یک الگوریتم شبیه *Bellman-Ford* ارزشیابی می‌شود؟ (بخش ۲۴.۱ را ملاحظه نمائید)

۲۵.۱ -۶ فرض کنید می‌خواهیم رأس‌های کوتاهترین مسیرها را در الگوریتم‌های این بخش محاسبه کنیم. نشان دهید که چگونه می‌توان ماتریس ماقبل π را از روی ماتریس وزن‌های کوتاهترین مسیر کامل شده L در زمان $O(n^3)$ به دست آورد.

۲۵.۱ -۷ رأس‌های روی کوتاهترین مسیرها هم می‌توانند در همان زمان محاسبه وزن‌های کوتاهترین مسیر محاسبه گردند. $\pi_{ij}^{(m)}$ را به عنوان ماقبل رأس j روی مسیر با وزن مینیمم i به j روی مسیر با وزن مینیمم i به j که شامل حداکثر m یال است، تعریف می‌کنیم. روال‌های *EXTEND-SHORTEST-PATHS* و *SLOW-ALL-PAIRS-SHORTEST-PATHS* را چنان تغییر دهید که ماتریس‌های $\Pi^{(1)},\Pi^{(2)},...,\Pi^{(n-1)}$ را هم، مانند $L^{(1)},L^{(2)},...,L^{(n-1)}$ محاسبه کند.

۲۵.۱ -۸ روال *FASTER-ALL-PAIRS-SHORTEST-PATHS* همان طور که نوشته شد، ما را ملزم به ذخیره $\lceil lg(n-1)\rceil$ ماتریس می‌کند که هر کدام n^2 درایه دارند و بنابراین کل فضای لازم، $\Theta(n^2lgn)$ است. روال را چنان تغییر دهید که تنها به دو ماتریس $n\times n$ و در نتیجه $\Theta(n^2)$ حافظه احتیاج داشته باشد.

۲۵.۱ -۹ *FASTER-ALL-PAIRS-SHORTEST-PATHS* را چنان تغییر دهید که بتواند وجود دور با وزن منفی را مشخص کند.

۲۵.۱ - ۱۰ الگوریتمی کارآمد برای یافتن طول (تعداد یال‌ها) یک دور با وزن منفی و طول مینیمم، ارائه دهید.

۲۵.۲ الگوریتم *Floyd-Warshall*

در این بخش از یک شکل متفاوت برنامه‌سازی پویا برای حل مسئله کوتاهترین مسیرهای بین همه جفت‌ها روی یک گراف جهت‌دار $G=(V,E)$ استفاده خواهیم کرد. الگوریتم حاصل که به عنوان الگوریتم *Floyd-Warshall* شناخته می‌شود در زمان $\Theta(V^3)$ اجرا می‌گردد. مانند قبل، یال‌ها با وزن منفی هم ممکن است وجود داشته باشند امّا فرض می‌کنیم که هیچ دوری با وزن منفی وجود ندارد. همانند بخش ۲۵.۱، فرآیند برنامه‌سازی پویا را برای پیشبرد الگوریتم، دنبال کنیم. بعد از مطالعه الگوریتم حاصل، یک روش مشابه برای یافتن بستار تعدی یک گراف جهت‌دار ارائه می‌کنیم.

ساختار کوتاهترین مسیر

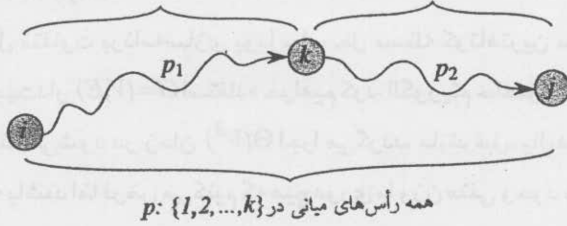
در الگوریتم *Floyd-Warshall* از توضیحی برای ساختار کوتاهترین مسیر استفاده می‌کنیم که با آن چه که در مورد الگوریتم‌های همه جفت، که مبتنی بر ضرب ماتریس‌ها بودند گفته شد، متفاوت است. این الگوریتم رأس‌های *میانی* کوتاهترین مسیر را در نظر می‌گیرد، که رأس میانی مسیر ساده $p=<v_1,v_2,...,v_l>$ هر رأس آن غیر از v_1 و v_l است، بعبارت دیگر هر رأسی در مجموعه $\{v_1,v_2,...,v_{(l-1)}\}$.

الگوریتم *Floyd-Warshall* مبتنی بر مشاهدات زیر است. در فرض ما که رأس‌های G، $V=\{1,2,...,n\}$ هستند، یک زیر مجموعه k رأسی $\{1,2,...,k\}$ را در نظر می‌گیریم. برای یک جفت رأس $i,j\in V$، همه رأس‌ها از i به j را که رأس‌های میانی آنها در $\{1,2,...,k\}$ قرار دارند در نظر بگیرید و فرض کنید p در بین آنها مسیری با وزن مینیمم است.(مسیر p یک مسیر ساده است) الگوریتم *Floyd-Warshall* از رابطه بین مسیر p و کوتاهترین مسیرها از i به j باهمه رأس‌های میانی در مجموعه $\{1,2,..,k-1\}$ استفاده می‌کند. این رابطه به این بستگی دارد که آیا k یک رأس میانی مسیر p است یا خیر.

- اگر k یک رأس میانی مسیر p نباشد آنگاه همه رأس‌های میانی مسیر p در مجموعه $\{1,2,...,k-1\}$ قرار دارند. بنابراین کوتاهترین مسیر از رأس i به رأس j با همه رأس‌های میانی واقع در مجموعه $\{1,2,...,k-1\}$ کوتاهترین مسیر از i به j با همه رأس‌های میانی واقع در مجموعه $\{1,2,...,k\}$ نیز می‌باشد.

- اگر k یک رأس میانی مجموعه p باشد، آنگاه مسیر p را مانند شکل ۲۵.۳ به $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ می‌شکنیم.بنابه لم ۲۴.۱، p_1 کوتاهترین مسیراز i به k با رأس‌های میانی واقع در $\{1,2,...,k\}$ می‌باشد.چون رأس k یک رأس میانی مسیر p_1 نیست، مشاهده می‌کنیم که p_1 کوتاهترین مسیر از i به k با رأس‌های میانی واقع در مجموعه $\{1,2,...,k-1\}$ است. به طور مشابه، p_2 کوتاهترین مسیر از رأس k به رأس j با

رأس‌های میانی واقع در مجموعه $\{1,2,...,k-1\}$ می‌باشد.



شکل ۲۵.۳ مسیر p کوتاهترین مسیر از رأس i به j است و k رأس میانی p با بزرگترین عدد می‌باشد. مسیر p_1 یعنی قسمتی از مسیر p که بین رأس i و رأس k واقع است، همه رأس‌های میانی در مجموعه $\{1,2,...,k-1\}$ را دارا است. همین مطالب برای مسیر p_2 از رأس k به رأس j برقرار می‌باشد.

یک راه حل بازگشتی برای مسئله کوتاهترین مسیرهای بین همه جفت‌ها

با اتکا به مشاهدات قبل، یک شکل بازگشتی از برآوردهای کوتاهترین مسیر و متفاوت با آن چه که در بخش ۲۵.۱ ذکر شد، تعریف می‌کنیم. فرض کنید $d_{ij}^{(k)}$ وزن کوتاهترین مسیر از رأس i به رأس j باشد که برای آن همه رأس‌های میانی در مجموعه $\{1,2,...,k\}$ قرار دارند. وقتی که $k=0$ است مسیر از رأس i به رأس j بدون رأس میانی با شماره بزرگتر از 0، هیچ رأس میانی ندارد. چنین مسیری حداکثر یک یال دارد و از این رو $d_{ij}^{(0)}=w_{ij}$ تعریفی بازگشتی در ادامه مبحث فوق به صورت زیر ارائه می‌گردد.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{اگر } k=0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{اگر } k\geq 1 \end{cases} \qquad (۲۵.۵)$$

چون برای هر مسیر، همه رأس‌های میانی در مجموعه $\{1,2,...,n\}$ واقعند، ماتریس $D^{(n)}=(d_{ij}^{(n)})$ جواب نهایی را ارائه می‌دهد: برای همه $i,j\in V$ داریم $d_{ij}^{(n)}=\delta(i,j)$

محاسبه وزن‌های کوتاهترین مسیر از پایین به بالا

بنا به رابطه بازگشتی (۲۵.۵) روال پایین به بالای زیر می‌تواند جهت محاسبه مقادیر $d_{ij}^{(k)}$ به ترتیب افزایش مقادیر k استفاده گردد. ورودی این روال، ماتریس W با ابعاد $n\times n$ است که در معادله (۲۵.۱) تعریف شد. روال، ماتریس $D^{(n)}$ شامل وزن‌های کوتاهترین مسیر را برمی‌گرداند

```
FLOYD-WARSHALL(W)
1  n ← rows[W]
2  D^(0) ← W
3  for k ← 1 to n
4       do for i ← 1 to n
```

5 **do for** $j \leftarrow 1$ **to** n

6 **do** $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$

7 **return** $D^{(n)}$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

شکل ۲۵.۴ توالی ماتریس‌های $D^{(k)}$ و $\Pi^{(k)}$ که به وسیله الگوریتم *Floyd-Warshall* برای گراف شکل ۲۵.۱، محاسبه گردیده است.

شکل ۲۵.۴ ماتریس $D^{(k)}$ که توسط الگوریتم *Floyd-Warshall* برای گراف ۲۵.۱ محاسبه گردید را نشان می‌دهد. زمان اجرای الگوریتم *Floyd-Warshall* به وسیله سه حلقه *for* تودرتو خطوط ۶-۳ تعیین می‌گردد. چون هر اجرای خط ۶، زمان $O(1)$ را صرف می‌کند، الگوریتم در زمان $\Theta(n^3)$ اجرا می‌گردد. همانند الگوریتم پایانی بخش ۲۵.۱، کد این روال فشرده است و شامل هیچ ساختمان داده پیچیده‌ای نیست و بنابراین ضریب ثابت در علامت گذاری Θ، مقداری کوچک است. بنابراین الگوریتم *Floyd-Warshall* حتی برای گراف‌های ورودی با سایز متوسط نیز قابل اعمال می‌باشد.

ساختن کوتاهترین مسیر

شیوه‌های مختلفی برای ساختن کوتاهترین مسیرها در الگوریتم *Floyd-Warshall* وجود دارد. یک راه، محاسبه ماتریس D وزن‌های کوتاهترین مسیر و سپس ساختن ماتریس ماقبل Π از روی ماتریس D می‌باشد. این شیوه می‌تواند چنان پیاده سازی شود که در زمان $O(n^3)$ اجرا گردد (تمرین ۶-۲۵.۱) می‌توان با فرستادن ماتریس ماقبل Π به روال *PRINT-ALL-PAIRS-SHORTEST-PATH*، از آن برای چاپ رأس‌های داخل کوتاهترین مسیر داده شده استفاده نمود.

می‌توانیم "همزمان" با محاسبه ماتریس‌های $D^{(k)}$ توسط الگوریتم *Floyd-Warshall*، ماتریس ماقبل Π را نیز محاسبه کنیم. به ویژه یک توالی از ماتریس‌های $\Pi^{(0)},\Pi^{(1)},...,\Pi^{(n)}$ را محاسبه کنیم که $\Pi=\Pi^{(n)}$ است و $\pi_{ij}^{(k)}$ به عنوان ماقبل رأس j در کوتاهترین مسیر از رأس i به رأس j با رأس‌های میانی واقع در مجموعه $\{1,2,...,k\}$، تعریف گردد.

می‌توانیم یک شکل بازگشتی از $\pi_{ij}^{(k)}$ ارائه دهیم. وقتی $k=0$ است، کوتاهترین مسیر از i به j ابداً هیچ رأس میانی ندارد. بنابراین

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{اگر } i=j \text{ یا } w_{ij}=\infty \\ i & \text{اگر } i\neq j \text{ و } w_{ij}<\infty \end{cases} \quad (۲۵.۶)$$

برای $k\geq 1$ اگر مسیر $i \rightsquigarrow k \rightsquigarrow j$ را در نظر بگیریم که در آن $k\neq j$ عنصر ماقبل j که انتخاب می‌کنیم همان عنصری است که به عنوان ماقبل j در کوتاهترین مسیر از k به j با همه رأس‌های میانی واقع در $\{1,2,...,k-1\}$، انتخاب کردیم. در غیر این صورت همان ماقبل j را انتخاب می‌کنیم که روی کوتاهترین مسیر از i، با همه رأس‌های میانی واقع در $\{1,2,...,k-1\}$ انتخاب کردیم. به عبارت دیگر برای $k\geq 1$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{اگر } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{اگر } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases} \quad (۲۵.۷)$$

انجام محاسبات ماتریس $\Pi^{(k)}$ در *FLOYD-WARSHALL* را به عنوان تمرین ۲- ۲۵.۲ واگذار می‌کنیم. شکل ۲۵.۴ توالی ماتریس‌های $\Pi^{(k)}$ که الگوریتم حاصل روی گراف شکل ۲۵.۱ محاسبه می‌کند

را نشان می‌دهد. این تمرین، همچنین از شما می‌خواهد ثابت کنید زیرگراف ماقبل $G_{\pi i}$، درخت کوتاهترین مسیرها با ریشه i می‌باشد که کار به مراتب سخت‌تری است.

همچنین روشی دیگر برای ساختن کوتاهترین مسیر در تمرین ۷- ۲۵.۲ ارائه شده است.

بستار تعدی یک گراف جهت دار[1]

در گراف جهت‌دار $G=(V,E)$ با مجموعه رأس $V=\{1,2,...,n\}$، ممکن است بخواهیم دریابیم که آیا برای همه جفت رأس‌های $i,j \in V$ مسیری از i به j در G وجود دارد یا نه. بستار تعدی G به صورت گراف $G^*=(V,E^*)$ تعریف می‌شود که در آن $E^*=\{(i,j):$ مسیری از رأس i به رأس j در G وجود دارد$\}$

یک راه محاسبه بستار تعدی گراف در زمان $\Theta(n^3)$ این است که وزن 1 را به هر یال E نسبت داده و الگوریتم *Floyd-Warshall* را اجرا کنیم. اگر مسیری از رأس i به رأس j وجود داشته باشد، داریم $d_{ij}<n$ در غیر این صورت $d_{ij}=\infty$ می‌شود.

یک راه مشابه دیگر برای محاسبه بستار تعدی G در زمان $\Theta(n^3)$ وجود دارد که در عمل می‌تواند در زمان و حافظه صرفه جویی کند. این روش شامل جایگزینی اعمال منطقی $\vee$ (OR منطقی) و $\wedge$ (AND منطقی) به جای اعمال حسابی min و $+$ در الگوریتم *Floyd-Warshall* است. برای $i,j,k=1,2,...,n$ مقدار $t_{ij}^{(k)}$ را 1 تعریف می‌کنیم اگر یک مسیر از رأس i به رأس j با رأس‌های میانی واقع در مجموعه $\{1,2,...,k\}$ در گراف G وجود داشته باشد و در غیر این صورت مقدار آن را 0 تعریف می‌کنیم. بستار تعدی $G^*=(V,E^*)$ را با قرار دادن یال (i,j) در E^* تولید می‌کنیم اگر و فقط اگر $t_{ij}^{(n)}=1$ باشد. یک تعریف بازگشتی از $t_{ij}^{(k)}$ مشابه رابطه بازگشتی (۲۵.۵) چنین است:

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E , \\ 1 & \text{if } i = j \text{ or } (i,j) \in E , \end{cases}$$

و برای $k \geq 1$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right) .$$

(۲۵.۸)

همانند الگوریتم *Floyd-Warshall*، ماتریس‌های $T^{(k)}=(t_{ij}^{(k)})$ را به ترتیب افزایش k محاسبه می‌کنیم.

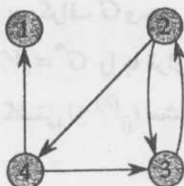
```
TRANSITIVE-CLOSURE(G)
1  n ← |V[G]|
2  for i ← 1 to n
3       do for j ← 1 to n
4              do if i = j or (i, j) ∈ E[G]
```

---

[1] Transitive closure

```
5                then t_ij^(0) ← 1
6                else t_ij^(0) ← 0
7   for k ← 1 to n
8       do for i ← 1 to n
9              do for j ← 1 to n
10                 do t_ij^(k) ← t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
11  return T^(n)
```

شکل ۲۵.۵، ماتریس‌های $T^{(k)}$ که به وسیله روال *TRANSITIVE-CLOSURE* روی یک گراف نمونه محاسبه گردیده‌اند رانشان می‌دهد. روال *TRANSITIVE-CLOSURE* همانند الگوریتم *Floyd-Warshall* درزمان $\Theta(n^3)$ اجرا می‌گردد. ولی روی بعضی از کامپیوترها اعمال منطقی روی مقادیر تک بیت، سریع‌تر از اعمال حسابی روی داده‌های کلمه‌ای صحیح اجرا می‌شوند. علاوه بر آن چون الگوریتم بستار تعدی مستقیم، تنها از مقدار بولی به جای مقادیر صحیح استفاده می‌کند، فضای مورد نیاز آن به اندازه یک ضریب به بزرگی حجم کلمه در حافظه کامپیوتر، کمتر از الگوریتم *Floyd-Warshall* است.



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

شکل ۲۵.۵ گراف جهت‌دار و ماتریس‌های $T^{(k)}$ که به وسیله الگوریتم بستار - تعدی محاسبه شده‌اند.

تمرین‌ها

۱- ۲۵.۲ الگوریتم *Floyd-Warshall* را روی گراف جهت‌دار وزن‌دار شکل ۲-۲۵ اجرا کنید. ماتریس

$D^{(k)}$ که در هر تکرار حلقه بیرونی حاصل می‌گردد را نمایش دهید.

۲۵.۲-۲ نشان دهید که چگونه می‌توان بستار تعدی را با استفاده از تکنیک بخش ۲۵.۱ محاسبه کرد.

۲۵.۲-۳ *FLOYD-WARSHALL* را چنان تغییر دهید که مطابق با معادلات (۲۵.۶) و (۲۵.۷)، محاسبه ماتریس‌های $\Pi^{(k)}$ را نیز شامل می‌شود. به طور دقیق ثابت کنید برای همه $i \in V$ زیرگراف ماقبل $G_{\pi,i}$، درخت کوتاهترین مسیرها با ریشه i است. (راهنمایی: برای نشان دادن این که $G_{\pi,i}$ بدون دور است، ابتدا نشان دهید که $\pi_{ij}^{(k)}=l$ بر طبق تعریف $\pi_{ij}^{(k)}$ بر این دلالت می‌کند که $d_{ij}^{(k)} \geq d_{il}^{(k)} + w_{lj}$ سپس از اثبات لم ۲۴.۱۶ استفاده نمائید.)

۲۵.۲-۴ همان طور که در فوق مشخص شد الگوریتم *Floyd-Warshall* به اندازه $\Theta(n^3)$ فضا لازم دارد زیرا برای هر $i,j,k=1,2,...,n$ $d_{ij}^{(k)}$ را محاسبه می‌کنیم. نشان دهید روال زیر که به سادگی همه اندیس‌های بالایی را حذف کرده صحیح می‌باشد و بنابراین تنها به اندازه $\Theta(n^2)$ فضا لازم داریم.

```
FLOYD-WARSHALL'(W)
1  n ← rows[W]
2  D ← W
3  for k ← 1 to n
4      do for i ← 1 to n
5             do for j ← 1 to n
6                    do d_ij ← min (d_ij , d_ik + d_kj)
7  return D
```

۲۵.۲-۵ فرض کنید تساوی معادله (۲۵.۷) را به شکل زیر تغییر دهیم.

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \quad \text{اگر} \\ \pi_{kj}^{(k-1)} & d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \quad \text{اگر} \end{cases}$$

آیا این تعریف دیگر ماتریس ماقبل Π، صحیح است؟

۲۵.۲-۶ چگونه می‌توان از خروجی الگوریتم *Floyd-Warshall* جهت کشف وجود دور با وزن منفی استفاده نمود؟

۲۵.۲-۷ راه دیگر برای تولید مجدد کوتاهترین مسیرها در الگوریتم *Floyd-Warshall* برای هر $i,j,k=1,2,...,n$ از مقادیر $\phi_{ij}^{(k)}$ استفاده می‌کند، که $\phi_{ij}^{(k)}$ بزرگترین عدد رأس میانی کوتاهترین مسیر از i به j است که در آن مسیر، همه رأس‌های میانی در مجموعه $\{1,2,...,k\}$ واقعند. یک فرمول بازگشتی برای $\phi_{ij}^{(k)}$ ارائه دهید. *FLOYD-WARSHALL* را چنان تغییر دهید که مقادیر $\phi_{ij}^{(k)}$ را محاسبه کند و روال *PRINT-ALL-PAIRS-SHORTEST-PATH* را به شکلی بازنویسی کنید که ماتریس $\phi=(\phi_{ij}^{(n)})$ را به عنوان ورودی دریافت کند. ماتریس ϕ از چه جهت شبیه جدول s در مسئله ضرب

زنجیره‌ای این ماتریس‌ها در بخش ۱۵.۲ است؟

۸- ۲۵.۲ یک الگوریتم با مرتبه زمانی $O(VE)$ برای محاسبه بستار تعدی گراف جهت دار $G=(V,E)$ ارائه دهید.

۹- ۲۵.۲ فرض کنید بستار تعدی یک گراف جهت‌دار بدون دور، می‌تواند در زمان $f=(|V|, |E|)$ محاسبه گردد، که f یک تابع صعودی یکنواخت از $|V|$ و $|E|$ است. نشان دهید که زمان لازم برای محاسبه بستار تعدی $G^*=(V,E^*)$ یک گراف معمول جهت‌دار $G=(V,E)$، برابر با $f(|V|,|E|)+O(V+E^*)$ است.

۲۵.۳ الگوریتم *Johnson* برای گراف‌های پراکنده (خلوت)[1]

الگوریتم *Johnson* کوتاهترین مسیرها بین همه جفت‌ها را در زمان $O(V^2lgV+VE)$ پیدا می‌کند. برای گراف‌های پراکنده، این الگوریتم به طور مجانبی بهتر از مجذور کردن مکرر ماتریس‌ها و یا الگوریتم *Floyd-Warshall* است. الگوریتم، یا ماتریس وزن‌های کوتاهترین مسیر برای همه جفت‌ها را برمی‌گرداند، یا بیان می‌کند که گراف ورودی شامل دوری با وزن منفی است. الگوریتم *Johnson* از الگوریتم‌های *Dijkstra* و *Bellman-Ford* که در فصل ۲۴ بررسی شدند، به عنوان زیر روال استفاده می‌کند.

الگوریتم *Johnson* از تکنیک وزن دهی مجدد[2] استفاده می‌کند که به شکل زیر عمل می‌کند. اگر همه وزن‌های w در گراف $G=(V,E)$ غیر منفی باشند، می‌توانیم کوتاهترین مسیرها بین همه جفت رأس‌ها را با اجرای الگوریتم *Dijkstra* یک بار برای هر رأس، به دست می‌آوریم؛ با استفاده از صف مینیمم اولویّت *heap* فیبوناچی، زمان اجرای این الگوریتم همه جفت‌ها $O(V^2lgV+VE)$ است. اگر G دارای یال‌های با وزن منفی باشد اما دور با وزن منفی نداشته باشد، به سادگی مجموعه جدیدی شامل وزن‌های غیر منفی یال‌ها محاسبه می‌کنیم که به ما اجازه می‌دهد که از همان شیوه قبلی استفاده کنیم. مجموعه جدید وزن‌ها $\widehat{w}$ باید دو ویژگی مهم داشته باشد.

۱. برای همه جفت رأس‌های $u,v\in V$ مسیر p کوتاهترین مسیر از u به v با استفاده از تابع وزن w است اگر و فقط اگر p کوتاهترین مسیر از u به v با استفاده از تابع وزن $\widehat{w}$ نیز باشد.

۲. برای همه یال‌های (u,v)، وزن جدید $\widehat{w}(u, v)$ غیر منفی است.

همان طور که خواهیم دید پیش پردازش G برای تعیین تابع جدید وزن $\widehat{w}$ می‌تواند در زمان $O(VE)$ اجرا شود.

1.sparse graphs

2.reweighting

حفظ کوتاهترین مسیرها بوسیله وزن دهی مجدد

همان طور که لم بعد نشان می‌دهد، به سهولت وزن دهی مجدد یال‌ها مطرح می‌گردد که اوّلین ویژگی بالا را برقرار می‌کند. از δ برای نشان دادن وزن‌های کوتاهترین مسیر که از تابع وزن w به دست می‌آید استفاده می‌کنیم و از $\widehat{\delta}$ برای نشان دادن وزن‌های کوتاهترین مسیر که از تابع وزن $\widehat{w}$ به دست می‌آید استفاده می‌کنیم.

لم ۲۵.۱ (وزن دهی مجدد، کوتاهترین مسیرها را تغییر نمی‌دهد)

در گراف وزن‌دار جهت‌دار $G=(V,E)$ با تابع وزن $w:E\rightarrow R$، فرض کنید $h:V\rightarrow R$ تابعی باشد که رأس‌ها را به اعداد حقیقی نگاشت می‌کند. برای هر یال $(u,v)\in E$ تعریف می‌کنیم:

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v)\ . \qquad (۲۵.۹)$$

فرض کنید $p=<v_0 v_1 ,...,v_k>$ مسیری از رأس v_0 به رأس v_k باشد. آنگاه p کوتاهترین مسیر از v_0 به v_k با تابع وزن w است اگر و فقط اگر کوتاهترین مسیر با تابع وزن $\widehat{w}$ باشد. به بیان دیگر $w(p)=\delta(v_0 v_k)$ اگر و فقط اگر $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$.

همچنین، G در استفاده از تابع وزن w دارای دوری با وزن منفی است اگر و فقط اگر G در استفاده از تابع وزن $\widehat{w}$ دارای دوری با وزن منفی باشد.

اثبات با نشان دادن این رابطه آغاز می‌کنیم که

$$\widehat{w}(p) = w(p) + h(v_0) - h(v_k)\ . \qquad (۲۵.۱۰)$$

داریم:

$$\begin{aligned}
\widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \qquad \text{(چون سری دارای خاصیت تلسکوپی است)} \\
&= w(p) + h(v_0) - h(v_k)\ .
\end{aligned}$$

بنابراین هر مسیر p از v_0 به v_k دارای $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$ است. اگر یک مسیر از v_0 به v_k در استفاده از تابع وزن w کوتاهتر از یک مسیر دیگر باشد، آنگاه این مسیر، در استفاده از $\widehat{w}$ نیز کوتاهتر می‌باشد. لذا $w(p)=\delta((v_0, v_k)$ اگر و فقط اگر $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$.

در نهایت، نشان می‌دهیم که G در استفاده از تابع وزن w، دوری با وزن منفی دارد اگر و فقط اگر G در استفاده از تابع وزن $\widehat{w}$ دوری با وزن منفی داشته باشد. دور $c=\langle v_0, v_1, ..., v_k \rangle$ را در نظر بگیرید که در آن $v_0=v_k$. بنا به معادله (۲۵.۱۰)

$$\begin{aligned}\widehat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c)\,,\end{aligned}$$

و بنابراین c در استفاده از w، دارای وزن منفی می‌باشد اگر و فقط اگر در استفاده از $\widehat{w}$ دارای وزن منفی باشد. ■

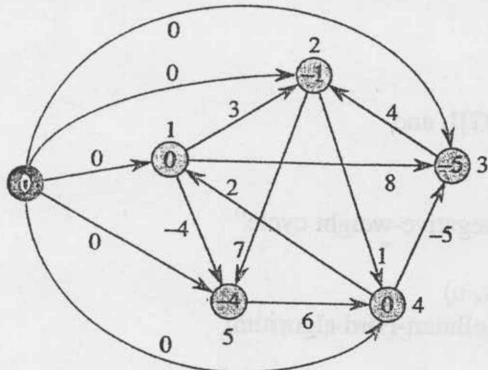
تولید وزن‌های غیر منفی به وسیله وزن دهی مجدد

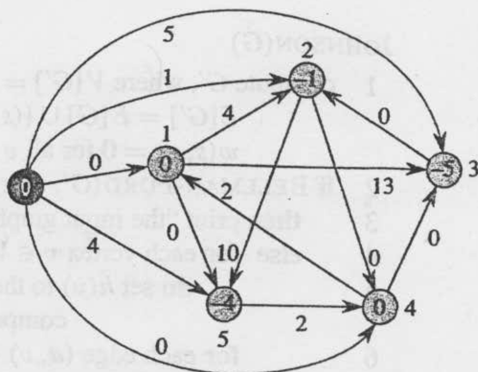
هدف بعدی ما این است که مطمئن شویم ویژگی دوم حفظ می‌گردد: می‌خواهیم $\widehat{w}(u,v)$ برای همه یال‌های $(u,v)\in E$، غیر منفی باشد. گراف وزن‌دار و جهت‌دار $G=(V,E)$ با تابع وزن $w:E\rightarrow R$ داده شده، گراف جدید $G'=(V',E')$ را می‌سازیم که برای رأس جدید $\{s\}$ $V'=V\cup\{s\}$، $s\notin V$ و $E'=E\cup\{(s,v):v\in V\}$ تابع وزن w را چنان توسعه دهیم که برای همه $v\in V$، $w(s,v)=0$ باشد. توجّه کنید که چون s یال ورودی ندارد، هیچ کوتاهترین مسیرهایی در G'، غیر از آنهایی که با مبدأ s آغاز می‌شوند s را شامل نمی‌گردند. علاوه بر این، G' دارای دور با وزن منفی نمی‌باشد اگر و فقط اگر G دارای دور با وزن منفی نباشد. شکل (a) ۲۵.۶ ، گراف G' متناظر با گراف G شکل ۲۵.۱ را نشان می‌دهد.

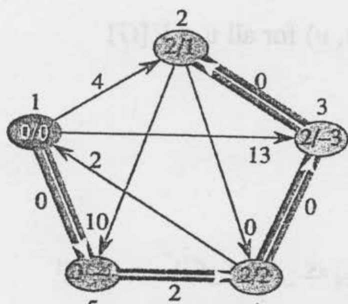
اینک فرض کنید G و G' هیچ دوری با وزن منفی نداشته باشند. برای تمام $v\in V'$، تعریف می‌کنیم $h(v)=\delta(s,v)$ بنا به نامساوی مثلث (لم ۲۴.۱۰) برای همه یال‌های $(u,v)\in E'$ داریم $h(v) \leq h(u) + w(u,v)$. بنابراین اگر وزن‌های جدید w' را طبق معادله (۲۵.۹) تعریف کنیم، داریم $\widehat{w}(u,v) = w(u,v) + h(u) - h(v) \geq 0$ ، ویژگی دوم برقرار می‌شود. شکل (b) ۲۵.۶ گراف G' شکل (a) ۲۵.۶ با یال‌هایی که مجدداً وزن دهی شده‌اند را نشان می‌دهد.

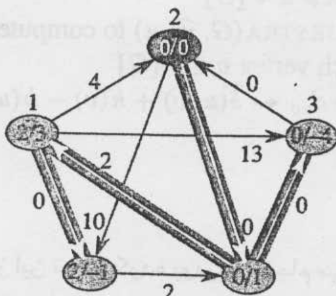
محاسبه کوتاهترین مسیرهای بین همه جفت‌ها

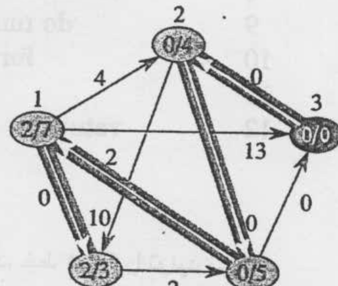
الگوریتم *Johnson* برای محاسبه کوتاهترین مسیرهای بین همه جفت‌ها از الگوریتم‌های *Bellman-Ford* (بخش ۲۴.۱) و *Dijkstra* (بخش ۲۴.۳) به عنوان زیر روال استفاده می‌کند. فرض می‌شود که یال‌ها در لیست‌های هم جواری ذخیره شده‌اند. الگوریتم، ماتریس $D=d_{ij}$ با ابعاد $|V|\times|V|$ را برمی‌گرداند که $d_{ij}=\delta(i,j)$ یا گزارش می‌دهد که گراف ورودی شامل دور با وزن منفی

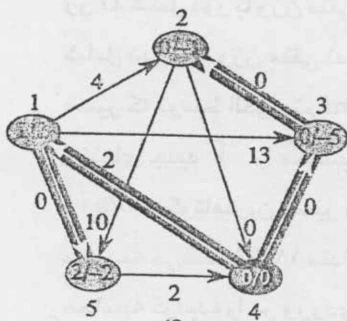


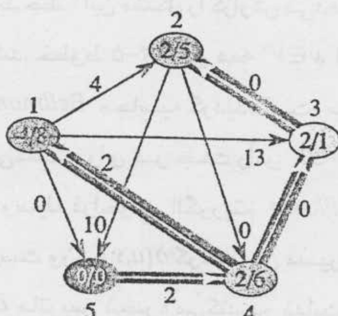












شکل ۲۵.۶ اجرای الگوریتم *Johnson* کوتاهترین مسیرهای بین همه جفت‌ها روی گراف شکل ۲۵.۱. *(a)* گراف G' با تابع وزن اولیه w. رأس جدید s سیاه رنگ است. مقدار داخل هر رأس v برابر است با $h(v)=\delta(s,v)$ *(b)* هر یال (u,v) توسط تابع وزن $\widehat{w}(u,v) = w(u,v) + h(u) - h(v)$ وزن‌دهی مجدد شده است. *(c)* – *(g)* نتیجه اجرای الگوریتم *Dijkstra* روی هر رأس G با استفاده ازتابع وزن w. در هر قسمت، رأس مبدأ u سیاه‌رنگ است و یال‌های سایه‌زده شده در درخت کوتاهترین‌مسیرهای محاسبه شده توسط الگوریتم قرار دارند که بوسیله "/" از هم جدا شده‌اند. مقدار $d_{uv}=\delta(u,v)$ برابر است با $\widehat{\delta}(u,v) + h(v) - h(u)$.

می‌باشد. همان طور که برای الگوریتم کوتاهترین مسیرهای بین همه جفت‌ها معمول است، فرض می‌کنیم که رأس‌ها از 1 تا $|V|$ شماره گذاری شده‌اند.

```
JOHNSON(G)
 1  compute G′, where V[G′] = V[G] ∪ {s},
            E[G′] = E[G] ∪ {(s, v) : v ∈ V[G]}, and
            w(s, v) = 0 for all v ∈ V[G]
 2  if BELLMAN-FORD(G′, w, s) = FALSE
 3     then print "the input graph contains a negative-weight cycle"
 4     else for each vertex v ∈ V[G′]
 5              do set h(v) to the value of δ(s, v)
                        computed by the Bellman-Ford algorithm
 6          for each edge (u, v) ∈ E[G′]
 7              do ŵ(u, v) ← w(u, v) + h(u) − h(v)
 8          for each vertex u ∈ V[G]
 9              do run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ V[G]
10                 for each vertex v ∈ V[G]
11                     do d_uv ← δ̂(u, v) + h(v) − h(u)
12          return D
```

این کد به سادگی اعمالی که پیش از این تعیین کرده بودیم را انجام می‌دهد. خط ۱، G' را تولید می‌کند. خط ۲، الگوریتم *Bellman-Ford* را روی G' با تابع وزن w در رأس مبدأ s اجرا می‌کند. اگر G'، و از این رو G، شامل دور با وزن منفی باشد خط ۳ این مشکل را گزارش می‌دهد. خطوط ۱۱-۴ فرض می‌کنند G' شامل دوری با وزن منفی نمی‌باشد. خطوط ۵-۴، برای همه $v \in V'$، $h(v)$ را با وزن $\delta(s,v)$ کوتاهترین مسیر که توسط الگوریتم *Bellman-Ford* محاسبه گردیده است، مقدار دهی می‌کند. خطوط ۷-۶ وزنهای جدید $\hat{w}$ را محاسبه می‌کنند. برای هر جفت رأس $u,v \in V$ حلقه *for* خطوط ۱۱-۸ وزن $\hat{\delta}(u,v)$ کوتاهترین مسیر را به وسیله فراخوانی الگوریتم *Dijkstra* یک بار از هر رأس مجموعه V محاسبه می‌کند. خط ۱۱ مقدار درست وزن $\delta(u,v)$ کوتاهترین مسیر که با استفاده از معادله (۲۵.۱۰) محاسبه گردیده را در ورودی d_{uv} ماتریس ذخیره می‌کند. در نهایت خط ۱۲ ماتریس کامل شده D را برمی‌گرداند. شکل ۲۵.۶ اجرای الگوریتم *Johnson* را نشان می‌دهد.

اگر صف مینیمم اولویت در الگوریتم *Dijkstra* با استفاده از *heap* فیبوناچی پیاده‌سازی شود، زمان اجرای الگوریتم *Johnson*، برابر است با $O(V^2 lgV+VE)$. پیاده‌سازی ساده‌تر *min-heap* دودویی، زمان اجرای $O(VElgV)$ را حاصل می‌کند که باز هم به طور مجانبی سریعتر از الگوریتم *Flod-Warshall* است اگر گراف خلوت باشد.

تمرین‌ها

۲۵.۳-۱ از الگوریتم *Johnson* برای یافتن کوتاهترین مسیرهای بین همه جفت رأس‌های گراف شکل ۲۵.۲ استفاده کنید. مقادیر h و $\widehat{w}$ که توسط الگوریتم محاسبه می‌گردند را نشان دهید.

۲۵.۳-۲ هدف از اضافه کردن رأس جدید s به V و تولید V' چیست؟

۲۵.۳-۳ فرض کنید برای همه یال‌های $(u,v) \in E$ داشته باشیم $w(u,v) \geq 0$. رابطه بین توابع وزن w و $\widehat{w}$ چیست؟

۲۵.۳-۴ پروفسور *Greenstreet* ادّعا می‌کند راه ساده‌تری از آنچه در الگوریتم *Johnson* استفاده شد برای وزن دهی مجدد یال‌ها وجود دارد. با قرار دادن $w^* = min_{(u,v) \in E}\{w(u,v)\}$ برای همه یال‌ها $(u,v) \in E$ چنین تعریف می‌کنیم: $\widehat{w}(u, v) = w(u, v) - w^*$
روش پروفسور در وزن دهی مجدد چه اشکالی دارد؟

۲۵.۳-۵ فرض کنید الگوریتم *Johnson* را روی گراف جهت‌دار G با تابع وزن w اجرا کنیم. نشان دهید اگر G شامل دور c با وزن 0 باشد، آنگاه برای هر یال (u,v) در c داریم $\widehat{w}(u, v)=0$.

۲۵.۳-۶ پروفسور *michener* ادّعا می‌کند که احتیاجی به ایجاد یک رأس مبدأ جدید در خط ۱ روال *JOHNSON* وجود ندارد. او ادّعا می‌کند به جای آن می‌توانیم از $G'=G$ استفاده کنیم و s را رأسی در $V[G]$ قرار دهیم. نمونه‌ای از یک گراف وزن‌دار جهت‌دار G ارائه دهید که برای آن، ترکیت ایدهٔ پروفسور و روال *JOHNSON* باعث ایجاد جواب‌های نادرست شود. سپس نشان دهید که اگر G همبند قوی باشد (هر رأس از همه رأس‌های دیگر قابل دستیابی باشد)، نتایجی که توسط روال *JOHNSON* همراه با تغییرات مورد نظر پروفسور برگردانده می‌شوند درست هستند.

مسائل

۲۵-۱ بستار تعدی یک گراف پویا

فرض کنید می‌خواهیم بستار تعدی گراف جهت‌دار $G=(V,E)$ را وقتی که یال‌ها را در E درج می‌کنیم، حفظ کنیم. به عبارت دیگر پس از هر یالی که درج شده است می‌خواهیم بستار تعدی یال هایی که تا به حال درج شده‌اند را به روزرسانی کنیم. فرض کنید G در ابتدا هیچ یالی نداشته باشد و بستار تعدی به وسیله یک ماتریس بولی نشان داده شود.

a. نشان دهید چگونه وقتی یک یال جدید به G اضافه می‌شود بستار تعدی $G^*=(V,E^*)$ گراف $G=(V,E)$ می‌تواند در زمان $O(V^2)$ به روزرسانی شود.

b. مثالی از گراف G و یک یال e ارائه دهید بطوریکه زمان $\Omega(V^2)$ برای به روزرسانی بستار تعدی پس از درج e در G، لازم باشد.

c. الگوریتمی کارآمد برای به روزرسانی بستار تعدی وقتی که یال‌ها در گراف درج می‌شوند، بیان نمائید. برای یک توالی از n درج، الگوریتم شما باید در زمان کل $\sum_{i=1}^{n} t_i = O(V^3)$ اجراگردد، که t_i زمان به روزرسانی بستار تعدی وقتی که iامین یال درج می‌شود می‌باشد. ثابت کنید الگوریتم شما به این حد زمانی دست می‌یابد.

۲-۲۵ کوتاهترین مسیرها در گراف‌های ϵ- چگال[1]

گراف $G=(V,E)$، ϵ- چگال است اگر برای ثابت ϵ در بازه $0<\epsilon\leq 1$، $|E| = \Theta(V^{1+\epsilon})$. با استفاده از d min-heap - تایی (مسئله ۲-۶ را ملاحظه کنید) در الگوریتم‌های کوتاهترین مسیرها روی گراف‌های ϵ - چگال می‌توانیم به زمان‌های اجرای الگوریتم‌های مبتنی بر *heap* فیبوناچی برسیم، بدون استفاده از ساختمان داده‌ای با این پیچیدگی.

a. زمان‌های اجرای مجانبی *INSERT*، *EXTRACT-MIN* و *DECREASE-KEY* به عنوان توابعی از d و تعداد عناصر یعنی n در یک d- *min-heap* تایی چیست؟ اگر برای ثابت $0<\alpha\leq 1$ قرار دهیم $d = \Theta(n^{\alpha})$، زمان‌های اجرای این روال‌ها چیست؟ این زمان‌های اجرا را با هزینه‌های سرشکن شده این اعمال بر روی یک *heap* فیبوناچی مقایسه کنید.

b. نشان دهید چگونه می‌توان در گراف ϵ-چگال جهت‌دار $G=(V,E)$ که یال با وزن منفی ندارد، کوتاهترین مسیرها از یک مبدأ واحد را در زمان $O(E)$ محاسبه نمود. (راهنمایی: d را به عنوان تابعی از ϵ در نظر بگیرید.)

c. نشان دهید چگونه می‌توان در گراف ϵ-چگال جهت‌دار $G=(V,E)$ که یال با وزن منفی ندارد، مسئله کوتاهترین مسیرهای بین همه جفت‌ها را در زمان $O(VE)$ حل کرد.

d. نشان دهید چگونه می‌توان در گراف ϵ- چگال جهت‌دار $G=(V,E)$ که ممکن است یال هایی با وزن منفی داشته باشد اما شامل دور با وزن منفی نیست، مسئله کوتاهترین مسیرهای همه جفت‌ها را در زمان $O(VE)$ حل نمود.

---

1. ϵ-dense graphs

۲۶ ماکزیمم جریان[1]

همان طور که می‌توانیم برای یافتن کوتاهترین مسیر از یک نقطه به یک نقطهٔ دیگر، نقشه جاده را با یک گراف جهت‌دار مدل کنیم، گراف جهت‌دار را نیز می‌توانیم به عنوان یک "شبکه جریان" تفسیر کنیم و از آن برای پاسخ به سئوالات مربوط به جریان‌های داده استفاده کنیم. خط سیر یک ماده در یک سیستم، از یک منبع که ماده در آن تولید می‌شود به یک چاه که در آن مصرف می‌شود را در نظر بگیرید. منبع ماده را با سرعتی یکنواخت تولید می‌کند و چاه آن را با همان سرعت مصرف می‌کند. "جریان" ماده در هر نقطه سیستم به طور شهودی نرخ حرکت آن است. شبکه‌های جریان می‌توانند در مدل کردن جریان مایعات در لوله‌ها، بخش‌های خطوط مونتاژ، جریان در شبکه‌های الکتریکی، اطلاعات در شبکه‌های ارتباطی و غیره استفاده شوند.

هر یال جهت‌دار در یک شبکه جریان می‌تواند مجرای عبور ماده در نظر گرفته شود. هر مجرا یک ظرفیت تعیین شده دارد که ماکزیمم نرخ جریان در آن مجرا می‌باشد، مانند عبور *200* گالون مایع در ساعت از لوله یا عبور *20* آمپر جریان الکتریکی در طول یک سیم. رأس‌ها محل‌های تقاطع مجرا هستند و غیر از منبع و چاه، ماده درون رأس‌ها بدون جمع شدن در آنها جریان می‌یابد. به بیان دیگر، نرخ ورود ماده به یک رأس با نرخ خروج ماده از آن رأس باید برابر باشد. این ویژگی را "بقاء جریان" می‌نامیم و معادلست با قانون جریان کریشهوف، وقتی که ماده جریان الکتریکی باشد.

در مسئله ماکزیمم جریان، می‌خواهیم بزرگ‌ترین نرخی که ماده با آن نرخ می‌تواند از منبع به چاه بدون تخلف از محدودیت‌های ظرفیتی جریان پیدا کند را محاسبه کنیم. این مسئله، یکی از ساده‌ترین مسایل مربوط به شبکه‌های جریان می‌باشد، همان طور که در این فصل خواهیم دید این مسئله می‌تواند به وسیله الگوریتم‌های کارآمدی حل شود. علاوه بر آن، تکنیک اساسی که در الگوریتم‌های ماکزیمم جریان مورد استفاده قرار می‌گیرند می‌توانند برای حل مسایل دیگر شبکه جریان تطبیق داده شوند.

1.maximum flow

این فصل دو شیوه کلی برای حل مسئله ماکزیمم جریان ارائه می‌دهد. بخش ۲۶.۱ ایده‌های شبکه‌های جریان و جریان‌ها را فرموله می‌کند و به طور رسمی مسئله ماکزیمم جریان را تعریف می‌کند. بخش ۲۶.۲ شیوه کلاسیک *Ford* و *Fulkerson* برای یافتن جریان‌های ماکزیمم را بیان می‌کند. یک کاربرد این شیوه، یافتن ماکزیمم تطبیق در یک گراف دو بخشی بدون جهت است که در بخش ۲۶.۳ ارائه شده است. بخش ۲۶.۴، روش *راندن* - بر چسب دهی مجدد را ارائه می‌کند، که زمینه بسیاری از سریع‌ترین الگوریتم‌ها برای مسایل شبکه جریان می‌باشد. بخش ۲۶.۵، الگوریتم "بر چسب دهی مجدد - به - جلو" را پوشش می‌دهد که پیاده‌سازی خاصی از روش راندن و بر چسب دهی مجدد است و در زمان $O(V^3)$ اجرا می‌گردد. اگر چه این الگوریتم سریع‌ترین الگوریتم شناخته شده نیست، اما تکنیک هایی را شرح می‌دهد که در سریع‌ترین الگوریتم‌ها از نظر مجانبی، استفاده می‌شوند و به طور معقول در عمل الگوریتم کارآمدی است.

۲۶.۱ شبکه‌های جریان [1]

در این بخش یک تعریف تئوری گراف از شبکه جریان را ارائه می‌دهیم، ویژگی‌های آنها را مورد بحث قرار داده و مسئله ماکزیمم جریان را به دقت تعریف می‌کنیم. همچنین علامت نویسی مفیدی را نیز معرفی می‌کنیم.

شبکه‌های جریان و جریان‌ها

شبکه جریان $G=(V,E)$ یک گراف جهت‌دار است که در آن هر یال $(u,v)\in E$ یک *ظرفیت* غیر منفی $c(u,v)\geq 0$ دارد. اگر $(u,v)\notin E$، فرض می‌کنیم $c(u,v)=0$ دو رأس شبکه جریان را مشخص می‌کنیم: منبع [2] s و چاه [3] t برای سهولت فرض می‌کنیم هر رأس روی مسیری از منبع به چاه قرار دارد. به عبارت دیگر برای هر رأس $v\in V$، مسیر $s \rightsquigarrow v \rightsquigarrow t$ وجود دارد. بنابراین گراف همبند است و $|E| \geq |V|-1$. شکل ۲۶.۱ نمونه‌ای از یک شبکه جریان را نمایش می‌دهد.

اینک آماده‌ایم که جریان‌ها را به طور رسمی‌تر تعریف کنیم. فرض کنید $G=(V,E)$ یک شبکه جریان باتابع ظرفیت c باشد. s را منبع و t را چاه شبکه در نظر بگیرد. یک *جریان* در G تابعی با مقدار حقیقی $f : V \times V \rightarrow \mathbf{R}$ است که در سه ویژگی زیر صدق می‌کند:

محدودیت ظرفیت: [4] برای همه $u,v\in V$ باید $f(u,v)\leq c(u,v)$ باشد.

تقارن مورب: [5] برای همه $u,v\in V$ باید $f(u,v)= -f(v,u)$ باشد.

---

1. flow networks
2. source
3. sink
4. capacity constraint
5. skew symmetry

بقاء جریان:[1] برای همه $u \in V-\{s,t\}$ باید

$$\sum_{v\in V} f(u, v) = 0$$

باشد.

کمیّت $f(u,v)$ که می‌تواند مثبت، صفر یا منفی باشد جریان از رأس u به رأس v نامیده می‌شود. مقدار جریان f به شکل زیر تعریف می‌شود:

$$|f| = \sum_{v\in V} f(s, v) \,, \tag{۲۶.۱}$$

به عبارت دیگر مجموع کل جریان خروجی از منبع. (در اینجا علامت $|\cdot|$ بر مقدار جریان دلالت دارد نه مقدار قدر مطلق یا اندازه.) در مسئله ماکزیمم جریان، شبکه جریان G با منبع s و چاه t مفروض است و می‌خواهیم جریان با مقدار ماکزیمم را پیدا کنیم.

قبل از دیدن مثالی از مسئله شبکه جریان، سه ویژگی را به طور مختصر بررسی می‌کنیم. محدودیت ظرفیت به سادگی می‌گوید که جریان از یک رأس به رأس دیگر نباید از ظرفیت داده شده تجاوز کند. تقارن مورّب ویژگی است که بیان می‌دارد جریان از رأس u به رأس v برابر مقدار منفی جریان در جهت معکوس است. ویژگی بقاء جریان بیان می‌کند که مجموع کل جریان ورودی و خروجی از یک رأس غیر از منبع و چاه، برابر صفر است. بنا به تقارن مورّب می‌توانیم ویژگی بقاء جریان را به صورت زیر بازنویسی کنیم: برای همه $v \in V - \{s,t\}$

$$\sum_{u\in V} f(u, v) = 0$$

به عبارت دیگر جریان کل به داخل یک رأس برابر صفر است.

هنگامی که (u,v) و (v,u) در E نیستند، جریانی بین u و v وجود ندارد و $f(u,v)=f(v,u)=0$ (تمرین ۲۶.۱-۱ از شما می‌خواهد این ویژگی را به صورت رسمی اثبات کنید.)

آخرین بخش از بررسی ما مربوط می‌شود به ویژگی‌های جریان که متعلق به جریان‌های مثبت است. *جریان کل مثبت ورودی* به رأس v به صورت زیر تعریف می‌شود:

$$\sum_{\substack{u\in V \\ f(u,v)>0}} f(u, v) \,. \tag{۲۶.۲}$$

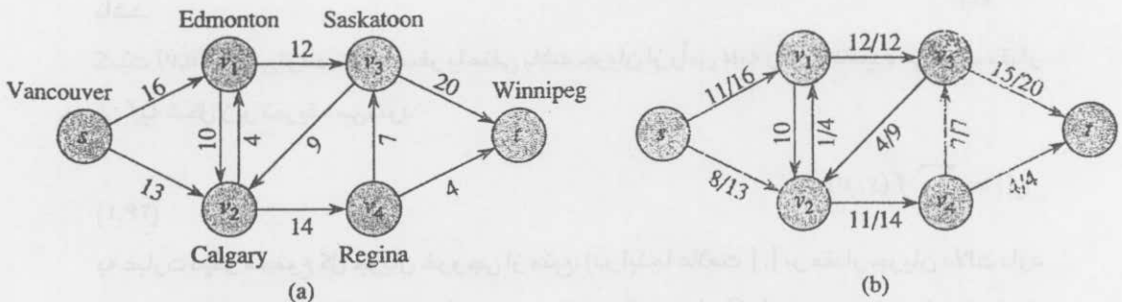
جریان کل مثبت خروجی از رأس به طور متقارن تعریف می‌شود. *جریان خالص کل*[2] یک رأس را به عنوان جریان کل مثبت خروجی از یک رأس منهای جریان کل ورودی به آن رأس تعریف می‌کنیم. یک تفسیر از ویژگی بقاء جریان این است که جریان کل مثبت ورودی به یک رأس غیر از منبع یا چاه، باید برابر با جریان کل مثبت خروجی از آن رأس باشد. این ویژگی که جریان خالص کل یک رأس باید برابر

---

1. flow conservation

2. total net flow

صفر باشد اغلب به طور غیر رسمی به صورت " جریان ورودی برابر است با جریان خروجی " به آن اشاره می‌شود.



شکل ۲۶.۱ *(a)* شبکه جریان G برای مسئله حمل و نقل کمپانی *LuckyPuck* کارخانه *Vancouver* منبع s و انبار *Winnipeg* چاه t است. توپ‌ها بین شهرهای واسط حمل می‌شوند اما فقط $c(u,v)$ صندوق در روز می‌تواند از شهر u به شهر v حمل شود. هر یال با ظرفیتش بر چسب گذاری شده است. *(b)* جریان f در G با مقدار $|f|=19$. تنها جریان‌های مثبت نشان داده شده‌اند. اگر $f(u,v)>0$ باشد یال (u,v) به صورت $f(u,v)/c(u,v)$ بر چسب دهی می‌شود. (علامت " / " صرفاً برای جدا کردن جریان و ظرفیت استفاده گردیده و نشان دهنده تقسیم نیست.) اگر $f(u,v)\leq 0$ باشد یال (u,v) فقط با ظرفیتش برچسب دهی می‌شود.)

مثالی از جریان

یک شبکه جریان می‌تواند مسئله حمل و نقل نشان داده شده در شکل *(a)* ۲۶.۱ را مدل سازی کند. کمپانی *Lucky Puck*، یک کارخانه (منبع s) در *Vancouver* دارد که توپ هاکی تولید می‌کند و یک انبار (چاه t) در *Winnipeg* داردکه آنها را انبار می‌کند. کمپانی *Lucky Puck* از یک شرکت دیگر برای حمل توپ‌ها از کارخانه به انبار کامیون اجاره می‌کند. چون کامیون‌ها در مسیرهای معینی (یال‌ها) بین شهرها (رأس‌ها) حرکت می‌کنند و ظرفیت محدودی دارند، *Lucky Puck* می‌تواند حداکثر $c(u,v)$ صندوق در روز بین هر جفت شهر u و v حمل کند که در شکل *(a)* ۲۶.۱ نشان داده شده است. *Lucky Puck* کنترلی روی این مسیر و ظرفیتها ندارد و بنابراین نمی‌تواند جریان شبکه نشان داده شده در شکل *(a)* ۲۶.۱ را تغییر دهد. هدف آنها این است که بیشترین تعداد صندوق p که در یک روز قابل حمل است را تعیین کنند و به همین مقدار تولید داشته باشند، زیرا تولید توپ به میزان بیش از آنچه که در یک روز قابل حمل به انبار است بی‌فایده می‌باشد. *Lucky Puck* به این موضوع توجّهی ندارد که چه مدت طول می‌کشد تا یک توپ از کارخانه به انبار برسد بلکه آنها به دنبال این هستند که p صندوق هر روز از کارخانه خارج شود و p صندوق هر روز به انبار برسد. در ظاهر به نظر مناسب می‌رسد که "جریان" حمل‌ونقل را با یک جریان در این شبکه مدل‌سازی کنیم، زیرا تعداد صندوق‌های حمل شده در یک روز از یک شهر به شهر دیگر تابع محدودیت‌های ظرفیت می‌باشد. به علاوه، بقاء جریان باید رعایت شود و برای یک موقعیت، میزان ورود توپ‌ها به یک شهر واسط باید با میزان خروج از آن شهر برابر باشد

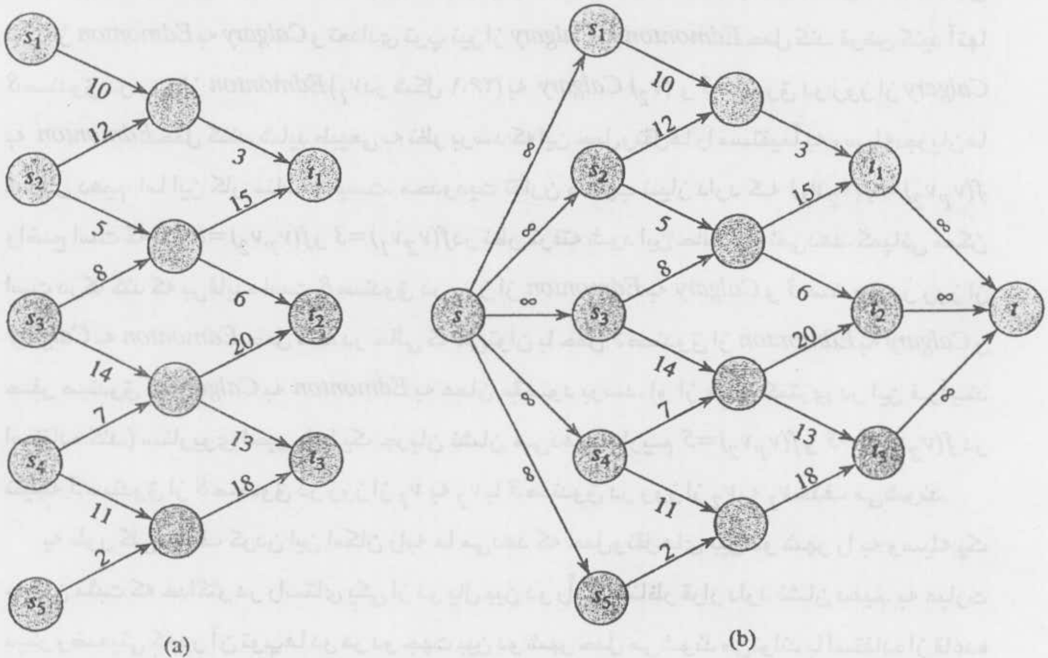
وگرنه صندوق‌ها در شهرهای واسط جمع می‌شوند.

اما یک تفاوت ظریف بین مسئله حمل‌ونقل و جریان‌ها وجود دارد. *Lucky Puck* ممکن است تعدادی توپ از *Edmonton* به *Calgary* و تعدادی توپ نیز از *Calgary* به *Edmonton* حمل کند. فرض کنید آنها 8 صندوق در روز از *Edmonton* (v_1 در شکل ۲۶.۱) به *Calgary* (v_2) و 3 صندوق در روز از *Calgary* به *Edmonton* حمل کنند. شاید طبیعی به نظر برسد که این حمل‌ونقل‌ها را مستقیماً به وسیله جریان‌ها نمایش دهیم، اما این کار مناسب نیست. محدودیت تقارن مورب نیاز دارد که $f(v_1,v_2)=-f(v_2,v_1)$ واضح است که اگر $f(v_1,v_2)=8$ و $f(v_2,v_1)=3$ در نظر گرفته شود این حالت رخ نمی‌دهد. کمپانی ممکن است درک کند که بی‌فایده است 8 صندوق در روز از *Edmonton* به *Calgary* و 3 صندوق در روز از *Calgary* به *Edmonton* حمل کند، در حالی که می‌توان با حمل 5 صندوق از *Edmonton* به *Calgary* و صفر صندوق از *Calgary* به *Edmonton* به همان مقصود برسد. (و از منابع کمتری در این فرآیند استفاده کند.) سناریوی اخیر را با یک جریان نشان می‌دهیم: داریم $f(v_1,v_2)=5$ و $f(v_2,v_1)=-5$ در نتیجه 3 صندوق از 8 صندوق در روز از v_1 به v_2 با 3 صندوق در روز از v_2 به v_1 حذف می‌شوند.

به طور کلی، حذف کردن این امکان رابه ما می‌دهد که حمل‌ونقل‌های بین دو شهر را به وسیله یک جریان مثبت که حداکثر در راستای یکی از دو یال بین دو رأس متناظر قرار دارد نشان دهیم. به عبارت دیگر وضعیتی که در آن توپ‌ها در هر دو جهت بین دو شهر حمل می‌شوند می‌تواند با استفاده از قاعده حذف، تبدیل به یک وضعیت معادل شود که در آن توپ‌ها فقط در یک جهت حمل می‌شوند: در جهت جریان مثبت.

در جریان f که از حمل‌ونقل‌های فیزیکی ناشی می‌شود، نمی‌توانیم میزان دقیق محموله‌ها را بازسازی کنیم. اگر بدانیم $f(u,v)=5$ این جریان ممکن است به علّت حمل 5 واحد از u به v باشد.یا ممکن است از حمل 8 واحد از u به v و حمل 3 واحد از v به u ناشی شود. نوعاً به این موضوع توجّهی نخواهیم کرد که حمل‌ونقل فیزیکی واقعی به چه شکل انجام گرفته است بلکه برای هر جفت رأس تنها به میزان خالص حمل شده بین آن دو رأس می‌پردازیم. اگر بخواهیم به حمل‌ونقل‌های اساسی انجام گرفته نیز توجّه کنیم باید از مدل متفاوتی استفاده کنیم که اطلاعاتی را درباره حمل‌ونقل‌ها در هر دو جهت نگه می‌دارد.

عمل حذف به طور ضمنی در الگوریتم‌های این فصل رخ خواهد داد. فرض کنید یال (u,v) جریانی با مقدار $f(u,v)$ دارد. در طی اجرای یک الگوریتم ممکن است جریان یال (v,u) را به اندازه d افزایش دهیم. از لحاظ ریاضی این عمل باید $f(u,v)$ را به اندازهٔ d کاهش دهد و از نظر مفهومی می‌توانیم این d واحد را مانند حذف d واحد از جریانی که در یال (u,v) برقرار است در نظر بگیریم.



شکل ۲۶.۲ تبدیل مسئله ماکزیمم جریان با چند منبع و چند چاه به مسئله‌ای با یک منبع و یک چاه. *(a)* شبکه جریان با 5 منبع $S=\{s_1,s_2,s_3,s_4,s_5\}$ و سه چاه $T=\{t_1,t_2,t_3\}$. *(b)* شبکه جریان معادل با یک منبع و یک چاه. فوق منبع s و یک یال با ظرفیت نامحدود از s به هر یک از منابع اضافه می‌کنیم. همچنین فوق چاه t و یک یال با ظرفیت نامحدود از هر چاه به t اضافه می‌کنیم.

شبکه با چند منبع و چاه

مسئله ماکزیمم جریان ممکن است به جای یک منبع و یک چاه، چندین منبع و چاه داشته باشد. برای مثال، کمپانی *Lucky Puck* واقعاً ممکن است یک مجموعه از m کارخانه $\{s_1,s_2,...,s_m\}$ و یک مجموعه از n انبار $\{t_1,t_2,...,t_n\}$ داشته باشد، مانند آن چه که در شکل ۲۶.۲ *(a)* نشان داده شده است. خوشبختانه این مسئله سخت‌تر از مسئله معمول ماکزیمم جریان نیست.

می‌توانیم مسئله تعیین ماکزیمم جریان یک شبکه با چند منبع و چاه را به یک مسئله معمول ماکزیمم جریان تبدیل کنیم. شکل ۲۶.۲ *(b)* نشان می‌دهد که چگونه شبکه قسمت *(a)* می‌تواند به یک شبکه جریان معمولی با تنها یک منبع و یک چاه تبدیل شود. یک فوق منبع[1] s و همچنین برای $i=1,2,..,m$ یک یال جهت‌دار (s,s_i) با ظرفیت $c(s,s_i)=\infty$ اضافه می‌کنیم. همچنین یک فوق چاه[2] t جدید ایجاد کرده و برای هر $i=1,2,..,n$ یک یال جهت‌دار (t_i,t) با ظرفیت $c(t_i,t)=\infty$ اضافه می‌کنیم. به طور

1.supersource   2. supersink

شهودی هر جریان در شبکه (a) با یک جریان در شبکه (b) متناظر است و بالعکس. تک منبع s به سادگی همان میزان جریان که منابع s_i تولید می‌کردند را ایجاد می‌کند و تک چاه t، به طور مشابه همان میزان جریانی را که چاه‌های t_i مصرف می‌کردند، مصرف می‌کند. تمرین ۳- ۲۶.۱ از شما می‌خواهد ثابت کنید که این دو مسئله معادلند.

کار با جریان‌ها

به توابع متعددی (مانند f) که به عنوان آرگومان دو رأس شبکه جریان را می‌گیرند، می‌رسیم. در این فصل از *نماد جمع ضمنی* که در آن یک یا هر دو آرگومان ممکن است مجموعه‌ای از رأس‌ها باشند استفاده می‌کنیم، با این تفسیر که مقداری که به آن دلالت می‌شود مجموع همه راه‌های ممکن برای جایگزینی آرگومان‌ها با اعضایشان می‌باشد. برای مثال اگر X و Y مجموعه هایی از رأس‌ها باشند آنگاه

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y) .$$

بنابراین محدودیت بقاء جریان می‌تواند به صورت محدودیت $f(u,V)=0$ برای همه $u \in V\text{-}\{s,t\}$ بیان شود. همچنین برای سهولت نوعاً هنگام استفاده از مجموعه در نماد جمع ضمنی، علامت {} آن مجموعه را حذف می‌کنیم. برای مثال در معادله $f(s,V\text{-}s)=f(s,V)$ عبارت $V\text{-}s$ به معنی مجموعه $V\text{-}\{s\}$ می‌باشد.

نماد ضمنی مجموعه اغلب معادلاتی که شامل جریان‌ها می‌باشند را ساده می‌کند. لم زیر که اثبات آن به عنوان تمرین ۴- ۲۶.۱ واگذار گردیده است، تعدادی از معمول‌ترین مشخصه‌ها که شامل جریان‌ها و نماد ضمنی مجموعه هستند را در برمی‌گیرد.

لم ۲۶.۱

فرض کنید $G=(V,E)$ یک شبکه جریان و f یک جریان در G باشد. آنگاه تساوی‌های زیر برقرارند:

۱. برای همه $X \subseteq V$ داریم $f(X,X) = 0$

۲. برای همه $X, Y \subseteq V$ داریم $f(X,Y)=-f(Y,X)$

۳. برای همه $X, Y, Z \subseteq V$ با شرط $X \cap Y=\phi$ جمعهای زیر را داریم:

$$f(Z,X \cup Y)=f(Z,X)+f(Z,Y) \quad \text{و} \quad f(X \cup Y,Z)=f(X,Z)+f(Y,Z)$$

■

به عنوان مثالی از کار کردن با نماد جمع ضمنی، می‌توانیم ثابت کنیم که مقدار یک جریان، مجموع کل جریان ورودی به چاه است. به عبارت دیگر

$$|f| = f(V, t) \qquad (۲۶.۳)$$

به طور شهودی انتظار داریم که این ویژگی برقرار باشد. بنا به بقاء جریان، همه رأس‌ها غیر از منبع و چاه مقادیر کل جریان مثبت ورودی و کل جریان مثبت خروجی برابری دارند. بنا به تعریف، منبع یک جریان خالص کل دارد که بزرگ‌تر از صفر است؛ به عبارت دیگر جریان مثبت خروجی از منبع بیشتر از جریان مثبت ورودی به آن است. به شکل متقارن، چاه تنها رأسی است که می‌تواند جریان خالص کل کوچکتر از صفر داشته باشد. به عبارت دیگر جریان مثبت ورودی به چاه بیشتر از جریان مثبت خروجی از آن است. اثبات عبارت فوق به شکل زیر است:

$$\begin{aligned}
|f| &= f(s, V) && \text{(بنا به تعریف)} \\
&= f(V, V) - f(V - s, V) && \text{(بنا به لم ۲۶.۱، قسمت (۳))} \\
&= -f(V - s, V) && \text{(بنا به لم ۲۶.۱، قسمت (۱))} \\
&= f(V, V - s) && \text{(بنا به لم ۲۶.۱، قسمت (۲))} \\
&= f(V, t) + f(V, V - s - t) && \text{(بنا به لم ۲۶.۱، قسمت (۳))} \\
&= f(V, t) && \text{(بنا به بقاء جریان)}
\end{aligned}$$

بعدها در این فصل این نتیجه را تعمیم می‌دهیم (لم ۲۶.۵).

تمرین‌ها

۲۶.۱ -۱ با استفاده از تعریف جریان ثابت کنید اگر $(u,v) \notin E$ و $(v,u) \notin E$ آنگاه $f(u,v)=f(v,u)=0$

۲۶.۱ -۲ ثابت کنید برای هر رأس v غیر از منبع و چاه، کل جریان مثبت ورودی به رأس v باید برابر کل جریان مثبت خروجی از v باشد.

۲۶.۱ -۳ ویژگی‌ها و تعاریف جریان را برای مسایل با چند منبع و چند چاه تعمیم دهید. نشان دهید جریان در یک شبکه جریان با چند منبع و چند چاه متناظر است با یک جریان با مقدار مشخص در شبکه‌ای با یک منبع و یک چاه، که به وسیله افزودن یک فوق منبع و یک فوق چاه به دست می‌آید، و بالعکس.

۲۶.۱ -۴ لم ۲۶.۱ را ثابت کنید.

۲۶.۱ -۵ برای شبکه جریان $G=(V,E)$ و جریان f نشان داده شده در شکل (b) ۲۶.۱، یک جفت زیر مجموعه $X,Y \subseteq V$ بیابید که برای آنها $f(X,Y)=-f(V-X,Y)$ سپس یک جفت زیر مجموعه $X,Y \subseteq V$ بیابید که برای آنها $f(X,Y) \neq -f(V-X,Y)$

۲۶.۱ -۶ در شبکه جریان $G=(V,E)$ فرض کنید f_1 و f_2 توابعی از $V \times V$ به R باشند. مجموع جریان f_1+f_2 تابعی از $V \times V$ به R است که برای تمام $u,v \in V$ به این شکل تعریف می‌شود:

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v) \quad (۲۶.۴)$$

اگر f_1 و f_2 جریان‌هایی در G باشند، جریان مجموع f_1+f_2 کدامیک از سه ویژگی جریان را باید حفظ کند و از کدامیک ممکن است تخطی کند؟

۲۶.۱-۷ فرض کنید f جریانی در یک شبکه و α یک عدد حقیقی باشد. *ضرب عددی جریان* که با αf نشان داده می‌شود تابعی از $V\times V$ به R است که به این شکل تعریف می‌شود:

$$(\alpha f)(u, v) = \alpha \cdot f(u, v) .$$

ثابت کنید جریان‌های یک شبکه، یک *مجموعه محدب*[1] را تشکیل می‌دهند. به عبارت دیگر نشان دهید اگر f_1 و f_2 جریان باشند برای تمام مقادیر α در بازه $0\leq\alpha\leq 1$، $\alpha f_1+(1-\alpha)f_2$ نیز جریان می‌باشد.

۲۶.۱-۸ مسئله ماکزیمم - جریان را به صورت یک مسئله برنامه‌سازی خطی بیان کنید.

۲۶.۱-۹ پروفسور *Adam* دارای دو فرزند است که متأسفانه همدیگر را دوست ندارند. مسئله آن قدر جدّی است که نه تنها آنها از قدم زدن با هم در مدرسه امتناع می‌کنند بلکه در واقع هر کدام از قدم‌زدن در مکانی که دیگری در آن روز روی آن گام نهاده امتناع می‌ورزند. بچه‌ها مشکلی با مسیرهایی که از یک گوشه عبور می‌کنند ندارند. خوشبختانه هم منزل پرفسور و هم مدرسه در گوشه‌ها قرار دارند امّا پرفسور مطمئن نیست که آیا ممکن است هر دو فرزند را به یک مدرسه بفرستد یا نه. پرفسور نقشه‌ای از شهر خود دارد. نشان دهید چگونه می‌توان مسئله تعیین اینکه آیا هر دو فرزندش می‌توانند به یک مدرسه بروند را به شکل یک مسئله ماکزیمم جریان درآورد.

۲۶.۲ روش *Ford-Fulkerson*

این بخش روش *Ford-Fulkerson* برای حل مسئله ماکزیمم جریان را بیان می‌کند. این شیوه را بیشتر یک "روش" می‌خوانیم تا یک "الگوریتم"، زیرا شامل چندین پیاده‌سازی با زمان‌های اجرای متفاوت می‌باشد. روش *Ford-Fulkerson* بستگی به سه ایده مهم دارد که جدا از روش هستند و مربوط به بسیاری از الگوریتم‌ها و مسایل جریان می‌باشند: شبکه‌های پسماند، مسیرهای تکمیلی و بریدگی‌ها. این ایده‌ها برای قضیّه مهم ماکزیمم - جریان مینیمم - بریدگی (قضیّه ۲۶.۷) ضروری هستند، که مقدار ماکزیمم جریان رابر حسب بریدگی‌های شبکه جریان توصیف می‌کنند. این بخش را با ارائه یک پیاده‌سازی معیّن از روش *Ford-Fulkerson* و تحلیل زمان اجرای آن به پایان می‌بریم.

Ford-Fulkerson روشی تکراری است. با $f(u,v)=0$ برای تمام $u,v\in V$ آغاز می‌کنیم، که یک جریان اولیّه با مقدار صفر ایجاد می‌کند. در هر تکرار، مقدار جریان را با یافتن "مسیر تکمیلی" افزایش می‌دهیم، که به طور ساده می‌توانیم آن را به عنوان مسیری از منبع s به سمت چاه t در نظر بگیریم که در راستای

---

1. convex set

آن جریان بیشتری را می‌توانیم بفرستیم، و سپس جریان را در راستای این مسیر تکمیل می‌کنیم. این فرآیند را تا زمانی که مسیر تکمیلی دیگری یافت نشود تکرار می‌کنیم. قضیّه ماکزیمم - جریان مینیمم - بریدگی نشان خواهد داد که در پایان کار، این فرآیند ماکزیمم جریان را برمی‌گرداند.

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p
3       do augment flow f along p
4  return f
```

شبکه‌های پسماند[1]

به طور شهودی، در یک شبکه جریان و در یک جریان مفروض، شبکه پسماند از یال‌هایی که می‌توانند جریان بیشتری را بپذیرند تشکیل شده است. به طور دقیق‌تر فرض کنید یک شبکه جریان $G=(V,E)$ با منبع s و چاه t داریم. f را به عنوان یک جریان در G فرض کنید و یک جفت رأس $u,v\in V$ را در نظر بگیرید. میزان جریان اضافی که می‌توانیم از u به v قبل از بیشتر شدن از ظرفیّت $c(u,v)$ وارد کنیم، *ظرفیّت پسماند*[2] (u,v) است که به شکل زیر به دست می‌آید:

$$c_f(u, v) = c(u, v) - f(u, v) . \qquad (۲۶.۵)$$

برای مثال اگر $c(u,v)=16$ و $f(u,v)=11$ باشد می‌توانیم قبل از این که از محدودیت ظرفیّت یال (u,v) تجاوز کنیم $c_f(u,v)=5$ واحد به $f(u,v)$ اضافه کنیم. وقتی که جریان $f(u,v)$ منفی است ظرفیّت پسماند $c_f(u,v)$ بزرگ‌تر از ظرفیت $c(u,v)$ می‌باشد. برای مثال اگر $c(u,v)=16$ و $f(u,v)=-4$ باشد آنگاه ظرفیّت پسماند $c_f(u,v)$ برابر 20 است. این وضعیّت را می‌توانیم اینگونه تفسیر کنیم: یک جریان 4 واحدی از v به u وجود دارد که می‌توانیم آن را با انتقال یک جریان 4 واحدی از u به v حذف کنیم. پس از آن می‌توانیم قبل از این که از محدودیت ظرفیّت یال (u,v) تخطی کنیم 16 واحد دیگر از u به v انتقال دهیم. بنابراین قبل از رسیدن به محدودیت ظرفیّت، یک جریان 20 واحدی اضافی انتقال دهیم که با جریان $f(u,v)=-4$ آغاز می‌شود.

در شبکه جریان $G=(V,E)$ و جریان f داده شده، شبکه پسماند G که به وسیله f حاصل می‌گردد، $G_f=(V,E_f)$ است که

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\} .$$

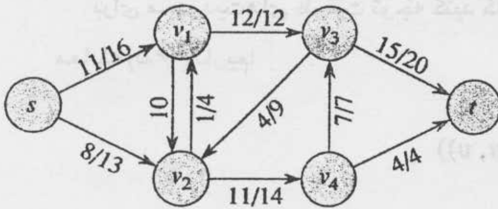
به عبارت دیگر همان طور که قبلاً گفته شد هر یال شبکه پسماند، یا *یال پسماند*، می‌تواند یک جریان بزرگ‌تر از صفر را بپذیرد. شکل ۲۶.۳ (a) شبکه جریان G و جریان f شکل ۲۶.۱ (a) را تکرار می‌کند، و

---

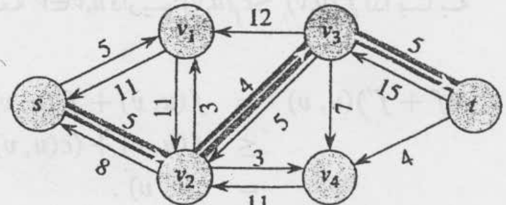
1. residual network
2. residual capacity

شکل (b) ۲۶.۳ شبکه پسماند G_f متناظر را نشان می‌دهد.

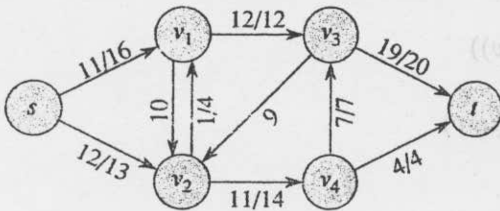
یال‌های درون E_f، یال‌های E یا معکوس آنها می‌باشند. اگر برای یال $(u,v) \in E$، $f(u,v) < c(u,v)$ باشد آنگاه $c_f(u,v) = c(u,v) - f(u,v) > 0$ و $(u,v) \in E_f$. اگر برای یال $(u,v) \in E$، $f(u,v) > 0$ باشد آنگاه $f(v,u) < 0$ است. در این حالت $c_f(v,u) = c(v,u) - f(v,u) > 0$ است و بنابراین $(v,u) \in E_f$ است. اگر هیچ کدام از دو یال (v,u) و (u,v) در شبکه اصلی ظاهر نشود آنگاه $c(u,v) = c(v,u) = 0$ و $f(u,v) = f(v,u) = 0$ است (بنا به تمرین ۱ - ۲۶.۱) و $c_f(u,v) = c_f(v,u) = 0$ می‌باشد. نتیجه می‌گیریم که یال (u,v) می‌تواند در یک شبکه پسماند ظاهر شود تنها اگر حداقل یکی از یال‌های (u,v) یا (v,u) در شبکه اصلی وجود داشته باشند و بنابراین $|E_f| \le 2|E|$



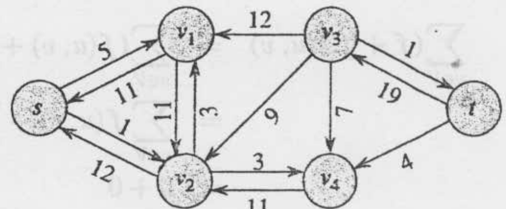
(a)



(b)



(c)



(d)

شکل ۲۶.۳ (a) شبکه جریان G و جریان f شکل (b) ۲٦.١ . (b) شبکه پسماند G_p با مسیر تکمیلی سایه‌زده شده p؛ ظرفیت پسماند آن برابر است با $c_f(p) = c(v_2, v_3) = 4$ (c) جریان درون G که از تکمیل جریان در راستای p به وسیله ظرفیت پسماند 5 حاصل می‌گردد. (d) شبکه پسماند که به وسیله جریان واقع در قسمت (c) ایجاد می‌گردد.

مشاهده می‌گردد که شبکه پسماند G_f، خود یک شبکه جریان با ظرفیت‌های c_f است. لم بعد نشان می‌دهد که چگونه یک جریان در شبکه پسماند با یک جریان در شبکه اصلی نسبت دارد.

لم ۲٦.۲

فرض کنید $G=(V,E)$ یک شبکه جریان با منبع s و چاه t و f یک جریان در G باشد. همچنین فرض کنید G_f یک شبکه پسماند از G که از f حاصل گردیده، و f' یک جریان در G_f باشد. آنگاه مجموع جریان $f+f'$

که به وسیله معادله (۲۶.۴) تعریف می‌گردد یک جریان در G با مقدار $|f+f'|=|f|+|f'|$ است.

اثبات باید صحّت این مطلب را تعیین کنیم که ویژگی‌های تقارن مورب، محدودیت ظرفیّت و بقاء جریان رعایت می‌شوند. برای تقارن مورب توجّه کنید که برای تمام $u,v\in V$ داریم:

$$
\begin{aligned}
(f+f')(u,v) &= f(u,v)+f'(u,v) \\
&= -f(v,u)-f'(v,u) \\
&= -(f(v,u)+f'(v,u)) \\
&= -(f+f')(v,u) .
\end{aligned}
$$

برای محدودیت‌های ظرفیّت توجّه کنید که برای همه $u,v\in V$ داریم: $f'(u,v) \le c_f(u,v)$ لذا بنا به معادله (۲۶.۵) داریم:

$$
\begin{aligned}
(f+f')(u,v) &= f(u,v)+f'(u,v) \\
&\le f(u,v)+(c(u,v)-f(u,v)) \\
&= c(u,v) .
\end{aligned}
$$

برای بقاء جریان توجّه داشته باشید که برای همه $u\in V-\{s,t\}$ داریم:

$$
\begin{aligned}
\sum_{v\in V}(f+f')(u,v) &= \sum_{v\in V}(f(u,v)+f'(u,v)) \\
&= \sum_{v\in V} f(u,v)+\sum_{v\in V} f'(u,v) \\
&= 0+0 \\
&= 0 .
\end{aligned}
$$

در نهایت داریم

$$
\begin{aligned}
|f+f'| &= \sum_{v\in V}(f+f')(s,v) \\
&= \sum_{v\in V}(f(s,v)+f'(s,v)) \\
&= \sum_{v\in V} f(s,v)+\sum_{v\in V} f'(s,v) \\
&= |f|+|f'| .
\end{aligned}
$$

■

مسیرهای تکمیلی[1]

در شبکه جریان $G=(V,E)$ و جریان f مفروض، مسیر تکمیلی p یک مسیر ساده از s به t در شبکه پسماند G_f است. بنا به تعریف شبکه پسماند، هر یال (u,v) روی مسیر تکمیلی، جریان اضافی مثبت از u به v را می‌پذیرد، بدون این که از محدودیت ظرفیت یال تخطی کند.

مسیر سایه‌زده شده در شکل (b) ۲۶.۳ یک مسیر تکمیلی است. در مواجهه با شبکه پسماند G_f در شکل به عنوان یک شبکه جریان، می‌توانیم جریان را در راستای هر یال این مسیر تا 4 واحد افزایش دهیم بدون این که از محدودیت ظرفیت تخطی کنیم ؛ زیرا کوچک‌ترین ظرفیّت پسماند روی این مسیر $c_f(v_2,v_3)=4$ است. ماکزیمم مقداری که به هر یال مسیر تکمیلی p می‌توانیم اضافه کنیم را ظرفیّت پسماند p می‌نامیم که به صورت زیر به دست می‌آید:

$c_f(p)=min\{c_f(u,v):$ در p قرار دارد $(u,v)\}$

لم بعد که اثبات آن به عنوان تمرین ۷- ۲۶.۲ واگذار گردیده بحث فوق را به طور دقیق‌تری بررسی می‌کند.

لم ۲۶.۳

فرض کنید $G=(V,E)$ یک شبکه جریان و f یک جریان در G و p یک مسیر تکمیلی در G_f باشد. تابع $f_p:V\times V\to R$ بدین شکل تعریف می‌شود:

$$f_p(u,v)=\begin{cases} c_f(p) & \text{اگر } (u,v) \text{ در } p \text{ قرار داشته باشد} \\ -c_f(p) & \text{اگر } (v,u) \text{ در } p \text{ قرار داشته باشد} \\ 0 & \text{در غیر اینصورت} \end{cases} \qquad (۲۶.۶)$$

آنگاه f_p یک جریان در G_f با مقدار $|f_p|=c_f(p)>0$ است. ■

قضیه فرعی زیر نشان می‌دهد که اگر f_p را به f اضافه کنیم، به جریان دیگری در G دست می‌یابیم که مقدار آن به ماکزیمم نزدیک‌تر است. شکل (c) ۲۶.۳ نتیجه افزودن f_p در شکل (b) ۲۶.۳ به f از شکل (a) ۲۶.۳ را نشان می‌دهد.

قضیه فرعی ۲۶.۴

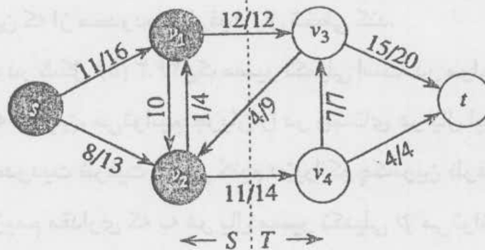
فرض کنید $G=(V,E)$ یک شبکه جریان، f یک جریان در G و p یک مسیر تکمیلی در G_f باشد. فرض کنید f_p به صورت معادله (۲۶.۶) تعریف شود. تابع $f':V\times V\to R$ را به شکل $f'=f+f_p$ تعریف کنید. آنگاه f' یک

---

1. Augmenting paths

جریان در G با مقدار $|f'|=|f|+|f_p|>f$ است.

اثبات مستقیماً از لم‌های ۲۶.۲ و ۲۶.۳ ثابت می‌شود. ■



شکل ۲۶.۴ بریدگی (S,T) در شبکه جریان شکل ۲۶.۱(b)، به طوری که $S=\{s,v_1,v_2\}$ و $T=\{v_3,v_4,t\}$. رأس‌های درون S سیاه و رأس‌های درون T سفید هستند. جریان خالص در راستای (S,T) برابر است با $f(S,T)$ و ظرفیّت برابر است با $c(S,T)=26$

بریدگی‌های شبکه جریان

روش *Ford-Fulkerson* به طور مکرر جریان را در راستای مسیرهای تکمیلی، تکمیل می‌کند تا این که یک ماکزیمم جریان پیدا شود. قضیّه ماکزیمم-جریان مینیمم-بریدگی که آن را به زودی اثبات خواهیم کرد. به ما می‌گوید که یک جریان ماکزیمم است اگر و فقط اگر شبکه پسماند آن شامل مسیر تکمیلی نباشد. اگر چه برای اثبات این قضیّه ابتدا باید مفهوم بریدگی یک شبکه جریان را بررسی کنیم.

بریدگی [1] (S,T) از شبکه جریان $G=(V,E)$، یک افراز از V به داخل S است و $T=V-S$، به طوری که $s\in S$ و $t\in T$ می‌باشد. (این تعریف شبیه تعریف "بریدگی" است که در درخت‌های پوشای مینیمم در فصل ۲۳ از آن استفاده کردیم، به جز این که در اینجا به جای گراف بدون جهت، یک گراف جهت‌دار را برش می‌زنیم و تأکید می‌کنیم که $t\in T, s\in S$ است.) اگر f یک جریان باشد، *جریان خالص* [2] در راستای بریدگی (S,T) به عنوان $f(S,T)$ تعریف می‌شود. *ظرفیّت بریدگی* (S,T)، $c(S,T)$ است. مینیمم بریدگی یک شبکه، یک بریدگی است که ظرفیّت آن نسبت به بقیه بریدگی‌های شبکه مینیمم می‌باشد.

شکل ۲۶.۴ بریدگی $(\{s,v_1,v_2\},\{v_3,v_4,t\})$ در شبکه شکل ۲۶.۱ (b) را نشان می‌دهد. جریان خالص در راستای این بریدگی برابر است با

$$\begin{aligned} f(v_1,v_3)+f(v_2,v_3)+f(v_2,v_4) &= 12+(-4)+11 \\ &= 19\,, \end{aligned}$$

و ظرفیّت آن برابر است با

$$\begin{aligned} c(v_1,v_3)+c(v_2,v_4) &= 12+14 \\ &= 26\,. \end{aligned}$$

1.cut 2.net flow

مشاهده می‌شود که جریان خالص در راستای یک بریدگی می‌تواند شامل جریان‌های منفی بین رأس‌ها باشد، اما ظرفیّت یک بریدگی تماماً از مقادیر غیر منفی تشکیل می‌شود. به بیان دیگر، جریان خالص در راستای بریدگی (S,T) از جریان‌های مثبت در هر دو جهت تشکیل می‌گردد: جریان مثبت از S به T جمع می‌گردد در حالی که جریان مثبت از T به S تفریق می‌شود. در سمت دیگر، ظرفیّت بریدگی (S,T) تنها از یال‌هایی که از S به T می‌روند محاسبه می‌گردد. یال‌هایی که از T به S می‌روند در محاسبه $c(S,T)$ لحاظ نمی‌شوند.

لم بعد نشان می‌دهد که جریان خالص در راستای هر بریدگی یکسان است و برابر است با مقدار جریان.

لم ۲۶.۵

فرض کنید f یک جریان در شبکه جریان G با منبع s و چاه t، و (S,T) یک بریدگی از G باشد. جریان خالص در راستای (S,T) برابر است با $f(S,T)=|f|$

اثبات با توجّه به این که $f(S-s,V)=0$ و بنا به بقاء جریان داریم:

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) && \text{(بنا به لم ۲۶.۱، قسمت (۳))} \\ &= f(S,V) && \text{(بنا به لم ۲۶.۱، قسمت (۱))} \\ &= f(s,V) + f(S-s,V) && \text{(بنا به لم ۲۶.۱، قسمت (۳))} \\ &= f(s,V) && (\text{زیرا } f(S-s,V)=0) \\ &= |f| . \end{aligned}$$

■

قضیه فرعی که مستقیماً از لم ۲۶.۵ حاصل می‌شود، نتیجه‌ای است که ما پیش از این آن را ثابت کردیم - معادله (۲۶.۳) - ، این که مقدار جریان، برابر جریان کل ورودی به چاه است.

قضیه فرعی دیگری که از لم ۲۶.۵ نتیجه می‌شود نشان می‌دهد که چگونه ظرفیّت‌های بریدگی می‌توانند برای محدود کردن مقدار جریان مورد استفاده قرار گیرند.

قضیه فرعی ۲۶.۶

مقدار جریان f در شبکه G از بالا به وسیله ظرفیّت یک بریدگی G محدود می‌شود.

اثبات فرض کنید (S,T) یک بریدگی از G و f یک جریان باشد. بنا به لم ۲۶.۵ و محدودیت‌های ظرفیّت داریم:

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u\in S}\sum_{v\in T} f(u,v) \\ &\le \sum_{u\in S}\sum_{v\in T} c(u,v) \\ &= c(S,T) . \end{aligned}$$

■

یک نتیجه مستقیم قضیه فرعی ۲۶.۶ این است که ماکزیمم جریان در یک شبکه از بالا به وسیله ظرفیت مینیمم - بریدگی شبکه محدود می‌گردد. قضیه مهم ماکزیمم-جریان مینیمم-بریدگی که اینک آن را بیان و اثبات می‌کنیم بیان می‌دارد که مقدار ماکزیمم جریان در حقیقت برابر ظرفیت مینیمم بریدگی است.

قضیه ۲۶.۷ (قضیه ماکزیمم-جریان مینیمم-بریدگی)

اگر f یک جریان در شبکه جریان $G=(V,E)$ با منبع s و چاه t باشد، آنگاه شرایط زیر معادلند:

۱- f یک ماکزیمم جریان در G است.

۲- شبکه پسماند G_f شامل هیچ مسیر تکمیلی نیست.

۳- برای بریدگی (S,T) از G داریم: $|f|=c(S,T)$.

اثبات (۱)⇐(۲): برای ایجاد تناقض فرض کنید f یک ماکزیمم جریان در G باشد اما G_f شامل مسیر تکمیلی p باشد. آنگاه بنا به قضیه فرعی ۲۶.۴ جریان مجموع $f+f_p$، که در آن f_p از معادله (۲۶۶) به دست می‌آید، یک جریان در G با مقدار اکیداً بزرگتر از $|f|$ می‌باشد و با این فرض که f یک ماکزیمم جریان است در تناقض می‌باشد.

(۲)⇐(۳): فرض کنید G_f هیچ مسیر تکمیلی نداشته باشد، به عبارت دیگر G_f شامل هیچ مسیری از s به t نباشد. تعریف می‌کنیم:

$S=\{v\in V$: مسیری از s به v در G_f وجود دارد$\}$

و $T=V-S$. افراز (S,T) یک بریدگی است: به طور بدیهی داریم $s\in S$ و $t\notin S$، زیرا مسیری از s به t در G_f وجود ندارد. برای هر جفت رأس u و v که $u\in S$ و $v\in T$ است داریم $f(u,v)=c(u,v)$ زیرا در غیر این صورت $(u,v)\in E_f$ می‌شود که v را در مجموعه S قرار می‌دهد. لذا بنا به لم ۲۶.۵ داریم

$|f|=f(S,T)=c(S,T)$

(۳)⇐(۱): بنابه قضیه فرعی ۲۶.۶ برای همه بریدگی‌های (S,T) داریم $|f|\le c(S,T)$ بنابراین شرط $|f|=c(S,T)$ بر این دلالت دارد که f یک ماکزیمم جریان است. ∎

الگوریتم اصلی Ford-Fulkerson

در هر تکرار روش Ford-Fulkerson مسیر تکمیلی p را یافته و جریان f روی هر یک از یال‌های p را به اندازهٔ ظرفیّت پسماند $c_f(p)$ افزایش می‌دهیم. پیاده‌سازی روش Ford-Fulkerson به شکل زیر ماکزیمم جریان در گراف $G=(V,E)$ را به وسیله به روزرسانی جریان $f[u,v]$ بین هر جفت رأس u و v که به

وسیله یک یال به هم متصلند محاسبه می‌کند[1]. اگر u و v به وسیله یال در هیچ یک از دو جهت به هم متصل نشده باشند، به طور ضمن فرض می‌کنیم که $f[u,v]=0$ فرض می‌شود که ظرفیّت‌های $c(u,v)$ همراه با گراف داده شوند و اگر $(u,v)\notin E$ باشد $c(u,v)=0$ است. ظرفیّت پسماند $c_f(u,v)$ بر طبق فرمول (۲۶.۵) محاسبه می‌شود. عبارت $c_f(p)$ در کد در حقیقت یک متغیر موقتی است که ظرفیّت پسماند مسیر p را ذخیره می‌کند.

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ E[G]
2      do f[u, v] ← 0
3         f[v, u] ← 0
4  while there exists a path p from s to t in the residual network G_f
5      do c_f(p) ← min {c_f(u, v) : (u, v) is in p}
6         for each edge (u, v) in p
7             do f[u, v] ← f[u, v] + c_f(p)
8                f[v, u] ← − f[u, v]
```

FORD-FULKERSON به سادگی روی شبه کد *FORD-FULKERSON-METHOD* که قبلاً ارائه شد، بسط می‌یابد. شکل ۵-۲۶ نتیجه هر تکرار در یک نمونه اجرای آن را نشان می‌دهد. خطوط ۳-۱ جریان f را با صفر مقدار دهی اولیّه می‌کند. حلقه *While* خطوط ۸-۴ به طور مکرر یک مسیر تکمیلی در G_f را یافته و جریان f یک ماکزیمم جریان است.

تحلیل *Ford-Fulkerson*

زمان اجرای *FORD-FULKERSON* به این بستگی دارد که مسیر تکمیلی p چگونه در خط ۴ تعیین گردد. اگر بد انتخاب شده باشد ممکن است حتی الگوریتم خاتمه هم نیابد: مقدار جریان با تکمیل کردن‌های پیاپی افزایش خواهد یافت، اما لزوماً به مقدار ماکزیمم جریان همگرایی پیدا نمی‌کند. هر چند اگر مسیر تکمیلی با استفاده از جستجوی اول سطح (در بخش ۲۲.۲ ملاحظه کردیم) انتخاب شود الگوریتم با مرتبه زمانی چند جمله‌ای اجرا می‌گردد. اگر چه قبل از اثبات این نتیجه، برای حالتی که در آن مسیر تکمیلی به طور دلخواه انتخاب می‌شود و همه ظرفیّت‌ها اعداد صحیح هستند، یک حد ساده فراهم می‌کنیم.

اغلب در عمل، مسئله ماکزیمم جریان با ظرفیّت‌های صحیح طرح می‌شود. اگر ظرفیّت‌ها اعداد گویا باشند می‌توان از یک تغییر مقیاس مناسب برای صحیح کردن همه آنها استفاده نمود. با این فرض، یک پیاده‌سازی ساده *FORD-FULKERSON* در زمان $O(E|f^*|)$ اجرا می‌شود، که f^* ماکزیم جریان

---

۱ ـ هنگامی که با یک شناسه ـ مانند f ـ به عنوان یک فیلد متغیر برخورد می‌کنیم از براکت‌های مربعی استفاده می‌کنیم، و هنگامی که به عنوان یک تابع با آن برخورد می‌کنیم از پرانتز استفاده می‌کنیم.

پیدا شده توسط الگوریتم می‌باشد. تحلیل بدین شکل انجام می‌گیرد: خطوط ۳-۱ زمان $O(E)$ را صرف می‌کنند. حلقه *While* خطوط ۸-۴ حداکثر $|f^*|$ بار اجرا می‌شود، زیرا مقدار جریان در هر تکرار حداقل به اندازهٔ یک واحدافزایش می‌یابد.

اگر ساختمان داده استفاده شده برای پیاده‌سازی شبکه $G=(V,E)$ را به شکلی کارآمد مدیریت کنیم عملی که در حلقه *While* انجام می‌گیرد می‌تواند مؤثر باشد. فرض کنید ساختمان داده‌ای متناظر با یک گراف جهت‌دار $G'=(V,E')$ نگه می‌داریم، که $E'=\{(u,v):(u,v)\in E$ یا $(v,u)\in E\}$ یال‌های شبکه G یال‌های G' نیز می‌باشند و بنابراین نگه داشتن ظرفیت‌ها و جریان‌ها در این ساختمان داده امر ساده‌ای است. جریان f در gرا در نظر بگیرید، یال‌های شبکه پسماند G_f تشکیل شده‌اند از همه یال‌های (u,v) از G' به طوری که $c(u,v)-f[u,v]\neq 0$. بنابراین اگر از جستجوی اول عمق یا جستجوی اول سطح استفاده کنیم، زمان لازم برای یافتن یک مسیر در شبکه پسماند برابر است با $O(V+E')=O(E)$. لذا هر تکرار حلقه *While* به اندازه $O(E)$ زمان می‌برد، که باعث می‌شود زمان کل اجرای *FORD-FULKERSON* برابر $O(E|f^*|)$ گردد.

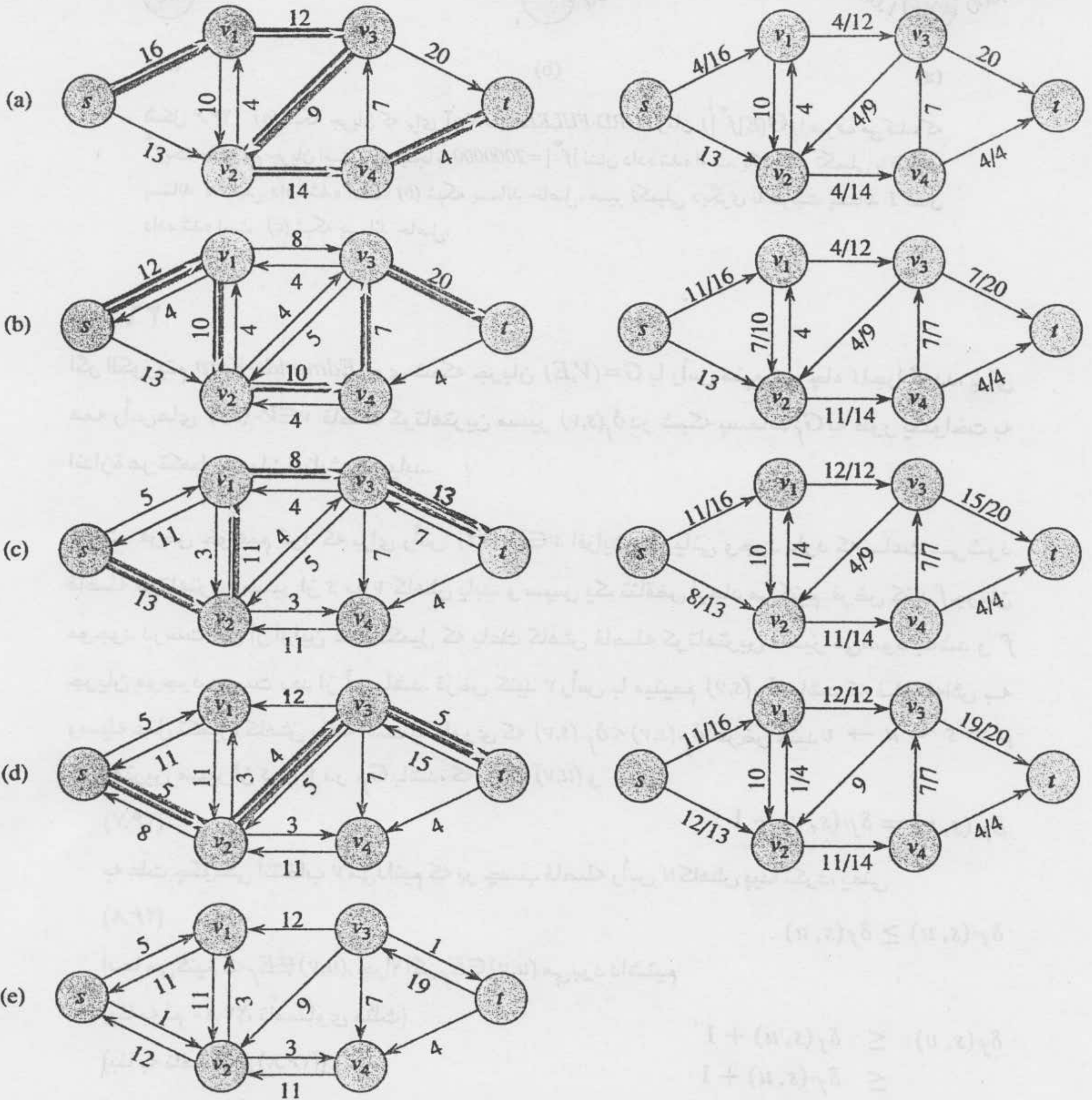
زمانی که ظرفیّت‌ها صحیح هستند و مقدار جریان بهینه $|f^*|$ کوچک است، زمان اجرای الگوریتم *Ford-Fulkerson* مناسب است. شکل ۲۶.۶ *(a)* مثالی ساده را نشان می‌دهد از این که چه اتّفاقی می‌تواند برای یک شبکه جریان ساده با $|f^*|$ بزرگ بیفتد. ماکزیمم جریان در این شبکه دارای مقدار *2000000* است: *1000000* واحد از جریان از مسیر $s\rightarrow u\rightarrow t$ و *1000000* واحد دیگر از مسیر $s\rightarrow v\rightarrow t$ عبور می‌کنند. اگر اوّلین مسیر تکمیلی پیدا شده توسط *FORD-FULKERSON*، $s\rightarrow u\rightarrow v\rightarrow t$ باشد که در شکل ۲۶.۶*(a)* نشان داده شده، جریان مقدار *1* را بعد از اوّلین تکرار حلقه خواهد داشت. شبکه پسماند حاصل در شکل ۲۶.۶*(b)* نشان داده شده است. اگر دومین تکرار، مسیر تکمیلی $s\rightarrow v\rightarrow u\rightarrow t$ را بیابد، مانند آن چه که در شکل ۲۶.۶ *(b)* نشان داده شده، آنگاه جریان دارای مقدار *2* است. شکل ۲۶.۶ *(c)* شبکه پسماند حاصل را نشان می‌دهد. می‌توانیم چنین ادامه دهیم، انتخاب مسیر تکمیلی $s\rightarrow u\rightarrow v\rightarrow t$ در تکرارهای با شماره فرد و انتخاب مسیر تکمیلی $s\rightarrow v\rightarrow u\rightarrow t$ در تکرارهای با شماره زوج. در کل *2000000* عمل تکمیلی را انجام می‌دهیم و در هر بار یک واحد به مقدار جریان اضافه می‌کنیم.

الگوریتم *Edmonds-Karp*

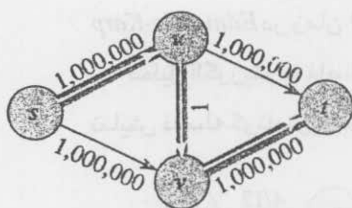
حد *FORD-FULKERSON* می‌تواند اصلاح گردد اگر محاسبه مسیر تکمیلی p در خط ۴ را به وسیله جستجوی اول سطح پیاده‌سازی کنیم، به عبارت دیگر اگر مسیر تکمیلی، کوتاهترین مسیر از s به t در شبکه پسماند باشد که در آن هر یال دارای فاصله (وزن) واحد است. روش *Ford-Fulkerson* که بدین شکل پیاده سازی شود را الگوریتم *Edmonds-Karp* می‌نامیم. اینک ثابت می‌کنیم که الگوریتم

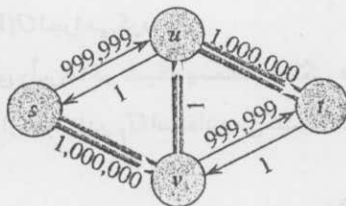
Edmonds-Karp در زمان $O(VE^2)$ اجرا می‌گردد.

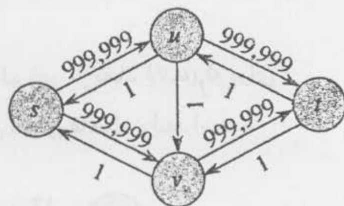
تحلیل الگوریتم به فاصله‌های رأس‌ها در شبکه پسماند بستگی دارد. لم زیر از نماد $\delta_f(u,v)$ برای نمایش فاصله کوتاهترین مسیر از u به v در G_f استفاده می‌کند که هر یال دارای فاصله واحد است.



شکل ۲۶.۵ اجرای الگوریتم اصلی *FORD-FULKERSON*. *(a)-(d)* تکرارهای پیاپی حلقه *While*. سمت چپ هر قسمت، شبکه پسماند G_f خط ۴ را با یک مسیر تکمیلی p سایه‌زده شده نشان می‌دهد. سمت راست هر قسمت، جریان جدید f که از افزودن f_p به f حاصل گردیده رانشان می‌دهد. شبکه پسماند در *(a)*، شبکه ورودی G می‌باشد. *(e)* شبکه پسماند در آخرین بررسی حلقه *while* این شبکه شامل هیچ مسیر تکمیلی نمی‌باشد و بنابراین جریان f نشان داده شده در *(d)*، ماکزیمم جریان است.







شکل ۲۶.۶ *(a)* شبکه جریان که برای آن *FORD-FULKERSON* زمان $\Theta(E|f^*|)$ را صرف می‌کند، که f^* یک ماکزیمم جریان است و در اینجا با $|f^*| = 2000000$ نشان داده شده است. یک مسیر تکمیلی با ظرفیّت پسماند *1* نمایش داده شده است. *(b)* شبکه پسماند حاصل. مسیر تکمیلی دیگری با ظرفیّت پسماند *1* نشان داده شده است. *(c)* شبکه پسماند حاصل.

لم ۲۶.۸

اگر الگوریتم *Edmonds-Karp* روی شبکه جریان $G=(V,E)$ با رأس منبع s و چاه t اجرا گردد، برای همه رأس‌های $v \in V-\{s,t\}$ فاصله کوتاهترین مسیر $\delta_f(s,v)$ در شبکه پسماند G_f به طور یکنواخت به اندازهٔ هر تکمیل جریان افزایش می‌یابد.

اثبات فرض خواهیم کرد که برای رأس $v \in V-\{s,t\}$، افزایش جریانی وجود دارد که باعث می‌شود فاصله کوتاهترین مسیر از s به v کاهش یابد، و سپس یک تناقض ایجاد می‌کنیم. فرض کنید f جریان موجود درست قبل از اولین عمل تکمیل که باعث کاهش فاصله کوتاهترین مسیر می‌شود باشد و f' جریان موجود درست بعد از آن باشد. فرض کنید v رأس با مینیمم $\delta_{f'}(s,v)$ باشد که فاصله‌اش به وسیله عمل تکمیل، کاهش یافته است به طوری که $\delta_{f'}(s,v) < \delta_f(s,v)$. فرض کنید $p = s \leadsto u \to v$ کوتاهترین مسیر از s به v در $G_{f'}$ باشد، که $(u,v) \in E_{f'}$ و

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1 \,. \qquad (۲۶.۷)$$

به علّت چگونگی انتخاب v می‌دانیم که بر چسب فاصله رأس u کاهش پیدا نکرد، یعنی

$$\delta_{f'}(s,u) \ge \delta_f(s,u) \,. \qquad (۲۶.۸)$$

ادعا می‌کنیم که $(u,v) \notin E_f$. چرا؟ اگر $(u,v) \in E_f$ می‌بود داشتیم

$$\begin{aligned} \delta_f(s,v) &\le \delta_f(s,u) + 1 && \text{(بنا به لم ۲۴.۱۰، نامساوی مثلث)} \\ &\le \delta_{f'}(s,u) + 1 && \text{(بنا به نامساوی (۲۶.۸))} \\ &= \delta_{f'}(s,v) && \text{(بنا به تساوی (۲۶.۷))} \end{aligned}$$

که با فرض $\delta_{f'}(s,v) < \delta_f(s,v)$ در تناقض است. چگونه می‌توانیم داشته باشیم $(u,v) \notin E_f$ و $(u,v) \in E_{f'}$؟ پیشبرد جریان باید جریان را از v به u افزایش می‌داد. الگوریتم *Edmonds-Karp* همیشه جریان را در راستای کوتاهترین مسیرها پیش می‌برد، و لذا کوتاهترین مسیر از s به u در G_f شامل (v,u) به عنوان

آخرین یال است. بنابراین

$$\begin{aligned} \delta_f(s,v) &= \delta_f(s,u)-1 \\ &\le \delta_{f'}(s,u)-1 && \text{((بنا به نامساوی(۲۶.۸)} \\ &= \delta_{f'}(s,v)-2 && \text{((بنا به تساوی(۲۶.۷)} \end{aligned}$$

که با فرض $\delta_{f'}(s,v)<\delta_f(s,v)$ در تناقض است.لذا فرض اینکه یک چنین رأس v وجود دارد غلط است. ■

قضیه بعد تعداد تکرارهای الگوریتم *Edmonds-Karp* را محدود می‌کند.

قضیه ۲۶.۹

اگر الگوریتم *Edmonds-Karp* روی شبکه جریان $G=(V,E)$ با منبع s و چاه t اجرا گردد، تعداد کل تکمیل‌های جریان که توسط الگوریتم اجرا می‌گردد برابر است با $O(VE)$.

اثبات می‌گوییم یال (u,v) در شبکه پسماند G_f روی مسیر تکمیلی p *بحرانی* است اگر ظرفیّت پسماند p برابر با ظرفیّت پسماند (u,v) باشد، به عبارت دیگر $c_f(p)=c_f(u,v)$ پس از این که جریان را در راستای مسیر تکمیلی تکمیل کردیم، هر یال بحرانی روی مسیر از شبکه پسماند ناپدید می‌شود. علاوه بر آن، حداقل یک یال روی هر مسیر تکمیلی باید بحرانی باشد. نشان خواهیم داد که هر یک از $|E|$ یال می‌تواند حداکثر $|V|/2-1$ بار بحرانی شود.

فرض کنید u و v رأس‌هایی در V باشند که به وسیله یک یال در E به هم متّصل شده‌اند. چون مسیرهای تکمیلی کوتاهترین مسیرها هستند، زمانی که (u,v) برای اوّلین بار بحرانی است داریم:

$$\delta_f(s,v)=\delta_f(s,u)+1\,.$$

زمانی که جریان تکمیل شد یال (u,v) از شبکه پسماند حذف می‌گردد. این یال نمی‌تواند بعداً روی مسیر تکمیلی دیگری دوباره ظاهر گردد تا زمانی که جریان u به v کاهش یابد، که تنها زمانی اتّفاق می‌افتد که (v,u) روی یک مسیر تکمیلی ظاهر می‌شود. اگر در هنگام به وقوع پیوستن این رویداد، f' جریان در G باشد آنگاه داریم:

$$\delta_{f'}(s,u)=\delta_{f'}(s,v)+1\,.$$

چون بنا به لم ۲۶.۸، $\delta_f(s,v)\le\delta_{f'}(s,v)$ است داریم:

$$\begin{aligned} \delta_{f'}(s,u) &= \delta_{f'}(s,v)+1 \\ &\ge \delta_f(s,v)+1 \\ &= \delta_f(s,u)+2\,. \end{aligned}$$

در نتیجه، زمانی که (u,v) بحرانی می‌شود، تا دفعه بعد که دوباره بحرانی می‌گردد فاصله u از منبع حداقل به اندازهٔ 2 افزایش می‌یابد. در ابتدا فاصله u از منبع حداقل صفر می‌باشد. رأس‌های واسط روی کوتاهترین مسیر از s به u نمی‌توانند شامل s، u یا t باشند (زیرا (u,v) روی مسیر بحرانی دلالت می‌کند بر این که $u\neq t$). بنابراین تا زمانی که u غیر قابل دستیابی از منبع شود فاصله آن حداکثر $|V|-2$ است. لذا (u,v) می‌تواند حداکثر $(|V|-2)/2=|V|/2-1$ بار بحرانی شود. چون $O(E)$ جفت رأس وجود دارند که در گراف پسماند بینشان یال می‌تواند وجود داشته باشد، تعداد کل یال‌های بحرانی در طی

اجرای کل الگوریتم *Edmonds-Karp* برابر $O(VE)$ است. هر تکمیل مسیر حداقل یک یال بحرانی دارد، و از این رو قضیه اثبات می‌شود. ■

چون هر زمانی که مسیر تکمیلی توسط جستجوی اول سطح پیدا می‌شود، هر تکرار روال *FORD-FULKERSON* می‌تواند در زمان $O(E)$ پیاده‌سازی شود. زمان اجرای کل الگوریتم *Edmonds-Karp*، $O(VE^2)$ می‌باشد. خواهیم دید که الگوریتم‌های راندن - برچسب دهی مجدد حتی به حدهای بهتری منجر می‌شوند. الگوریتم بخش ۲۶.۴ روشی برای دست یافتن به زمان اجرای $O(V^2E)$ ارائه می‌کندکه پایه و اساس الگوریتم با مرتبه زمانی $O(V^3)$ بخش ۲۶.۵ را تشکیل می‌دهد.

تمرین‌ها

۲۶.۲-۱ در شکل ۲۶.۱ (b) جریان در راستای بریدگی $(\{s,v_2v_4\},\{v_1v_3t\})$ چقدر است؟ ظرفیّت این بریدگی چقدر است؟

۲۶.۲-۲ اجرای الگوریتم *Edmonds-Karp* روی شبکه جریان شکل ۲۶.۱ (a) را نشان دهید.

۲۶.۲-۳ در مثال شکل ۲۶.۵ مینیمم بریدگی متناظر با ماکزیمم جریان نشان داده شده چیست؟ کدام دو مسیر از مسیرهای تکمیلی ظاهر شده دراین مثال جریان را حذف می‌کنند؟

۲۶.۲-۴ ثابت کنید برای هر جفت رأس u و v و تابع ظرفیّت c و تابع جریان f داریم:

فرمول $c_f(u,v)+c_f(v,u)=c(u,v)+c(v,u).$

۲۶.۲-۵ به خاطر بیاورید که ساختار بخش ۲۶.۱ که یک شبکه جریان چند منبعی، چند چاهی را به یک شبکه تک منبعی تک چاهی تبدیل می‌کند یال‌هایی با ظرفیّت نامحدود اضافه می‌کند. ثابت کنید هر جریان در شبکه حاصل یک مقدار محدود دارد اگر یال‌های شبکه چند منبعی، چند چاهی اوّلیه دارای ظرفیّت محدود باشند.

۲۶.۲-۶ فرض کنید هر منبع s_i در مسئله چند منبعی، چند چاهی دقیقاً p_i واحد جریان تولید می‌کند، به طوری که $f(s_i,V)=p_i$. همچنین فرض کنید هر چاه t_j دقیقاً q_j واحد مصرف می‌کند به طوری که $f(V,t_j)=q_j$ که $\sum_i p_i=\sum_j q_j$. نشان دهید چگونه مسئله یافتن جریان f که از این محدودیت‌های اضافی پیروی می‌کند را به مسئله یافتن ماکزیمم جریان در یک شبکه جریان تک منبعی، تک چاهی تبدیل کنیم.

۲۶.۲-۷ لم ۲۶.۳ را ثابت کنید.

۲۶.۲-۸ نشان دهید ماکزیمم جریان در شبکه $G=(V,E)$ همیشه می‌تواند به وسیله یک توالی از حداکثر $|E|$ مسیر تکمیلی پیدا شود. (راهنمایی: مسیرها را پس از یافتن ماکزیمم جریان تعیین کنید.)

۲۶.۲-۹ همبندی یالی[1] یک گراف بدون جهت، مینیمم تعداد k یال است که باید حذف گردند تا گراف

1.edge connectivity

غیر همبند شود. برای مثال همبندی یالی درخت، یک و همبندی یک زنجیره حلقه‌ای از رأس‌ها، 2 است. نشان دهید چگونه همبندی یالی یک گراف بدون جهت $G=(V,E)$ به وسیله اجرای یک الگوریتم ماکزیمم جریان روی حداکثر $|V|$ شبکه جریان که هر کدام شامل $O(V)$ رأس و $O(E)$ یال می‌باشند می‌تواند تعیین گردد.

۱۰- ۲۶.۲ فرض کنید شبکه جریان $G=(V,E)$ یال‌های متقارن دارد، به عبارت دیگر $(u,v)\in E$ است اگر و فقط اگر $(v,u)\in E$ باشد. نشان دهید الگوریتم *Edmonds-Karp* پس از حداکثر $|V||E|/4$ تکرار خاتمه می‌یابد. (راهنمایی: برای هر یال (u,v) در نظر بگیرید که چگونه $\delta(s,u)$ و $\delta(v,t)$ بین زمان هایی که در آنها (u,v) بحرانی است تغییر می‌کنند.)

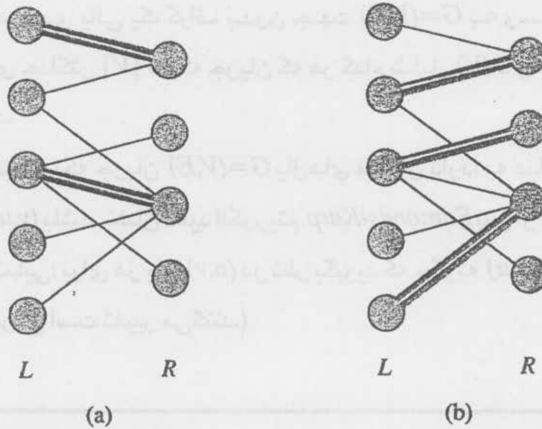
۲۶.۳ تطبیق دو بخشی ماکزیمم[1]

برخی مسایل ترکیبی می‌توانند به سادگی تبدیل به مسایل ماکزیمم جریان شوند. مسئله ماکزیمم جریان چند منبعی چند چاهی بخش ۲۶.۱، نمونه‌ای از آنها است. مسایل ترکیبی دیگری نیز وجود دارند که در نگاه اول به نظر می‌رسد ارتباط ناچیزی با شبکه‌های جریان دارند، اما در واقع می‌توانند به مسایل ماکزیمم جریان تبدیل شوند. این بخش یک چنین مسئله‌ای رابیان می‌کند: یافتن ماکزیمم تطبیق در یک گراف دو بخشی. برای حل این مسئله، از مزیّت ویژگی صحیح بودن که توسط روش *Ford-Fulkerson* تأمین می‌گردد سود خواهیم برد. همچنین خواهیم دید که روش *Ford-Fulkerson* می‌تواند در حل مسئله تطبیق-دوبخشی- ماکزیمم روی گراف $G=(V,E)$ در زمان $O(VE)$ مورد استفاده قرار گیرد.

مسئله تطبیق-دوبخشی- ماکزیمم

در گراف بدون جهت $G=(V,E)$، *تطبیق*، یک زیر مجموعه از یال‌ها $M\subseteq E$ است که برای همه رأس‌های $v\in V$، حداکثر یک یال از M با V برخورد دارد. گوییم رأس $v\in V$ با تطبیق M *مطابقت یافته است*، اگر یک یال در M با v برخورد داشته باشد؛ در غیر این صورت v *مطابقت نیافته است*. *تطبیق ماکزیمم* یک تطبیق با اندازهٔ ماکزیمم است، به عبارت دیگر تطبیق M، به طوری که برای هر تطبیق M' داریم $|M|\geq|M'|$. در این بخش توجّه خود را به یافتن ماکزیمم تطبیق‌ها در گراف‌های دو بخشی معطوف می‌کنیم. فرض می‌کنیم که مجموعه رأس‌ها می‌تواند به $V=L\cup R$ افراز شود، که L و R جدا از هم می‌باشند و همه یال‌های E بین L و R قرار دارند. به علاوه فرض می‌کنیم هر رأس در V حداقل یک یال برخوردی دارد. شکل ۲۶.۷ ایده یک تطبیق را توضیح می‌دهد.

1.maximum bipartite matching



شکل ۲۶.۷ گراف دوبخشی $G=(V,E)$ با افراز $V=L \cup R$. *(a)* یک تطبیق با اندازهٔ 2 *(b)* یک ماکزیمم تطبیق با اندازهٔ 3

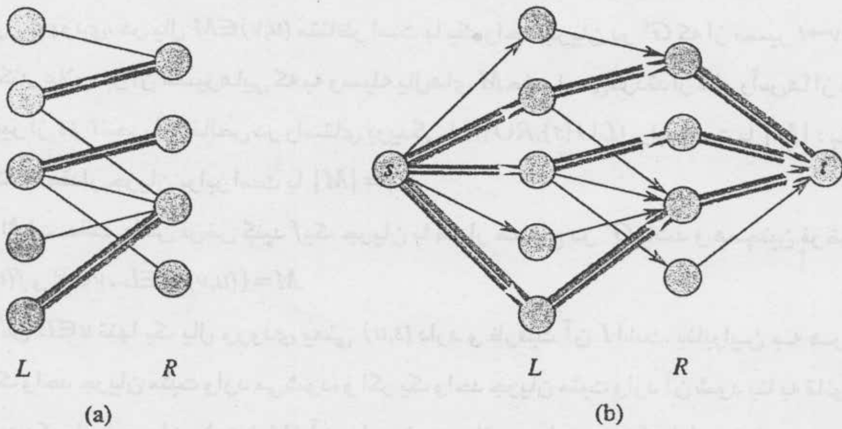
مسئله یافتن ماکزیمم تطبیق در یک گراف دوبخشی کاربردهای عملی زیادی دارد. به عنوان مثال ممکن است تطبیق یک مجموعه L از ماشین‌ها با یک مجموعه R از کارها که باید به طور همزمان اجرا گردند را در نظر بگیریم. حضور یال (u,v) در E بدین معنا است که یک ماشین خاص $u \in L$ قادر به اجرای کار خاص $v \in R$ است. ماکزیمم تطبیق برای بیشترین تعداد ماشین ممکن، کار تأمین می‌کند.

یافتن یک تطبیق دوبخشی ماکزیمم

می‌توانیم از روش *Ford-Fulkerson* برای یافتن تطبیق ماکزیمم در یک گراف دوبخشی بدون جهت $G=(V,E)$ در زمان چند جمله‌ای برحسب $|V|$ و $|E|$، استفاده می‌کنیم. لم این کار ساختن یک شبکه جریان است که در آن جریان‌ها با ماشین‌ها متناظرند، همان طور که در شکل ۲۶۸ نشان داده شده است. شبکه *جریان متناظر* [1] $G'=(V',E')$ برای گراف دو بخشی G را چنین تعریف می‌کنیم: فرض می‌کنیم منبع s و چاه t رأس‌های جدیدی باشند که در V قرار ندارند، و فرض می‌کنیم $V'=V \cup \{s,t\}$. اگر افراز مجموعه رأس‌های G به شکل $V=L \cup R$ باشد یال‌های جهت دار G' یال‌های E هستند که جهت آنها از L به R است، به همراه یال‌های جدید V:

$$\begin{aligned} E' = \; & \{(s,u) : u \in L\} \\ & \cup \{(u,v) : u \in L,\ v \in R,\ \text{and } (u,v) \in E\} \\ & \cup \{(v,t) : v \in R\} \,. \end{aligned}$$

1.corresponding flow network



شکل ۲۶.۸ شبکه جریان متناظر با یک گراف دو بخشی. *(a)* گراف دوبخشی $G=(V,E)$ با افراز $V=L\cup R$ از شکل ۲۶.۷ یک ماکزیمم تطبیق به وسیله یال‌های سایه‌زده شده نشان داده شده است. *(b)* شبکه جریان متناظر G' با نمایش ماکزیمم جریان. هر یال ظرفیّت واحد دارد. یال‌های سایه زده شده جریان 1 را دارند و بقیه یال‌ها جریانی را حمل نمی‌کنند. یال‌های سایه‌زده شده از L به R متناظر با یال‌های ماکزیمم تطبیق گراف دو بخشی هستند.

جهت کامل کردن ساختار به هر یک از یال‌های E' ظرفیّت واحد را انتساب می‌دهیم. چون هر رأس در V حداقل یک یال برخوردی دارد لذا $|E|\geq|V|/2$. بنابراین $|E|\leq|E'|=|E|+|V|\leq 3|E|$ و لذا $|E'|=\Theta(E)$.

لم بعد نشان می‌دهد که یک تطبیق در G مستقیماً با یک جریان در شبکه جریان متناظر با G، یعنی G'، متناظر است. می‌گوییم جریان f در شبکه جریان $G=(V,E)$ *مقداری صحیح* است اگر $f(u,v)$ برای همه $(u,v)\in V\times V$ صحیح باشد.

لم ۲۶.۱۰

فرض کنید $G=(V,E)$ یک گراف دوبخشی با افراز $V=L\cup R$ و $G'=(V',E')$ شبکه جریان متناظر با آن باشد. اگر M یک تطبیق در G باشد آنگاه جریان f با مقدار صحیح در G' وجود دارد که مقدار آن $|f|=|M|$ است. برعکس اگر f یک جریان با مقدار صحیح در G' باشد آنگاه یک تطبیق M در G با اندازهٔ $|M|=|f|$ وجود دارد.

اثبات ابتدا نشان می‌دهیم که تطبیق M در G متناظر با یک جریان با مقدار صحیح در G' است. f را به شکل زیر تعریف می‌کنیم. اگر $(u,v)\in M$ باشد آنگاه $f(s,u)=f(u,v)=f(v,t)=1$ و $f(u,s)=f(v,u)=f(t,v)=-1$. برای همه یال‌های دیگر $(u,v)\in E'$، $f(u,v)=0$ تعریف می‌کنیم. به سادگی

مشخص می‌گردد که f در تقارن مورب، محدودیت‌های ظرفیّت و بقاء جریان صدق می‌کند.

به طور شهودی، هر یال $(u,v) \in M$ متناظر است با یک واحد جریان در G' که از مسیر $s \to u \to v \to t$ عبور می‌کند. علاوه بر آن مسیرهایی که به وسیله یال‌های M حاصل می‌گردند از نظر رأس‌ها از هم جدا هستند، غیر از s و t. جریان خالص در راستای بریدگی $(L \cup \{s\}, R \cup \{t\})$ برابر است با $|M|$؛ بنابراین بنا به لم ۲۶.۵ مقدار جریان برابر است با $|f| = |M|$.

برای اثبات حالت عکس فرض کنید f یک جریان با مقدار صحیح در G' باشد و همچنین فرض کنید $M = \{(u,v): u \in L ، v \in R$ و $f(u,v) > 0\}$

هر رأس $u \in L$ تنها یک یال ورودی یعنی (s,u) دارد و ظرفیت آن 1 است. بنابراین به هر $u \in L$ حداکثر یک واحد جریان مثبت وارد می‌شود، و اگر یک واحد جریان مثبت وارد آن شود بنا به قانون بقاء جریان باید یک واحد جریان مثبت نیز از آن خارج شود. علاوه بر این چون f دارای مقدار صحیح است برای هر رأس $u \in L$ یک واحد جریان می‌تواند حداکثر به یک یال وارد و حداکثر از یک یال خارج شود. بنابراین یک واحد جریان مثبت به u وارد می‌شود اگر و فقط اگر دقیقاً یک رأس $v \in R$ وجود داشته باشد که $f(u,v) = 1$ و حداکثر یک یال از $u \in L$ خارج می‌گردد که جریان مثبت حمل می‌کند. یک بحث متقارن نیز می‌تواند برای هر $v \in R$ انجام گیرد. بنابراین مجموعه M که در بیان لم تعریف شد یک تطبیق است.

برای تحقیق این که $|M| = |f|$ است، مشاهده کنید که برای هر رأس تطبیق یافته $u \in L$ داریم $f(s,u) = 1$، برای هر یال $(u,v) \in E - M$ داریم $f(u,v) = 0$ در نتیجه

$$
\begin{aligned}
|M| &= f(L, R) \\
&= f(L, V') - f(L, L) - f(L, s) - f(L, t) \qquad \text{(بنا به لم ۲۶.۱)}
\end{aligned}
$$

می‌توانیم عبارت فوق را به طور قابل ملاحظه‌ای ساده کنیم. بقاء جریان بیان می‌دارد که $f(L, V') = 0$ است؛ لم ۲۶.۱ بیان می‌دارد که $f(L,L) = 0$ است؛ تقارن مورب بیان می‌دارد که $-f(L,s) = f(s,L)$؛ و چون یالی از L به t وجود ندارد داریم $f(L,t) = 0$ بنابراین

$$
\begin{aligned}
|M| &= f(s, L) \\
&= f(s, V') \qquad \text{(زیرا همه یالهای خروجی از } s \text{ به } L \text{ می‌روند.)} \\
&= |f| \qquad \text{(بنا به تعریف } |f|\text{)}
\end{aligned}
$$

■

براساس لم ۲۶.۱۰ می‌خواهیم نتیجه بگیریم که یک ماکزیمم تطبیق در گراف دوبخشی G متناظر است با ماکزیمم جریان در شبکه جریان متناظرش یعنی G'، و بنابراین می‌توانیم یک ماکزیمم تطبیق در G را به وسیله اجرای الگوریتم جریان روی G' محاسبه کنیم. تنها مشکل این استدلال آن است که الگوریتم ماکزیمم جریان ممکن است جریانی در G' را برگرداند که برای آن $f(u,v)$ عدد صحیحی نباشد، در حالی که مقدار $|f|$ باید صحیح باشد. قضیه زیر نشان می‌دهد که اگر از روش

Ford-Fulkerson استفاده کنیم این مشکل رخ نمی‌دهد.

قضیه ۲۶.۱۱ (قضیه صحیح بودن)

اگر تابع ظرفیّت c تنها مقادیر صحیح را دریافت کند، آنگاه ماکزیمم جریان f که توسط روش *Ford-Fulkerson* تولید می‌گردد این ویژگی را دارا است که $|f|$ یک مقدار صحیح است. علاوه بر این برای همه رأس‌های u و v مقدار $f(u,v)$ صحیح است.

اثبات اثبات به وسیله استقرار روی تعداد تکرارها انجام می‌گیرد. آن را به عنوان تمرین ۲-۲۶.۳ واگذار می‌کنیم. ■

اینک می‌توانیم قضیه فرعی زیر که مربوط به لم ۲۶.۱۰ است را ثابت کنیم.

قضیه فرعی ۲۶.۱۲

اندازه ماکزیمم تطبیق M در گراف دوبخشی G، مقدار ماکزیمم جریان f در شبکه جریان متناظرش یعنی G' است.

اثبات از فهرست اصطلاحات لم ۲۶.۱۰ استفاده می‌کنیم. فرض کنید M یک ماکزیمم تطبیق در G است و فرض کنید جریان متناظر f در G' ماکزیمم نیست. بنابراین یک ماکزیمم جریان f' در G' وجود دارد به طوری که $|f'| > |f|$. چون ظرفیّت‌ها در G' مقادیر صحیحی هستند، بنا به قضیه ۲۶.۱۱ می‌توانیم فرض کنیم که f' مقدار صحیحی است. بنابراین f' متناظر است با یک تطبیق M' در G با اندازهٔ $|M'| = |f'| > |f| = |M|$، که با این فرض که M یک ماکزیمم تطبیق است در تناقض قرار دارد. در حالتی مشابه می‌توانیم نشان دهیم اگر f ماکزیمم جریان در G' باشد، تطبیق متناظر با آن یک ماکزیمم تطبیق در G است. ■

بنابراین در گراف دوبخشی بدون جهت G، می‌توانیم یک ماکزیمم تطبیق به وسیله ایجاد یک شبکه جریان G' بیابیم، روش *Ford-Fulkerson* را اجرا کرده، و مستقیماً به یک ماکزیمم تطبیق M از ماکزیمم جریان صحیح f که پیدا شده دست یابیم. چون هر تطبیق در یک گراف دوبخشی دارای حداکثر اندازه $min(L,R)=O(V)$ است، مقدار ماکزیمم جریان در G' برابر $O(V)$ می‌باشد. بنابراین می‌توانیم ماکزیمم تطبیق در یک گراف دوبخشی را در زمان $O(VE')=O(VE)$ بیابیم، زیرا $|E'|=\Theta(E)$.

تمرین‌ها

۱-۲۶.۳ الگوریتم *Ford-Fulkerson* را روی شبکه جریان شکل ۲۶.۸ (b) اجرا کنید و بعد از هر تکمیل جریان شبکه پسماند را نمایش دهید. رأس‌های L را از بالا به پایین از 1 تا 5 و رأس‌های R را از

بالا به پایین از 6 تا 9 شماره گذاری کنید. برای هر تکرار، مسیر تکمیلی را انتخاب کنید که از نظر لغوی کوچکترین است.

۲۶.۳-۲ قضیه ۲۶.۱۱ را اثبات کنید.

۲۶.۳-۳ فرض کنید $G=(V,E)$ یک گراف دوبخشی با افراز $V=L\cup R$ و G' شبکه جریان متناظر آن باشد. یک حد بالای مناسب برای هر طول مسیر تکمیلی پیدا شده در G' در طی اجرای FORD-FULKERSON ارائه دهید.

۲۶.۳-۴ * یک *تطبیق کامل*[1]، تطبیقی است که در آن هر رأس مطابقت پیدا می‌کند. فرض کنید $G=(V,E)$ یک گراف دوبخشی بدون جهت با افراز $V=L\cup R$ باشد که $|L|=|R|$. برای هر $X\subseteq V$ همسایه X را به شکل زیر تعریف کنید.

$N(X)=\{y\in V:\ (x,y)\in E$, $x\in X$ برای یک$\}$

به عبارت دیگر مجموعه رأس‌های مجاور با عضوی از X قضیه *Hall* را اثبات کنید: یک تطبیق کامل در G وجود دارد اگر و فقط اگر برای هر زیر مجموعه $A\subseteq L$ داشته باشیم $|A|\le|N(A)|$.

۲۶.۳-۵ * می‌گوییم گراف دو بخشی $G=(V,E)$ که $V=L\cup R$، d - منتظم[2] نباشد اگر دقیقاً هر رأس $v\in V$ دارای درجه d باشد. هر گراف دوبخشی d - منتظم دارای $|L|=|R|$ است. ثابت کنید که هر گراف دوبخشی d- منتظم یک تطبیق با اندازهٔ $|L|$ دارد با توجّه به این موضوع که مینیمم بریدگی شبکه جریان متناظر، دارای ظرفیّت $|L|$ است.

* ۲۶.۴ الگوریتم‌های راندن - برچسب‌دهی‌مجدد[3]

در این بخش روش "راندن - برچسب‌دهی‌مجدد" برای محاسبه ماکزیمم جریان‌ها را مطرح می‌کنیم. تا امروز، بسیاری از سریع‌ترین الگوریتم‌های ماکزیمم جریان از لحاظ مجانبی، الگوریتم‌های راندن - برچسب‌دهی‌مجدد بوده‌اند، و سریع‌ترین پیاده‌سازی‌های الگوریتم‌های ماکزیمم جریان بر مبنای روش راندن - برچسب‌دهی‌مجدد بوده‌اند. دیگر مسایل جریان مانند مسئله جریان مینیمم هزینه می‌توانند توسط روش‌های راندن - برچسب‌دهی‌مجدد به طور کارآمد حل گردند. این بخش الگوریتم ماکزیمم جریان "ژنریک " *Goldberg* را معرّفی می‌کند، که دارای یک پیاده‌سازی ساده با زمان اجرای $O(V^2E)$ است، که به موجب آن حد $O(VE^2)$ الگوریتم *Edmonds-Karp* بهبود می‌یابد. بخش ۲۶.۵ الگوریتم ژنریک را برای دستیابی به الگوریتم دیگری از نوع راندن - برچسب دهی مجدد که در زمان $O(V^3)$ اجرا می‌گردد اصلاح می‌کند.

الگوریتم‌های راندن - برچسب دهی مجدد در یک حالت محلی‌تر از روش *Ford-Fulkerson* کار

---

1. perfect matching
2. d-regular
3. push-relabel algorithms

می‌کنند. به جای امتحان کردن کل شبکه پسماند $G=(V,E)$ برای یافتن یک مسیر تکمیلی، الگوریتم‌های راندن – برچسب دهی مجدد در هر لحظه روی یک رأس کار می‌کنند و تنها به همسایه‌های آن رأس در شبکه پسماند نگاه می‌کنند. علاوه بر آن برخلاف روش *Ford-Fulkerson* الگوریتم‌های راندن – برچسب دهی مجدد در طول اجرای خود ویژگی بقاء جریان را حفظ نمی‌کند. اگر چه این الگوریتم‌ها یک *جریان ماقبل*[1] را نگه می‌دارد که تابع $f:V\times V \rightarrow R$ است که در ویژگی‌های زیر صدق می‌کند: تقارن مورب، محدودیت‌های ظرفیّت، و آرام‌سازی بقاء جریان به شکل مقابل: برای همه رأس‌های $u\in V\text{-}\{s\}$ $f(V,u)\geq 0$ به عبارت دیگر جریان خالص کل در هر رأس غیر از منبع غیر منفی است. جریان خالص کل در رأس u را *جریان اضافی*[2] به داخل u می‌نامیم، که به شکل زیر به دست می‌آید:

$$e(u)=f(V,u) \qquad (۲۶.۹)$$

می‌گوییم رأس $u\in V\text{-}\{s,t\}$ *در حال سرریز شدن*[3] است اگر $e(u)>0$ باشد.

این بخش را با توصیف نمودن شهودی که در پس روش راندن – برچسب دهی مجدد قرار دارد آغاز می‌کنیم. سپس دو عملی که این روش استفاده می‌کند را بررسی خواهیم کرد: "راندن" جریان ماقبل و "برچسب دهی مجدد" یک رأس. در نهایت یک الگوریتم راندن – برچسب دهی مجدد را مطرح و صحّت و زمان اجرای آن را تحلیل خواهیم نمود.

شهود

شهود واقع در پس روش راندن – برچسب دهی مجدد بر حسب جریان‌های شارّه به بهترین شکل درک می‌گردد: شبکه جریان $G=(V,E)$ را به عنوان سیستمی از لوله‌های به هم متّصل با ظرفیّت‌های مشخص در نظر می‌گیریم. با به کارگیری این شباهت در روش *Ford-Fulkerson* می‌توان گفت هر مسیر تکمیلی در شبکه باعث ایجاد یک مسیر جریان شارّه اضافی و بدون نقاط انشعاب که از منبع به چاه جریان دارد می‌شود. روش *Ford-Fulkerson* پیاپی و تا وقتی که ممکن باشد مسیرهای جریان را اضافه می‌کند.

الگوریتم ژنریک راندن – برچسب‌دهی‌مجدد شهود نسبتاً متفاوتی دارد. مانند قبل یال‌های جهت‌دار متناظر با لوله‌ها هستند. رأس‌ها که نقاط تقاطع لوله‌ها می‌باشند دارای دو ویژگی جالب هستند. اول این که برای اختصاص جا به جریان اضافی، هر رأس یک لوله جریان خروجی دارد که به یک مخزن بزرگ دلخواه که در آنجا شارّه ذخیره می‌شود، منتهی می‌گردد. دوم این که هر رأس، مخزن آن، و همه اتّصالات لوله‌ای آن روی بستری قرار دارند که ارتفاع آن با پیشرفت اجرای الگوریتم افزایش می‌یابد.

ارتفاع رأس‌ها تعیین می‌کند که جریان چگونه رانده شود: جریان تنها در سرازیری رانده می‌شود، به عبارت دیگر از یک رأس مرتفع‌تر به یک رأس پایین‌تر. جریان از رأس پایین‌تر به رأس بالاتر ممکن

---

1. preflow
2. excess flow
3. overflowing

است مثبت باشد اما اعمالی که جریان را می‌رانند تنها آن را در سرازیری به جریان در می‌آورند. ارتفاع منبع در $|V|$ و ارتفاع چاه در صفر ثابت شده است. ارتفاع رأس‌های دیگر از صفر آغاز می‌شوند و با گذر زمان افزایش می‌یابند. الگوریتم ابتدا هر چقدر جریان را که ممکن باشد از سرازیری منبع به طرف چاه می‌فرستند. مقدار جریانی که می‌فرستد دقیقاً برای پر کردن ظرفیّت همه لوله‌های خروجی از منبع کافی است: به عبارت دیگر الگوریتم به اندازهٔ ظرفیّت بریدگی $(s,V-s)$ جریان می‌فرستد. در ابتدا که جریان وارد یک رأس میانی می‌شود در مخزن رأس جمع می‌گردد و عاقبت از آنجا وارد سرازیری می‌شود. عاقبت ممکن است که تنها لوله هایی که از رأس u خارج می‌شوند و هنوز با جریان اشباع نشده‌اند به رأس هایی متّصل گردند که همسطح با u یا در سطوح بالاتر از u قرار دارند. در این حالت برای رها کردن رأس سرزیر شده u از جریان اضافی‌اش، باید ارتفاع آن را افزایش دهیم - عملی که آن را "برچسب دهی مجدد" رأس u می‌نامیم. ارتفاع آن به اندازه یک واحد بیشتر از ارتفاع پایین‌ترین رأس همسایه‌اش که یک لوله اشباع نشده به آن متّصل است افزایش می‌یابد. لذا بعد از این که رأس برچسب دهی مجدد شد حداقل یک لوله خروجی وجود دارد که در راستای آن می‌توان جریان بیشتری را انتقال داد.

در نهایت همه جریان‌هایی که می‌توانند به سمت چاه حرکت کنند به آنجا رسیده‌اند. جریان بیشتری نمی‌تواند به چاه برسد زیرا لوله‌ها از محدودیت‌های ظرفیّت تبعیّت می‌کنند: میزان جریان در راستای هر بریدگی به وسیله ظرفیّت بریدگی محدود می‌شود. سپس برای تبدیل کردن جریان ماقبل به یک جریان "مجاز"، الگوریتم جریان‌های اضافی جمع شده در مخزن‌های رأس‌های سرریز شده را با ادامه دادن برچسب‌دهی مجدد رأس‌ها به بالاترین ارتفاع ثابت $|V|$ منبع، به منبع برمی‌گرداند. همان طور که خواهیم دید زمانی که همه مخزن‌ها خالی شدند، جریان ماقبل نه تنها یک جریان مجاز است بلکه ماکزیمم جریان نیز می‌باشد.

اعمال اصلی

از بحث قبل دیدیم که دو عمل اصلی وجود دارند که توسط الگوریتم راندن - برچسب دهی مجدد انجام می‌شوند: راندن جریان اضافی از یک رأس به یکی از رأس‌های همسایه‌اش، و برچسب دهی مجدد یک رأس. قابلیّت اجرای این اعمال بستگی به ارتفاع رأس‌ها دارد، که اینک آن را به طور دقیق تعریف می‌کنیم.

فرض کنید $G=(V,E)$ یک شبکه جریان با منبع s و چاه t و f یک جریان ماقبل در G باشد. تابع $h:V\to N$ *تابع ارتفاع*[1] است اگر $h(s)=|V|$، $h(t)=0$ و برای هر یال پسماند $(u,v)\in E_f$

---

۱ - در متون عملی، تابع ارتفاع معمولاً تابع فاصله "نامیده می‌شود و ارتفاع یک رأس،" برچسب فاصله" نامیده می‌شود از اصطلاح "ارتفاع" استفاده می‌کنیم زیرا به شهود واقع در پس الگوریتم بیشتر دلالت می‌کند. از اصطلاح "برچسب‌دهی مجدد" جهت اشاره به عملی که

$h(u) \le h(v)+1$

مستقیماً به لم زیر می‌رسیم.

لم ۲۶.۱۳

فرض کنید G یک شبکه جریان، و f یک جریان ماقبل در G، و h تابع ارتفاع روی V باشد. برای هر دو رأس $u,v \in V$ اگر $h(u) > h(v)+1$ آنگاه (u,v) یالی در گراف پسماند نمی‌باشد.

■

عمل راندن

عمل اصلی $PUSH(u,v)$ می‌تواند استفاده شود اگر u یک رأس سرزیر شده باشد، $c_f(u,v)>0$، و $h(u)=h(v)+1$ شبه کد زیر، جریان ماقبل f در شبکه مفروض $G=(V,E)$ را به روزرسانی می‌کند. فرض می‌شود که ظرفیّت‌های پسماند نیز می‌توانند در زمان ثابت با c و f داده شده محاسبه گردند. جریان اضافی ذخیره شده در رأس u تحت عنوان خصوصیّت $e[u]$ و ارتفاع u تحت عنوان خصوصیّت $h[u]$ نگهداری می‌شوند. عبارت $d_f(u,v)$ یک متغیّر موقتی است که میزان جریانی که می‌تواند از u به v رانده شود را نگه می‌دارد.

```
PUSH(u, v)
1  ▷ Applies when: u is overflowing, c_f(u, v) > 0, and h[u] = h[v] + 1.
2  ▷ Action: Push d_f(u, v) = min(e[u], c_f(u, v)) units of flow from u to v.
3  d_f(u, v) ← min(e[u], c_f(u, v))
4  f[u, v] ← f[u, v] + d_f(u, v)
5  f[v, u] ← −f[u, v]
6  e[u] ← e[u] − d_f(u, v)
7  e[v] ← e[v] + d_f(u, v)
```

فرض می‌شود که رأس u، یک مازاد مثبت $e[u]$ دارد و ظرفیت پسماند (u,v) مثبت است. بنابراین می‌توانیم جریان از u به v را به اندازه $d_f(u,v) = min(e[u], c_f(u,v))$ افزایش دهیم بدون این که باعث منفی شدن $e[u]$ یا تخطی از محدودیت ظرفیت $c(u,v)$ گردد. خط ۳ مقدار $d_f(u,v)$ را محاسبه می‌کند و f را در خطوط ۵-۴ و e را در خطوط ۷-۶ به روزرسانی می‌کند. بنابراین اگر f قبل از فراخوانی $PUSH$ یک جریان ماقبل باشد پس از آن هم یک جریان ماقبل باقی می‌ماند. مشاهده می‌گردد که هیچ چیز در کد $PUSH$ به ارتفاع‌های u و v بستگی ندارد، زیرا از استفاده از آنها جلوگیری می‌کنیم مگر آن

ارتفاع یک رأس را افزایش می‌دهد استفاده می‌کنیم. ارتفاع یک رأس به فاصله‌اش از چاه t بستگی دارد، همان طور که در جستجوی اول سطح ترانهاده G^T مشخص می‌شود.

که $h[u]=h[v]+1$ شود. بنابراین جریان اضافی به سرازیری با تنها اختلاف ارتفاع 1 رانده می‌شود. بنا به لم ۲۶.۱۳ هیچ یال پسماندی بین دو رأسی که اختلاف ارتفاع شان بیشتر از 1 است وجود ندارد، بنابراین چون خاصیّت h در واقع یک تابع ارتفاع است، هیچ نتیجه‌ای از راندن جریان به سرازیری با اختلاف ارتفاع بیش از 1، حاصل نمی‌گردد.

عمل $PUSH(u,v)$ را یک *راندن* از u به v می‌نامیم. می‌گوییم عمل راندن برای رأس u به کار رفته است، اگر عمل راندن برای یال (u,v) که از رأس u خارج می‌شود به کار رود. اگر یال (u,v) *اشباع* شود این عمل یک *راندن اشباع کننده*[1] است، (پس از آن $c_f(u,v)=0$)، و در غیر این صورت یک *راندن غیر اشباع‌کننده*[2] است. اگر یالی اشباع شده باشد در شبکه پسماند ظاهر نمی‌گردد. لم ساده زیر یکی از نتایج راندن غیر اشباع کننده را بیان می‌کند.

لم ۲۶.۱۴

پس از راندن غیر اشباع کننده از u به v، رأس u دیگر در حال سرریز شدن نمی‌باشد.

اثبات چون راندن غیر اشباع کننده بود، مقدار جریان $d_f(u,v)$ که در واقع رانده می‌شود، باید برابر $e[u]$ قبل از عمل راندن باشد. از آن جایی که $e[u]$ به اندازهٔ همین مقدار کاهش می‌یابد لذا پس از راندن برابر صفر می‌گردد. ■

عمل برچسب دهی مجدد

اگر u در حال سرریز شدن باشد و اگر برای همهٔ یال‌های $(u,v)\in E_f$ داشته باشیم $h[u]\le h[v]$ آنگاه عمل اصلی $RELABEL(u)$ مورد استفاده قرار می‌گیرد. به بیان دیگر می‌توانیم رأس سرریز u را برچسب دهی مجدد کنیم، اگر برای هر رأس v که برای آن ظرفیّت پسماند از u به v وجود دارد، جریان به این علّت که v پایین‌تر از u قرار ندارد بتواند از u به v رانده شود (با توجّه به تعریف به خاطر داشته باشید که رأس‌های منبع s و چاه t نمی‌توانند سرریز شوند، و بنابراین s و t برچسب دهی مجدد نمی‌شوند.)

```
RELABEL(u)
1  ▷ Applies when: u is overflowing and for all v ∈ V such that (u, v) ∈ E_f,
        we have h[u] ≤ h[v].
2  ▷ Action: Increase the height of u.
3  h[u] ← 1 + min {h[v] : (u, v) ∈ E_f}
```

---

1. saturating push
2. nonsaturating push

هنگامی که عمل $RELABEL(u)$ را فراخوانی می‌کنیم، می‌گوییم رأس u برچسب دهی مجدد شده است. توجّه داشته باشید هنگامی که u برچسب دهی مجدد می‌شود E_f باید شامل حداقل یک یال خروجی از u باشد، زیرا مینیمم سازی که در کد آمده روی یک مجموعه غیر تهی اجرا می‌شود. این ویژگی از این فرض به دست می‌آید که u سرریز است. چون $e[u] > 0$ داریم $e[u]=f(V,u)>0$ و از این رو باید حداقل یک رأس v وجود داشته باشد به طوری که $f[v,u] > 0$ ولی آن گاه:

$$\begin{aligned} c_f(u,v) &= c(u,v) - f[u,v] \\ &= c(u,v) + f[v,u] \\ &> 0 , \end{aligned}$$

که نشان می‌دهد $(u,v) \in E_f$ است. بنابراین عمل $RELABEL(u)$ بیشترین ارتفاعی که با توجّه به محدودیّت‌های توابع ارتفاع، مجاز می‌باشد را به u نسبت می‌دهد.

الگوریتم ژنریک (عمومی)

الگوریتم راندن - برچسب دهی مجدد از زیر روال زیر برای ایجاد یک جریان ماقبل اولیّه در شبکه جریان استفاده می‌کند.

```
INITIALIZE-PREFLOW(G, s)
 1  for each vertex u ∈ V[G]
 2       do h[u] ← 0
 3          e[u] ← 0
 4  for each edge (u, v) ∈ E[G]
 5       do f[u, v] ← 0
 6          f[v, u] ← 0
 7  h[s] ← |V[G]|
 8  for each vertex u ∈ Adj[s]
 9       do f[s, u] ← c(s, u)
10          f[u, s] ← −c(s, u)
11          e[u] ← c(s, u)
12          e[s] ← e[s] − c(s, u)
```

$INITIALIZE\text{-}PREFLOW$ جریان ماقبل اولیّه f که به شکل زیر تعریف می‌شود را ایجاد می‌کند:

$$f[u,v] = \begin{cases} c(u,v) & \text{اگر } u=s \\ -c(v,u) & \text{اگر } v=s \\ 0 & \text{در غیر اینصورت} \end{cases} \qquad (۲۶.۱۰)$$

به عبارت دیگر هر یالی که از منبع s خارج می‌شود به اندازهٔ ظرفیت پر می‌شود، و همه یال‌های دیگر هیچ جریانی را حمل نمی‌کنند. برای هر رأس v که مجاور منبع است در ابتدا داریم $e[v]=c(s,v)$ و $e[s]$ با مقدار منفی مجموع این ظرفیت‌ها، مقدار دهی اولیّه می‌شود. همچنین الگوریتم ژنریک با تابع ارتفاع اولیّه h که به شکل زیر است شروع می‌شود:

$$h[u] = \begin{cases} |V| & \text{اگر } u=s \\ 0 & \text{در غیر اینصورت} \end{cases}$$

این تابع، یک تابع ارتفاع است زیرا تنها یال‌های (u,v) که برای آنها داریم $h[u]>h[v]+1$ یال‌هایی هستند که برای آنها $u=s$ است، واین یال‌ها اشباع شده‌اند که بدین معنی است که در شبکه پسماند قرار ندارند.

مقدار دهی اولیّه که توسط یک توالی از اعمال راندن و برچسب دهی مجدد دنبال می‌شود، و با نظم خاصی اجرا نمی‌شود و الگوریتم *GENERIC-PUSH-RELABEL* را حاصل می‌کند.

```
GENERIC-PUSH-RELABEL(G)
1  INITIALIZE-PREFLOW(G, s)
2  while there exists an applicable push or relabel operation
3       do select an applicable push or relabel operation and perform it
```

لم زیر به ما می‌گوید از آن جایی که یک رأس سرریز وجود دارد، حداقل یکی از دو عمل اصلی انجام می‌گیرد.

لم ۲۶.۱۵ (یک رأس سرریز می‌تواند رانده شود یا برچسب دهی مجدد گردد.)

فرض کنید $G=(V,E)$ یک شبکه جریان با منبع s و چاه t، f یک جریان ماقبل و h یک تابع ارتفاع برای f باشد .اگر u یک رأس سرریز باشد، یکی از دو عمل راندن یا برچسب دهی مجدد برای آن استفاده می‌شود.

اثبات برای هر یال پسماند (u,v) داریم $h[u]\leq h[v]+1$، زیرا h یک تابع ارتفاع است. اگر عمل راندن در مورد u انجام نگیرد برای همه یال‌های پسماند (u,v) باید داشته باشیم $h(u)<h(v)+1$، که نتیجه می‌دهد $h(u)\leq h(v)$ بنابراین عمل برچسب دهی مجدد می‌تواند در مورد u به کار برده شود. ■

صحّت روش راندن - برچسب دهی مجدد

برای نشان دادن این که الگوریتم راندن - برچسب دهی مجدد ژنریک، مسئله ماکزیمم جریان را حل

می‌کند ابتدا باید ثابت کنیم اگر این الگوریتم خاتمه یابد جریان ماقبل f یک ماکزیمم جریان است. سپس باید ثابت کنیم الگوریتم خاتمه می‌یابد. این کار را با مشاهداتی در خصوص تابع ارتفاع h آغاز می‌کنیم.

لم ۲۶.۱۶ (ارتفاع رأس‌ها هرگز کاهش نمی‌یابد)

در طی اجرای *GENERIC-PUSH-RELABEL* روی شبکه جریان $G=(V,E)$، برای هر رأس $u \in V$ ارتفاع $h[u]$ هرگز کاهش نمی‌یابد. علاوه بر این هر گاه عمل برچسب دهی مجدد رأس u انجام گرفت، ارتفاع $h[u]$ به اندازه حداقل یک واحد افزایش می‌یابد.

اثبات چون ارتفاع رأس‌ها تنها در طی اعمال برچسب دهی مجدد تغییر می‌کند، کافی است قسمت دوم لم را ثابت کنیم. اگر قرار باشد رأس u برچسب دهی مجدد شود، آنگاه برای همه رأس‌های v که $(u,v) \in E_f$ است داریم $h[u] \le h[v]$. بنابراین $h[u] < 1 + min\{h[v]: (u,v) \in E_f\}$، و لذا این عمل باید $h[u]$ را افزایش دهد. ■

لم ۲۶.۱۷

فرض کنید $G=(V,E)$ یک شبکه جریان با منبع s و چاه t باشد در طی *GENERIC-PUSH-RELABEL* روی G، خاصیّت h به عنوان یک تابع ارتفاع حفظ می‌گردد.

اثبات اثبات به وسیله استقرا روی تعداد اعمال اصلی که اجرا می‌شود صورت می‌گیرد. در ابتدا همان طور که قبلاً دیدیم، h یک تابع ارتفاع است.

ادّعا می‌کنیم اگر h یک تابع ارتفاع باشد عمل *RELABEL(u)* هم h را به صورت یک تابع ارتفاع باقی می‌گذارد. اگر یال پسماند $(u,v) \in E_f$ که از u خارج می‌شود را مشاهده کنیم آنگاه عمل *RELABEL(u)* تضمین می‌کند که پس از آن، $h[u] \le h[v]+1$ اینک یال پسماند (w,u) که وارد u می‌شود را در نظر بگیرید. بنا به لم ۲۶.۱۶ داریم $h[w] \le h[u]+1$، قبل از این که عمل *RELABEL(u)* باعث شود که $h[w] < h[u]+1$ شود. بنابراین عمل *RELABEL(u)*، h را یک تابع ارتفاع باقی می‌گذارد.

اینک عمل *PUSH(u,v)* را در نظر بگیرید. این عمل ممکن است یال (u,v) را به E_f اضافه و ممکن است (u,v) را از E_f حذف کند. در حالت اول داریم: $h[v]=h[u]-1<h[u]+1$ و h یک تابع ارتفاع باقی می‌ماند. در حالت بعد، حذف (u,v) از شبکه پسماند، محدودیت متناظر با آن را نیز حذف می‌کند و باز هم h یک تابع ارتفاع باقی می‌ماند.

■

لم زیر یک ویژگی مهم توابع ارتفاع را بیان می‌کند.

لم ۲۶.۱۸

فرض کنید $G=(V,E)$ یک شبکه جریان با منبع s و چاه t و f یک جریان ماقبل در G و h یک تابع ارتفاع روی V باشد. آنگاه هیچ مسیری از منبع s به چاه t در شبکه پسماند G_f وجود ندارد.

اثبات برای ایجاد تناقض فرض کنید مسیر $p=<v_0,v_1,...,v_k>$ از s به t در G_f وجود دارد که در آن $v_0=s$ و $v_k=t$. بدون از دست کلیّت، p یک مسیر ساده است و بنابراین $k<|V|$ می‌باشد. برای $i=0,1,...,k-1$ $(v_i,v_{i+1})\in E_f$ است. چون h یک تابع ارتفاع است برای $i=0,1,...,k-1$ داریم $h(v_i)\le h(v_{i+1})$ ترکیب این نامساوی‌ها روی مسیر p به $h(s)\le h(t)+k$ می‌انجامد. اما چون $h(t)=0$ است داریم $h(s)\le k<|V|$، که با این نیاز که در یک تابع ارتفاع باید $h(s)=|V|$ باشد در تناقض قرار دارد. ■

اینک آماده‌ایم تا نشان دهیم که اگر یک الگوریتم راندن - برچسب دهی مجدد ژنریک خاتمه یابد، جریان ماقبلی که توسط این الگوریتم محاسبه می‌شود یک ماکزیمم جریان است.

قضیه ۲۶.۱۹ (صحّت الگوریتم راندن - برچسب دهی مجدد ژنریک)

اگر الگوریتم *GENERIC-PUSH-RELABEL* با اجرا روی شبکه جریان $G=(V,E)$ با منبع s و چاه t خاتمه یابد، آنگاه جریان ماقبل f که توسط آن محاسبه می‌گردد یک ماکزیمم جریان برای G است.

اثبات از ثابت حلقه زیر استفاده می‌کنیم:

هر بار که تست حلقه *while* در خط ۲ روال *GENERIC-PUSH-RELABEL* اجرا می‌گردد f یک جریان ماقبل است.

مقدار دهی اولیه: *INITIALIZE-PREFLOW* جریان ماقبل f را تولید می‌کند.

نگهداری: تنها اعمال موجود در حلقه *while* خطوط ۳-۲، راندن و برچسب دهی مجدد هستند. اعمال برچسب دهی مجدد تنها بر خصوصیّات ارتفاع تأثیر می‌گذارند نه بر مقادیر جریان؛ از این رو این اعمال اگر f یک جریان ماقبل باشد تأثیری بر آن نمی‌گذارند. همان طور که در صفحات قبل بیان شد اگر f قبل از عمل راندن، یک جریان ماقبل باشد پس از آن هم یک جریان ماقبل باقی می‌ماند.

خاتمه: در پایان، هر رأس در مجموعه $V-\{s,t\}$ باید جریان اضافی 0 را داشته باشد، زیرا بنا به لم‌های ۲۶.۱۵ و ۲۶.۱۷ واین قاعده ثابت که f همیشه یک جریان ماقبل است، هیچ رأس سرریزی وجود ندارد. بنابراین f یک جریان است. چون h یک تابع ارتفاع است لم ۲۶.۱۸ به ما می‌گوید هیچ مسیری از s به t در شبکه جریان پسماند G_f وجود ندارد. لذا بنا به قضیه ماکزیمم-جریان‌مینیمم-بریدگی (قضیه ۲۶.۷)، f یک ماکزیمم جریان است. ■

تحلیل روش راندن – برچسب‌دهی مجدد

برای نشان دادن این که الگوریتم راندن – برچسب‌دهی مجدد ژنریک حقیقتاً خاتمه می‌یابد، تعداد اعمالی که این الگوریتم اجرا می‌کند را محدود می‌کنیم. هر یک از سه نوع عمل – برچسب دهی مجدد، راندن‌های اشباع کننده و راندن‌های غیر اشباع کننده – به طور جداگانه محدود می‌شوند. با دانستن این حدود ساختن الگوریتمی که در زمان $O(V^2E)$ اجرا می‌شود مسئله ساده‌ای است. اما قبل از آغاز تحلیل لم مهمی را ثابت می‌کنیم.

لم ۲۶.۲۰

فرض کنید $G=(V,E)$ یک شبکه جریان با منبع s و چاه t و f یک جریان ماقبل در G باشد. آنگاه برای هر رأس سرریز u، یک مسیر ساده از u به s در شبکه پسماند G_f وجود دارد.

اثبات برای یک رأس سرزیر u قرار دهید { یک مسیر ساده از u به v در G_f وجود دارد:$U=\{v$ و برای ایجاد تناقض فرض کنید $s\notin U$ قرار دهید $\overline{U}=V-U$.

برای هر جفت رأس $w\in\overline{U}$ و $v\in U$ ادّعا می‌کنیم $f(w,v)\leq 0$. چرا؟ اگر $f(w,v)>0$ باشد آنگاه $f(v,w)<0$ است که به نوبه خود بیان می‌کند $c(v,w)-f(v,w)>0$ است. از این رو $(v,w)\in E_f$ وجود دارد و لذا یک مسیر ساده بشکل $u \rightsquigarrow v \rightarrow w$ در G_f موجود است که با انتخابمان از w در تناقض است.

بنابراین باید داشته باشیم $f(\overline{U},U)\leq 0$، چون هر عبارت در این مجموع ضمنی غیر منفی است و از این رو

$e(U)=f(V,U)$	(بنا به تساوی ۲۶.۹)
$=f(\overline{U},U)+f(U,U)$	(بنا به لم ۲۶.۱ قسمت (۳))
$=f(\overline{U},U)$	(بنا به لم ۲۶.۱ قسمت (۱))
≤ 0	

جریان‌های اضافی برای همه رأس‌های واقع در $V-\{s\}$ غیر منفی هستند؛ زیرا فرض کرده‌ایم $U\subseteq V-\{s\}$. بنابراین برای همه رأس‌های $v\in U$ باید داشته باشیم $e(v)=0$. به ویژه $e(u)=0$، که با این فرض که u سرریز است در تناقض قرار دارد. ■

لم بعدی ارتفاع رأس‌ها را محدود می‌کند و قضیه فرعی آن تعداد اعمال برچسب دهی مجدد که در مجموع اجرا می‌شوند را محدود می‌کند.

لم ۲۶.۲۱

فرض کنید $G=(V,E)$ یک شبکه جریان با منبع s و چاه t باشد. در هر لحظه از اجرای روال

GENERIC-PUSH-RELABEL روی G، برای همه رأس‌های $u\in V$ داریم $h[u]\leq 2|V|-1$.

اثبات ارتفاع منبع s و چاه t هرگز تغییر نمی‌کند زیرا این رأس‌ها بنا به تعریف سرریز نمی‌شوند. بنابراین همیشه داریم $h[s]+|V|$ و $h[t]=0$ که هیچ کدام از این دو مقدار بزرگتر از $2|V|-1$ نمی‌باشند.

اینک رأس $u\in V-\{s,t\}$ را در نظر بگیرید. در ابتدا $h[u]=0\leq 2|V|-1$. نشان خواهیم داد که پس از هر عمل برچسب دهی مجدد، باز هم داریم $h[u]\leq 2|V|-1$. هنگامی که u برچسب دهی مجدد می‌شود سرریز است، و لم ۲۶.۲۰ می‌گوید مسیر ساده p از u به s در G_f وجود دارد. فرض کنید $p=<v_0,v_1,...,v_k>$ که $v_0=u$، $v_k=s$ و $k\leq|V|-1$، زیرا p یک مسیر ساده است. برای $i=0,1,...,k-1$ داریم $(v_i,v_{i+1})\in E_f$، و لذا بنا به لم ۲۶.۱۷ داریم $h[v_i]\leq h[v_{i+1}]+1$. با بسط این نامساوی‌ها روی مسیر p خواهیم داشت:

$$h[u]=h[v_0]\leq h[v_k]+k\leq h[s]+(|V|-1)=2|V|-1 \quad \blacksquare$$

قضیه فرعی ۲۶.۲۲ (محدودیت روی اعمال برچسب دهی مجدد)

فرض کنید $G=(V,E)$ یک شبکه جریان با منبع s و چاه t باشد. آنگاه در طی از اجرای روال GENERIC-PUSH-RELABEL روی G، تعداد اعمال برچسب‌دهی مجدد برای هر رأس حداکثر $2|V|-1$ و برای همه رئوس حداکثر $(2|V|-1)(|V|-2)<2|V|^2$ است.

اثبات تنها $|V|-2$ رأس از مجموعه $V-\{s,t\}$ ممکن است برچسب دهی مجدد شوند. فرض کنید $u\in V-\{s,t\}$. عمل $RELABEL(u)$ مقدار $h[u]$ را افزایش می‌دهد. مقدار $h[u]$ در ابتدا صفر است و بنا به لم ۲۶.۲۱ حداکثر به $2|V|-1$ افزایش می‌یابد. بنابراین هر رأس $u\in V-\{s,t\}$ حداکثر $2|V|-1$ بار برچسب‌دهی‌مجدد می‌شود و تعداد کل اعمال برچسب دهی مجدد که انجام می‌گیرد حداکثر $(2|V|-1)(|V|-2)<2|V|^2$ است. ■

لم ۲۶.۲۱ همچنین ما را در محدود کردن تعداد راندن‌های اشباع کننده کمک می‌کند.

لم ۲۶.۲۳ (محدودیت روی راندن‌های اشباع کننده)

در طی اجرای GENERIC-PUSH-RELABEL روی شبکه جریان $G=(V,E)$، تعداد راندن‌های اشباع کننده کمتر از $2|V||E|$ است.

اثبات برای هر جفت رأس $u,v\in V$، راندن‌های اشباع کننده از u به v و از v به u را با هم دیگر خواهیم شمرد، آنها را راندن‌های اشباع کننده بین u و v می‌نامیم. اگر چنین راندن‌هایی وجود داشته باشد حداقل یکی از یال‌های (u,v) و (v,u) حتماً در E قرار دارند. اینک فرض کنید راندن اشباع کننده از u به v

رخ داده باشد. در آن هنگام $h[v]=h[u]-1$. برای این که بعداً راندن دیگری از u به v رخ دهد الگوریتم باید ابتدا جریان را از v به u براند، که این کار نمی‌تواند اتفاق بیفتد تا زمانی که $h[v]=h[u]+1$ شود. چون $h[u]$ هرگز کاهش نمی‌یابد برای این که $h[v]=h[u]+1$ شود، مقدار $h[v]$ باید حداقل به اندازه 2 واحد افزایش یابد. همین طور $h[u]$ نیز باید حداقل به اندازه 2 واحد بین راندن‌های اشباع کننده از v به u افزایش یابد. ارتفاع‌ها از صفر آغاز می‌شوند و بنا به لم ۲۶.۲۱ هرگز بیشتر از $2|V|-1$ نمی‌شوند، که نشان می‌دهد تعداد دفعاتی که یک رأس می‌تواند افزایش ارتفاعی به اندازهٔ 2 داشته باشد، کمتر از $|V|$ است. چون حداقل یکی از دو ارتفاع $h[u]$ و $h[v]$ باید بین هر دو راندن اشباع کننده بین u و v به اندازه 2 واحد افزایش یابد، کمتر از $2|V|$ راندن اشباع کننده بین u و v وجود دارد. با ضرب تعداد یال‌ها دراین مقدار، حدی کمتر از $2|V||E|$ برای تعداد کل راندن‌های اشباع کننده به دست می‌آید. ■

لم بعد تعداد راندن‌های غیر اشباع کننده در الگوریتم راندن - برچسب دهی مجدد ژنریک را محدود می‌کند.

لم ۲۶.۲۴ (محدودیت روی راندن‌های غیر اشباع کننده)

در طی اجرای *GENERIC-PUSH-RELABEL* روی شبکه جریان $G=(V,E)$، تعداد راندن‌های غیر اشباع کننده کمتر از $4|V|^2(|V|+|E|)$ است.

اثبات تابع پتانسیل $\Phi=\sum_{v:e(v)>0} h[v]$ را تعریف می‌کنیم. درابتدا $\Phi=0$ ومقدار Φ ممکن است بعد از هر برچسب‌دهی مجدد، راندن اشباع‌کننده، و راندن غیر اشباع‌کننده تغییر کند. میزان افزایش Φ که راندن‌های اشباع کننده و برچسب‌دهی‌های مجدد می‌توانند در آن سهیم باشند را محدود می‌کنیم. سپس نشان خواهیم داد که هر عمل راندن غیر اشباع کننده باید Φ را حداقل یک واحد کاهش دهد، و از این محدودیّت‌ها برای ایجاد یک حد بالا روی تعداد راندن‌های غیر اشباع کننده استفاده خواهیم کرد.

دو راهی که ممکن است در آنها Φ افزایش یابد را بررسی می‌کنیم. اول، برچسب دهی مجدد رأس u مقدار Φ را به اندازه کمتر از $2|V|$ افزایش می‌دهد، زیرا مجموعه‌ای که روی آن حاصلجمع گرفته می‌شود همان مجموعه قبلی است و برچسب دهی مجدد نمی‌تواند ارتفاع u را به مقدار بیش از ماکزیمم ارتفاع ممکن آن افزایش دهد، که این مقدار بنا به لم ۲۶.۲۱ حداکثر برابر $2|V|-1$ است. دوم، راندن اشباع‌کننده از رأس u به v مقدار Φ را به اندازه کمتر از $2|V|$ افزایش می‌دهد، زیرا هیچ ارتفاعی تغییر نمی‌کند و تنها رأس v که ارتفاعش حداکثر $2|V|-1$ است می‌تواند سرریز شود.

اینک نشان می‌دهیم که راندن غیر اشباع کننده از u به v مقدار Φ را به اندازه حداقل 1 واحد کاهش می‌دهد، چرا؟ قبل از راندن غیر اشباع کننده u سرریز بود، و v ممکن است سرریز شده یا نشده باشد. بنا به لم ۲۶.۱۴، u پس از راندن دیگر سرریز نیست، علاوه بر این v باید پس از عمل راندن سرریز باشد مگر آن که منبع باشد. بنابراین تابع پتانسیل Φ دقیقاً به اندازه $h[u]$ کاهش یافته است، و به اندازه 0 یا $h[v]$

افزایش یافته است. چون $h[u]-h[v]=1$ است، تأثیر خالص این است که تابع پتانسیل حداقل به اندازه 1 کاهش یافته است.

بنابراین در طی اجرای الگوریتم میزان کل افزایش Φ مربوط به برچسب‌دهی‌های مجدد و راندن‌های اشباع کننده می‌باشد و بنا به قضیه فرعی ۲۶.۲۲ و لم ۲۶.۲۳ به مقداری کمتر از $(2|V|)(2|V|^2)+(2|V|)(2|V||E|)=4|V|^2(|V|+|E|)$ محدود می‌شود. چون $\Phi\geq 0$ است میزان کل کاهش و لذا تعداد کل راندن‌های غیر اشباع کننده کمتر از $4|V|^2([V]+|E|)$ است. ■

با محدود کردن تعداد برچسب دهی‌های مجدد، راندن‌های اشباع کننده و غیر اشباع کننده بستر مناسبی را برای تحلیل روال *PUSH-RELABEL* و از این رو هر الگوریتم مبتنی بر روش راندن - برچسب دهی مجدد مهیا کرده‌ایم.

قضیه ۲۶.۲۵

در طی اجرای *GENERIC-PUSH-RELABEL* روی شبکه جریان $G=(V,E)$، تعداد اعمال اصلی برابر $O(V^2E)$ است.

اثبات مستقیماً از قضیه فرعی ۲۶.۲۲ و لم‌های ۲۶.۲۳ و ۲۶.۲۴ اثبات می‌گردد. ■

بنابراین الگوریتم پس از $O(V^2E)$ عمل خاتمه می‌یابد. تنها چیزی که باقی می‌ماند ارائه یک روش کارآمد برای پیاده‌سازی هر عمل و برای انتخاب عمل مناسب جهت اجرا است.

قضیه فرعی ۲۶.۲۶

یک پیاده‌سازی از الگوریتم راندن-برچسب دهی مجدد ژنریک وجود دارد که روی شبکه جریان $G=(V,E)$ در زمان $O(V^2E)$ اجرا می‌گردد.

اثبات تمرین ۱- ۲۶.۴ از شما می‌خواهد الگوریتم ژنریک را طوری پیاده‌سازی کنید که هر عمل برچسب دهی مجدد در حداکثر $O(V)$ و هر عمل راندن در حداکثر $O(1)$ اجرا شود. همچنین از شما می‌خواهد ساختمان داده‌ای طراحی کنید که این امکان را بدهد که یک عمل قابل اجرا را در زمان $O(1)$ انتخاب کنید. از آنجا قضیه فرعی اثبات می‌شود. ■

تمرین‌ها

۱- ۲۶.۴ نشان دهید که الگوریتم راندن-برچسب دهی مجدد ژنریک را چگونه می‌توان پیاده‌سازی کرد که در آن هر عمل برچسب دهی مجدد در زمان $O(V)$، هر عمل راندن در زمان $O(1)$ و انتخاب یک عمل قابل اجرا در زمان $O(1)$ صورت گیرد و زمان کل اجرا $O(V^2E)$ باشد.

۲۶.۴-۲ ثابت کنید الگوریتم راندن-برچسب دهی مجدد ژنریک برای اجرای همه $O(V^2)$ عمل برچسب دهی مجدد زمان کل $O(VE)$ را صرف می‌کند.

۲۶.۴-۳ فرض کنید یک ماکزیمم جریان در شبکه جریان $G=(V,E)$ با استفاده از الگوریتم راندن-برچسب دهی مجدد بدست آمده است. الگوریتمی سریع برای یافتن یک مینیمم بریدگی در G ارائه دهید.

۲۶.۴-۴ یک الگوریتم راندن-برچسب دهی مجدد کارآمد برای یافتن ماکزیمم تطبیق در یک گراف دوبخشی ارائه دهید. الگوریتم خود را تحلیل کنید.

۲۶.۴-۵ فرض کنید ظرفیّت همه رأس‌های موجود در شبکه جریان $G=(V,E)$ در مجموعه $\{1,2,...,k\}$ قرار دارند. زمان اجرای الگوریتم راندن-برچسب دهی مجدد ژنریک را بر حسب $|V|$ ، $|E|$ و k تحلیل کنید. (راهنمایی: هر یال قبل از اشباع شدن، چند عمل راندن غیر اشباع کننده را می‌پذیرد؟)

۲۶.۴-۶ نشان دهید خط ۷ روال *INITIALIZE-PREFLOW* می‌تواند به شکل زیر تغییر کند بدون این که اثری بر صحّت یا کارآیی مجانبی الگوریتم راندن-برچسب دهی مجدد داشته باشد.

$$7 \quad h[s] \leftarrow |V[G]| - 2$$

۲۶.۴-۷ فرض کنید $\delta_f(u,v)$ فاصله u و v در شبکه پسماند G_f باشد. نشان دهید چگونه GENERIC-PUSH-RELABEL می‌تواند تغییر یابد تا این ویژگی‌ها که $h[u]<|V|$ دلالت می‌کند بر $h[u] \leq \delta_f(u,t)$ و $h[u] \geq |V|$ دلالت می‌کند بر $h[u] - |V| \leq \delta_f(u,s)$ حفظ شود.

۲۶.۴-۸ نشان دهید تعداد راندن‌های غیر اشباع کنند که توسط اجرای *GENERIC-PUSH-RELABEL* روی شبکه جریان $G=(V,E)$ صورت می‌گیرد برای $|V|\geq 4$ حداکثر $4|V|^2|E|$ است.

۲۶.۵* الگوریتم برچسب دهی مجدد-به-جلو[1]

روش راندن-برچسب دهی مجدد این امکان را به ما می‌دهد که اعمال اصلی را با هر ترتیبی انجام دهیم. اگر چه با انتخاب دقیق ترتیب و مدیریت کارآمد ساختمان داده شبکه، می‌توانیم مسئله ماکزیمم جریان را سریعتر از حد $O(V^2E)$ که از قضیه فرعی ۲۶.۲۶ نتیجه شده است، حل کنیم. اینک الگوریتم برچسب دهی مجدد به جلو را بررسی می‌کنیم، یک الگوریتم راندن-برچسب دهی مجدد که زمان اجرایش $O(V^3)$ است و از لحاظ مجانبی حداقل به خوبی $O(V^2E)$ می‌باشد و برای شبکه‌های متراکم مناسب‌تر است.

1.relabel-to-front algorithm

الگوریتم برچسب دهی مجدد-به-جلو یک لیست از رأس‌های شبکه را نگه می‌دارد. با آغاز از جلو، الگوریتم لیست را پویش کرده و به طور پی‌درپی رأس سرریز u را انتخاب و آن را تخلیه می‌کند، به عبارت دیگر اعمال راندن و برچسب دهی مجدد را تا وقتی که دیگر u جریان اضافی مثبت نداشته باشد، اجرا می‌کند. هر وقت یک رأس برچسب دهی مجدد می‌شود، به جلوی لیست انتقال می‌یابد (از این رو به این الگوریتم "برچسب دهی مجدد-به-جلو" می‌گویند) و الگوریتم عمل پویش را از نو آغاز می‌کند.

صحّت تحلیل الگوریتم برچسب دهی مجدد-به-جلو بستگی به ایده *یال‌های مجاز* دارد: یالهایی در شبکه پسماند که در راستای آنها جریان می‌تواند رانده شود. پس از اثبات بعضی ویژگی‌های شبکه‌ای از یالهای مجاز، عمل تخلیه را بررسی و سپس خود الگوریتم برچسب دهی مجدد-به-جلو را بیان کرده و تحلیل خواهیم نمود.

یال‌های مجاز و شبکه‌ها

اگر $G=(V,E)$ یک شبکه جریان با منبع s و چاه t و f یک جریان ماقبل در G، و h یک تابع ارتفاع باشد آنگاه می‌گوییم (u,v) یک *یال مجاز*[1] است اگر $c_f(u,v)>0$ و $h(u)=h(v)+1$ در غیر این صورت (u,v) غیر مجاز[2] است. شبکه مجاز[3] $G_{f,h}=(V,E_{f,h})$ است که $E_{f,h}$ مجموعه یال‌های مجاز می‌باشد.

شبکه مجاز تشکیل شده است از یال‌هایی که در طول آنها جریان می‌تواند رانده شود. لم بعد نشان می‌دهد که این شبکه یک گراف جهت‌دار بدون دور *(dag)* است.

لم ۲۶.۲۷ (شبکه مجاز، بدون دور است)

اگر $G=(V,E)$ یک شبکه جریان، f یک جریان ماقبل در G، و h یک تابع ارتفاع روی G باشد آنگاه شبکه مجاز $G_{f,h}=(V,E_{f,h})$ بدون دور است.

اثبات اثبات به وسیله استقرا انجام می‌گیرد. فرض کنید $G_{f,h}$ شامل دور $p=\langle v_0,v_1,\ldots,v_k\rangle$ باشد که $v_0=v_k$ و $K>0$ چون هر یال در p مجاز است برای $i=1,2,\ldots,k$ داریم: $h(v_{i-1})=h(v_i)+1$ از جمع حول این دور چنین به دست می‌آید

$$\sum_{i=1}^{k} h(v_{i-1}) = \sum_{i=1}^{k}(h(v_i)+1) = \sum_{i=1}^{k} h(v_i) + k .$$

چون هر رأس در دور p دقیقاً یک بار در هر یک از جمع‌ها ظاهر می‌شود، به تناقض $k=0$ می‌رسیم.

1. admissible edge
2. inadmissible
3. admissible network

دو لم بعد نشان می‌دهند که چگونه اعمال راندن و برچسب دهی مجدد، شبکه مجاز را تغییر می‌دهند.

لم ۲۶.۲۸

فرض کنید $G=(V,E)$ یک شبکه جریان، f یک جریان ماقبل و خصوصیّت h یک تابع ارتفاع باشد. اگر رأس u سرریز و یال (u,v) مجاز باشد آنگاه $PUSH(u,v)$ انجام می‌شود. این عمل یال‌های مجاز جدیدی را ایجاد نمی‌کند اما ممکن است باعث شود (u,v) غیر مجاز شود.

اثبات بنا به تعریف یال مجاز، جریان می‌تواند از u به v رانده شود. چون u سرریز است عمل $PUSH(u,v)$ صورت می‌پذیرد. تنها یال پسماند جدید که می‌تواند به وسیله رانده شدن جریان ایجاد شود یال (u,v) است. چون $h[v]=h[u]-1$ یال (u,v) نمی‌تواند مجاز شود. اگر این عمل یک راندن اشباع‌کننده باشد، آنگاه پس از آن $c_f(u,v)=0$ و (u,v) غیر مجاز می‌شود. ■

لم ۲۶.۲۹

فرض کنید $G=(V,E)$ یک شبکه جریان، f یک جریان ماقبل در G و خصوصیّت h یک تابع ارتفاع باشد. اگر رأس u سرریز باشد و هیچ یال مجاز خروجی از u وجود نداشته باشد آنگاه $RELABEL(u)$ انجام می‌شود. پس از عمل برچسب دهی مجدد حداقل یک یال مجاز خروجی از u وجود دارد، اما هیچ یال مجاز ورودی به u وجود ندارد.

اثبات اگر u سرریز باشد آنگاه بنا به لم ۲۶.۱۵، عمل راندن یا عمل برچسب دهی مجدد برای آن انجام می‌شود. اگر هیچ یال مجاز خروجی از u وجود نداشته باشد آنگاه هیچ جریانی نمی‌تواند از u رانده شود و بنابراین $RELABEL(u)$ انجام می‌گیرد. پس از عمل برچسب دهی مجدد $h[u]=1+min\{h[v]:(u,v)\in E_f\}$

بنابراین اگر v رأسی است که شامل مقدار مینیمم این مجموعه می‌باشد، آنگاه یال (u,v) مجاز می‌شود. از این رو پس از برچسب دهی مجدد، حداقل یک یال مجاز خروجی از u وجود دارد.

برای نشان دادن اینکه پس از عمل برچسب‌دهی مجدد، هیچ یال مجازی به u وارد نمی‌شود، فرض کنید رأس v وجود دارد به طوری که (v,u) مجاز است. آنگاه پس از برچسب دهی مجدد $h[v]=h[u]+1$ است. اما بنا به لم ۲۶.۱۳، هیچ یال پسماندی بین رأس‌هایی که اختلاف ارتفاعشان بیشتر از 1 است وجود ندارد. علاوه بر آن برچسب دهی مجدد یک رأس، شبکه پسماند را تغییر نمی‌دهد. لذا (v,u) در شبکه پسماند قرار ندارد و از اینرو نمی‌تواند در شبکه مجاز قرار داشته باشد. ■

لیست‌های مجاور[1]

یال‌ها در الگوریتم برچسب دهی مجدد-به-جلو در داخل "لیست‌های مجاور" سازماندهی می‌شوند. در شبکه جریان مفروض $G=(V,E)$، لیست مجاور $N[u]$ برای رأس $u \in V$ یک لیست یک پیوندی از مجاورهای u در G است. بنابراین رأس v در لیست $N[u]$ قرار دارد اگر $(u,v) \in E$ یا $(v,u) \in E$ باشد. لیست مجاور $N[u]$ دقیقاً شامل رأس‌های v است که برای آنها ممکن است یک یال پسماند (u,v) وجود داشته باشد. توسط $head[N[u]]$ به اولین رأس در $N[u]$ اشاره می‌شود. به رأسی که در لیست مجاور پس از v قرار دارد توسط $next\text{-}neighbor[v]$ اشاره می‌شود؛ این اشاره‌گر NIL است اگر v آخرین رأس در لیست مجاور باشد.

الگوریتم برچسب دهی مجدد-به-جلو در راستای هر لیست مجاور، با یک نظم دلخواه که در طول اجرای الگوریتم تعیین می‌گردد، می‌چرخد. برای هر رأس u فیلد $current[u]$ به رأس جاری مورد نظر در $N[u]$ اشاره می‌کند. در ابتدا، $current[u]$ مقدار $head[N[u]]$ را می‌گیرد.

تخلیه یک رأس سرریز

رأس سرریز u به وسیله راندن همه جریان‌های اضافی‌اش در راستای یال‌های مجاز به رأس‌های مجاورش، و برچسب دهی مجدد u که برای تبدیل یال‌های خروجی از u به یال‌های مجاز، ضروری می‌باشد، تخلیه می‌شود. شبه کد به شکل زیر است.

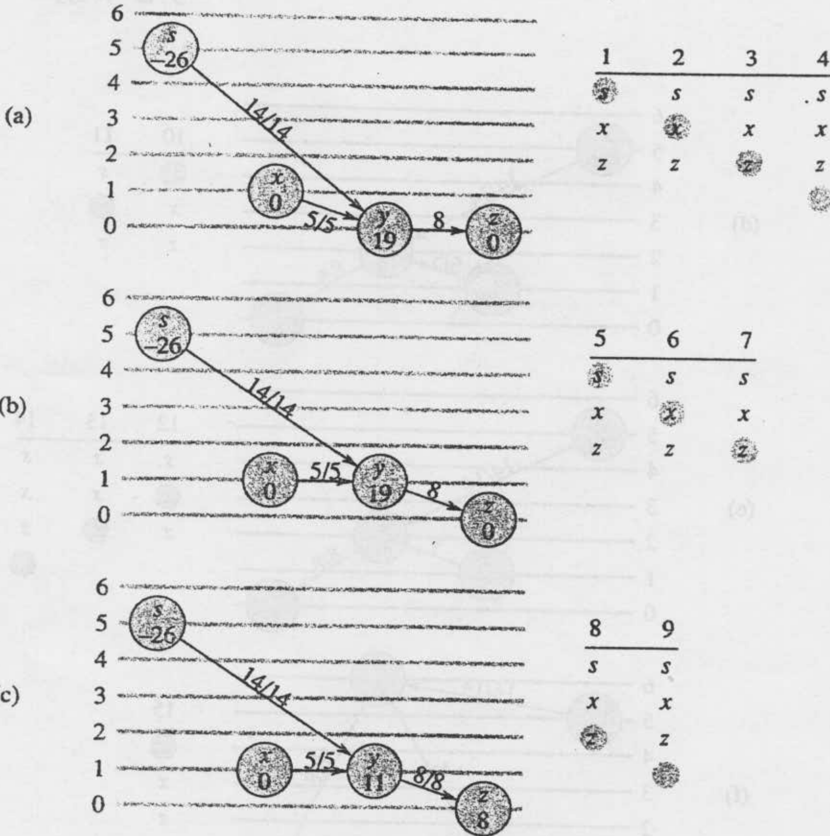
```
DISCHARGE(u)
1  while e[u] > 0
2      do v ← current[u]
3         if v = NIL
4            then RELABEL(u)
5                 current[u] ← head[N[u]]
6         elseif c_f(u, v) > 0 and h[u] = h[v] + 1
7            then PUSH(u, v)
8         else current[u] ← next-neighbor[v]
```

شکل ۲۶.۹ تکرارهای حلقه *while* خطوط ۸-۱، که تا زمانیکه رأس u جریان اضافی مثبت دارد اجرا می‌شود، را به صورت گام به گام نشان می‌دهد. هر تکرار، وابسته به رأس جاری v در لیست مجاور $N[u]$ دقیقاً یکی از سه عمل زیر را اجرا می‌کند.

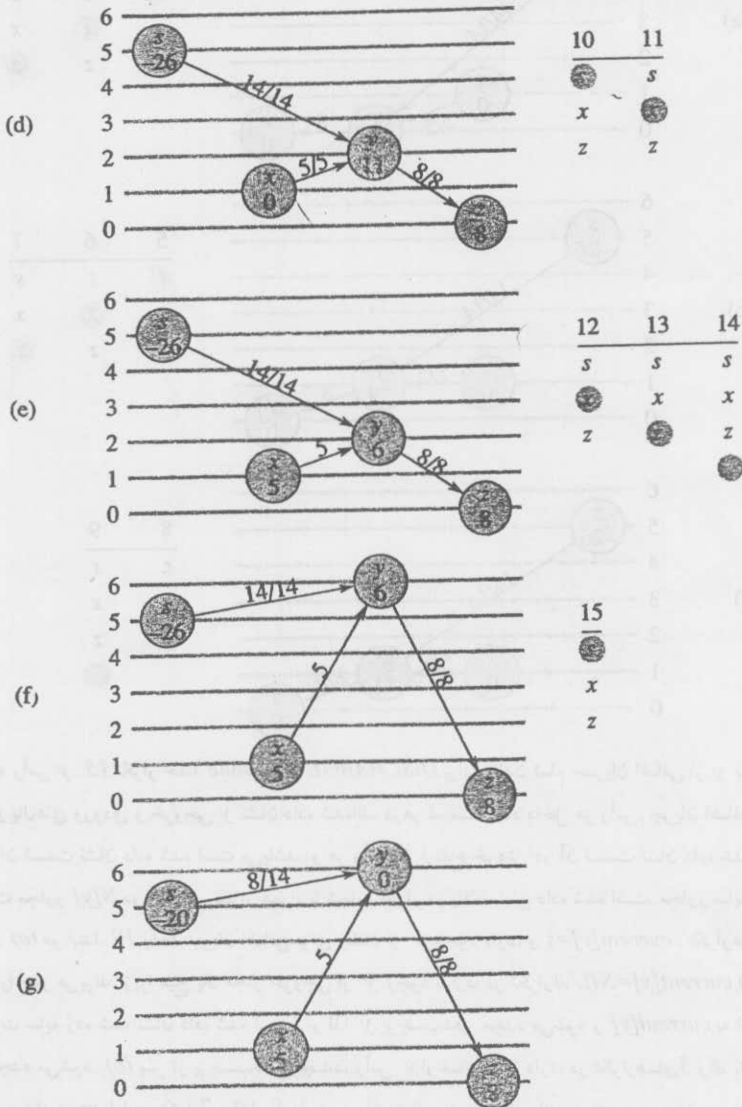
۱. اگر v برابر NIL باشد آنگاه به انتهای $N[u]$ رسیده‌ایم. خط ۴، رأس u را برچسب دهی مجدد می‌کند و سپس خط ۵ همسایه جاری u را مقداردهی مجدد می‌کند تا اولین عنصر $N[u]$ باشد. (لم ۲۶.۳۰

---

1. neighbor lists



شکل۹. ۲۶ تخلیه رأس y. 15 تکرار حلقه *while* روال *DISCHARGE* برای راندن تمام جریان اضافی از y بکار می‌رود. تنها، مجاورهای y و یال‌های ورودی و خروجی y نشان داده شده‌اند. درهر قسمت، عدد داخل هر رأس، جریان اضافه‌اش در آغاز اولین تکرار که در آن قسمت نشان داده شده است می‌باشد، و هر رأس در ارتفاع خودش در آن قسمت نشان داده شده است. در سمت راست، لیست مجاور *N[y]* در آغاز هر تکرار همراه با شماره تکرار در بالا نمایش داده شده است. مجاور سایه زده شده، *current[y]* است. *(a)* در ابتدا، *19* واحد جریان اضافی برای راندن از y وجود دارد، و *current[y]=s* . تکرارهای *1*، *2* و *3* تنها *current[y]* را جلو می‌برند، زیرا هیچ یال مجاز خروجی از y وجود ندارد. در تکرار*4*، *current[y]=NIL* (که در زیر لیست مجاور بصورت سایه زده شده نشان داده شده است)، و لذا y برچسب‌دهی مجدد می‌شود و *current[y]* به ابتدای لیست مجاور مقداردهی مجدد می‌شود. *(b)* پس از برچسب‌دهی مجدد، رأس y ارتفاع *1* را دارد. در تکرارهای*5* و*6*، یال‌های *(y,s)* و*(y,x)*، یال‌هایی غیرمجاز هستند اما در تکرار*7*، مقدار *8* واحد جریان اضافی از y به z رانده می‌شود. به علت عمل راندن، *current[y]* در این تکرار به جلو برده نمی‌شود. *(c)* چون عمل راندن در تکرار*7*، یال *(y,z)* را اشباع کرده است، در تکرار*8* این یال غیرمجازاست. در تکرار*9*، *current[y]=NIL* و بنابراین دوباره رأس y برچسب‌دهی مجدد و *current[y]* مقداردهی مجدد می‌شود. *(d)* درتکرار *10*، یال *(y,s)* غیر مجاز است اما *15* واحد جریان اضافی در تکرار*11* از y به x رانده می‌شود. *(e)* چون *current[y]* در تکرار *11* به جلو برده نشده است، تکرار*12*، یال *(y,x)* را غیرمجاز می‌یابد. تکرار*13*، یال *(y,z)* را غیرمجاز می‌یابد و تکرار *14*، رأس y را برچسب‌دهی مجدد کرده و *current[y]* را مقداردهی مجدد می‌کند. *(f)*تکرار*15*، *6* واحد جریان اضافی را از y به s می‌راند. *(g)* اینک رأس y هیچ جریان اضافی ندارد و *DISCHARGE* خاتمه می‌یابد. در این مثال *DISCHARGE* با اشاره‌گر جاری واقع در ابتدای لیست مجاور شروع می‌شود و خاتمه می‌یابد، امادر حالت کلی این مورد الزامی نیست.



که در ادامه آمده است بیان می‌کند که عمل برچسب دهی مجدد در این وضعیت به کار می‌رود.)

۲. اگر v غیر NIL و (u,v) یک یال مجاز باشد (که توسط تست خط ۶ تعیین گردیده است) آنگاه خط ۷ مقداری از (یا ممکن است همه) جریان اضافی u را به سمت رأس v براند.

۳. اگر v غیر NIL ولی (u,v) غیر مجاز باشد آنگاه خط ۸، $current[u]$ در لیست مجاور $N[u]$ را یک مکان به جلو می‌برد.

مشاهده می‌شود اگر $DISCHARGE$ برای رأس سرریز u فراخوانی شود، آنگاه آخرین عملی که

توسط *DISCHARGE* اجرا می‌شود باید راندن از u باشد. چرا؟ روال تنها زمانی خاتمه می‌یابد که $e[u]$ صفر شود، و عمل برچسب دهی مجدد و یا عمل پیش بردن اشاره‌گر $current[u]$ به جلو بر مقدار $e[u]$ تأثیر نمی‌گذارد.

باید مطمئن باشیم زمانی که *PUSH* و *RELABEL* توسط *DISCHARGE* فراخوانی می‌شوند، این اعمال قابل اجرا هستند. لم بعد این حقیقت را اثبات می‌کند.

لم ۲۶.۳۰

اگر *DISCHARGE*، $PUSH(u,v)$ را در خط ۷ فراخوانی کند آنگاه عمل راندن برای (u,v) قابل اجرا است.

اگر *DISCHARGE*، $RELABEL(u)$ را در خط ۴ فراخوانی کند آنگاه عمل برچسب دهی مجدد برای u قابل اجرا است.

اثبات تست‌های خطوط ۱ و ۶ اطمینان حاصل می‌کنند که عمل راندن تنها زمانی اتفاق می‌افتد که قابل اجرا باشد، که اوّلین قسمت لم را ثابت می‌کند.

برای اثبات قسمت دوم لم، طبق تست خط ۱ و لم ۲۶.۲۹ تنها لازم است نشان دهیم همه یال‌هایی که از u خارج می‌شوند غیر مجاز هستند. مشاهده می‌کنید همان طور که $DISCHARGE(u)$ به طور پی‌درپی فراخوانی می‌شود، اشاره‌گر $current[u]$ در لیست $N[u]$ به پایین حرکت می‌کند. هر "گذر" از ابتدای $N[u]$ آغاز و با $current[u]=NIL$ خاتمه می‌یابد، که در آن مرحله u برچسب دهی مجدد شده و یک گذر جدید آغاز می‌گردد. برای اشاره‌گر $current[u]$ که در طی یک گذر از رأس $v\in N[u]$ می‌گذرد، یال (u,v) باید توسط تست خط ۶ غیر مجاز فرض شود. بنابراین زمانی که گذر تکمیل می‌شود، هر یال خروجی از u در زمانی در طی گذر، غیر مجاز تعیین گردیده است. مشاهده کلیدی آن است که در پایان گذر، هر یال خروجی از u هنوز غیر مجاز است. چرا؟ بنا به لم ۲۶.۲۸، راندن‌ها نمی‌توانند یال‌های مجاز ایجاد کنند و لذا یال خروجی از u را تغییر نمی‌دهند. بنابراین هر یال مجاز باید توسط عمل برچسب‌دهی مجدد ایجاد شده باشد. اما رأس u در طی گذر برچسب دهی مجدد نشده است، و بنا به لم ۲۶.۲۹، هر رأس v دیگر که در طی گذر برچسب دهی مجدد شده است پس از برچسب دهی مجدد، هیچ یال مجاز ورودی ندارد. بنابراین در پایان گذر هم یال‌های خروجی از u غیر مجاز باقی می‌مانند و لم ثابت می‌شود. ■

الگوریتم برچسب دهی مجدد-به-جلو

در الگوریتم برچسب دهی مجدد-به-جلو، لیست پیوندی L را نگه می‌داریم که از همه رأس‌های واقع در $V\text{-}\{s,t\}$ تشکیل شده است. همان طور که در ثابت حلقه زیر خواهیم دید یک ویژگی کلیدی این است که

رأس‌های واقع در L به طور موضعی مطابق با شبکه مجاز مرتب شده‌اند. (از لم ۲۶.۲۷ به خاطر آورید که شبکه مجاز، یک *dag* است.)

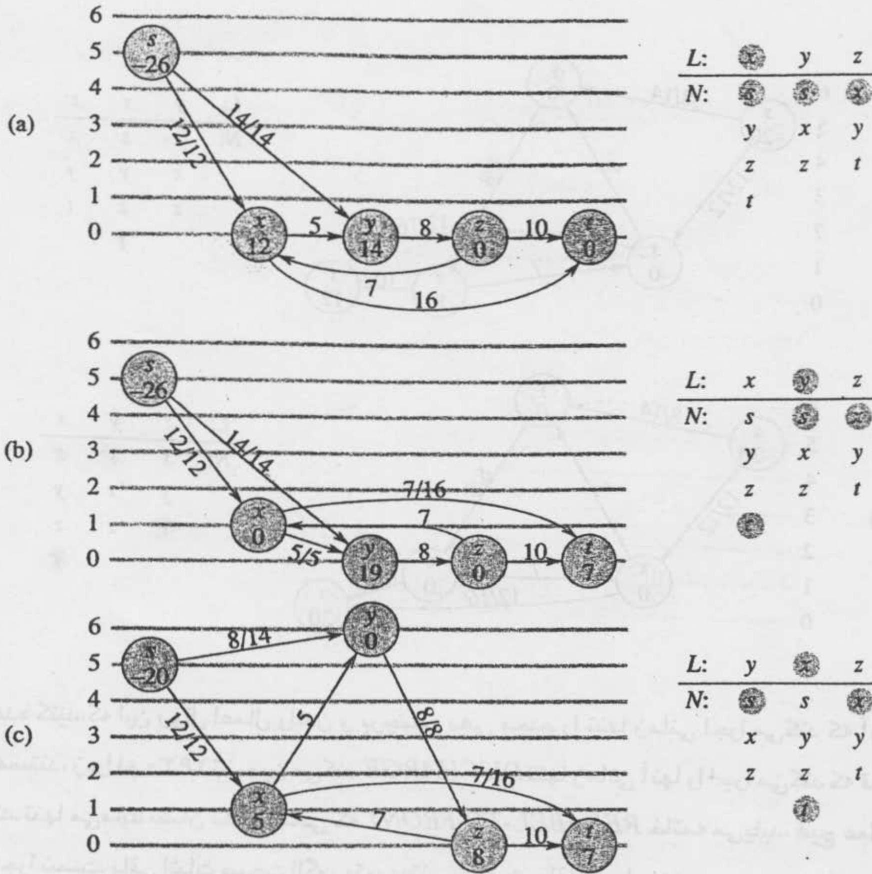
شبه کد برچسب دهی مجدد-به-جلو فرض می‌کند که از قبل، لیست مجاور $N[u]$ برای هر رأس u ایجاد شده است. همچنین فرض می‌کند $next[u]$ به رأسی که در لیست L پس از u آمده است اشاره می‌کند و طبق معمول اگر u آخرین رأس لیست باشد، $next[u]=NIL$.

```
RELABEL-TO-FRONT(G, s, t)
 1  INITIALIZE-PREFLOW(G, s)
 2  L ← V[G] − {s, t}, in any order
 3  for each vertex u ∈ V[G] − {s, t}
 4       do current[u] ← head[N[u]]
 5  u ← head[L]
 6  while u ≠ NIL
 7       do old-height ← h[u]
 8          DISCHARGE(u)
 9          if h[u] > old-height
10             then move u to the front of list L
11          u ← next[u]
```

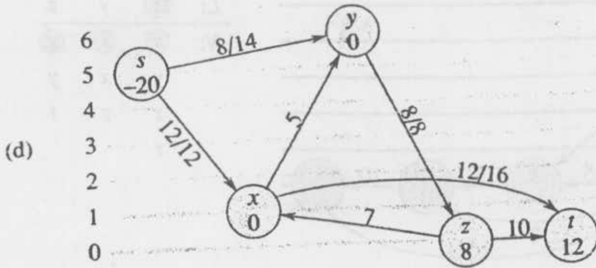
الگوریتم برچسب دهی مجدد-به-جلو به شکل زیر عمل می‌کند. خط ۱ جریان ماقبل و ارتفاع‌ها را با همان مقادیر الگوریتم راندن-برچسب دهی مجدد مقدار دهی اولیه می‌کند. خط ۲ لیست L را مقدار دهی اوّلیه می‌کند تا با هر ترتیبی شامل همه رأس‌هایی باشد که به صورت بالقوه سرریز هستند. خطوط ۴-۳، اشاره‌گر *current* هر رأس u را با اوّلین رأس لیست مجاور u مقدار دهی اوّلیه می‌کنند.

همان طور که در شکل ۲۶.۱۰ نشان داده شده است حلقه *while* خطوط ۱۱-۶ در طول لیست L اجرا می‌شود و رأس‌ها را تخلیه می‌کند. خط ۵ سبب می‌شود که حلقه با اوّلین رأس لیست آغاز می‌شود. هر بار در طول حلقه، یک رأس u در خط ۸ تخلیه می‌شود. اگر u به وسیله روال *DISCHARGE* برچسب دهی مجدد شده باشد خط ۱۰ آن را به جلوی لیست L انتقال می‌دهد. تشخیص این موضوع به وسیله ذخیره ارتفاع u در متغیر *old-height* قبل از عمل تخلیه (خط ۷)، و مقایسه این ارتفاع ذخیره شده با ارتفاع u پس از تخلیه (خط ۹) صورت می‌گیرد. خط ۱۱ باعث می‌شود تکرار بعدی حلقه *while* از رأسی که در لیست L پس از u آمده است استفاده کند. اگر u به ابتدای لیست انتقال داده شده باشد رأسی که در تکرار بعدی از آن استفاده می‌شود رأسی است که در مکان جدید u در لیست پس از آن واقع است.

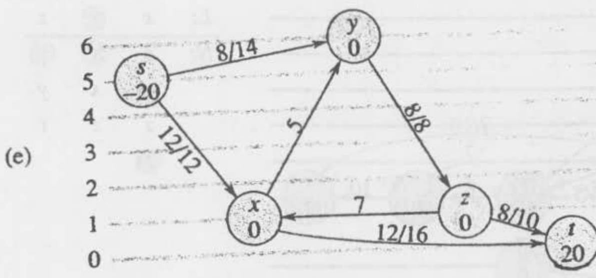
برای اینکه نشان دهیم *RELABEL-TO-FRONT* ماکزیمم جریان را محاسبه می‌کند، نشان خواهیم داد که این روال یک پیاده‌سازی از الگوریتم راندن-برچسب دهی مجدد ژنریک است. ابتدا



شکل ۲۶.۱۰ عملکرد *RELABEL-TO-FRONT* *(a)* شبکه جریان درست قبل از اوّلین تکرار حلقه *while* در ابتدا 26 واحد جریان، منبع s را ترک می‌کنند. در سمت راست، لیست اوّلیه $L=\langle x,y,z\rangle$ نشان داده شده است، که در ابتدا $u=x$ در زیر هر رأس واقع در لیست L، لیست مجاورش همراه با مجاور جاری‌اش که سایه‌زده شده است قرار دارد. رأس x تخلیه می‌شود. این رأس با ارتفاع 1 برچسب دهی مجدد می‌شود، 5 واحد جریان اضافه به y رانده می‌شوند، و 7 واحد باقیمانده جران اضافی به چاه t رانده می‌شوند. چون x برچسب دهی مجدد می‌شود به ابتدای لیست L انتقال می‌یابد که در این حالت ساختار L را تغییر نمی‌دهد. *(b)* رأس بعدی در لیست L که پس از x تخلیه می‌شود رأس y است. شکل ۲۶.۹ جزئیات عمل تخلیه y در این وضعیّت را نشان می‌دهد. چون y برچسب دهی مجدد می‌شود، به ابتدای L منتقل می‌شود. *(c)* اینک رأس x در L پس از رأس y قرار دارد، و بنابراین دوباره تخلیه می‌شود، و تمام 5 واحد جریان اضافی را به t می‌راند. چون رأس x دراین عمل تخلیه، برچسب دهی مجدد نمی‌شود در مکان خود در لیست L باقی می‌ماند. *(d)* چون رأس z در لیست L پس از x قرار دارد تخلیه می‌شود. این رأس با ارتفاع 1 برچسب دهی مجدد شده و همه 8 واحد جریان اضافی به t رانده می‌شوند. چون z برچسب دهی مجدد شده است به جلوی L منتقل می‌شود. *(e)* اینک رأس y پس از رأس z در L قرار دارد و بنابراین تخلیه می‌شود .اما چون y جریان اضافی ندارد، *DISCHARGE* بلافاصله بازگشت می‌کند و y در جای خود در لیست باقی می‌ماند. سپس رأس x تخلیه می‌شود. چون این رأس نیز جریان اضافی ندارد *DISCHARGE* دوباره بازگشت می‌کند و x در جای خود در L باقی می‌ماند. *RELABEL-TO-FRONT* به انتهای لیست L رسیده و خاتمه می‌یابد. هیچ رأس سرریزی وجود ندارد و جریان ماقبل، ماکزیمم جریان است.



L:	y	x	z
N:	s	s	x
	x	y	y
	z	z	t
		t	



L:	z	y	x
N:	x	s	s
	y	x	y
	t	z	z
			t

مشاهده کنید که این روال اعمال راندن و برچسب دهی مجدد را تنها زمانی اجرا می‌کند که آنها قابل اجرا هستند، زیرا لم ۲۶.۳۰ تضمین می‌کند *DISCHARGE* تنها زمانی آنها را اجرا می‌کند که قابل اجرا هستند. تنها می‌ماند نشان دهیم زمانی که *RELABEL-TO-FRONT* خاتمه می‌یابد، هیچ عمل اصلی قابل اجرا نیست. باقی اثبات صحت الگوریتم، مبتنی بر ثابت حلقه زیر است:

در هر تست خط ۶ روال *RELABEL-TO-FRONT*، لیست L یک ترتیب موضعی از رأس‌های شبکه مجاز $G_{f,h}=(V,E_{f,h})$ است و هیچ رأس قبل از u در لیست، جریان اضافی ندارد.

مقدار دهی اولیّه: بلافاصله پس از این که *INITIALIZE-PREFLOW* اجرا شده است، $h[s]=|V|$ و برای همه $v\in V\text{-}\{s\}$ داریم $h[v]=0$. از آنجا که $|V|\geq 2$ است (چون V حداقل شامل s و t می‌باشد) هیچ یالی نمی‌تواند مجاز باشد. بنابراین $E_{f,h}=\phi$ و هر ترتیبی از $V\text{-}\{s,t\}$ یک ترتیب موضعی از $G_{f,h}$ است. از آنجا که u در آغاز در ابتدای لیست L است هیچ رأسی قبل از آن وجود ندارد و بنابراین هیچ رأسی با جریان اضافی نیز قبل از آن وجود نخواهد داشت.

RELABEL-TO-FRONT به انتهای لیست L رسیده و خاتمه می‌یابد. هیچ رأس سرریزی وجود ندارد و جریان ماقبل، ماکزیمم جریان است.

نگهداری: برای این که ببینیم ترتیب موضعی بوسیله هر تکرار حلقه *while* حفظ می‌شود، با در نظر گرفتن این مطلب که شبکه مجاز تنها بوسیلهٔ اعمال راندن و برچسب دهی مجدد تغییر می‌کند آغاز می‌کنیم. بنا به لم ۲۶.۲۸ اعمال راندن باعث نمی‌شوند که یال‌ها مجاز گردند. بنابراین یال‌های مجاز

تنها به وسیله اعمال برچسب دهی مجدد می‌توانند ایجاد شوند. اگر چه پس از آنکه رأس u برچسب دهی مجدد می‌شود لم ۲۶.۲۹ بیان می‌کند که هیچ یال مجاز ورودی به u وجود ندارد، اما ممکن است یال‌های مجاز خروجی از u وجود داشته باشند. لذا به وسیله انتقال u به جلوی L الگوریتم اطمینان حاصل می‌کند که یال‌های مجاز خروجی از u ترتیب مرتب موضعی را رعایت می‌کنند.

برای این که ببینیم هیچ رأسی قبل از u در L جریان اضافی ندارد، بوسیله u' به رأسی که در تکرار بعدی، u خواهد بود اشاره می‌کنیم. رأسهایی که در تکرار بعدی قبل از u' قرار دارند عبارتند از رأس جاری u (طبق خط ۱۱)، و همان رأس‌های قبل (اگر u برچسب دهی مجدد نشده باشد) یا هیچ رأس دیگری (اگر u برچسب دهی مجدد شده باشد). چون رأس u تخلیه شده است، دیگر پس از آن هیچ جریان اضافی ندارد. بنابراین اگر u در طی تخلیه برچسب دهی مجدد شود، هیچ رأسی قبل از u' جریان اضافی ندارد. اگر u در طی تخلیه برچسب دهی مجدد نشود، هیچ رأسی قبل از آن در لیست در طول این تخلیه، جریان اضافی حاصل نمی‌کند زیرا L در طی تخلیه، مرتب شده موضعی باقی مانده است (همان طور که قبلاً ذکر شد یال‌های مجاز تنها به وسیله برچسب دهی مجدد ایجاد می‌شوند نه راندن) و بنابراین هر عمل راندن باعث می‌شود جریان اضافی تنها به رأس‌های پایین‌تر در لیست (به به s یا t) منتقل شود. دوباره هیچ رأسی قبل از u'، جریان اضافی ندارد.

خاتمه: وقتی که حلقه خاتمه می‌یابد، u دقیقاً از انتهای L عبور کرده است و بنابراین ثابت حلقه تضمین می‌کند که جریان اضافی هر رأس 5 است. لذا هیچ عمل اصلی بکار برده نمی‌شود.

تحلیل

اینک نشان خواهیم داد *RELABEL-TO-FRONT* روی شبکه جریان $G=(V,E)$ در زمان $O(V^3)$ اجرا می‌شود. چون این الگوریتم یک پیاده‌سازی الگوریتم راندن-برچسب دهی مجدد ژنریک است، از قضیه فرعی ۲۶.۲۲ استفاده می‌کنیم که حد $O(V)$ را برای تعداد اعمال برچسب دهی مجدد که روی هر رأس اجرا شده‌اند و حد $O(V^2)$ را برای تعداد اعمال برچسب دهی مجدد که روی کل رأس‌ها اجرا شده‌اند فراهم می‌کند. به علاوه تمرین ۲- ۲۶.۴ حد $O(VE)$ را روی زمان کل صرف شده برای اجرای اعمال برچسب دهی مجدد در نظر می‌گیرد، و لم ۲۶.۲۳ حد $O(VE)$ را روی تعداد کل اعمال راندن اشباع کننده فراهم می‌کند.

قضیه ۲۶.۳۱

زمان اجرای *RELABEL-TO-FRONT* روی شبکه جریان $G=(V,E)$ برابر $O(V^3)$ است.

اثبات فرض کنید یک "فاز" الگوریتم برچسب دهی مجدد-به-جلو، زمان بین دو عمل برچسب دهی مجدد متوالی باشد. $O(V^2)$ فاز وجود دارد زیرا $O(V^2)$ عمل برچسب دهی مجدد وجود دارد. هر فاز از

حداکثر $|V|$ فراخوانی *DISCHARGE* تشکیل شده است. که می‌توانند به شکل زیر مشاهده شوند. اگر *DISCHARGE* عمل برچسب دهی مجدد را اجرا نکند آنگاه فراخوانی بعدی *DISCHARGE* در قسمت پایین‌تر لیست L صورت می‌گیرد، و طول L کمتر از $|V|$ است. اگر *DISCHARGE* برچسب دهی مجدد را اجرا کند، فراخوانی بعدی *DISCHARGE* متعلق به فاز متفاوتی است. چون هر فاز شامل حداکثر $|V|$ فراخوانی *DISCHARGE* است و $O(V^2)$ فاز وجود دارد، تعداد دفعاتی که *DISCHARGE* در خط ۸ روال *RELABEL-TO-FRONT* فراخوانی می‌شود $O(V^3)$ است. بنابراین کل کاری که توسط حلقه *while* در روال *RELABEL-TO-FRONT* اجرا می‌شود، به استثنای کاری که در داخل *DISCHARGE* اجرا می‌شود، حداکثر $O(V^3)$ است.

اینک باید کاری که در داخل *DISCHARGE* در طی اجرای الگوریتم انجام می‌شود را محدود کنیم. هر تکرار حلقه *while* در داخل *DISCHARGE* یکی از سه عمل را اجرا می‌کند. کل کاری که در هر یک از این سه عمل صورت می‌پذیرد را تحلیل خواهیم کرد.

با اعمال برچسب دهی مجدد (خطوط ۵-۴) آغاز می‌کنیم. تمرین ۲-۲۶.۴ برای همه $O(V^2)$ برچسب دهی مجدد که اجرا می‌شوند، حد زمانی $O(VE)$ را فراهم می‌کند.

اکنون فرض کنید این عمل اشاره‌گر $current[u]$ را در خط ۸ بروزرسانی می‌کند. این عمل هر بار که یک رأس u برچسب دهی مجدد می‌شود، $O(degree(u))$ بار و برای همه رأس‌ها، $O(V.degree(u))$ بار اتفاق می‌افتد. بنابراین برای همه رأس‌ها کل کاری که جهت جلو بردن اشاره‌گر در لیست‌های مجاور صورت می‌گیرد بنا به لم دست‌دهی[1]، $O(VE)$ است.

سومین نوع عملی که توسط *DISCHARGE* اجرا می‌شود عمل راندن است (خط ۷). از قبل می‌دانیم تعداد کل اعمال راندن اشباع کننده $O(VE)$ است. مشاهده می‌شود اگر یک عمل راندن غیر اشباع کننده اجرا می‌شود، *DISCHARGE* فوراً بازگشت می‌کند زیرا این راندن جریان اضافی را به صفر کاهش می‌دهد. بنابراین حداکثر یک راندن غیراشباع کننده در هر فراخوانی *DISCHARGE* می‌تواند وجود داشته باشد. همان طور که مشاهده کرده‌ایم *DISCHARGE*، $O(V^3)$ بار فراخوانی می‌شود و بنابراین کل زمانی که جهت اجرای راندن‌های غیراشباع کننده صرف می‌شود $O(V^3)$ است. بنابراین زمان اجرای *RELABEL-TO-FRONT* برابر $O(V^3+VE)$ است که برابر است با $O(V^3)$.

■

تمرین‌ها

۱-۲۶.۵ اجرای *RELABEL-TO-FRONT* برای شبکه جریان شکل (a) ۲۶.۱ را به روش شکل ۲۶.۱۰ توضیح دهید. فرض کنید ترتیب اوّلیه رأس‌های L به شکل $<v_1,v_2,v_3,v_4>$ است و لیست‌های

۱ـ اگر $G=(V,E)$ یک گراف بدون جهت باشد آنگاه داریم $\sum_{v \in V} \text{degree}(v) = 2|E|$.

مجاور عبارتند از

$$
\begin{aligned}
N[v_1] &= \langle s, v_2, v_3 \rangle , \\
N[v_2] &= \langle s, v_1, v_3, v_4 \rangle , \\
N[v_3] &= \langle v_1, v_2, v_4, t \rangle , \\
N[v_4] &= \langle v_2, v_3, t \rangle .
\end{aligned}
$$

۲۶.۵-۲* می‌خواهیم الگوریتم راندن-برچسب دهی مجدد را پیاده‌سازی کنیم که در آن یک صف اوّلین ورودی-اوّلین خروجی از رأس‌های سرریز را نگه می‌داریم. الگوریتم به طور پی‌درپی رأس واقع در ابتدای صف را تخلیه می‌کند، و هر رأسی که قبل از تخلیه سرریز نبوده ولی پس از آن سرریز می‌شود در انتهای صف قرار می‌گیرد. پس از آن که رأس واقع در ابتدای صف تخلیه شد، حذف می‌گردد. زمانی که صف خالی باشد الگوریتم خاتمه می‌یابد. نشان دهید این الگوریتم می‌تواند چنان پیاده‌سازی شود که ماکزیمم جریان را در زمان $O(V^3)$ محاسبه کند.

۲۶.۵-۳ نشان دهید الگوریتم ژنریک اگر *DISCHARGE* بوسیله محاسبه ساده $h[u] \leftarrow h[u]+1$ مقدار $h[u]$ را به روزرسانی کند، باز هم کار می‌کند. این تغییر چگونه در تحلیل *RELABEL-TO-FRONT* تأثیر می‌گذارد؟

۲۶.۵-۴* نشان دهید اگر همیشه بالاترین رأس سرریز را تخلیه کنیم، روش راندن-برچسب دهی مجدد می‌تواند در زمان $O(V^3)$ اجرا شود.

۲۶.۵-۵ فرض کنید در بعضی مراحل اجرای الگوریتم راندن-برچسب دهی مجدد، یک عدد صحیح $0<k\leq|V|-1$ وجود دارد که برای آن هیچ رأسی دارای $h[v]=k$ نیست. نشان دهید همه رأس‌ها با $h[v]>k$ در طرف منبع یک مینیمم بریدگی هستند. اگر چنین kای وجود داشته باشد *gap heuristic*، هر رأس $v \in V-s$ که برای آن $h[v]>k$ است را به صورت $h[v] \leftarrow \max(h[v], |v|+1)$ به روزرسانی می‌کند. نشان دهید خصوصیّت h بدست آمده، یک تابع ارتفاع است. *gap heuristic* در پیاده‌سازی‌های روش راندن-برچسب دهی مجدد که در عمل به خوبی اجرا می‌شوند لازم است.)

مسائل

۲۶-۱ مسئله گریز[1]

یک شبکه[2] $n\times n$ همان طور که در شکل ۲۶.۱۱ نشان داده شده است، یک گراف بدون جهت و متشکل از n سطر و n ستون از رأس‌ها می‌باشد. به رأس واقع در iامین سطر و jامین ستون با (i,j) اشاره

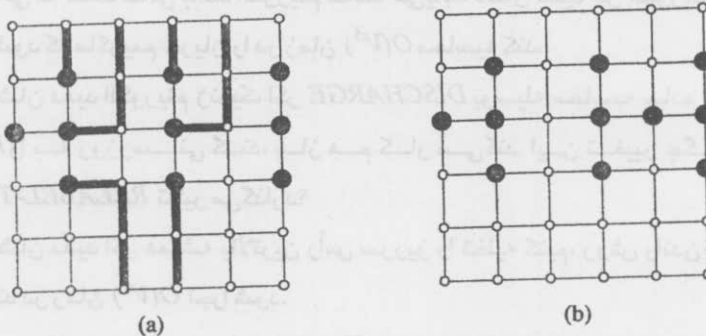
1.escape problem
2.grid

می‌کنیم. همه رأس‌های شبکه دقیقاً 4 مجاور دارند، غیر از رأس‌های مرزی که نقاط (i,j) هستند و برای آنها $i=1$، $i=n$، $j=1$ یا $j=n$

با در نظر گرفتن $m \le n^2$ نقاط آغازین داده شده $(x_1,y_1),(x_2,y_2),...,(x_m,y_m)$ در یک شبکه، مسئله گریز این است که تعیین کنیم آیا m مسیر جدا از هم از نقاط شروع به m نقطه متفاوت مرزی وجود دارد یا نه. برای مثال، شبکه شکل (a) ۲۶.۱۱ یک گریز دارد اما شبکه شکل (b) ۲۶.۱۱ خیر.

a یک شبکه جریان که در آن رأس‌ها نیز مانند یال‌ها دارای ظرفیّت هستند را در نظر بگیرید. به عبارت دیگر جریان کل مثبت ورودی به یک رأس، تابع محدودیت ظرفیّت است. نشان دهید تعیین ماکزیمم جریان در یک شبکه با ظرفیّت‌های یال‌ها و رأس‌ها می‌تواند به یک مسئله ماکزیمم جریان عادی در یک شبکه جریان با سایز قابل مقایسه تبدیل شود.

b. یک الگوریتم کارآمد برای حل مسئله گریز ارائه دهید، و زمان اجرای آن را تحلیل کنید.



شکل ۲۶.۱۱ شبکه‌ها برای مسئله گریز، نقاط آغازین سیاه و دیگر رأس‌های شبکه سفید هستند. (a) شبکه‌ای با یک گریز، که به وسیله مسیرهای سایه‌زده شده نشان داده شده است. (b) شبکه‌ای بدون گریز.

۲۶-۲ مینیمم پوشش مسیر[1]

یک پوشش مسیر گراف جهت‌دار $G=(V,E)$، مجموعه p از مسیرهای جدا از هم است به طوری که هر رأس مجموعه V دقیقاً در یک مسیر p قرار می‌گیرد. مسیرها ممکن است از هر جایی آغاز یا در هر جایی خاتمه یابند، و هر طولی نیز ممکن است داشته باشند از جمله صفر. مینیمم پوشش مسیر گراف G، پوشش مسیری است که شامل کمترین مسیرهای ممکن باشد.

a. الگوریتمی کارآمد ارائه دهید که مینیمم پوشش مسیر گراف جهت‌دار بدون دور $G=(V,E)$ را بیابد. (راهنمایی: با فرض این که $V=\{1,2...,n\}$، گراف $G'=(V',E')$ را بسازید که

1.minimum path cover

$$V' = \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\} ,$$
$$E' = \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\}$$

و الگوریتم ماکزیمم جریان را اجرا کنید.)

b. آیا الگوریتم شما برای گراف‌های جهت‌دار دارای دور نیز کار می‌کند؟ توضیح دهید.

۲۶-۳ آزمایشهای شاتل فضایی

پروفسور $Spock$ مشاور $NASA$ است، که یک سری پروازهای شاتل فضایی را برنامه ریزی می‌کند و باید تعیین کند که چه آزمایشات تجاری انجام شود و چه تجهیزاتی در هر پرواز قرار گیرند. برای هر پرواز، $NASA$ مجموعه $E=\{E_1,E_2,...,E_m\}$ از آزمایشات را در نظر می‌گیرد، و حامی مالی آزمایش E_j موافقت کرده است که p_j دلار برای نتایج آزمایش به $NASA$ پرداخت کند. آزمایشات از مجموعه $I=\{I_1,I_2,...,I_n\}$ از تجهیزات استفاده می‌کنند؛ هر آزمایش E_j همه تجهیزات واقع در زیر مجموعه $R_j \subseteq I$ را احتیاج دارد. هزینه حمل تجهیزات I_k c_k دلار است. کار پرو فسور این است که الگوریتم کارآمدی بیابد که تعیین کند کدام آزمایشات انجام شوند و کدام تجهیزات برای پرواز مورد نظر حمل شوند تا درآمد خالص که کل درآمد حاصل از انجام آزمایشات منهای هزینه کل تمام تجهیزات حمل شده می‌باشد، ماکزیمم شود.

شبکه G زیر را در نظر بگیرید. شبکه شامل رأس منبع s، رأس‌های $I_1,I_2,...,I_n$، رأس‌های $E_1,E_2,...,E_m$، و رأس چاه t است. برای $k=1,2,...,n$ یال (s,I_k) با ظرفیّت c_k و برای $j=1,2,...,m$ یال (E_j,t) با ظرفیّت p_j وجود دارد. برای $k=1,2,...,n$ و $j=1,2,...,m$ اگر $I_k \in R_j$ باشد آنگاه یال (I_k,E_j) با ظرفیّت بی نهایت وجود دارد.

a. نشان دهید اگر برای بریدگی با ظرفیّت محدود (S,T) از G، $E_j \in T$ باشد آنگاه برای هر $I_k \in R_j$ داریم $I_k \in T$

b. نشان دهید چگونه می‌توان درآمد خالص ماکزیمم را از ظرفیّت مینیمم بریدگی گراف G و مقادیر داده شده p_j تعیین کرد.

c. الگوریتمی کارآمد برای تعیین اینکه کدام آزمایشات اجرا شوند و کدام تجهیزات حمل شوند ارائه دهید. زمان اجرای الگوریتم خود را برحسب n,m و $r = \sum_{j=1}^{m} |R_j|$ تحلیل کنید.

۲۶-۴ به روزرسانی ماکزیمم جریان

فرض کنید $G=(V,E)$ یک شبکه جریان با منبع s، چاه t و ظرفیّت‌های صحیح باشد. فرض کنید یک ماکزیمم جریان در G به ما داده شده است.

a. فرض کنید ظرفیّت تک یال $(u,v) \in E$ به اندازه 1 واحد افزایش یابد. الگوریتمی با زمان اجرای

$O(V+E)$ برای به روزرسانی ماکزیمم جریان ارائه دهید.

b. فرض کنید ظرفیّت تک یال $(u,v)\in E$ به اندازه 1 واحد کاهش یابد. الگوریتمی با زمان اجرای $O(V+E)$ برای به روزرسانی ماکزیمم جریان ارائه دهید.

۲۶-۵ ماکزیمم جریان بوسیله مقیاس دهی[1]

فرض کنید $G=(V,E)$ یک شبکه جریان با منبع s، چاه t و ظرفیّت صحیح $c(u,v)$ روی هر یال $(u,v)\in E$ باشد. قرار دهید $C = \max_{(u,v)\in E} c(u, v)$.

a. ثابت کنید که مینیمم بریدگی شبکه G دارای حداکثر ظرفیّت $C|E|$ می‌باشد.

b. برای عدد داده شده K نشان دهید که یک مسیر تکمیلی با ظرفیّت حداقل K می‌تواند در زمان $O(E)$ پیدا شود، اگر چنین مسیری وجود داشته باشد. تغییر زیر از *FORD-FULKERSON-METHOD* می‌تواند در محاسبه ماکزیمم جریان در G مورد استفاده قرار گیرد.

```
MAX-FLOW-BY-SCALING(G, s, t)
1  C ← max_{(u,v)∈E} c(u, v)
2  initialize flow f to 0
3  K ← 2^{⌊lg C⌋}
4  while K ≥ 1
5      do while there exists an augmenting path p of capacity at least K
6             do augment flow f along p
7         K ← K/2
8  return f
```

c. ثابت کنید که *MAX-FLOW-BY-SCALING* ماکزیمم جریان را برمی‌گرداند.

d. نشان دهید هر بار که خط ۴ اجرا می‌شود ظرفیّت مینیمم بریدگی گراف پسماند G_f حداکثر $2K|E|$ است.

e. ثابت کنید که حلقه *while* داخلی خطوط ۶-۵ برای هر مقدار K، $O(E)$ بار اجرا می‌شود.

f. نتیجه بگیرید که *MAX-FLOW-BY-SCALING* می‌تواند چنان پیاده‌سازی شود که در زمان $O(E^2 lgc)$ اجرا گردد.

۲۶-۶ ماکزیمم جریان با ظرفیّت‌های منفی

فرض کنید شبکه جریان (همان طور که دارای یال‌های با ظرفیّت‌های مثبت است) دارای یال‌هایی با

---
1. scaling

ظرفیّت‌های منفی باشد. در چنین شبکه‌ای لزوماً یک جریان ممکن وجود ندارد.

a. یال (u,v) در شبکه جریان $G=(V,E)$ با $c(u,v)<0$ را در نظر بگیرید. به طور خلاصه مفهوم این ظرفیّت منفی را برحسب جریان بین u و v شرح دهید.

فرض کنید $G=(V,E)$ یک شبکه جریان با یال‌های با ظرفیّت منفی باشد و فرض کنید s و t، منبع و چاه G باشند. شبکه جریان معمول $G'=(V',E')$ با تابع ظرفیّت c'، منبع s' و چاه t' را تشکیل دهید که

$$V' = V \cup \{s', t'\}$$

و

$$\begin{aligned} E' = E &\cup \{(u, v) : (v, u) \in E\} \\ &\cup \{(s', v) : v \in V\} \\ &\cup \{(u, t') : u \in V\} \\ &\cup \{(s, t), (t, s)\} \ . \end{aligned}$$

ظرفیّت‌ها را به شکل زیر به یال‌ها انتساب می‌دهیم. برای هر یال $(u,v) \in E$ قرار می‌دهیم

$$c'(u, v) = c'(v, u) = (c(u, v) + c(v, u))/2 \ .$$

برای هر رأس $u \in V$ قرار می‌دهیم

$$c'(s', u) = \max(0, (c(V, u) - c(u, V))/2)$$

و

$$c'(u, t') = \max(0, (c(u, V) - c(V, u))/2) \ .$$

همچنین قرار می‌دهیم

$$c'(s, t) = c'(t, s) = \infty.$$

b. ثابت کنید اگر یک جریان ممکن در G وجود داشته باشد آنگاه همه ظرفیّت‌های واقع در G' غیر منفی هستند و یک ماکزیمم جریان در G' وجود دارد بطوری که همه یال‌ها به چاه t' اشباع شده هستند.

c. عکس قسمت (b) را ثابت کنید. اثبات شما باید سودمند باشد به عبارت دیگر با دریافت یک جریان در G' که همه یال‌ها به چاه t' را اشباع می‌کند، اثبات شما باید نشان دهد چگونه می‌توان به یک جریان ممکن در G دست یافت.

d. الگوریتمی را توصیف کنید که یک ماکزیمم جریان ممکن در G را می‌یابد. زمان اجرا در بدترین حالت الگوریتم ماکزیمم جریان معمول روی یک گراف با $|V|$ رأس و $|E|$ یال را بوسیلهٔ $MF(|V|,|E|)$ نشان دهید. الگوریتم خود جهت محاسبه ماکزیمم جریان یک شبکه جریان با ظرفیّت‌های منفی را بر حسب MF تحلیل کنید.

۲۶-۷ الگوریتم تطبیق دوبخشی *Hopcroft-Karp*

در این مسئله برای یافتن ماکزیمم تطبیق در یک گراف دوبخشی، الگوریتم سریعتری را توصیف می‌کنیم که مربوط به *Hopcroft* و *Karp* است. الگوریتم در زمان $O(\sqrt{V}E)$ اجرا می‌شود. گراف بدون جهت دوبخشی $G=(V,E)$ که در آن $V=L\cup R$ و همه یال‌ها دقیقاً یک نقطه انتهایی در L دارند را در نظر بگیرید، و فرض کنید M یک تطبیق در G باشد. می‌گوییم مسیر ساده p در G نسبت به M یک مسیر تکمیلی است اگر از یک رأس تطبیق نیافته در L آغاز شود، در یک رأس تطبیق نیافته در R خاتمه یابد، و یال‌هایش متناوباً به M و $E\text{-}M$ تعلق داشته باشند. (این تعریف از مسیر تکمیلی، مربوط اما متفاوت با مسیر تکمیلی در شبکه جریان است.) در این مسئله مسیر را به جای یک توالی از رأس‌ها، یک توالی از یال‌ها در نظر می‌گیریم. کوتاهترین مسیر تکمیلی نسبت به تطبیق M، مسیر تکمیلی با مینیمم تعداد یال‌ها است. در مجموعه‌های A و B، *تفاضل متقارن*[1] $A\oplus B$ به شکل $(A\text{-}B)\cup(B\text{-}A)$ تعریف می‌شود، به عبارت دیگر اعضایی که دقیقاً در یکی از دو مجموعه قرار دارند.

a. نشان دهید اگر M یک تطبیق و P یک مسیر تکمیلی نسبت به M باشد آنگاه تفاضل متقارن $M\oplus P$ یک تطبیق است و $|M\oplus P|=|M|+1$. نشان دهید اگر $P_1,P_2,...,P_k$ نسبت به M مسیرهای تکمیلی جدا از هم باشند آنگاه تفاضل متقارن $M\oplus(P_1\cup P_2\cup...\cup P_k)$ یک تطبیق با اندازه $|M|+K$ است.

ساختار کلی الگوریتم ما به شکل زیر است:

```
HOPCROFT-KARP(G)
1  M ← Ø
2  repeat
3      let 𝒫 ← {P1, P2, ..., Pk} be a maximum set of vertex-disjoint
              shortest augmenting paths with respect to M
4      M ← M ⊕ (P1 ∪ P2 ∪ ··· ∪ Pk)
5    until 𝒫 = Ø
6  return M
```

باقی این مسئله از شما می‌خواهد تعداد تکرارها در الگوریتم را تحلیل کنید (به عبارت دیگر تعداد تکرارها در حلقه *repeat*) و پیاده‌سازی خط ۳ را شرح دهید.

b. با دو تطبیق داده شده M^*,M در G، نشان دهید که هر رأس در گراف $G'=(V,M\oplus M^*)$ دارای حداکثر درجه 2 می‌باشد. نتیجه بگیرید که G' یک واحد جدا از مسیرهای ساده و یا دورها می‌باشد. ثابت کنیدکه یال‌ها در هر چنین مسیر ساده یا دوری، متناوباً متعلق به M یا M^* هستند. ثابت کنید اگر $|M|\le|M^*|$، آنگاه $M\oplus M^*$ شامل حداقل $|M^*|-|M|$ مسیر تکمیلی جدا نسبت به M است. *l*

1.symmetric difference

را در طول کوتاهترین مسیر تکمیلی نسبت به تطبیق M، و $P_1, P_2, ..., P_k$ را بزرگترین مجموعه از مسیرهای‌تکمیلی جداباطول l نسبت‌به M در نظر بگیرید. قرار دهید $M' = M \oplus (P_1 \cup P_2 \cup ... \cup P_k)$ ، و فرض کنید P کوتاهترین مسیر تکمیلی نسبت به M' است.

c. نشان دهیداگر P از نظر رأس از $P_1, P_2, ..., P_k$ جدا باشد آنگاه P بیش از l یال دارد.

d. اکنون فرض کنید P از نظر رأس از $P_1, P_2, ..., P_k$ جدا نیست. فرض کنید A مجموعه یال‌های $(M \oplus M') \oplus P$ باشد. نشان دهید که $A = (P_1 \cup P_2 \cup ... \cup P_k) \oplus P$ و این که $|A| \geq (k+1)l$. نتیجه بگیرید که P بیش از l یال دارد.

e. ثابت کنید اگر کوتاهترین مسیر تکمیلی برای M دارای طول l باشد اندازه ماکزیمم تطبیق حداکثر $|M| + |V| / l$ است.

f. نشان دهید تعداد تکرارهای حلقه *repeat* در الگوریتم حداکثر $2\sqrt{V}$ است. (راهنمایی: پس از تکرار شماره $\sqrt{V}$، M چقدر می‌تواند رشد کند؟)

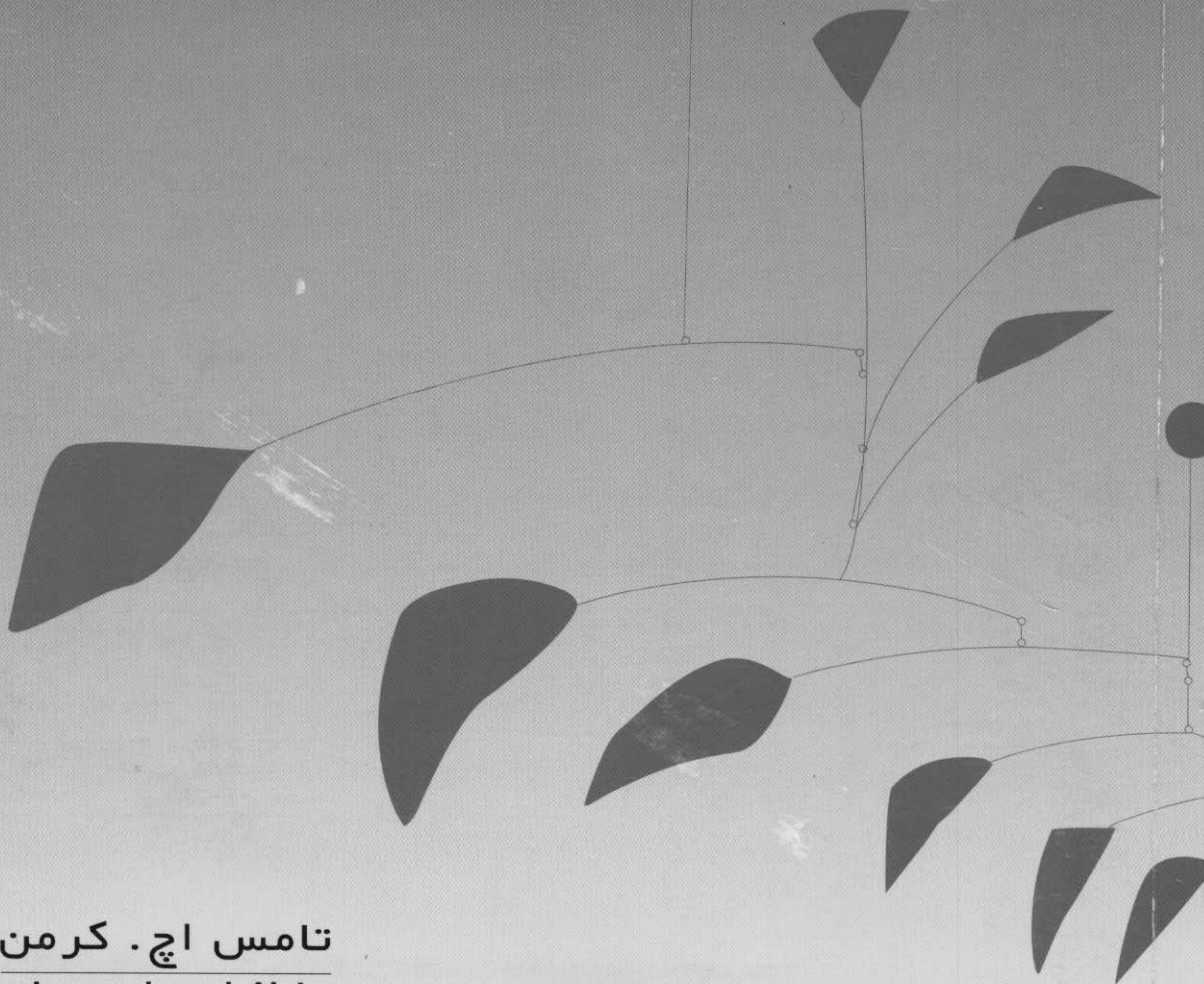
g. الگوریتمی ارائه دهید که در زمان $O(E)$ اجرا شود و برای تطبیق داده شده M، بزرگترین مجموعه از کوتاهترین مسیرهای تکمیلی $P_1, P_2, ..., P_k$ که از نظر رأس جدا از هم می‌باشند را بیابد. نتیجه بگیرید که زمان کل اجرای *HOPCROFT-KARP* برابر $O(\sqrt{V}E)$ می‌باشد.

مقدمه‌ای بر
الگوریتم‌ها

چاپ سوم

ویراست دوم

ترجمه : گروه مهندسی پژوهشی خوارزمی



تامس اچ. کرمن
چارلز ای. لیزرسان
رانولدال. رایوست
کلیفورد. استاین

INTRODUCTION TO
ALGORITHMS
SECOND EDITION

Thomas H. Cormen , Charles E. Leiserson , Ronald L. Rivest and Clifford Stein

در موضوع الگوریتم ها، کتابهای متعددی انتشار یافته است که هر چند با دقت تدوین شده اند، کامل نیستند و کتابهای دیگری که مطالب بسیاری را پوشش می دهند اما فاقد دقت کافی می باشند. **مقدمه ای بر الگوریتم ها** ، دقت و جامعیت را با هم جمع کرده است. این کتاب مقدمه ای گسترده برای مطالعه مدرن الگوریتم های کامپیوتر فراهم می کند.در این کتاب الگوریتم های زیادی ارائه و به طور عمیق پوشش داده شده اند ، که باعث می شود طراحی و تحلیل این الگوریتم ها برای خوانندگان در سطوح گوناگون امکان پذیر باشد.سعی کرده ایم بدون از دست دادن عمق مطلب یا دقت ریاضی،توضیحات را به زبان ساده بیان کنیم. اولین ویرایش اثر حاضر بعنوان مرجع استاندارد برای افراد حرفه ای و یک کتاب درسی پرکاربرد در دانشگاه های سرتاسر جهان مورد استفاده قرار گرفت. ویرایش دوم به نقش الگوریتم ها، تحلیل احتمالی و الگوریتم های تصادفی و برنامه سازی خطی و همچنین به بازنگری وسیع تمام بخش های کتاب می پردازد . در یک تغییر ظریف اما مهم ، ثابت های حلقه درابتدای کتاب معرفی شده اند ودر سرتاسر کتاب برای اثبات صحت الگوریتم ها به کار رفته اند . منابع مورد نیاز دانشجویان و مدرسان عزیز درسایت http://www.mhhe.com/cormen قابل دسترس می باشد .

تامس اچ.کرمن ، دانشیار علوم کامپیوتر در دانشکده دارتموث است.
چارلز ای.لیرزسان استاد علوم کامپیوتر و مهندسی برق در مو سسه فن آوری ماساچوست می باشد .
رانولد ال. رایوست استاد علوم کامپیوتر در موسسه فن آوری ماساچوست است.
کلیفورد استاین دانشیار مهندسی صنعتی و تحقیقات علمی در دانشگاه کلمبیا می باشد .

اساتید و دانشجویان گرامی می توانند نظریات و پیشنهادات خود را
در مورد این کتاب به آدرس پست الکترونیکی
kharazmi.regroup@gmail.com **ارسال نمایند.**

مشهد: خیابان سعدی، پاساژ مهتاب ، شماره ۲۵ ، نشر درخشش
تلفن : ۲۲۵۱۹۲۳ فاکس : ۸۴۴۶۵۰۷
قیمت : ۵۵۰۰۰ ریال
نشر درخشش