جلد ۱

ویراست سوم

چاپ سوم

# مقدمه‌ای بر الگوریتم‌ها


نویسندگان

توماس کورمن

چارلز لیزرسون

رونالد ریوست

کلیفورد استین

ترجمه

**مهندس دهقان طرزه**

با مقدمهٔ

**دکتر یحیی تابش**

هیئت علمی دانشگاه صنعتی شریف

| | | |
|---|---|---|
| سرشناسه | : | توماس کرمن ... [و دیگران]؛ Cormen, Thomas H |
| عنوان و نام پدیدآور | : | مقدمه‌ای بر الگوریتم‌ها/ توماس کرمن، چارلز لیزرسون، رونالد ریوست، کلیفورد استین؛ [ترجمه] علی دهقان. |
| مشخصات نشر | : | تهران: نص، ۱۳۹۳. |
| مشخصات ظاهری | : | ۲ جلدی، مصور، جدول، نمودار، دورنگ. |
| قیمت | : | ۲۰۰۰۰ تومان |
| شابک ج ۱ | : | ۹۷۸-۹۶۴-۴۱۰-۲۵۸-۵ |
| وضعیت فهرست نویسی | : | فیپا. |
| عنوان اصلی | : | Introduction to algorithms, 3rd ed, c2009 |
| موضوع | : | برنامه‌نویسی. |
| موضوع | : | الگوریتم‌های کامپیوتری. |
| شناسه افزوده | : | کرمن، تامس. |
| شناسه افزوده | : | توماس اچ .کورمن، چارلز لیزرسون، رونالد ریوست، کلیفورد استین. |
| شناسه افزوده | : | دهقان، علی، ۱۳۶۴ - [ مترجم]. |
| رده‌بندی کنگره | : | QA۷۶/۶/م۶۷ ۱۳۸۹ |
| رده‌بندی دیویی | : | ۰۰۵/۱ |
| شماره کتابشناسی ملی | : | ۲۰۷۳۹۱۴ |

موسسه علمی فرهنگی

**مقدمه‌ای بر الگوریتم‌ها**/ جلد ۱ (ویراست سوم)

توماس کرمن/ چارلز لیزرسون/ رونالد ریوست/ کلیفورد استین

مهندس علی دهقان طرزه با نظارت دکتر یحیی تابش

چاپ سوم: زمستان ۹۳

شمارگان: ۱۰۰۰

ناشر: نص

طراحی، چاپ، صحافی: مؤسسه علمی فرهنگی «نص»

قیمت: ۲۰۰۰۰ تومان



**دفتر**: تهران، میدان انقلاب، خ منیری جاوید، بن بست مبین، شماره ۶

تلفن: ۶۶۴۱۲۳۸۵ - ۶۶۴۶۵۶۷۴ - ۶۶۹۵۳۸۸۳ فاکس: ۶۶۹۵۷۶۹۰

ایمیل : info@nasspub.com

**فروشگاه**: تهران - ضلع جنوب شرقی میدان انقلاب، شماره ۲۵ تلفن: ۶۶۴۰۵۳۷۲

وب سایت : www.nasspub.com



شابک ج ۱: ۹۷۸-۹۶۴-۴۱۰-۲۵۸-۵ ISBN: 978-964-410-258-5

شابک دوره: ۹۷۸-۹۶۴-۴۱۰-۲۶۰-۸ ISBN: 978-964-410-260-8

# مقدمهٔ دکتر یحیی تابش

الگوریتم یکی از مهم‌ترین مفاهیم علوم عقلی و ابداعات بشری است. هزاران سال است که ریاضی‌دانان، فلاسفه و دانشمندان علوم مختلف بر تفکر ساختار یافته (یا تفکر الگوریتمی) در توسعه‌ی دانش اتفاق نظر داشته‌اند. در سده‌های اخیر با رشد روزافزون ریاضی و علوم طبیعی، الگوریتم به شاخه‌ای مستقل و بالنده از علوم تبدیل شده‌است. با ابداع ماشین‌های محاسبه و ظهور رایانه، الگوریتم باز هم نقش برجسته‌تری در کاربرد و تکوین علوم پیدا کرده است.

تدریس الگوریتم‌ها به عنوان یک درس پایه برای دانشجویان علوم نظری و کاربردی و کامپیوتر از اواسط سده‌ی بیستم در دانشگاه‌ها هم رواج یافت. به دنبال این موضوع، کتاب‌های متعددی با موضوع «الگوریتم‌ها» تألیف شد و در دسترس دانشگاهیان قرار گرفت. مؤلفین این کتاب‌ها افرادی توانا و شایسته بودند، و کتاب‌هایشان در زمان خود مورد استقبال دانشجویان و اساتید قرار گرفت. با ارائه کتاب «مقدمه‌ای بر الگوریتم‌ها» در سال ۱۹۹۰، نوشته‌ی کرمِن و همکاران به سرعت توجه علاقه‌مندان به این اثر جلب شد. در مدت کوتاهی این کتاب به پرفروش‌ترین کتاب الگوریتم‌ها تبدیل شد و به عنوان مرجع درسی در بهترین دانشگاه‌های سراسر دنیا مورد استفاده قرار گرفت. در اکثر دانشگاه‌های ایران هم این کتاب جای خود را به عنوان مرجع دو درس «ساختمان داده‌ها و الگوریتم‌ها» و «طراحی و تحلیل الگوریتم‌ها» باز کرد، که به همین ضرورت از ویرایش‌های قبلی آن چند ترجمه به زبان فارسی صورت گرفته است.

کتابی که در دست دارید، ترجمه‌ای از آخرین ویرایش (ویرایش سوم) کتاب زبان انگلیسی با سال انتشار ۲۰۰۹ می‌باشد. در ترجمه‌ی کتاب سعی شده که در عین دقت در انتقال درست و علمی مطلب، سادگی و رسایی نثر حفظ شود. انتخاب معادل‌ها و واژه‌های مناسب یکی از دغدغه‌های هر مترجم کتاب‌های تخصصی است، که مترجم این کتاب به خوبی از عهده‌ی آن برآمده است.

به علت ماهیت خاص کتاب، مطالب آن نسبتاً سنگین است و در نتیجه خواندن آن می‌تواند برای دانشجو خسته‌کننده باشد. به همین علت با ابتکار ناشر، کتاب به صورت دو رنگ چاپ شده تا به سهولت در خواندن کتاب کمک کند.

طی سال‌ها تدریس در دانشگاه صنعتی شریف، متوجه شده‌ام که کتاب خوب در ارتقاء سطح آموزش از اهمیت ویژه‌ای برخوردار است. امیدوارم این کتاب هم بتواند به نوبه‌ی خود سهمی در بهبود توان علمی کشور داشته باشد.

دکتر یحیی تابش
عضو هیئت علمی دانشگاه صنعتی شریف

# مقدمه‌ی مترجم

از ابتدای پیدایش بشر، یکی از دغدغه‌های مهم انسان (و شاید مهم‌ترین آن‌ها) یافتن معنی، هدف، یا انگیزه‌ای برای زیستن روی کره‌ی خاکی بوده است. هدفی که تمامی ادیان بر روی آن تمرکز داشته‌اند، و متفکران مختلف در طول سالیان، راه‌کارهای گوناگونی برای دست‌یابی به آن ارائه کرده‌اند. چیزی که اکثر این ادیان و مکاتب در آن اتفاق نظر داشته‌اند، نقش خود فرد در تعیین هدف زندگی است، که از طرف بسیاری از پیروان این مکاتب نادیده گرفته می‌شود. هیچ مکتبی نمی‌تواند دستورالعملی جامع برای دست‌یابی تک‌تک افراد به تعالی ارائه دهد، چرا که هر کس در آفرینش منحصر به فرد است و باید نقشی متفاوت در این دنیا ایفا کند، و درک این نقش میسر نخواهد شد مگر از طریق تفکر و تعقل خود فرد (حقیقتی که تمامی ادیان الهی به آن تأکید دارند). وظیفه‌ی مکاتب فقط ارائه‌ی رویکرد کلی است، و در نهایت این شمایید که می‌توانید مسیر دقیق رشد خود را تعیین کنید.

حتماً تا به حال با افرادی مواجه شده‌اید که صرفاً به خاطر فشار اطرافیان، احساس ناتوانی یا دلایل دیگر، در زمینه‌ای تحصیل یا کار می‌کنند که به آن علاقه ندارند. این تلاش معمولاً نتیجه‌ای ندارد جز سرخوردگی و هدر رفتن استعدادهای درونی فرد. نیت از نگارش این مقدمه هم چیزی نیست جز واداشتن خودم و شما به لحظه‌ای تأمل. تأمل در مسیری که در پیش گرفته‌ایم و در آن گام برمی‌داریم. تأمل در مورد این که آیا این مسیر ما را به رشد و تعالی فردی نزدیک می‌کند یا نه. این که آیا تا کنون سعی کرده‌ایم به فلسفه‌ی وجودی خود پی ببریم، یا ما هم جزء افرادی هستیم که ناخواسته و بدون هدف در مسیری که دیگران برای ما تعیین کرده‌اند گام برمی‌داریم.

به هیچ وجه انتظار ندارم نظرات این حقیر را بپذیرید، بلکه فقط امیدوارم مشوقی باشم برای تفکر بیشتر در مورد راهی که تا کنون پیموده‌ایم و مسیری که پیش رو داریم، چرا که تفکر لازمه‌ی حصول آگاهی است و حرکت ناآگاهانه به مقصد درستی منتهی نخواهد شد. از این رو تقاضا دارم قبل از خواندن این کتاب (یا انجام هر کار دیگر) لحظه‌ای صادقانه در مورد اهدافی که در زندگی دارید بیاندیشید، و همواره سعی کنید فقط در راستای نیل به آن اهداف حرکت کنید. امیدوارم خواندن این کتاب برای شما گامی باشد در همین راستا.

✦✦✦

تمام دغدغه‌ی این‌جانب در برگردان این اثر، انتقال صحیح و روان مفاهیم ارائه شده در کتاب بوده است. در این راستا از ترجمه‌ی کلمه به کلمه به شدت پرهیز کرده، و نهایت تلاش خود را به کار برده‌ام که قبل از آغاز ترجمه‌ی هر بخش، ابتدا به دقت و با جزئیات تمام مطالب آن را مرور کنم تا بتوانم به درستی آن‌ها را به خواننده منتقل کنم. امیدوارم این تلاش نتیجه‌بخش بوده، و گامی باشد هر چند ناچیز در راستای آموزش علاقه‌مندان به مبحث الگوریتم.

در انتها از شما خواننده‌ی عزیز تقاضا دارم که در بهبود این اثر یاری خود را از ما دریغ نکنید. مسلماً این اثر بدون نقص نیست، ولی بهره‌مندی از دیدگاه‌های سازنده‌ی شما خوانندگان ما را به ارتقای سطح کیفی ترجمه در چاپ‌های آتی دلگرم می‌کند. در صورت مشاهده‌ی هر گونه خطا در نگارش یا ترجمه و یا داشتن هر گونه پیشنهاد یا انتقاد، لطفاً به آدرس‌های زیر ارسال فرمائید.

ali.dehghan.tarzeh@gmail.com
info@nasspub.com

علی دهقان طرزه
تابستان ۱۳۸۹

# مقدمهٔ نویسندگان

## مقدمه

قبل از به وجود آمدن کامپیوترها، الگوریتم‌ها وجود داشتند. اکنون که کامپیوترها به وجود آمده‌اند، وجود الگوریتم‌ها بسیار پررنگ‌تر از قبل شده است، چرا که در قلب محاسبات کامپیوتری قرار دارند.

کتاب حاضر، مقدمه‌ای است جامع برای آموزش مدرن الگوریتم‌های کامپیوتری. این کتاب الگوریتم‌های بسیاری معرفی و آن‌ها را به صورت عمیق بررسی می‌کند، ولی با این حال این معرفی طوری است که طراحی و تحلیل آن‌ها برای خوانندگان تمام سطوح قابل دسترس است. سعی شده است که بدون از دست رفتن عمق بررسی یا دقت ریاضی، توضیحات به صورت مقدماتی باشد تا برای تمام خوانندگان قابل فهم باشد.

هر فصل یک الگوریتم، یک تکنیک طراحی، یک حوزه‌ی کاربرد، یا یک موضوع مربوط را ارائه می‌کند. الگوریتم‌ها به زبان انگلیسی و به شکل سودوکد طراحی شده‌اند تا برای تمام کسانی که به صورت سطحی با برنامه‌نویسی آشنایی دارند، قابل فهم باشد. کل کتاب حاوی بیش از ۲۴۴ شکل است که نحوه‌ی اجرای الگوریتم‌ها را مشخص می‌کنند. از آن‌جایی که *کارایی* را به عنوان یک معیار طراحی می‌شناسیم، تحلیل دقیق زمان اجرای تمام الگوریتم‌ها را در کتاب گنجانده‌ایم.

ساختار متن کتاب طوری است که برای استفاده در واحدهای درسی الگوریتم‌ها یا ساختمان داده‌ها در مقاطع تحصیلی کارشناسی و یا کارشناسی ارشد قابل استفاده باشد. چون در کتاب بحث‌هایی در مورد مطالب مهندسی در طراحی الگوریتم‌ها، و همچنین جنبه‌های ریاضی وجود دارد، می‌توان از آن به عنوان خودآموز هم استفاده کرد.

## برای آموزگاران

این کتاب طوری طراحی شده است که هم فراگیر باشد و هم کامل. برای درس‌های مختلفی این کتاب را مفید خواهید یافت، از یک واحد ساختمان داده‌ها در دوره‌ی کارشناسی تا یک واحد الگوریتم در دوره‌ی کارشناسی ارشد. از آن‌جایی که محتویات این کتاب بسیار بیشتر از مطالب مورد نیاز در یک درس در طول یک ترم است، باید به آن به صورت یک «قفسه» نگاه کنید، که می‌توانید مطالبی را که بیشترین همخوانی را با واحد مورد نظر شما دارد، انتخاب کرده و آموزش دهید.

احتمالا مرتب کردن فصول به طور دلخواه، به طوری که با درس شما متناسب باشد، برای شما کار ساده‌ای خواهد بود. فصول نسبتا مستقل هستند، و بنابراین نیازی نیست که نگران وجود وابستگی‌های غیر منتظره از فصلی به فصل دیگر باشید. در هر فصل، ابتدا مطالب ساده‌تر و سپس مطالب سخت‌تر گنجانده شده‌اند، به همراه بخش‌بندی‌هایی که مرز مطالب را مشخص می‌کنند. در یک درس در دوره‌ی کارشناسی، ممکن است بخواهید که فقط از بخش‌های اول یک فصل استفاده کنید، و در یک درس در دوره‌ی کارشناسی ارشد، ممکن است بخواهید تمام فصل را پوشش دهید.

کتاب شامل بیش از ۹۲۰ تمرین و بیش از ۱۴۰ مسئله است. هر بخش با چند تمرین، و هر فصل با چند مسئله به پایان می‌رسد. معمولا تمرین‌ها سؤال‌های کوتاهی هستند که تسلط اولیه بر روی مباحث را آزمایش می‌کنند. بعضی از آن‌ها تمرین‌های خودآموز ساده هستند، در حالی که بعضی دیگر مقداری محکم‌تر هستند، و می‌توان از آن‌ها به عنوان تکلیف استفاده کرد. مسئله‌ها بررسی‌های موردی ظریف‌تری هستند که معمولا مطالب جدیدی را معرفی می‌کنند؛ آن‌ها معمولا شامل چندین سؤال هستند که دانشجو را طی مراحل مورد نیاز برای رسیدن به یک جواب کلی راهنمایی می‌کند.

بخش‌ها و تمرین‌هایی که ستاره (★) دارند، بیشتر برای دانشجویان دوره‌ی کارشناسی ارشد مناسب هستند تا دانشجویان دوره‌ی کارشناسی. یک بخش ستاره‌دار لزوما مشکل‌تر از یک بخش بدون ستاره نیست، ولی ممکن است به درک مطالب پیشرفته‌تری نیاز داشته باشد. متشابها، تمرین‌های ستاره‌دار ممکن است به یک پیش‌زمینه‌ی قبلی و یا خلاقیت بالاتر از سطح متوسط احتیاج داشته باشند.

## برای دانشجو

امیدواریم که این کتاب یک معرفی لذت‌بخش در زمینه‌ی الگوریتم‌ها برای شما فراهم کرده باشد. سعی کرده‌ایم که تمام الگوریتم‌ها را قابل فهم و جذاب توصیف کنیم. برای کمک به شما وقتی که به الگوریتم‌های مشکل یا ناآشنا بر می‌خورید، هر کدام از الگوریتم‌ها به صورت قدم به قدم توصیف شده است. همچنین توضیحات دقیقی از ریاضیات مورد نیاز برای تحلیل الگوریتم فراهم شده است. اگر از قبل با یک موضوع آشنایی دارید، ساختار فصل‌ها را طوری خواهید یافت که می‌توانید از بخش‌های مقدماتی صرف نظر کرده و به سرعت به بخش‌های پیش‌رفته‌تر برسید.

حجم این کتاب زیاد است، و احتمالا در کلاس شما فقط بخشی از مطالب آن پوشش داده خواهد شد. با این حال، سعی شده است که کتاب هم اکنون به عنوان یک کتاب درسی، و بعدا به عنوان یک مرجع ریاضی و یا یک راهنمایی مهندسی برای شما قابل استفاده باشد.

پیش‌زمینه‌های مورد نیاز برای این کتاب چیست؟

- باید مقداری تجربه‌ی برنامه‌نویسی داشته باشید. به خصوص، باید با رویه‌های بازگشتی و ساختمان‌های داده‌ی ساده مانند آرایه‌ها و لیست‌های پیوندی آشنا باشید.
- باید تا حدودی در اثبات کردن به وسیله‌ی استقرا مهارت داشته باشید. مقدار کمی از کتاب به پیش‌زمینه‌ای مقدماتی در حسابان نیاز دارد. غیر از آن، قسمت‌های I و VIII تمام تکنیک‌های ریاضی مورد نیاز را به شما آموزش می‌دهند.

## برای کاربران

دامنه‌ی وسیع مطالب این کتاب، آن را به یک راهنمای مناسب بر روی الگوریتم‌ها تبدیل کرده است. از آن جایی که مطالب هر فصل نسبتا مستقل است، می‌توانید فقط بر روی مباحثی تمرکز کنید که مورد نیاز شما هستند.

اکثر الگوریتم‌هایی که در مورد آن‌ها بحث می‌کنیم، در عمل کاربردهای فراوانی دارند. بنابراین در کتاب به مسائل پیاده‌سازی و مهندسی هم اشاره شده است. معمولا برای الگوریتم‌های کمی که فقط کاربرد نظری دارند، جایگزین‌های کاربردی هم ارائه خواهیم کرد.

اگر می‌خواهید هر یک از الگوریتم‌ها را پیاده‌سازی کنید، ترجمه‌ی سودوکد به زبان برنامه‌نویسی مورد نظر خود را کاری نسبتا سرراست خواهید یافت. سودوکدها طوری طراحی شده‌اند که الگوریتم‌ها را به صورت واضح و مختصر شرح دهند. به همین دلیل در آن‌ها به مدیریت خطا (error-handling) و مسائل مهندسی نرم‌افزار که به فرض‌های خاص در مورد محیط برنامه‌نویسی نیاز دارند، اشاره نشده است. سعی شده است که توصیف هر الگوریتم ساده و مستقیم باشد تا خصوصیات منحصر به فرد یک زبان برنامه‌نویسی خاص، شفافیت آن را از بین نبرد.

# فهرست مطالب

# بخش اول مبانی

شامل فصل‌های :

۱ نقش الگوریتم‌ها در محاسبات
۲ آغاز
۳ رشد توابع
۴ بازگشت‌ها
۵ آنالیز احتمالات و الگوریتم‌های تصادفی

در این بخش، با نحوه‌ی تفکر در مورد طراحی و تحلیل الگوریتم‌ها آشنا خواهید شد. هدف از این فصل این است که شما را مختصرا با روش توصیف الگوریتم‌ها، بعضی از استراتژی‌های طراحی که بعدا در این کتاب از آن‌ها استفاده خواهیم کرد، و بسیاری از ایده‌های اصلی استفاده شده در تحلیل الگوریتم‌ها آشنا کند. فصل‌های بعدی در این کتاب از مطالب این فصل استفاده خواهند کرد.

فصل ۱ به بازبینی الگوریتم‌ها و نقش آن‌ها در سیستم‌های محاسباتی مدرن می‌پردازد. این فصل الگوریتم‌ها را تعریف، و چند مثال از آن‌ها ارائه می‌کند. همچنین حالتی را نشان می‌دهد که در آن الگوریتم‌ها یک تکنولوژی هستند، درست مانند سخت‌افزار سریع، واسط گرافیکی کاربر، سیستم‌های شیئ گرا، و شبکه‌ها.

در فصل ۲ اولین الگوریتم‌های خود را خواهیم دید، که مسئله‌ی مرتب‌سازی $n$ عدد را حل می‌کنند. این الگوریتم‌ها به زبان سودوکد نوشته شده‌اند، که با این که مستقیما به هر زبان برنامه‌نویسی قابل تبدیل نیست، ولی به اندازه‌ی کافی ساختار الگوریتم را روشن می‌کند که یک برنامه‌نویس می‌تواند آن را با زبان دلخواه خود پیاده‌سازی کند. الگوریتم‌های مرتب‌سازی که در این جا آن‌ها را بررسی می‌کنیم عبارتند از مرتب‌سازی درجی، که از یک رویکرد پیش‌رونده استفاده می‌کند، و مرتب‌سازی ادغامی، که از یک تکنیک بازگشتی به نام «تقسیم و حل» استفاده می‌کند. با این که زمان مورد نیاز هر دو با رشد $n$ افزایش می‌یابد، ولی سرعت رشد در دو الگوریتم متفاوت است. این زمان‌های اجرا را در فصل ۲ تعیین، و یک نماد مناسب برای توصیف آن‌ها ارائه می‌کنیم.

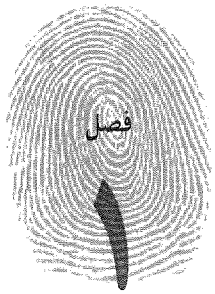
این نماد در فصل ۳ به طور دقیق تعریف می‌شود، که در آن جا به آن نماد حدی می‌گوییم. این فصل با نمادهای حدی متعددی آغاز می‌شود، که از آن‌ها برای تعیین کران بالا و/یا پایین زمان اجرای

الگوریتم‌ها استفاده می‌کنیم. نتیجه‌ی اصلی فصل ۳ ارائه‌ی نمادهای ریاضی است. هدف آن بیشتر این است که نحوه‌ی استفاده‌ی شما از نمادها شبیه این کتاب باشد، تا آموختن مفاهیم ریاضی جدید.

فصل ۴ متد تقسیم و حل معرفی شده در فصل ۲ را با عمق بیشتری بررسی می‌کند. به طور خاص، فصل ۴ حاوی متدهایی است برای حل رابطه‌های بازگشتی، که برای توصیف زمان اجرای الگوریتم‌های بازگشتی مناسب است. یک تکنیک قدرتمند «متد اصلی» است، که از آن می‌توان برای حل رابطه‌های بازگشتی حاصل از الگوریتم‌های تقسیم و حل استفاده کرد. بخش اعظم فصل ۴ به اثبات صحت متد اصلی اختصاص دارد، با این حال می‌توانید بدون از خواندن آن صرف‌نظر کنید.

فصل ۵ تحلیل احتمالاتی و الگوریتم‌های تصادفی را معرفی می‌کند. معمولا از تحلیل احتمالاتی برای تعیین زمان اجرای یک الگوریتم در حالت‌هایی استفاده می‌کنیم که به خاطر وجود یک توزیع احتمالاتی ذاتی، ممکن است زمان اجرای آن برای ورودی‌های با اندازه‌ی ثابت تفاوت داشته باشد. در بعضی موارد، فرض می‌کنیم که ورودی‌ها یک توزیع شناخته شده دارند، و بنابراین از زمان اجرا بر روی تمام ورودی‌های ممکن متوسط می‌گیریم. در موارد دیگر، توزیع احتمالاتی حاصل از ورودی نیست، بلکه حاصل از انتخاب‌های تصادفی است که در حین اجرای الگوریتم انجام می‌شود. الگوریتمی که رفتار آن علاوه بر ورودی‌ها، توسط مقادیر تصادفی انتخاب شده توسط یک تولید کننده‌ی اعداد تصادفی تعیین می‌شود، یک الگوریتم تصادفی نام دارد. می‌توانیم از الگوریتم‌های تصادفی استفاده کنیم و یک توزیع احتمالاتی خاص به ورودی‌ها بدهیم – که تضمین می‌کند که هیچ ورودی خاصی نمی‌تواند همیشه باعث کارایی پایین شود – و یا حتی میزان خطای الگوریتم‌هایی را که در یک محدوده‌ی خاص مجاز به تولید نتایج نادرست هستند، پایین بیاوریم.

پیوست‌های الف–پ حاوی مطالب ریاضی دیگری هستند که ممکن است حین خواندن این کتاب آن‌ها را مفید بیابید. احتمالا شما اکثر مطالب این پیوست‌ها را قبل از خواندن این کتاب دیده‌اید (ولی نمادهایی قراردادی خاصی که در این کتاب از آن‌ها استفاده می‌کنیم ممکن است با چیزی که شما قبلا دیده‌اید متفاوت باشند)، بنابراین باید به پیوست‌ها به چشم مطالب مرجع نگاه کنید. از سوی دیگر، ممکن است بسیاری از مطالب بخش ۱ برای شما نا آشنا باشد. تمام فصل‌ها در بخش ۱ و پیوست‌ها با دید آموزشی نگارش شده‌اند.




# نقش الگوریتم‌ها در محاسبات

الگوریتم چیست؟ چرا آموختن الگوریتم‌ها مفید است؟ نقش الگوریتم‌ها در رابطه با تکنولوژی‌های کامپیوتری دیگر چیست؟ در این فصل به این سؤال‌ها جواب خواهیم داد.

## ۱-۱ الگوریتم‌ها

به طور غیر رسمی، به هر رویه‌ی محاسباتی خوش تعریف که یک یا چند مقدار را به عنوان *ورودی* می‌گیرد، و یک یا چند مقدار را به عنوان *خروجی* بازمی‌گرداند *الگوریتم* (algorithm) گفته می‌شود. بنابراین، الگوریتم دنباله‌ای از مراحل محاسباتی است که ورودی را به خروجی تبدیل می‌کنند.

علاوه بر آن، می‌توان الگوریتم را به چشم ابزاری برای حل *مسائل محاسباتی* خوش تعریف دید. به طور کلی، صورت مسئله رابطه‌ی بین ورودی و خروجی را مشخص می‌کند، و الگوریتم یک رویه‌ی محاسباتی برای رسیدن به این رابطه فراهم می‌کند.

به عنوان مثال، ممکن است بخواهیم دنباله‌ای از اعداد را به صورت نزولی مرتب کنیم. این مسئله به کرات در عمل پیش می‌آید، و زمینه‌ی مناسبی برای معرفی بسیاری از تکنیک‌های طراحی استاندارد و ابزارهای تحلیل فراهم می‌آورد. در زیر به طور رسمی *مسئله‌ی مرتب‌سازی* را تعریف می‌کنیم:

- *ورودی:* دنباله‌ای از $n$ عدد $\langle a_1, a_2, ..., a_n \rangle$.
- *خروجی:* یک جایگشت (بازمرتب‌سازی) $\langle a'_1, a'_2, ..., a'_n \rangle$ از دنباله‌ی ورودی به طوری که داشته باشیم $a'_1 \le a'_2 \le ... \le a'_n$ .

برای مثال، اگر ورودی دنباله‌ی ⟨۳۱,۴۱,۵۹,۲۶,۴۱,۵۸⟩ باشد، یک الگوریتم مرتب‌سازی دنباله‌ی ⟨۲۶,۳۱,۴۱,۴۱,۵۸,۵۹⟩ را به عنوان خروجی بازمی‌گرداند. به این دنباله‌ی ورودی یک *نمونه* (instance)

از مسئله‌ی مرتب‌سازی گفته می‌شود. به طور کلی، نمونه شامل ورودیی از مسئله (که شرایط آن را ارضا می‌کند) و برای حل مسئله مورد نیاز است، می‌باشد.

از آن جایی که برنامه‌های بسیاری از مرتب‌سازی به عنوان یک مرحله‌ی میانی استفاده می‌کنند، در حال حاضر الگوریتم‌های متنوعی برای این کار در اختیار ما قرار دارد. اینکه کدام الگوریتم برای یک برنامه‌ی خاص مناسب‌تر است، به عوامل زیادی بستگی دارد، از جمله تعداد اقلامی که باید مرتب شوند، ترتیب فعلی اقلام (ممکن است ترتیب اولیه تقریباً مرتب باشد)، محدودیت‌های احتمالی بر روی مقدار اقلام، و نوع دستگاه ذخیره‌سازی که برای مرتب‌سازی استفاده می‌شود (حافظه‌ی اصلی کامپیوتر، انواع دیسک‌ها و یا حتی نوار).

الگوریتمی را صحیح گویند که برای تمام نمونه‌های ورودی، خروجی صحیح تولید کند. به بیان دیگر الگوریتم صحیح مسئله‌ی محاسباتی مورد نظر را *حل می‌کند*. الگوریتمی نادرست است که برای بعضی ورودی‌ها هرگز پایان نیابد، و یا جواب غلط تولید کند. بر خلاف تصور الگوریتم‌های غلط در بعضی موارد می‌توانند مفید باشند، اگر بتوان نرخ خطای آن‌ها را کنترل کرد. نمونه‌ای از این الگوریتم‌ها را در فصل ۳۱ برای یافتن اعداد اول بسیار بزرگ خواهیم دید. با این حال، به طور معمول فقط به الگوریتم‌های صحیح توجه داریم.

یک الگوریتم را می‌توان به زبان روزمره، به صورت یک برنامه‌ی کامپیوتری، و یا حتی به صورت یک طراحی سخت‌افزاری توصیف کرد. تنها نکته‌ای که باید رعایت شود این است که توصیف الگوریتم باید توضیح دقیقی از رویه‌ی محاسباتی لازم برای حل مسئله فراهم کند.

## چه نوع مسائلی با الگوریتم حل می‌شوند؟

مرتب‌سازی فقط یکی از مسائلی است که برای آن الگوریتم‌هایی طراحی شده است. (مسلما با دیدن حجم کتاب حاضر به این مسئله پی برده‌اید!) کاربردهای الگوریتم در همه جا دیده می‌شود.

- پروژه‌ی ژنوم انسان در رسیدن به این اهداف گام‌های بزرگی برداشته است: شناسایی تمامی ۱۰۰,۰۰۰ ژن در DNA انسان، تشخیص دنباله‌ی ۳ میلیارد جفت باز شیمیایی که DNA انسان را می‌سازند، ذخیره‌ی این اطلاعات در پایگاه داده، و طراحی ابزاری برای تحلیل این داده‌ها. هر کدام از این مراحل به الگوریتم‌های پیچیده‌ای نیاز دارند. با اینکه حل این مسائل از محدوده‌ی کتاب حاضر خارج است، ایده‌های بسیاری از فصل‌های این کتاب در حل این مسائل بیولوژیکی به کار رفته‌اند، که دانشمندان را قادر می‌سازند که علاوه بر حل مسائل، از منابع هم به شکلی با صرفه استفاده کنند. با استخراج اطلاعات بیشتر از تکنیک‌های آزمایشگاهی، در وقت (هم نیروی انسانی، هم ماشین‌ها) و پول صرفه‌جویی خواهد شد.
- اینترنت مردم سراسر دنیا را قادر می‌سازد که با سرعت زیاد به حجم زیادی از اطلاعات دسترسی داشته باشند. به کمک الگوریتم‌های هوشمند، سایت‌های اینترنتی قادر به مدیریت و پردازش حجم زیادی از داده‌ها هستند. نمونه‌هایی از مسائلی که باید حل شوند عبارتند از: یافتن مسیرهای مناسب برای جابه‌جایی داده‌ها (تکنیک‌هایی برای حل این نوع مسائل را در

فصل ۲۴ خواهیم دید)، و طراحی یک موتور جستجو برای یافتن سریع صفحه‌هایی با محتوای مورد نظر (تکنیک‌های مرتبط را در فصل‌های ۱۱ و ۳۲ خواهیم دید).

- تجارت الکترونیک ما را قادر می‌سازد کالاها و خدمات را از طریق اینترنت مبادله کنیم. برای استفاده‌ی وسیع از تجارت الکترونیک، باید اطلاعاتی نظیر شماره‌ی کارت اعتباری، گذرواژه‌ها و داده‌های بانکی محرمانه باقی بمانند. کدگذاری کلید عمومی (public-key cryptography) و امضاهای دیجیتال (فصل ۳۱) از فناوری‌های اصلی مورد استفاده برای این هدف هستند که بر مبنای الگوریتم‌های عددی (numerical algorithms) و نظریه‌ی اعداد (number theory) بنا شده‌اند.
- در صنعت و دیگر فعالیت‌های اقتصادی، تخصیص‌دهی منابع کم‌یاب باید به بهینه‌ترین شکل ممکن انجام شود. یک کمپانی نفتی می‌خواهد بداند فروش محصولات در کجا سودآورتر است. یک کاندیدای ریاست جمهوری می‌خواهد بداند هزینه کردن برای تبلیغات در کجا مقرون به صرفه است، و شانس او را برای پیروزی در انتخابات افزایش می‌دهد. یک شرکت هواپیمایی می‌خواهد طوری خدمه را به پروازهای مختلف بگمارد که کمترین هزینه را داشته باشد، تمام پروازها پوشش داده شوند، و قوانین دولتی مربوط به خدمه‌ی پرواز رعایت شده باشند. تمام این مثال‌ها را می‌توان به کمک برنامه‌ریزی خطی (linear programming) حل کرد. در مورد برنامه‌ریزی خطی در فصل ۲۹ بحث خواهد شد.

جزئیات بعضی از این مسائل خارج از حوصله‌ی این کتاب است. با این حال در این جا تکنیک‌هایی ارائه خواهد شد که برای حل مسائل مشابه، مناسبند. همچنین در این کتاب، نحوه‌ی حل بسیاری از مسائل کاربردی را خواهیم دید، مانند نمونه‌های زیر:

- نقشه‌ای داریم از جاده‌ها، که در آن فاصله‌ی بین هر دو تقاطع مجاور مشخص شده است. هدف یافتن کوتاهترین مسیر از تقاطعی خاص به تقاطع دیگر است. تعداد مسیرهای موجود می‌تواند بسیار زیاد باشد. چطور می‌توان فهمید که کدام یک از این مسیرها، کوتاه‌ترین است؟ در این جا، نقشه‌ی جاده‌ها (که مدلی از خود جاده‌ها است) را به صورت یک گراف مدل می‌کنیم (با گراف‌ها در فصل ۱۰ و پیوست ب آشنا خواهیم شد)، و کوتاه‌ترین مسیر از یک رأس به یک رأس دیگر را در این گراف پیدا می‌کنیم. در فصل ۲۴ خواهیم دید که چطور می‌توان این مسئله را با صرف زمان قابل قبول حل کرد.
- دو دنباله‌ی مرتب از نمادها به صورت $X = \langle x_1, x_2, ..., x_m \rangle$ و $Y = \langle y_1, y_2, ..., y_n \rangle$ داریم، و می‌خواهیم بلندترین زیردنباله‌ی مشترک $X$ و $Y$ را بیابیم. یک زیردنباله از $X$ عبارت است از خود $X$ که بعضی از عناصر آن (یا همه یا هیچ یک از آن‌ها) حذف شده‌اند. مثلاً $\langle B, C, E, G \rangle$ می‌تواند یک زیردنباله از $\langle A, B, C, D, E, F, G \rangle$ باشد. طول بلندترین زیردنباله‌ی مشترک $X$ و $Y$ معیاری است برای تعیین میزان شباهت این دو دنباله. برای مثال اگر دو دنباله، دو رشته‌ی DNA باشند، در صورتی که زیردنباله‌ی مشترک بلندی داشته باشند، می‌توانیم آن‌ها را مشابه در نظر بگیریم. اگر $X$ دارای $m$ نماد و $Y$ دارای $n$ نماد باشد، آن گاه

$X$ و $Y$ به ترتیب $2^m$ و $2^n$ زیردنباله دارند. انتخاب تمام زیردنباله‌های $X$ و $Y$ و مقایسه‌ی آن‌ها با هم می‌تواند بسیار زمان‌بر باشد، مگر این که $n$ و $m$ خیلی کوچک باشند. در فصل ۱۵ خواهیم دید که چطور می‌توان با استفاده از یک تکنیک کلی به نام برنامه‌ریزی پویا، این مسئله را به صورت بسیار کارامدتر حل کرد.

- یک طرح مکانیکی به صورت مجموعه‌ای از قطعات داریم، که در آن هر قطعه ممکن است حاوی قطعات دیگر باشد، و می‌خواهیم قطعات را طوری لیست کنیم که هر قطعه قبل از تمام قطعاتی بیاید که از آن استفاده می‌کند. اگر در این طرح از $n$ قطعه استفاده شده باشد، در این صورت $n!$ ترتیب ممکن وجود دارد، که در آن $n!$ نشان‌دهنده‌ی تابع فاکتوریل است. چون تابع فاکتوریل حتی از توابع نمایی هم سریع‌تر رشد می‌کند، نخواهیم توانست تمام ترتیب‌های ممکن را بسازیم و در آن‌ها این خصوصیت را بررسی کنیم (مگر این که تعداد قطعات کم باشد). این مسئله، نمونه‌ای است از مرتب‌سازی توپولوژیکی، که در فصل ۲۲ نحوه‌ی حل بهینه‌ی آن را خواهیم دید.
- $n$ نقطه در یک صفحه داریم، و می‌خواهیم چندضلعی محاطی (convex hull) برای این نقطه‌ها پیدا کنیم. چندضلعی محاطی، کوچکترین چندضلعی محدبی است که تمام نقاط را در بر می‌گیرد. برای درک بهتر، می‌توانیم هر نقطه را به صورت یک میخ در نظر بگیریم که سر آن از یک صفحه‌ی چوبی بیرون زده است. در این صورت، چندضلعی محاطی یک کش محکم خواهد بود که تمام میخ‌ها را در بر گرفته است. هر میخی که کش با آن در تماس است، یک رأس از چندضلعی محدب خواهد بود. (شکل ۳۳-۶ را ببینید.) هر کدام از $2^n$ زیرمجموعه‌ی نقاط ممکن است مجموعه‌ی رئوس چندضلعی محدب باشد. دانستن مجموعه‌ی نقاط چندضلعی محدب به تنهایی کافی نیست، چرا که باید ترتیب آن‌ها را نیز بدانیم. بنابراین انتخاب‌های زیادی برای چندضلعی محدب وجود دارد. در فصل ۳۳ دو روش مناسب برای یافتن چندضلعی محدب خواهیم آموخت.

این لیست‌ها به هیچ وجه کامل نیستند (که باز هم احتمالاً این مسئله را از وزن این کتاب حدس زده‌اید)، ولی دو ویژگی مشترک بین بسیاری از مسائل الگوریتمی جالب را نشان می‌دهند:

۱. معمولاً راه‌حل‌های زیادی برای این گونه مسائل وجود دارد، که اکثر قریب به اتفاق آن‌ها مسئله را حل نمی‌کنند. پیدا کردن راه حل صحیح، یا «بهترین» راه حل، می‌تواند بسیار دشوار باشد.
۲. این مسائل کاربردهای زیادی دارند. از لیست بالا، مسئله‌ی کوتاه‌ترین مسیر نمونه‌ی بسیار مناسبی است. یک موسسه‌ی حمل و نقل علاقه دارد کوتاه‌ترین مسیرها را در یک شبکه‌ی جاده‌ای و یا ریلی بداند، زیرا مسیر کوتاه‌تر هزینه‌ی سوخت و کار کمتری برای مؤسسه در بر دارد. یا یک مسیریاب اینترنتی برای رساندن سریع بسته‌ها نیاز دارد از کوتاه‌ترین مسیرها اطلاع داشته باشد. یا کسی که می‌خواهد از نیویورک به بوستون برود، ممکن است بخواهد اطلاعات مسیر مناسب را از یک وب سایت دریافت کند، و یا هنگام رانندگی از GPS کمک بگیرد.

هر مسئله‌ای که توسط الگوریتم‌ها حل می‌شود، لزوماً مجموعه راه حل‌های احتمالی روشنی ندارد. برای مثال، فرض کنید مجموعه‌ای از اعداد داریم که نشان دهنده‌ی نمونه‌های یک سیگنال هستند، و می‌خواهیم تبدیل فوریه‌ی گسسته‌ی این نمونه‌ها را بیابیم. تبدیل فوریه‌ی گسسته دامنه‌ی زمانی را به دامنه‌ی فرکانسی تبدیل کرده، مجموعه‌ای از ضرایب عددی به ما می‌دهد، به طوری که می‌توانیم قدرت فرکانس‌های مختلف سیگنال داده شده را بیابیم. تبدیل فوریه‌ی گسسته علاوه بر این که در قلب پردازش سیگنال قرار دارد، کاربردهایی هم در فشرده‌سازی داده‌ها و ضرب چندجمله‌ای‌ها و اعداد بزرگ دارد. فصل ۳۰ یک الگوریتم بهینه به نام تبدیل فوریه‌ی سریع (که معمولاً با نام FFT خوانده می‌شود) برای این مسئله ارائه می‌کند و همچنین طرح یک مدار سخت‌افزاری برای محاسبه‌ی FFT به دست می‌دهد.

## ساختمان‌های داده

در این کتاب ساختمان‌های داده‌ی مختلفی خواهیم دید. یک *ساختمان داده* (data structure) روشی است برای ذخیره و سازمان‌دهی داده‌ها با هدف تسهیل در دسترسی و تغییر آن‌ها. هیچ ساختمان داده‌ای وجود ندارد که برای تمام اهداف مناسب باشد، و به همین خاطر مهم است نقاط ضعف و قوت هر کدام از آن‌ها را بدانیم.

## تکنیک‌ها

با این که می‌توانید از این کتاب به عنوان یک مرجع استفاده کنید، ممکن است به مسئله‌ای برخورد کنید که نتوانید برای آن الگوریتم آماده‌ای بیابید (مانند بسیاری از تمرین‌ها و مسئله‌های همین کتاب!). این کتاب به شما تکنیک‌هایی برای طراحی و تحلیل الگوریتم‌ها خواهد آموخت، که خودتان می‌توانید به کمک آن الگوریتم‌هایی طراحی کرده، درستی آن‌ها را اثبات، و میزان کارایی آن‌ها را درک کنید. فصل‌های مختلف، جنبه‌های مختلف حل مسئله‌های الگوریتمی را پوشش می‌دهند. بعضی از فصل‌ها به مسئله‌های خاص می‌پردازند، مثلاً یافتن میانه و آمارهای ترتیبی در فصل ۹، محاسبه‌ی درخت پوشای کمینه در فصل ۲۳، و تعیین شار بیشینه در یک شبکه در فصل ۲۶. در فصل‌های دیگر تکنیک‌های دیگری ارائه می‌شود، مثلاً تکنیک تقسیم و حل در فصل ۴، برنامه‌ریزی پویا در فصل ۱۵، و تحلیل سرشکن در فصل ۱۷.

## مسائل سخت

قسمت اعظم این کتاب در مورد الگوریتم‌های کارامد است. مقیاس معمول ما برای کارایی، سرعت است، یعنی مدت زمانی که طول می‌کشد یک الگوریتم به نتیجه برسد. با این حال، مسائلی هستند که برای آن‌ها هیچ الگوریتم کارامدی یافت نشده است. در فصل ۳۴ در مورد زیرمجموعه‌ای جذاب از این مسائل، به نام NP–کامل‌ها ، بحث خواهد شد.

چرا مسائل NP–کامل جالب توجه هستند؟ اول، با این که هیچ الگوریتم کارامدی برای آن‌ها پیدا

نشده است، تا به حال کسی نتوانسته ثابت کند که برای حل آن‌ها الگوریتم کارامد وجود ندارد. به عبارت دیگر، کسی نمی‌داند برای این مسائل الگوریتم کارایی وجود دارد یا خیر. دوم، مجموعه‌ی مسائل NP-کامل این خصوصیت را دارند که اگر یک الگوریتم کارامد برای یکی از آن‌ها وجود داشته باشد، آن گاه برای تمام آن‌ها الگوریتم‌های کارامد وجود خواهد داشت. رابطه‌ی میان مسائل NP-کامل، عدم وجود الگوریتم برای آن‌ها را تبدیل به مسئله‌ای وسوسه‌انگیز می‌کند. سوم، بسیاری از مسائل NP-کامل شبیه (ولی نه دقیقا مشابه) مسائلی هستند که برای آن‌ها الگوریتم کارامد وجود دارد. یک تغییر کوچک در صورت مسئله، تغییری بسیار بزرگ در کارایی بهترین جواب موجود ایجاد می‌کند.

آشنایی با مسائل NP-کامل از این رو مهم است که بعضی از آن‌ها به کرات در مسائل کاربردی ظاهر می‌شوند. اگر از شما خواسته شود که برای یک مسئله‌ی NP-کامل الگوریتمی کارامد بیابید، احتمالاً زمان زیادی را صرف این جستجوی بیهوده خواهید کرد. اگر بتوانید نشان دهید که این مسئله NP-کامل است، می‌توانید در عوض زمان خود را صرف یافتن یک راه حل خوب (نه لزوما بهترین راه حل) بکنید.

به عنوان مثالی ملموس، یک شرکت حمل و نقل را در نظر بگیرید که یک انبار مرکزی دارد. هر روز کامیون‌ها در این انبار مرکزی بارگیری شده و به نقاط مختلف فرستاده می‌شوند تا محموله‌ها را به مقصد برساند. در پایان روز کامیون‌ها باید به انبار مرکزی برگردند تا برای بارگیری روز بعد آماده باشند. برای کاهش در هزینه‌ها، شرکت می‌خواهد ترتیبی برای رساندن محموله‌ها در نظر بگیرد که مجموع مسافت طی شده توسط هر کامیون کمینه شود. این مسئله، همان مسئله‌ی معروف «فروشنده‌ی دوره‌گرد» (traveling-salesman problem) است، و مسئله‌ای NP-کامل . این مسئله هیچ الگوریتم کارامد شناخته شده‌ای ندارد. با این حال، با در نظر گرفتن چند فرض، الگوریتم‌های کارایی وجود دارند که جوابی که تولید می‌کنند با جواب بهینه فاصله‌ی چندانی ندارد. در فصل ۳۵ در مورد این «الگوریتم‌های تقریبی» بحث خواهد شد.

## محاسبه‌ی موازی

برای سال‌هایی طولانی می‌توانستیم روی افزایش یکنواخت سرعت پردازنده‌ها حساب کنیم. ولی محدودیت‌های فیزیکی مانعی اساسی برای افزایش همیشگی سرعت پردازنده‌ها ایجاد می‌کنند: چون چگالی توان نسبت به سرعت پردازنده به صورت فراخطی افزایش می‌یابد، احتمال ذوب شدن تراشه در سرعت‌های به اندازه‌ی کافی بالا وجود دارد. بنابراین برای انجام پردازش بیشتر در واحد زمان، تراشه‌ها طوری طراحی می‌شوند که به جای یک «هسته»، چندین «هسته‌ی» پردازش داشته باشند. می‌توان این پردازنده‌های چندهسته‌ای را به چندین پردازنده‌ی متوالی بر روی یک تراشه تشبیه کرد؛ به عبارت دیگر، آن‌ها نوعی «پردازنده‌ی موازی» هستند. برای دست‌یابی به بالاترین کارایی از پردازنده‌های چندهسته‌ای، باید الگوریتم‌هایی طراحی کنیم که ذاتا موازی عمل می‌کنند. در فصل ۲۷ مدلی برای الگوریتم‌های «چند ریسمانی» خواهیم دید که از مزیت وجود چند هسته روی یک تراشه

استفاده می‌کند. این مدل از دید تئوری مزیت‌هایی دارد، و مبنای چندین برنامه‌ی موفق کامپیوتری است، از جمله برنامه‌ای برای قهرمانی در شطرنج.

## تمرین‌ها

۱-۱-۱ یک مثال کاربردی در دنیای واقعی بیابید که به مرتب‌سازی نیاز داشته باشد، و یا مثالی که به یافتن چندضلعی محاطی نیاز داشته باشد.

۱-۱-۲ علاوه بر سرعت، چه عامل‌های دیگری در دنیای واقعی می‌توانند برای کارایی مهم باشند؟

۱-۱-۳ یک ساختمان داده را که قبلا دیده‌اید انتخاب کنید و در مورد نقاط قوت و ضعف آن بحث کنید.

۱-۱-۴ مسائل کوتاه‌ترین مسیر و فروشنده‌ی دوره‌گرد که در بالا ارائه شد چه شباهت‌هایی با هم دارند؟ تفاوت‌های آن‌ها چیست؟

۱-۱-۵ یک مسئله در دنیای واقعی بیابید که در آن فقط بهترین جواب مناسب است. سپس مسئله‌ای بیابید که در آن جوابی که «تقریباً» بهترین است هم برای حل مشکل کافی باشد.

## ۱-۲ الگوریتم به عنوان یک فناوری

فرض کنید کامپیوترها بینهایت سریع بودند، و حافظه‌ی آن‌ها نیز رایگان بود. آیا در این صورت دلیلی برای یادگیری الگوریتم‌ها وجود داشت؟ بله، حتی اگر فقط برای این باشد که نشان دهیم روش ما در جایی پایان می‌یابد و جواب درست تولید می‌کند.

اگر کامپیوترها بینهایت سریع بودند، هر روش درستی برای حل مسئله کافی بود. احتمالاً جوابی را انتخاب می‌کردیم که در چارچوب اصول مهندسی نرم‌افزار باشد (مثلاً طراحی خوبی داشته و مستند شده باشد). ولی اغلب ساده‌ترین روش را انتخاب می‌کنیم.

کامپیوترها ممکن است سریع باشند، ولی بینهایت سریع نیستند، و حافظه هم ممکن است ارزان باشد، ولی رایگان نیست. بنابراین، زمان محاسبه و همچنین فضای حافظه منابعی محدود به شمار می‌روند. از این منابع باید با درایت استفاده کرد، و الگوریتم‌هایی که از نظر زمان یا حافظه کارامد باشند، در استفاده‌ی درست از منابع به ما کمک می‌کنند.

### کارایی

معمولاً الگوریتم‌های مختلفی که برای حل یک مسئله‌ی خاص طراحی شده‌اند، در کارایی با یکدیگر تفاوت‌های چشمگیری دارند. این تفاوت‌ها می‌توانند بسیار قابل توجه‌تر از تفاوت‌های سخت‌افزاری و نرم‌افزاری باشند.

به عنوان مثال، در فصل ۲ دو الگوریتم برای مرتب‌سازی خواهیم دید. الگوریتم اول، که مرتب‌سازی درجی نام دارد، $n$ داده را تقریباً در زمان $c_1 n^2$، داده را مرتب می‌کند، که در آن $c_1$ ثابتی مستقل از $n$ است. الگوریتم دوم، که به مرتب‌سازی ادغامی معروف است، تقریباً در زمان $c_2 n \lg n$ مرتب‌سازی را انجام می‌دهد، که در آن $\lg n$ همان $\log_2 n$ است، و $c_2$ نیز ثابتی مستقل از $n$ است. معمولاً ثابت مرتب‌سازی درجی کوچک‌تر از ثابت مرتب‌سازی ادغامی است، یعنی $c_1 < c_2$. در اینجا می‌خواهیم نشان دهیم که اهمیت ثابت‌ها در زمان اجرای الگوریتم بسیار کمتر از وابستگی به اندازه‌ی ورودی $n$ است. در جایی که مرتب‌سازی ادغامی عامل زمانی $\lg n$ دارد، عامل زمانی مرتب‌سازی درجی برابر با $n$ (که بسیار بزرگ‌تر است) خواهد بود. (برای مثال اگر $n = ۱۰۰۰$ آن گاه $\lg n$ تقریباً برابر است با ۱۰، و وقتی $n$ یک میلیون باشد، $\lg n$ کمی کم‌تر از ۲۰ خواهد بود.) با این که مرتب‌سازی درجی برای ورودی‌های کوچک از مرتب‌سازی ادغامی سریع‌تر است، زمانی که اندازه‌ی ورودی $n$ به اندازه‌ی کافی بزرگ شود، تأثیر عامل $\lg n$ در مرتب‌سازی ادغامی در مقابل $n$ در مرتب‌سازی درجی بسیار بیشتر از تفاوت ثابت‌ها در این دو تابع می‌شود. ثابت $c_1$ هر اندازه از $c_2$ کوچک‌تر باشد، باز هم نقطه‌ای است که در آن سرعت مرتب‌سازی ادغامی از مرتب‌سازی درجی بیشتر خواهد شد.

به عنوان یک مثال واضح، فرض کنید دو کامپیوتر داریم، یکی سریع (کامپیوتر A) و یکی کند (کامپیوتر B). مرتب‌سازی درجی را بر روی کامپیوتر سریع، و مرتب سازی ادغامی را بر روی کامپیوتر کند اجرا می‌کنیم. هر دوی این کامپیوترها باید ده میلیون عدد را مرتب کنند. (با این که ده میلیون ممکن است زیاد به نظر برسد، ولی اگر از اعداد هشت بیتی استفاده کنیم، آن گاه ورودی حدود ۸۰ مگابایت فضا اشغال می‌کند، که حتی بر روی حافظه‌ی یک لپ‌تاپ ارزان قیمت هم چندین کپی از آن جای می‌گیرد.) فرض کنید کامپیوتر A در هر ثانیه ده میلیارد دستورالعمل اجرا می‌کند (سریع‌تر از هر کامپیوتر رویه‌ای موجود در زمان نوشتن کتاب حاضر)، در مقابل فقط ده میلیون دستورالعمل در ثانیه برای کامپیوتر B. یعنی از نظر قدرت خام محاسباتی، کامپیوتر A هزار برابر سریع‌تر از کامپیوتر B است. برای این که تفاوت‌ها باز هم چشمگیرتر شود، فرض کنید که ماهرترین برنامه‌نویس دنیا، مرتب‌سازی درجی را به زبان ماشین در کامپیوتر A می‌نویسد، که کد تولید شده با $۲n^2$ دستورالعمل $n$ عدد را مرتب می‌کند. (در این جا $c_1$ برابر است با ۲.) از طرف دیگر یک برنامه‌نویس معمولی در یک زبان سطح بالا با یک کامپایلر غیربهینه، مرتب‌سازی ادغامی را برای کامپیوتر B نوشته است، به طوری که کد تولید شده برای اجرای مرتب‌سازی ادغامی به $۵۰ n \lg n$ دستورالعمل احتیاج دارد (در این جا $c_2$ برابر است با ۵۰). برای مرتب‌سازی یک میلیون عدد، کامپیوتر A به

$$\frac{۲.(۱۰^۷)^۲ \text{ دستورالعمل}}{۱۰^{۱۰} \text{ دستورالعمل / زمان}} = ۲۰,۰۰۰$$

ثانیه (بیش از ۵/۵ ساعت) زمان احتیاج دارد، در حالی که کامپیوتر B در زمان

$$\frac{\text{دستورالعمل } 50.10^7 \lg 10^7}{\text{زمان / دستورالعمل } 10^7} \approx 1163$$

ثانیه (کم‌تر از ۲۰ دقیقه) مرتب‌سازی را انجام می‌دهد. با استفاده از یک الگوریتم با رشد کم‌تر، حتی با یک کامپایلر ضعیف، کامپیوتر B تقریباً ۱۷ برابر سریع‌تر از کامپیوتر A برنامه را اجرا می‌کند! برای مرتب‌سازی صد میلیون عدد برتری مرتب‌سازی ادغامی باز هم بیشتر جلوه می‌کند: در حالی که مرتب‌سازی درجی تقریباً به زمان ۲۳ روز احتیاج دارد، مرتب‌سازی ادغامی در کم‌تر از چهار ساعت کار را انجام می‌دهد. به طور کلی با رشد اندازه‌ی مسئله، برتری نسبی مرتب‌سازی ادغامی هم رشد می‌کند.

## الگوریتم‌ها و فناوری‌های دیگر

مثال بالا نشان می‌دهد که الگوریتم هم یک *فناوری* است، درست مانند سخت‌افزار کامپیوتر. کارایی کلی کامپیوتر، همان اندازه که به انتخاب سخت‌افزار سریع بستگی دارد، به انتخاب الگوریتم کارامد نیز وابسته است. سرعت الگوریتم‌ها، درست مانند تکنولوژی‌های دیگر کامپیوتر، روز به روز رو به افزایش است.

شاید از خود بپرسید آیا به راستی الگوریتم‌ها در کامپیوترهای امروزی به اندازه‌ی فناوری‌های دیگر اهمیت دارند؟ فناوری‌هایی مانند:

- معماری‌های پیشرفته‌ی کامپیوتر و تکنولوژی‌های تولید،
- واسط گرافیکی کاربر (GUI) با استفاده‌ی ساده و کاربرپسند،
- سیستم‌های شیئ‌گرا،
- تکنولوژی‌های یکپارچه‌ی وب، و
- شبکه‌های سریع، باسیم یا بی‌سیم.

جواب مثبت است. فقط بعضی از کاربردهای خاص هستند که مستقیماً به الگوریتم‌ها نیاز ندارند (مانند برخی برنامه‌های ساده‌ی مبتنی بر شبکه). مثلاً یک سرویس تحت وب را در نظر بگیرید که تعیین می‌کند چطور می‌توان از مکانی به مکان دیگر سفر کرد. پیاده‌سازی این سرویس به سخت‌افزار سریع، واسط گرافیکی کاربر، شبکه‌ی گسترده، و احتمالاً طراحی شیئ‌گرا احتیاج دارد. از طرفی برای انجام بعضی اعمال به الگوریتم‌ها هم نیاز خواهیم داشت. اعمالی مانند یافتن مسیرها (احتمالاً با استفاده از الگوریتم کوتاه‌ترین مسیر)، تبدیل (render) نقشه‌ها، و وارد کردن آدرس‌ها.

علاوه بر این‌ها، حتی کاربردی که در مستقیماً به الگوریتم احتیاجی ندارد، باز هم به شدت بر پایه‌ی الگوریتم‌ها بنا شده است. اگر این کاربرد به سخت‌افزارهای سریع وابسته باشد، طراحی

سخت‌افزارها نیازمند الگوریتم‌ها است. اگر این کاربرد به واسط گرافیکی کاربر احتیاج داشته باشد، طراحی واسط گرافیکی کاربر نیازمند الگوریتم‌ها است. اگر این کاربرد به شبکه نیاز داشته باشد، مسیریابی در شبکه وابسته به الگوریتم‌ها است. اگر برنامه در زبانی غیر از زبان ماشین نوشته شده باشد، پس به وسیله‌ی یک کامپایلر (compiler)، مفسر (interpreter)، و یا اسمبلر (assembler) پردازش می‌شود، که همه‌ی آن‌ها به شدت از الگوریتم‌ها استفاده می‌کنند. الگوریتم‌ها در مرکز اکثر فناوری‌های مورد استفاده‌ی کامپیوترهای امروزی هستند.

همچنین با گسترش روزافزون توانایی‌های کامپیوترها، مسائلی که به کمک آن‌ها حل می‌کنیم هم روز به روز بزرگ‌تر می‌شوند. همان طور که در بالا در مقایسه‌ی مرتب‌سازی درجی و مرتب‌سازی ادغامی دیدیم، با بزرگ شدن اندازه‌ی مسئله‌ها، تفاوت سرعت و کارایی بین الگوریتم‌ها بسیار حیاتی‌تر می‌شود.

آشنایی با دانش و تکنیک‌های قوی الگوریتمی، چیزی است که برنامه‌نویس‌های خبره را از تازه‌کارها جدا می‌کند. به کمک فناوری‌های مدرن، می‌توانیم بدون آشنایی با الگوریتم‌ها بعضی کارها را انجام دهیم، ولی با پیش‌زمینه‌ای خوب در مورد الگوریتم‌ها، از پس انجام کارهای بسیار بیشتری بر خواهیم آمد!

## تمرین‌ها

۱-۲-۱ مثالی از یک برنامه‌ی کاربردی ارائه دهید که در سطح کاربرد به محتوای الگوریتمی نیاز دارد، و در مورد کاربرد الگوریتم در این برنامه بحث کنید.

۱-۲-۲ فرض کنید می‌خواهیم پیاده‌سازی الگوریتم‌های مرتب‌سازی درجی و مرتب‌سازی ادغامی را روی یک ماشین خاص مقایسه می‌کنیم. برای یک ورودی با اندازه‌ی $n$، الگوریتم مرتب‌سازی درجی با اجرای $8n^2$ دستورالعمل مرتب‌سازی را انجام می‌دهد، در حالی که مرتب‌سازی ادغامی به $64n \log n$ دستورالعمل احتیاج دارد. برای چه مقادیری از $n$، مرتب‌سازی درجی سریع‌تر از مرتب‌سازی ادغامی اجرا می‌شود؟

۱-۲-۳ برای این که یک الگوریتم با زمان $100n^2$ سریع‌تر از یک الگوریتم با سرعت $2^n$ بر روی یک ماشین اجرا شود، کوچک‌ترین مقدار اندازه‌ی ورودی $n$ باید چقدر باشد؟

## مسائل

۱-۱ *مقایسه‌ی زمان‌های اجرا*

برای هر تابع $f(n)$ و زمان $t$ در جدول زیر، بزرگ‌ترین اندازه‌ی ورودی $n$ را تعیین کنید که با آن مسئله می‌تواند در زمان t حل شود. فرض کنید که الگوریتم برای حل مسئله به $f(n)$ میکروثانیه احتیاج دارد.

مسائل

| | ۱ ثانیه | ۱ دقیقه | ۱ ساعت | ۱ روز | ۱ ماه | ۱ سال | ۱ قرن |
|---|---|---|---|---|---|---|---|
| $\lg n$ | | | | | | | |
| $\sqrt{n}$ | | | | | | | |
| $n$ | | | | | | | |
| $n \lg n$ | | | | | | | |
| $n^2$ | | | | | | | |
| $n^3$ | | | | | | | |
| $2^n$ | | | | | | | |
| $n!$ | | | | | | | |


# آغاز

در این فصل با چارچوب مورد استفاده در ادامه‌ی کتاب (برای تفکر در مورد طراحی و تحلیل الگوریتم‌ها) آشنا خواهیم شد. مطالب این فصل مستقل است، ولی ارجاءهایی به موضوعات فصل‌های ۳ و ۴ دارد. (همچنین این فصل شامل کار با سری‌ها است، که در پیوست الف روش حل آن‌ها را خواهیم دید.)

با بررسی الگوریتم مرتب‌سازی درجی آغاز می‌کنیم، که در فصل ۱ برای حل مسئله‌ی مرتب‌سازی معرفی شد. چیزی به نام «شبه‌کد» (pseudocode) تعریف می‌کنیم، که احتمالاً خواننده‌هایی که قبلا برنامه‌نویسی با کامپیوتر را تجربه کرده‌اند با آن آشنا هستند، و از آن برای نشان دادن الگوریتم‌ها استفاده می‌کنیم. بعد از تعریف الگوریتم، نشان می‌دهیم که مرتب‌سازی درجی به درستی عمل می‌کند، و زمان اجرای آن را تحلیل می‌کنیم. در تحلیل زمان اجرای الگوریتم، نمادی را معرفی خواهیم کرد که نشان‌دهنده‌ی نسبت افزایش زمان اجرا به افزایش تعداد عناصری است که باید مرتب شوند. در ادامه‌ی بحث مرتب‌سازی درجی، با روش تقسیم و حل (divide-and-conquer) (در بعضی مراجع، این روش به نام روش تفرقه بینداز و حکومت کن شناخته می‌شود. — م) برای طراحی الگوریتم‌ها آشنا می‌شویم و از آن برای ایجاد الگوریتمی به نام مرتب‌سازی ادغامی استفاده می‌کنیم. در نهایت با تحلیل زمان اجرای مرتب‌سازی ادغامی، بحث را خاتمه می‌دهیم.

## ۲-۱ مرتب‌سازی درجی

اولین الگوریتم مورد بحث، مرتب‌سازی درجی، *مسئله‌ی مرتب‌سازی* را که در فصل اول معرفی شد، حل می‌کند. صورت این مسئله عبارت است از:

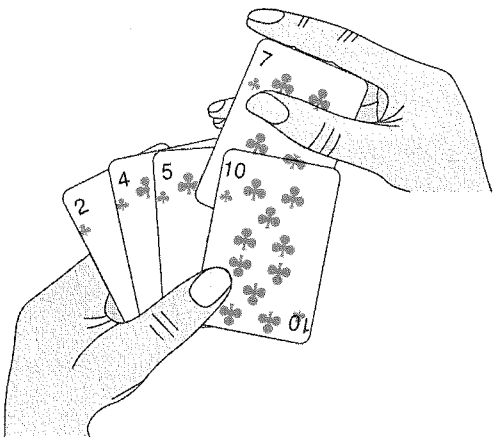
- **ورودی:** دنباله‌ای از $n$ عدد $\langle a_1, a_2, \ldots, a_n \rangle$.

• *خروجی:* یک جایگشت $\langle a'_1, a'_2, ..., a'_n \rangle$ از دنباله‌ی ورودی، به طوری که داشته باشیم $a'_1 \leq a'_2 \leq ... \leq a'_n$.

اعدادی که باید مرتب کنیم با نام *کلیدها* (key) هم شناخته می‌شوند. با این که از نظر مفهومی می‌خواهیم یک دنباله را مرتب کنیم، ولی ورودی به صورت آرایه‌ای با $n$ عنصر به ما داده می‌شود.

در این کتاب، معمولاً الگوریتم‌ها را به صورت برنامه‌هایی به زبان *شبه‌کد* نشان می‌دهیم، که از خیلی جنبه‌ها شبیه زبان‌های C، C++، Java، Pyton، یا Pascal هستند. اگر با یکی از این زبان‌ها آشنا باشید، در خواندن کدهای این کتاب مشکل چندانی نخواهید داشت. چیزی که شبه‌کد را از زبان‌های برنامه‌نویسی واقعی جدا می‌کند، این است که در شبه‌کد از ساده‌ترین و روشن‌ترین روش ممکن برای بیان الگوریتم استفاده می‌کنیم. بعضی مواقع واضح‌ترین روش بیان الگوریتم به زبان انگلیسی است، پس اگر در میان قطعه‌ای از کد «واقعی» اصطلاح و یا جمله‌ای انگلیسی دیدید، تعجب نکنید. تفاوت دیگر بین شبه‌کد و کد واقعی این است که در شبه‌کد معمولاً مسائل مهندسی نرم‌افزار در نظر گرفته نمی‌شود. معمولاً مسائل مربوط به تجرید داده (data abstraction)، پیمانه‌بندی (modularity)، و مدیریت خطا (error handling) نادیده گرفته می‌شوند تا نکات اصلی الگوریتم با حداکثر اختصار بیان شوند.

با مرتب‌سازی درجی شروع می‌کنیم، که الگوریتمی کارامد برای مرتب‌سازی تعداد کمی عنصر است. مرتب‌سازی درجی، همان روشی است که اکثر مردم برای مرتب کردن دسته‌ی کارت‌های بازی از آن استفاده می‌کنند. ابتدا دست چپ ما خالی است، و کارت‌ها روی میز قرار گرفته‌اند. در هر مرحله یک کارت از روی میز برداشته و آن را در جای درست خود در دست چپ قرار می‌دهیم. برای یافتن مکان درست برای یک کارت، آن را با هر کدام از کارت‌هایی که در دست چپ قرار دارند، از چپ به راست، مقایسه می‌کنیم، مانند شکل ۲-۱. در هر لحظه، کارت‌های موجود در دست چپ مرتب شده هستند، و این کارت‌ها همان‌هایی هستند که در شروع کار، در مکان‌های ابتدایی دسته‌ی روی میز بوده‌اند.



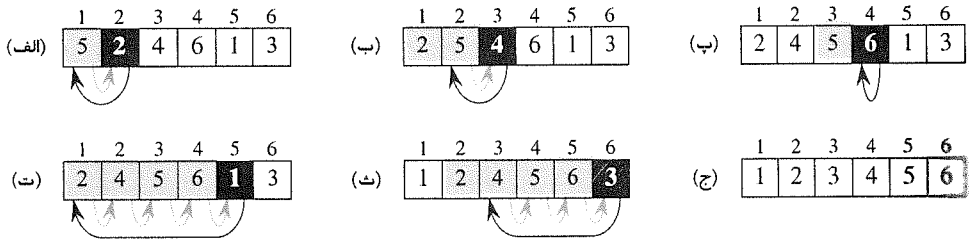

شکل ۲-۱ مرتب‌سازی دسته‌ی کارت‌ها با استفاده از روش درجی.

در ادامه، شبه‌کد مرتب‌سازی درجی به شکل رویه‌ای با نام INSERTION-SORT خواهد آمد، که پارامتر ورودی آن آرایه‌ی $A[1..n]$ است. این آرایه حاوی دنباله‌ای از اعداد به طول $n$ است که باید مرتب شوند. (در کد زیر، تعداد اعضای آرایه با $A.length$ مشخص شده است.) اعداد ورودی به صورت *درجا* (in place) مرتب می‌شوند، یعنی اعداد درون خود آرایه‌ی $A$ جابه‌جا می‌شوند، و در هر زمان حداکثر تعداد ثابتی از آن‌ها خارج آرایه ذخیره خواهند شد. وقتی INSERTION-SORT پایان می‌یابد، آرایه‌ی ورودی $A$ شامل دنباله‌ی مرتب شده‌ی خروجی خواهد بود.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      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              A[i + 1] = A[i]
7              i = i − 1
8          A[i + 1] = key
```

## ثابت‌های حلقه و درستی مرتب‌سازی درجی

شکل ۲-۲ نشان می‌دهد که الگوریتم مرتب‌سازی درجی برای ورودی $A = \langle 5,2,4,6,1,3 \rangle$ چطور عمل می‌کند. اندیس $j$ نشان‌دهنده‌ی کارت فعلی است که باید در دست چپ جایگذاری شود. هنگام شروع هر تکرار از حلقه‌ی for، که با $j$ اندیس‌گذاری شده است، زیرآرایه‌ی $A[1..j-1]$ کارت‌های مرتب شده‌ی درون دست را تشکیل می‌دهد، و زیرآرایه‌ی باقی‌مانده‌ی $A[j+1..n]$ متناظر است با دسته‌ی کارت‌های باقی‌مانده بر روی میز. در واقع عناصر $A[1..j-1]$، کارت‌هایی هستند که در ابتدا در مکان‌های ۱ تا $j-1$ قرار داشتند، ولی حالا به شکل مرتب شده. به این خصوصیات زیرآرایه‌ی $A[1..j-1]$ به صورت رسمی *ثابت‌های حلقه* (loop invariant) می‌گوییم:




**شکل ۲-۲** عملیات INSERTION-SORT روی آرایه‌ی $A = \langle 5,2,4,6,1,3 \rangle$. اندیس‌های آرایه در بالای مستطیل‌ها دیده می‌شوند، و مقدار عناصر آرایه درون مستطیل‌ها قرار دارند. (الف)-(ث) عملیات حلقه‌ی for در خطوط ۱-۸ در هر مرحله مستطیل تیره نشان دهنده‌ی کلید $A[j]$ است، که در خط ۵ با مقادیر درون مستطیل‌های کم‌رنگ سمت چپ خود مقایسه می‌شود. فلش‌های خاکستری مقادیر آرایه را نشان می‌دهند که در خط ۶ یک خانه به راست منتقل می‌شوند، و فلش سیاه مکانی را نشان می‌دهد که کلید فعلی در خط ۸ به آنجا منتقل می‌شود. (ج) آرایه‌ی مرتب شده‌ی نهایی.

- در آغاز هر تکرار حلقه‌ی for خطوط ۱-۸، زیرآرایه‌ی $A[1 .. j-1]$ شامل عناصری است که در ابتدا در مکان‌های $A[1 .. j-1]$ قرار داشتند، ولی حالا به صورت مرتب شده.

در این جا از ثابت‌های حلقه استفاده برای نشان دادن درستی الگوریتم استفاده می‌کنیم. برای این کار باید سه چیز را در مورد ثابت حلقه نشان دهیم:

- شروع (Initialization): این که ثابت حلقه قبل از شروع اولین تکرار حلقه برقرار است.
- ادامه (Maintenance): این که اگر ثابت حلقه قبل از شروع یک تکرار حلقه درست باشد، تا قبل از شروع حلقه‌ی بعد نیز برقرار باقی می‌ماند.
- پایان (Temination): پس از اتمام حلقه، ثابت حلقه خصوصیتی مفید به ما می‌دهد که در نشان دادن درستی الگوریتم به ما کمک می‌کند.

اگر دو خصوصیت اول درست باشند، به این معنی است که ثابت حلقه قبل از هر تکرار حلقه برقرار است. (مسلما برای نشان دادن این که ثابت حلقه قبل از هر تکرار برقرار است، باید از حقایقی غیر از خود ثابت حلقه استفاده کنیم.) به شباهت روش بالا و استقرای ریاضی توجه کنید، که در آن برای نشان دادن درستی یک خصوصیت، باید برقراری حالت پایه و گام استقرا را اثبات کنیم. در این جا نشان دادن درستی ثابت حلقه قبل از شروع حلقه، مشابه حالت پایه در استقرا است، و نشان دادن درست باقی ماندن ثابت حلقه در هر بار تکرار حلقه، مشابه گام استقرا.

شاید خصوصیت سوم مهم‌ترین آن‌ها باشد، چرا که در این جا می‌خواهیم از ثابت حلقه برای نشان دادن درستی الگوریتم استفاده کنیم. معمولاً از ثابت حلقه به همراه شرطی که باعث پایان یافتن حلقه شده است استفاده می‌کنیم. خصوصیت پایان در ثابت حلقه با نحوه‌ی استفاده‌ی معمول از استقرای ریاضی تفاوت دارد، چرا که در آن گام استقرا تا بی‌نهایت ادامه پیدا می‌کند؛ در این جا «استقرا» با پایان تکرار حلقه خاتمه می‌یابد.

اجازه دهید ببینیم چطور این خصوصیات برای مرتب‌سازی درجی برقرارند.

- شروع: با نشان دادن درستی ثابت حلقه قبل از اولین اجرای حلقه آغاز می‌کنیم، زمانی که داریم $j = 2$.[1] بنابراین زیرآرایه‌ی $A[1 .. j-1]$ فقط حاوی عنصر $A[1]$ است، که در ابتدا هم در همان $A[1]$ قرار داشته است. به علاوه این زیرآرایه مرتب شده است (چرا که فقط یک عضو دارد)، که نشان می‌دهد ثابت حلقه قبل از شروع اولین تکرار حلقه برقرار است.
- ادامه: سپس به بررسی خصوصیت دوم می‌پردازیم: نشان دادن درستی ثابت حلقه پس از هر تکرار حلقه. به طور کلی، بدنه‌ی حلقه‌ی for خارجی به ترتیب عناصر $A[j-1]$، $A[j-2]$، $A[j-3]$، و... را بررسی می‌کند تا مکان مناسب را برای $A[j]$ بیابد (خطوط ۴-۷)، و در همان جا $A[j]$ درج (insert) می‌شود (خط ۸). آن گاه زیرآرایه‌ی $A[1 .. j]$ حاوی عناصری

[1] وقتی حلقه، یک حلقه‌ی for باشد، لحظه‌ای که ثابت حلقه را درست قبل از اولین تکرار بررسی می‌کنیم، دقیقاً بعد از مقداردهی اولیه‌ی متغیر شمارنده‌ی حلقه و دقیقاً قبل از اولین تست در سرآیند حلقه است. برای INSERTION-SORT این زمان بعد از مقداردهی $j$ با ۲، و قبل از تست درستی عبارت $j \leq A.length$ خواهد بود.

خواهد بود که در ابتدا در $A[۱ .. j]$ قرار داشتند، ولی این بار به صورت مرتب. سپس افزایش $j$ برای تکرار بعدی حلقه‌ی for درستی ثابت حلقه را حفظ می‌کند.

برای اثبات رسمی‌تر خصوصیت دوم باید یک ثابت حلقه برای حلقه‌ی while بیابیم. با این حال در این جا ترجیح می‌دهیم زیاد وارد جزئیات رسمی نشویم، و به تحلیل غیر رسمی خود در اثبات خصوصیت دوم برای حلقه‌ی خارجی بسنده می‌کنیم.

- پایان: در نهایت خواهیم دید که پس از پایان حلقه چه اتفاقی می‌افتد. برای مرتب‌سازی درجی، حلقه‌ی **for** خارجی زمانی پایان می‌یابد که داشته باشیم $j > A.length = n$. چون هر تکرار حلقه $j$ را یکی افزایش می‌دهد، هنگام پایان باید داشته باشیم $j = n + ۱$. با مقداردهی $j = n + ۱$ در ثابت حلقه، می‌بینیم که زیرآرایه‌ی $A[۱..n]$ شامل عناصری است که در ابتدا در $A[۱..n]$ قرار داشتند، ولی اکنون به شکل مرتب شده. اما زیرآرایه‌ی $A[۱..n]$ کل آرایه است! بنابراین تمام آرایه مرتب شده است، که نشان می‌دهد الگوریتم درست است.

در ادامه‌ی این فصل و همین طور در فصل‌های دیگر، از روش ثابت حلقه برای نشان دادن درستی الگوریتم‌ها استفاده خواهیم کرد.

## قراردادهای شبه‌کد

در شبه‌کدهای خود از قراردادهای زیر استفاده خواهیم کرد.

۱. تورفتگی، ساختار بلوک را نشان می‌دهد. به عنوان مثال، بدنه‌ی حلقه‌ی for که در خط ۱ شروع می‌شود، شامل خطوط ۲-۸ است، و بدنه‌ی حلقه‌ی while که در خط ۵ شروع می‌شود، شامل خطوط ۶-۷، و نه خط ۸. از این شیوه‌ی تورفتگی برای عبارات if-then[۱] هم استفاده خواهیم کرد. استفاده از تورفتگی به جای روش‌های معمول مشخص کردن ساختار بلوک، مانند begin و end، در تمیزی و کاهش حجم کد بسیار مؤثر خواهد بود.

۲. ساختارهای حلقه، مانند while، for، و repeat-until و ساختار شرطی if-else نمایشی مشابه زبان‌های C، C++، Java، Pyton، و Pascal دارند.[۲] در این کتاب، متغیر حلقه بعد از خروج از حلقه مقدار خود را حفظ می‌کند، برخلاف بعضی موقعیت‌ها که در C++، جاوا یا پاسکال به وجود می‌آید. بنابراین دقیقاً بعد از اتمام یک حلقه‌ی for، مقدار متغیر حلقه برابر است با اولین مقداری که از کران حلقه بیشتر است. در اثبات درستی مرتب‌سازی درجی از این خاصیت استفاده کردیم. سرآیند (header) حلقه‌ی for در خط ۱ عبارت است از $\text{for } j = 2 \text{ to } A.length$، که

---

[۱] در یک عبارت if-else، فرورفتگی else برابر با فرورفتگی if متناظر با آن خواهد بود. با این که از کلمه‌ی کلیدی then صرف نظر می‌کنیم، ولی اکثراً قسمتی را که بعد از برقراری شرط if اجرا می‌شود، عبارت then می‌نامیم. برای تست‌های چندحالته، از elseif برای تست‌های بعدی استفاده می‌کنیم.

[۲] اکثر زبان‌های مبتنی بر ساختار بلوک دارای ساختمان‌های مشابه هستند، ولی گرامر دقیق آن‌ها ممکن است با هم متفاوت باشد. Pyton فاقد حلقه‌های repeat-until است، و نحوه‌ی عمل‌کرد حلقه‌های for آن هم کمی با حلقه‌های for این کتاب تفاوت دارد.

وقتی حلقه پایان می‌یابد، داریم $j = A.length + 1$ (یا $j = n + 1$، چرا که $n = A.length$). از کلمه‌ی کلیدی to برای زمانی استفاده می‌کنیم که در حلقه‌ی for، شمارنده‌ی حلقه در هر تکرار افزایش می‌یابد، و از کلمه‌ی کلیدی downto برای نشان دادن کاهش متغیر حلقه استفاده می‌کنیم. وقتی متغیر حلقه با مقداری بیش از ۱ افزایش می‌یابد، مقدار تغییر بعد از کلمه‌ی کلیدی اختیاری by خواهد آمد.

۳. نماد "//" نشان می‌دهد که ادامه‌ی خط شامل توضیحات (comment) است.

٤. یک انتساب (assignment) چندتایی به شکل $i = j = e$، به هر دو متغیر $i$ و $j$ مقدار عبارت $e$ را می‌دهد. این عبارت، دقیقاً معادل دو عبارت $j = e$ و $i = j$ است که پشت سر هم قرار بگیرند.

۵. متغیرها (مانند $i$، $j$، و $key$) درون رویه‌ها به صورت محلی (local) هستند. از متغیرهای سراسری (global) بدون اشاره‌ی مستقیم استفاده نخواهد شد.

۶. دسترسی به عناصر آرایه‌ها به این صورت خواهد بود: نام آرایه، و در ادامه اندیس آن درون کروشه([]). مثلاً $A[i]$ نشان‌دهنده‌ی عنصر $i$ام آرایه‌ی $A$ است. نماد ".." برای نشان دادن محدوده‌ای از مقادیر در یک آرایه مورد استفاده قرار می‌گیرد. بنابراین $A[1..j]$ زیرآرایه‌ای از $A$ شامل $j$ عنصر $A[1], A[2], ..., A[j]$ را مشخص می‌کند.

۷. داده‌های مرکب (compound data) معمولاً به صورت *اشیاء* (object) سازماندهی می‌شوند، که از *خصیصه‌ها* (attribute) یا *فیلدها* (field) تشکیل شده‌اند. دسترسی به خصیصه‌ای خاص از یک شیئ با روشی مشابه بسیاری از زبان‌های برنامه‌نویسی شیئ‌گرا انجام می‌شود: نام شیئ، نقطه، و پس از آن نام خصیصه. به عنوان مثال یک آرایه را به صورت یک شیئ در نظر می‌گیریم که خصوصیت *length* تعداد عناصر درون آن را مشخص می‌کند. برای نشان دادن تعداد عناصر درون آرایه‌ی $A$ می‌نویسیم $A.length$.

متغیری که یک آرایه یا یک شیئ را مشخص می‌کند، اشاره‌گر(pointer)ی است به داده‌ای که آن آرایه یا شیئ را نشان می‌دهد. برای تمام فیلدهای $f$ از شیئ $x$، انتساب $y = x$ باعث انتساب $y.f = x.f$ هم می‌شود. به علاوه، اگر بعد از آن قرار دهیم $x.f = ۳$، آن گاه نه تنها $x.f = ۳$، بلکه داریم $y.f = ۳$. یعنی بعد از انتساب $y = x$، $x$ و $y$ به یک شیئ اشاره می‌کنند.

در این جا نماد خصیصه‌ها می‌تواند به صورت «آبشاری» هم عمل کند. برای مثال فرض کنید خصیصه‌ی $f$، خود اشاره‌گری باشد به شیئ خاصی که دارای یک خصیصه‌ی $g$ است. در این صورت عبارت $x.f.g$ به صورت ضمنی نشان دهنده‌ی $(x.f).g$ است. به عبارت دیگر اگر قرار دهیم $y = x.f$، آن گاه $x.f.g$ معادل خواهد بود با $y.g$.

بعضی مواقع یک اشاره‌گر به هیچ شیئی اشاره نمی‌کند. در این موارد، مقدار خاص NIL را به آن نسبت می‌دهیم.

۸. پارامترها *با مقدار* (by value) به رویه‌ها ارسال می‌شوند: یعنی رویه‌ی فراخوانی شده یک کپی از پارامتر را دریافت می‌کند، و اگر به آن مقداری نسبت دهد، تغییر ایجاد شده در متغیر اصلی (در رویه‌ی فراخوانی‌کننده) تأثیری *نخواهد داشت*. وقتی اشیاء به رویه‌ها ارسال می‌شوند، اشاره‌گری که به داده‌های شیئ اشاره می‌کند، کپی می‌شود، ولی فیلدهای شیئ کپی نمی‌شوند. به عنوان مثال اگر $x$ یک پارامتر از رویه‌ی فراخوانی کننده باشد، انتساب $x = y$ در رویه‌ی فراخوانی شده، برای رویه‌ی فراخوانی‌کننده قابل رؤیت نیست، ولی انتساب $x.f = ۳$، در رویه‌ی فراخوانی کننده دیده خواهد شد. به طور مشابه، آرایه‌ها با اشاره‌گر به رویه‌ها ارسال می‌شوند، یعنی به جای کل آرایه، یک اشاره‌گر به آرایه ارسال می‌شود، و تغییر روی عناصر آرایه برای رویه‌ی فراخوانی‌کننده قابل رؤیت است.

۹. یک عبارت **return** کنترل را به نقطه‌ی فراخوانی در رویه‌ی فراخوانی‌کننده بازمی‌گرداند. اکثر عبارت‌های **return** یک مقدار هم به رویه‌ی فراخوانی‌کننده بازمی‌گردانند. شبه‌کدهای این کتاب با اکثر زبان‌های برنامه‌نویسی تفاوت دارند، چرا که اجازه می‌دهند مقادیر متعددی در یک عبارت **return** بازگردانده شود.

۱۰. عملگرهای منطقی "and" و "or" به صورت *میان‌بری* (short circuiting) هستند. یعنی هنگام ارزیابی عبارت "$x$ and $y$"، ابتدا $x$ ارزیابی می‌شود. اگر مقدار $x$ نادرست (FALSE) باشد، به این معنی است که کل عبارت نمی‌تواند درست (TRUE) باشد، پس احتیاجی به ارزیابی $y$ نیست. از طرف دیگر اگر $x$ درست باشد، باید $y$ را ارزیابی کنیم تا بتوانیم مقدار کل عبارت را تشخیص دهیم. به طور مشابه در عبارت "$x$ or $y$"، فقط در صورتی مقدار $y$ را ارزیابی می‌کنیم که مقدار $x$ نادرست باشد. میان‌بر زدن در ارزیابی عملگرهای منطقی به ما اجازه می‌دهد که عبارات منطقی مانند "$x \neq \text{NIL}$ and $x.f = y$" را بنویسیم، بدون این که نگران این باشیم که اگر $x$ برابر NIL باشد، هنگام ارزیابی $x.f$ چه رخ می‌دهد.

## تمرین‌ها

۲-۱-۱ با استفاده از شکل ۲-۲ به عنوان مدل، عملیات INSERTION-SORT را روی آرایه‌ی $A = \langle ۳۱,۴۱,۵۹,۲۶,۴۱,۵۸ \rangle$ نشان دهید.

۲-۱-۲ رویه‌ی INSERTION-SORT را طوری بازنویسی کنید که به جای مرتب کردن نزولی، آرایه را به صورت صعودی مرتب کند.

۲-۱-۳ مسئله‌ی جستجوی زیر را در نظر بگیرید:

- *ورودی:* دنباله‌ای از $n$ عدد $A = \langle a_۱, a_۲, \ldots, a_n \rangle$ و یک مقدار $v$.
- *خروجی:* یک اندیس $i$ به طوری که داشته باشیم $v = A[i]$، و یا NIL در صورت موجود نبودن مقدار $v$ در آرایه‌ی $A$.

یک شبه‌کد برای *جستجوی خطی* بنویسید، که کل دنباله را برای مقدار $v$ جستجو می‌کند. با استفاده از یک ثابت حلقه درستی الگوریتم خود را نشان دهید. اطمینان حاصل کنید که ثابت حلقه هر سه خصوصیت را داراست.

۲-۱-۴ مسئله‌ی جمع دو عدد صحیح $n$ بیتی دودوی را که در دو آرایه‌ی $n$ عنصری $A$ و $B$ ذخیره شده‌اند، در نظر بگیرید. جمع دو عدد باید به شکل دودویی در یک آرایه‌ی $(n+1)$ عنصری $C$ ذخیره شود. مسئله را به صورت رسمی تعریف کرده و شبه‌کدی برای جمع دو عدد طراحی کنید.

## ۲-۲ تحلیل الگوریتم‌ها

*تحلیل* (analyze) یک الگوریتم یعنی پیش‌بینی منابعی که اجرای الگوریتم به آن‌ها نیاز دارد. گه‌گاه منابعی مانند حافظه‌ی کامپیوتر، پهنای باند شبکه و یا سخت افزار کامپیوتر مسئله‌ی اصلی هستند، ولی در اکثر مواقع این زمان محاسبه است که نیاز به پیش‌بینی دارد. به طور کلی با تحلیل الگوریتم‌های مختلف برای یک مسئله، کارامدترین آن‌ها مشخص می‌شود. احتمالاً چنین تحلیلی بیش از یک الگوریتم مناسب برای مسئله مشخص می‌کند، ولی معمولاً الگوریتم‌های نامناسب زیادی در این فرایند حذف می‌شوند.

قبل از این که بتوانیم یک الگوریتم را تحلیل کنیم، باید یک مدل از تکنولوژی مورد استفاده‌ی خود داشته باشیم، شامل مدلی از منابع تکنولوژی مورد نظر و هزینه‌های آن‌ها. در بخش اعظم این کتاب، تکنولوژی مورد استفاده را یک ماشین تک پردازنده با *دسترسی تصادفی* (random-access machine, RAM) در نظر می‌گیریم، و آگاه هستیم که الگوریتم‌ها به صورت برنامه‌ی کامپیوتر پیاده‌سازی می‌شوند. در مدل RAM دستورالعمل‌ها یکی پس از دیگری اجرا می‌شوند، بدون هیچ گونه عملیات هم زمان.

برای تحلیل دقیق باید دستورالعمل‌های مدل RAM و هزینه‌ی آن‌ها را به دقت تعیین کنیم. ولی انجام این کار بسیار خسته کننده است، و توجه ما را از طراحی و تحلیل الگوریتم‌ها دور می‌کند. از طرفی باید دقت کنیم که از مدل RAM استفاده‌ی نادرستی نکرده باشیم. به عنوان مثال، اگر یک RAM دستورالعملی برای مرتب کردن داشته باشد چه طور خواهد شد؟ در این صورت فقط با یک دستورالعمل می‌توانیم مرتب‌سازی را انجام دهیم. چنین مدلی غیر واقعی خواهد بود، چرا که کامپیوترهای واقعی چنین دستورالعملی ندارند. بنابراین الگوی ما نحوه‌ی عمل‌کرد کامپیوترهای واقعی خواهد بود. مدل RAM شامل دستورالعمل‌هایی است که در کامپیوترهای واقعی وجود دارند: عملیات ریاضی (جمع، تفریق، ضرب، تقسیم، باقیمانده، کف (جزء صحیح)، و سقف)، جابه‌جایی داده (بارگذاری(load)، ذخیره، کپی)، و کنترل (پرش شرطی و غیر شرطی، فراخوانی زیرروال و بازگشت). هر دستورالعمل به مقدار ثابتی زمان برای اجرا نیاز دارد.

انواع داده در مدل RAM اعداد صحیح و اعشاری هستند. با حال که در این کتاب معمولاً خود را درگیر دقت اعداد نمی‌کنیم، در بعضی کاربردها دقت اعداد مهم هستند. همچنین یک کران برای اندازه‌ی هر کلمه‌ی داده‌ای در نظر می‌گیریم. برای مثال وقتی با ورودی‌های با اندازه‌ی $n$ کار می‌کنیم، معمولاً فرض می‌کنیم که اعداد صحیح با $c \lg n$ بیت نشان داده می‌شوند، که در آن $c$ ثابتی است بزرگتر یا مساوی ۱. ثابت $c$ باید بزرگتر یا مساوی یک باشد تا هر کلمه بتواند مقدار $n$ را در خود نگه دارد، که ما را قادر می‌سازد عناصر ورودی را با اندیس‌های ۱ تا $n$ شماره‌گذاری کنیم، و همین طور مقدار $c$ را محدود می‌کنیم که اندازه‌ی کلمات به صورت بی کران بزرگ نشوند. (اگر اندازه‌ی کلمات به صورت بی‌کران بزرگ شوند، می‌توانیم مقدار زیادی داده را در یک کلمه ذخیره کنیم و در زمان ثابت روی آن عملیات انجام دهیم – یک سناریوی غیر ممکن.)

کامپیوترهای واقعی دستورالعمل‌هایی دارند که در لیست بالا نیامده است، و چنین دستورالعمل‌هایی ابهاماتی در مدل RAM ایجاد می‌کنند. به عنوان مثال آیا به توان رساندن عملیاتی با زمان ثابت است؟ در حالت کلی، نه. وقتی $x$ و $y$ اعداد حقیقی باشند، محاسبه‌ی $x^y$ به چندین دستورالعمل نیاز خواهد داشت. با این حال در حالت‌های خاص عملیات توان در زمان ثابت اجرا خواهد شد. بسیاری از کامپیوترها، یک دستورالعمل جابه‌جایی به چپ (shift left) دارند، که در زمان ثابت بیت‌های یک عدد صحیح را $k$ تا به چپ جابه‌جا می‌کند. در اکثر کامپیوترها جابه‌جا کردن بیت‌های یک عدد صحیح به اندازه‌ی یک خانه به چپ، معادل ضرب عدد در ۲ است، و $k$ شیفت به چپ برابر با ضرب عدد در $2^k$. از این رو این کامپیوترها می‌توانند با $k$ بار جابه‌جایی به چپ، $2^k$ را در زمان ثابت محاسبه کنند. (البته اگر $k$ از تعداد بیت‌های یک کلمه در کامپیوتر فراتر نرود.) در مدل RAM تلاش خواهیم کرد که از چنین ابهاماتی دوری کنیم، ولی وقتی که $k$ یک عدد صحیح مثبت و اندازه‌ی کافی کوچک باشد، محاسبه‌ی $2^k$ را به صورت یک دستورالعمل با زمان ثابت در نظر می‌گیریم.

در مدل RAM سلسله مراتب حافظه (memory hierarchy) را که در کامپیوترهای امروزی مرسوم است (حافظه‌های مجازی (virtual memory) و یا کَش (cache)) مدل نخواهیم کرد. مدل‌های محاسباتی بسیاری هستند که تأثیرات سلسله مراتب حافظه را در نظر می‌گیرند، که بعضی مواقع در برنامه‌های واقعی روی کامپیوترهای واقعی مفید است. تعدادی از برنامه‌های این کتاب هم تأثیرات سلسله مراتب حافظه را در نظر می‌گیرند، ولی در اکثر موارد در تحلیل‌های این کتاب از این تأثیرات صرف نظر می‌شود. مدل‌هایی که سلسله مراتب حافظه را در نظر می‌گیرند، از مدل RAM پیچیده‌تر هستند و کار کردن با آن‌ها مشکل‌تر است. به علاوه تحلیل مدل RAM معمولاً پیشگوی مناسبی برای کارایی برنامه‌ها بر روی ماشین‌های واقعی است.

در مدل RAM تحلیل حتی یک الگوریتم ساده می‌تواند بسیار دشوار باشد. ابزارهای ریاضی مورد نیاز می‌تواند شامل ترکیبیات، نظریه‌ی احتمالات، مهارت جبر و توانایی تشخیص عبارت‌های تأثیرگذار در فرمول‌ها باشد. از آن جایی که ممکن است رفتار یک الگوریتم برای ورودی‌های مختلف متفاوت باشد، به روش‌هایی نیاز خواهیم داشت برای تحلیل کلی این رفتار به صورت یک فرمول ساده و قابل فهم.

با این که معمولاً برای تحلیل یک الگوریتم فقط از یک مدل استفاده می‌کنیم، باز هم انتخاب‌های زیادی برای تشریح تحلیل خود داریم. روش مورد استفاده باید به آسانی قابل نوشتن و دستکاری باشد، منابع مهم مورد نیاز الگوریتم را نشان دهد، و از جزئیات اضافی پرهیز کند.

## تحلیل مرتب‌سازی درجی

زمان مورد نیاز رویه‌ی INSERTION-SORT بستگی به ورودی دارد: مرتب‌سازی هزار عدد بسیار بیشتر از مرتب‌سازی سه عدد زمان می‌برد. به علاوه INSERTION-SORT ممکن است برای دو ورودی با تعداد عنصر برابر، بسته به مقدار مرتب بودن آن‌ها در اول کار، به زمان‌های متفاوتی نیاز داشته باشد. معمولاً زمان مورد نیاز یک الگوریتم با اندازه‌ی ورودی رشد می‌کند، بنابراین مرسوم است که زمان اجرای الگوریتم‌ها را به صورت تابعی از اندازه‌ی ورودی نشان دهیم. برای این کار باید اصطلاحات «زمان اجرا» و «اندازه‌ی ورودی» را با دقت بیشتری تعریف کنیم.

بهترین تعریف برای *اندازه‌ی ورودی* به مسئله‌ی مورد بررسی وابسته است. برای بسیاری از مسائل، مانند مرتب‌سازی یا محاسبه‌ی تبدیل فوریه‌ی توابع، طبیعی‌ترین مقیاس *تعداد عناصر ورودی است*— مثلاً اندازه‌ی آرایه‌ی ورودی در مسئله‌ی مرتب‌سازی. برای بسیاری مسائل دیگر، مانند ضرب دو عدد در یکدیگر، بهترین مقیاس برای اندازه‌ی ورودی کل تعداد بیت‌هایی است که برای نشان دادن ورودی به صورت دودویی احتیاج داریم. بعضی مواقع مناسب‌تر است که اندازه‌ی ورودی را به جای یک عدد، با دو عدد نشان دهیم. مثلاً اگر ورودی الگوریتم یک گراف باشد، می‌توان اندازه‌ی ورودی را با تعداد رأس‌ها و تعداد یال‌های گراف مشخص کرد. بنابراین در بررسی هر مسئله ابتدا مشخص خواهیم کرد که از چه مقیاسی به عنوان اندازه‌ی ورودی استفاده خواهیم کرد.

*زمان اجرای* یک الگوریتم برای یک ورودی خاص عبارت است از تعداد اعمال، یا *مراحل* (step) اصلی انجام شده. در صورت امکان بهتر است دستورالعمل را طوری تعریف کنیم که مستقل از ماشین باشد. فعلا اجازه دهید دید زیر را داشته باشیم: اجرای هر خط شبه‌کد به مقدار ثابتی زمان نیاز دارد. ممکن است زمان اجرای خطوط مختلف متفاوت باشد، ولی فرض می‌کنیم که اجرای خط $i$ام به زمان $c_i$ نیاز دارد، و $c_i$ یک ثابت است. این دیدگاه با مدل RAM هم‌خوانی دارد، همچنین با پیاده‌سازی شبه‌کد روی اکثر کامپیوترهای واقعی.[1]

در ادامه‌ی بحث، توضیح زمان اجرای INSERTION-SORT را از شکل نامفهوم بالا به توصیفی ساده و دقیق تبدیل خواهیم کرد. همچنین به کمک این شکل ساده می‌توانیم تشخیص دهیم که آیا یک الگوریتم از الگوریتم دیگر کاراتر است یا خیر.

---

[1] چند نکته‌ی ظریف در این جا وجود دارد. مراحل محاسباتی که ما در این جا به زبان مجاوره آن‌ها را بیان می‌کنیم، معمولاً نسخه‌ای از یک رویه هستند که به بیش از یک مقدار زمان ثابت برای اجرا نیاز دارند. به عنوان مثال، بعداً در این کتاب ممکن است از «مرتب‌سازی بر حسب مختصات x» صحبت کنیم، که همان طور که خواهیم دید، به بیش از زمان ثابت نیاز دارد. همچنین توجه کنید که عبارتی که یک زیرروال را فراخوانی می‌کند به زمان ثابت نیاز دارد، ولی یک زیرروال، بعد از این که فراخوانی شد، ممکن است در زمان بیشتری اجرا شود. یعنی روند *فراخوانی* یک زیرروال–ارسال پارامترها به آن و غیره– را از روند *اجرای* زیرروال جدا می‌کنیم.

با تعیین هزینه‌ی زمانی هر خط از رویه‌ی INSERTION-S0RT و تعداد تکرار هر کدام شروع می‌کنیم. فرض می‌کنیم $t_j$، برای $j = ۲,۳,...,n$ ($n = A.length$)، تعداد تکرارهای حلقه‌ی while در خط ۵ برای هر $j$ باشد. وقتی یک حلقه‌ی for و یا while به صورت طبیعی پایان می‌یابد (یعنی به علت نادرست بودن شرط حلقه)، تست اول حلقه یک بار بیشتر از بدنه‌ی حلقه اجرا شده است. فرض می‌کنیم توضیحات قابل اجرا نیستند، و بنابراین به زمان احتیاج ندارند.

| | INSERTION-SORT($A$) | هزینه | دفعات تکرار |
|---|---|---|---|
| 1 | for $j = 2$ to $A.length$ | $c_۱$ | $n$ |
| 2 | $key = A[j]$ | $c_۲$ | $n-۱$ |
| 3 | // Insert $A[j]$ into the sorted sequence $A[1..j-1]$. | ۰ | $n-۱$ |
| 4 | $i = j-1$ | $c_۴$ | $n-۱$ |
| 5 | while $i > 0$ and $A[i] > key$ | $c_۵$ | $\sum_{j=۲}^{n} t_j$ |
| 6 | $A[i+1] = A[i]$ | $c_۶$ | $\sum_{j=۲}^{n} (t_j - ۱)$ |
| 7 | $i = i-1$ | $c_۷$ | $\sum_{j=۲}^{n} (t_j - ۱)$ |
| 8 | $A[i+1] = key$ | $c_۸$ | $n-۱$ |

زمان اجرای کل الگوریتم برابر است با مجموع زمان اجرای تک تک عبارات. عبارتی که به $c_i$ دستورالعمل احتیاج دارد و $n$ بار اجرا می‌شود، در کل زمان اجرا به اندازه‌ی $c_i n$ تأثیر خواهد داشت[۱]. برای محاسبه‌ی $T(n)$، زمان اجرای INSERTION-SORT برای یک ورودی با $n$ مقدار، حاصل ضرب *هزینه‌ها* در *تعداد تکرار* را با هم جمع می‌کنیم، که به دست می‌دهد:

$$T(n) = c_۱ n + c_۲(n-۱) + c_۴(n-۱) + c_۵ \sum_{j=۲}^{n} t_j + c_۶ \sum_{j=۲}^{n} (t_j - ۱)$$
$$+ c_۷ \sum_{j=۲}^{n} (t_j - ۱) + c_۸(n-۱)$$

حتی وقتی اندازه‌ی ورودی ثابت باشد، ممکن است زمان اجرای الگوریتم به *مقادیر* داده شده در ورودی بستگی داشته باشد. برای مثال در INSERTION-SORT بهترین حالت زمانی اتفاق می‌افتد که آرایه از ابتدا مرتب شده باشد. برای $j = ۲,۳,...,n$، در اجرای خط ۵ وقتی که $i$ مقدار اولیه‌ی $j-۱$ را دارد، می‌بینیم که $A[i] \le key$. بنابراین برای $j = ۲,۳,...,n$ داریم $t_j = ۱$، و در بهترین حالت زمان اجرا برابر است با

$$T(n) = c_۱ n + c_۲(n-۱) + c_۴(n-۱) + c_۵(n-۱) + c_۸(n-۱)$$
$$= (c_۱ + c_۲ + c_۴ + c_۵ + c_۸)n - (c_۲ + c_۴ + c_۵ + c_۸)$$

[۱] این خصوصیت لزوماً برای یک منبع مانند حافظه برقرار نیست. عبارتی که به $m$ کلمه از حافظه دسترسی پیدا می‌کند، و این کار را $n$ بار انجام می‌دهد، در کل لزوماً به $mn$ کلمه از حافظه را ارجاع نمی‌کند.

این زمان اجرا را می‌توان به صورت $an+b$ نشان داد، که در آن $a$ و $b$ ثابت‌اند و به هزینه‌های $c_i$ بستگی دارند. بنابراین تابع بالا یک *تابع خطی* نسبت به $n$ است.

بدترین حالت زمانی است که آرایه به صورت برعکس مرتب شده باشد – یعنی به صورت نزولی. باید هر عنصر $A[j]$ را با تمام عناصر زیرآرایه‌ی مرتب شده‌ی $A[1..j-1]$ مقایسه کنیم، و بنابراین برای $j = 2,3,...,n$ داریم $t_j = j$. با توجه به این که

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

و

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

(برای حل مجموع‌های بالا، به پیوست الف مراجعه کنید)، متوجه خواهیم شد که در بدترین حالت، زمان اجرای INSERTION-SORT برابر است با

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

می‌توان بدترین زمان اجرا را به صورت $an^2 + bn + c$ نشان داد، که در آن $a$، $b$، و $c$ ثابت‌هایی هستند که به هزینه‌ی هر عبارت ($c_i$) بستگی دارند. بنابراین، این تابع یک *تابع درجه دوم* نسبت به $n$ است.

با این که معمولاً مانند مرتب‌سازی درجی زمان اجرای یک الگوریتم با یک ورودی خاص ثابت است، در فصل‌های بعد چند الگوریتم تصادفی جذاب خواهیم دید که رفتارشان، حتی برای یک ورودی خاص، متغیر است.

## تحلیل بدترین حالت و حالت متوسط

در تحلیل بالا از مرتب‌سازی درجی، هم به بهترین حالت توجه کردیم، که در آن آرایه‌ی ورودی از ابتدا به صورت صعودی مرتب شده بود، و هم بدترین حالت، که در آن آرایه‌ی ورودی به صورت نزولی مرتب شده بود. با این حال در ادامه‌ی این کتاب، فقط بر روی *زمان اجرای بدترین حالت* متمرکز خواهیم شد، یعنی بیشترین زمان مورد نیاز ممکن برای الگوریتم با یک ورودی با اندازه‌ی $n$. سه دلیل برای این کار وجود دارد.

بدترین حالت زمان اجرای یک الگوریتم، یک کران بالا برای زمان اجرای الگوریتم با هر ورودی است. دانستن آن، به ما این ضمانت را می‌دهد که زمان اجرا هیچ وقت بیشتر از آن طول نمی‌کشد. در این صورت نیازی نخواهیم داشت که یک حدس معقول برای زمان اجرا داشته باشیم، و امیدوار باشیم که وضعیت چندان بدتر از این حدس نخواهد شد.

- برای بعضی الگوریتم‌ها، در اغلب مواقع بدترین حالت زمان اجرا اتفاق می‌افتد. مثلاً در جستجوی یک پایگاه داده برای یک قطعه‌ی خاص از اطلاعات، وقتی که اطلاعات مورد نظر در پایگاه داده نباشد بدترین حالت زمان اجرا رخ می‌دهد. در بعضی کاربردها جستجو برای اطلاعات ناموجود بسیار رایج است.
- حالت متوسط زمان اجرا، معمولاً به همان بدی زمان اجرا در بدترین حالت است. فرض کنید $n$ عدد به صورت تصادفی انتخاب و مرتب‌سازی درجی را بر روی آن اجرا می‌کنیم. پیدا کردن جای $A[j]$ در زیرآرایه‌ی $A[۱..j-۱]$ چقدر طول می‌کشد؟ به طور متوسط، نصف عناصر $A[۱..j-۱]$ از $A[j]$ کوچک‌تر، و نصف از آن بزرگ‌تر هستند. پس به طور متوسط نیمی از زیرآرایه را باید بررسی کنیم، یعنی $t_j = j/۲$. اگر زمان اجرای متوسط را بر این اساس محاسبه کنیم، متوجه می‌شویم که برابر است با یک تابع درجه دو نسبت به اندازه‌ی ورودی، درست مانند بدترین حالت زمان اجرا.

در بعضی حالت‌های خاص، ممکن است بخواهیم *زمان اجرای متوسط*، یا *امیدریاضی زمان اجرا* را بدانیم. کاربرد تکنیک *تحلیل احتمالی* (probabilistic analysis) را بر روی الگوریتم‌های مختلف در طول این کتاب خواهیم دید. حیطه‌ی استفاده از تحلیل زمان متوسط محدود است، چرا که احتمالاً تعریف ورودی «متوسط» برای مسئله‌ای خاص مشخص نیست. اکثراً فرض می‌کنیم تمام ورودی‌های با یک اندازه تقریباً مشابه هستند. در عمل ممکن است این فرض درست نباشد، ولی بعضی مواقع می‌توانیم از یک *الگوریتم تصادفی* (randomized algorithm) استفاده کنیم، که با انتخاب‌های تصادفی ما را قادر می‌سازد تا به کمک تحلیل احتمالی، به امیدریاضی زمان اجرا دست یابیم. الگوریتم‌های تصادفی را در فصل ۵ و فصل‌های بعد از آن بررسی خواهیم کرد.

## مرتبه‌ی رشد

برای تحلیل رویه‌ی INSERTION-SORT از چند فرض ساده کننده برای ساده شدن تحلیل استفاده کردیم. ابتدا از هزینه‌ی واقعی هر عبارت صرف‌نظر کرده و آن‌ها را با ثابت‌های $c_i$ نشان دادیم. سپس دیدیم که حتی این ثابت‌ها هم از مقداری که ما نیاز داریم، جزئیات بیشتری به دست می‌دهند: بدترین حالت زمان اجرا برابر است با $an^۲+bn+c$، که در آن $a$، $b$، و $c$ ثابت‌هایی هستند که به $c_i$ها بستگی دارند. پس نه تنها از هزینه‌های واقعی، که از هزینه‌های ساده شده نیز چشم پوشی کردیم.

در این‌جا از یک ساده‌سازی دیگر نیز استفاده می‌کنیم. این ساده‌سازی *نرخ رشد* (rate of growth) یا *مرتبه‌ی رشد* (order of growth) زمان اجرا است، و در واقع این سرعت رشد زمان اجرا است که مورد نظر ماست. بنابراین در فرمول سرعت رشد، فقط جمله‌ی اول (در مثال بالا، $an^۲$) را در نظر می‌گیریم، چرا که جمله‌های با درجه‌ی پایین در $n$های بزرگ تأثیر چندانی ندارند. به علاوه در جمله‌ی اول هم از ضریب ثابت صرف نظر می‌کنیم، زیرا با رشد $n$، ضرایب ثابت اهمیت خود را در محاسبه‌ی کارایی در کامپیوتر از دست می‌دهند. برای مرتب‌سازی درجی وقتی از جمله‌های با درجه‌ی پایین‌تر و ضرایب ثابت صرف‌نظر کنیم، فقط عامل $n^۲$ از جمله‌ی اول باقی می‌ماند. می‌گوییم بدترین حالت زمان اجرای

الگوریتم مرتب‌سازی درجی از مرتبه‌ی $\theta(n^2)$ (بخوانید تتای $n^2$) است. در این فصل به صورت غیر رسمی از نماد $\theta$ استفاده می‌کنیم؛ این نماد در فصل ۳ به صورت دقیق تعریف خواهد شد.

معمولاً می‌گوییم الگوریتمی از الگوریتم دیگر کاراتر (سریع‌تر) است اگر زمان اجرای بدترین حالت آن، سرعت رشد کم‌تری داشته باشد. به دلیل صرف نظر از ضرایب ثابت و جمله‌های با درجه‌ی پایین، ممکن است این ارزیابی در ورودی‌های کوچک درست نباشد. ولی برای ورودی‌های به اندازه‌ی کافی بزرگ، مثلاً یک الگوریتم از مرتبه‌ی زمانی $\theta(n^2)$، در بدترین حالت بسیار سریع‌تر از یک الگوریتم از مرتبه‌ی زمانی $\theta(n^3)$ اجرا می‌شود.

## تمرین‌ها

۲-۲-۱ مرتبه‌ی زمانی تابع $\frac{n^3}{1000} - 100n^2 - 100n + 3$ را به وسیله‌ی نماد $\theta$ مشخص کنید.

۲-۲-۲ فرض کنید می‌خواهیم $n$ عدد را که در آرایه‌ی $A$ ذخیره شده‌اند، مرتب کنیم. راه حل زیر را در نظر بگیرید: ابتدا کوچک‌ترین عنصر را در $A$ پیدا می‌کنیم، و جای آن را با عنصر $A[1]$ عوض می‌کنیم. سپس دومین عنصر کوچک را یافته و جای آن را با $A[2]$ عوض می‌کنیم. همین کار را برای $n-1$ عنصر اول $A$ انجام می‌دهیم. یک رویه برای این الگوریتم بنویسید (نام این الگوریتم *مرتب‌سازی انتخابی* (selection sort) است). ثابت‌های حلقه‌ی این الگوریتم را مشخص کنید؟ چرا باید الگوریتم را فقط برای $n-1$ عنصر اول (به جای تمام $n$ عنصر) اجرا کنیم؟ بهترین و بدترین حالت زمان اجرای این الگوریتم را به وسیله‌ی نماد $\theta$ مشخص کنید.

۲-۲-۳ دوباره جستجوی خطی (تمرین ۲-۱-۳) را در نظر بگیرید. فرض کنید احتمال قرار داشتن تمام کلیدها در یک مکان خاص از آرایه یکسان است. در این صورت به طور متوسط چند عنصر باید بررسی شوند تا کلید مورد نظر پیدا شود؟ در بدترین حالت چه طور؟ بدترین حالت و حالت متوسط زمان اجرای جستجوی خطی (با استفاده از نماد $\theta$) چیست؟ جواب‌های خود را توجیه کنید.

۲-۲-۴ چطور می‌توان (تقریباً) هر الگوریتمی را طوری تغییر داد که زمان اجرای آن در بهترین حالت بسیار خوب باشد؟

## ۲-۳ طراحی الگوریتم‌ها

تکنیک‌های زیادی برای طراحی الگوریتم‌ها وجود دارد. در مرتب‌سازی درجی از رویکردی بر مبنای رشد استفاده کردیم: اگر زیرآرایه‌ی $A[1..j-1]$ مرتب شده باشد، با قرار دادن عنصر $A[j]$ در مکان مناسب، زیرآرایه‌ی $A[1..j]$ هم مرتب شده بود.

در این بخش یک رویکرد طراحی دیگر به نام «تقسیم و حل» (divide and conquer) را مورد بحث

قرار می‌دهیم، که در فصل ۴ با جزئیات کامل آن را بررسی خواهیم کرد. با استفاده از این رویکرد الگوریتمی برای مرتب‌سازی طراحی می‌کنیم که بدترین حالت زمان اجرای آن بسیار کم‌تر از مرتب‌سازی درجی است. یکی از مزایای الگوریتم‌هایی که با تکنیک تقسیم و حل طراحی می‌شوند، این است که می‌توان زمان اجرای آن‌ها را به راحتی به کمک تکنیک‌هایی که در فصل ۴ معرفی می‌شوند، تعیین کرد.

### ۲-۳-۱ رویکرد تقسیم و حل

بسیاری از الگوریتم‌های پرکاربرد، ساختاری *بازگشتی* (recursive) دارند: رویه‌ی آن‌ها برای حل یک مسئله، خود را یک یا چند بار به صورت بازگشتی فراخوانی می‌کنند تا زیرمسئله‌های بسیار مشابهی را حل کنند. این الگوریتم‌ها معمولاً رویکرد *تقسیم و حل* را دنبال می‌کنند: مسئله را به چندین زیرمسئله‌ی مشابه ولی با اندازه‌ی کوچک‌تر تقسیم کرده، زیرمسئله‌ها را به صورت بازگشتی حل می‌کنند، و سپس با ترکیب جواب زیرمسئله‌ها با یکدیگر جوابی برای مسئله‌ی اصلی می‌یابند.

الگوی تقسیم و حل در هر مرحله از بازگشت شامل سه مرحله‌ی زیر است:

- تقسیم مسئله به تعدادی زیرمسئله، که نمونه‌های کوچک‌تری از همان مسئله هستند.
- حل زیرمسئله‌ها به صورت بازگشتی، و یا به صورت غیر بازگشتی در صورت کوچک بودن اندازه‌ی زیرمسئله‌ها به اندازه‌ی کافی.
- ترکیب جواب زیرمسئله‌ها و تولید جواب مسئله‌ی اصلی.

الگوریتم *مرتب‌سازی ادغامی* (merge sort) دقیقاً از الگوی تقسیم و حل پیروی می‌کند. شکل کلی این الگوریتم به صورت زیر است.

- تقسیم: تقسیم دنباله‌ی $n$ عنصری مورد نظر به دو دنباله‌ی $n/2$ عنصری.
- حل: مرتب‌سازی دو زیردنباله به صورت بازگشتی و به وسیله‌ی الگوریتم مرتب‌سازی ادغامی.
- ترکیب: ادغام دو زیردنباله‌ی مرتب شده، و ارسال دنباله‌ی مرتب شده به خروجی به عنوان جواب.

این بازگشت[1] زمانی به پایین‌ترین مرحله‌ی خود می‌رسد که طول زیردنباله‌هایی که باید مرتب شوند یک باشد، که در این حالت نیازی به انجام عملیات بر روی زیرآرایه‌ها نداریم، چرا که هر دنباله به طول ۱ ذاتا مرتب شده است.

عمل کلیدی در الگوریتم مرتب‌سازی ادغامی، ترکیب دو زیردنباله‌ی مرتب شده است. برای انجام این کار از یک رویه‌ی کمکی به نام $\text{MERGE}(A,p,q,r)$ استفاده می‌کنیم، که در آن $A$ یک آرایه و $p$، $q$، و $r$ اندیس‌هایی در آرایه هستند، طوری که $p \le q < r$. این رویه فرض می‌کند آرایه‌های

[1] در طول این کتاب، از عبارت بازگشت هم برای کلمه‌ی "recurse" (به معنی فراخوانی بازگشتی) استفاده شده است، و هم برای کلمه‌ی "return" (به معنی خروج از تابع و ارسال مقدار نهایی به خروجی)، ولی همیشه می‌توانید با توجه به محتوای متن، این دو مفهوم را از یکدیگر تشخیص دهید. - م

$A[p..q]$ و $A[q+1..r]$ مرتب شده هستند، و آن‌ها را *ادغام* کرده و یک آرایه‌ی مرتب شده تولید می‌کند که جای زیرآرایه‌ی فعلی $A[p..r]$ را می‌گیرد.

رویه‌ی MERGE از مرتبه‌ی زمانی $\theta(n)$ است، که در آن $n = r - p + 1$ تعداد کل عناصری است که باید ادغام شوند، و به صورت زیر کار می‌کند: به ایده‌ی کارت‌های بازی بر می‌گردیم. فرض کنید دو دسته کارت که روی آن‌ها به بالا است بر روی میز قرار دارند. هر دو دسته مرتب شده هستند، و کوچک‌ترین کارت روی بقیه قرار دارد. می‌خواهیم دو دسته را در هم ادغام کنیم به طوری که دسته‌ی نهایی ایجاد شده، مرتب و به پشت روی میز قرار داشته باشد. قدم اصلی عبارت است از برداشتن کارت کوچک‌تر از بین دو کارت رویی دو دسته و گذاشتن آن در دسته‌ی خروجی به صورت پشت و رو. این مرحله را آن قدر ادامه می‌دهیم که یکی از دسته‌ها خالی شود، که در این حالت فقط کافی است که دسته‌ی باقی مانده را برداریم و پشت و رو بر روی دسته‌ی خروجی قرار دهیم. هر کدام از قدم‌های اصلی در زمان ثابت انجام می‌شوند، چرا که فقط دو کارت رویی را چک می‌کنیم. از آن جایی که حداکثر $n$ بار این مرحله را انجام می‌دهیم، کل ادغام در زمان $\theta(n)$ انجام می‌شود.

شبه‌کد زیر ایده‌ی بالا را پیاده سازی می‌کند، با این تفاوت که در این جا احتیاجی به بررسی تهی بودن دسته‌های ورودی نیست. برای این کار زیر هر یک از دسته‌ها، یک کارت *نگهبان* (sentinel) قرار می‌دهیم، که برای ساده‌تر شدن کد حاوی مقداری خاص است. در این جا از $\infty$ به عنوان مقدار نگهبان استفاده می‌کنیم. در این صورت هر جایی که یک کارت با مقدار $\infty$ رو شد، مقدار آن از کارت دیگر بزرگ‌تر خواهد بود، تا وقتی که هر دو کارت نگهبان رو شوند. ولی این اتفاق زمانی می‌افتد که همه‌ی کارت‌های غیر نگهبان در دسته‌ی خروجی قرار گرفته باشند. از آن جایی که می‌دانیم دقیقاً $r - p + 1$ کارت باید در دسته‌ی خروجی قرار گیرند، می‌توانیم هر گاه که این اتفاق افتاد عملیات را متوقف کنیم.

```
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       L[i] = A[p + i − 1]
6   for j = 1 to n2
7       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      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

رویه‌ی MERGE به صورت زیر کار می‌کند: خط ۱ طول زیرآرایه‌ی $A[p..q]$، یعنی $n_1$ را محاسبه

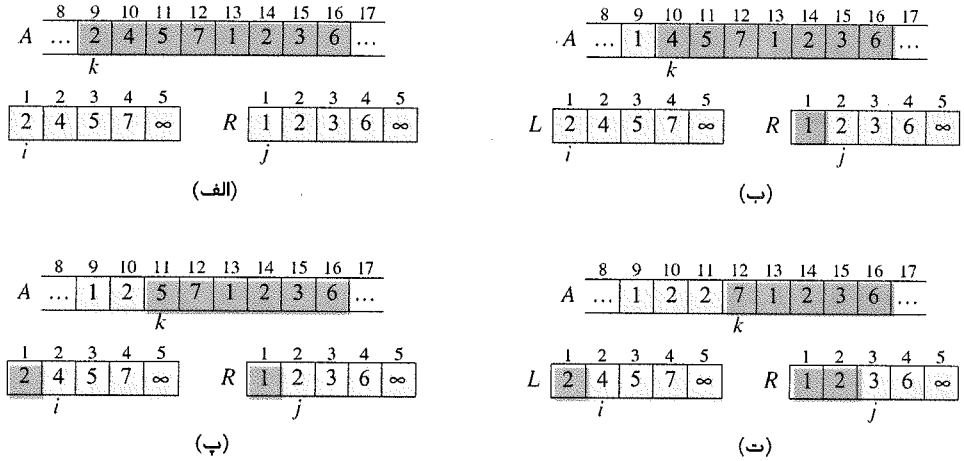
می‌کند، و خط ۲ طول زیرآرایه‌ی $A[q+۱..r]$، یعنی $n_۲$ را. سپس در خط ۳ آرایه‌های $L$ و $R$ (چپ و راست) را با طول‌های $n_۱+۱$ و $n_۲+۱$ می‌سازیم؛ فضای اضافی در هر آرایه مقدار نگهبان را ذخیره خواهد کرد. حلقه‌ی for در خطوط ۴-۵ زیرآرایه‌ی $A[p..q]$ را در $L[۱..n_۱]$ کپی می‌کند، و حلقه‌ی for خطوط ۶-۷، زیرآرایه‌ی $A[q+۱..r]$ را در $R[۱..n_۲]$. در خطوط ۸-۹ مقادیر نگهبان در انتهای آرایه‌های $L$ و $R$ قرار می‌گیرند. خطوط ۱۰-۱۷، همان طور که در شکل ۲-۳ مشخص شده است، $r-p+۱$ بار مرحله‌ی اصلی را با حفظ ثابت حلقه‌ی زیر انجام می‌دهند:

- در شروع هر بار اجرای حلقه‌ی for در خطوط ۱۲-۱۷، زیرآرایه‌ی $A[p..k-۱]$ شامل $k-p$ عنصر کوچک $L[۱..n_۱+۱]$ و $R[۱..n_۲+۱]$، به صورت مرتب شده است. به علاوه، $L[i]$ و $R[j]$ کوچک‌ترین عناصر آرایه‌های $L[۱..n_۱+۱]$ و $R[۱..n_۲+۱]$ هستند، و هنوز در $A$ کپی نشده‌اند.

باید نشان دهیم که این ثابت حلقه قبل از اولین اجرای حلقه‌ی for خطوط ۱۲-۱۷ برقرار است، هر تکرار حلقه، ثابت حلقه را حفظ می‌کند، و ثابت حلقه یک خصوصیت مفید برای نشان دادن صحت الگوریتم پس از پایان حلقه فراهم می‌کند.

- شروع: قبل از اولین تکرار حلقه، داریم $k=p$، پس زیرآرایه‌ی $A[p..k-۱]$ تهی است. این آرایه‌ی تهی حاوی $k-p=۰$ عنصر کوچک $L$ و $R$ است، و از آن جایی که $i=j=۱$، هر دو عنصر $L[i]$ و $R[j]$ کوچک‌ترین عناصر زیرآرایه‌های مربوطه هستند که هنوز در $A$ کپی نشده‌اند.
- ادامه: برای این که نشان دهیم هر تکرار حلقه، ثابت حلقه را حفظ می‌کند، اجازه دهید فرض کنیم $L[i]\le R[j]$. در این صورت $L[i]$ کوچک‌ترین عنصری است که هنوز در $A$ کپی نشده است. از آن جایی که $A[p..k-۱]$ حاوی $k-p$ عنصر کوچک است، بعد از این که در خط ۱۴ مقدار $L[i]$ در $R[j]$ کپی شد، زیرآرایه‌ی $A[p..k-۱]$ حاوی $p-k+۱$ عنصر کوچک خواهد بود. افزایش $k$ (در سرآیند حلقه‌ی for) و $i$ (در خط ۱۵) دوباره ثابت حلقه را برای تکرار بعدی برقرار می‌کند. در عوض اگر $L[i]>R[j]$، آن گاه در خطوط ۱۶-۱۷ عملیات مورد نیاز برای برقراری ثابت حلقه انجام خواهد شد.
- پایان: بعد از خروج از حلقه داریم $k=r+۱$. طبق ثابت حلقه زیرآرایه‌ی $A[p..k-۱]$، که همان $A[p..r]$ است، حاوی $k-p=r-p+۱$ عنصر کوچک آرایه‌های $L[۱..n_۱+۱]$ و $R[۱..n_۲+۱]$ به صورت مرتب شده خواهد بود. آرایه‌های $L$ و $R$ روی هم $n_۱+n_۲+۲=r-p+۳$ عنصر دارند، که تمام آن‌ها غیر از دو عنصر بزرگ در $A$ کپی شده‌اند، و این دو عنصر بزرگ، همان مقادیر نگهبان هستند.

باید نشان دهیم که این ثابت حلقه قبل از اولین تکرار حلقه‌ی for در خطوط ۱۲-۱۷ برقرار است، بعد از هر بار تکرار حلقه برقرار می‌ماند، و خصوصیت مورد نظر را برای اثبات درستی الگوریتم در پایان اجرای حلقه به دست می‌دهد.




**شکل ۲-۳** عملیات خطوط ۱۰-۱۷ در فراخوانی MERGE $(A,9,12,16)$، وقتی زیرآرایه‌ی $A[9..16]$ حاوی عناصر $\langle 1,4,5,7,1,2,3,6\rangle$ است. بعد از کپی و درج داده‌های نگهبان، آرایه‌ی $L$ حاوی عناصر $\langle 2,4,5,7,\infty\rangle$ خواهد بود، و آرایه‌ی $R$ حاوی عناصر $\langle 1,2,3,6,\infty\rangle$. مکان‌هایی که در $A$ با سایه‌ی کمرنگ مشخص شده‌اند حاوی مقادیر نهایی خود هستند، و مکان‌های با سایه‌ی کمرنگ در $L$ و $R$ مقادیری هستند که هنوز در $A$ کپی نشده‌اند. به طور کلی سایه‌ی کمرنگ همیشه نشان‌دهنده‌ی مقادیری است که در ابتدا در $A[9..16]$ قرار داشته‌اند، به علاوه‌ی مقادیر نگهبان. مکان‌های با سایه‌ی پررنگ در $A$ حاوی مقادیری هستند که باید روی آن‌ها کپی شود، و همین رنگ در $L$ و $R$ نشان‌دهنده‌ی مقادیری است که قبلا در $A$ کپی شده‌اند. (الف)-(ح) آرایه‌های $A$، $L$، و $R$ و اندیس‌های مربوط $k$، $i$ و $j$ قبل از هر بار تکرار حلقه در خطوط ۱۲-۱۷.

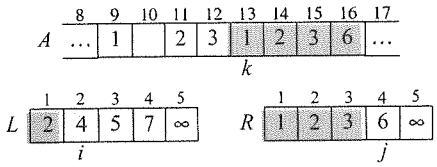
نشان می‌دهیم رویه‌ی MERGE در زمان $\theta(n)$ اجرا می‌شود، که در آن $n = r - p + 1$. توجه کنید که خطوط ۱-۳ و ۸-۱۱ در زمان ثابت اجرا می‌شوند، زمان اجرای حلقه‌ی for در خطوط ۴-۷ برابر است با $\theta(n_1 + n_2) = \theta(n)$[1]، حلقه‌ی for خطوط ۱۲-۱۷ تعداد $n$ بار تکرار می‌شود، و هر بار تکرار به زمان ثابت نیاز دارد.

اکنون می‌توانیم رویه‌ی MERGE را به عنوان یک زیرروال در الگوریتم مرتب‌سازی ادغامی به کار گیریم. رویه‌ی MERGE-SORT $(A,p,r)$ عناصر زیرآرایه‌ی $A[p..r]$ را مرتب می‌کند. اگر $p \geq r$، زیرآرایه حداکثر یک عنصر دارد و بنابراین مرتب شده است (نیاز به انجام هیچ عملیاتی نیست). در غیر این صورت مرحله‌ی تقسیم به سادگی $q$ را که $A[p..r]$ را به دو زیرآرایه تقسیم می‌کند، محاسبه خواهد کرد: زیرآرایه‌های $A[p..q]$ شامل $\lceil n/2 \rceil$ عنصر، و $A[q+1..r]$ شامل $\lceil n/2 \rceil$ عنصر.[2]

---

[1] در فصل ۳ خواهیم دید که چگونه با تساوی‌هایی که حاوی نماد $\theta$ هستند برخورد کنیم.

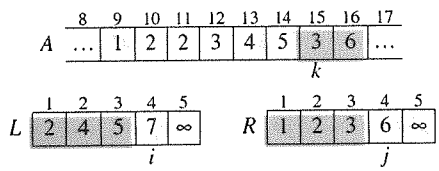
[2] عبارت $\lceil x \rceil$ نشان‌دهنده‌ی کوچک‌ترین عدد صحیح بزرگ‌تر یا مساوی $x$ است، و $\lfloor x \rfloor$ نشان‌دهنده‌ی بزرگ‌ترین عدد صحیح کوچک‌تر یا مساوی $x$. این نمادها در فصل ۳ تعریف خواهند شد. ساده‌ترین راه برای بررسی این که




(ج)



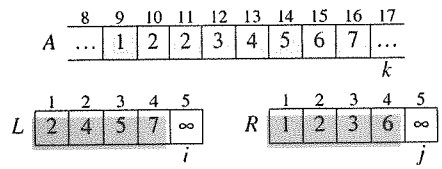

(ث)




(ح)




(چ)




(خ)

**شکل ۲-۳** ادامه

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

برای مرتب‌سازی تمام آرایه‌ی $A = \langle A[1], A[2], ..., A[n] \rangle$، فراخوانی اولیه‌ی MERGE-SORT $(A, 1, A.length)$ را انجام می‌دهیم، که در آن $A.lentgh = n$. شکل ۲-۴ عملیات رویه را به صورت پایین به بالا (bottom-up) نشان می‌دهد، که در آن $n$ توانی از ۲ است. این الگوریتم تشکیل شده است از ادغام جفت‌هایی از دنباله‌های تک عنصری برای ساختن دنباله‌های دو عنصری مرتب شده، ادغام جفت‌هایی از دنباله‌های دو عنصری برای ساختن دنباله‌های ۴ عنصری مرتب شده، و الی آخر، تا در نهایت دو دنباله‌ی $n/2$ عنصری با هم ادغام شده و دنباله‌ی مرتب شده‌ی نهایی با طول $n$ را تشکیل دهند.

## ۲-۳-۲ تحلیل الگوریتم‌های تقسیم و حل

وقتی یک الگوریتم حاوی فراخوانی خود به صورت بازگشتی است، معمولاً می‌توان زمان اجرای آن را

---

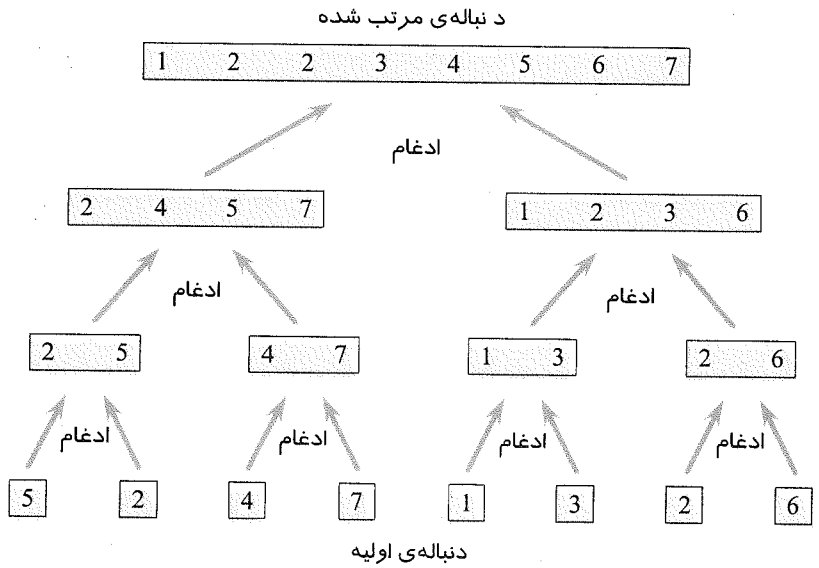
مقداردهی $q$ با $\lfloor (p+r)/2 \rfloor$، زیرآرایه‌های $A[p..q]$ و $A[q+1..r]$ را با اندازه‌های $\lceil n/2 \rceil$ و $\lfloor n/2 \rfloor$ تولید می‌کند، این است که چهار حالتی را که از زوج یا فرد بودن $p$ یا $r$ به وجود می‌آیند، بررسی کنیم.

به کمک یک *معادله‌ی بازگشتی* (recurrence equation) و یا به اختصار *بازگشت* (recurrence) توصیف کرد. در این روش زمان اجرای کلی برای مسئله‌ای با اندازه‌ی $n$ بر حسب زمان اجرا با ورودی‌های کوچک‌تر توصیف می‌شود. سپس می‌توانیم از ابزارهای ریاضی برای حل معادله‌ی بازگشتی و تعیین کرانی بر روی کارایی الگوریتم استفاده کنیم.

برای تعیین یک معادله‌ی بازگشتی برای یک الگوریتم تقسیم و حل می‌توانیم از سه مرحله‌ی الگوی اصلی (که قبلا ارائه شد) استفاده کنیم. مانند قبل فرض می‌کنیم $T(n)$ زمان اجرای الگوریتم بر روی یک ورودی با اندازه‌ی $n$ باشد. اگر اندازه‌ی مسئله به اندازه‌ی کافی کوچک باشد، مثلاً $n \le c$، که $c$ یک ثابت است، در این صورت الگوریتم در زمان ثابت جواب را تولید خواهد کرد، که آن را به صورت $\theta(1)$ می‌نویسیم. فرض کنید مرحله‌ی تقسیم $a$ زیرمسئله تولید می‌کند، که اندازه‌ی هر کدام $1/b$ اندازه‌ی مسئله‌ی اولیه است. (برای الگوریتم مرتب‌سازی ادغامی، هر دوی $a$ و $b$ برابر ۲ هستند، ولی الگوریتم‌های زیاد دیگری بر مبنای تقسیم و حل خواهیم دید که در آن‌ها داریم $a \neq b$.) برای حل زیرمسئله‌ای با اندازه‌ی $n/b$ به زمان $T(n/b)$ نیاز داریم، پس برای حل $a$ تا از آن‌ها، $aT(n/b)$ زمان صرف خواهد شد. اگر زمان تقسیم مسئله به زیرمسئله‌ها برابر $D(n)$ و زمان ترکیب جواب زیرمسئله‌ها و تولید جواب اصلی برابر $C(n)$ باشد، معادله‌ی بازگشتی زیر را خواهیم داشت:

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

در فصل ۴، روش حل معادله‌های بازگشتی به این شکل را خواهیم دید.




**شکل ۲-۴** عملیات مرتب‌سازی ادغامی روی آرایه‌ی $A = \langle 5,2,4,7,1,3,2,6 \rangle$. با پیش رفتن الگوریتم، از پایین به بالا طول آرایه‌های مرتب‌شده‌ای که با هم ادغام می‌شوند، افزایش می‌یابد.

## تحلیل مرتب‌سازی ادغامی

با این که رویه‌ی MERGE-SORT حتی وقتی تعداد عناصر زوج نباشد، به درستی کار می‌کند، اگر فرض کنیم اندازه‌ی مسئله‌ی اصلی توانی از ۲ است، تحلیل بر مبنای بازگشت ساده‌تر خواهد شد. در این صورت هر مرحله‌ی تقسیم دو زیردنباله دقیقاً با اندازه‌ی $n/b$ تولید می‌کند. در فصل ۴ خواهیم دید که این فرض بر روی مرتبه‌ی زمانی جواب معادله‌ی بازگشتی تأثیری نخواهد گذاشت.

برای محاسبه‌ی مقدار $T(n)$ (بدترین حالت زمان اجرای مرتب‌سازی ادغامی برای $n$ عدد) استدلال زیر را خواهیم داشت. اجرای مرتب‌سازی ادغامی با یک عنصر به زمان ثابت نیاز دارد. وقتی $n > ۱$ عنصر داشته باشیم، زمان اجرا را به صورت زیر خواهیم شکست:

- تقسیم: مرحله‌ی تقسیم فقط نقطه‌ی میانی آرایه را محاسبه می‌کند، که به زمان ثابت احتیاج دارد. بنابراین $D(n) = \theta(۱)$.
- حل: این مرحله به صورت بازگشتی دو زیرمسئله با اندازه‌ی $n/۲$ را محاسبه می‌کند، که $۲T(n/۲)$ به زمان اجرای الگوریتم اضافه خواهد کرد.
- ترکیب: قبلا هم دیدیم که رویه‌ی MERGE برای یک آرایه‌ی $n$ عنصری به زمان $\theta(n)$ احتیاج دارد، بنابراین $C(n) = \theta(n)$.

توابع $D(n)$ و $C(n)$ به ترتیب از مرتبه‌ی زمانی $\theta(n)$ و $\theta(۱)$ هستند، بنابرای جمع این دو در تحلیل مرتب‌سازی ادغامی برابر $\theta(n)$ خواهد بود. اضافه کردن این عبارت به $۲T(n/۲)$ از مرحله‌ی حل، معادله‌ی بازگشتی زیر را برای $T(n)$ (بدترین حالت زمان اجرای مرتب‌سازی ادغامی) به دست خواهد داد:

$$T(n) = \begin{cases} \theta(۱) & \text{اگر } n = ۱ \\ ۲T(n/۲) + \theta(n) & \text{اگر } n > ۱ \end{cases} \qquad (۲-۱)$$

در فصل ۴ قضیه‌ی اصلی (master theorem) را خواهیم دید و با استفاده از آن نشان می‌دهیم که $T(n) = \theta(n \lg n)$، که در آن $\lg n$ همان $\log_۲ n$ است. از آن جایی که رشد تابع لگاریتم از تمام توابع خطی کندتر است، با ورودی‌های به اندازه‌ی کافی بزرگ، مرتب‌سازی ادغامی با زمان اجرای $\theta(n \lg n)$، در بدترین حالت کارامدتر از مرتب‌سازی درجی با زمان اجرای $\theta(n^۲)$ است.

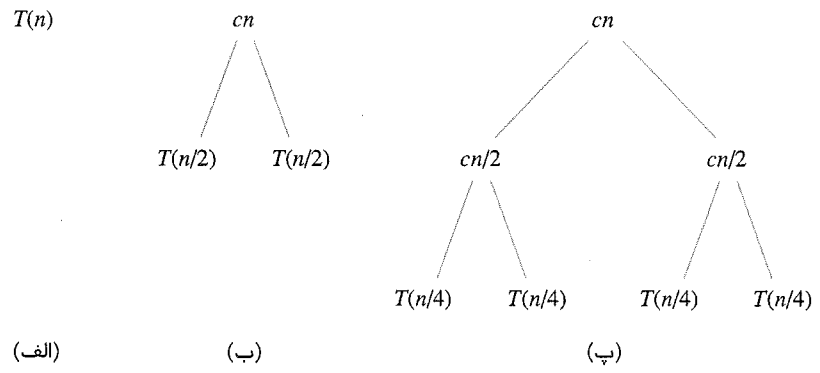
برای درک این که چرا جواب رابطه‌ی ۲-۱ عبارت است از $T(n) = \theta(n \lg n)$ احتیاجی به قضیه‌ی اصلی نداریم. اجازه دهید معادله‌ی بازگشتی ۲-۱ را به صورت زیر بازنویسی کنیم:

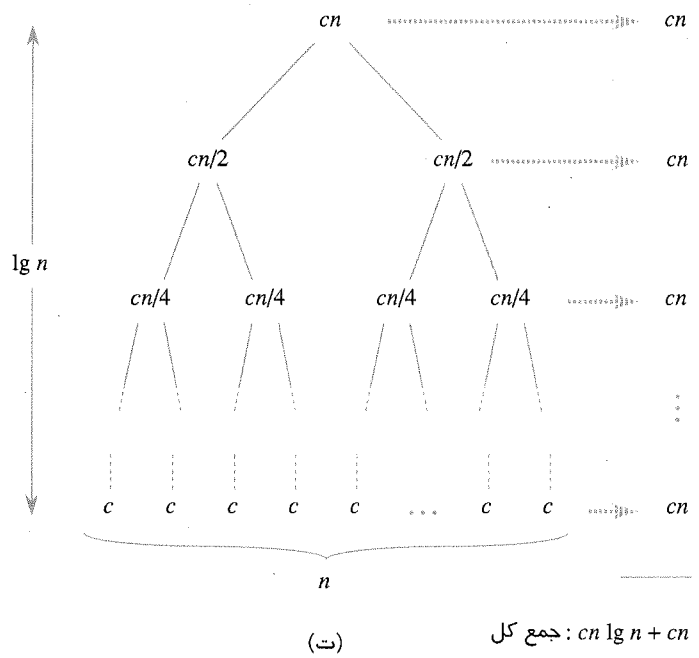
$$T(n) = \begin{cases} c & \text{اگر } n = ۱ \\ ۲T(n/۲) + cn & \text{اگر } n > ۱ \end{cases} \qquad (۲-۲)$$

که در آن ثابت $c$ نشان‌دهنده‌ی زمان مورد نیاز برای حل مسائل با اندازه‌ی ۱، و همچنین زمان صرف شده برای هر یک از عناصر آرایه در مراحل تقسیم و ترکیب است.[۱]

---

[۱] بعید است که دقیقاً یک ثابت هم نشان‌دهنده‌ی زمان حل مسئله‌های با اندازه‌ی ۱ باشد و هم زمان مراحل تقسیم و ترکیب




**شکل ۲-۵** ساختار یک درخت بازگشتی برای رابطه‌ی $T(n) = 2T(n/2) + cn$. در قسمت (الف) عبارت $T(n)$ را می‌بینیم که تدریجا در قسمت‌های (ب)-(ت) بسط داده می‌شود تا درخت بازگشتی تشکیل شود. در قسمت (ت) درخت کامل $\lg n + 1$ سطح دارد (ارتفاع آن $\lg n$ است)، و هزینه‌ی هر سطح برابر است با $cn$. بنابراین هزینه‌ی کلی $cn \lg n + cn$ را خواهیم داشت، با مرتبه‌ی زمانی $\theta(n \lg n)$.

---

برای هر عنصر آرایه. وقتی که می‌خواهیم کران بالای زمان اجرا را به دست آوریم، می‌توانیم با در نظر گرفتن $c$ به صورت ثابت بزرگ‌تر از میان این دو (یا وقتی که می‌خواهیم کران پایین زمان اجرا را به دست آوریم، با در نظر گرفتن $c$ به صورت ثابت کوچک‌تر) این مشکل را حل کنیم. هر دو کران زمانی از مرتبه‌ی $n \lg n$ خواهند بود، که زمان اجرای کلی $\theta(n \lg n)$ را به دست می‌دهد.

شکل ۲-۵ نحوه‌ی حل رابطه‌ی بازگشتی ۲-۲ را نشان می‌دهد. برای سادگی فرض می‌کنیم $n$ توانی از ۲ است. قسمت (الف) شکل نشان‌دهنده‌ی $T(n)$ است، که در قسمت (ب) به صورت یک درخت معادل (نشان دهنده‌ی رابطه‌ی بازگشتی) بسط داده شده است. عبارت $cn$ ریشه‌ی درخت است (هزینه در بالاترین مرحله‌ی بازگشت) و دو زیر درخت فرزند ریشه، دو رابطه‌ی بازگشتی کوچک‌تر $T(n/2)$ هستند. قسمت (پ) همین فرایند را نشان می‌دهد که یک مرحله جلوتر رفته و $T(n/2)$ را هم بسط می‌دهد. همین طور با بسط دادن هر گره در درخت و شکستن آن‌ها به اجزای تشکیل‌دهنده ادامه می‌دهیم، تا وقتی که اندازه‌ی مسئله‌ها به ۱ برسد، که هزینه‌ی هر کدام $c$ است. قسمت (ت) *درخت بازگشتی* (recursion tree) حاصل را نشان می‌دهد.

در مرحله‌ی بعد به صورت افقی هزینه‌ی گره‌های هر سطح را با هم جمع می‌کنیم. هزینه‌ی بالاترین سطح برابر است با $cn$. سطح بعد هزینه‌ای معادل با $c(n/2)+c(n/2)=cn$ دارد. به همین شکل هزینه‌ی سطح بعدی برابر است با $c(n/4)+c(n/4)+c(n/4)+c(n/4)=cn$، و الی آخر. به طور کلی سطح $i$ام زیر ریشه حاوی $2^i$ گره است، و هزینه‌ی هر گره برابر است با $c(n/2^i)$، بنابراین هزینه‌ی کلی $i$امین سطح زیر ریشه برابر است با $2^i c(n/2^i)=cn$.

تعداد کل سطوح درخت بازگشتی در شکل ۲-۵ برابر است با $\lg n+1$، که در آن $n$ برابر است با تعداد برگ‌های درخت، و متناسب است با اندازه‌ی ورودی. به کمک استقرا و به صورت غیر رسمی می‌توان به راحتی این واقعیت را نشان داد. حالت اولیه زمانی اتفاق می‌افتد که $n$ برابر ۱ باشد، که در این صورت فقط یک سطح وجود دارد. از آن جایی که $\lg 1=0$ تعداد سطوح درخت برابر خواهد بود با $\lg n+1$. حال طبق استقرا فرض کنید که تعداد سطوح در یک درخت بازگشتی با $2^i$ گره برابر است با $\lg 2^i+1=i+1$ (چرا که برای هر $i$، داریم $\lg 2^i=i$). از آن جایی که فرض کرده‌ایم اندازه‌ی ورودی توانی از ۲ است، اندازه‌ی بعدی که برای ورودی در نظر می‌گیریم، $2^{i+1}$ است. یک درخت با $n=2^{i+1}$ برگ، یک سطح بیشتر از درختی با $2^i$ برگ دارد، و بنابراین تعداد کل سطوح برابر است با $\lg 2^{i+1}+1=(i+1)+1$.

برای محاسبه‌ی هزینه‌ی کلی رابطه‌ی بازگشتی ۲-۲ به سادگی هزینه‌ی تمام سطوح را با یکدیگر جمع می‌کنیم. درخت بازگشتی $\lg n+1$ سطح دارد، که هزینه‌ی هر کدام $cn$ است. پس کل هزینه برابر خواهد بود با $cn(\lg n+1)=cn\lg n+cn$. با صرف نظر از جمله‌ی با درجه‌ی پایین‌تر و ثابت $c$، نتیجه‌ی دلخواه $\theta(n\lg n)$ را خواهیم داشت.

## تمرین‌ها

۲-۳-۱ با استفاده از شکل ۲-۴ به عنوان الگو، عملیات مرتب‌سازی ادغامی را روی آرایه‌ی $A=\langle 3,41,52,26,38,57,9,49\rangle$ نشان دهید.

۲-۳-۲ رویه‌ی MERGE را طوری بازنویسی کنید که احتیاجی به استفاده از مقادیر نگهبان نداشته باشد. در عوض هر گاه یکی از آرایه‌های $L$ یا $R$ تهی شد، با کپی کردن آرایه‌ی دیگر در $A$

کار را به پایان رساند.

۲-۳-۴ با استفاده از استقرای ریاضی نشان دهید که اگر $n$ توانی از ۲ باشد، جواب رابطه‌ی بازگشتی

$$T(n)=\begin{cases}۲ & \text{اگر } n=۲\\ ۲T(n/۲)+n & \text{اگر } n=۲^k \text{ برای } k>۱\end{cases}$$

برابر است با $T(n)=n\lg n$.

۲-۳-۴ مرتب‌سازی درجی را می‌توان به صورت یک رویه‌ی بازگشتی به صورت زیر تعریف کرد: برای مرتب کردن $A[۱..n]$، به صورت بازگشتی $A[۱..n-۱]$ را مرتب می‌کنیم و سپس $A[n]$ را در جای خود در آرایه‌ی مرتب شده‌ی $A[۱..n-۱]$ قرار می‌دهیم. یک رابطه‌ی بازگشتی برای زمان اجرای این نسخه از مرتب‌سازی درجی بنویسید.

۲-۳-۵ به مسئله‌ی جستجو باز می‌گردیم (تمرین ۲-۱-۳ را ببینید). دقت کنید که اگر دنباله‌ی $A$ مرتب شده باشد، می‌توانیم $v$ را با عنصر وسط آرایه مقایسه کرده و به این صورت، نیمی از دنباله را از رویه‌ی جستجو حذف کنیم. الگوریتم *جستجوی دودویی* (Binary search) به صورت بازگشتی این کار را تکرار، و در هر تکرار اندازه‌ی آرایه را نصف می‌کند. یک شبه‌کد به صورت تکراری یا بازگشتی برای جستجوی دودویی بنویسید. بحث کنید که بدترین حالت زمان اجرای جستجوی دودویی از مرتبه‌ی $\theta(\lg n)$ است.

۲-۳-۶ دقت کنید که حلقه‌ی while در خطوط ۵-۷ رویه‌ی INSERTION-SORT در بخش ۲-۱، از جستجوی خطی (آخر به اول) برای یافتن مکان عناصر در زیرآرایه‌ی مرتب شده‌ی $A[۱..j-۱]$ استفاده می‌کند. آیا می‌توانیم برای بهبود بدترین حالت زمان اجرای مرتب‌سازی درجی به $\theta(n\lg n)$، به جای جستجوی خطی از جستجوی دودویی استفاده کنیم؟

★ ۲-۳-۷ یک الگوریتم با زمان اجرای $\theta(n\lg n)$ ارائه دهید که با دریافت یک مجموعه‌ی $S$ از $n$ عدد و یک عدد دیگر $x$، تعیین می‌کند که آیا در $S$ دو عدد با مجموع $x$ وجود دارد یا خیر.

## مسائل

**۲-۱ *استفاده از مرتب‌سازی درجی برای آرایه‌های کوچک در مرتب‌سازی ادغامی***

با این که در بدترین حالت، زمان اجرای مرتب‌سازی ادغامی برابر با $\theta(n\lg n)$، و زمان اجرای مرتب‌سازی درجی برابر با $\theta(n^۲)$ است، ضرایب ثابت در مرتب‌سازی درجی باعث می‌شوند که زمان اجرای آن برای $n$های کوچک در عمل از مرتب‌سازی ادغامی بهتر باشد. بنابراین خوب است که در مرتب‌سازی ادغامی، زمانی که اندازه‌ی مسئله‌ها به اندازه‌ی کافی کوچک

می‌شود، از مرتب‌سازی درجی استفاده کنیم. نسخه‌ای از مرتب‌سازی ادغامی را در نظر بگیرید که در آن $n/k$ زیرآرایه‌ی با طول $k$ با استفاده از مرتب‌سازی درجی مرتب شده، و سپس به صورت استاندارد با هم ادغام می‌شوند. مقدار $k$ بعداً مشخص خواهد شد.

I. نشان دهید که با استفاده از مرتب‌سازی درجی می‌توان $n/k$ زیرلیست (هر کدام با طول $k$) را در بدترین حالت در زمان $\theta(nk)$ مرتب کرد.

II. نشان دهید که زیرلیست‌ها را می‌توان در بدترین حالت در زمان $\theta(n \lg(n/k))$ با هم ادغام کرد.

III. با اطلاع از این که این الگوریتم اصلاح شده در بدترین حالت در زمان $\theta(nk + n \lg(n/k))$ اجرا می‌شود، بزرگ‌ترین مقدار $k$ برحسب تابعی از $n$ (و با استفاده از نماد $\theta$) چقدر باید باشد که در آن صورت تابع با همان زمان اجرای مرتب‌سازی ادغامی استاندارد اجرا شود؟

IV. در عمل $k$ باید چطور انتخاب شود؟

## ۲-۳ *درستی مرتب‌سازی حبابی*

مرتب‌سازی حبابی (bubblesort) یک الگوریتم مرتب‌سازی معروف است. در این الگوریتم مرتباً جای دو عنصر همسایه (در صورت لزوم) با هم عوض می‌شود تا در نهایت آرایه مرتب شود.

```
BUBBLESORT(A)
1  for i = 1 to A.length
2      for j = A.length downto i + 1
3          if A[j] < A[j - 1]
4              then exchange A[j] with A[j –1]
```

I. فرض کنید $A'$ خروجی BUBBLESORT$(A)$ باشد. برای نشان دادن این که BUBBLESORT درست کار می‌کند، باید نشان دهیم که این رویه پایان می‌یابد، و پس از اتمام آن داریم:

$$A'[1] \le A'[2] \le \ldots \le A'[n]$$

که در آن $n = A.length$. چه چیز دیگری باید اثبات شود تا مطمئن شویم که BUBBLESORT به درستی مرتب‌سازی را انجام می‌دهد؟

در دو بخش بعد نامساوی ۲-۳ را اثبات خواهد شد.

II. با جزئیات کامل، یک ثابت حلقه برای حلقه‌ی for خطوط ۲-۴ تعریف کنید، و نشان دهید که این ثابت حلقه برقرار است. اثبات شما باید از ساختار ثابت حلقه که در این فصل معرفی شد، استفاده کند.

III. با استفاده از وضعیت پایانی ثابت حلقه که در بخش II اثبات شد، یک ثابت حلقه برای حلقه‌ی for خطوط ۱-۴ تعریف، وبا استفاده از آن نامساوی ۲-۳ را ثابت کنید. اثبات شما باید از ساختار ثابت حلقه که در این فصل معرفی شد، استفاده کند.

IV. بدترین حالت زمان اجرای مرتب‌سازی حبابی چیست؟ این زمان اجرا را با زمان اجرای مرتب‌سازی درجی مقایسه کنید.

### ۲-۳ درستی قانون هورنر (Horner's rule)

قطعه کد زیر قانون هورنر را برای تعیین مقدار یک چند جمله‌ای به شکل زیر پیاده‌سازی می‌کند:

$$P(x)=\sum_{k=0}^{n} a_k x^k = a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + x a_n)\ldots))$$

که در آن ضرایب $a_0, a_1, \ldots, a_n$ و یک مقدار برای $x$ داده شده است.

```
1  y = 0
2  for i = n downto 0
3      y = a_i + x · y
```

I. زمان اجرای این قطعه کد بر حسب نماد $\theta$ چقدر است؟

II. یک شبه‌کد برای پیاده‌سازی الگوریتم معمول تعیین مقدار چند جمله‌ای‌ها بنویسید، که در آن مقدار هر جمله از ابتدا محاسبه می‌شود. زمان اجرای این الگوریتم چقدر است؟ این زمان اجر را با زمان اجرای الگوریتم قانون هورنر مقایسه کنید.

III. ثابت حلقه‌ی زیر را در نظر بگیرید.

در شروع هر بار تکرار حلقه‌ی while در خطوط ۲-۳ ، داریم

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

مجموعی که هیچ جمله‌ای ندارد را برابر با ۰ در نظر بگیرید. با پیروی از ساختاری که در این فصل معرفی شد، از این ثابت حلقه استفاده کرده و نشان دهید که در پایان داریم $y = \sum_{k=0}^{n} a_k x^k$.

IV. نتیجه بگیرید که قطعه کد بالا به درستی مقدار چند جمله‌ای با ضرایب $a_0, a_1, \ldots, a_n$ را محاسبه می‌کند.

### ۲-۴ وارونگی‌ها

فرض کنید که $A[1..n]$ یک آرایه از $n$ عدد متمایز باشد. اگر داشته باشیم $i<j$ و $A[i]>A[j]$، آن گاه جفت $(i,j)$ یک *وارونگی* (inversion) در آرایه‌ی $A$ نامیده می‌شود.

I. پنج وارونگی در آرایه‌ی $\langle 2,3,8,6,1\rangle$ بیابید.

II. چه آرایه‌ای از مجموعه‌ی $\{1,2,\ldots,n\}$ دارای بیشترین وارونگی‌هاست؟ این آرایه چند وارونگی دارد؟

III. چه رابطه‌ای بین زمان اجرای مرتب‌سازی درجی و تعداد وارونگی‌های آرایه‌ی ورودی وجود دارد؟ جواب خود را توجیه کنید.

IV. یک الگوریتم با بدترین حالت زمان اجرای $\theta(n \lg n)$ طراحی کنید که تعداد وارونگی‌ها را برای هر جایگشتی از $n$ عنصر تعیین می‌کند. (*راهنمایی:* مرتب‌سازی ادغامی را اصلاح کنید.)


# رشد توابع

## ۳-۰ مقدمه

مرتبه‌ی رشد زمان اجرای یک الگوریتم، که در فصل ۲ تعریف شد، توصیف ساده‌ای از سرعت اجرای الگوریتم به دست می‌دهد و ما را قادر می‌سازد که کارایی الگوریتم‌های مختلف را با هم مقایسه کنیم. وقتی اندازه‌ی ورودی $n$ به اندازه‌ی کافی بزرگ شد، مرتب‌سازی ادغامی با بدترین حالت زمان اجرای $\theta(n \lg n)$ سریع‌تر از مرتب‌سازی درجی با بدترین حالت زمان اجرای $\theta(n^2)$ خواهد بود. با این که بعضی مواقع می‌توانیم زمان اجرای دقیق یک الگوریتم را محاسبه کنیم، همان طور که در فصل ۲ برای مرتب‌سازی درجی این کار را انجام دادیم، دقت مضاعف حاصل ارزش تلاش مورد نیاز را ندارد. برای ورودی‌های بزرگ ضرایب ثابت و جمله‌های با درجه‌ی پایین‌تر تحت تأثیر اندازه‌ی ورودی مغلوب می‌شوند، و اثر آن‌ها از بین می‌رود.

وقتی با هدف اهمیت دادن به مرتبه‌ی رشد زمان اجرا، فقط ورودی‌های به اندازه‌ی کافی بزرگ را در نظر می‌گیریم، در واقع داریم کارایی *حدی* (asymptotic) الگوریتم‌ها را مطالعه می‌کنیم. یعنی تنها نکته‌ی مهم برای ما سرعت رشد زمان اجرا است، وقتی که اندازه‌ی ورودی *به صورت حدی* و بدون کران رشد می‌کند. معمولاً الگوریتمی که به صورت حدی از بقیه‌ی الگوریتم‌ها کاراتر باشد، بهترین انتخاب برای تمام ورودی‌ها، غیر از ورودی‌های کوچک است.

در این فصل روش‌های استاندارد مختلفی برای ساده کردن تحلیل حدی الگوریتم‌ها خواهیم دید. بخش بعد با تعریف انواع نمادهای حدی شروع می‌شود، که یک نمونه از این نمادها (نماد $\theta$) را قبلا دیده‌ایم. سپس قراردادهای مختلفی را برای این نمادها تعریف می‌کنیم، که در کتاب حاضر هم از آن‌ها استفاده خواهد شد. نهایتاً رفتار توابعی را که معمولاً در تحلیل الگوریتم‌ها مشاهده می‌شود، مختصراً بررسی خواهیم کرد.

## ۱-۳ نمادهای حدی

نمادهایی که از آن‌ها برای توصیف زمان اجرای حدی یک الگوریتم استفاده می‌کنیم، از طریق توابعی تعریف می‌شوند که دامنه‌ی آن‌ها مجموعه‌ی اعداد طبیعی $N = \{0,1,2,...\}$ است. این نمادها برای توصیف تابع بدترین حالت زمان اجرا، که معمولاً فقط برای ورودی‌های با اندازه‌ی صحیح تعریف می‌شود، مناسب است. با این حال بعضی مواقع نیاز داریم که استفاده از این نمادهای حدی را به شکل‌های مختلف تغییر دهیم. مثلاً می‌توان به راحتی دامنه‌ی این نمادها را به اعداد حقیقی گسترش داد، و یا به زیرمجموعه‌ای از اعداد صحیح محدود کرد. ولی مهم است که معنی دقیق این نمادها را بدانیم تا این تغییر استفاده، به استفاده‌ی ناصحیح تبدیل نشود. در این بخش نمادهای حدی اولیه تعریف، و همچنین چند استفاده‌ی معمول دیگر آن‌ها معرفی خواهد شد.

### نمادهای حدی، توابع، و زمان‌های اجرا

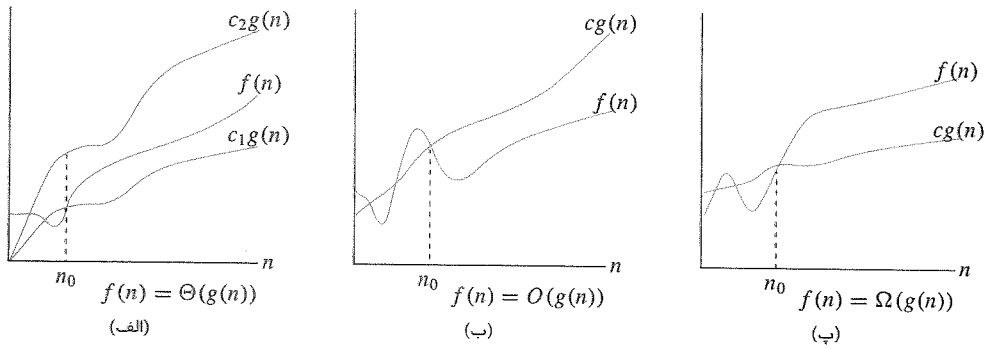
معمولاً از نمادهای حدی برای توصیف زمان اجرای الگوریتم‌ها استفاده می‌کنیم، همان طور که قبلا در مورد بدترین حالت زمان اجرای زمان اجرای $\theta(n^2)$ برای مرتب‌سازی درجی این کار را کردیم. ولی باید بدانید که نمادهای حدی در واقع برای توابع به کار می‌روند. به خاطر بیاورید که بدترین حالت زمان اجرای مرتب‌سازی درجی را به صورت $an^2+bn+c$، به ازای ثابت‌های $a$، $b$، و $c$، تعریف کردیم. با نوشتن این که زمان اجرای مرتب‌سازی درجی $\theta(n^2)$ است، بعضی جزئیات تابع را در نظر نگرفتیم. چون نمادهای حدی برای توابع به کار می‌روند، چیزی که ما به صورت $\theta(n^2)$ نوشتیم در واقع تابع $an^2+bn+c$ بود، که در این جا اتفاقاً توصیف بدترین حالت زمان اجرای مرتب‌سازی درجی است.

در این کتاب معمولاً برای توابعی از نمادهای حدی استفاده می‌کنیم که توصیف زمان اجرای یک الگوریتم باشند. ولی می‌توان نمادهای حدی را برای توابعی که خصوصیات دیگری از الگوریتم‌ها را توصیف می‌کنند هم به کار برد (به عنوان مثال، مقدار حافظه‌ی مصرفی توسط الگوریتم)، و یا حتی توابعی که هیچ ربطی به هیچ الگوریتمی ندارند!

حتی وقتی نمادهای حدی را برای زمان اجرای الگوریتم‌ها به کار می‌بریم، باید درک کنیم که هدف ما *کدام* زمان اجرا است. بعضی مواقع می‌خواهیم زمان اجرا را در بدترین حالت بدانیم. ولی معمولاً باید زمان اجرا را برای تمام ورودی‌ها توصیف کنیم. به عبارت دیگر، اکثراً می‌خواهیم یک عبارت کلی داشته باشیم که تمام ورودی‌ها را پوشش دهد، نه فقط ورودی‌های بدترین حالت را. بعداً نمادهای حدی را خواهیم دید که زمان اجرای تابع را مستقل از ورودی توصیف می‌کنند.

### نماد $\theta$

در فصل ۲ دیدیم که بدترین حالت زمان اجرای مرتب‌سازی درجی برابر است با $T(n) = \theta(n^2)$. اجازه دهید معنی دقیق این نماد را تعریف کنیم. برای یک تابع $g(n)$ داده شده، $\theta(g(n))$ را به صورت مجموعه‌ی توابعی تعریف می‌کنیم که




**شکل ۳-۱** مثال‌های گرافیکی از نمادهای $\theta$، $O$، و $\Omega$. در هر قسمت مقدار $n_0$ نشان داده شده کم‌ترین مقدار ممکن است؛ هر مقداری بزرگ‌تر آن را می‌توان به جای آن به کار برد. (الف) نماد $\theta$ یک تابع را بین ضرایب ثابتی محدود می‌کند. می‌نویسیم $f(n) = \theta(g(n))$ اگر ثابت‌های مثبت $n_0$، $c_1$، و $c_2$ موجود باشند به طوری که در سمت راست $n_0$، مقدار $f(n)$ همیشه بین (یا مماس با) $c_1g(n)$ و $c_2g(n)$ باشد. (ب) نماد $O$ یک کران بالا با یک ضریب ثابت برای یک تابع می‌دهد. می‌نویسیم $f(n) = O(g(n))$ اگر ثابت‌های مثبت $n_0$ و $c$ موجود باشند به طوری که در سمت راست $n_0$، مقدار $f(n)$ همیشه مماس و یا زیر $cg(n)$ باشد. (پ) نماد $\Omega$ یک کران پایین با یک ضریب ثابت برای یک تابع می‌دهد. می‌نویسیم $f(n) = \Omega(g(n))$ اگر ثابت‌های مثبت $n_0$ و $c$ موجود باشند به طوری که در سمت راست $n_0$، مقدار $f(n)$ همیشه مماس و یا بالای $cg(n)$ باشد.

$$\theta(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{l} \text{ثابت‌های مثبت } c_1 ,\ c_2 \text{ و } n_0 \text{ موجود باشند به طوری که} \\ \text{برای تمام } n \geq n_0 \text{ داشته باشیم } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \end{array} \right\}$$

یک تابع $f(n)$ به مجموعه‌ی $\theta(g(n))$ تعلق دارد اگر ثابت‌های مثبت $c_1$ و $c_2$ وجود داشته باشند به طوری که برای $n$های به اندازه‌ی کافی بزرگ، بتوان $f(n)$ را بین $c_1 g(n)$ و $c_2 g(n)$ محدود کرد. از آن جایی که $\theta(g(n))$ یک مجموعه است، برای این که مشخص کنیم $f(n)$ به $\theta(g(n))$ تعلق دارد، می‌توانیم بنویسیم "$f(n) \in \theta(g(n))$". ولی معمولاً این عبارت به شکل "$f(n) = \theta(g(n))$" نوشته می‌شود. این استفاده‌ی غیر معمول از علامت مساوی (=) برای نشان دادن عضویت در مجموعه ممکن است در ابتدا گیج کننده باشد، ولی بعداً در همین بخش خواهیم دید که مزایایی هم دارد.

شکل ۳-۱(الف) تصویری ملموس از توابع $f(n)$ و $g(n)$، در حالتی که $f(n) = \theta(g(n))$ نشان می‌دهد. برای تمام مقادیر $n$ در سمت راست $n_0$، مقدار $f(n)$ مماس یا بالای $c_1 g(n)$، و مماس یا زیر $c_2 g(n)$ است. به عبارت دیگر برای هر $n \geq n_0$، مقدار $f(n)$ برابر است با $g(n)$ در یک ضریب که این ضریب از حد خاصی فراتر نمی‌رود. می‌گوییم که $g(n)$ *یک کران حدی نزدیک* (asymptotically tight bound) برای $f(n)$ است.

[1] در نماد مجموعه‌ها، خط عمودی به معنی «به طوری که» می‌باشد.

تعریف $\theta(g(n))$ ایجاب می‌کند که هر عضو $f(n) \in \theta(g(n))$ *به صورت حدی نامنفی* باشد، یعنی برای $n$های به اندازه‌ی کافی بزرگ، $f(n)$ نامنفی باشد. (یک تابع به صورت حدی مثبت تابعی است که برای $n$های به اندازه‌ی کافی بزرگ، مثبت باشد.) به همین شکل خود تابع $g(n)$ باید به صورت حدی نامنفی باشد، وگرنه مجموعه‌ی $\theta(g(n))$ تهی خواهد بود. بنابراین در ادامه فرض می‌کنیم هر تابعی که نماد $\theta$ برای آن به کار می‌رود به صورت حدی نامنفی است. این فرض برای نمادهای حدی دیگری که در این کتاب به کار می‌روند نیز برقرار است.

در فصل ۲ مفهوم $\theta$ را به صورت غیر رسمی تعریف کردیم، که عبارت بود از حذف جمله‌های با درجه‌ی پایین و همچنین ضریب ثابت جمله‌ی با بالاترین درجه. اجازه دهید مختصرا با استفاده از تعریف دقیق نماد $\theta$، این حدس را تأیید کنیم که $\frac{1}{2}n^2 - 3n = \theta(n^2)$. برای این کار باید ثابت‌های $c_1$، $c_2$، و $n_0$ را تعیین کنیم به طوری که

$$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$$

برای هر $n \ge n_0$. اگر دو طرف تساوی را به $n^2$ تقسیم کنیم، خواهیم داشت

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

با انتخاب $c_2 \ge \frac{1}{2}$، نامساوی سمت راست برای هر $n \ge 1$ برقرار خواهد بود. به همین شکل با انتخاب $c_1 \le \frac{1}{14}$، نامساوی سمت چپ برای هر $n \ge 7$ برقرار خواهد بود. بنابراین با انتخاب $c_1 = \frac{1}{14}$، $c_2 = \frac{1}{2}$، و $n_0 = 7$، می‌توانیم نشان دهیم که $\frac{1}{2}n^2 - 3n = \theta(n^2)$. مطمئنا انتخاب‌های دیگری هم برای این ثابت‌ها وجود دارد، ولی نکته‌ی مهم این است که *حداقل* یک انتخاب وجود داشته باشد. توجه کنید که این ثابت‌ها به تابع $\frac{1}{2}n^2 - 3n$ بستگی دارند؛ تابعی دیگر از مرتبه‌ی $\theta(n^2)$ به ثابت‌های دیگری احتیاج خواهد داشت.

همچنین می‌توانیم از تعریف رسمی استفاده کنیم و نشان دهیم که $6n^3 \ne \theta(n^2)$. با استفاده از برهان خلف فرض کنید ثابت‌های $c_2$ و $n_0$ وجود دارند به طوری که برای هر $n \ge n_0$ داشته باشیم $6n^3 \le c_2 n^2$. ولی در این صورت با تقسیم دو طرف به $n^2$ خواهیم داشت $n \le c_2/6$، که نمی‌تواند برای $n$های به دلخواه بزرگ برقرار باشد، چرا که $c_2$ ثابت است.

به طور شهودی می‌توان از جمله‌های درجه پایین برای یک تابع مثبت حدی صرف‌نظر کرد، چرا که این جمله‌ها برای $n$های بزرگ ناچیز هستند. وقتی $n$ به اندازه‌ی کافی بزرگ باشد، کسر کوچکی از بالاترین جمله کافی است تا تأثیر جمله‌های با درجه‌ی پایین‌تر را کاملا از بین ببرد. بنابراین انتخاب مقداری که کمی از ضریب ثابت جمله‌ی با بالاترین درجه بزرگ‌تر است برای $c_2$، و انتخاب مقداری که کمی از آن کوچک‌تر است برای $c_1$، نامساوی‌های نماد $\theta$ را ارضا می‌کند. به همین صورت از ضریب ثابت بالاترین جمله هم می‌توان صرف نظر کرد، چرا که فقط $c_1$ و $c_2$ را به اندازه‌ی یک ضریب ثابت تغییر می‌دهد.

برای مثال یک تابع درجه دو به صورت $f(n)=an^2+bn+c$ را درنظر بگیرید، که در آن $a$، $b$، و $c$ ثابت هستند و $a>0$. با دور انداختن جمله‌های پایین‌تر و ضریب ثابت، خواهیم داشت $f(n)=\theta(n^2)$. به صورت رسمی برای نشان دادن این قضیه، ثابت‌های $c_1=a/4$، $c_2=7a/4$، و $n_0=2\cdot\max\left(\left(|b|/a\right)\cdot\sqrt{\left(|c|/a\right)}\right)$ را در نظر می‌گیریم. خواننده می‌تواند بررسی کند که برای $n\geq n_0$ رابطه‌ی $0\leq c_1n^2\leq an^2+bn+c\leq c_2n^2$ برقرار است. به طور کلی برای تمام چندجمله‌ای‌های $p(n)=\sum_{i=0}^{d}a_in^i$، که در آن $a_i$ها ثابت هستند و $a_d>0$، داریم $p(n)=\theta(n^d)$ (مسئله‌ی ۳-۱ را ببینید).

از آن جایی که تمام ثابت‌ها چندجمله‌ای‌های درجه ۰ هستند، می‌توانیم هر تابع ثابتی را به صورت $\theta(n^0)$، یا $\theta(1)$ نشان دهیم. نماد دوم تا حدودی ابهام دارد، چرا که مشخص نیست در آن چه متغیری به سمت بی‌نهایت میل می‌کند.[1] معمولاً وقتی عبارت $\theta(1)$ را به کار می‌بریم، منظور یا یک ثابت است یا یک تابع ثابت نسبت به متغیری خاص.

## نماد $O$

نماد $\theta$ به صورت حدی یک تابع را از بالا و پایین محدود می‌کند. وقتی که فقط یک کران بالای حدی داشته باشیم، از نماد $O$ استفاده می‌کنیم. برای یک تابع $g(n)$ داده شده، $O(g(n))$ (بخوانید Oی بزرگ $g(n)$، یا Oی $g(n)$) را به این صورت تعریف می‌کنیم:

$$O(g(n))=\left\{ f(n) \;\middle|\; \begin{array}{l}\text{ثابت‌های مثبت } c \text{ و } n_0 \text{ موجود باشند به طوری که}\\ \text{برای هر } n\geq n_0 \text{ داشته باشیم } 0\leq f(n)\leq cg(n)\end{array} \right\}.$$

از نماد $O$ برای تعیین یک کران بالا با ضریب ثابت برای تابع استفاده می‌کنیم. شکل ۳-۱ (ب) نماد $O$ را به صورت شهودی نشان می‌دهد. برای تمام مقادیر $n$ در سمت راست $n_0$، مقدار تابع $f(n)$ مماس یا زیر $g(n)$ است.

برای نشان دادن این که یک تابع عضوی از مجموعه‌ی $O(g(n))$ است می‌نویسیم $f(n)=O(g(n))$. توجه داشته باشید که $f(n)=\theta(g(n))$ نتیجه می‌دهد $f(n)=O(g(n))$، چرا که نماد $\theta$ حالت خاصی از نماد $O$ است. اگر بخواهیم از نظریه‌ی مجموعه‌ها استفاده کنیم، خواهیم داشت $\theta(g(n))\subseteq O(g(n))$. بنابراین اثبات بالا که نشان می‌داد هر تابع درجه دو عضو مجموعه‌ی $\theta(n^2)$ است، همچنین نشان می‌دهد که هر تابع درجه دو عضو $O(n^2)$ هم هست. نکته‌ی جالب‌تر این که هر تابع خطی مانند $an+b$ که در آن $a>0$ نیز عضو $O(n^2)$ است، و می‌توان به سادگی با قرار دادن

---

[1] مشکل واقعی این است که در نمادهای معمول ما برای توابع، تفاوت میان توابع و مقادیر مشخص نیست. در جبر $\lambda$ ($\lambda$-calculus)، پارامترهای توابع به صورت صریح مشخص شده‌اند: تابع $n^2$ را می‌توان به صورت $\lambda n\cdot n^2$ و یا حتی $\lambda r\cdot r^2$ نوشت. با این حال توصیف یک نماد دقیق‌تر اعمال جبری را پیچیده خواهد کرد، و بنابراین ترجیح خواهیم داد که از همین روش نادرست استفاده کنیم.

$c = a + |b|$ و $n_0 = \max(1, -b/a)$ آن را اثبات کرد.

برای بعضی خواننده‌ها که قبلا از نماد $O$ استفاده کرده‌اند ممکن است عجیب باشد که مثلاً می‌گوییم $n = O(n^2)$. در ادبیات الگوریتم‌ها از نماد $O$ برای نشان دادن حدود نزدیک استفاده می‌شود، یعنی همان چیزی که ما برای نماد $\theta$ تعریف کردیم. با این حال در این کتاب هر جا می‌نویسیم $f(n) = O(g(n))$، صرفا منظورمان این است که ضریبی ثابت از $g(n)$ یک کران بالای حدی برای $f(n)$ است، و چیزی در مورد میزان نزدیکی این حد نگفته‌ایم. اکنون تفاوت میان کران‌های بالای حدی و کران‌های حدی نزدیک در ادبیات الگوریتم‌ها استاندارد شده است.

با استفاده از نماد $O$ معمولاً می‌توانیم زمان اجرای یک الگوریتم را فقط با نگاهی کلی به ساختار الگوریتم تخمین بزنیم. به عنوان مثال ساختار دو حلقه‌ی for تودرتو در الگوریتم مرتب‌سازی درجی در فصل ۲، بی درنگ یک کران بالای $O(n^2)$ را برای بدترین حالت زمان اجرای تابع نتیجه می‌دهد: هزینه‌ی هر بار اجرای حلقه‌ی for داخلی از بالا توسط $O(1)$ (ثابت) محدود شده است، اندیس‌های $i$ و $j$ هر دو حداکثر $n$ هستند، و حلقه‌ی داخلی حداکثر یک بار برای $n^2$ جفت مقدار $i$ و $j$ اجرا می‌شود.

از آن جایی که نماد $O$ یک کران بالا مشخص می‌کند، وقتی از آن برای تعیین کران بدترین حالت زمان اجرای یک الگوریتم استفاده می‌کنیم، یک کران برای تمام ورودی‌های الگوریتم خواهیم داشت — عبارت کلی که قبلا در مورد آن بحث کردیم. بنابراین کران $O(n^2)$ برای بدترین حالت زمان اجرای مرتب‌سازی درجی، برای بقیه‌ی ورودی‌های آن هم قابل استفاده است. ولی کران $\theta(n^2)$ برای بدترین حالت زمان اجرای مرتب‌سازی درجی، کران $\theta(n^2)$ را برای تمام ورودی‌ها نتیجه نمی‌دهد. به عنوان مثال در فصل ۲ دیدیم که اگر ورودی از قبل مرتب شده باشد مرتب‌سازی درجی در زمان $\theta(n)$ اجرا می‌شود.

اصولا از نظر فنی درست نیست که بگوییم زمان اجرای مرتب‌سازی درجی $O(n^2)$ است، چرا که برای بعضی از ورودی‌ها این طور نیست، و زمان اجرا برای ورودی‌های مختلف متفاوت است. وقتی می‌گوییم «زمان اجرا $O(n^2)$ است»، منظور این است که یک تابع $f(n)$ از مرتبه‌ی $O(n^2)$ وجود دارد به طوری که برای هر مقداری از $n$، مستقل از این که کدام ورودی با اندازه‌ی $n$ انتخاب شده است، زمان اجرا از بالا توسط مقدار $f(n)$ محدود شده است. به عبارت دیگر منظور این است که بدترین حالت زمان اجرای الگوریتم $O(n^2)$ است.

### نماد $\Omega$

همان طور که نماد $O$ یک کران *بالای* حدی برای تابع مشخص می‌کند، نماد $\Omega$ یک *کران پایین حدی* برای تابع تعیین می‌کند. برای تابع داده شده‌ی $g(n)$، منظور از $\Omega(g(n))$ (بخوانید امگای بزرگ $g(n)$، یا امگای $g(n)$) مجموعه‌ی توابع زیر است:

$$\Omega(g(n)) = \left\{ f(n) \middle| \begin{array}{l} \text{ثابت‌های مثبت } c \text{ و } n_0 \text{ موجود باشند به طوری که} \\ \text{برای هر } n \geq n_0 \text{ داشته باشیم } 0 \leq cg(n) \leq f(n) \text{ .} \end{array} \right\}$$

شکل ۳-۱ (پ) نماد $\Omega$ را به صورت شهودی نشان می‌دهد. برای تمام مقادیر $n$ در سمت راست $n_0$، مقدار $f(n)$ مماس یا بالای $cg(n)$ است.

از روی تعریف نمادهای حدی که قبلا دیدیم، می‌توانیم به راحتی قضیه‌ی مهم زیر را اثبات کنیم (تمرین ۳-۱-۵ را ببینید).

**قضیه‌ی ۳-۱** برای هر دو تابع $f(n)$ و $g(n)$ داریم $f(n)=\theta(g(n))$ اگر و تنها اگر $f(n)=O(g(n))$ و $f(n)=\Omega(g(n))$.

به عنوان یک مثال از کاربرد این قضیه، اثبات این که برای هر ثابت $a$، $b$، و $c$ داریم $an^2+bn+c=\theta(n^2)$، بی‌درنگ نتیجه می‌دهد $an^2+bn+c=\Omega(n^2)$ و $an^2+bn+c=O(n^2)$.

در عمل به جای استفاده از قضیه‌ی ۳-۱ برای تعیین کران بالا و کران پایین از روی کران حدی نزدیک، مانند مثال بالا، معمولاً از آن برای تعیین کران حدی نزدیک از روی کران بالا و کران پایین استفاده می‌کنیم.

وقتی می‌گوییم *زمان اجرای* (بدون پیشوند) یک الگوریتم از مرتبه‌ی $\Omega(g(n))$ است، منظور این است که *مستقل از ورودی با اندازه‌ی $n$ که به الگوریتم می‌دهیم*، برای $n$های به اندازه‌ی کافی بزرگ، زمان اجرا روی آن ورودی حداقل برابر با یک ثابت ضرب در $g(n)$ است. در واقع داریم یک کران پایین بر روی بهترین حالت زمان اجرای یک الگوریتم می‌دهیم. برای مثال بهترین حالت زمان اجرای مرتب‌سازی درجی $\Omega(n)$ است، که بر این دلالت دارد که زمان اجرای مرتب‌سازی درجی $\Omega(n)$ است.

بنابراین زمان اجرای مرتب‌سازی درجی به هر دو کران $\Omega(n)$ و $O(n^2)$ تعلق دارد، چرا که همیشه بین یک تابع خطی برحسب $n$ و یا یک تابع درجه دو برحسب $n$ خواهد بود. به علاوه این دو زمان اجرا تا حد ممکن به زمان اجرای واقعی نزدیک هستند: مثلاً زمان اجرای مرتب‌سازی درجی از مرتبه‌ی $\Omega(n^2)$ نیست، چرا که یک ورودی وجود دارد که مرتب‌سازی درجی در زمان $\theta(n)$ آن را اجرا می‌کند (مثلاً وقتی که آرایه از قبل مرتب شده باشد). با این حال متناقض نیست اگر بگوییم بگوییم *بدترین حالت* زمان اجرای مرتب‌سازی درجی از مرتبه‌ی $\Omega(n^2)$ است، چرا که یک ورودی وجود دارد که الگوریتم آن را در زمان $\Omega(n^2)$ اجرا می‌کند.

## نمادهای حدی در تساوی‌ها و نامساوی‌ها

قبلا دیدیم که چطور می‌توان از نمادهای حدی در فرمول‌های ریاضی استفاده کرد. مثلاً هنگام معرفی نماد $O$، نوشتیم "$n=O(n^2)$". همچنین ممکن است بنویسیم $2n^2+3n+1=2n^2+\theta(n)$. چطور می‌توان این فرمول‌ها را تفسیر کرد؟

وقتی یک نماد حدی به تنهایی (یعنی بدون یک فرمول بزرگ‌تر) در سمت راست یک تساوی (یا نامساوی) قرار می‌گیرد، مانند $n=O(n^2)$، در این صورت همان طور که قبلا هم ذکر شد، منظور از علامت مساوی عضویت در مجموعه است: $n\in O(n^2)$. با این حال به طور کلی وقتی نماد حدی در

یک فرمول قرار می‌گیرد، آن را به صورت یک تابع ناشناس تفسیر می‌کنیم که جزئیات آن تابع برای ما اهمیتی ندارد. به عنوان مثال فرمول $2n^2+3n+1=2n^2+\theta(n)$ یعنی $2n^2+3n+1=2n^2+f(n)$، که در آن $f(n)$ تابعی در مجموعه‌ی $\theta(n)$ است. در مثال بالا $f(n)=3n+1$، که عضو مجموعه‌ی $\theta(n)$ است.

استفاده از نمادهای حدی بدین شکل، به ما کمک می‌کند که از جزئیات غیر ضروری و شلوغی در فرمول‌ها پرهیز کنیم. برای مثال در فصل ۲ بدترین حالت زمان اجرای مرتب‌سازی ادغامی را به صورت فرمول بازگشتی زیر تعریف کردیم:

$$T(n)=2T(n/2)+\theta(n)$$

اگر فقط به رفتار حدی $T(n)$ علاقه‌مند باشیم، توصیف تمام جملات درجه‌ی پایین‌تر به صورت دقیق هیچ فایده‌ای نخواهد داشت؛ تمام جزئیات لازم در جمله‌ی $\theta(n)$ وجود دارد.

تعداد توابع ناشناس در یک عبارت برابر است با تعداد دفعاتی که نماد حدی در آن عبارت ظاهر می‌شود. به عنوان مثال در عبارت

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

فقط یک تابع ناشناس وجود دارد (تابعی از $i$). پس این عبارت با عبارت $O(1)+O(2)+\ldots+O(n)$ یکسان نیست، که البته عبارت دوم تفسیر واضحی ندارد.

در بعضی موارد، نمادهای حدی در سمت چپ یک تساوی ظاهر می‌شوند، مانند

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

این قبیل تساوی‌ها را با استفاده از این قانون تفسیر می‌کنیم: *مستقل از این که توابع ناشناس چگونه در سمت چپ علامت تساوی انتخاب شوند، می‌توان توابع سمت راست تساوی را طوری انتخاب کرد که تساوی درست باشد.* بنابراین معنی مثال بالا این است که برای هر تابع $f(n)\in\theta(n)$، یک تابع $g(n)\in\theta(n^2)$ وجود دارد به طوری که برای هر $n$ داریم $2n^2+f(n)=g(n)$. به عبارت دیگر سمت راست تساوی سطح پایین‌تری از جزئیات نسبت به سمت چپ تساوی فراهم می‌آورد.

همچنین می‌توان چندین رابطه از این نوع را به صورت زنجیری پشت سر هم قرار داد، مانند

$$2n^2+3n+1=2n^2+\theta(n)=\theta(n^2)$$

می‌توانیم هر تساوی را با استفاده از قوانین بالا به صورت جداگانه تفسیر کنیم. تساوی اول می‌گوید که *تابعی* مانند $f(n)\in\theta(n)$ وجود دارد که برای هر $n$ داریم $2n^2+3n+1=2n^2+f(n)$. معنی تساوی دوم این است که برای هر تابع $g(n)\in\theta(n)$ (مانند $f(n)$ در تساوی قبل)، یک تابع مانند $h(n)\in\theta(n^2)$ وجود دارد که برای $n$ داریم $2n^2+g(n)=h(n)$. توجه کنید که این تفسیر نتیجه می‌دهد که $2n^2+3n+1=\theta(n^2)$، که همان چیزی است که این تساوی‌های زنجیری به طور ضمنی القا می‌کنند.

### نماد $o$

کران بالای حدی که توسط نماد $O$ توصیف می‌شود، ممکن است نزدیک باشد و یا نباشد. کران $2n^2 = O(n^2)$ به صورت حدی نزدیک است، ولی کران $2n = O(n^2)$ این طور نیست. بنابراین از نماد $o$ استفاده می‌کنیم تا کران‌های بالای حدی را نشان دهیم که نزدیک نیستند. به صورت رسمی $o(g(n))$ (بخوانید اوی کوچک $g(n)$) را به صورت مجموعه‌ی زیر تعریف می‌کنیم:

$$o(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{r} \text{برای هر ثابت } c>0 \text{، یک ثابت } n_0>0 \text{ موجود باشد به طوری} \\ \text{که برای تمام } n \geq n_0 \text{ داشته باشیم } 0 \leq f(n) < cg(n) \end{array} \right\}$$

مثلاً $2n = o(n^2)$، ولی $2n^2 \neq o(n^2)$.

تعریف نماد $O$ مشابه نماد $o$ است. تفاوت اصلی در این است که در $f(n) = O(g(n))$، کران $0 \leq f(n) \leq cg(n)$ برای یک ثابت $c>0$ برقرار است، ولی در $f(n) = o(g(n))$، کران $0 \leq f(n) < cg(n)$ باید برای هر $c>0$ برقرار باشد. به طور شهودی در نماد $o$، با بزرگ شدن $n$ تابع $f(n)$ از $g(n)$ دور می‌شود؛ یعنی

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \qquad (۱-۳)$$

در بعضی کتاب‌ها از این حد برای تعریف نماد $o$ استفاده می‌شود؛ در تعریف این کتاب، علاوه بر تعریف بالا، تابع ناشناس باید به صورت حدی نامنفی باشد.

### نماد $\omega$

متشابهاً نماد $\omega$ نسبت به $\Omega$، مانند نماد $o$ نسبت به $O$ است. از نماد $\omega$ استفاده می‌کنیم تا کران‌های پایینی را نشان دهیم که نزدیک نیستند. یک روش برای تعریف این نماد به صورت زیر است:

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

با این حال، به صورت رسمی نماد $\omega(g(n))$ (بخوانید امگای کوچک $g(n)$) را به صورت مجموعه‌ی زیر تعریف می‌کنیم:

$$\omega(g(n)) = \left\{ f(n) \;\middle|\; \begin{array}{r} \text{برای هر ثابت } c>0 \text{ یک ثابت } n_0>0 \text{ موجود باشد به طوری} \\ \text{که برای تمام } n \geq n_0 \text{ داشته باشیم } 0 \leq cg(n) < f(n). \end{array} \right\}$$

برای مثال $n^2/2 = \omega(n)$، ولی $n^2/2 \neq \omega(n^2)$. رابطه‌ی $f(n) = \omega(g(n))$ ایجاب می‌کند که

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

اگر [illegible] موجود باشد. یعنی با نزدیک شدن $n$ به بی‌نهایت، $f(n)$ نسبت به $g(n)$ به صورت [illegible] بزرگ می‌شود.

## مقایسه‌ی توابع

بسیاری از خصوصیات رابطه‌ای اعداد حقیقی برای مقایسه‌ی حدود هم قابل استفاده هستند. در رابطه‌های زیر، فرض کنید که $f(n)$ و $g(n)$ به صورت حدی مثبت هستند.

**تعدی:**

$$f(n)=\theta(g(n)) \text{ و } g(n)=\theta(h(n)) \text{ نتیجه می‌دهد } f(n)=\theta(h(n))،$$
$$f(n)=O(g(n)) \text{ و } g(n)=O(h(n)) \text{ نتیجه می‌دهد } f(n)=O(h(n))،$$
$$f(n)=\Omega(g(n)) \text{ و } g(n)=\Omega(h(n)) \text{ نتیجه می‌دهد } f(n)=\Omega(h(n))،$$
$$f(n)=o(g(n)) \text{ و } g(n)=o(h(n)) \text{ نتیجه می‌دهد } f(n)=o(h(n))،$$
$$f(n)=\omega(g(n)) \text{ و } g(n)=\omega(h(n)) \text{ نتیجه می‌دهد } f(n)=\omega(h(n)).$$

**انعکاس پذیری:**

$$f(n)=\theta(f(n))،$$
$$f(n)=O(f(n))،$$
$$f(n)=\Omega(f(n)).$$

**تقارن:**

$$f(n)=\theta(g(n)) \text{ اگر و فقط اگر } g(n)=\theta(f(n)).$$

**تقارن ترانهاده:**

$$f(n)=O(g(n)) \text{ اگر و فقط اگر } g(n)=\Omega(f(n))،$$
$$f(n)=o(g(n)) \text{ اگر و فقط اگر } g(n)=\omega(f(n)).$$

از آن جایی که این خواص برای نمادهای حدی برقرار است، می‌توان بین مقایسه‌ی توابع $f$ و $g$ و مقایسه‌ی اعداد حقیقی $a$ و $b$ شباهت‌هایی برقرار کرد:

$$f(n)=O(g(n)) \text{ مشابه است با } a\le b،$$
$$f(n)=\Omega(g(n)) \text{ مشابه است با } a\ge b،$$
$$f(n)=\theta(g(n)) \text{ مشابه است با } a=b،$$
$$f(n)=o(g(n)) \text{ مشابه است با } a<b،$$
$$f(n)=\omega(g(n)) \text{ مشابه است با } a>b.$$

می‌گوییم $f(n)$ *به صورت حدی کوچک‌تر* از $g(n)$ است اگر $f(n)=o(g(n))$، و $f(n)$ *به صورت حدی بزرگ‌تر* از $g(n)$ است اگر $f(n)=\omega(g(n))$.

با این حال یکی از خصوصیات اعداد حقیقی برای نمادهای حدی برقرار نیست:

- **سه بخشی بودن (trichotomy)**: برای هر دو عدد حقیقی $a$ و $b$، دقیقاً یکی از سه رابطه‌ی زیر برقرار است: $a<b$، $a=b$، یا $a>b$.

هر دو عدد حقیقی قابل مقایسه هستند، ولی نمی‌توان هر دو تابع را به صورت حدی مقایسه کرد.

یعنی ممکن است دو تابع $f(n)$ و $g(n)$ وجود داشته باشند که هیچ کدام از دو حالت $f(n)=O(g(n))$ و $f(n)=\Omega(g(n))$ برقرار نباشد. مثلاً، توابع $n$ و $n^{1+\sin n}$ را نمی‌توان با استفاده از نمادهای حدی مقایسه کرد، چرا که مقدار نما در تابع $n^{1+\sin n}$ بین ۰ و ۲ تغییر می‌کند و تمام مقادیر بین این دو عدد را می‌پذیرد.

## تمرین‌ها

۳-۱-۱ فرض کنید $f(n)$ و $g(n)$ دو تابع به صورت حدی نامنفی باشند. با استفاده از تعریف اولیه‌ی نماد $\theta$ ثابت کنید که $\max(f(n), g(n))=\theta(f(n)+g(n))$.

۳-۱-۲ نشان دهید که برای هر دو عدد ثابت $a$ و $b$، که $b>0$، داریم

$$(n+a)^b=\theta(n^b) \qquad (۲-۳)$$

۳-۱-۳ توضیح دهید که چرا عبارت «زمان اجرای الگوریتم $A$ حداقل $O(n^2)$ است» معنی ندارد.

۳-۱-۴ آیا عبارت $2^{n+1}=O(2^n)$ درست است؟ عبارت $2^{2n}=O(2^n)$ چطور؟

۳-۱-۵ قضیه‌ی ۳-۱ را اثبات کنید.

۳-۱-۶ ثابت کنید که زمان اجرای یک الگوریتم از مرتبه‌ی $\theta(g(n))$ است اگر و فقط اگر بدترین حالت زمان اجرای آن $O(g(n))$ و بهترین حالت زمان اجرای آن $\Omega(g(n))$ باشد.

۳-۱-۷ ثابت کنید که $o(g(n))\cap\omega(g(n))$ مجموعه‌ی تهی است.

۳-۱-۸ می‌توانیم تعریف خود از نمادهای حدی را برای دو پارامتر $n$ و $m$ گسترش دهیم، که به صورت مستقل و با سرعت‌های مختلف به سمت بی‌نهایت میل می‌کنند. برای یک تابع $g(n,m)$ داده شده، $O(g(n,m))$ را به صورت مجموعه‌ی توابع زیر تعریف می‌کنیم:

$$O(g(n))=\left\{ f(n,m) \;\middle|\; \begin{array}{l}\text{ثابت‌های مثبت } c \text{ ، } n_0 \text{ و } m_0 \text{ موجود باشند به طوری که برای}\\ \text{هر } n\geq n_0 \text{ و } m\geq m_0 \text{ داشته باشیم } 0\leq f(n,m)\leq cg(n,m)\end{array} \right\}.$$

تعاریف مشابهی برای $\Omega(g(n,m))$ و $\theta(g(n,m))$ ارائه دهید.

## ۳-۲ نمادهای استاندارد و توابع متعارف

در این بخش تعدادی از توابع ریاضی استاندارد و نمادها و رابطه‌ی بین آن‌ها را بررسی خواهیم کرد. همچنین روش استفاده از نمادهای حدی را شرح خواهیم داد.

### یکنواختی (monotonicity)

تابع $f(n)$ *صعودی یکنواخت* است اگر $m \le n$ نتیجه دهد $f(m) \le f(n)$. به همین شکل $f(n)$ *نزولی یکنواخت* است اگر $m \le n$ نتیجه دهد $f(m) \ge f(n)$. تابع $f(n)$ *صعودی اکید* است اگر $m < n$ نتیجه دهد $f(m) < f(n)$، و *نزولی اکید* است اگر $m > n$ نتیجه دهد $f(m) > f(n)$.

### کف و سقف

برای هر عدد حقیقی $x$، بزرگ‌ترین عدد صحیح کوچک‌تر یا مساوی $x$ را به صورت $\lfloor x \rfloor$ (بخوانید کف $x$)، و کوچک‌ترین عدد حقیقی بزرگتر یا مساوی $x$ را به صورت $\lceil x \rceil$ (بخوانید سقف $x$) نشان می‌دهیم. برای تمام اعداد حقیقی $x$

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1 \qquad (۳-۳)$$

برای تمام اعداد صحیح $n$،

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

و برای تمام اعداد حقیقی $x \ge 0$ و اعداد صحیح $a,b > 0$،

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil \qquad (۴-۳)$$

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor \qquad (۵-۳)$$

$$\left\lceil \frac{a}{b} \right\rceil \le \frac{a + (b-1)}{b} \qquad (۶-۳)$$

$$\left\lfloor \frac{a}{b} \right\rfloor \ge \frac{a - (b-1)}{b} \qquad (۷-۳)$$

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

### حساب پیمانه‌ای (Modular arithmetic)

برای هر عدد صحیح $a$ و هر عدد صحیح مثبت $n$، مقدار $a \bmod n$ برابر است با *باقیمانده‌ی* کسر $a/n$:

$$a \bmod n = a - n\lfloor a/n \rfloor \qquad (۸-۳)$$

نتیجه می‌شود که

$$0 \le a \bmod n < n \qquad (۹-۳)$$

بعد از تعریف دقیق باقیمانده‌ی تقسیم یک عدد صحیح بر عدد صحیح دیگر، مناسب است که برای

تساوی باقیمانده‌ها نمادی تعریف کنیم. اگر $(a \bmod n) = (b \bmod n)$، می‌نویسیم $a \equiv b \pmod n$ و می‌گوییم $a$ به پیمانه‌ی $n$ (یا مُد $n$) برابر است با $b$. به عبارت دیگر اگر $a$ و $b$ باقیمانده‌ی یکسانی در تقسیم بر $n$ داشته باشند، همچنین می‌توان گفت $a \equiv b \pmod n$ اگر و تنها اگر $b - a$ بر $n$ بخش پذیر باشد. می‌نویسیم $a \not\equiv b \pmod n$ اگر $a$ به پیمانه‌ی $n$ برابر با $b$ نباشد.

## چندجمله‌ای‌ها

برای عدد صحیح نا منفی $d$ یک *چندجمله‌ای از درجه‌ی* $d$، تابعی مانند $p(n)$ به شکل

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

است که در آن ثابت‌های $a_0, a_1, ..., a_d$ *ضرایب* چندجمله‌ای هستند، و $a_d \neq 0$. یک چندجمله‌ای به صورت حدی مثبت است اگر و فقط اگر $a_d > 0$. برای یک چندجمله‌ای مثبت حدی $p(n)$ از درجه‌ی $d$ 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. می‌گوییم تابع $f(n)$ *کران چندجمله‌ای دارد* اگر ثابتی مانند $k$ وجود داشته باشد که $f(n) = O(n^k)$.

## توابع نمایی

برای تمام اعداد حقیقی $a > 0$، $m$، و $n$، اتحادهای زیر را داریم:

$$a^0 = 1$$
$$a^1 = a$$
$$a^{-1} = 1/a$$
$$(a^m)^n = a^{mn}$$
$$(a^m)^n = (a^n)^m$$
$$a^m a^n = a^{m+n}$$

برای هر $n$ و $a \geq 1$، تابع $a^n$ نسبت به $n$ صعودی یکنواخت است. در صورت لزوم فرض می‌کنیم $0^0 = 1$.

نسبت سرعت رشد توابع نمایی و چندجمله‌ای‌ها را می‌توان به کمک تساوی زیر تعیین کرد. برای تمام ثابت‌های $a$ و $b$ که $a > 1$،

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 \qquad (۳-۱۰)$$

که به کمک آن می‌توانیم نتیجه بگیریم

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

بنابراین، هر تابع نمایی با پایه‌ای بزرگ‌تر از ۱ سریع‌تر از هر چندجمله‌ای رشد می‌کند.

اگر عدد ...۲٫۷۱۸۲۸، پایه‌ی لگاریتم حقیقی را با $e$ نشان دهیم، برای تمام اعداد حقیقی $x$ داریم:

$$e^x = ۱ + x + \frac{x^۲}{۲!} + \frac{x^۳}{۳!} + \ldots = \sum_{i=۰}^{\infty} \frac{x^i}{i!} \qquad (۱۱-۳)$$

که در آن "!" نشان‌دهنده‌ی تابع فاکتوریل است، که در ادامه‌ی همین بخش تعریف خواهد شد. برای تمام اعداد حقیقی $x$ نامساوی زیر برقرار است:

$$e^x \geq ۱ + x \qquad (۱۲-۳)$$

که حالت تساوی فقط برای $x = ۰$ برقرار خواهد بود. وقتی $|x| \leq ۱$ تقریب زیر را خواهیم داشت:

$$۱ + x \leq e^x \leq ۱ + x + x^۲ \qquad (۱۳-۳)$$

وقتی $x \to ۰$، مقدار $۱ + x$ تقریب خوبی برای $e^x$ خواهد بود:

$$e^x = ۱ + x + \theta(x^۲)$$

(در این تساوی از نماد حدی برای توصیف رفتار حدی $x$ وقتی $x \to ۰$، به جای $x \to \infty$ استفاده شده است.) برای تمام $x$های حقیقی داریم:

$$\lim_{n \to \infty} \left(۱ + \frac{x}{n}\right)^n = e^x \qquad (۱۴-۳)$$

## لگاریتم‌ها

در این کتاب، از نمادهای زیر استفاده خواهیم کرد:

$$\lg n = \log_۲ n \quad \text{(لگاریتم دودویی)}$$
$$\ln n = \log_e n \quad \text{(لگاریتم طبیعی)}$$
$$\lg^k n = (\lg k)^n \quad \text{(به توان رساندن)}$$
$$\lg \lg n = \lg(\lg n) \quad \text{(ترکیب)}$$

قرارداد مهمی که در این جا برای راحتی از آن استفاده خواهیم کرد، این است که *تابع لگاریتم فقط عبارت دقیقاً بعد از خود را شامل می‌شود*، مثلاً $\lg n + k$ یعنی $(\lg n) + k$، نه $\lg(n + k)$. اگر $b > ۱$ یک ثابت باشد، آن گاه برای $n > ۰$ تابع $\log_b n$ صعودی اکید است.

برای تمام اعداد حقیقی $a > ۰$، $b > ۰$، $c > ۰$، و $n$،

$$a = b^{\log_b a}$$
$$\log_c (ab) = \log_c a + \log_c b$$
$$\log_b a^n = n \log_b a \qquad (۱۵-۳)$$
$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b (1/a) = -\log_b a$$
$$\log_b a = \frac{1}{\log_a b} \qquad (۳-۱۶)$$
$$a^{\log_b c} = c^{\log_b a}$$

که در تمام تساوی‌های بالا پایه‌ی لگاریتم‌ها ۱ نیستند.

طبق تساوی (۳-۱۵)، تغییر پایه‌ی لگاریتم از یک ثابت به ثابت دیگر مقدار لگاریتم را فقط به اندازه‌ی یک ضریب ثابت تغییر خواهد داد. از این رو معمولاً وقتی ضرایب ثابت برای ما مهم نیستند، می‌نویسیم "$\lg n$"، درست مانند نماد $O$. در علوم کامپیوتر عدد ۲ معمول‌ترین پایه برای لگاریتم‌ها است، چرا که بسیاری از الگوریتم‌ها و ساختمان‌های داده شامل تقسیم مسئله به دو قسمت می‌شوند.

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

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

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

$$\frac{x}{1+x} \le \ln(1+x) \le x \qquad (۳-۱۷)$$

که در آن، حالت تساوی فقط برای $x = 0$ برقرار است.

می‌گوییم تابع $f(n)$ یک *کران چندجمله‌ای لگاریتمی دارد* اگر ثابت $k$ وجود داشته باشد به طوری که $f(n) = O(\lg^k n)$. می‌توان با جایگزینی $n$ با $\lg n$ و $a$ با $2^a$ در تساوی (۳-۹) رابطه‌ی بین رشد چندجمله‌ای‌ها و توابع چندجمله‌ای لگاریتمی را به دست آورد، که می‌دهد:

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

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

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

بنابراین هر تابع چندجمله‌ای مثبت، سریع‌تر از هر تابع چندجمله‌ای لگاریتمی رشد می‌کند.

## فاکتوریل

نماد $n!$ (بخوانید $n$ فاکتوریل) برای اعداد صحیح $n > 0$ به صورت زیر تعریف می‌شود:

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

بنابراین داریم $n! = 1 \times 2 \times 3 \times \dots \times n$.

یک کران بالای ضعیف برای تابع فاکتوریل $n! \le n^n$ است، چرا که هر کدام از $n$ عبارت در ضرب فاکتوریل حداکثر $n$ است. *تقریب استرلینگ* (Stirling's approximation)،

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1+\theta\left(\frac{1}{n}\right)\right) \qquad (۱۸-۳)$$

که در آن $e$ پایه‌ی لگاریتم طبیعی است، یک کران بالا (و همچنین پایین) نزدیک‌تر به ما می‌دهد. می‌توان اثبات کرد (تمرین ۳-۲-۳ را ببینید)

$$\begin{aligned} n! &= o(n^n) \\ n! &= \omega(2^n) \\ \lg(n!) &= \theta(n \lg n) \end{aligned} \qquad (۱۹-۳)$$

که برای اثبات تساوی (۳-۱۹) می‌توان از تقریب استرلینگ استفاده کرد. همچنین تساوی زیر برای هر $n \geq 1$ برقرار است:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{a_n} \qquad (۲۰-۳)$$

که در آن

$$\frac{1}{12n+1} < a_n < \frac{1}{12n} \qquad (۲۱-۳)$$

## تکرار توابع

از نماد $f^{(i)}(n)$ استفاده می‌کنیم تا نشان دهیم که تابع $f(n)$ مکرراً و به تعداد $i$ بار بر روی یک مقدار اولیه‌ی $n$ اعمال شده است. به صورت رسمی اگر $f(n)$ یک تابع بر روی اعداد حقیقی باشد، برای اعداد صحیح نامنفی $i$، تعریف بازگشتی زیر را خواهیم داشت:

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

برای مثال اگر $f(n) = 2n$ آن گاه $f^{(i)}(n) = 2^i n$.

## تابع لگاریتم تکراری

از نماد $\lg^* n$ (بخوانید لوگ اِستار $n$) استفاده می‌کنیم تا تابع لگاریتم تکراری را نشان دهیم، که به صورت زیر تعریف می‌شود. اگر $\lg^{(i)} n$ به صورت بالا تعریف شده باشد، که در آن $f(n) = \lg n$، از آن جایی که تابع لگاریتم برای اعداد نامثبت تعریف نشده است $\lg^{(i)} n$ فقط در صورتی تعریف شده است که $\lg^{(i-1)} n > 0$. دقت کنید که بین $\lg^{(i)} n$ (تابع لگاریتم که $i$ بار بر روی عدد اولیه‌ی $n$ اعمال شده است) و $\lg^i n$ (تابع لگاریتم $n$ به توان $i$) تمایز قائل شوید. تعریف تابع لگاریتم تکراری به شکل زیر است:

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

تابع لگاریتم تکراری رشد به شدت کندی دارد:

$$\lg^* ۲ = ۱$$
$$\lg^* ۴ = ۲$$
$$\lg^* ۱۶ = ۳$$
$$\lg^* ۶۵۵۳۶ = ۴$$
$$\lg^* (۲^{۶۵۵۳۶}) = ۵$$

از آن جایی که تعداد اتم‌ها در قسمت قابل رؤیت کیهان با $۱۰^{۸۰}$ تقریب زده می‌شود، که بسیار کم‌تر از $۲^{۶۵۵۳۶}$ است، به ندرت به ورودی با اندازه‌ی $n$ برمی‌خوریم که $\lg^* n > ۵$.

## اعداد فیبوناچی

*اعداد فیبوناچی* (Fibonacci numbers) با رابطه‌های بازگشتی زیر تعریف می‌شوند:

$$F_۰ = ۰$$
$$F_۱ = ۱ \qquad (۲۲-۳)$$
$$F_i = F_{i-۱} + F_{i-۲} \quad \text{برای } i \geq ۲$$

بنابراین، هر عدد فیبوناچی برابر است با مجموع دو عدد قبلی، که دنباله‌ی زیر را به دست می‌دهد:

۰, ۱, ۱, ۲, ۳, ۵, ۸, ۱۳, ۲۱, ۳۴, ۵۵, ...

اعداد فیبوناچی با *نسبت طلایی* ($\phi$) و جفت آن ($\hat{\phi}$) رابطه دارند، که دو ریشه‌ی معادله‌ی زیر هستند:

$$x^۲ = x + ۱ \qquad (۲۳-۳)$$

که این دو عدد به صورت زیر تعریف می‌شوند (تمرین ۳-۲-۶ رابببنید):

$$\phi = \frac{۱+\sqrt{۵}}{۲}$$
$$= ۱.۶۱۸۰۳... \qquad (۲۴-۳)$$
$$\hat{\phi} = \frac{۱-\sqrt{۵}}{۲}$$
$$= -۰.۶۱۸۰۳...$$

به خصوص داریم:

$$F_i = \frac{\phi^i - \hat{\phi}}{\sqrt{۵}}$$

که می‌توان آن را به کمک استقرا اثبات کرد (تمرین ۳-۲-۷). از آن جایی که $|\hat{\phi}| < ۱$، داریم

$$\frac{|\hat{\phi}^i|}{\sqrt{۵}} < \frac{۱}{\sqrt{۵}}$$
$$< \frac{۱}{۲}$$

که نتیجه می‌دهد:

$$F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor \qquad (۳-۲۵)$$

بنابراین عدد فیبوناچی $i$ام، $F_i$، برابر است با $\phi^i/\sqrt{5}$ که به نزدیک‌ترین عدد صحیح گرد شده باشد. نتیجه می‌گیریم که اعداد فیبوناچی به صورت نمایی رشد می‌کنند.

## تمرین‌ها

۳-۲-۱ نشان دهید که اگر $f(n)$ و $g(n)$ توابع صعودی یکنواخت باشند، آن گاه توابع $f(n)+g(n)$ و $f(g(n))$ هم صعودی یکنواخت خواهند بود، به علاوه اگر $f(n)$ و $g(n)$ نامنفی هم باشند، $f(n)\cdot g(n)$ هم صعودی یکنواخت خواهد بود.

۳-۲-۲ تساوی ۳-۱۶ را اثبات کنید.

۳-۲-۳ تساوی ۳-۱۹ را اثبات کنید. همچنین اثبات کنید که $n! = \omega(2^n)$ و $n! = o(n^n)$.

۳-۲-۴ ★ آیا تابع $\lceil \lg n \rceil !$ کران چندجمله‌ای دارد؟ تابع $\lceil \lg \lg n \rceil !$ چطور؟

۳-۲-۵ ★ کدام یک به صورت حدی بزرگ‌تر است: $\lg(\lg^* n)$ یا $(\lg^* n)$؟

۳-۲-۶ نشان دهید که نسبت طلایی $\phi$ و جفت آن $\hat{\phi}$، هر دو تساوی $x^2 = x + 1$ را ارضا می‌کنند.

۳-۲-۷ به کمک استقرا اثبات کنید که اعداد فیبوناچی، تساوی

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

را ارضا می‌کنند، که در آن $\phi$ نسبت طلایی، و $\hat{\phi}$ جفت آن است.

۳-۲-۸ نشان دهید که $k \ln k = \theta(n)$ نتیجه می‌دهد $k = \theta(n/\ln n)$.

## مسائل

۳-۱ ***رفتار حدی چندجمله‌ای‌ها***

فرض کنید

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

که در آن $a_d > 0$، یک چندجمله‌ای درجه $d$ نسبت به $n$ باشد. با استفاده از تعریف نمادهای حدی خصوصیات زیر را اثبات کنید ( $k$ یک ثابت است).

I. اگر $k \geq d$، آن گاه $p(n) = O(n^k)$.

II. اگر $k \leq d$، آن گاه $p(n) = \Omega(n^k)$.

III. اگر $k = d$، آن گاه $p(n) = \theta(n^k)$.

IV. اگر $k > d$، آن گاه $p(n) = o(n^k)$.

V. اگر $k < d$، آن گاه $p(n) = \omega(n^k)$.

## ۳-۲ رشد حدی نسبی

برای هر جفت عبارت $(A, B)$ در جدول زیر، تعیین کنید که آیا $A$ از مرتبه‌ی $O$، $o$، $\Omega$، $\omega$، و یا $\theta$ نسبت به $B$ است. فرض کنید که $k \geq 1$، $\epsilon > 0$، و $c > 1$ ثابت هستند. جواب شما باید به صورت «بله» یا «خیر» در جدول زیر باشد.

| $A$ | $B$ | $O$ | $o$ | $\Omega$ | $\omega$ | $\theta$ |
|---|---|---|---|---|---|---|
| $\lg^k n$ | $n^\epsilon$ | | | | | |
| $n^k$ | $c^n$ | | | | | |
| $\sqrt{n}$ | $n^{\sin n}$ | | | | | |
| $2^n$ | $2^{n/2}$ | | | | | |
| $n^{\lg c}$ | $c^{\lg n}$ | | | | | |
| $\lg(n!)$ | $\lg(n^n)$ | | | | | |

## ۳-۳ درجه‌بندی نرخ رشد حدی توابع

I. توابع زیر را بر حسب سرعت رشد درجه‌بندی کنید؛ یعنی ترتیبی از توابع به صورت $g_1, g_2, \ldots, g_{30}$ بیابید که روابط $g_1 = \Omega(g_2)$، $g_2 = \Omega(g_3)$، ...، $g_{29} = \Omega(g_{30})$ را ارضا کند. لیست خود را در سطوح مختلف طبقه بندی کنید به طوری که $f(n)$ و $g(n)$ به یک سطح تعلق داشته باشند اگر و فقط اگر $f(n) = \theta(g(n))$.

| $\lg(\lg^* n)$ | $2^{\lg^* n}$ | $(\sqrt{2})^{\lg n}$ | $n^2$ | $n!$ | $(\lg n)!$ |
|---|---|---|---|---|---|
| $\left(\frac{3}{2}\right)^n$ | $n^3$ | $\lg^2 n$ | $\lg(n!)$ | $2^{2^n}$ | $n^{1/\lg n}$ |
| $\ln \ln n$ | $\lg^* n$ | $n.2^n$ | $n^{\lg \lg n}$ | $\ln n$ | $1$ |
| $2^{\lg n}$ | $(\lg n)^{\lg n}$ | $e^n$ | $4^{\lg n}$ | $(n+1)!$ | $\sqrt{\lg n}$ |
| $\lg^*(\lg n)$ | $2^{\sqrt{2 \lg n}}$ | $n$ | $2^n$ | $n \lg n$ | $2^{n+1}$ |

II. مثالی از یک تابع نامنفی $f(n)$ ارائه کنید که برای تمام توابع $g_i(n)$ در قسمت (I) تابع $f(n)$ نه از مرتبه‌ی $O(g_i(n))$ باشد و نه از مرتبه‌ی $\Omega(g_i(n))$.

### ۳-۴ خصوصیات نمادهای حدی

فرض کنید $f(n)$ و $g(n)$ توابع مثبت حدی باشند. هر کدام از حدس‌های زیر را ثابت یا رد کنید.

I. $f(n)=O(g(n))$ نتیجه می‌دهد $g(n)=O(f(n))$.

II. $f(n)+g(n)=\theta(\min(f(n),g(n)))$.

III. $f(n)=O(g(n))$ نتیجه می‌دهد $\lg(f(n))=O(\lg(g(n)))$، که در آن $\lg(g(n))\geq 1$ و برای تمام $n$های به اندازه‌ی کافی بزرگ، $f(n)\geq 1$.

IV. $f(n)=O(g(n))$ نتیجه می‌دهد $2^{f(n)}=O(2^{g(n)})$.

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

VI. $f(n)=O(g(n))$ نتیجه می‌دهد $g(n)=\Omega(f(n))$.

VII. $f(n)=\theta(f(n/2))$.

VIII. $f(n)+o(f(n))=\theta(f(n))$.

### ۳-۵ نسخه‌های مختلف $O$ و $\Omega$

در بعضی کتاب‌ها تعریف $\Omega$ اندکی متفاوت از تعریفی است که در این کتاب دیدیم؛ اجازه دهید از نماد $\overset{\infty}{\Omega}$ (بخوانید امگا بی‌نهایت) برای این تعریف جایگزین استفاده کنیم. می‌گوییم $f(n)=\overset{\infty}{\Omega}(g(n))$ اگر یک ثابت مثبت $c$ وجود داشته باشد به طوری که برای تعداد نامحدودی عدد صحیح $n$ داشته باشیم $f(n)\geq cg(n)\geq 0$.

I. نشان دهید برای هر دو تابع $f(n)$ و $g(n)$ که به صورت حدی نامنفی هستند، $f(n)=O(g(n))$، یا $f(n)=\overset{\infty}{\Omega}(g(n))$، و یا هر دوی آن‌ها صحیح است، در حالی که اگر از $\Omega$ به جای $\overset{\infty}{\Omega}$ استفاده کنیم، این تعبیر درست نیست.

II. مزایا و معایب احتمالی استفاده از $\overset{\infty}{\Omega}$ به جای $\Omega$ را برای تعیین زمان اجرای الگوریتم‌ها شرح دهید.

همچنین در بعضی کتاب‌ها نماد $O$ را کمی متفاوت تعریف می‌کنند؛ اجازه دهید از نماد $O'$ به جای این تعریف جایگزین استفاده کنیم. می‌گوییم $f(n)=O'(g(n))$ اگر و فقط اگر $|f(n)|=O(g(n))$.

III. اگر در قضیه‌ی ۳-۱ به جای نماد $O$ از $O'$ استفاده کنیم، ولی نماد $\Omega$ همچنان سر جای خود باقی بماند، در هر جهت از عبارت «اگر و تنها اگر» چه رخ می‌دهد؟

در بعضی کتاب‌ها از نماد $\widetilde{O}$ (بخوانید او تیلدا) برای نشان دادن نماد $O$ که در آن از عوامل لگاریتمی صرف نظر شده است استفاده می‌کنند:

$$\widetilde{O}(g(n)) = \left\{ f(n) \middle| \begin{array}{l} \text{ثابت‌های مثبت } c,\ k \text{ و } n_0 \text{ موجود باشند به طوری که برای} \\ \text{تمام } n \geq n_0 \text{ داشته باشیم } 0 \leq f(n) \leq cg(n)\lg^k(n) \end{array} \right\}.$$

IV. $\widetilde{\Omega}$ و $\widetilde{\theta}$ را به صورت مشابه تعریف، و یک قضیه مشابه قضیه‌ی ۳-۱ برای آن‌ها اثبات کنید.

## ۳-۶ *توابع تکراری*

عملگر * که در تابع $\lg^*$ استفاده شد را می‌توان برای هر تابع صعودی یکنواخت $f(n)$ روی اعداد حقیقی به کار برد. برای یک ثابت داده شده‌ی $c \in R$، تابع تکراری $f_c^*$ را به صورت زیر تعریف می‌کنیم:

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

که نیازی نیست در تمام موارد تعریف شده باشد. به عبارت دیگر کمیت $f_c^*(n)$ برابر است با کم‌ترین تعداد تکرار لازم تابع $f$ برای این که بتوان خروجی تابع را به $c$ یا کم‌تر رساند.

برای هر کدام از توابع $f(n)$ و ثابت‌های $c$ زیر، نزدیک‌ترین کران ممکن را برای $f_c^*(n)$ بدهید.

| $f(n)$ | $c$ | $f_c^*(n)$ |
|---|---|---|
| $n-1$ | ۰ | |
| $\lg n$ | ۱ | |
| $n/2$ | ۱ | |
| $n/2$ | ۲ | |
| $\sqrt{n}$ | ۲ | |
| $\sqrt{n}$ | ۱ | |
| $n^{1/3}$ | ۲ | |
| $n/\lg n$ | ۲ | |


# تقسیم و حل

در بخش ۲-۳-۱ دیدیم که مرتب‌سازی ادغامی نمونه‌ای است از رویکرد تقسیم و حل. به خاطر بیاورید که در تقسیم و حل مسئله را به صورت بازگشتی حل می‌کنیم، و در هر سطح از بازگشت سه مرحله عملیات بر روی آن انجام می‌دهیم:

- تقسیم مسئله به تعدادی زیرمسئله که نمونه‌های کوچک‌تری از همان مسئله هستند.
- حل زیرمسئله‌ها به صورت بازگشتی. اگر اندازه‌ی زیرمسئله‌ها به اندازه‌ی کافی کوچک باشد، حل آن‌ها به صورت مستقیم.
- ترکیب جواب زیرمسئله‌ها برای دستیابی به جواب مسئله‌ی اصلی.

وقتی زیرمسئله‌ها به اندازه‌ای بزرگ هستند که آن‌ها را به صورت بازگشتی حل کنیم، به آن‌ها *حالت بازگشتی* (recursive case) می‌گوییم. وقتی زیرمسئله‌ها به اندازه‌ی کافی کوچک شدند که دیگر نیازی به انجام بازگشت نبود، می‌گوییم به «انتهای بازگشت» رسیده‌ایم، و باید *حالت پایه* (base case) را حل کنیم. بعضی مواقع علاوه بر زیرمسئله‌هایی که نمونه‌های کوچک‌تری از مسئله‌ی اصلی هستند، زیرمسئله‌هایی داریم که تفاوت‌هایی با مسئله‌ی اصلی دارند. حل چنین مسئله‌هایی را بخشی از مرحله‌ی ترکیب خواهیم دانست.

در این فصل الگوریتم‌های بیشتری خواهیم دید که بر پایه‌ی روش تقسیم و حل بنا شده اند. اولین آن‌ها مسئله‌ی زیرآرایه‌ی بیشینه را حل می‌کند: این الگوریتم به عنوان ورودی، آرایه‌ای از اعداد را دریافت و یک زیرآرایه‌ی پیوسته با مجموع بیشینه را به عنوان خروجی ارائه می‌کند. سپس دو الگوریتم تقسیم و حل برای ضرب ماتریس‌های $n \times n$ خواهیم دید. یکی از آن‌ها در زمان $\theta(n^3)$ اجرا می‌شود، که کارایی آن تفاوتی با روش سرراست ضرب ندارد. ولی دیگری، الگوریتم استراسن، در زمان $O(n^{2/81})$ اجرا می‌شود که به صورت حدی از متد سرراست سریع‌تر است.

## بازگشت‌ها

بازگشت‌ها شانه به شانه همراه رویکرد تقسیم و حل حضور دارند، چرا که روشی طبیعی برای توصیف زمان اجرای الگوریتم‌های تقسیم و حل ارائه می‌کنند. *بازگشت* (recurrence)، یک تساوی یا نامساوی است که یک تابع را بر حسب مقدار همان تابع برای ورودی‌های کوچک‌تر توصیف می‌کند. برای مثال در بخش ۲-۳-۲ بدترین حالت زمان اجرای رویه‌ی MERGE-SORT را توسط رابطه‌ی بازگشتی

$$T(n)=\begin{cases}\theta(1) & \text{اگر } n=1\\ 2T(n/2)+\theta(n) & \text{اگر } n>1\end{cases} \qquad (۱-۴)$$

توصیف کردیم، که جواب آن برابر است با $T(n)=\theta(n\lg n)$.

بازگشت‌ها می‌توانند شکل‌های مختلفی به خود بگیرند. برای مثال یک الگوریتم بازگشتی ممکن است مسئله را به زیرمسئله‌هایی با اندازه‌های نابرابر تقسیم کند، مثلاً تقسیم ۲/۳ به ۱/۳. اگر مراحل تقسیم و ترکیب در زمان خطی اجرا شوند، چنین الگوریتمی رابطه‌ی بازگشتی

$$T(n)=T(2n/3)+T(n/3)+\theta(n)$$

را به ما می‌دهد.

لزوماً نیازی نیست که اندازه‌ی زیرمسئله‌ها کسر ثابتی از مسئله‌ی اصلی باشند. مثلاً یک نسخه‌ی بازگشتی از جستجوی خطی (تمرین ۲-۱-۳)، فقط یک زیرمسئله حاوی فقط یک عنصر کم‌تر از مسئله‌ی اصلی می‌سازد. هر فراخوانی بازگشتی به یک زمان ثابت به علاوه‌ی زمان فراخوانی‌های بازگشتی نیاز خواهد داشت، که به رابطه‌ی بازگشتی زیر منجر می‌شود:

$$T(n)=T(n-1)+\theta(1)$$

در این فصل سه متد برای حل رابطه‌های بازگشتی ارائه می‌شود – یعنی، برای تعیین کران‌های حدی "$O$" یا "$\theta$" بر روی جواب:

- در *متد جانشین‌سازی*، ابتدا یک کران برای رابطه‌ی بازگشتی حدس می‌زنیم و سپس از استقرای ریاضی برای اثبات کران خود استفاده می‌کنیم.
- *متد درخت بازگشتی*، رابطه‌ی بازگشتی را به یک درخت تبدیل می‌کند که گره‌های آن نشان‌دهنده‌ی سطوح مختلف بازگشت است؛ برای حل این بازگشت از تکنیک‌های تعیین کران سری‌ها استفاده می‌کنیم.
- *متد اصلی* برای روابط بازگشتی به شکل

$$T(n)=aT(n/b)+f(n) \qquad (۲-۴)$$

کران‌هایی تعیین می‌کند، که در آن $a\geq 1$، $b>1$، و $f(n)$ یک تابع داده شده است. با چنین بازگشت‌هایی زیاد برخورد خواهیم کرد. یک رابطه‌ی بازگشتی با فرمت داده شده در تساوی (۴-۲)، الگوریتم تقسیم و حلی را توصیف می‌کند که $a$ زیرمسئله می‌سازد، اندازه‌ی هر یک

از این زیرمسئله‌ها $1/b$ اندازه‌ی مسئله‌ی اصلی است، و در آن مراحل تقیسم و ترکیب روی هم به زمان $f(n)$ نیاز دارند.

برای استفاده از این روش نیاز دارید که سه حالت مختلف را به خاطر بسپارید، ولی وقتی این کار را انجام دادید، تعیین کران‌های حدی برای بسیاری از روابط بازگشتی ساده، آسان خواهد شد. برای تعیین زمان اجرای الگوریتم‌های تقسیم و حل مسئله‌های زیرآرایه‌ی بیشینه و ضرب ماتریس‌ها، از متد اصلی استفاده خواهیم کرد، و همچنین برای تعیین زمان اجرای بسیاری دیگر از الگوریتم‌های تقسیم و حل ارائه شده در این کتاب.

هر از گاهی به بازگشت‌هایی برخورد خواهیم کرد که به جای تساوی، به صورت نامساوی هستند، مانند $T(n) \le 2T(n/2) + \theta(n)$. چون چنین رابطه‌ای فقط یک کران بالا بر روی $T(n)$ تعیین می‌کند، جواب آن را با استفاده از نماد $O$ توصیف خواهیم کرد، و نه نماد $\theta$. به طور مشابه اگر نامساوی به شکل $T(n) \ge 2T(n/2) + \theta(n)$ بیان شود، آن گاه چون رابطه‌ی بازگشتی یک کران بالا بر روی $T(n)$ به ما می‌دهد، در جواب آن از نماد $\Omega$ استفاده خواهیم کرد.

## نکات فنی

در عمل هنگام حل و بحث روابط بازگشتی، از جزئیات فنی خاصی صرف نظر می‌کنیم. برای مثال اگر MERGE-SORT را بر روی $n$ عنصر فراخوانی کنیم، که در آن $n$ فرد است، به زیرمسئله‌هایی با اندازه‌های $\lfloor n/2 \rfloor$ و $\lceil n/2 \rceil$ می‌رسیم. اندازه‌ی هیچ کدام از زیرمسئله‌ها $n/2$ نیست، چرا که وقتی $n$ فرد است، $n/2$ عدد صحیح نخواهد بود. در واقع رابطه‌ی بازگشتی که بدترین حالت زمان اجرای MERGE-SORT را توصیف می‌کند به صورت زیر است:

$$T(n) = \begin{cases} \theta(1) & \text{اگر } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \theta(n) & \text{اگر } n > 1 \end{cases} \qquad (3-4)$$

مناطق مرزی، یکی دیگر از جزئیاتی است که معمولاً از آن صرف نظر می‌کنیم. از آن جایی که زمان اجرای یک الگوریتم روی یک ورودی با اندازه‌ی ثابت مقدار ثابتی است، رابطه‌های بازگشتی که زمان اجرای الگوریتم‌ها را مشخص می‌کنند، معمولاً برای $n$های به اندازه‌ی کافی کوچک مقداری برابر با $T(n) = \theta(1)$ دارند. بالطبع برای سادگی معمولاً از عبارات مربوط به مقادیر مرزی در رابطه‌های بازگشتی صرف‌نظر می‌کنیم و فرض می‌کنیم که $T(n)$ برای $n$های کوچک ثابت است. برای مثال معمولاً رابطه‌ی بازگشتی ۴-۱ را به صورت

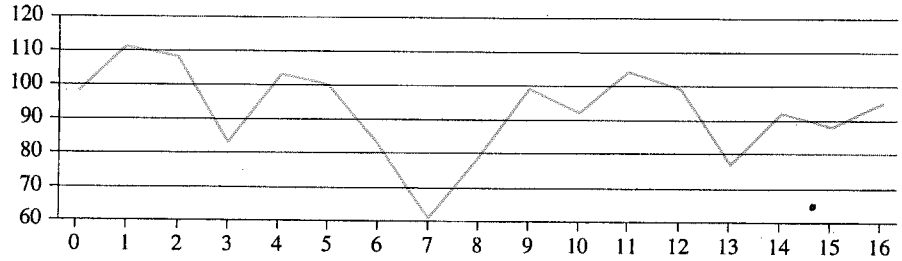
$$T(n) = 2T(n/2) + \theta(n) \qquad (4-4)$$

می‌نویسیم، بدون این که برای مقادیر کوچک $n$ رابطه‌ی جدایی مشخص کنیم. دلیل این کار این است که با این که تغییر مقدار $T(1)$ جواب رابطه‌ی بازگشتی را عوض می‌کند، ولی این تغییر هیچ وقت بیشتر از یک مقدار ثابت نخواهد بود، و در نتیجه مرتبه‌ی رشد بدون تغییر باقی خواهد ماند.

وقتی در مورد رابطه‌های بازگشتی بحث می‌کنیم، معمولاً از کف‌ها، سقف‌ها، و مقادیر مرزی صرف‌نظر می‌کنیم. فعلاً بدون در نظر گرفتن این جزئیات ادامه می‌دهیم، و بعداً تعیین می‌کنیم که آیا آن‌ها اهمیت دارند یا خیر. البته معمولاً اهمیتی ندارند، ولی مهم است بدانیم کی دارند. تجربه برای دانستن این مسئله کمک خواهد کرد، همین طور قضایایی که اولین آن‌ها قضیه‌ی ۴-۱ است. این قضیه می‌گوید این جزئیات تأثیری در کران‌های حدی بسیاری از روابط بازگشتی توصیف کننده‌ی الگوریتم‌های تقسیم و حل ندارند. با این حال در این فصل در مورد تعدادی از این جزئیات صحبت خواهیم کرد تا نکات مثبت روش‌های حل روابط بازگشتی را نشان دهیم.

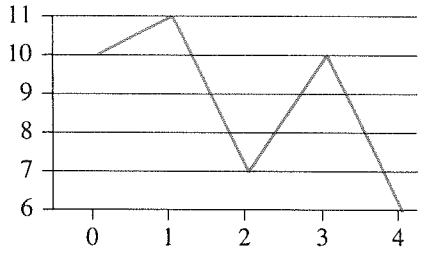
## ۴-۱ مسئله‌ی زیرآرایه‌ی بیشینه

فرض کنید فرصت سرمایه‌گذاری در یک کمپانی مواد شیمیایی فرار به شما داده شده است. مانند مواد شیمیایی که کمپانی تولید می‌کند، قیمت سهام کمپانی مواد شیمیایی فرار هم نسبتا فرار است. شما اجازه دارید یک بار یک سهم را خریداری کرده و در یک زمان دلخواه آن را بفروشید، که این خرید و فروش‌ها بعد از پایان روز کاری و تثبیت قیمت‌های آن روز انجام می‌شود. برای جبران این محدودیت، قیمت سهام آینده به شما اطلاع داده خواهد شد. هدف، بیشینه کردن سود است. شکل ۴-۱، قیمت سهام را در یک دوره‌ی ۱۷ روزه نشان می‌دهد. شما می‌توانید سهام را در هر زمانی پس از روز ۰، که قیمت در آن روز ۱۰۰ دلار است، خریداری کنید. مسلماً بهترین روش برای بیشینه کردن سود، «ارزان خری و گران فروشی»– خرید در پایین‌ترین قیمت ممکن و فروش در بالاترین قیمت ممکن– است. متأسفانه ممکن است نتوانید در یک دوره‌ی داده شده، در پایین‌ترین قیمت ممکن خریده و در بالاترین قیمت ممکن بفروشید. در شکل ۴-۱ پایین‌ترین قیمت پس از روز ۷ رخ می‌دهد، که بعد از بالاترین قیمت (روز ۱) است.




| روز | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| قیمت | 100 | 113 | 110 | 85 | 105 | 102 | 86 | 63 | 81 | 101 | 94 | 106 | 101 | 79 | 94 | 90 | 97 |
| تغییر | | 13 | −3 | −25 | 20 | −3 | −16 | −23 | 18 | 20 | −7 | 12 | −5 | −22 | 15 | −4 | 7 |

**شکل ۴-۱** اطلاعات مربوط به قیمت سهام در کمپانی مواد شیمیایی فرار پس از پایان خرید و فروش در یک دوره‌ی ۱۷ روزه. محور افقی نمودار روز معامله را نشان می‌دهد، و محور عمودی قیمت را. ردیف پایین جدول تغییر در قیمت نسبت به روز قبل را نشان می‌دهد.




| روز | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| قیمت | 10 | 11 | 7 | 10 | 6 |
| تغییر | | 1 | −4 | 3 | −4 |

**شکل ٤-٢** یک مثال که نشان می‌دهد بیش‌ترین سود همیشه از پایین‌ترین قیمت آغاز و یا به بالاترین قیمت ختم نمی‌شود. دوباره، محور افقی نشان‌دهنده‌ی روز و محور عمودی نشان‌دهنده‌ی قیمت است. در این جا بیش‌ترین سود، که برابر با ۳ دلار بر سهم است، با خرید بعد از روز ۲ و فروش بعد از روز ۳ به دست می‌آید. قیمت ۷ دلار در روز دوم، پایین‌ترین قیمت کلی، و یا قیمت ۱۰ دلار در روز سوم، بالاترین قیمت کلی نیستند.

ممکن است فکر کنید همیشه می‌توانید با خریداری در پایین‌ترین قیمت یا فروش در بالاترین قیمت، سود خود را بیشینه کنید. برای مثال در شکل ۴-۱، می‌توانیم با خرید در پایین‌ترین قیمت، بعد از روز ۷، سود خود را بیشینه کنیم. اگر این استراتژی همیشه کار می‌کرد آن گاه تعیین روش دستیابی به بالاترین سود همیشه آسان بود: یافتن بالاترین و پایین‌ترین قیمت‌ها، بررسی قیمت‌های قبل از بالاترین قیمت و یافتن پایین‌ترین آن‌ها، بررسی قیمت‌های بعد از پایین‌ترین قیمت و یافتن بالاترین آن‌ها، و انتخاب جفت با اختلاف بیشتر. شکل ۴-۲ یک مثال نقض ساده را نشان می‌دهد، که در آن بالاترین سود نه با خرید به کم‌ترین قیمت و نه با فروش به بیش‌ترین قیمت به دست می‌آید.

## یک راه‌حل بی‌خردانه

به راحتی می‌توان یک راه حل بی‌خردانه برای این مسئله پیدا کرد: هر جفت تاریخ‌ها برای خرید و فروش را امتحان کنید، که در آن تاریخ خرید قبل از تاریخ فروش است. یک دوره‌ی $n$ روزه، $\binom{n}{2}$ تاریخ با این شرایط دارد. از آن‌جایی که $\binom{n}{2}$ از مرتبه‌ی $\theta(n^2)$ است، و در بهترین حالت می‌توانیم امیدوار باشیم که بررسی هر جفت از تاریخ‌ها در زمان ثابت انجام شود، این رویکرد در زمان $\Omega(n^2)$ اجرا می‌شود. آیا راه حل بهتر از این هم وجود دارد؟

## یک تبدیل

برای طراحی یک الگوریتم با زمان اجرای $o(n^2)$، با نگاهی کمی متفاوت ورودی را بررسی می‌کنیم. می‌خواهیم دنباله‌ای بیابیم که تغییر خالص قیمت روی آن از روز اول تا روز آخر بیشینه باشد. به جای در نظر گرفتن قیمت‌های روزانه، اجازه دهید تغییرات روزانه در قیمت را در نظر بگیریم، که در آن تغییر قیمت در روز $i$ برابر است با اختلاف قیمت‌های روز $i-1$ و روز $i$. جدول شکل ۴-۱ این تغییرات روزانه را در ردیف پایین نشان می‌دهد. این ردیف را به صورت یک آرایه‌ی $A$ در نظر

می‌گیریم، که در شکل ۴-۳ نشان داده شده است. اکنون هدف این خواهد بود که زیرآرایه‌ی ناتهی و پیوسته‌ای از $A$ بیابیم که مجموع مقادیر آن بیشینه باشد. این زیرآرایه‌ی پیوسته را *زیرآرایه‌ی بیشینه* (maximum subarray) می‌نامیم. برای مثال در آرایه‌ی شکل ۴-۳، زیرآرایه‌ی بیشینه آرایه‌ی $A[1..16]$ عبارت است از $A[8..11]$، با مجموع ۴۳. بنابراین بهترین کار این است که سهام را قبل از روز ۸ (یعنی بعد از روز ۷) خریده و بعد از روز ۱۱ بفروشید، و به سود ۴۳ دلار بر سهم دست یابید.

در نگاه اول این تبدیل هیچ کمکی نمی‌کند. هنوز برای یک دوره‌ی $n$ روزه باید $\binom{n-1}{2} = \theta(n^2)$ زیرآرایه را چک کنیم. تمرین ۴-۱-۲ از شما می‌خواهد نشان دهید که با این که محاسبه‌ی هزینه‌ی یک زیرآرایه ممکن است به زمانی متناسب با طول زیرآرایه نیاز داشته باشد، هنگام محاسبه‌ی مجموع تمام $\theta(n^2)$ زیرآرایه، می‌توانیم محاسبه را طوری سازماندهی کنیم که محاسبه‌ی مجموع هر زیرآرایه، با داشتن مجموع زیرآرایه‌های قبل، در زمان $O(1)$ انجام شود، و در نتیجه این راه حل بی‌خردانه به $\theta(n^2)$ زمان نیاز داشته باشد.

اجازه دهید به دنبال یک راه حل کاراتر برای مسئله‌ی زیرآرایه‌ی بیشینه بگردیم. در ادامه معمولاً از «یک» زیرآرایه‌ی بیشینه صحبت خواهیم کرد، و نه از «تنها» زیرآرایه‌ی بیشینه، چرا که ممکن است بیش از یک زیرآرایه وجود داشته باشد که به مجموع بیشینه دست یابد.

مسئله‌ی زیرآرایه‌ی بیشینه فقط زمانی جالب توجه است که آرایه حاوی اعداد منفی هم باشد. اگر تمام ورودی‌های آرایه نامنفی باشند، آن گاه مسئله‌ی زیرآرایه‌ی بیشینه هیچ چالشی پیش روی ما قرار نمی‌دهد، چرا که در این صورت کل آرایه بیش‌ترین مجموع را خواهد داشت.

## راه حلی با استفاده از تقسیم و حل

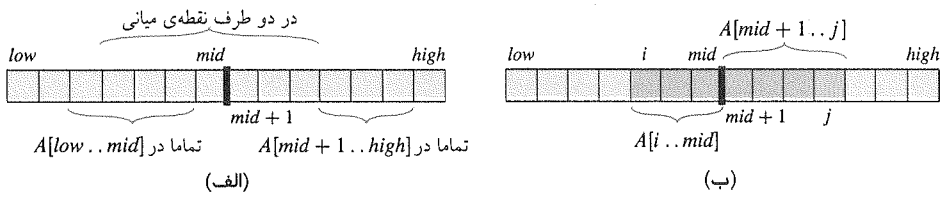
کمی به این فکر کنید که چطور می‌توان با استفاده از تکنیک تقسیم و حل مسئله‌ی زیرآرایه‌ی بیشینه را حل کرد. فرض کنید می‌خواهیم زیرآرایه‌ی بیشینه را در زیرآرایه‌ی $A[low\ ..\ high]$ بیابیم. تقسیم و حل پیشنهاد می‌کند که در صورت امکان، زیرآرایه را به دو زیرآرایه با اندازه‌ی مساوی تقسیم کنیم. یعنی نقطه‌ی میانی برای زیرآرایه، مثلاً $mid$، می‌یابیم، و سپس زیرآرایه‌های $A[low\ ..\ mid]$ و $A[mid+1\ ..\ high]$ را در نظر می‌گیریم. همان طور که شکل ۴-۴ (الف) نشان می‌دهد، هر زیرآرایه‌ی پیوسته‌ی $A[i\ ..\ j]$ از $A[low\ ..\ high]$ باید دقیقاً در یکی از مکان‌های زیر قرار داشته باشد:

- تماماً در زیرآرایه‌ی $A[low\ ..\ mid]$، به طوری که $low \le i \le j \le mid$،

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 13 | -3 | -25 | 20 | -3 | -16 | -23 | 18 | 20 | -7 | 12 | -5 | -22 | 15 | -4 | 7 |

زیرآرایه‌ی ماکزیمم

**شکل ٤-٣** تغییر قیمت سهام به صورت مسئله‌ی زیرآرایه‌ی بیشینه. در این جا زیرآرایه‌ی $A[8..11]$ با مجموع ۴۳، بیش‌ترین مجموع را در میان تمام زیرآرایه‌های پیوسته‌ی $A$ دارد.




**شکل ٤-٤** (الف) موقعیت‌های ممکن برای زیرآرایه‌های $A[low\ ..\ high]$: تماماً در $A[low\ ..\ mid]$، تماماً در $A[mid+1\ ..\ high]$، یا دوطرف نقطه‌ی میانی. (ب) هر زیرآرایه از $A[low\ ..\ high]$ که از نقطه‌ی میانی عبور می‌کند از دو زیرآرایه‌ی $A[i\ ..\ mid]$ و $A[mid+1\ ..\ j]$ تشکیل شده است، که در آن $low \le i \le mid$ و $mid < j \le high$.

- تماماً در زیرآرایه‌ی $A[mid+1\ ..\ high]$، به طوری که $mid < i \le j \le high$، یا
- دو طرف نقطه‌ی میانی، به طوری که $low \le i \le mid < j \le high$.

بنابراین یک زیرآرایه‌ی بیشینه از آرایه‌ی $A[low\ ..\ high]$ هم باید دقیقاً در یکی از این مکان‌ها قرار داشته باشد. در واقع یک زیرآرایه‌ی بیشینه از $A[low\ ..\ high]$ باید بیش‌ترین مجموع را در میان تمام زیرآرایه‌هایی داشته باشد که تماماً در $A[low\ ..\ mid]$ هستند، تماماً در $A[mid+1\ ..\ high]$ هستند، و یا از نقطه‌ی میانی عبور می‌کنند. می‌توانیم زیرآرایه‌های بیشینه $A[low\ ..\ mid]$ و $A[mid+1\ ..\ high]$ را به صورت بازگشتی بیابیم، چرا که این دو زیرمسئله، نمونه‌های کوچک‌تری از مسئله‌ی یافتن زیرآرایه‌ی بیشینه هستند. بنابراین تنها کاری که باقی می‌ماند این است که یک زیرآرایه‌ی بیشینه بیابیم که از نقطه‌ی میانی عبور می‌کند، و در نهایت زیرآرایه‌ی با بیش‌ترین مجموع در میان این سه زیرآرایه را انتخاب کنیم.

به سادگی می‌توانیم زیرآرایه‌ی بیشینه عبور کننده از نقطه‌ی میانی را در زمان خطی نسبت به اندازه‌ی زیرآرایه‌ی $A[low\ ..\ high]$ بیابیم. این مسئله، یک نمونه‌ی کوچک‌تر از مسئله‌ی اصلی نیست، چرا که محدودیت اضافی عبور کردن زیرآرایه‌ی جواب از نقطه‌ی میانی را دارد. همان طور که شکل ۴-۴ (ب) نشان می‌دهد، هر زیرآرایه که از نقطه‌ی میانی عبور می‌کند، خود از دو زیرآرایه‌ی $A[i\ ..\ mid]$ و $A[mid+1\ ..\ j]$ تشکیل شده است، که در آن $low \le i \le mid$ و $mid < j \le high$. بنابراین فقط نیاز داریم زیرآرایه‌های بیشینه به شکل $A[i\ ..\ mid]$ و $A[mid+1\ ..\ j]$ را یافته و آن‌ها را ترکیب کنیم. رویه‌ی FIND-MAX-CROSSING-SUBARRAY به عنوان ورودی، آرایه‌ی $A$ و اندیس‌های $low$، $mid$، و $high$ را دریافت می‌کند، و یک سه‌تایی نشان‌دهنده‌ی حدود زیرآرایه‌ی بیشینه‌ای که از نقطه‌ی میانی عبور می‌کند، به همراه مجموع مقادیر این زیرآرایه را بازمی‌گرداند.

```
FIND-MAX-CROSSING-SUBBARRAY(A, low, mid, high)
1  left-sum = -∞
2  sum = 0
3  for i = mid downto low
4      sum = sum + A[i]
```

```
5       if sum > left-sum
6           left-sum = sum
7           max-left = i
8   right-sum = -∞
9   sum = 0
10  for j = mid + 1 to high
11      sum = sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

رویه به صورت زیر کار می‌کند. خطوط ۱-۷ یک زیرآرایه از نیمه‌ی چپ ( $A[low \;..\; mid]$ ) می‌یابد. از آن جایی که این زیرآرایه باید شامل $A[mid]$ باشد، حلقه‌ی for خطوط ۳-۷ با اندیس $i$، از مقدار $mid$ آغاز کرده و به سمت پایین حرکت می‌کند تا به $low$ برسد، و به این صورت تمام زیرآرایه‌های در نظر گرفته شده به شکل $A[i \;..\; mid]$ هستند. خطوط ۱-۲ متغیرهای $left\text{-}sum$ و $sum$ را مقداردهی اولیه می‌کنند، که این متغیرها به ترتیب بیش‌ترین مجموع یافته شده تا کنون، و مجموع ورودی‌های $A[i \;..\; mid]$ را در خود نگه می‌دارند. هر گاه در خط ۵ یک زیرآرایه‌ی $A[i \;..\; mid]$ با مجموع مقادیر بیش‌تر از $left\text{-}sum$ پیدا شد، در خط ۶ متغیر $left\text{-}sum$ را با مجموع عناصر این زیرآرایه، و در خط ۷ متغیر $max\text{-}left$ را با اندیس $i$ به هنگام سازی می‌کنیم. خطوط ۸-۱۴ به طور مشابه همین کار را برای نیمه‌ی سمت راست، یعنی $A[mid+1 \;..\; high]$ انجام می‌دهند. در این جا حلقه‌ی for خطوط ۱۰-۱۴ با آغاز از $mid+1$، اندیس $j$ را افزایش می‌دهند تا به $high$ برسد، به طوری که تمام زیرآرایه‌های در نظر گرفته شده به صورت $A[mid+1 \;..\; j]$ هستند. نهایتاً خط ۱۵ اندیس‌های $max\text{-}left$ و $max\text{-}right$ را، که نشان‌دهنده‌ی یک زیرآرایه‌ی بیشینه هستند که از نقطه‌ی میانی عبور می‌کند، به همراه مجموع مقادیر زیرآرایه‌ی $A[max\text{-}left \;..\; max\text{-}right]$، یعنی $left\text{-}sum + right\text{-}sum$، بازمی‌گرداند.

اگر زیرآرایه‌ی $A[low \;..\; high]$ حاوی $n$ عنصر باشد، ادعا می‌کنیم که فراخوانی FIND-MAX-CROSSING-SUBARRAY $(A, low, mid, high)$ در زمان $\theta(n)$ اجرا می‌شود. چون هر تکرار هر یک از دو حلقه‌ی for به زمان $\theta(1)$ نیاز دارد، فقط باید تعداد کل تکرارها را بشماریم. تعداد تکرارهای حلقه‌ی for خطوط ۳-۷ برابر است با $mid\text{-}low+1$، و تعداد تکرارهای حلقه‌ی for خطوط ۱۰-۱۴ برابر با $high\text{-}mid$، و بنابراین تعداد کل تکرارها عبارت است از

$$(mid\text{-}low+1)+(high\text{-}mid)=high\text{-}low+1=n$$

با داشتن یک رویه‌ی FIND-MAX-CROSSING-SUBARRAY با زمان خطی، می‌توانیم شبه‌کدی برای یک الگوریتم تقسیم و حل برای حل مسئله‌ی زیرآرایه‌ی بیشینه بنویسیم:

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
1  if high == low
2      return (low, high, A[low])          // base case: only one element
3  else mid = ⌊(low + high)/2⌋
4      (left-low, left-high, left-sum) =
           FIND-MAXIMUM-SUBARRAY(A, low, mid)
5      (right-low, right-high, right-sum) =
           FIND-MAXIMUM-SUBARRAY(A, mid+1, high)
6      (cross-low, cross-high, cross-sum) =
           FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
8          return (left-low, left-high, left-sum)
9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10         return (right-low, right-high, right-sum)
11     else return (cross-low, cross-high, cross-sum)
```

فراخوانی اولیه‌ی FIND-MAXIMUM-SUBARRAY $(A, 1, A.length)$ یک زیرآرایه‌ی بیشینه از $A[1 .. n]$ می‌یابد.

مانند FIND-MAXIMUM-SUBARRAY، رویه‌ی بازگشتی FIND-MAX-CROSSING-SUBARRAY یک سه‌تایی حاوی اندیس‌های زیرآرایه‌ی بیشینه، به همراه مجموع مقادیر زیرآرایه‌ی بیشینه بازمی‌گرداند. خط ۱ حالت پایه را بررسی می‌کند، که در آن زیرآرایه فقط یک عنصر دارد. زیرآرایه‌ای که فقط یک عنصر داشته باشد فقط یک زیرآرایه دارد — خودش — و بنابراین خط ۲ یک سه‌تایی بازمی‌گرداند با اندیس‌های شروع و پایان همان یک عنصر، به همراه مقدار آن. خطوط ۳-۱۱ مسئول حالت بازگشتی هستند. خط ۳ قسمت تقسیم را انجام می‌دهد، که همان محاسبه‌ی اندیس $mid$ مربوط به نقطه‌ی میانی است. اجازه دهید زیرآرایه‌ی $A[low .. mid]$ را *زیرآرایه‌ی چپ* و زیرآرایه‌ی $A[mid+1 .. high]$ را *زیرآرایه‌ی راست* بنامیم. از آن جایی که می‌دانیم زیرآرایه‌ی $A[low .. high]$ حداقل حاوی دو عنصر است، هر یک از زیرآرایه‌های چپ و راست باید حداقل حاوی یک عنصر باشند. خطوط ۴ و ۵ بخش حل را انجام می‌دهند، بدین صورت که زیرآرایه‌های بیشینه را به ترتیب در زیرآرایه‌های چپ و راست پیدا می‌کنند. خطوط ۶-۱۱ بخش ترکیب را انجام می‌دهند. خط ۶ یک زیرآرایه‌ی بیشینه پیدا می‌کند که از نقطه‌ی میانی عبور می‌کند. (به خاطر بیاورید که چون خط ۶ زیرمسئله‌ای را حل می‌کند که نمونه‌ی کوچک‌تری از مسئله‌ی اصلی نیست، آن را جزء قسمت ترکیب در نظر می‌گیریم.) خط ۷ تست می‌کند که آیا زیرآرایه‌ی چپ حاوی یک زیرآرایه با مجموع بیشینه هست یا نه، و خط ۸ آن زیرآرایه‌ی بیشینه را بازمی‌گرداند. در غیر این صورت، خط ۹ تست می‌کند که آیا زیرآرایه‌ی راست حاوی یک زیرآرایه با مجموع بیشینه هست یا نه، و خط ۱۰ آن زیرآرایه را

بازمی‌گرداند. اگر نه زیرآرایه‌ی چپ و نه زیرآرایه‌ی راست حاوی یک زیرآرایه با مجموع بیشینه بودند، پس زیرآرایه‌ی بیشینه باید از نقطه‌ی میانی عبور کند، و خط ۱۱ آن زیرآرایه را بازمی‌گرداند.

## تحلیل الگوریتم تقسیم و حل

در ادامه، یک رابطه‌ی بازگشتی ارائه می‌کنیم که زمان اجرای رویه‌ی بازگشتی FIND-MAXIMUM-SUBARRAY را توصیف می‌کند. مشابه تحلیلی که برای مرتب‌سازی ادغامی در بخش ۲-۳-۲ انجام دادیم، برای سادگی فرض می‌کنیم که اندازه‌ی مسئله‌ی اصلی توانی از ۲، و در نتیجه اندازه‌ی تمام زیرمسئله‌ها یک عدد صحیح است. زمان اجرای FIND-MAXIMUM-SUBARRAY بر روی یک زیرآرایه‌ی $n$ عنصری را با $T(n)$ نشان می‌دهیم. برای شروع، خط ۱ به زمان ثابت نیاز دارد. حالت پایه وقتی $n = ۱$، ساده است: خط ۲ در زمان ثابت اجرا می‌شود، و بنابراین

$$T(۱) = \theta(۱) \tag{۴-۵}$$

حالت بازگشتی زمانی رخ می‌دهد که $n > ۱$. خطوط ۱ و ۳ در زمان ثابت انجام می‌شوند. هر یک از زیرمسئله‌های حل شده در خطوط ۴ و ۵ بر روی یک زیرآرایه با $n/۲$ عنصر است (این فرض که اندازه‌ی مسئله‌ی اصلی توانی از ۲ است، تضمین می‌کند که $n/۲$ یک عدد صحیح باشد)، و بنابراین $T(n/۲)$ زمان برای حل هر یک از آن‌ها صرف می‌کنیم. چون باید دو زیرمسئله را حل کنیم – یکی برای زیرآرایه‌ی چپ و یکی برای زیرآرایه‌ی راست – سهم خطوط ۴ و ۵ در زمان اجرا برابر است با $۲T(n/۲)$. همان طور که قبلا دیدیم، فراخوانی FIND-MAX-CROSSING-SUBARRAY در خط ۶ در زمان $\theta(n)$ انجام می‌شوند. خطوط ۷-۱۱ فقط به زمان $\theta(۱)$ نیاز دارند. بنابراین برای حالت بازگشتی داریم

$$\begin{aligned} T(n) &= \theta(۱) + ۲T(n/۲) + \theta(n) + \theta(۱) \\ &= ۲T(n/۲) + \theta(n) \end{aligned} \tag{۴-۶}$$

ترکیب تساوی‌های (۴-۵) و (۴-۶) یک رابطه‌ی بازگشتی برای زمان اجرای FIND-MAXIMUM-SUBARRAY به ما می‌دهد:

$$T(n) = \begin{cases} \theta(۱) & \text{اگر } n = ۱ \\ ۲T(n/۲) + \theta(n) & \text{اگر } n > ۱ \end{cases} \tag{۴-۷}$$

این رابطه‌ی بازگشتی مشابه رابطه‌ی (۴-۱) برای مرتب‌سازی ادغامی است. همان طور که در بخش ۴-۵ با استفاده از متد اصلی خواهیم دید، جواب این رابطه‌ی بازگشتی برابر است با $T(n) = \theta(n \lg n)$. همچنین می‌توانید دوباره درخت بازگشتی شکل ۲-۵ را مشاهده کنید تا ببینید که چرا جواب $T(n) = \theta(n \lg n)$ است.

بنابراین می‌بینیم که روش تقسیم و حل به الگوریتمی ختم می‌شود که از روش بی‌خردانه سریع‌تر است. با مرتب‌سازی ادغامی، و اکنون با مسئله‌ی زیرآرایه‌ی بیشینه، کم‌کم متوجه می‌شویم که متد تقسیم و حل چقدر می‌تواند قدرت‌مند باشد. بعضی مواقع این متد به سریع‌ترین الگوریتم ممکن از نظر حدی برای یک مسئله ختم می‌شود، و بعضی مواقع می‌توان از روش‌های دیگر به الگوریتم‌های

سریع‌تر از آن هم دست یافت. همان طور که تمرین ۴-۱-۵ نشان می‌دهد، در واقع یک الگوریتم با زمان خطی برای مسئله‌ی زیرآرایه‌ی بیشینه وجود دارد که از روش تقسیم و حل استفاده نمی‌کند.

### تمرین‌ها

۴-۱-۱ اگر تمام عناصر $A$ منفی باشند، FIND-MAXIMUM-SUBARRAY چه چیزی بازمی‌گرداند؟

۴-۱-۲ یک شبه‌کد برای روش بی‌خردانه‌ی حل مسئله‌ی زیرآرایه‌ی بیشینه بنویسید. شبه‌کد شما باید در زمان $\theta(n^2)$ اجرا شود.

۴-۱-۳ هر دو الگوریتم بی‌خردانه و بازگشتی را برای حل مسئله‌ی زیرآرایه‌ی بیشینه بر روی کامپیوتر خود پیاده‌سازی کنید. اندازه‌ی $n_0$ برای مسئله که بعد از آن الگوریتم بازگشتی، الگوریتم بی‌خردانه را از نظر سرعت اجرا شکست می‌دهد (نقطه‌ی برخورد)، چیست؟ سپس حالت پایه‌ی الگوریتم بازگشتی را تغییر دهید به طوری که هر گاه که اندازه‌ی مسئله کوچک‌تر از $n_0$ است، از الگوریتم بی‌خردانه استفاده کند. آیا این کار نقطه‌ی برخورد را تغییر می‌دهد؟

۴-۱-۴ فرض کنید تعریف مسئله‌ی زیرآرایه‌ی بیشینه را تغییر می‌دهیم به طوری که نتیجه می‌تواند یک زیرآرایه‌ی تهی باشد. مجموع مقادیر یک زیرآرایه‌ی تهی را برابر با ۰ در نظر می‌گیریم. در هر یک از این الگوریتم‌ها چه تغییری باید بدهیم تا اجازه دهند که زیرآرایه‌ی جواب، تهی باشد؟

۴-۱-۵ با استفاده از ایده‌های زیر یک الگوریتم غیر بازگشتی با زمان خطی برای مسئله‌ی زیرآرایه‌ی مازکزیمم ارائه کنید. از انتهای چپ آرایه آغاز کرده و به سمت راست ادامه دهید، و در عین حال اطلاعات زیرآرایه‌ی بیشینه‌ای که تا بدین لحظه مشاهده کرده‌اید را در جایی ذخیره کنید. با اطلاع از زیرآرایه‌ی بیشینه در زیرآرایه‌ی $A[1..j]$، با استفاده از مشاهده‌ی زیر، جواب را به زیرآرایه‌ی $A[1..j+1]$ گسترش دهید: یک زیرآرایه‌ی بیشینه از $A[1..j+1]$، یا یک زیرآرایه‌ی بیشینه از $A[1..j]$ است، و یا یک زیرآرایه‌ی $A[i..j+1]$ که در آن $1 \le i \le j+1$. با دانستن زیرآرایه‌ی بیشینه‌ای که در اندیس $j$ پایان می‌یابد، یک زیرآرایه‌ی بیشینه به صورت $A[i..j+1]$ را در زمان ثابت به دست آورید.

## ۴-۲ الگوریتم استراسن برای ضرب ماتریس‌ها

اگر از قبل با ماتریس‌ها آشنایی داشته باشید، احتمالاً نحوه‌ی ضرب آن‌ها در یکدیگر را هم بلدید. (در غیر این صورت باید بخش ت-۱ از پیوست ت را بخوانید.) اگر $A = (a_{ij})$ و $B = (b_{ij})$ ماتریس‌های مربعی $n \times n$ باشند، در ضرب $C = A \cdot B$ درایه‌ی $c_{ij}$ را برای $i, j = 1, 2, \ldots, n$ به صورت زیر

تعریف می‌کنیم:

$$c_{ij} = \sum_{i=1}^{n} a_{ik} \cdot b_{kj} \qquad (۸-۴)$$

باید $n^2$ داریه‌ی ماتریسی را محاسبه کنیم، که هر کدام مجموع $n$ مقدار هستند. رویه‌ی زیر دو ماتریس $A$ و $B$ با اندازه‌ی $n \times n$ را گرفته و ماتریس حاصل‌ضرب $C$ با اندازه‌ی $n \times n$ را محاسبه کرده و بازمی‌گرداند. فرض می‌کنیم هر ماتریس یک خصیصه‌ی *rows* دارد که نشان‌دهنده‌ی تعداد سطرهای ماتریس است.

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

رویه‌ی SQUARE-MATRIX-MULTIPLY به صورت زیر کار می‌کند. حلقه‌ی for در خطوط ۳-۷ درایه‌های هر سطر $i$ را محاسبه می‌کند، و در هر سطر $i$، حلقه‌ی for خطوط ۴-۷ هر یک از درایه‌های $c_{ij}$ را برای هر ستون $j$ به دست می‌آورد. خط ۵ متغیر $c_{ij}$ را با ۰ مقداردهی اولیه می‌کند تا محاسبه‌ی مجموع داده شده در تساوی (۴-۸) را آغاز کنیم، و هر تکرار حلقه‌ی for در خطوط ۶-۷ یک جمله از تساوی (۴-۸) را به نتیجه اضافه می‌کند.

چون هر یک از سه حلقه‌ی for تودرتو دقیقاً $n$ بار تکرار می‌شوند، و هر تکرار خط ۷ به زمان خطی نیاز دارد، رویه‌ی SQUARE-MATRIX-MULTIPLY در زمان $\theta(n^3)$ اجرا می‌شود.

در ابتدا ممکن است فکر کنید که هر الگوریتم ضرب ماتریس‌ها باید $\Omega(n^3)$ زمان بگیرد، چرا که تعریف طبیعی ضرب ماتریس‌ها به این تعداد ضرب نیاز دارد. ولی چنین چیزی درست نیست: روشی برای ضرب ماتریس‌ها وجود دارد با زمان اجرای $o(n^3)$. در این بخش الگوریتم بازگشتی قابل توجه استراسن (Strassen) را برای ضرب ماتریس‌های $n \times n$ خواهیم دید. این الگوریتم در زمان $\theta(n^{\lg 7})$ اجرا می‌شود، که این زمان اجرا را در بخش ۴-۵ اثبات خواهیم کرد. از آن جایی که $\lg 7$ بین ۲٫۸۰ و ۲٫۸۱ قرار دارد، الگوریتم استراسن در زمان $O(2{,}81)$ اجرا می‌شود، که به صورت حدی بهتر از رویه‌ی ساده‌ی SQUARE-MATRIX-MULTIPLY است.

## یک الگوریتم تقسیم و حل ساده

برای این که کارها ساده‌تر انجام شود، وقتی که از الگوریتم تقسیم و حل برای محاسبه‌ی ضرب

ماتریسی $C = A \cdot B$ استفاده می‌کنیم، فرض می‌کنیم که $n$ در هر یک از ماتریس‌های $n \times n$ توانی از ۲ است. این فرض را می‌کنیم چون که در هر مرحله‌ی تقسیم، ماتریس‌های $n \times n$ را بـه چهـار مـاتریس $n/۲ \times n/۲$ تقسیم می‌کنیم، و با این فرض که $n$ توانی از ۲ است، تضمین می‌کنیم که تـا وقتـی $n \geq ۲$، حاصل تقسیم $n/۲$ یک عدد صحیح است.

فرض کنید هر یک از ماتریس‌های $A$، $B$، و $C$ را به چهار ماتریس $n/۲ \times n/۲$

$$A = \begin{pmatrix} A_{۱۱} & A_{۱۲} \\ A_{۲۱} & A_{۲۲} \end{pmatrix}, \quad B = \begin{pmatrix} B_{۱۱} & B_{۱۲} \\ B_{۲۱} & B_{۲۲} \end{pmatrix}, \quad C = \begin{pmatrix} C_{۱۱} & C_{۱۲} \\ C_{۲۱} & C_{۲۲} \end{pmatrix} \tag{۹-۴}$$

تقسیم کرده‌ایم به طوری که تساوی $C = A \cdot B$ به شکل زیر در آید:

$$\begin{pmatrix} C_{۱۱} & C_{۱۲} \\ C_{۲۱} & C_{۲۲} \end{pmatrix} = \begin{pmatrix} A_{۱۱} & A_{۱۲} \\ A_{۲۱} & A_{۲۲} \end{pmatrix} \cdot \begin{pmatrix} B_{۱۱} & B_{۱۲} \\ B_{۲۱} & B_{۲۲} \end{pmatrix} \tag{۱۰-۴}$$

تساوی (۴-۱۰) معادل است با چهار تساوی

$$C_{۱۱} = A_{۱۱} \cdot B_{۱۱} + A_{۱۲} \cdot B_{۲۱}, \tag{۱۱-۴}$$

$$C_{۱۲} = A_{۱۱} \cdot B_{۱۲} + A_{۱۲} \cdot B_{۲۲}, \tag{۱۲-۴}$$

$$C_{۲۱} = A_{۲۱} \cdot B_{۱۱} + A_{۲۲} \cdot B_{۲۱}, \tag{۱۳-۴}$$

$$C_{۲۲} = A_{۲۱} \cdot B_{۱۲} + A_{۲۲} \cdot B_{۲۲}. \tag{۱۴-۴}$$

هر یک از این چهار تساوی حاوی دو ضرب ماتریس‌های $n/۲ \times n/۲$ و جمع حاصل‌ضـرب $n/۲ \times n/۲$ آن‌ها است. با استفاده از این تساوی‌ها می‌توان یک الگوریتم بازگشتی تقسیم و حل سرراست توسعه داد.

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  if n == 1
4      c11 = a11 · b11
5  else partition A, B, and C as in equations (4-9)
6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10 return C
```

این شبه‌کد نگاهی دارد به یک نکته‌ای ظریف ولی مهم در پیاده‌سازی. چگونه در خط ۵ ماتریس‌ها را تقسیم‌بندی می‌کنیم؟ اگر می‌خواستیم ۱۲ ماتریس جدید $n/2 \times n/2$ بسازیم، باید $\theta(n^2)$ زمان صرف کپی درایه‌های ماتریس‌ها می‌کردیم. در واقع می‌توانیم بدون کپی درایه‌ها ماتریس‌ها را تقسیم‌بندی کنیم، که راه آن استفاده از محاسبات اندیسی است. یک زیرماتریس را با دامنه‌ای از اندیس‌های سطری و دامنه‌ای از اندیس‌های ستونی در ماتریس اولیه می‌شناسیم. با این روش زیرماتریس‌ها را با اندکی تفاوت از ماتریس اولیه نشان می‌دهیم، که همان نکته‌ی ظریفی است که به آن اشاره شد. فایده‌ی این کار این است: از آن جایی که زیرماتریس‌ها را با محاسبات اندیسی مشخص می‌کنیم، اجرای خط ۵ فقط به زمان $\theta(1)$ نیاز دارد (هر چند بعداً خواهیم دید که این در کران حدی زمان اجرا تأثیری ندارد که درایه‌ها را کپی کنیم یا تقسیم‌بندی را درجا انجام دهیم).

اکنون یک رابطه‌ی بازگشتی ارائه می‌کنیم که نشان‌دهنده‌ی زمان اجرای SQUARE-MATRIX-MULTIPLY-RECURSIVE است. فرض کنید $T(n)$ زمان ضرب دو ماتریس $n \times n$ با استفاده از این رویه باشد. در حالت پایه، وقتی $n = 1$، فقط یک ضرب اسکالر در خط ۴ انجام می‌دهیم، و بنابراین

$$T(1) = \theta(1) \tag{۴-۱۵}$$

حالت بازگشتی وقتی رخ می‌دهد که $n > 1$. همان طور که بحث شد، با استفاده از محاسبات اندیسی تقسیم ماتریس‌ها در خط ۵ به زمان $\theta(1)$ نیاز دارد. در خطوط ۶-۹ به صورت بازگشتی و به تعداد هشت بار، رویه‌ی SQUARE-MATRIX-MULTIPLY-RECURSIVE را فراخوانی می‌کنیم. چون هر فراخوانی بازگشتی دو ماتریس $n/2 \times n/2$ را در هم ضرب می‌کند و $T(n/2)$ به کل زمان اجرا اضافه می‌کند، کل زمان صرف شده در هشت فراخوانی بازگشتی برابر است با $8T(n/2)$. همچنین باید چهار جمع ماتریسی انجام شده در خطوط ۶-۹ را در نظر بگیریم. هر یک از این ماتریس‌ها حاوی $n^2/4$ درایه است، و بنابراین هر یک از چهار جمع ماتریسی به $\theta(n^2)$ زمان نیاز دارد. چون تعداد جمع‌های ماتریسی ثابت است، کل زمان صرف شده برای جمع ماتریس‌ها در خطوط ۶-۹ برابر است با $\theta(n^2)$. (دوباره از محاسبات اندیسی استفاده می‌کنیم تا نتیجه‌ی جمع‌های ماتریسی را در مکان‌های مناسب ماتریس $C$ قرار دهیم، که یک سربار $\theta(1)$ برای هر ورودی ایجاد می‌کند.) بنابراین کل زمان حالت بازگشتی برابر است با زمان تقسیم‌بندی، زمان تمام فراخوانی‌های بازگشتی، و زمان مورد نیاز برای جمع ماتریس‌های حاصل از فراخوانی‌های بازگشتی:

$$\begin{aligned} T(n) &= \theta(1) + 8T(n/2) + \theta(n^2) \\ &= 8T(n/2) + \theta(n^2) \end{aligned} \tag{۴-۱۶}$$

توجه کنید که اگر تقسیم‌بندی را با استفاده از کپی ماتریس‌ها انجام می‌دادیم، که $\theta(n^2)$ زمان می‌گرفت، رابطه‌ی بازگشتی تغییری نمی‌کرد و بنابراین کل زمان اجرا فقط با یک فاکتور ثابت افزایش پیدا می‌کرد.

ترکیب تساوی‌های (۴-۱۵) و (۴-۱۶) رابطه‌ی بازگشتی برای زمان اجرای رویه‌ی

SQUARE-MATRIX-MULTIPLY-RECURSIVE را به دست می‌دهد:

$$T(n) = \begin{cases} \theta(1) & \text{اگر } n = 1 \\ 8T(n/2) + \theta(n^2) & \text{اگر } n > 1 \end{cases} \qquad (۴-۱۷)$$

همان طور که از قضیه‌ی اصلی در بخش ۴-۵ خواهیم دید، جواب رابطه‌ی بازگشتی (۴-۱۷) عبارت است از $T(n) = \theta(n^3)$. بنابراین این رویکرد ساده‌ی تقسیم و حل از نظر حدی سریع‌تر از رویه‌ی SQUARE-MATRIX-MULTIPLY نیست.

قبل از این که بررسی الگوریتم استراسن را آغاز کنیم، اجازه دهید ببینیم که عناصر تساوی (۴-۱۶) چگونه حاصل می‌شوند. تقسیم‌بندی هر ماتریس $n \times n$ به کمک محاسبات اندیسی $\theta(1)$ زمان می‌گیرد، ولی دو ماتریس را باید تقسیم‌بندی کنیم. با این که می‌توانیم بگوییم تقسیم‌بندی دو ماتریس به $\theta(2)$ زمان نیاز دارد، ثابت ۲ در نماد $\theta$ جذب می‌شود. جمع دو ماتریس هر کدام با مثلاً $k$ ورودی، به $\theta(k)$ زمان نیاز دارد. از آن جایی که ماتریس‌هایی که با هم جمع می‌کنیم هر کدام $n^2/4$ ورودی دارند، می‌توان گفت که جمع هر جفت از ماتریس‌ها به $\theta(n^2/4)$ زمان نیاز دارد. هر چند دوباره، نماد $\theta$ ضریب ثابت ۱/۴ را در خود جذب می‌کند. چهار جمع ماتریسی مانند این داریم، و دوباره به جای این که بگوییم این اعمال $\theta(4n^2)$ زمان می‌گیرند، می‌گوییم $\theta(n^2)$ زمان می‌گیرند. (مسلماً مشاهده کرده‌اید که می‌توانستیم بگوییم چهار ضرب ماتریسی $\theta(4n^2/4)$ زمان می‌گیرند، و $4n^2/4 = n^2$، ولی نکته این جا است که نماد $\theta$ ضرایب ثابت را در خود جذب می‌کند، هر چه که باشند.) بنابراین دو جمله‌ی $\theta(n^2)$ برای ما باقی می‌ماند، که می‌توانیم آن‌ها را در یکی ترکیب کنیم.

ولی وقتی هشت فراخوانی بازگشتی را در نظر می‌گیریم، نمی‌توانیم ضریب ۸ را جذب کنیم. به عبارت دیگر باید بگوییم که این فراخوانی‌ها مجموعا $8T(n/2)$ زمان می‌گیرند، نه $T(n/2)$. دلیل را می‌توانید با نگاهی به درخت بازگشتی شکل ۲-۵ برای رابطه‌ی (۲-۱) (که مشابه رابطه‌ی (۴-۷) است) درک کنید، که در آن $T(n) = 2T(n/2) + \theta(n)$. ضریب ۲ تعیین می‌کند هر گره در درخت بازگشتی چند فرزند دارد، که آن هم به نوبه‌ی خود تعیین می‌کند در هر سطح درخت چند جمله به مجموع کلی اضافه می‌شود. اگر می‌خواستیم ضریب ۸ را در تساوی (۴-۱۶)، و یا ضریب ۲ را در تساوی (۴-۱) در نظر نگیریم، درخت بازگشتی خطی بود نه «انبوه»، و هر سطح فقط یک جمله به مجموع کل اضافه می‌کرد.

بنابراین در خاطر داشته باشید با این که نمادهای حدی ضرایب ثابت را در خود جذب می‌کنند، نمادهای بازگشتی مانند $T(n/2)$ این کار را نمی‌کنند.

## روش استراسن

کلید روش استراسن این است که کمی از انبوهیت درخت بازگشتی بکاهیم. یعنی به جای انجام هشت ضرب بازگشتی بر روی ماتریس‌های $n/2 \times n/2$، فقط ۷ ضرب انجام دهیم. هزینه‌ی حذف یک ضرب ماتریسی چندین جمع $n/2 \times n/2$ اضافی است، ولی تعداد این جمع‌ها ثابت است. مانند قبل وقتی رابطه‌ی بازگشتی زمان اجرا را می‌نویسیم، تعداد ثابت جمع‌های ماتریسی در نماد $\theta$ جذب می‌شود.

روش استراسن اصلاً بدیهی نیست. (این احتمالاً بزرگ‌ترین دست‌کم گیری در این کتاب است.) این روش چهار مرحله دارد:

۱. تقسیم ماتریس‌های ورودی $A$ و $B$ و ماتریس خروجی $C$ به ماتریس‌های $n/۲\times n/۲$ طبق تساوی (۴-۹). این مرحله به کمک محاسبات اندیسی در زمان $\theta(۱)$ انجام می‌شود، درست مانند رویه‌ی SQUARE-MATRIX-MULTIPLY-RECURSIVE.

۲. ساختن ۱۰ ماتریس $S_۱,S_۲,...,S_{۱۰}$، که هر یک از آن‌ها یک ماتریس $n/۲\times n/۲$ حاصل از جمع یا تفریق دو تا از ماتریس‌های ساخته شده در مرحله‌ی ۱ است. تمام این ۱۰ ماتریس را می‌توانیم در زمان $\theta(n^۲)$ بسازیم.

۳. محاسبه‌ی هفت ضرب ماتریسی $P_۱,P_۲,...,P_۷$ به صورت بازگشتی و با استفاده از زیرماتریس‌های ساخته شده در مرحله‌ی ۱ و ماتریس‌های ساخته شده در مرحله‌ی ۲. هر یک $P_i$ها یک ماتریس $n/۲\times n/۲$ است.

۴. محاسبه‌ی زیرماتریس‌های نهایی $C_{۱۱},C_{۱۲},C_{۲۱},C_{۲۲}$ از ماتریس نتیجه‌ی $C$، با جمع و تفریق ترکیب‌های مختلف ماتریس‌های $P_i$. تمام چهار زیرماتریس را می‌توانیم در زمان $\theta(n^۲)$ محاسبه کنیم.

به زودی جزئیات مراحل ۲-۴ را خواهیم دید، ولی تا همین جا برای ارائه‌ی یک رابطه‌ی بازگشتی برای زمان اجرای روش استراسن اطلاعات کافی داریم. اجازه دهید فرض کنیم وقتی اندازه‌ی ماتریس‌ها به ۱ می‌رسد، یک ضرب ساده‌ی اسکالر انجام می‌دهیم، درست مانند خط ۴ رویه‌ی SQUARE-MATRIX-MULTIPLY-RECURSIVE. وقتی $n>۱$، مراحل ۱، ۲، و ۴ در کل $\theta(n^۲)$ زمان می‌گیرند، و مرحله‌ی ۳ به هفت ضرب ماتریسی $n/۲\times n/۲$ نیاز دارد. بنابراین به رابطه‌ی بازگشتی زیر برای زمان اجرای الگوریتم استراسن می‌رسیم:

$$T(n)=\begin{cases}\theta(۱) & \text{اگر } n=۱\\ ۷T(n/۲)+\theta(n^۲) & \text{اگر } n>۱\end{cases} \qquad (۴-۱۸)$$

در روش بالا یکی از ضرب‌های ماتریسی را حذف کردیم، که هزینه‌ی آن تعداد ثابتی جمع ماتریسی بود. وقتی روابط بازگشتی و معنی آن‌ها را بهتر درک کنیم، می‌بینیم که این کار در واقع زمان اجرای حدی را پایین می‌آورد. طبق متد اصلی در بخش ۴-۵ جواب رابطه‌ی (۴-۱۸) عبارت است از $T(n)=\theta(n^{\lg ۷})$.

اکنون به جزئیات می‌پردازیم. در مرحله‌ی دو، ۱۰ ماتریس زیر را می‌سازیم:

$$S_۱=B_{۱۲}-B_{۲۲},$$
$$S_۲=A_{۱۱}+A_{۱۲},$$
$$S_۳=A_{۲۱}+A_{۲۲},$$
$$S_۴=B_{۲۱}-B_{۱۱},$$
$$S_۵=A_{۱۱}+A_{۲۲},$$

$$S_۶ = B_{۱۱} + B_{۲۲},$$
$$S_۷ = A_{۱۲} - A_{۲۲},$$
$$S_۸ = B_{۲۱} + B_{۲۲},$$
$$S_۹ = A_{۱۱} - A_{۲۱},$$
$$S_{۱۰} = B_{۱۱} + B_{۱۲}.$$

از آن جایی که باید ماتریس‌های $n/۲ \times n/۲$ را ۱۰ بار جمع یا تفریق کنیم، این مرحله بـه $\theta(n^۲)$ زمـان نیاز دارد.

در مرحله‌ی ۳ به صورت بازگشتی ماتریس‌های $n/۲ \times n/۲$ را هفـت بـار در هـم ضـرب مـی‌کنـیم تـا مـاتریس‌هـای $n/۲ \times n/۲$ زیـر را بـه دسـت آوریـم، کـه هـر کـدام از آن‌هـا جمـع یـا تفریقـی از حاصل‌ضرب‌های زیرماتریس‌های $A$ و $B$ هستند:

$$P_۱ = A_{۱۱} \cdot S_۱ = A_{۱۱} \cdot B_{۱۲} - A_{۱۱} \cdot B_{۲۲},$$
$$P_۲ = S_۲ \cdot B_{۲۲} = A_{۱۱} \cdot B_{۲۲} + A_{۱۲} \cdot B_{۲۲},$$
$$P_۳ = S_۳ \cdot B_{۱۱} = A_{۲۱} \cdot B_{۱۱} + A_{۲۲} \cdot B_{۱۱},$$
$$P_۴ = A_{۲۲} \cdot S_۴ = A_{۲۲} \cdot B_{۲۱} - A_{۲۲} \cdot B_{۱۱},$$
$$P_۵ = S_۵ \cdot S_۶ = A_{۱۱} \cdot B_{۱۱} + A_{۱۱} \cdot B_{۲۲} + A_{۲۲} \cdot B_{۱۱} + A_{۲۲} \cdot B_{۲۲},$$
$$P_۶ = S_۷ \cdot S_۸ = A_{۱۲} \cdot B_{۲۱} + A_{۱۲} \cdot B_{۲۲} - A_{۲۲} \cdot B_{۲۱} - A_{۲۲} \cdot B_{۲۲},$$
$$P_۷ = S_۹ \cdot S_{۱۰} = A_{۱۱} \cdot B_{۱۱} + A_{۱۱} \cdot B_{۱۲} - A_{۲۱} \cdot B_{۱۱} - A_{۲۱} \cdot B_{۱۲}.$$

توجه کنید که تنها ضرب‌هایی که باید انجام دهیم، ضرب‌های ستون وسـط تسـاوی‌هـای بـالا هسـتند. سمت راست فقط این ضرب‌ها را بر حسب زیرماتریس‌های اصلی ساخته شـده در مرحلـه‌ی ۱ نشـان می‌دهد.

در مرحله‌ی ۴ ماتریس‌های $P_i$ ساخته شده در مرحلـه‌ی ۳ را جمـع و تفریـق مـی‌کنـیم تـا چهـار زیرماتریس $n/۲ \times n/۲$ ماتریس حاصل‌ضرب $C$ را بسازیم. با عملیات زیر آغاز می‌کنیم:

$$C_{۱۱} = P_۵ + P_۴ - P_۲ + P_۶$$

سمت راست تساوی را باز می‌کنیم تا ببینیم $C_{۱۱}$ معادل چیست. برای سادگی در خوانـدن هـر یـک از $P_i$ها را در یک خط نشان داده، و جمله‌های مشابه را در ستون‌های یکسان قرار می‌دهیم:

$$\begin{array}{llll} A_{۱۱} \cdot B_{۱۱} & + A_{۱۱} \cdot B_{۲۲} & + A_{۲۲} \cdot B_{۱۱} & + A_{۲۲} \cdot B_{۲۲} \\ & - A_{۲۲} \cdot B_{۱۱} & & + A_{۲۲} \cdot B_{۲۱} \\ & - A_{۱۱} \cdot B_{۲۲} & & - A_{۱۲} \cdot B_{۲۲} \\ & & - A_{۲۲} \cdot B_{۲۲} - A_{۲۲} \cdot B_{۲۱} + A_{۱۲} \cdot B_{۲۲} + A_{۱۲} \cdot B_{۲۱} & \\ \hline A_{۱۱} \cdot B_{۱۱} & & & + A_{۱۲} \cdot B_{۲۱} \end{array}$$

که متناظر است با تساوی (۴-۱۱).

به طور مشابه قرار می‌دهیم $C_{۱۲} = P_۱ + P_۲$، و بنابراین $C_{۱۲}$ برابر است با

$$\begin{array}{l} A_{۱۱} \cdot B_{۱۲} - A_{۱۱} \cdot B_{۲۲} \\ \qquad + A_{۱۱} \cdot B_{۲۲} + A_{۱۲} \cdot B_{۲۲} \\ \hline A_{۱۱} \cdot B_{۱۲} \qquad\qquad + A_{۱۲} \cdot B_{۲۲} \end{array}$$

که متناظر است با تساوی (۴-۱۲).

قرار دادن $C_{۲۱} = P_۳ + P_۴$، تساوی زیر را برای $C_{۲۱}$ به ما خواهد داد:

$$\begin{array}{l} A_{۲۱} \cdot B_{۱۱} + A_{۲۲} \cdot B_{۱۱} \\ \qquad - A_{۲۲} \cdot B_{۱۱} + A_{۲۲} \cdot B_{۲۱} \\ \hline A_{۲۱} \cdot B_{۱۲} \qquad\qquad + A_{۲۲} \cdot B_{۲۲} \end{array}$$

که متناظر است با تساوی (۴-۱۳).

نهایتاً قرار می‌دهیم $C_{۲۲} = P_۵ + P_۱ - P_۳ - P_۷$ که برابر خواهد بود با

$$\begin{array}{llll} A_{۱۱} \cdot B_{۱۱} + A_{۱۱} \cdot B_{۲۲} + A_{۲۲} \cdot B_{۱۱} + A_{۲۲} \cdot B_{۲۲} & & & \\ \quad - A_{۱۱} \cdot B_{۲۲} & & + A_{۱۱} \cdot B_{۱۲} & \\ & - A_{۲۲} \cdot B_{۱۱} & & - A_{۲۱} \cdot B_{۱۱} \\ - A_{۱۱} \cdot B_{۱۱} & & - A_{۱۱} \cdot B_{۱۲} + A_{۲۱} \cdot B_{۱۱} + A_{۲۱} \cdot B_{۱۲} & \\ \hline & A_{۲۲} \cdot B_{۲۲} & & + A_{۲۱} \cdot B_{۱۲} \end{array}$$

که متناظر است با تساوی (۴-۱۴). به طور کلی در مرحله‌ی ۴، هشت بار ماتریس‌های $n/۲ \times n/۲$ را با هم جمع یا از هم تفریق می‌کنیم، و در نتیجه این مرحله به $\theta(n^۲)$ زمان نیاز دارد.

بنابراین می‌بینیم که الگوریتم استراسن، که از مراحل ۱ تا ۴ نشان داده شده به دست می‌آید، به درستی حاصل‌ضرب ماتریس‌ها را محاسبه می‌کند، و رابطه‌ی بازگشتی (۴-۱۸) نشان‌دهنده‌ی زمان اجرای آن است. از آن جایی که در بخش ۴-۵ خواهیم دید که جواب این رابطه‌ی بازگشتی $T(n) = \theta(n^{\lg ۷})$ است، الگوریتم استراسن به صورت حدی از رویه‌ی سرراست SQUARE-MATRIX-MULTIPLY سریع‌تر است.

## تمرین‌ها

۴-۲-۱ با استفاده از الگوریتم استراسن ضرب ماتریسی زیر را محاسبه کنید:

$$\begin{pmatrix} ۱ & ۳ \\ ۷ & ۵ \end{pmatrix} \begin{pmatrix} ۶ & ۸ \\ ۴ & ۲ \end{pmatrix}$$

روند کار خود را نشان دهید.

۴-۲-۲ یک رویه برای الگوریتم استراسن بنویسید.

۴-۲-۳ چطور می‌توان الگوریتم استراسن طوری تغییر داد تا ماتریس‌های $n \times n$ را در شرایطی محاسبه کند که $n$ توانی از ۲ نیست؟ نشان دهید که الگوریتم حاصل در زمان $\theta(n^{\lg ۷})$

اجرا می‌شود.

۴-۲-۴ اگر بتوانیم ماتریس‌های ۳×۳ را با $k$ ضرب اسکالر (بدون در نظر گرفتن جابه‌جایی‌پذیری ضرب) در هم ضرب کنیم، بزرگ‌ترین $k$ چند است به طوری که بتوانیم ماتریس‌های $n\times n$ را در زمان $o(n^{\lg 7})$ در هم ضرب کنیم؟ زمان اجرای این الگوریتم چگونه خواهد بود؟

۴-۲-۵ V.Pan روشی ابداع کرده است که به وسیله‌ی آن می‌توان ماتریس‌های ۶۸×۶۸ را با ۱۳۲,۴۶۴ ضرب، ماتریس‌های ۷۰×۷۰ را با ۱۴۳,۶۴۰ ضرب، و ماتریس‌های ۷۲×۷۲ را با ۱۵۵,۴۲۴ ضرب در هم ضرب کرد. اگر از این روش‌ها در یک الگوریتم تقسیم و حل برای ضرب ماتریس‌ها استفاده کنیم، کدام یک سریع‌ترین زمان اجرای حدی را خواهد داشت؟ سرعت این روش نسبت به الگوریتم استراسن چگونه خواهد بود؟

۴-۲-۶ با استفاده از الگوریتم استراسن به عنوان یک رویه‌ی کمکی، با چه سرعتی می‌توانید یک ماتریس $kn\times n$ را در یک ماتریس $n\times kn$ ضرب کنید؟ همین سؤال را برای حالتی جواب دهید که ترتیب ماتریس‌های ورودی جابه‌جا شده است.

۴-۲-۷ نشان دهید که چگونه می‌توان فقط با استفاده از سه ضرب حقیقی، اعداد مختلط $a+bi$ و $c+di$ را در هم ضرب کرد. الگوریتم شما باید $a$، $b$، $c$، و $d$ را به عنوان ورودی دریافت کرده و قسمت حقیقی $ac-bd$ و قسمت موهومی $ad+bc$ را به صورت جداگانه ارائه کند.

## ۴-۳ روش جانشین‌سازی برای حل روابط بازگشتی

اکنون که دیدیم چگونه می‌توان از بازگشت‌ها برای توصیف زمان اجرای الگوریتم‌های تقسیم و حل استفاده کنیم، روش حل بازگشت‌ها را خواهیم آموخت. این بخش را با روش جانشین‌سازی آغاز می‌کنیم.

*روش جانشین‌سازی* (substitution method) برای حل روابط بازگشتی شامل دو مرحله است:

۱. حدس زدن شکل جواب.
۲. استفاده از استقرای ریاضی برای یافتن محتوای جواب و نشان دادن درستی آن.

هنگام اعمال فرض استقرا به مقادیر کوچک، جوابی که حدس زده‌ایم را جانشین تابع می‌کنیم؛ نام «روش جانشین‌سازی» از این جا حاصل می‌شود. این روش قدرت‌مند است، ولی برای به کار بردن آن باید اول شکل جواب را حدس بزنیم.

از روش جانشین‌سازی می‌توان برای تعیین هر دو نوع کران‌های حدی (بالا و پایین) در روابط بازگشتی استفاده کرد. برای مثال اجازه دهید کران بالای رابطه‌ی بازگشتی

$$T(n)=2T\left(\lfloor n/2\rfloor\right)+n \qquad (۴-۱۹)$$

را تعیین کنیم، که مشابه بازگشت‌های ۴-۳ و ۴-۴ است. حدس می‌زنیم که جواب $T(n) = O(n \lg n)$ باشد. اکنون برای استفاده از روش جانشین‌سازی، باید نشان دهیم برای انتخاب مناسبی از ثابت $c > 0$، رابطه‌ی $T(n) \le cn \lg n$ برقرار است. با این فرض آغاز می‌کنیم که رابطه‌ی مذکور برای هر $m < n$، به طور خاص برای $m = \lfloor n/2 \rfloor$ درست است، که نتیجه می‌دهد $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. جایگذاری در بازگشت به دست می‌دهد:

$$\begin{aligned} T(n) &\le 2\left(c \lfloor n/2 \rfloor \lg\left(\lfloor n/2 \rfloor\right)\right) + n \\ &\le cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\le cn \lg n \end{aligned}$$

که مرحله‌ی آخر در صورتی درست است که $c \ge 1$.

اکنون برای تکمیل استقرای ریاضی باید نشان دهیم جواب برای نقاط مرزی هم درست است. برای این کار، به طور معمول نشان می‌دهیم که مقادیر مرزی برای پایه‌ی استقرا مناسب هستند. برای رابطه‌ی بازگشتی ۴-۱۹ باید نشان دهیم که می‌توانیم مقدار ثابت $c$ را به حد کافی بزرگ انتخاب کنیم، به طوری که کران $T(n) = cn \lg n$ برای مقادیر مرزی هم برقرار باشد. این مسئله، در بعضی موارد می‌تواند مشکل‌ساز باشد. اجازه دهید فرض کنیم $T(1) = 1$ اولین نقطه‌ی مرزی رابطه‌ی بازگشتی است. بنابراین برای $n = 1$، کران $T(n) = cn \lg n$ نتیجه می‌دهد $T(1) = c1 \lg 1 = 0$، که با $T(1) = 1$ تناقض دارد، و اثبات پایه‌ی استقرا شکست می‌خورد.

در اثبات‌های استقرایی به سادگی می‌توان بر این گونه مسائل چیره شد. مثلاً در رابطه‌ی ۴-۱۹ از این نکته‌ی نمادهای حدی استفاده می‌کنیم که رابطه‌ی $T(n) \le cn \lg n$ فقط باید برای $n > n_0$ برقرار باشد، که $n_0$ ثابتی است که *انتخاب آن بر عهده‌ی ماست*. شرط مرزی $T(1) = 1$ را که کمی مشکل‌ساز است نگه می‌داریم، ولی در اثبات استقرایی در نظر نمی‌گیریم. دقت کنید برای $n > 3$ رابطه‌ی بازگشتی مستقل از $T(1)$ است. بنابراین می‌توانیم $T(1)$ را با $T(2)$ و $T(3)$ به عنوان پایه‌ی اثبات استقرایی جایگزین کنیم، یعنی $n_0 = 2$. توجه داشته باشید که بین پایه‌ی رابطه‌ی بازگشتی ($n = 1$) و پایه‌ی استقرا ($n = 2$ و $n = 3$) تفاوت وجود دارد. با $T(1) = 1$، از روی رابطه‌ی بازگشتی می‌بینیم که $T(2) = 4$ و $T(3) = 5$. اکنون با انتخاب ثابت $c \ge 1$ به گونه‌ای که $T(2) \le c2 \lg 2$ و $T(3) \le c3 \lg 3$ می‌توانیم اثبات استقرایی را کامل کنیم. می‌بینیم که هر انتخابی به شکل $c \ge 2$ برای برقرار بودن حالت‌های پایه‌ی $n = 2$ و $n = 3$ کافی است. برای اکثر روابط بازگشتی که بعداً خواهیم دید، ساده است که حالت پایه را برای برقرار بودن فرض استقرا برای $n$های کوچک به جلو ببریم، و همچنین اکثراً از جزئیات صرف نظر خواهیم کرد.

## انتخاب یک حدس مناسب

متأسفانه هیچ روش کلی برای حدس جواب درست روابط بازگشتی وجود ندارد. حدس جواب نیاز به تجربه، و ندرتاً نیاز به خلاقیت دارد. ولی خوشبختانه راه‌هایی وجود دارند که برای انتخاب یک حدس

خوب به شما کمک می‌کنند. همچنین می‌توانید از درخت‌های بازگشتی برای حدس‌های خوب استفاده کنید، که آن‌ها را در بخش ۴-۴ خواهیم دید.

اگر یک رابطه‌ی بازگشتی شبیه به رابطه‌ای است که قبلا دیده‌اید، حدس یک جواب مشابه منطقی خواهد بود. برای مثال رابطه‌ی زیر را در نظر بگیرید:

$$T(n) = ۲T\left(\lfloor n/۲ \rfloor + ۱۷\right) + n$$

که به خاطر اضافه شدن ۱۷ در لیست آرگومان‌های $T$ در سمت راست، به نظر دشوار می‌آید. ولی به طور شهودی این جمله‌ی اضافه اساساً نمی‌تواند بر روی جواب بازگشت تأثیر بگذارد. وقتی $n$ بزرگ می‌شود، تفاوت بین $T\left(\lfloor n/۲ \rfloor\right)$ و $T\left(\lfloor n/۲ \rfloor + ۱۷\right)$ چندان زیاد نخواهد بود: هر دوی آن‌ها تقریباً $n$ را نصف می‌کنند. در نتیجه حدس می‌زنیم که جواب $T(n) = O(n \lg n)$ باشد، که می‌توان به کمک روش جایگزینی درستی آن را اثبات کرد (تمرین ۴-۳-۶).

روش دیگر یافتن یک حدس خوب، این است که با تعیین کران‌های بالا و پایین نه چندان نزدیک برای رابطه، دامنه‌ی عدم قطعیت خود را کاهش دهیم. به عنوان مثال ممکن است برای رابطه‌ی ۴-۱۹، برای شروع کران پایین $T(n) = \Omega(n)$ را انتخاب کنیم، چرا که در بازگشت جمله‌ی $n$ را داریم و همچنین می‌توانیم کران بالای $T(n) = O(n^۲)$ را برای بازگشت اثبات کنیم. سپس می‌توانیم کم کم کران بالا را افزایش، و کران پایین را کاهش دهیم تا جواب ما به جواب درست میل کند، که همان $T(n) = \theta(n \lg n)$ است.

## مهارت‌ها

بعضی مواقع می‌توانید به درستی کران حدی جواب یک رابطه‌ی بازگشتی را حدس بزنید، ولی متوجه می‌شوید که استفاده از استقرای ریاضی برای اثبات آن بی‌فایده است. معمولاً مشکل این جا است که فرض استقرا برای اثبات جزئیات کران به اندازه‌ی کافی قوی نیست. وقتی به چنین مانعی برمی‌خوریم، بازنویسی حدس و کم کردن یک جمله با درجه‌ی پایین به استقرای ریاضی اجازه می‌دهد که کار خود را به درستی انجام دهد.

رابطه‌ی بازگشتی زیر را در نظر بگیرید:

$$T(n) = T\left(\lfloor n/۲ \rfloor\right) + T\left(\lceil n/۲ \rceil\right) + ۱$$

حدس می‌زنیم که جواب $T(n) = O(n)$ باشد، و سعی می‌کنیم نشان دهیم که برای یک انتخاب مناسب برای ثابت $c$ داریم $T(n) \le cn$. با جایگذاری این حدس در بازگشت، داریم:

$$\begin{aligned} T(n) &\le c\lfloor n/۲ \rfloor + c\lceil n/۲ \rceil + ۱ \\ &= cn + ۱ \end{aligned}$$

که $T(n) \le cn$ را برای هیچ انتخابی از $c$ نتیجه نمی‌دهد. در این جا ممکن است وسوسه بشویم که حدس خود را افزایش بدهیم، مثلاً $T(n) = O(n^۲)$ که به سادگی می‌توان آن را اثبات کرد، ولی در واقع همان جواب اول ($T(n) = O(n)$) درست است. با این حال برای اثبات این جواب باید فرضی قوی‌تر برای استقرا در نظر بگیریم.

به طور شهودی حدس ما تقریباً درست است: فقط به اندازه‌ی یک ثابت ۱ کم داریم، یک جمله‌ی درجه پایین. علی‌رقم این قضیه، استقرای ریاضی فقط زمانی به درستی کار می‌کند که فرض استقرای ما دقیق باشد. اکنون با کم کردن یک جمله‌ی با درجه‌ی پایین از حدس قبلی خود این مشکل را حل می‌کنیم. حدس جدید این خواهد بود: $T(n) \le cn - d$، که در آن $d \ge 0$ یک ثابت است. اکنون داریم:

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

در صورتی که $d \ge 1$ باشد. مانند قبل ثابت $c$ باید به گونه‌ای انتخاب شود که رابطه در مقادیر مرزی صدق کند.

ممکن است کم کردن یک جمله‌ی درجه پایین‌تر را غیر منطقی بدانید. اگر با ریاضی نتوانیم حدس خود را اثبات کنیم، باید حدس خود را افزایش دهیم، درست است؟ نه لزوماً! هنگام اثبات یک کران بالا با استفاده از استقرا، ممکن است اثبات یک کران بالای ضعیف‌تر واقعا سخت‌تر باشد، چرا که برای اثبات کران ضعیف‌تر باید از همان کران ضعیف‌تر در اثبات استقرایی استفاده کنیم. در مثال فعلی وقتی رابطه‌ی بازگشتی بیش از یک عبارت بازگشتی دارد، باید جمله‌ی با درجه‌ی پایین‌تر را از هر یک از عبارت‌های بازگشتی کران ارائه شده یک بار کم کنیم. در مثال بالا ثابت $d$ را دو بار کم کردیم، یک بار برای جمله‌ی $T(\lfloor n/2 \rfloor)$ و یک بار برای جمله‌ی $T(\lceil n/2 \rceil)$. در پایان به نامساوی $T(n) \le cn - 2d + 1$ رسیدیم، و یافتن مقادیری برای $d$ به طوری که $cn - 2d + 1$ کوچک‌تر از $cn - d$ باشد کار سختی نبود.

## دوری از دام‌ها

در استفاده از نمادهای حدی به راحتی ممکن است دچار خطا شویم. مثلاً در رابطه‌ی بازگشتی ۴-۱۹، می‌توانیم با حدس $T(n) = O(n)$، به غلط «اثبات» کنیم که $T(n) \le cn$، بدین صورت که

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

چرا که $c$ یک ثابت است. اشتباه ما این جا است که، *شکل دقیق* فرض استقرا را اثبات نکرده‌ایم، یعنی $T(n) \le cn$. بنابراین وقتی می‌خواهیم نشان دهیم که $T(n) = O(n)$، صریحاً اثبات خواهیم کرد که $T(n) \le cn$.

## تغییر متغیرها

بعضی مواقع با مقدار کمی‌عملیات جبری می‌توان یک رابطه‌ی بازگشتی نا آشنا را تبدیل به یک رابطه‌ی آشنا کرد. به عنوان یک مثال رابطه‌ی زیر را در نظر بگیرید:

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

که حل آن به نظر مشکل می‌آید. با این حال با تغییر متغیر می‌توانیم حل آن را راحت کنیم. برای راحتی نگران گرد کردن مقادیر، مثلاً $\sqrt{n}$، به اعداد صحیح نخواهیم بود. با جایگذاری $m = \lg n$ داریم:

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

اکنون می‌توانیم با نام گذاری $S(m) = T(2^m)$، رابطه‌ی بازگشتی

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

را بسازیم، که بسیار شبیه بازگشت ۴-۱۹ است. مسلماً این رابطه جوابی مشابه رابطه‌ی ۴-۱۹ دارد: $S(m) = O(m \lg m)$. با برگشت از $S(m)$ به $T(n)$ به دست می‌آوریم

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

## تمرین‌ها

۴-۳-۱ نشان دهید جواب رابطه‌ی $T(n) = T(n-1) + n$ برابر است با $O(n^2)$.

۴-۳-۲ نشان دهید جواب رابطه‌ی $T(n) = T(\lceil n/2 \rceil) + 1$ برابر است با $O(\lg n)$.

۴-۳-۳ قبلا دیدیم که جواب رابطه‌ی $T(n) = 2T(\lfloor n/2 \rfloor) + n$ برابر است با $O(n \lg n)$. نشان دهید جواب این بازگشت، $\Omega(n \lg n)$ هم هست. نتیجه بگیرید که $T(n) = \theta(n \lg n)$.

۴-۳-۴ نشان دهید که با در نظر گرفتن یک فرض استقرای متفاوت، می‌توانیم مشکل مقدار مرزی $T = 1$ را برای رابطه‌ی ۴-۴، بدون اصلاح مقادیر مرزی برای اثبات استقرایی حل کنیم.

۴-۳-۵ نشان دهید که $\theta(n \lg n)$، جواب رابطه‌ی بازگشتی «دقیق» (۴-۳) برای مرتب‌سازی ادغامی‌است.

۴-۳-۶ نشان دهید که جواب رابطه‌ی $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ برابر است با $O(n \lg n)$.

۴-۳-۷ با استفاده از متد اصلی در بخش ۴-۵، می‌توان نشان داد که جواب بازگشت $T(n) = T(n/3) + n$ برابر است با $T(n) = \theta(n^{\log_3 4})$. نشان دهید که اثبات با استفاده از روش جانشین‌سازی با فرض $T(n) \le cn^{\log_3 4}$ شکست می‌خورد. سپس نشان دهید که چگونه می‌توان با کم کردن یک جمله‌ی با درجه‌ی پایین‌تر، اثبات را به درستی انجام داد.

۴-۳-۸ با استفاده از متد اصلی در بخش ۴-۵ می‌توان نشان داد که جواب بازگشت $T(n) = T(n/2) + n^2$ برابر است با $T(n) = \theta(n^2)$. نشان دهید که یک اثبات با استفاده از روش جانشین‌سازی با فرض $T(n) \le cn^2$ شکست می‌خورد. سپس نشان دهید که چگونه می‌توان با کم کردن یک جمله‌ی با درجه‌ی پایین‌تر اثبات را به درستی انجام داد.

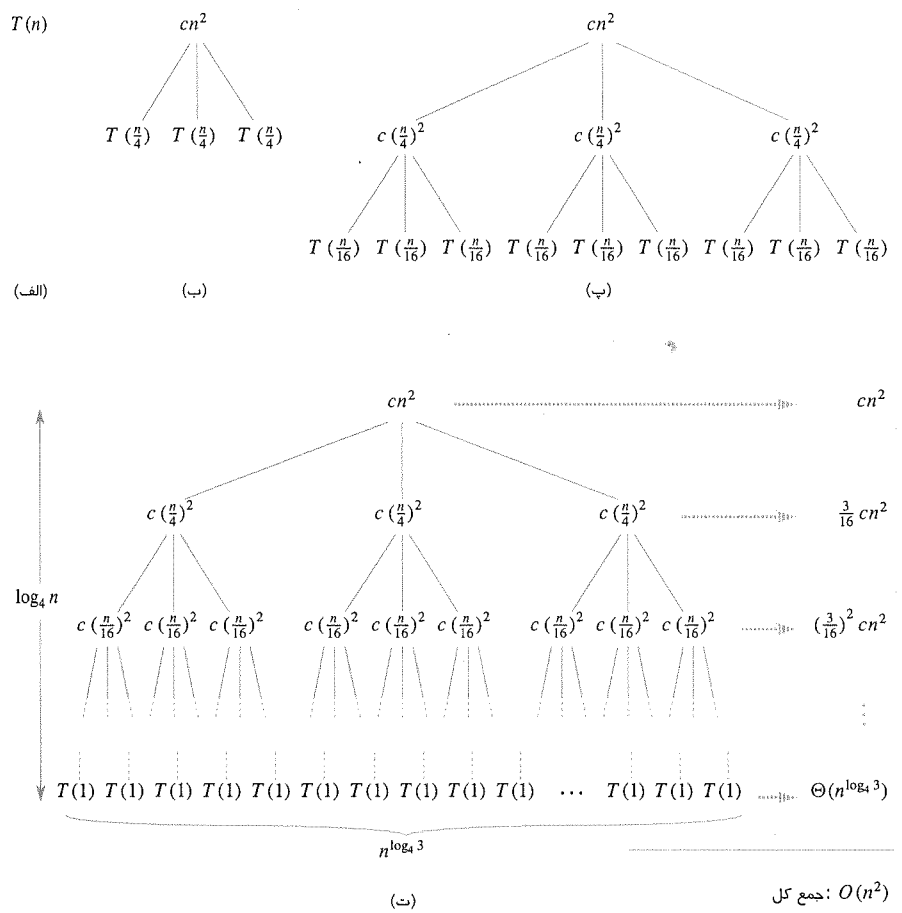
## ۴-۴ روش درخت بازگشتی برای حل بازگشت‌ها

با این که روش جایگزینی می‌تواند اثباتی مختصر و مفید برای درستی جواب رابطه‌های بازگشتی بدهد، بعضی مواقع پیدا کردن یک حدس مناسب برای آن مشکل است. رسم یک درخت بازگشتی، مانند کاری که در تحلیل رابطه‌ی بازگشتی مرتب‌سازی ادغامی در بخش ۲-۳-۲ کردیم، روشی سرراست برای یافتن یک حدس خوب است. در *درخت بازگشتی* (recursion tree) هر گره نشان‌دهنده‌ی هزینه‌ی یک زیرمسئله در مجموعه‌ی فراخوانی‌های بازگشتی تابع است. برای یافتن هزینه‌ی رابطه‌ی بازگشتی، هزینه‌های درون هر سطح را (قبل از اجرا شدن توابع در آن سطح)، با هم جمع می‌کنیم تا برای هر سطح یک هزینه‌ی اولیه به دست آوریم، و سپس هزینه‌ی اولیه‌ی تمام سطوح را با هم جمع می‌کنیم و هزینه‌ی کلی را به دست می‌آوریم.

بهتر است که از درخت بازگشتی برای یافتن یک حدس خوب استفاده کنیم، و سپس با استفاده از روش جایگزینی آن حدس را اثبات کنیم. وقتی از درخت بازگشتی برای یافتن یک حدس استفاده می‌کنید، معمولاً می‌توانید مقدار کمی «بی‌دقتی» در مقادیر به خرج دهید، چرا که بعداً درستی حدس خود را بررسی خواهید کرد. با این حال اگر در رسم درخت‌های بازگشتی و جمع هزینه‌ی سطوح بسیار دقیق باشید، می‌توانید مستقیماً از آن به عنوان اثبات جواب یک رابطه‌ی بازگشتی استفاده کنید. در این بخش از درخت‌های بازگشتی برای یافتن حدس‌های خوب استفاده می‌کنیم، و در بخش ۴-۶ با استفاده‌ی مستقیم از آن‌ها قضیه‌ای را به اثبات خواهیم رساند که پایه‌ی قضیه‌ی اصلی را تشکیل خواهد داد.

برای مثال اجازه دهید ببینیم چگونه به کمک یک درخت بازگشتی می‌توانیم حدسی خوب برای رابطه‌ی بازگشتی $T(n) = 3T(\lfloor n/4 \rfloor) + \theta(n^2)$ بیابیم. با تمرکز بر روی یافتن یک کران بالا برای جواب شروع می‌کنیم. از آن جایی که می‌دانیم کف‌ها و سقف‌ها معمولاً در جواب رابطه‌های بازگشتی تأثیری ندارند (این مثالی از بی‌دقتی است که می‌توانیم داشته باشیم)، یک درخت بازگشتی برای $T(n) = 3T(n/4) + cn^2$ می‌سازیم، که به طور غیر مستقیم می‌دانیم ثابت $c$ بزرگتر از صفر است.

شکل ۴-۵ تشکیل درخت بازگشتی از روی رابطه‌ی $T(n) = 3T(n/4) + cn^2$ را نشان می‌دهد. برای راحتی فرض می‌کنیم $n$ توانی از ۴ است (مثال دیگری از بی دقتی) تا تمام زیرمسئله‌ها عدد صحیح باشند. قسمت (الف) از شکل $T(n)$ را نشان می‌دهد، که در قسمت (ب) گسترش می‌یابد و به یک درخت معادل تبدیل می‌شود. این درخت نشان‌دهنده‌ی بازگشت است. جمله‌ی $cn^2$ در ریشه نشان‌دهنده‌ی هزینه در بالاترین سطح بازگشت است، و سه زیر درخت ریشه، هزینه‌ی زیرمسئله‌های ایجاد شده با اندازه‌ی $n/4$ را نشان می‌دهند. در قسمت (پ) این فرایند یک مرحله جلوتر می‌رود، و هر گره با هزینه‌ی $T(n/4)$ در قسمت (ب) گسترش می‌یابد. هزینه‌ی هر یک از سه فرزند ریشه برابر $c(n/4)^2$ است. همین طور با گسترش هر گره و شکستن آن به اجزای تشکیل‌دهنده در رابطه‌ی بازگشتی ادامه می‌دهیم.




**شکل ٤-٥** ساختار درخت بازگشتی برای رابطه‌ی $T(n) = 3T(n/4) + cn^2$. قسمت (الف) $T(n)$ را نشان می‌دهد، که مکررا در قسمت‌های (ب)-(ت) گسترش می‌یابد تا درخت بازگشتی تشکیل شود. درخت کاملا گسترش یافته در قسمت (ت) ارتفاعی برابر با $\log_4 n$ دارد (تعداد سطوح آن برابر است با $\log_4 n + 1$).

از آن جایی که به ازای هر مرحله‌ای که از ریشه دور می‌شویم، اندازه‌ی زیرمسئله‌ها با فاکتور ۴ کاهش می‌یابد، در نهایت باید به یک مقدار مرزی برسیم. در چه فاصله‌ای از ریشه به عدد یک می‌رسیم؟ اندازه‌ی یک زیرمسئله در عمق $i$ برابر است با $n/4^i$. بنابراین اندازه‌ی زیرمسئله‌ها زمانی به یک می‌رسد که داشته باشیم $n/4^i = 1$، یعنی وقتی که $i = \log_4 n$. در نتیجه درخت دارای $\log_4 n + 1$ سطح است (که این سطوح عبارتند از ۰، ۱، ۲، ...، $\log_4 n$).

سپس هزینه را در هر سطح درخت تعیین می‌کنیم. هر سطح سه برابر سطح بالایی گره دارد، و بنابراین تعداد گره‌ها در سطح $i$ام برابر است با $3^i$.

از آن جایی که اندازه‌ی زیرمسئله‌ها با هر سطح پایین رفتن با ضریب ۴ کاهش می‌یابد، هر گره در عمق $i$، برای $i = ۰,۱,۲,...,\log_۴ n - ۱$، هزینه‌ای برابر با $c(n/۴^i)^۲$ دارد. با ضرب هزینه‌ی گره‌ها در تعداد آن‌ها می‌بینیم که هزینه‌ی کلی در عمق $i$، برای $i = ۰,۱,۲,...,\log_۴ n - ۱$، برابر است با $۳^i c(n/۴^i)^۲ = (۳/۱۶)^i cn^۲$. آخرین سطح در عمق $\log_۴ n$ دارای $۳^{\log_۴ n} = n^{\log_۴ ۳}$ گره است، هر یک با هزینه‌ی $T(۱)$، که هزینه‌ی کلی $n^{\log_۴ ۳} T(۱)$ را به دست می‌دهد. این هزینه معادل همان $\theta(n^{\log_۴ ۳})$ است، چرا که فرض کردیم $T(۱)$ ثابت است.

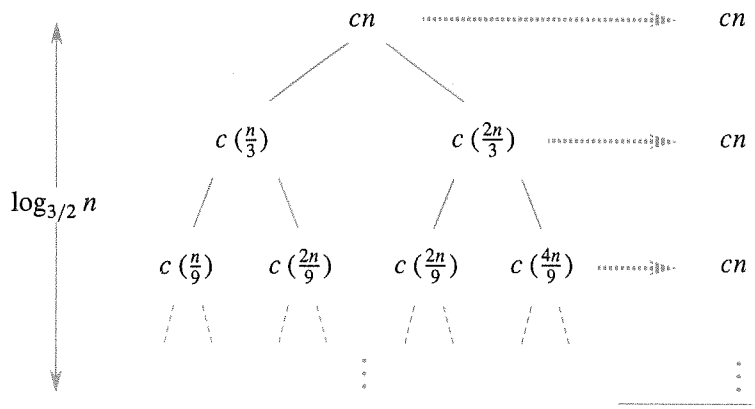
اکنون هزینه‌ی تمام سطوح را با هم جمع می‌کنیم تا هزینه‌ی کل درخت را به دست آوریم:

$$\begin{aligned} T(n) &= cn^۲ + \frac{۳}{۱۶}cn^۲ + \left(\frac{۳}{۱۶}\right)^۲ cn^۲ + ... + \left(\frac{۳}{۱۶}\right)^{\log_۴ n - ۱} cn^۲ + \theta\left(n^{\log_۴ ۳}\right) \\ &= \sum_{i=۰}^{\log_۴ n - ۱} \left(\frac{۳}{۱۶}\right)^i cn^۲ + \theta\left(n^{\log_۴ ۳}\right) \\ &= \frac{\left(\frac{۳}{۱۶}\right)^{\log_۴ n} - ۱}{\left(\frac{۳}{۱۶}\right) - ۱} cn^۲ + \theta\left(n^{\log_۴ ۳}\right), \qquad \text{(طبق تساوی (الف-۵))} \end{aligned}$$

عبارت آخر تا حدودی شلوغ به نظر می‌رسد. ولی متوجه می‌شویم که می‌توانیم از مقداری بی‌دقتی استفاده کرده و یک سری هندسی نزولی (همگرا) را به عنوان یک کران بالا تعریف کنیم. با یک مرحله بازگشت به عقب و استفاده از تساوی (الف-۶) داریم

$$\begin{aligned} T(n) &= \sum_{i=۰}^{\log_۴ n - ۱} \left(\frac{۳}{۱۶}\right)^i cn^۲ + \theta\left(n^{\log_۴ ۳}\right) \\ &< \sum_{i=۰}^{\infty} \left(\frac{۳}{۱۶}\right)^i cn^۲ + \theta\left(n^{\log_۴ ۳}\right) \\ &= \frac{۱}{۱ - \left(\frac{۳}{۱۶}\right)} cn^۲ + \theta\left(n^{\log_۴ ۳}\right) \\ &= \frac{۱۶}{۱۳} cn^۲ + \theta\left(n^{\log_۴ ۳}\right) \\ &= O(n^۲) \end{aligned}$$

بنابراین برای بازگشت اولیه‌ی $T(n) = ۳T(\lfloor n/۴ \rfloor) + \theta(n^۲)$، حدس $T(n) = O(n^۲)$ را تشکیل دادیم. در این مثال ضرایب $cn^۲$ یک سری هندسی نزولی را تشکیل می‌دهند، و طبق تساوی (الف-۶) مجموع این ضرایب از بالا کرانی برابر با $\frac{۱۶}{۱۳}$ دارد. سهم ریشه در کل هزینه برابر است با $cn^۲$، که ضریب ثابتی است از کل هزینه. بنابراین کل هزینه توسط هزینه‌ی ریشه تعیین می‌شود.




**شکل ۴-۶** یک درخت بازگشتی برای رابطه‌ی بازگشتی $T(n) = T(n/3) + T(2n/3) + cn$.

در واقع اگر $O(n^2)$ یک کران بالا برای رابطه‌ی بازگشتی باشد (که به زودی آن را بررسی می‌کنیم)، باید یک کران نزدیک باشد. چرا؟ اولین فراخوانی بازگشتی هزینه‌ای برابر با $\theta(n^2)$ دارد، و بنابراین باید $\Omega(n^2)$ یک کران پایین برای بازگشت باشد.

اکنون می‌توانیم با استفاده از روش جایگزینی درستی حدس خود را بررسی کنیم، یعنی این حدس که $T(n) = O(n^2)$ یک کران بالا است برای رابطه‌ی بازگشتی $T(n) = 3T(\lfloor n/4 \rfloor) + \theta(n^2)$. می‌خواهیم نشان دهیم که برای یک ثابت $d > 0$ داریم $T(n) \le dn^2$. با استفاده از همان ثابت $c > 0$ که قبلا از آن استفاده کردیم، 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 مرحله‌ی آخر در صورتی برقرار است که داشته باشیم $d \ge \left(\frac{16}{13}\right)c$.

به عنوان یک مثال پیچیده‌تر شکل ۴-۶ درخت بازگشتی $T(n) = T(n/3) + T(2n/3) + O(n)$ را نشان می‌دهد.

(دوباره، برای راحتی از کف‌ها و سقف‌ها صرف نظر می‌کنیم.) مانند قبل ضریب ثابت در جمله‌ی $O(n)$ را $c$ در نظر می‌گیریم. وقتی مقادیر را در عرض سطوح درخت بازگشتی با هم جمع می‌کنیم، در هر سطح مقدار $cn$ را خواهیم داشت. طولانی‌ترین مسیر از ریشه به یک برگ $n \to (2/3)n \to (2/3)^2 n \to \ldots \to 1$ است. از آن جایی که اگر $k = \log_{3/2} n$ داریم $(2/3)^k n = 1$،

ارتفاع درخت $\log_{۳/۲} n$ است.

به طور شهودی انتظار داریم جواب رابطه‌ی بازگشتی حداکثر برابر باشد با تعداد سطوح ضرب در هزینه‌ی هر سطح، یعنی $O(cn \log_{۳/۲} n) = O(n \lg n)$. ولی شکل ۴-۶ فقط یک سطح از درخت بازگشتی را نشان می‌دهد، و هزینه‌ی تمام سطوح درخت $cn$ نیست. هزینه‌ی برگ‌ها را در نظر بگیرید. اگر این درخت بازگشتی یک درخت دودویی کامل بود، $۲^{\log_{۳/۲} n} = n^{\log_{۳/۲} ۲}$ برگ داشت. از آن جایی که هزینه‌ی هر برگ ثابت است، هزینه‌ی کلی تمام برگ‌ها ($\theta(n^{\log_{۳/۲} ۲})$ خواهد بود، که چون $\log_{۳/۲} ۲$ ثابتی است بزرگ‌تر از ۱، از مرتبه‌ی $\omega(n \lg n)$ است. با این حال این درخت بازگشتی یک درخت دودویی کامل نیست، و بنابراین کم‌تر از $n^{\log_{۳/۲} ۲}$ برگ دارد. به علاوه همین طور که از ریشه به سمت پایین حرکت می‌کنیم گره‌های بیشتری حذف می‌شوند. در نتیجه هزینه‌ی سطوح پایین‌تر درخت بازگشتی کم‌تر از $cn$ است. می‌توانیم با تلاش بیشتر هزینه‌ها را با دقت بیشتری محاسبه کنیم، ولی به خاطر داشته باشید که در این جا فقط می‌خواهیم یک حدس بزنیم تا بتوانیم از آن در روش جایگزینی استفاده کنیم. اجازه دهید این مقدار بی‌دقتی را در نظر نگیریم و اثبات کنیم که حدس $O(n \lg n)$ برای یک کران بالا صحیح است.

در واقع می‌توانیم از روش جایگزینی استفاده کنیم و نشان دهیم که $O(n \lg n)$ یک کران بالا برای رابطه‌ی بازگشتی است. نشان می‌دهیم که $T(n) \le dn \lg n$، که در آن $d$ یک ثابت مثبت مفید است. داریم:

$$\begin{aligned}
T(n) &\le T(n/۳) + T(۲n/۳) + cn \\
&\le d(n/۳)\lg(n/۳) + d(۲n/۳)\lg(۲n/۳) + cn \\
&= \left(d(n/۳)\lg n - d(n/۳)\lg ۳\right) + \left(d(۲n/۳)\lg n - d(۲n/۳)\lg(۳/۲)\right) + cn \\
&= dn\lg n - d\left((n/۳)\lg ۳ + (۲n/۳)\lg(۳/۲)\right) + cn \\
&= dn\lg n - d\left((n/۳)\lg ۳ + (۲n/۳)\lg ۳ - (۲n/۳)\lg ۲\right) + cn \\
&= dn\lg n - dn\left(\lg ۳ - (۲/۳)\right) + cn \\
&\le dn\lg n
\end{aligned}$$

که رابطه‌ی آخر در صورتی صحیح است که داشته باشیم $d \ge c/(\lg ۳ - (۲/۳))$. بنابراین نیازی به دقت بیشتر در محاسبه‌ی هزینه‌های درخت بازگشتی نداریم.

## تمرین‌ها

۴-۴-۱ با استفاده از درخت‌های بازگشتی یک کران بالای حدی خوب برای رابطه‌ی $T(n) = ۳T(\lfloor n/۲ \rfloor) + n$ تعیین کنید. از متد جانشین‌سازی استفاده کرده و صحت جواب خود را بررسی کنید.

۴-۴-۲ با استفاده از درخت‌های بازگشتی یک کران بالای حدی خوب برای رابطه‌ی $T(n) = T(n/۲) + n^۲$ تعیین کنید. از متد جانشین‌سازی استفاده کرده و صحت جواب خود را بررسی کنید.

۴-۴-۴ با استفاده از درخت‌های بازگشتی یک کران بالای حدی خوب برای رابطه‌ی $T(n) = 4T(n/2+2) + n$ تعیین کنید. از متد جانشین‌سازی استفاده کرده و صحت جواب خود را بررسی کنید.

۴-۴-۴ با استفاده از درخت‌های بازگشتی یک کران بالای حدی خوب برای رابطه‌ی $T(n) = 4T(n-1) + 1$ تعیین کنید. از متد جانشین‌سازی استفاده کرده و صحت جواب خود را بررسی کنید.

۴-۴-۵ با استفاده از درخت‌های بازگشتی یک کران بالای حدی خوب برای رابطه‌ی $T(n) = T(n-1) + T(n/2) + n$ تعیین کنید. از متد جانشین‌سازی استفاده کرده و صحت جواب خود را بررسی کنید.

۴-۴-۶ با استفاده از درخت‌های بازگشتی نشان دهید که جواب رابطه‌ی بازگشتی $T(n) = T(n/3) + T(2n/3) + cn$، که در آن $c$ یک ثابت است، برابر است با $\Omega(n \lg n)$.

۴-۴-۷ درخت بازگشتی رابطه‌ی $T(n) = 4T(\lfloor n/2 \rfloor) + cn$ را بکشید ( $c$ یک ثابت است) و سپس یک کران حدی نزدیک برای آن بیابید. به کمک روش جایگزینی صحت جواب خود را تأیید کنید.

۴-۴-۸ به کمک درخت‌های بازگشتی یک جواب حدی نزدیک برای رابطه‌ی بازگشتی $T(n) = T(n-a) + T(a) + cn$ بیابید، که در آن $a \geq 1$ و $c > 0$ ثابت هستند.

۴-۴-۹ با استفاده از درخت‌های بازگشتی یک جواب حدی نزدیک برای رابطه‌ی بازگشتی $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$ بیابید، که در آن $0 < \alpha < 1$ و $c > 0$ ثابت هستند.

## ۴-۵ قضیه‌ی اصلی برای حل رابطه‌های بازگشتی

قضیه‌ی اصلی یک روش مستقیم برای حل رابطه‌های بازگشتی به شکل

$$T(n) = aT(n/b) + f(n) \qquad (۴-۲۰)$$

فراهم می‌آورد، که در آن $a \geq 1$ و $b > 1$ ثابت هستند، و $f(n)$ یک تابع به صورت حدی مثبت. برای استفاده از قضیه‌ی اصلی باید سه حالت مختلف را به خاطر بسپارید، ولی به کمک آن می‌توانید جواب بسیاری از رابطه‌های بازگشتی را به راحتی، حتی بدون نیاز به کاغذ و قلم تعیین کنید.

رابطه‌ی بازگشتی (۴-۲۰)، زمان اجرای یک الگوریتم را مشخص می‌کند که مسئله‌ای با اندازه‌ی $n$ را به $a$ زیرمسئله، هر کدام با اندازه‌ی $n/b$ تقسیم می‌کند. $a$ زیرمسئله به صورت بازگشتی و هر کدام در زمان $T(n/b)$ حل می‌شوند. هزینه‌ی تقسیم مسئله و ترکیب زیرجواب مسئله‌ها برابر است با $f(n)$. به عنوان مثال در رابطه‌ی بازگشتی حاصل از الگوریتم استراسن، داریم $a = 7$، $b = 2$، و $f(n) = \theta(n^2)$.

به عنوان یک مسئله‌ی تکنیکی لازم به ذکر است که رابطه‌ی بازگشتی بالا در واقع خوش تعریف نیست، چرا که ممکن است $n/b$ اصلا عدد صحیح نباشد. ولی نکته‌ی مهم این است که جایگذاری هر یک از جمله‌های ضرایب $a$ با $T(n/b)$ یا $T(\lfloor n/b \rfloor)$، بر روی رفتار بازگشت تأثیری نخواهد گذاشت. (این مسئله را در بخش بعد اثبات می‌کنیم.) با این حال معمولاً ساده است که در نوشتن رابطه‌های بازگشتی مسائل تقسیم و حل بدین شکل، از نمادهای کف و سقف صرف نظر کنیم.

## قضیه‌ی اصلی

روش تعیین جواب روابط بازگشتی که در بالا گفته شد، از قضیه‌ی زیر استفاده می‌کند.

**قضیه‌ی ۴.۱ (قضیه‌ی اصلی)**

فرض کنید $a \geq 1$ و $b > 1$ ثابت باشند، $f(n)$ یک تابع، و $T(n)$ بر روی اعداد صحیح نامنفی به صورت زیر تعریف شده باشد:

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

که در آن فرض می‌کنیم معنی $n/b$، یکی از دو عبارت $\lfloor n/b \rfloor$، ویا $\lceil n/b \rceil$ باشد. در این صورت می‌توان کران حدی $T(n)$ را به صورت زیر تعیین کرد:

۱. اگر $f(n) = O(n^{\log_b a - \varepsilon})$ برای یک ثابت $\varepsilon > 0$، آن گاه $T(n) = \theta(n^{\log_b a})$.

۲. اگر $f(n) = O(n^{\log_b a})$ آن گاه $T(n) = \theta(n^{\log_b a} \lg n)$.

۳. اگر $f(n) = O(n^{\log_b a + \varepsilon})$ برای یک ثابت $\varepsilon > 0$، و اگر $af(n/b) < cf(n)$ برای یک ثابت $c < 1$ و تمام $n$های به اندازه‌ی کافی بزرگ، آن گاه $T(n) = \theta(f(n))$.

قبل از استفاده از قضیه‌ی اصلی در چند مثال، اجازه دهید سعی کنیم معنی آن را بهتر درک کنیم. در هر یک از سه حالت، تابع $f(n)$ را با $n^{\log_b a}$ مقایسه کردیم. به طور شهودی جواب رابطه‌ی بازگشتی توسط تابع بزرگ‌تر بین دو تابع تعیین می‌شود. اگر مانند حالت اول تابع $n^{\log_b a}$ بزرگ‌تر باشد، آن گاه جواب برابر است با $T(n) = \theta(n^{\log_b a})$. اگر مانند حالت سوم تابع $f(n)$ بزرگ‌تر باشد، آن گاه جواب برابر است با $T(n) = \theta(f(n))$. اگر مانند حالت دوم دو تابع اندازه‌ی یکسانی داشته باشند، آن را در یک عامل لگاریتمی ضرب می‌کنیم، و جواب برابر است با $T(n) = \theta(n^{\log_b a} \lg n) = \theta(f(n) \lg n)$.

فراتر از این شهود تعدادی نکته‌ی فنی وجود دارد که باید آن‌ها را درک کرد. در حالت اول $f(n)$ نه تنها باید از $n^{\log_b a}$ کوچک‌تر باشد، بلکه باید به صورت *چندجمله‌ای* کوچک‌تر باشد. یعنی $f(n)$ باید به صورت حدی، و با یک ضریب $n^{\varepsilon}$ کوچک‌تر از $n^{\log_b a}$ باشد، که در آن $\varepsilon > 0$ یک ثابت است. در حالت سوم تابع $f(n)$ نه تنها باید بزرگ‌تر از $n^{\log_b a}$ باشد، بلکه باید به صورت چندجلمه‌ای بزرگ‌تر باشد، و به علاوه باید شرط $af(n/b) \leq cf(n)$ را ارضا کند. این شرط برای اکثر توابعی که کران چندجلمه‌ای دارند و ما به آن‌ها بر می‌خوریم، برقرار است.

مهم است متوجه باشیم که این سه حالت تمام حالت‌های ممکن برای $f(n)$ را پوشش نمی‌دهند. یک شکاف بین حالت‌های ۱ و ۲ وجود دارد، و آن زمانی است که $f(n)$ کوچک‌تر از $n^{\log_b a}$ باشد، ولی نه به صورت چندجمله‌ای. متشابها یک شکاف هم بین حالت‌های ۲ و ۳ وجود دارد، و آن زمانی است که $f(n)$ بزرگ‌تر از $n^{\log_b a}$ باشد، ولی نه به صورت چندجمله‌ای. اگر $f(n)$ درون یکی از این شکاف‌ها قرار بگیرد، و یا شرط حالت سوم برقرار نباشد، نمی‌توان از قضیه‌ی اصلی برای حل بازگشت استفاده کرد.

## استفاده از قضیه‌ی اصلی

برای استفاده از قضیه‌ی اصلی می‌توانیم به سادگی (در صورت امکان) تعیین کنیم که کدام یک از حالت‌ها اتفاق افتاده است، وسپس جواب را بنویسیم.

به عنوان مثال اول رابطه‌ی زیر را در نظر بگیرید:

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

برای این رابطه‌ی بازگشتی داریم $a = 9$، $b = 3$، و $f(n) = n$، پس داریم $n^{\log_b a} = n^{\log_3 9} = \theta(n^2)$. از آن جایی که $f(n) = O(n^{\log_3 9 - \varepsilon})$، که در آن $\varepsilon = 1$، می‌توانیم حالت اول قضیه‌ی اصلی را به کار ببریم و نتیجه بگیریم که $T(n) = \theta(n^2)$.

اکنون رابطه‌ی

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

را در نظر بگیرید، که در آن $a = 1$، $b = \frac{2}{3}$، $f(n) = 1$، و $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. حالت دوم قابل استفاده است، چرا که $f(n) = \theta(n^{\log_b a}) = \theta(1)$، و جواب بازگشت برابر است با $T(n) = \theta(\lg n)$.

برای رابطه‌ی 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 آن جایی که $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$، که در آن $\varepsilon \approx 0.2$، در صورتی حالت سوم اتفاق افتاده است که بتوانیم نشان دهیم شرط حالت سوم برای $f(n)$ ارضا شده است. برای $n$های به اندازه‌ی کافی بزرگ برای $c = 3/4$ داریم $af(n/b) = 3(n/4)\lg(n/4) \le (3/4) n \lg n = cf(n)$. در نتیجه طبق حالت سوم، جواب رابطه‌ی بازگشتی برابر است با $T(n) = \theta(n \lg n)$.

قضیه‌ی اصلی در مورد رابطه‌ی بازگشتی

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

کاربرد ندارد، با این که به نظر می‌آید که این رابطه شکل مناسب را داشته باشد: $a = 2$، $b = 2$، $f(n) = n \lg n$، و $n^{\log_b a} = n$. ممکن است فکر کنید می‌توان حالت سوم را به کار برد، چرا که

$f(n) = n \lg n$ به صورت حدی بزرگ‌تر از $n$ است، ولی مشکل این است که به صورت *چندجمله‌ای* بزرگ‌تر نیست. نسبت $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ برای هر ثابت مثبت $\varepsilon$ به صورت حدی از $n^{\varepsilon}$ کوچک‌تر است، در نتیجه رابطه‌ی بازگشتی در شکاف بین حالت‌های ۲ و ۳ می‌افتد (جواب را در تمرین ۴-۶-۲ ببینید).

اجازه دهید از قضیه‌ی اصلی برای حل روابط بازگشتی که در بخش‌های ۴-۱ و ۴-۲ دیدیم، استفاده کنیم. بازگشت (۴-۷)،

$$T(n) = 2T(n/2) + \theta(n)$$

زمان اجرای الگوریتم‌های تقسیم و حل مسئله‌ی زیرآرایه‌ی بیشینه و مرتب‌سازی ادغامی را توصیف می‌کند. (مانند قبل از ذکر حالت پایه در بازگشت صرف نظر می‌کنیم.) در این جا داریم $a = 2$، $b = 2$، و $f(n) = \theta(n)$. بنابراین داریم $n^{\log_b a} = n^{\log_2 2} = n$. حالت ۲ کاربرد دارد، چرا که $f(n) = \theta(n)$ و جواب $T(n) = \theta(n \lg n)$ را داریم.

رابطه‌ی بازگشتی (۴-۱۷)،

$$T(n) = 8T(n/2) + \theta(n^2)$$

زمان اجرای اولین الگوریتم تقسیم و حلی را که برای ضرب ماتریس‌ها دیدیم، توصیف می‌کند. اکنون داریم $a = 8$، $b = 2$، و $f(n) = \theta(n^2)$، و داریم $n^{\log_b a} = n^{\log_2 8} = n^3$. چون $n^3$ به صورت چندجمله‌ای بزرگ‌تر از $\theta(n)$ است (یعنی $f(n) = O(n^{3-\varepsilon})$ برای $\varepsilon = 1$)، حالت ۱ کاربرد دارد، و $T(n) = \theta(n^3)$.

نهایتاً، بازگشت (۴-۱۸) را در نظر بگیرید،

$$T(n) = 7T(n/2) + \theta(n^2)$$

که زمان اجرای الگوریتم استراسن را توصیف می‌کند. در این جا داریم $a = 7$، $b = 2$، $f(n) = \theta(n^2)$، و $n^{\log_b a} = n^{\log_2 7}$. با بازنویسی $\log_2 7$ به صورت $\lg 7$ و به خاطر آوردن این که $2{,}80 < \lg 7 < 2{,}81$، می‌بینیم که $f(n) = O(n^{\lg 7 - \varepsilon})$ برای $\varepsilon = 0{,}8$. دوباره حالت ۱ کاربرد دارد، و جواب $T(n) = \theta(n^{\lg 7})$ را خواهیم داشت.

## تمرین‌ها

۴-۵-۱ با استفاده از قضیه‌ی اصلی، برای روابط بازگشتی زیر کران‌های حدی نزدیک بیابید.

I. $T(n) = 3T(n/4) + 1$.

II. $T(n) = 3T(n/4) + \sqrt{n}$.

III. $T(n) = 3T(n/4) + n$.

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

۴-۵-۲ پروفسور Caesar می‌خواهد یک الگوریتم برای ضرب ماتریس‌ها ارائه کند که به صورت

حدی از الگوریتم استراسن سریع‌تر باشد. الگوریتم او از رویکرد تقسیم و حل استفاده می‌کند، بدین صورت که هر ماتریس را به قطعه‌هایی با اندازه‌ی $n/4 \times n/4$ تقسیم می‌کند، طوری که مراحل تقسیم و ترکیب $\theta(n^2)$ زمان می‌برند. او می‌خواهد تعیین کند که با تولید چند زیرمسئله، الگوریتم حاصل از الگوریتم استراسن سریع‌تر خواهد بود. اگر این الگوریتم $a$ زیرمسئله تولید کند، آن گاه رابطه‌ی بازگشتی زمان اجرا به صورت $T(n) = aT(n/4) + \theta(n^2)$ خواهد بود. بزرگ‌ترین مقدار برای $a$ به طوری که الگوریتم پروفسور Caesar از الگوریتم استراسن سریع‌تر باشد چقدر است؟

۴-۵-۳ با استفاده از قضیه‌ی اصلی نشان دهید که جواب رابطه‌ی بازگشتی $T(n) = T(n/2) + \theta(1)$ برای جستجوی دودویی برابر است با $T(n) = \theta(\lg n)$. (تمرین ۲-۳-۵ را برای توضیح جستجوی دودویی ببینید.)

۴-۵-۴ آیا می‌توان از قضیه‌ی اصلی برای تغیین جواب رابطه‌ی $T(n) = 4T(n/2) + n^2 \lg n$ استفاده کرد؟ دلیل جواب خود را بیان کنید. یک کران بالای حدی برای این بازگشت بدهید.

۴-۵-۵★ شرط $af(n/b) \le cf(n)$ را برای یک ثابت $c < 1$ در نظر بگیرید، که قسمتی از حالت ۳ قضیه‌ی اصلی است. یک مثال از ثابت‌های $a \ge 1$ و $b > 1$ و تابع $f(n)$ بدهید که تمام شرایط حالت ۳ را دارد، غیر از شرط بالا.

## ۴-۶★ اثبات قضیه‌ی اصلی

در این بخش اثباتی برای قضیه‌ی اصلی (قضیه‌ی ۴-۱) خواهیم دید. برای استفاده از قضیه، احتیاجی به درک اثبات آن نیست.

این اثبات دو بخش دارد. قسمت اول رابطه‌ی بازگشتی مربوط به قضیه‌ی اصلی (رابطه‌ی ۴-۲۰) را تحت شرایطی که $T(n)$ فقط روی توان‌های $b > 1$ تعریف شده است، تحلیل می‌کند، یعنی برای $n = 1, b, b^2, \ldots$. این قسمت تمام شهود لازم را برای درک دلیل درستی قضیه‌ی اصلی به دست می‌دهد. قسمت دوم نشان می‌دهد که چطور می‌توان این تحلیل را برای تمام مقادیر صحیح $n$ گسترش داد، و صرفاً شامل تکنیک‌های ریاضی برای رسیدگی کردن به کف‌ها و سقف‌ها است.

در این بخش بعضی مواقع از نمادهای حدی استفاده‌ی نادرست می‌کنیم، بدین صورت که آن‌ها را برای توضیح رفتار تابعی به کار می‌بریم که فقط بر روی توان‌های $b$ تعریف شده است. به خاطر بیاورید که در تعریف نمادهای حدی باید کران‌ها بر روی تمام $n$های به اندازه‌ی کافی بزرگ تعریف شده باشند، نه فقط آن‌هایی که توان $b$ هستند. از آن جایی که می‌توانیم نمادهای حدی جدیدی بسازیم که به جای اعداد صحیح، بر روی مجموعه‌ی $\{b^i : i = 0, 1, \ldots\}$ تعریف شده باشند، این استفاده‌ی نادرست چندان مهم نیست.

با این حال همیشه هنگام استفاده از نمادهای حدی روی مجموعه‌های محدود، باید مواظب باشیم که نتایج نادرست نگیریم. به عنوان مثال اثبات $T(n)=O(n)$، وقتی $n$ توانی از ۲ است، تضمین نمی‌کند که $T(n)=O(n)$. تابع $T(n)$ می‌تواند به صورت

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

تعریف شده باشد، که در آن بهترین کران بالایی که برای تمام مقادیر $n$ کاربرد دارد $T(n)=O(n^2)$ است. به خاطر این نتایج غیر قابل چشم پوشی هیچ‌گاه از نمادهای حدی برای مجموعه‌های محدود استفاده نخواهیم کرد، مگر این که صریحاً خلاف آن ذکر شود.

### ۴-۶-۱ اثبات برای توان‌های کامل

قسمت اول اثبات قضیه‌ی اصلی رابطه‌ی بازگشتی ۴-۲۰،

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

را تحلیل می‌کند، تحت این شرط که $n$ توانی از $b>1$ است (لازم نیست $b$ عدد صحیح باشد). این تحلیل شامل سه لم می‌شود. اولی مسئله را به ارزیابی عبارتی شامل یک سری، کاهش می‌دهد. لم دوم کران‌هایی برای این مجموع می‌یابد، و سومی، دو لم اول را با یکدیگر ادغام می‌کند تا قضیه‌ی اصلی را برای حالتی که $n$ توانی از $b$ است، اثبات کند.

**لم ۴-۲**

فرض کنید $a\geq 1$ و $b>1$ ثابت باشند، و $f(n)$ یک تابع نامنفی که روی توان‌های $b$ تعریف شده است. $T(n)$ را روی توان‌های $b$ به صورت رابطه‌ی بازگشتی زیر تعریف می‌کنیم:

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

که در آن $i$ یک عدد صحیح مثبت است. آن گاه

$$T(n)=\theta(n^{\log_b a})+\sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \qquad (۲۱-۴)$$

*اثبات* از درخت بازگشتی شکل ۴-۷ استفاده می‌کنیم. هزینه‌ی ریشه‌ی درخت هزینه‌ای $f(n)$ است و $a$ فرزند دارد، هر کدام با هزینه‌ی $f(n/b)$. (ساده‌تر است که $a$ را یک عدد صحیح فرض کنیم، مخصوصاً وقتی در مورد درخت بازگشتی صحبت می‌کنیم، ولی در واقع احتیاجی به این کار نیست.) هر کدام از این فرزندان، $a$ فرزند با هزینه‌ی $f(n/b^2)$ دارند، و بنابراین $a^2$ گره با فاصله‌ی ۲ از ریشه خواهیم داشت. در حالت کلی $a^j$ گره در فاصله‌ی $j$ از ریشه وجود دارد، که هزینه‌ی هر کدام

برابر است با $f(n/b^j)$. هزینه‌ی هر برگ برابر است با $T(1)=\theta(1)$، و هر گره در عمق $\log_b n$ قرار دارد، چرا که $n/b^{\log_b n}=1$. در کل $a^{\log_b n}=n^{\log_b a}$ برگ در درخت وجود دارد.

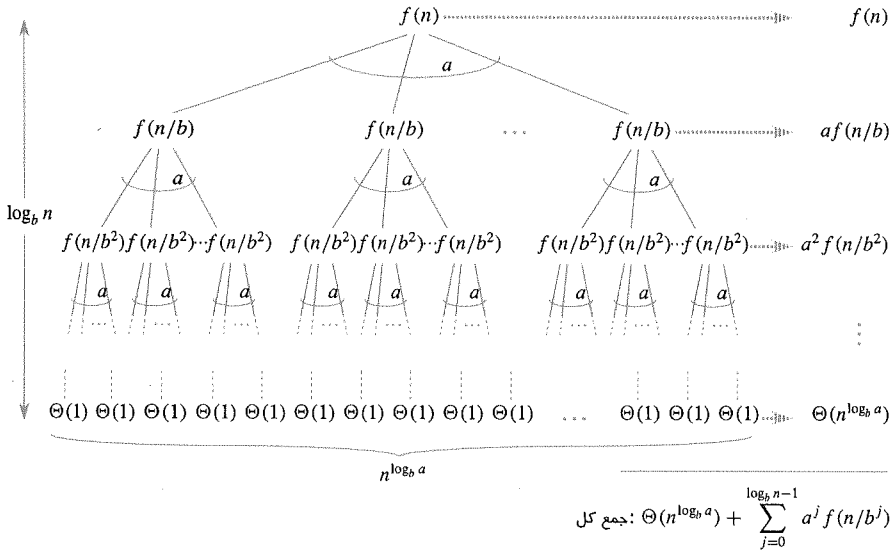
می‌توان با جمع هزینه‌ی تمام سطوح درخت، تساوی (۴-۲۱) را به دست آورد، همان طور که در شکل نشان داده شده است. هزینه‌ی سطح $j$ ام از گره‌های داخلی برابر است با $a^j f(n/b^j)$، و بنابراین جمع تمام سطوح داخلی برابر است با

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

در الگوریتم تقسیم و حل مربوط به این رابطه، این مجموع نشان دهنده‌ی مرحله‌ی تقسیم مسئله‌ها به زیرمسئله‌ها، و سپس ترکیب زیرمسئله‌ها است. هزینه‌ی تمام برگ‌ها، که همان هزینه‌ی حل تمام زیرمسئله‌های با اندازه‌ی ۱ است، برابر است با $\theta(n^{\log_b a})$.

در مورد درخت بازگشتی، سه حالت قضیه‌ی اصلی مربوط می‌شود به زمانی که تمام هزینه‌ی درخت (۱) از هزینه‌ی برگ‌ها ناشی می‌شود، (۲) به صورت مساوی در میان سطوح درخت پخش شده است، و یا (۳) از هزینه‌ی ریشه‌ی درخت ناشی می‌شود.

در تساوی (۴-۲۱) مجموع، نشان‌دهنده‌ی هزینه‌ی مرحله‌های تقسیم و ترکیب در الگوریتم تقسیم و حل مربوط به رابطه‌ی بازگشتی است. لم بعد کران‌های حدی برای رشد مجموع تعیین می‌کند.




**شکل ٤-٧** درخت بازگشتی ساخته شده توسط رابطه‌ی $T(n)=aT(n/b)+f(n)$. درخت شکل، یک درخت $a$تایی کامل با $n^{\log_b a}$ برگ و ارتفاع $\log_b n$ است. هزینه‌ی هر سطح در سمت راست آن، و جمع آن‌ها در تساوی (۴-۲۱) نشان داده شده است.

**لم ۴-۳**

فرض کنید $a \geq 1$ و $b > 1$ ثابت باشند و $f(n)$ یک تابع نامنفی که بر روی توان‌های $b$ تعریف شده‌است. تابع $g(n)$ را روی توان‌های $b$ به صورت زیر تعریف می‌کنیم:

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(n/b^j\right) \qquad (۲۲-۴)$$

برای مجموع بالا برای توان‌های $b$ می‌توان به صورت زیر کران تعریف کرد:

۱. اگر $f(n) = O(n^{\log_b a - \varepsilon})$ برای یک ثابت $\varepsilon$، آن گاه $g(n) = O(n^{\log_b a})$.

۲. اگر $f(n) = \theta(n^{\log_b a})$ آن گاه $g(n) = \theta(n^{\log_b a} \lg n)$.

۳. اگر $af(n/b) \leq cf(n)$ برای یک ثابت $c < 1$ و تمام $n$هایی که $n \geq b$، آن گاه $g(n) = \theta(f(n))$.

*اثبات* برای حالت ۱ داریم $f(n) = O(n^{\log_b a - \varepsilon})$، که $f(n/b^j) = O((n/b^j)^{\log_b a})$ را نتیجه می‌دهد. با جایگذاری در تساوی (۴-۲۲) خواهیم داشت

$$g(n) = O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b_j}\right)^{\log_b a - \varepsilon}\right) \qquad (۲۳-۴)$$

با فاکتورگیری از جمله‌ها و ساده‌سازی، یک کران با نماد $O$ برای مجموع تعریف می‌کنیم که یک سری هندسی صعودی بر جای می‌گذارد:

$$\begin{aligned}
\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \varepsilon} &= n^{\log_b a - \varepsilon} \sum_{j=0}^{\log_b n - 1} \left(\frac{ab^\varepsilon}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a - \varepsilon} \sum_{j=0}^{\log_b n - 1} \left(b^\varepsilon\right)^j \\
&= n^{\log_b a - \varepsilon} \left(\frac{b^{\varepsilon \log_b n} - 1}{b^\varepsilon - 1}\right) \\
&= n^{\log_b a - \varepsilon} \left(\frac{n^\varepsilon - 1}{b^\varepsilon - 1}\right).
\end{aligned}$$

از آن جایی که $b$ و $\varepsilon$ ثابت هستند، می‌توانیم عبارت آخر را به صورت $n^{\log_b a - \varepsilon} O(n^\varepsilon) = O(n^{\log_b a})$ بنویسیم. با جایگذاری این عبارت در مجموع تساوی (۴-۲۳) خواهیم داشت

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

و حالت اول اثبات شده است.

چون در حالت ۲ فرض می‌کنیم $f(n) = \theta(n^{\log_b a})$، داریم $f(n/b^j) = \theta((n/b^j)^{\log_b a})$. با جایگذاری در تساوی (۴-۲۲) خواهیم داشت

$$g(n)=\theta\left(\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a}\right) \qquad (۲۴-۴)$$

مانند حالت ۱ مجموع را با نماد $\theta$ محدود می‌کنیم، ولی این بار یک سری هندسی به دست نخواهیم آورد. در عوض به دست می‌آوریم که تمام جمله‌ها در مجموع یکسان هستند:

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

با جایگذاری این عبارت در مجموع تساوی (۴-۲۴) خواهیم داشت

$$\begin{aligned}g(n)&=\theta(n^{\log_b a}\log_b n)\\ &=\theta(n^{\log_b a}\lg n)\end{aligned}$$

و حالت ۲ اثبات شده است.

حالت ۳ را هم می‌توان به همین شکل اثبات کرد. از آن جایی که $f(n)$ در تعریف $g(n)$ (تساوی (۴-۲۲)) ظاهر شده است و تمام جمله‌های $g(n)$ نامنفی هستند، می‌توانیم نتیجه بگیریم که برای توان‌های $b$ داریم $g(n)=\Omega(f(n))$. در تعریف لم فرض می‌کنیم که برای یک ثابت $c<1$ و تمام $n$ های به اندازه‌ی کافی بزرگ داریم $af(n/b)\le cf(n)$. این فرض را به‌صورت $f(n/b)\le(c/a)f(n)$ بازنویسی، و $j$ بار تکرار می‌کنیم، که به دست می‌دهد $f(n/b^j)\le(c/a)^j f(n)$، یا به طور معادل $a^j f(n/b^j)\le c^j f(n)$. در این عبارت فرض می‌کنیم مقادیری که تکرار را بر روی آن‌ها انجام می‌دهیم به اندازه‌ی کافی بزرگ هستند. چون آخرین و کوچک‌ترین مقدار مانند این، $n/b^{j-1}$ است، کافی است که فرض کنیم $n/b^j$ به اندازه‌ی کافی بزرگ است.

با جایگذاری در تساوی (۴-۲۲) و ساده سازی، یک سری هندسی خواهیم داشت، ولی بر خلاف سری حالت ۱ جملات این سری نزولی هستند. برای پوشش جملاتی استفاده می‌کنیم که با فرض بزرگ بودن $n$ سازگار نیستند از یک جمله‌ی $O(1)$ استفاده می‌کنیم:

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

چرا که $c$ ثابت است. بنابراین می‌توانیم نتیجه بگیریم که برای توان‌های $b$ داریم $g(n) = \theta(f(n))$. حالت ۳ اثبات شده است، که اثبات لم را کامل می‌کند. ■

اکنون می‌توانیم نسخه‌ای از قضیه‌ی اصلی را که برای توان‌های $b$ است اثبات کنیم.

**لم ۴-۴**

فرض کنید $a \geq 1$ و $b > 1$ ثابت باشند، و $f(n)$ یک تابع نامنفی که روی توان‌های $b$ تعریف شده است. $T(n)$ را روی توان‌های $b$ به صورت رابطه‌ی بازگشتی زیر تعریف می‌کنیم:

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

که در آن $i$ یک عدد مثبت است. در این صورت می‌توان کران حدی $T(n)$ را بر روی توان‌های $b$ به صورت زیر تعریف کرد.

۱. اگر $f(n) = O(n^{\log_b a - \varepsilon})$ برای یک ثابت $\varepsilon > 0$، آن گاه $T(n) = \theta(n^{\log_b a})$.
۲. اگر $f(n) = \theta(n^{\log_b a})$، آن گاه $T(n) = \theta(n^{\log_b a} \lg n)$.
۳. اگر $f(n) = \Omega(n^{\log_b a + \varepsilon})$ برای یک ثابت $\varepsilon > 0$، و اگر $af(n/b) \leq cf(n)$ برای یک ثابت $c < 1$ و تمام $n$های به اندازه‌ی کافی بزرگ، آن گاه $T(n) = \theta(f(n))$.

*اثبات* از کران لم ۴-۳ برای ارزیابی مجموع (۴-۲۱) در 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،

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

و برای حالت ۳،

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

چرا که $f(n) = \Omega(n^{\log_b a + \varepsilon})$. ■

## ۴-۶-۲ کف‌ها و سقف‌ها

برای تکمیل قضیه‌ی اصلی باید تحلیل خود را به حالتی که از کف‌ها و سقف‌ها در رابطه‌ی بازگشتی قضیه‌ی اصلی استفاده می‌شود، گسترش دهیم تا بازگشت برای تمام اعداد صحیح تعریف شده باشد، نه فقط توان‌های $b$. تعیین یک کران پایین بر روی

$$T(n) = aT\left(\lceil n/b \rceil\right) + f(n) \qquad (۲۵-۴)$$

و یک کران بالا بر روی

$$T(n) = aT\left(\lceil n/b \rceil\right) + f(n) \qquad (۲۶-۴)$$

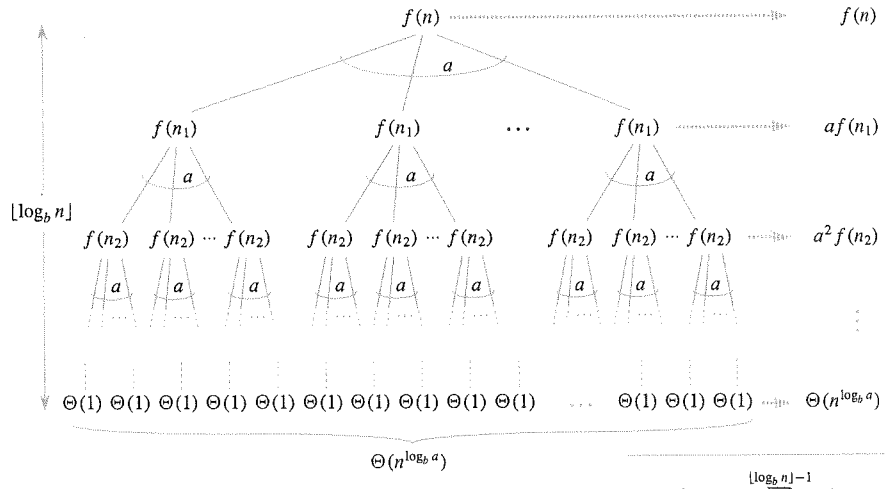
کار سرراستی است، چرا که می‌توان از نامساوی $\lceil n/b \rceil \geq n/b$ برای حالت اول و از نامساوی $\lfloor n/b \rfloor \leq n/b$ برای حالت دوم استفاده کرد. تعیین کران پایین برای بازگشت (۴-۲۵) تقریباً مانند تعیین کران بالا برای بازگشت (۴-۲۶) است، بنابراین در این جا فقط دومی را انجام می‌دهیم.

درخت بازگشتی شکل ۴-۷ را اصلاح می‌کنیم تا درخت شکل ۴-۸ به دست آید. همین‌طور که روی درخت به سمت پایین حرکت می‌کنیم، دنباله‌ای بازگشتی از فراخوانی روی آرگومان‌های زیر می‌بینیم:

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

اجازه دهید عنصر $j$ام در دنباله را با $n_j$ نشان دهیم، که

$$n_j = \begin{cases} n & \text{اگر } j = 0 \\ \lceil n_{j-1}/b \rceil & \text{اگر } j > 0 \end{cases} \qquad (۲۷-۴)$$




**شکل ۴-۸** درخت مربوط به رابطه‌ی بازگشتی $T(n) = aT\left(\lceil n/b \rceil\right) + f(n)$. آرگومان بازگشتی $n_j$ توسط رابطه‌ی (۴-۲۷) تعیین می‌شود.

اولین هدف این است که عمق $k$ را طوری بیابیم که $n_k$ ثابت باشد. با استفاده از نامساوی $x \le \lceil x \rceil < x + ۱$ به دست می‌آوریم

$$
\begin{aligned}
n_۰ &\le n,\\
n_۱ &\le \frac{n}{b} + ۱,\\
n_۲ &\le \frac{n}{b^۲} + \frac{۱}{b} + ۱,\\
n_۳ &\le \frac{n}{b^۳} + \frac{۱}{b^۲} + \frac{۱}{b} + ۱,\\
&\vdots
\end{aligned}
$$

به طور کلی،

$$
\begin{aligned}
n_j &\le \frac{n}{b^j} + \sum_{i=۰}^{j-۱} \frac{۱}{b^i}\\
&< \frac{n}{b^j} + \sum_{i=۰}^{\infty} \frac{۱}{b^i}\\
&= \frac{n}{b^j} + \frac{b}{b-۱}
\end{aligned}
$$

با قرار دادن $j = \lfloor \log_b n \rfloor$ خواهیم داشت

$$
\begin{aligned}
n_{\lfloor \log_b n \rfloor} &< \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-۱}\\
&< \frac{n}{b^{\log_b n - ۱}} + \frac{b}{b-۱}\\
&= \frac{n}{n/b} + \frac{b}{b-۱}\\
&= b + \frac{b}{b-۱}\\
&= O(۱)
\end{aligned}
$$

و بنابراین می‌بینیم که در عمق $\lfloor \log_b n \rfloor$ اندازه‌ی مسئله ثابت خواهد بود.
از شکل ۴-۴ داریم

$$T(n) = \theta(n^{\log_b a}) + \sum_{j=۰}^{\lfloor \log_b n \rfloor - ۱} a^j f(n_j) \qquad (۲۸-۴)$$

که تقریباً مشابه تساوی (۴-۲۱) است، با این تفاوت که در این جا $n$ یک عدد صحیح دلخواه است، نه توانی از $b$.
اکنون می‌توانیم مجموع زیر را ارزیابی کنیم:

$$g(n)=\sum_{j=0}^{\lfloor \log_b n \rfloor -1} a^j f(n_j) \qquad (۲۹-۴)$$

که از روی (۴-۲۸) و روشی مشابه لم ۴-۳ آن را به دست آورده‌ایم. با شروع از حالت ۳، اگر $af\left(\lceil n/b \rceil\right) \le cf(n)$ برای $n > b + b/(b-1)$، که در آن $c < 1$ یک ثابت است، آن گاه خواهیم داشت $a^j f(n_j) \le c^j f(n)$. بنابراین مجموع را در تساوی (۴-۲۹) می‌توانیم همانند لم ۴-۳ ارزیابی کنیم. برای حالت ۲، داریم $f(n) = \theta(n^{\log_b a})$. اگر بتوانیم نشان دهیم که $f(n_j) = O(n^{\log_b a}/a^j)$ آن گاه اثبات حالت ۲ از لم ۴-۳ به کمک ما خواهد آمد. مشاهده کنید که $O((n/b^j)^{\log_b a}) = O(n^{\log_b a})$ نتیجه می‌دهد $b^j/n \le 1$. کران $j \le \lfloor \log_b n \rfloor$ نتیجه می‌دهد که یک ثابت $c > 0$ وجود دارد به طوری که برای تمام $n_j$ها به اندازه کافی بزرگ،

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

چرا که $c(1+b/(b-1))^{\log_b a}$ ثابت است. بنابراین حالت ۲ اثبات شده است. اثبات حالت ۱ تقریباً به همین شکل است. نکته‌ی اصلی این است که کران $f(n_j) = O(n^{\log_b a - \varepsilon})$ را اثبات کنیم، که مشابه اثبات مربوطه در حالت ۲ است، هر چند جبر مورد نیاز پیچیده‌تر خواهد بود.

اکنون کران‌های بالا را برای تمام اعداد صحیح $n$ در قضیه‌ی اصلی اثبات کرده‌ایم. اثبات کران‌های پایین هم به طور مشابه انجام می‌شود.

## تمرین‌ها

★ ۴-۶-۱ یک توصیف دقیق و ساده برای $n_j$ در تساوی (۴-۲۷) بدهید، در حالتی که $b$ جای یک عدد حقیقی، یک عدد صحیح مثبت است.

★ ۴-۶-۲ نشان دهید که اگر $f(n) = \theta(n^{\log_b a} \lg^k n)$، که در آن $k \ge 0$، آن گاه رابطه‌ی بازگشتی قضیه‌ی اصلی جوابی به شکل $T(n) = \theta(n^{\log_b a} \lg^{k+1} n)$ دارد. برای سادگی جواب را برای توان‌های $b$ بیابید.

★ ۴-۶-۳ نشان دهید اگر شرط $af(n/b) \le cf(n)$ برای یک ثابت $c \le 1$ نتیجه دهد که یک ثابت $\varepsilon > 0$ وجود دارد به طوری که $f(n) = \Omega(n^{\log_b a+\varepsilon})$، آن گاه حالت ۳ قضیه‌ی اصلی اضافی است.

## مسائل

### ۴-۱ مثال‌هایی از رابطه‌های بازگشتی

در هر یک از بازگشت‌های زیر یک کران بالا و یک کران پایین برای $T(n)$ بیابید. فرض کنید $T(n)$ برای $n \le 2$ ثابت است. کران‌ها را تا حد ممکن نزدیک بیابید، و درستی جواب‌های خود را تأیید کنید.

I. $T(n) = 3T(n/2) + n^4$.
II. $T(n) = T(7n/10) + n$.
III. $T(n) = 16T(n/4) + n^2$.
IV. $T(n) = 7T(n/3) + n^2$.
V. $T(n) = 7T(n/2) + n^2$.
VI. $T(n) = 2T(n/4) + \sqrt{n}$.
VII. $T(n) = T(n-2) + n^2$.

### ۴-۲ هزینه‌ی ارسال پارامترها

در طول این کتاب فرض خواهیم کرد که ارسال پارامتر به رویه‌ها در زمان ثابت انجام می‌شود، حتی اگر یک آرایه‌ی $N$ عنصری را ارسال کنیم. در اکثر سیستم‌ها این فرض درست است، چرا که یک اشاره‌گر به آرایه ارسال می‌شود و نه خود آرایه. در این مسئله سه استراتژی ارسال پارامتر را بررسی می‌کنیم:

۱. آرایه توسط اشاره‌گر ارسال می‌شود. زمان $= \theta(1)$.
۲. یک کپی از آرایه ارسال می‌شود. زمان $= \theta(N)$، که در آن $N$ اندازه‌ی آرایه است.
۳. فقط قسمتی از آرایه ارسال می‌شود که ممکن است توسط رویه‌ی فراخوانی کننده مورد استفاده قرار گیرد. زمان $= \theta(q - p + 1)$، اگر زیرآرایه‌ی $A[p \,..\, q]$ ارسال شود.

I. الگوریتم جستجوی دودویی بازگشتی را برای یافتن یک عنصر خاص در آرایه‌ای مرتب در نظر بگیرید (تمرین ۲-۳-۵). روابط بازگشتی برای بدترین حالت زمان اجرای جستجوی دودویی در هر یک از روش‌های ارسال بالا بنویسید، و سپس کران‌های بالای مناسبی برای بازگشت‌های خود بیابید. فرض کنید $N$ اندازه‌ی مسئله‌ی اصلی و $n$ اندازه‌ی یک زیرمسئله است.
II. قسمت قبل را برای الگوریتم MERGE-SORT در بخش ۲-۳-۱ انجام دهید.

### ۴-۳ مثال‌های بیشتری از روابط بازگشتی

در هر یک از بازگشت‌های زیر کران‌های بالا و پایین مناسب برای $T(n)$ بیابید. فرض کنید $T(n)$ برای $n$های به اندازه‌ی کافی کوچک، ثابت است. تا حد ممکن کران‌های نزدیک بیابید، و درستی جواب خود را بررسی کنید.

I. $T(n) = ۴T(n/۳) + n \lg n$.

II. $T(n) = ۳T(n/۳) + n/\lg n$.

III. $T(n) + ۴T(n/۲) + n^۲\sqrt{n}$.

IV. $T(n) = ۳T(n/۳ - ۲) + n/۲$.

V. $T(n) = ۳T(n/۲) + n/\lg n$.

VI. $T(n) = T(n/۲) + T(n/۴) + T(n/۸) + n$.

VII. $T(n) = T(n-۱) + ۱/n$.

VIII. $T(n) = T(n-۱) + \lg n$.

IX. $T(n) = T(n-۲) + ۱/\lg n$.

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

### ۴-۴ اعداد فیبوناچی (Fibonacci numbers)

در این مسئله خصوصیات اعداد فیبوناچی را بررسی می‌کنیم. این اعداد به صورت رابطه‌ی بازگشتی (۳-۲۲) تعریف می‌شوند. از تکنیک توابع مولد (generating functions) برای حل رابطه‌ی بازگشتی اعداد فیبوناچی استفاده خواهیم کرد. تابع مولد (یا سری توانی رسمی) $\mathcal{F}$ را به صورت زیر تعریف کنید:

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=۰}^{\infty} F_i z^i \\ &= ۰ + z + z^۲ + ۲z^۳ + ۳z^۴ + ۵z^۵ + ۸z^۶ + ۱۳z^۷ + ۲۱z^۸ + \ldots\end{aligned}$$

که در آن $F_i$ نشان دهنده‌ی $i$امین عدد فیبوناچی است.

I. نشان دهید که $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^۲\mathcal{F}(z)$.

II. نشان دهید که

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{۱ - z - z^۲} \\ &= \frac{z}{(۱ - \phi z)(۱ - \hat{\phi} z)} \\ &= \frac{۱}{\sqrt{۵}}\left(\frac{۱}{۱ - \phi z} - \frac{۱}{۱ - \hat{\phi} z}\right)\end{aligned}$$

که در آن

$$\phi = \frac{۱+\sqrt{۵}}{۲} = ۱٫۶۱۸۰۳...$$

و

$$\hat{\phi} = \frac{۱-\sqrt{۵}}{۲} = -۰٫۶۱۸۰۳...$$

III. نشان دهید که

$$F(z) = \sum_{i=۰}^{\infty} \frac{۱}{\sqrt{۵}} \left(\phi^i - \hat{\phi}^i\right) z^i$$

IV. با استفاده از بخش III اثبات کنید که برای $i > ۰$ داریم $F_i = \frac{\phi^i}{\sqrt{۵}}$، که در آن سمت راست تساوی به نزدیک‌ترین عدد صحیح گرد شده است. (*راهنمایی:* توجه کنید که $|\hat{\phi}| < ۱$.)

## ۵-۴ *آزمایش تراشه‌های VLSI*

پروفسور Diogenes تعداد $n$ تراشه‌ی VLSI[۱] مشابه دارد که قادر به تست کردن یکدیگر هستند. برنامه‌ی آزمایش پروفسور در هر زمان دو تراشه را با هم آزمایش می‌کند. وقتی برنامه بارگذاری (load) شد هر تراشه، تراشه‌ی دیگر را آزمایش می‌کند و گزارش می‌دهد که آیا تراشه‌ی دیگر سالم است یا خراب. یک تراشه‌ی سالم همیشه جواب درستی در مورد تراشه‌ی دیگر می‌دهد، ولی نمی‌توان به جواب تراشه‌های خراب اطمینان کرد. بنابراین چهار خروجی ممکن یک آزمایش از قرار زیر هستند:

| نتیجه‌ی تراشه‌ی A | نتیجه‌ی تراشه‌ی B | نتیجه‌گیری کلی |
|---|---|---|
| B سالم است | A سالم است | هر دو سالم هستند، یا هر دو خراب هستند |
| B سالم است | A خراب است | حداقل یکی خراب است |
| B خراب است | A سالم است | حداقل یکی خراب است |
| B خراب است | A خراب است | حداقل یکی خراب است |

I. نشان دهید که اگر بیش از $n/۲$ تراشه‌ها خراب باشند، پروفسور با استفاده از هیچ استراتژی نمی‌تواند با اطمینان تشخیص دهد که کدام یک از تراشه‌ها سالم هستند. فرض کنید تراشه‌های بد می‌توانند برای گمراه کردن پروفسور توطئه بچینند!

II. مسئله‌ی یافتن یک تراشه‌ی سالم را در میان $n$ تراشه در حالتی در نظر بگیرید که بیش از $n/۲$ تراشه‌ها سالم هستند. نشان دهید که با $\lfloor n/۲ \rfloor$ آزمایش دوتایی می‌توان مسئله را به یک

[۱] VLSI مخفف «مدارهای مجتمع با اندازه‌ی بسیار بزرگ (very large scale integration)» است، که یک تکنولوژی مدارهای مجتمع است که امروزه از آن برای ساختن اکثر زیرپردازنده‌ها استفاده می‌شود.

مسئله‌ی مشابه با (تقریباً) نصف اندازه‌ی مسئله‌ی اصلی تبدیل کرد.

III. نشان دهید که می‌توان تراشه‌های سالم را با $\theta(n)$ آزمایش دوتایی شناسایی کرد، با فرض این که بیش از $n/2$ تراشه‌ها سالم هستند. روابط بازگشتی ارائه دهید که تعداد آزمایش‌های دوتایی لازم برای این کار را نشان می‌دهند، و سپس آن‌ها را حل کنید.

## ۴-۶ آرایه‌های Monge

یک آرایه‌ی $A$ با اندازه‌ی $m \times n$ از اعداد حقیقی، یک *آرایه‌ی Monge* است اگر برای تمام اعداد $i$، $j$، $k$ و $l$، که $1 \le i \le k \le m$ و $1 \le j < l \le n$، داشته باشیم:

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

یعنی هر جایی در آرایه که دو ردیف و دو ستون از یک آرایه‌ی Monge انتخاب کنیم و چهار عنصر درون تقاطع ردیف‌ها و ستون‌ها را در نظر بگیریم، مجموع عناصر بالا-چپ و پایین- راست کمتر یا مساوی مجموع عناصر پایین-چپ و بالا- راست است. برای مثال آرایه‌ی زیر Monge است:

| | | | | |
|---|---|---|---|---|
| ۱۰ | ۱۷ | ۱۳ | ۲۸ | ۲۳ |
| ۱۷ | ۲۲ | ۱۶ | ۲۹ | ۲۳ |
| ۲۴ | ۲۸ | ۲۲ | ۳۴ | ۲۴ |
| ۱۱ | ۱۳ | ۶ | ۱۷ | ۷ |
| ۴۵ | ۴۴ | ۳۲ | ۳۷ | ۲۳ |
| ۳۶ | ۳۳ | ۱۹ | ۲۱ | ۶ |
| ۷۵ | ۶۶ | ۵۱ | ۵۳ | ۳۴ |

I. اثبات کنید که یک آرایه Monge است اگر و تنها اگر برای هر $i = 1, 2, ..., m-1$ و $j = 1, 2, ..., n-1$ داشته باشیم:

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

(*راهنمایی:* برای بخش «تنها اگر»، از استقرا بر روی ردیف‌ها و ستون‌ها به صورت جداگانه استفاده کنید.)

II. آرایه‌ی زیر Monge نیست. با تغییر یکی از عناصر، آن را به یک آرایه‌ی Monge تبدیل کنید. (*راهنمایی:* از قسمت I استفاده کنید.)

| | | | |
|---|---|---|---|
| ۳۷ | ۲۳ | ۲۲ | ۳۲ |
| ۲۱ | ۶ | ۷ | ۱۰ |
| ۵۳ | ۳۴ | ۳۰ | ۳۱ |
| ۳۲ | ۱۳ | ۹ | ۶ |
| ۴۳ | ۲۱ | ۱۵ | ۸ |

III. فرض کنید $f(i)$ اندیس ستونی باشد که حاوی چپ‌ترین کمینه ردیف $i$ است (اگر دو

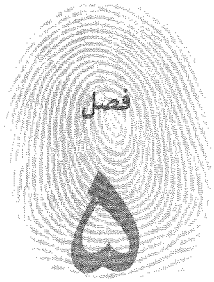
مقدار کمینه با هم مساوی باشند، چپ‌ترین کمینه، کمینه‌ای است که اندیس کم‌تری دارد–م).
اثبات کنید که برای هر آرایه‌ی Monge با ابعاد $m \times n$ داریم $f(1) \leq f(2) \leq \ldots \leq f(m)$.

IV. در زیر یک الگوریتم تقسیم و حل می‌بینیم که چپ‌ترین کمینه را در هر یک از ردیف‌های یک آرایه‌ی Monge با ابعاد $m \times n$ می‌یابد:

- یک زیر ماتریس $A'$ از روی $A$ بسازید که شامل ردیف‌های زوج $A$ باشد. به طور بازگشتی چپ‌ترین کمینه هر ردیف در $A'$ را بیابید. آن گاه چپ‌ترین کمینه را در ردیف‌های فرد $A$ بیابید.

توضیح دهید که (با فرض دانستن چپ‌ترین کمینه‌ها در ردیف‌های زوج) چطور می‌توان چپ‌ترین کمینه را در ردیف‌های فرد $A$ در زمان $O(m+n)$ پیدا کرد.

V. رابطه‌ی بازگشتی مربوط به زمان اجرای الگوریتم قسمت قبل را بنویسید. نشان دهید که جواب آن برابر است با $O(m + n \lg m)$.




# تحلیل احتمالاتی و الگوریتم‌های تصادفی

در این فصل در مورد تحلیل احتمالاتی و الگوریتم‌های تصادفی بحث خواهد شد. اگر با اصول نظریه‌ی احتمالات آشنا نیستید باید پیوست پ را بخوانید، که مروری است بر همین موضوع. در طول این کتاب بارها به تحلیل احتمالاتی و الگوریتم‌های تصادفی برخواهیم خورد.

## ۵-۱ مسئله‌ی استخدام

فرض کنید باید یک منشی جدید استخدام کنید. تلاش قبلی شما برای استخدام منشی مناسب ناموفق بوده، و تصمیم گرفته‌اید که از یک آژانس استخدام کمک بگیرید. آژانس استخدام روزانه یک متقاضی برای شما می‌فرستد. شما با آن شخص مصاحبه می‌کنید و تصمیم می‌گیرید که او را استخدام کنید یا نه. باید برای هر متقاضی مقداری حق‌الزحمه به آژانس پرداخت کنید. ولی استخدام یک متقاضی از آن هم پر هزینه‌تر است، چرا که باید ابتدا منشی فعلی خود را اخراج کنید، و مقدار زیادی حق‌الزحمه‌ی استخدام به آژانس بپردازید. مجبورید در هر زمان بهترین شخص ممکن را برای کار خود داشته باشید. بنابراین تصمیم می‌گیرید که بعد از مصاحبه با هر شخص، اگر آن شخص بهتر از منشی فعلی بود منشی خود را اخراج کرده و او را استخدام کنید. در این صورت باید هزینه‌ی مربوطه را به آژانس استخدام بپردازید، ولی می‌خواهید بدانید که این هزینه چقدر خواهد شد.

رویه‌ی HIRE-ASSISTANT این استراتژی را به صورت شبه‌کد نشان می‌دهد. این رویه فرض می‌کند که کاندیداهای شغل منشی‌گری به ترتیب از ۱ تا $n$ شماره گذاری شده‌اند. همچنین فرض می‌کند که بعد از مصاحبه با هر متقاضی، شما قادر هستید تشخیص دهید که آیا این متقاضی بهترین شخصی بوده که تا به حال برای کار خود دیده‌اید یا خیر. برای مقداردهی اولیه، رویه یک متقاضی مجازی با شماره‌ی ۰ می‌سازد که از همه برای شغل منشی‌گری بدتر است.

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

مدل هزینه در این مسئله با هزینه‌ای که در فصل ۲ توضیح دادیم تفاوت دارد. در این جا زمان اجرای رویه‌ی HIRE-ASSISTANT برای ما مهم نیست، بلکه هزینه‌ی مصاحبه و اخراج افراد است که اهمیت دارد. در ظاهر تحلیل هزینه‌ی این الگوریتم بسیار متفاوت است با مثلاً، تحلیل زمان اجرای مرتب‌سازی ادغامی. با این حال ابزارهای تحلیل ثابت هستند، چه بخواهیم هزینه‌ی این مسئله را محاسبه کنیم، و چه بخواهیم زمان اجرا را محاسبه کنیم. در هر دو حالت تعداد دفعات اجرای دستورات خاصی را خواهیم شمرد.

مصاحبه هزینه‌ی پایینی دارد، که ما آن را $c_i$ فرض می‌کنیم. ولی استخدام هزینه‌ی بالایی دارد، مثلاً $c_h$. فرض کنید $m$ تعداد افرادی باشد که استخدام شده‌اند. در این صورت کل هزینه‌ی الگوریتم برابر است با $O(nc_i + mc_h)$. مستقل از این که چند نفر را استخدام می‌کنیم، همیشه باید هزینه‌ی مصاحبه با $n$ متقاضی را پرداخت کنیم. بنابراین در این جا بر روی تحلیل هزینه‌ی $mc_h$ تمرکز می‌کنیم، یعنی هزینه‌ی استخدام‌ها. این مقدار برای اجراهای مختلف الگوریتم متفاوت است.

این سناریو به عنوان مدلی برای بسیاری از مثال‌های معمول محاسباتی عمل می‌کند. معمولاً مسئله بدین شکل است که می‌خواهیم یک مقدار بیشینه یا کمینه را در یک دنباله با آزمایش تک تک مقادیر، و تعیین یک «برنده» در هر لحظه بیابیم. به کمک مدل مسئله‌ی استخدام می‌توانیم دریابیم که باید برنده‌ی خود را هر چند وقت یک بار به روز رسانی می‌کنیم.

### تحلیل بدترین حالت زمان اجرا

در بدترین حالت مجبوریم که تمام متقاضیانی را که با آن‌ها مصاحبه می‌کنیم، استخدام کنیم. این حالت زمانی اتفاق می‌افتد که متقاضیان از نظر کیفیت به صورت صعودی مرتب شده باشند، که در این صورت باید $n$ نفر را استخدام کنیم، با هزینه‌ی استخدام $O(nc_h)$.

با این حال منطقی است که انتظار داشته باشیم متقاضیان همیشه بدین صورت مرتب نشده باشند. در واقع ما نه اطلاعاتی در مورد ترتیب آمدن متقاضیان داریم، و نه می‌توانیم کنترلی بر روی این ترتیب اعمال کنیم. بنابراین طبیعی است که بخواهیم بدانیم به طور معمول، و یا در حالت متوسط چه انتظاری باید داشت.

### تحلیل احتمالاتی

*تحلیل احتمالاتی* (probabilistic analysis) استفاده از احتمالات در تحلیل مسائل است. معمولاً از تحلیل احتمالاتی استفاده می‌کنیم تا زمان اجرای یک الگوریتم را تحلیل کنیم. ولی بعضی مواقع از آن برای اندازه‌گیری مقادیر دیگر بهره می‌گیریم، مثلاً هزینه‌ی استخدام در رویه‌ی HIRE-ASSISTANT . برای انجام یک تحلیل احتمالاتی، باید اطلاعاتی در مورد توزیع ورودی‌ها داشته

باشیم، یا حداقل فرضیاتی بر روی آن در نظر بگیریم. سپس الگوریتم خود را تحلیل می‌کنیم و یک زمان اجرا برای حالت متوسط به دست می‌آوریم، که در آن میانگین بر روی توزیع تمام ورودی‌های ممکن محاسبه می‌شود. در واقع داریم متوسط زمان اجرا را برای تمام ورودی‌های ممکن محاسبه می‌کنیم. این زمان اجرا را *زمان اجرای حالت متوسط* می‌نامیم.

هنگام تعیین توزیع ورودی‌ها باید دقت زیادی بکنیم. برای بعضی مسائل منطقی است که فرضی در مورد مجموعه‌ی تمام ورودی‌ها بکنیم، و آن گاه می‌توانیم برای درک مسئله یا طراحی یک الگوریتم کارامد از تکنیک‌های تحلیل احتمالاتی استفاده کنیم. برای دیگر مسائل نمی‌توان توصیفی منطقی برای توزیع ورودی‌ها به دست آورد، و بنابراین استفاده از تحلیل احتمالاتی برای آن‌ها شدنی نیست.

برای مسئله‌ی استخدام می‌توانیم فرض کنیم که متقاضیان با یک ترتیب تصادفی داده می‌شوند. معنی این جمله برای این مسئله چیست؟ ما فرض می‌کنیم که می‌توانیم هر دو متقاضی را با هم مقایسه کنیم و تعیین کنیم که کدام یک برای این شغل بهتر است: یعنی یک ترتیب عام بین متقاضیان وجود دارد. (اگر معنی ترتیب‌های عام را نمی‌دانید، به پیوست پ مراجعه کنید.) بنابراین می‌توانیم به هر متقاضی یک رتبه از ۱ تا $n$ بدهیم، که برای این کار، از $rank(i)$ برای نشان دادن رتبه‌ی متقاضی $i$ام استفاده خواهیم کرد، و همچنین قرارداد می‌کنیم که رتبه‌ی بالاتر نشان‌دهنده‌ی بهتر بودن یک متقاضی برای شغل موردنظر است. لیست مرتب $\langle rank(1), rank(2), \ldots, rank(n) \rangle$ یک جایگشت از لیست $\langle 1, 2, \ldots, n \rangle$ است. تصادفی بودن ترتیب متقاضیان، معادل این است که بگوییم احتمال این که هر یک از $n!$ جایگشت اعداد ۱ تا $n$ به عنوان لیست رتبه‌ها ظاهر شود، مساوی است. به طور معادل می‌توانیم بگوییم که رتبه‌ها یک *جایگشت تصادفی یکنواخت* را تشکیل می‌دهند: یعنی هر یک از $n!$ جایگشت ممکن با احتمال یکسان ظاهر می‌شوند.

بخش ۵-۲ شامل یک تحلیل احتمالاتی بر روی مسئله‌ی استخدام است.

## الگوریتم‌های تصادفی

برای استفاده از تحلیل احتمالاتی باید چیزی در مورد توزیع ورودی‌ها بدانیم. در بسیاری مواقع دانش ما در مورد توزیع ورودی‌ها بسیار اندک است. حتی اگر چیزی در مورد این توزیع بدانیم، ممکن است نتوانیم این دانش را به صورت محاسباتی مدل کنیم. با این حال معمولاً می‌توانیم با تصادفی کردن رفتار قسمتی از الگوریتم، از احتمالات به عنوان ابزاری برای طراحی و تحلیل الگوریتم‌ها استفاده کنیم.

در مسئله‌ی استخدام ممکن است به نظر برسد که متقاضیان با یک ترتیب تصادفی برای ما فرستاده می‌شوند، ولی هیچ راهی برای فهمیدن درستی این مسئله نداریم. بنابراین برای توسعه‌ی یک الگوریتم تصادفی برای این مسئله، باید کنترل بیشتری روی ترتیب مصاحبه با متقاضیان داشته باشیم. از این رو مدل را اندکی تغییر می‌دهیم. می‌گوییم آژانس استخدام لیستی از $n$ متقاضی دارد، و آن‌ها این لیست را پیشاپیش برای ما می‌فرستند. هر روز به صورت تصادفی یکی از متقاضیان را برای مصاحبه انتخاب می‌کنیم. با این که هیچ اطلاعاتی در مورد متقاضیان (غیر از نام آن‌ها) نداریم، ولی تفاوت بزرگی ایجاد کرده‌ایم. به جای تکیه بر یک حدس که متقاضیان با یک ترتیب تصادفی برای مصاحبه نزد ما می‌آیند، عملاً کنترل این ترتیب را به دست گرفته‌ایم و یک ترتیب تصادفی را بر آن‌ها تحمیل کرده‌ایم.

به طور کلی‌تر به یک الگوریتم *تصادفی* می‌گوییم اگر رفتار آن علاوه بر مقدار ورودی‌ها، به مقادیر تولید شده توسط یک *تولید کننده‌ی اعداد تصادفی* (rabdin-number generator) بستگی داشته باشد. فرض خواهیم کرد که یک رویه‌ی تولید کننده‌ی اعداد تصادفی با نام RANDOM در دسترس داریم. یک فراخوانی از رویه‌ی RANDOM$(a,b)$ یک عدد تصادفی بین $a$ و $b$ شامل خود $a$ و $b$ بازمی‌گرداند، که احتمال بروز هر یک از اعداد برابر است. به عنوان مثال، RANDOM$(۰,۱)$ با احتمال ۱/۲ عدد ۰ و با احتمال ۱/۲ عدد ۱ را بازمی‌گرداند. یک فراخوانی به صورت RANDOM$(۳,۷)$، یکی از اعداد۳، ۴، ۵، ۶ یا ۷ را بازمی‌گرداند، هر یک با احتمال ۱/۵. هر عدد تولید شده توسط RANDOM مستقل از اعداد تولید شده در فراخوانی‌های قبلی است. می‌توانید فراخوانی رویه‌ی RANDOM را مانند پرتاب یک تاس $(b-a+۱)$ وجهی فرض کنید. (در عمل، اکثر محیط‌های برنامه‌نویسی از یک *تولیدکننده‌ی اعداد شبه تصادفی* (pseudorandom-number generator) استفاده می‌کنند: یک الگوریتم قطعی (deterministic) که اعدادی تولید می‌کند که تصادفی به نظر می‌رسند.)

هنگام تحلیل زمان اجرای یک الگوریتم تصادفی، امیدریاضی زمان اجرا را بر روی توزیع مقادیر بازگردانده شده توسط تولید کننده‌ی اعداد تصادفی محاسبه می‌کنیم. برای تمییز میان این الگوریتم‌ها و آن‌هایی که ورودی تصادفی دارند، زمان اجرای یک الگوریتم تصادفی را *امیدریاضی زمان اجرا* (expected running time) می‌نامیم. به طور کلی زمان اجرای متوسط را وقتی به کار می‌بریم که توزیع احتمالاتی بر روی ورودی‌های الگوریتم اعمال می‌شود، و از امیدریاضی زمان اجرا برای مواقعی استفاده می‌کنیم که الگوریتم به خودی خود ورودی‌های تصادفی تولید می‌کند.

## تمرین‌ها

۵-۱-۱ نشان دهید که فرض توانایی تشخیص بهترین متقاضی در میان متقاضیان ملاقات شده (در خط ۴ رویه‌ی HIRE-ASSISTANT)، ایجاب می‌کند که یک ترتیب عام بر روی رتبه‌ی متقاضیان داشته باشیم.

۵-۱-۲ یک پیاده‌سازی از رویه‌ی RANDOM$(a,b)$ ارائه کنید که فقط با فراخوانی RANDOM $(۰,۱)$ کار می‌کند. امیدریاضی زمان اجرا برای این رویه به صورت تابعی از $a$ و $b$ چیست؟

۵-۱-۳ ★ فرض کنید می‌خواهیم هر یک از اعداد ۰ و ۱ را با احتمال ۱/۲ در خروجی نمایش دهیم. چیزی که در اختیار داریم، یک رویه‌ی BIASED-RANDOM است، که یکی از اعداد ۰ و ۱ را در خروجی نمایش می‌دهد. ولی احتمال چاپ ۱ در خروجی برابر $p$ و احتمال چاپ ۰ در خروجی $p-۱$ است، که در آن $۰ < p < ۱$. نکته این جا است که اطلاعی از مقدار $p$ نداریم. الگوریتمی ارائه کنید که با استفاده از رویه‌ی BIASED-RANDOM به عنوان یک زیرروال، با احتمال ۱/۲ مقدار ۰ و با احتمال ۱/۲ مقدار ۱ را به خروجی می‌دهد. امیدریاضی زمان اجرا به شکل تابعی از $p$ چیست؟

## ۵-۲ متغیرهای تصادفی شاخص

برای تحلیل بسیاری از الگوریتم‌ها، از جمله مسئله‌ی استخدام، از متغیرهای تصادفی شاخص استفاده خواهیم کرد. متغیرهای تصادفی شاخص روش ساده‌ای برای تبدیل احتمالات به امیدریاضی فراهم می‌کنند. فرض کنید یک فضای نمونه‌ی $S$ و یک پیشامد $A$ داریم. *متغیر تصادفی شاخص* $I\{A\}$ مربوط به پیشامد $A$ به صورت زیر تعریف می‌شود:

$$I\{A\} = \begin{cases} ۱ & \text{اگر } A \text{ رخ دهد} \\ ۰ & \text{اگر } A \text{ رخ ندهد} \end{cases} \qquad (۵-۱)$$

به عنوان یک مثال ساده، اجازه دهید امیدریاضی تعداد شیرها را در پرتاب یک سکه‌ی متقارن تعیین کنیم. فضای نمونه در این جا $S = \{H, T\}$ است و داریم $\Pr\{H\} = \Pr\{T\} = ۱/۲$. سپس می‌توانیم یک متغیر تصادفی شاخص $X_H$ تعریف کنیم متناظر با شیر آمدن سکه، که همان پیشامد $H$ است. این متغیر تعداد شیر آمدن‌ها را در پرتاب سکه نشان خواهد داد، و اگر سکه شیر بیاید ۱، و در غیر این صورت ۰ خواهد بود. می‌نویسیم:

$$X_H = I\{Y = H\} = \begin{cases} ۱ & \text{اگر } H \text{ رخ دهد} \\ ۰ & \text{اگر } T \text{ رخ دهد} \end{cases}$$

امیدریاضی تعداد شیر آمدن‌ها در یک بار پرتاب سکه به سادگی برابر است با امیدریاضی متغیر $X_H$:

$$\begin{aligned} E[X_H] &= E[I\{H\}] \\ &= ۱ \cdot \Pr\{H\} + ۰ \cdot \Pr\{T\} \\ &= ۱ \cdot (۱/۲) + ۰ \cdot (۱/۲) \\ &= ۱/۲ \end{aligned}$$

بنابراین امیدریاضی تعداد شیرها در یک بار پرتاب سکه‌ای متقارن برابر است با ۱/۲. همان طور که لم زیر نشان خواهد داد، امیدریاضی مقدار متغیر تصادفی شاخص مربوط به یک پیشامد $A$ برابر است با احتمال وقوع $A$.

**لم ۵-۱** یک فضای نمونه‌ی $S$ و یک پیشامد $A$ در این فضای نمونه داریم. اگر $X_A = I\{A\}$ آن گاه داریم $E[X_A] = \Pr\{A\}$.

*اثبات* طبق تعریف متغیرهای تصادفی شاخص در تساوی (۵-۱) و تعریف امیدریاضی، داریم:

$$\begin{aligned} E[X_A] &= E[I\{A\}] \\ &= ۱ \cdot \Pr\{A\} + ۰ \cdot \Pr\{\bar{A}\} \\ &= \Pr\{A\} \end{aligned}$$

که در آن $\bar{A}$ نشان‌دهنده‌ی $S - A$ است، یعنی مکمل $A$. ■

با این که ممکن است متغیرهای تصادفی شاخص برای کاربردی مانند شمردن تعداد شیرها در پرتاب فقط یک سکه نامناسب به نظر برسند، ولی برای تحلیل موقعیت‌هایی که چند آزمایش تصادفی را به صورت هم‌زمان انجام می‌دهیم، مناسب هستند. مثلاً متغیرهای تصادفی شاخص راهی در اختیار ما می‌گذارند که به راحتی به نتیجه‌ی تساوی (پ-۳۷) برسیم. در این تساوی تعداد شیرها را در پرتاب $n$ سکه می‌شماریم، بدین صورت که جداگانه احتمال آمدن ۰ شیر، ۱ شیر، ۲ شیر و ... را محاسبه می‌کنیم. با این حال روش ساده‌تر ارائه شده در تساوی (پ-۳۸)، در واقع به صورت غیر مستقیم از متغیرهای تصادفی شاخص استفاده می‌کند. به صورت واضح‌تر، فرض می‌کنیم $X_i$ متغیر تصادفی شاخص مربوط به پیشامد شیر آمدن در $i$امین پرتاب سکه باشد:

$X_i = I\{$نتیجه‌ی $i$امین آزمایش، $H$ باشد$\}$

می‌خواهیم امیدریاضی تعداد شیرها را محاسبه کنیم، بنابراین از دو طرف تساوی بالا امیدریاضی می‌گیریم:

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

سمت راست تساوی بالا امیدریاضی مجموع $n$ متغیر تصادفی است. طبق لم ۵-۱ می‌توانیم به سادگی امیدریاضی هر متغیر تصادفی را محاسبه کنیم. طبق تساوی (پ-۲۱) - خطی بودن امیدریاضی- محاسبه‌ی امیدریاضی مجموع ساده است: این امیدریاضی برابر است با مجموع امیدریاضی $n$ متغیر تصادفی. خطی بودن امیدریاضی، استفاده از متغیرهای تصادفی شاخص را تبدیل به یک تکنیک تحلیلی قدرتمند می‌کند؛ این تکنیک حتی زمانی که بین متغیرهای تصادفی وابستگی وجود دارد قابل استفاده است. اکنون به سادگی می‌توانیم امیدریاضی تعداد شیرها را محاسبه کنیم:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[X_i] \\ &= \sum_{i=1}^{n} \frac{1}{2} \\ &= \frac{n}{2} \end{aligned}$$

بنابراین در مقابل روش استفاده شده در تساوی (پ-۳۷)، متغیرهای تصادفی شاخص به شکل محسوسی محاسبات را ساده می‌کنند. در ادامه‌ی این کتاب از متغیرهای تصادفی شاخص به کرات استفاده خواهد شد.

## تحلیل مسئله‌ی استخدام با استفاده از متغیرهای تصادفی شاخص

اکنون به مسئله‌ی استخدام بازمی‌گردیم. می‌خواهیم امیدریاضی تعداد افرادی را که به عنوان منشی استخدام می‌کنیم به دست آوریم. برای استفاده از تحلیل احتمالاتی فرض می‌کنیم که متقاضیان با یک ترتیب تصادفی نزد ما خواهند آمد، همان طور که در بخش قبل بحث کردیم. (در بخش ۵-۳ خواهیم دید که چطور نیاز به این فرض را از بین ببریم.) فرض کنید $X$ متغیر تصادفی باشد که مقدار آن برابر است با تعداد دفعاتی که یک منشی جدید استخدام می‌کنیم. آن گاه می‌توانیم از تعریف امیدریاضی در تساوی (پ-۲۰) استفاده کنیم، که در این صورت داریم:

$$E[X] = \sum_{x=1}^{n} x \Pr\{X = x\}$$

ولی محاسبه‌ی این عبارت مشکل خواهد بود. در عوض می‌توانیم از متغیرهای تصادفی شاخص استفاده کنیم، که محاسبه را به شدت ساده خواهند کرد.

برای استفاده از متغیرهای تصادفی شاخص، به جای محاسبه‌ی $E[X]$ با تعریف یک متغیر برای تعداد دفعاتی که یک منشی جدید استخدام می‌کنیم، $n$ متغیر تعریف می‌کنیم که هر کدام نشان‌دهنده‌ی استخدام شدن یا نشدن یکی از متقاضیان است. به خصوص فرض می‌کنیم $X_i$ متغیر تصادفی شاخص مربوط به پیشامد استخدام شدن $i$امین متقاضی باشد. بنابراین

$$X_i = I\{\text{متقاضی } i \text{ استخدام شده است}\} = \begin{cases} 1 & \text{اگر متقاضی } i \text{ استخدام شود} \\ 0 & \text{اگر متقاضی } i \text{ استخدام نشود} \end{cases}$$

و همچنین

$$X = X_1 + X_2 + \ldots + X_n \qquad (۵-۲)$$

طبق لم ۵-۱ خواهیم داشت

$$E[X_i] = \Pr\{\text{متقاضی } i \text{ استخدام شود}\}$$

و بنابراین باید احتمال اجرای خطوط ۵-۶ رویه‌ی HIRE-ASSISTANT را محاسبه کنیم.

در خط ۵، متقاضی $i$ فقط زمانی استخدام می‌شود که از تمام متقاضیان ۱ تا $i-1$ بهتر باشد. از آن جایی که فرض کرده‌ایم متقاضیان با ترتیبی تصادفی برای مصاحبه می‌آیند، بنابراین $i$ متقاضی اول نیز با ترتیب تصادفی آمده‌اند. هر کدام از این $i$ متقاضی با احتمال برابر بهترین متقاضی (تا کنون) هستند. متقاضی $i$ام با احتمال $1/i$ بهترین متقاضی در میان متقاضیان ۱ تا $i-1$ است، و بنابراین با احتمال $1/i$ استخدام می‌شود. طبق لم ۵-۱ نتیجه می‌گیریم که

$$E[X_i] = \frac{1}{i} \qquad (۵-۳)$$

اکنون می‌توانیم $E[X]$ را محاسبه کنیم:

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] \quad \text{طبق تساوی (۵-۲)}$$

$$= \sum_{i=1}^{n} E[X_i] \quad \text{طبق خطی بودن امیدریاضی} \tag{۵-۴}$$

$$= \sum_{i=1}^{n} \frac{1}{i} \quad \text{طبق تساوی (۵-۳)}$$

$$= \ln n + O(1) \quad \text{طبق تساوی (الف-۷)} \tag{۵-۵}$$

با این که با $n$ متقاضی مصاحبه می‌کنیم، ولی به طور متوسط تقریباً $\ln n$ نفر از آن‌ها را استخدام می‌کنیم. این مسئله را در لم زیر جمع بندی می‌کنیم.

**لم ۵-۲** با فرض این که متقاضیان با یک ترتیب تصادفی ظاهر می‌شوند، هزینه‌ی کلی الگوریتم HIRE-ASSISTANT برابر است با $O(c_h \ln n)$.

*اثبات* این کران مستقیماً از تعریف هزینه‌ی مسئله‌ی استخدام و تساوی (۵-۵) نتیجه می‌شود، که نشان می‌دهد امیدریاضی تعداد استخدام‌ها تقریباً برابر است با $\ln n$. ■

هزینه‌ی متوسط استخدام بسیار کم‌تر از بدترین حالت هزینه‌ی استخدام، یعنی $O(nc_h)$ است.

## تمرین‌ها

۵-۲-۱ در HIRE-ASSISTANT با فرض این که متقاضیان با یک ترتیب تصادفی ظاهر می‌شوند، احتمال این که دقیقاً یک نفر را استخدام کنیم چقدر است؟ احتمال این که دقیقاً $n$ نفر را استخدام کنیم چقدر است؟

۵-۲-۲ در HIRE-ASSISTANT با فرض این که متقاضیان با یک ترتیب تصادفی ظاهر می‌شوند، احتمال این که دقیقاً دو نفر را استخدام کنیم چقدر است؟

۵-۲-۳ با استفاده از متغیرهای تصادفی شاخص، امیدریاضی مجموع $n$ تاس را محاسبه کنید.

۵-۲-۴ با استفاده از متغیرهای تصادفی شاخص، مسئله‌ی زیر را که به *مسئله‌ی اتاق لباس* معروف است حل کنید. هر کدام از $n$ مشتری یک کلاه به متصدی لباس‌ها در یک رستوران می‌دهند. متصدی لباس‌ها، کلاه‌ها را به صورت تصادفی به مشتریان بازمی‌گرداند. امیدریاضی تعداد مشتریانی که کلاه خود را پس می‌گیرند چقدر است؟

۵-۲-۵ فرض کنید $A[1..n]$ یک آرایه‌ی $n$ تایی از اعداد یکتا باشد. اگر $i < j$ و $A[i] > A[j]$، آن گاه جفت $(i, j)$ یک *وارونگی* (inversion) در آرایه‌ی $A$ خواهد بود. (برای توضیح بیشتر در مورد وارونگی‌ها مسئله‌ی ۲-۴ را ببینید.) فرض کنید هر یک از عناصر $A$ به

صورت تصادفی، مستقل و یکنواخت از بین عناصر ۱ تا $n$ انتخاب می‌شوند. با استفاده از متغیرهای تصادفی شاخص امیدریاضی تعداد وارونگی‌ها را بیابید.

## ۵-۳ الگوریتم‌های تصادفی

در بخش قبل نشان دادیم که اطلاع از توزیع ورودی‌ها چگونه می‌تواند در تحلیل رفتار حالت متوسط یک الگوریتم مفید باشد. در بسیاری مواقع چنین دانشی در مورد ورودی‌ها نخواهیم داشت، و در نتیجه تحلیل حالت متوسط امکان پذیر نخواهد بود. همان طور که در بخش ۵-۱ ذکر شد، در این موارد ممکن است بتوانیم از یک الگوریتم تصادفی استفاده کنیم.

برای مسئله‌ای مانند مسئله‌ی استخدام، که در آن فرض تساوی احتمال بروز هر یک از جایگشت‌ها کمک کننده است، تحلیل احتمالاتی راهنمایی خواهد بود به سمت توسعه‌ی یک الگوریتم تصادفی. به جای در نظر گرفتن توزیعی برای ورودی‌ها، یک توزیع به آن‌ها تحمیل می‌کنیم. به خصوص قبل از اجرای الگوریتم، به صورت تصادفی متقاضیان را مرتب می‌کنیم تا این خصوصیت را که احتمال تمام جایگشت‌ها برابر باشد تحمیل کرده باشیم. این تغییر انتظار استخدام (تقریباً) $\ln n$ منشی جدید را تغییر نمی‌دهد. در واقع این یعنی برای هر ورودی که به الگوریتم می‌دهیم (نه فقط آن‌هایی که توزیعشان را خود تعیین می‌کنیم) امیدریاضی استخدام‌ها $\ln n$ خواهد بود.

اکنون اندکی بیشتر تفاوت میان تحلیل احتمالاتی و الگوریتم‌های تصادفی را توضیح خواهیم داد. در بخش ۵-۲ ادعا کردیم که با فرض تصادفی بودن ترتیب متقاضیان با ترتیب، امیدریاضی تعداد منشی‌های استخدام شده تقریباً برابر $\ln n$ است. دقت داشته باشید که در این جا الگوریتم قطعی است: برای هر ورودی خاص، تعداد دفعاتی که یک منشی جدید استخدام می‌کنیم یکسان است. به علاوه تعداد استخدام‌ها برای ورودی‌های مختلف متفاوت است و به رتبه‌ی متقاضیان مختلف بستگی دارد. از آن جایی که این عدد فقط به رتبه‌ی متقاضیان بستگی دارد، می‌توانیم یک ورودی خاص را با لیست کردن رتبه‌ی متقاضیان به ترتیب نشان دهیم، مثلاً $\langle rank(1), rank(2), ..., rank(n)\rangle$. برای لیست رتبه‌ی $A_1 = \langle 1,2,3,4,5,6,7,8,9,10\rangle$، همیشه ۱۰ بار یک منشی جدید استخدام خواهد شد، چرا که هر متقاضی جدید از متقاضی قبلی بهتر است و خطوط ۵-۶ الگوریتم در هر تکرار حلقه اجرا خواهد شد. برای لیست رتبه‌ی $A_2 = \langle 10,9,8,7,6,5,4,3,2,1\rangle$ فقط یک بار و در اولین تکرار حلقه منشی جدید استخدام خواهد شد. همچنین برای لیست رتبه‌ی $A_3 = \langle 5,2,1,8,4,7,10,9,3,6\rangle$ سه استخدام خواهیم داشت (متقاضیان با رتبه‌های ۵، ۸ و ۱۰). با توجه به این که هزینه در این الگوریتم به تعداد استخدام‌ها بستگی دارد، می‌بینیم که ورودی‌هایی وجود دارند که پرهزینه هستند، مانند $A_1$، آن‌هایی که کم هزینه هستند، مانند $A_2$، و آن‌هایی که هزینه‌ی متوسطی دارند، مانند $A_3$.

از طرف دیگر الگوریتم تصادفی را در نظر بگیرید که ابتدا یک جایگشت به لیست متقاضیان می‌دهد، و سپس بهترین آن‌ها را برای استخدام در نظر می‌گیرد. در این حالت خاصیت تصادفی بودن

در الگوریتم است، نه در ورودی. با داشتن یک ورودی خاص مانند $A_3$ در بالا، نمی‌توانیم بگوییم که چند بار مقدار بیشینه به روز رسانی خواهد شد، چرا که این کمیت برای اجراهای مختلف الگوریتم متفاوت است. اولین باری که الگوریتم را بر روی $A_3$ اجرا می‌کنیم، ممکن است الگوریتم از روی آن جایگشت $A_1$ را بسازد و ۱۰ بار منشی جدید استخدام کند، در حالی که در اجرای دوم، ممکن است جایگشت $A_2$ ساخته شود و فقط یک استخدام داشته باشیم. برای اجرای سوم ممکن است تعداد متفاوتی استخدام داشته باشیم. هر بار که الگوریتم را اجرا می‌کنیم، نتیجه به یک انتخاب تصادفی وابسته است و ممکن است با اجرای قبلی متفاوت باشد. برای این الگوریتم و بسیاری از الگوریتم‌های تصادفی دیگر، *هیچ ورودی خاصی بدترین حالت هزینه را نتیجه نخواهد داد.* حتی بدترین دشمن شما هم نمی‌تواند یک آرایه‌ی ورودی تولید کند که مطمئن باشد هزینه‌ی بالایی دارد، چرا که جایگشت تصادفی ترتیب را در ورودی بی تأثیر خواهد کرد. یک الگوریتم تصادفی فقط در صورتی هزینه‌ی بالایی خواهد داشت که تولید کننده‌ی اعداد تصادفی، یک جایگشت «شوم» تولید کند!

برای مسئله‌ی استخدام، تنها تغییر مورد نیاز در کد این است که آرایه‌ی ورودی را به صورت تصادفی مرتب کنیم.

```
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      interview candidate i
5      if candidate i is better than candidate best
6          best = i
7          hire candidate i
```

با این تغییر ساده یک الگوریتم تصادفی ساخته‌ایم که کارایی آن برابر حالتی است که فرض کردیم متقاضیان به صورت تصادفی ارائه می‌شوند.

**لم ۵-۳** امیدریاضی هزینه‌ی استخدام در رویه‌ی RANDOMIZED-HIRE-ASSISTANT، برابر با $O(c_h \ln n)$ است.

*اثبات* پس از مرتب کردن آرایه‌ی ورودی، به موقیتی مانند تحلیل احتمالاتی HIRE-ASSISTANT دست یافته‌ایم. ■

مقایسه‌ی لم ۵-۲ با ۵-۳، تفاوت میان تحلیل احتمالاتی و الگوریتم‌های تصادفی را مشخص می‌کند. در لم ۵-۲ فرضی می‌کنیم در مورد ورودی، در حالی که در لم ۵-۳ چنین فرضی نخواهیم کرد، در عوض مقداری زمان اضافه صرف تصادفی کردن ورودی خواهد شد. برای تطابق با اصطلاحات استفاده شده تا کنون، لم ۵-۲ را در مورد هزینه‌ی متوسط استخدام و لم ۵-۳ را در مورد امیدریاضی هزینه‌ی استخدام تفسیر می‌کنیم. در ادامه‌ی این بخش در مورد بعضی مسائل که در تصادفی کردن ورودی‌ها وجود دارد، بحث خواهیم کرد.

## مرتب‌سازی تصادفی آرایه‌ها

بسیاری از الگوریتم‌های تصادفی ورودی را با جایگشت دادن به آرایه‌ی ورودی داده شده، تصادفی می‌کنند. (البته راه‌های دیگری هم برای این کار وجود دارد.) در این جا دو روش این کار را مورد بررسی قرار خواهیم داد. فرض می‌کنیم یک آرایه‌ی $A$ به ما داده شده است، که بدون از دست دادن کلیت مسئله، شامل عناصر ۱ تا $n$ است. هدف، ساختن یک جایگشت تصادفی بر روی این آرایه است.

یک روش معمول این است که به هر یک از عناصر $A[i]$ یک اولویت تصادفی $P[i]$ بدهیم و سپس عناصر $A$ را بر حسب این اولیت مرتب کنیم. مثلاً اگر ورودی اولیه‌ی ما آرایه‌ی $A = \langle ۱,۲,۳,۴ \rangle$ باشد و از اولویت‌های تصادفی $P = \langle ۳۶,۳,۹۷,۱۹ \rangle$ استفاده کنیم، آرایه‌ی $B = \langle ۲,۴,۱,۳ \rangle$ را به عنوان جایگشت تصادفی خواهیم ساخت، چرا که دومین اولویت کوچک‌ترین آن‌ها است، و بعد از آن چهارمی، سپس اولی، و در نهایت سومی. این رویه را PERMUTE-BY-SORTING می‌نامیم:

```
PERMUTE-BY-SORTING(A)
1  n = A.length
2  let P[1..n] be a new array
3  for i = 1 to n
4      P[i] = RANDOM(1,n^3)
5  sort A, using P as sort keys
```

خط ۴ یک عدد تصادفی بین ۱ و $n^3$ انتخاب می‌کند. از بازه‌ی ۱ تا $n^3$ استفاده می‌کنیم تا احتمال یکتا بودن اولویت‌ها را بیشتر کنیم. (تمرین ۵-۳-۵ از شما می‌خواهد اثبات کنید که یکتا بودن تمام اولویت‌ها حداقل $1/n$ است، و در تمرین ۵-۳-۶ باید الگوریتم را طوری پیاده‌سازی کنید که در صورت یکسان بودن دو یا چند اولویت، باز هم به درستی کار کند.) اجازه دهید فرض کنیم تمام اولویت‌ها یکتا هستند.

مرحله‌ی زمان‌بر در این رویه، مرحله‌ی مرتب‌سازی در خط ۵ است. همان طور که در فصل ۸ خواهیم دید اگر از یک الگوریتم مرتب‌سازی مقایسه‌ای استفاده کنیم، زمان مرتب‌سازی $\Omega(n \lg n)$ خواهد بود. قبلا با الگوریتم مرتب‌سازی ادغامی به این کران پایین رسیده‌ایم. (در قسمت دوم، الگوریتم‌های مرتب‌سازی دیگری خواهیم دید که زمان اجرای آن‌ها $\theta(n \lg n)$ است. تمرین ۸-۳-۴ از شما می‌خواهد مسئله‌ی بسیار مشابه مرتب‌سازی اعداد ۰ تا $n^3 - ۱$ را در زمان $O(n)$ حل کنید.) پس از مرتب‌سازی، اگر $P[i]$ برابر با $j$امین اولیت کوچک باشد، $A[i]$ در مکان $j$ در خروجی خواهد بود. بدین صورت به یک جایگشت دست خواهیم یافت. اثبات این که این رویه یک *جایگشت تصادفی یکنواخت* تولید می‌کند، یعنی احتمال تولید تمام جایگشت‌های اعداد ۱ تا $n$ با برابر است، باقی خواهد ماند.

**لم ۵-۴** با فرض این که تمام اولویت‌ها متفاوت هستند، رویه‌ی PERMUTE-BY-SORT یک جایگشت تصادفی یکنواخت تولید خواهد کرد.

*اثبات* با این حالت خاص شروع می‌کنیم که در آن هر عنصر $A[i]$، $i$امین اولویت کوچک را دریافت می‌کند. نشان خواهیم داد که این جایگشت دقیقاً با احتمال $1/n!$ تولید می‌شود. برای $i = 1,2,...,n$، فرض کنید $E_i$ پیشامد دریافت $i$امین اولویت کوچک توسط $A[i]$ باشد. آن گاه باید احتمال این را محاسبه کنیم که برای تمام $i$ها پیشامد $E_i$ اتفاق بیافتد، که به صورت زیر است:

$$\Pr\{E_1 \cap E_2 \cap E_3 \cap ... \cap E_{n-1} \cap E_n\}$$

با استفاده از تمرین پ-۲-۵، این احتمال برابر است با

$$\Pr\{E_1\} \cdot \Pr\{E_2 \mid E_1\} \cdot \Pr\{E_3 \mid E_2 \cap E_1\} \cdot \Pr\{E_4 \mid E_3 \cap E_2 \cap E_1\}$$
$$... \Pr\{E_i \mid E_{i-1} \cap E_{i-2} \cap ... \cap E_1\} ... \Pr\{E_n \mid E_{n-1} \cap ... \cap E_1\}$$

داریم $\Pr\{E_1\} = 1/n$، چرا که این همان احتمال انتخاب کوچک‌ترین اولویت از میان $n$ اولویت است. سپس مشاهده می‌کنیم که $\Pr\{E_2 \mid E_1\} = 1/(n-1)$، چرا که اگر $A[1]$ کوچک‌ترین اولویت باشد، هر یک از $n-1$ عنصر شانس برابری برای دریافت دومین اولویت کوچک دارند. به طور کلی برای $i = 2,3,...,n$ داریم $\Pr\{E_i \mid E_{i-1} \cap E_{i-2} \cap ... \cap E_1\} = 1/(n-i+1)$، زیرا اگر عناصر $A[1]$ تا $A[i-1]$، اولین $i-1$ اولویت کوچک را دریافت کنند (به ترتیب)، هر یک از $n-(i-1)$ عنصر باقی مانده شانس برابری برای دریافت $i$امین اولویت کوچک دارند. بنابراین داریم

$$\Pr\{E_1 \cap E_2 \cap E_3 \cap ... \cap E_{n-1} \cap E_n\} = \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right)...\left(\frac{1}{2}\right)\left(\frac{1}{1}\right)$$
$$= \frac{1}{n!}$$

و نشان داده‌ایم که احتمال تولید آن جایگشت خاص $1/n!$ است.

می‌توان این اثبات را برای هر جاگشت خاصی از اولویت‌ها گسترش داد. یک جایگشت ثابت $\sigma = \langle \sigma(1), \sigma(2), ..., \sigma(n) \rangle$ را از مجموعه‌ی $\{1,2,...,n\}$ در نظر بگیرید. اجازه دهید اولویت اختصاص داده شده به $A[i]$ را با $r_i$ نشان دهیم، که در آن عنصر رتبه‌ی $i$ با $j$امین اولویت کوچک $j$ است. اگر $E_i$ را این پیشامد تعریف کنیم که عنصر $A[i]$، $\sigma(i)$امین اولویت کوچک را دریافت کند، یا $r_i = \sigma(i)$، آن گاه اثبات مشابهی برای آن قابل استفاده است. بنابراین اگر احتمال هر جایگشت خاص را در نظر بگیریم محاسبه‌ی آن مانند محاسبه‌ی بالا خواهد بود، و در نتیجه احتمال رخ دادن آن جایگشت برابر خواهد بود با $1/n!$. ■

ممکن است فکر کنید برای اثبات این که یک جایگشت، تصادفی و یکنواخت است، کافی است نشان دهیم که احتمال قرار گرفتن عنصر $A[i]$ در مکان $j$ برابر است با $1/n$. تمرین ۵-۳-۴ نشان می‌دهد که در واقع این شرط ناکافی است.

روشی بهتر برای ساختن یک جایگشت تصادفی جابه‌جایی عناصر در محل است. رویه‌ی

RANDOMIZE-IN-PLACE این کار را در زمان $O(n)$ انجام می‌دهد. در تکرار $i$، عنصر $A[i]$ به صورت تصادفی از میان عناصر $A[i]$ تا $A[n]$ انتخاب می‌شود. بعد از تکرار $i$ام، $A[i]$ هیچ وقت تغییر نخواهد کرد.

```
RANDOMIZE-IN-PLACE(A)
1  n = A.length
2  for i = 1 to n
3      swap A[i] with A[RANDOM(i, n)]
```

در ادامه از یک ثابت حلقه استفاده خواهیم کرد تا نشان دهیم رویه‌ی RANDOMIZE-IN-PLACE یک جایگشت تصادفی یکنواخت تولید می‌کند. اگر یک مجموعه‌ی $n$ عنصری داشته باشیم، یک $k$-جایگشت از آن، یک دنباله‌ی $k$تایی از $n$ عنصر است. (پیوست ب را ببینید.) به طور کلی تعداد $k$-جایگشت‌های موجود در یک مجموعه‌ی $n$تایی برابر است با $n!/(n-k)!$.

**لم ۵-۵** رویه‌ی RANDIMIZE-IN-PLACE یک جایگشت تصادفی یکنواخت تولید می‌کند.

*اثبات* از ثابت حلقه‌ی زیر استفاده می‌کنیم:

- دقیقاً قبل از تکرار $i$ام حلقه در خطوط ۲-۳، برای هر $(i-1)$-جایگشت ممکن، زیرآرایه‌ی $A[1..i-1]$ با احتمال $(n-i+1)!/n!$ حاوی این $(i-1)$-جایگشت خواهد بود.

باید نشان دهیم که این ثابت حلقه قبل از اولین تکرار حلقه برقرار است، با هر تکرار حلقه برقرار می‌ماند، و این ثابت حلقه یک خصوصیت مفید برای اثبات درستی لم بالا پس از پایان حلقه به ما می‌دهد.

- آغاز: لحظه‌ی دقیقاً قبل از اولین تکرار حلقه را در نظر بگیرید، وقتی که $i=1$. ثابت حلقه می‌گوید که برای هر ۰-جایگشت ممکن، زیرآرایه‌ی $A[1..0]$ با احتمال $(n-i+1)!/n!$ شامل این ۰-جایگشت است. زیرآرایه‌ی $A[1..0]$ تهی است، و یک ۰-جایگشت شامل هیچ عنصری نیست. بنابراین $A[1..0]$ با احتمال ۱ ممکن است شامل هر یک از ۰-جایگشت‌ها باشد، و ثابت حلقه قبل از اولین تکرار حلقه برقرار است.
- ادامه: فرض می‌کنیم دقیقاً قبل از $(i-1)$امین تکرار، هر یک از $(i-1)$-جایگشت‌های ممکن با احتمال $(n-i+1)!/n!$ در زیرآرایه‌ی $A[1..i-1]$ قرار دارند، و اثبات می‌کنیم که بعد از تکرار $i$ام، احتمال وجود هر یک از $i$-جایگشت ممکن در زیرآرایه‌ی $A[1..i]$ برابر است با $(n-i)!/n!$. آن گاه افزایش $i$ در تکرار بعد ثابت حلقه را برقرار نگاه خواهد داشت.

اجازه دهید تکرار $i$ام را بررسی کنیم. یک $i$-جایگشت خاص را در نظر گرفته، عناصر درون آن را با $\langle x_1, x_2, ..., x_n \rangle$ نام گذاری کنید. این جایگشت شامل یک $(i-1)$-جایگشت است، به علاوه‌ی یک مقدار $x_i$ که الگوریتم آن را در $A[i]$ قرار می‌دهد. فرض کنید $E_1$ این پیشامد باشد که در $i-1$ تکرار اول، $(i-1)$-جایگشت $\langle x_1, ..., x_{i-1} \rangle$ در $A[1..i-1]$ قرار

گرفته باشند. طبق ثابت حلقه، $\Pr\{E_1\} = (n-i+1)!/n!$. همچنین فرض کنید $E_2$ پیشامد قرار گرفتن $x_i$ در مکان $A[i]$ در تکرار $i$ام باشد. $i-$جایگشت $\langle x_1,...,x_i \rangle$ دقیقاً زمانی در $A[1..i]$ قرار خواهد گرفت که هر دو پیشامد $E_1$ و $E_2$ اتفاق بیافتند، و بنابراین باید $\Pr\{E_2 \cap E_1\}$ را محاسبه کنیم. با استفاده از تساوی (پ-۱۴) خواهیم داشت:

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\}$$

احتمال $\Pr\{E_2 \mid E_1\}$ برابر است با $1/(n-i+1)$، چرا که در خط ۳ الگوریتم $x_i$ را به صورت تصادفی از میان تعداد $n-i+1$ عنصر در مکان‌های $A[i..n]$ انتخاب خواهد کرد. بنابراین داریم

$$\begin{aligned}\Pr\{E_1 \cap E_2\} &= \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \\ &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\ &= \frac{(n-i)!}{n!}\end{aligned}$$

• **پایان:** در انتها داریم $i = n+1$ و زیرآرایه‌ی $A[1..n]$ با احتمال $(n-(n+1)+1)!/n! = 0!/n! = 1/n!$ حاوی یک $n-$جایگشت خاص خواهد بود.

بنابراین RANDOMIZE-IN-PLACE یک جایگشت تصادفی یکنواخت تولید خواهد کرد. ■

در بسیاری مواقع، ساده‌ترین و کاراترین روش برای حل مسائل استفاده از یک الگوریتم تصادفی است. در ادامه‌ی این کتاب از الگوریتم‌های تصادفی به کرات استفاده خواهیم کرد.

## تمرین‌ها

۵-۳-۱ پروفسور Marceau با ثابت حلقه‌ی استفاده شده در اثبات لم ۵-۵ مخالف است. به نظر او این ثابت حلقه قبل از اولین تکرار حلقه برقرار نیست، چرا که می‌توان به راحتی ادعا کرد که یک زیرآرایه‌ی خالی حاوی هیچ ۰-جایگشتی نیست. بنابراین احتمال این که یک زیرآرایه‌ی خالی حاوی یک ۰-جایگشت باشد، ۰ است، و ثابت حلقه قبل از شروع حلقه درست نیست. رویه‌ی RANDOMIZE-IN-PLACE را طوری بازنویسی کنید که ثابت حلقه‌ی مربوطه قبل از اجرای اولین حلقه روی یک زیرآرایه‌ی ناتهی اعمال شود، و سپس اثبات لم ۵-۵ را برای رویه‌ی خود اصلاح کنید.

۵-۳-۲ پروفسور Kelp می‌خواهد رویه‌ای بنویسد که به صورت تصادفی، جایگشتی غیر از جایگشت همانی (جایگشت مرتب ۱ تا $n$) را تولید کند. او رویه‌ی زیر را برای این کار نوشته است:

PERMUTE-WITHOUT-IDENTITY($A$)

```
1  n = A.length
2  for i = 1 to n
3      swap A[i] with A[RANDOM(i + 1, n)]
```

آیا این رویه همان کاری را که پروفسور Kelp می‌خواهد انجام می‌دهد؟

۵-۳-۳ فرض کنید به جای جابه‌جا کردن عنصر $A[i]$ با یک عنصر تصادفی در زیرآرایه‌ی $A[i \mathinner{..} n]$ آن را با یک عنصر تصادفی در کل آرایه عوض کنیم:

```
PERMUTE-WITH-ALL(A)
1  n = A.length
2  for i = 1 to n
3      swap A[i] with A[RANDOM(1, n)]
```

آیا این کد یک جایگشت تصادفی یکواخت تولید می‌کند؟ جواب خود را توجیه کنید.

۵-۳-۴ پروفسور Armstrong رویه‌ی زیر را برای تولید یک جایگشت تصادفی یکنواخت ارائه کرده است:

```
PERMUTE-BY-CYCLIC(A)
1  n = A.length
2  let B[1..n] be a new array
3  offset = RANDOM(1, n)
4  for i = 1 to n
5      dest = i + offset
6      if dest > n
7          dest = dest − n
8      B[dest] = A[i]
9  return B
```

نشان دهید که هر عنصر $A[i]$ با احتمال $1/n$ در هر یک از مکان‌های $B$ قرار خواهد گرفت. سپس نشان دهید که پروفسور Armstrong اشتباه می‌کند، و جایگشت حاصل تصادفی یکنواخت نیست.

۵-۳-۵★ اثبات کنید که در آرایه‌ی $P$ در رویه‌ی PERMUTE-BY-SORTING، احتمال این که تمام عناصر یکتا باشند حداقل $1 - 1/n$ است.

۵-۳-۶ توضیح دهید چطور می‌توان الگوریتم PERMUTE-BY-SORTING را طوری بازنویسی کرد که در حالت‌هایی که دو یا چند اولویت یکسان هستند، به مشکل برنخورد. یعنی رویه‌ی شما باید حتی وقتی که دو یا چند عنصر یکسان هستند، یک جایگشت تصادفی یکنواخت تولید کند.

۵-۳-۶ فرض کنید می‌خواهیم یک نمونه‌ی تصادفی از مجموعه‌ی $\{1, 2, 3, \ldots, n\}$ بسازیم، یعنی یک زیرمجموعه‌ی $m$ تایی $S$، به طوری که $0 \le m \le n$، و احتمال تولید تمام زیرمجموعه‌های $m$ تایی با هم برابر باشد. یک روش بدین صورت است که قرار دهیم $A[i] = i$ برای

$n$، $i = ۱, ۲, ۳, ..., n$، رویه‌ی RANDOMIZE-IN-PLACE ($A$) را فراخوانی کرده، سپس به سادگی $m$ عنصر اول آرایه را انتخاب کنیم. این روش $n$ بار رویه‌ی RANDOM را فراخوانی می‌کند. اگر $n$ بسیار بزرگ‌تر از $m$ باشد، باید بتوانیم با فراخوانی‌های بسیار کم‌تری از RANDOM یک نمونه‌ی تصادفی بسازیم. نشان دهید که رویه‌ی بازگشتی زیر یک زیرمجموعه‌ی تصادفی $S$ از $\{1, 2, 3, ..., n\}$ با $m$ عنصر بازمی‌گرداند، که در آن احتمال رخ‌داد تمام زیرمجموعه‌های $m$تایی با هم برابر است، و رویه‌ی RANDOM فقط $m$ بار فراخوانی می‌شود:

```
RANDOM-SAMPLE(m, n)
1  if m == 0
2      return ϕ
3  else S = RANDOM-SAMPLE(m-1, n-1)
4      i = RANDOM(1, n)
5      if i ∈ S
6          S = S ∪ {n}
7      else S = S ∪ {i}
8      return S
```

## ۴-۵★ تحلیل احتمالاتی و استفاده‌های بیشتر متغیرهای تصادفی شاخص

در این بخش با ارائه‌ی مثال‌هایی تحلیل احتمالاتی را بیشتر شرح خواهیم داد. در مثال اول، احتمال یکسان بودن روز تولد افراد در یک اتاق $k$ نفری بررسی خواهد شد. مثال دوم، پرتاب تصادفی توپ‌ها در سبدها را آزمایش خواهد کرد. مثال سوم در مورد شیر آمدن‌های پی‌درپی در پرتاب یک سکه است. در مثال آخر نسخه‌ی دیگری از مسئله‌ی استخدام بررسی خواهد شد که در آن برای اتخاذ تصمیم نهایی، نیازی نیست که با تمامی متقاضیان مصاحبه کنیم.

### ۱-۴-۵ پارادوکس تولد

برای اولین مثال *پارادوکس تولد* را بررسی می‌کنیم. حداقل چند نفر باید در یک اتاق باشند که با احتمال ۵۰٪ دو نفر روز تولد یکسانی داشته باشند؟ جواب به طرز شگفت‌انگیزی کوچک است. پارادوکس در این‌جا است همان طور که در ادامه خواهیم دید، که این عدد بسیار کوچک‌تر از تعداد روزهای سال و یا حتی نصف تعداد روزهای سال است.

برای جواب دادن به این سؤال افراد حاضر در اتاق را با شماره‌های $1, 2, ..., k$ اندیس‌گذاری می‌کنیم، که در آن $k$ تعداد افراد اتاق است. با صرف نظر از سال‌های کبیسه فرض می‌کنیم تمام سال‌ها $n = ۳۶۵$ روز دارند. برای $i = 1, 2, ..., k$ فرض کنید $b_i$ روز تولد شخص $i$ام در اتاق باشد، که داریم $1 \le b_i \le n$. همچنین فرض می‌کنیم تولدها به صورت یکنواخت در بین تمام $n$ روز سال پخش شده‌اند، و بنابراین $\Pr\{b_i = r\} = 1/n$ برای $i = 1, 2, ..., k$ و $r = 1, 2, ..., n$.

احتمال این که دو شخص خاص، مثلاً $i$ و $j$ روز تولد یکسانی داشته باشند بستگی به این دارد که انتخاب تصادفی تولدها مستقل باشد یا خیر. از این به بعد فرض می‌کنیم تولدها مستقل هستند، و احتمال این که تولد $i$ام و $j$ام هر دو در روز $r$ باشند برابر است با:

$$\begin{aligned}\Pr\{b_i = r \text{ و } b_j = r\} &= \Pr\{b_i = r\}\Pr\{b_j = r\} \\ &= ۱/n^۲\end{aligned}$$

بنابراین احتمال این که تولد هر دو در یک روز باشد برابر است با:

$$\begin{aligned}\Pr\{b_i = b_j\} &= \sum_{r=۱}^{n} \Pr\{b_i = r \text{ و } b_j = r\} \\ &= \sum_{r=۱}^{n} \left(\frac{۱}{n^۲}\right) \\ &= ۱/n\end{aligned} \qquad (۶-۵)$$

برای درک شهودی بهتر می‌توانیم از این دید به مسئله نگاه کنیم: وقتی $b_i$ انتخاب شد، احتمال این که $b_j$ در همان روز به دنیا آمده باشد $۱/n$ است. پس احتمال این که $i$ و $j$ روز تولد یکسانی داشته باشند برابر است با احتمال این که تولد یکی از آن‌ها در یک روز خاص بیافتد. با این حال توجه داشته باشید که این تطبیق به فرض مستقل بودن تولدها بستگی دارد.

برای تحلیل احتمال این که روز تولد حداقل دو نفر یکسان باشد، می‌توانیم پیشامد مکمل آن را بررسی کنیم. احتمال این که حداقل تولد دو نفر در یک روز به دنیا آمده باشند برابر است با ۱ منهای احتمال این که تولد همه متفاوت باشد. احتمال این که $k$ نفر تولد متفاوت داشته باشند برابر است با:

$$B_k = \bigcap_{i=۱}^{k} A_i$$

که در آن $A_i$ پیشامد متفاوت بودن روز تولد شخص $i$ام و شخص $j$ام برای هر $j < i$ است. از آن جایی که می‌توانیم بنویسیم $B_k = A_k \cap B_{k-۱}$، از تساوی (پ-۱۶) رابطه‌ی بازگشتی زیر را خواهیم داشت:

$$\Pr\{B_k\} = \Pr\{b_{k-۱}\}\ \Pr\{A_k \mid B_{k-۱}\} \qquad (۷-۵)$$

که در آن $\Pr\{B_۱\} = \Pr\{A_۱\} = ۱$ را به عنوان شرط اولیه در نظر گرفته‌ایم. به عبارت دیگر احتمال این که روزهای تولد $b_۱, b_۲, ..., b_k$ متفاوت باشند برابر است با احتمال این که روزهای $b_۱, b_۲, ..., b_k$ متفاوت باشند ضرب در احتمال این که برای $i = ۱, ۲, ..., k-۱$ داشته باشیم $b_k \neq b_i$، با این شرط که اعداد $b_۱, b_۲, ..., b_{k-۱}$ متفاوت هستند.

اگر $b_۱, b_۲, ..., b_{k-۱}$ متفاوت باشند، احتمال شرطی $b_k \neq b_i$ برای $i = ۱, ۲, ..., k-۱$ برابر است با $\Pr\{A_k \mid B_{k-۱}\} = (n-k+۱)/n$، چرا که در $n$ روز، $n-(k-۱)$ روز وجود دارد که خالی است. با

اعمال بازگشتی رابطه‌ی (۵-۷) به دست می‌آوریم:

$$\begin{aligned}
\Pr\{B_k\} &= \Pr\{B_{k-1}\}\Pr\{A_k \mid B_{k-1}\} \\
&= \Pr\{B_{k-2}\}\Pr\{A_{k-1} \mid B_{k-2}\}\Pr\{A_k \mid B_{k-1}\} \\
&\vdots \\
&= \Pr\{B_1\}\Pr\{A_2 \mid B_1\}\Pr\{A_3 \mid B_2\}\ldots\Pr\{A_k \mid B_{k-1}\} \\
&= 1\cdot\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\ldots\left(\frac{n-k+1}{n}\right) \\
&= 1\cdot\left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\ldots\left(1-\frac{k-1}{n}\right)
\end{aligned}$$

طبق $1+x \le e_x$ (نامساوی (۳-۱۲)) خواهیم داشت:

$$\begin{aligned}
\Pr\{B_k\} &< e^{-1/n}e^{-2/n}\ldots e^{-(k-1)/n} \\
&= e^{-\sum_{i=1}^{k-1} i/n} \\
&= e^{-k(k-1)/2n} \\
&\le \frac{1}{2}
\end{aligned}$$

که در آن $-k(k-1)/2n \le \ln(1/2)$. وقتی داشته باشیم $k(k-1) \ge 2n\ln 2$ (یا به طور معادل، با حل معادله‌ی درجه‌ی دو، $k \ge (1+\sqrt{1+(8\ln 2)n})/2$) احتمال این که تمام $k$ تولد متفاوت باشند حداقل $1/2$ است. برای $n = 365$ باید داشته باشیم $k \ge 23$. بنابراین اگر حداقل ۲۳ نفر در یک اتاق باشند احتمال این که حداقل دو نفر روز تولد یکسان داشته باشند حداقل $1/2$ است. در مریخ هر سال ۶۶۹ روز مریخی است؛ بنابراین می‌توانیم با ۳۱ مریخی خصوصیت یکسانی داشته باشیم!

## تحلیل با استفاده از متغیرهای تصادفی شاخص

می‌توان با استفاده از متغیرهای تصادفی شاخص روشی ساده‌تر ولی تقریبی برای تحلیل پارادوکس تولد ارائه کرد. برای هر جفت $(i,j)$ از $k$ نفر در اتاق، متغیر تصادفی شاخص $X_{ij}$ را برای $1 \le i < j \le k$ بدین صورت تعریف می‌کنیم:

$$\begin{aligned}
X_{ij} &= I\{\text{شخص } i \text{ و شخص } j \text{ روز تولد یکسانی دارند}\} \\
&= \begin{cases} 1 & \text{اگر شخص } i \text{ و شخص } j \text{ روز تولد یکسانی دارند} \\ 0 & \text{در غیر اینصورت} \end{cases}
\end{aligned}$$

طبق تساوی (۵-۶) احتمال این که دو نفر روز تولد یکسانی داشته باشند برابر است با $1/n$، و بنابراین طبق لم ۵-۱ داریم:

$$\begin{aligned}
E[X_{ij}] &= \Pr\{\text{شخص } i \text{ و شخص } j \text{ روز تولد یکسانی دارند}\} \\
&= \frac{1}{n}
\end{aligned}$$

فرض می‌کنیم متغیر تصادفی $X$ تعداد جفت‌هایی را بشمارد که روز تولد یکسانی دارند. در این صورت خواهیم داشت:

$$X = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}$$

با امیدریاضی گرفتن از دو طرف (و توجه به این که امیدریاضی خطی است) به دست می‌آوریم:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}\right] \\ &= \sum_{i=1}^{k} \sum_{j=i+1}^{k} E[X_{ij}] \\ &= \binom{k}{2}\frac{1}{n} \\ &= \frac{k(k-1)}{2n} \end{aligned}$$

بنابراین وقتی $k(k-1) \geq 2n$ امیدریاضی تعداد جفت‌هایی از افراد که روز تولد یکسانی دارند حداقل ۱ است. از این رو اگر حداقل $\sqrt{2n}+1$ نفر در یک اتاق باشند می‌توانیم انتظار داشته باشیم که حداقل دو نفر روز تولد یکسانی دارند. برای $n = 365$، اگر $k = 28$، امیدریاضی تعداد جفت‌هایی که روز تولد یکسانی دارند برابر است با $\frac{(28\times27)}{(2\times365)} \approx 1.0356$.

پس با حداقل ۲۸ نفر می‌توانیم انتظار داشته باشیم که حداقل دو روز تولد یکسان باشند. در مریخ، که هر سال ۶۹۹ روز مریخی طول می‌کشد، حداقل به ۳۸ مریخی احتیاج داریم.

در تحلیل اول که در آن فقط از احتمالات استفاده شد، حداقل افرادی را محاسبه کردیم که نیاز داشتیم تا با احتمال حداقل ۱/۲ یک جفت از افراد روز تولد یکسانی داشته باشند. در تحلیل دوم که در آن از متغیرهای تصادفی شاخص استفاده شد، حداقل تعداد افرادی را به دست آوردیم که امیدریاضی را به ۱ برساند. با این که تعداد دقیق افراد در دو محاسبه متفاوت است، ولی از نظر حدی هر دو از یک مرتبه هستند: $\theta(\sqrt{n})$.

## ۵-۴-۲ توپ‌ها و سبدها

فرآیند پرتاب توپ‌هایی یکسان را در $b$ سبد مختلف با شماره‌های $1, 2, \ldots, b$ در نظر بگیرید. پرتاب‌ها مستقل هستند، و در هر پرتاب، احتمال قرار گرفتن توپ در تمام سبدها یکسان است. احتمال این که یک توپ در هریک از سبدها بیافتد برابر خواهد بود با $1/b$. پس فرآیند پرتاب توپ‌ها دنباله‌ای از آزمایش‌های برنولی است (پیوست پ-۴ رابببنید) که احتمال موفقیت در آن‌ها برابر است با $1/b$، و موفقیت یعنی افتادن توپ در یک سبد داده شده. این مدل به خصوص برای تحلیل توابع درهم‌ساز

مفید است (فصل ۱۱ را ببینید)، و به کمک آن می‌توان سؤالات بسیار جذابی را در مورد پرتاب توپ‌ها حل کرد. (در مسئله‌ی پ-۱ چند سؤال اضافی در مورد پرتاب توپ‌ها خواهیم دید.)

*چند توپ در یک سبد خاص خواهد افتاد؟* تعداد توپ‌هایی که در یک سبد خاص می‌افتند از توزیع دوجمله‌ای $b\left(k;n,1/b\right)$ پیروی می‌کند. اگر $n$ توپ را پرتاب کنیم، تساوی (پ-۳۷) می‌گوید که امیدریاضی تعداد توپ‌هایی که در یک سبد خاص می‌افتند برابر است با $n/b$.

*به طور متوسط چند توپ باید پرتاب شود تا در یک سبد خاص توپی قرار گیرد؟* تعداد پرتاب‌ها قبل از این که یک سبد خاص توپی دریافت کند از توزیع هندسی با احتمال $1/b$ پیروی می‌کند، و طبق تساوی (پ-۳۲) امیدریاضی تعداد پرتاب‌ها برای وجود توپ در یک سبد خاص برابر است با $1/(1/b)=b$.

*چند توپ باید پرتاب کرد تا هر سبد حداقل حاوی یک توپ باشد؟* اجازه دهید پرتابی را یک «موفقیت» بنامیم که در آن توپ به سبدی خالی هدایت می‌شود. می‌خواهیم امیدریاضی تعداد پرتاب‌های مورد نیاز برای $b$ موفقیت را محاسبه کنیم.

می‌توان فرض کرد که موفقیت‌ها، پرتاب‌ها را به مراحل مختلف تقسیم می‌کنند. مرحله‌ی $i$ام شامل پرتاب‌های میان موفقیت $(i-1)$ام و موفقیت $i$ام خواهد بود. کل مرحله‌ی اول عبارت است از پرتاب اول، چرا که وقتی تمام سبدها خالی هستند، مطمئناً پرتاب اول یک موفقیت خواهد بود. برای هر پرتاب در طول مرحله‌ی $i$ام، $i-1$ سبد خالی و $b-i+1$ سبد حاوی توپ داریم. پس برای هر پرتاب در طول مرحله‌ی $i$ام، احتمال بدست آوردن موفقیت برابر است با $(b-i+1)/b$.

فرض کنید $n_i$ تعداد پرتاب‌ها در مرحله‌ی $i$ام باشد. پس تعداد پرتاب‌های مورد نیاز برای بدست آوردن $b$ موفقیت عبارت است از $n=\sum_{i=1}^{b} n_i$. هر متغیر تصادفی $b_i$ دارای یک توزیع هندسی با احتمال موفقیت $(b-i+1)/b$ است، و طبق تساوی (پ-۳۲) داریم

$$E[n_i]=\frac{b}{b-i+1}$$

طبق خطی بودن امیدریاضی،

$$\begin{aligned}
E[n] &= E\left[\sum_{i=1}^{b} n_i\right] \\
&= \sum_{i=1}^{b} E[n_i] \\
&= \sum_{i=1}^{b} \frac{b}{b-i+1} \\
&= b\sum_{i=1}^{b} \frac{1}{i} \\
&= b\left(\ln b + O(1)\right) \qquad \text{(طبق تساوی (الف-۷))}
\end{aligned}$$

یعنی تقریباً با $b \ln b$ پرتاب می‌توان انتظار داشت که تمامی سبدها حاوی حداقل یک توپ باشند. این مسئله به *مسئله‌ی کلکسیونر کوپن* هم معروف است، که می‌گوید برای جمع‌آوری تمام $b$ کوپن مختلف تقریباً به $b \ln b$ کوپن تصادفی نیاز داریم.

## ۵-۴-۳ دوره‌ی پیروزی

فرض کنید یک سکه‌ی متقارن را $n$ بار پرتاب می‌کنیم. اگر آمدن شیر یک موفقیت باشد، چه انتظاری می‌توانیم در مورد طول بلندترین دوره‌ی موفقیت داشته باشیم؟ جواب، همان طور که در ادامه خواهیم دید، $\theta(\lg n)$ است.

ابتدا اثبات می‌کنیم که امیدریاضی طول بلندترین دوره‌ی موفقیت برابر است با $O(\lg n)$. احتمال این که هر پرتاب سکه شیر بیاید، ۱/۲ است. فرض کنید $A_{ik}$ پیشامد شروع یک موفقیت $k$تایی در پرتاب $i$ام باشد، یا به طور دقیق‌تر، پیشامد این که پرتاب‌های $i, i+1, \ldots, i+k-1$ شیر بیایند، که در آن $1 \le k \le n$ و $1 \le i \le n-k+1$. از آن جایی که پرتاب سکه‌ها از هم مستقل هستند، برای هر پیشامد $A_{ik}$ی خاص احتمال این که تمام $k$ بار پرتاب شیر بیایند عبارت است از:

$$\Pr\{A_{ik}\} = \frac{1}{2^k} \tag{۵-۸}$$

برای $k = 2\lceil \lg n \rceil$ داریم

$$\begin{aligned}
\Pr\left\{A_{i,2\lceil \lg n \rceil}\right\} &= \frac{1}{2^{2\lceil \lg n \rceil}} \\
&\le \frac{1}{2^{2\lg n}} \\
&= \frac{1}{n^2}
\end{aligned}$$

از این رو احتمال این که یک دوره‌ی موفقیت با طول حداقل $2\lceil \lg n \rceil$ در $i$امین پرتاب شروع شود، بسیار کم است. حداکثر $n - 2\lceil \lg n \rceil + 1$ مکان وجود دارند که می‌توانند نقطه‌ی آغازین این دوره‌ی موفقیت باشند. پس احتمال این که یک دوره‌ی موفقیت $2\lceil \lg n \rceil$تایی با هر پرتابی شروع شود عبارت است از:

$$\begin{aligned}
\Pr\left\{\bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil}\right\} &\le \sum_{i=1}^{n-2\lceil \lg n \rceil+1} \frac{1}{n^2} \\
&< \sum_{i=1}^{n} \frac{1}{n^2} \\
&= \frac{1}{n}
\end{aligned} \tag{۵-۹}$$

چرا که طبق نامساوی بول (پ-۱۹)، احتمال گروهی از پیشامدها حداکثر برابر جمع احتمال آن‌ها است. (توجه کنید که نامساوی بول حتی برای پیشامدهایی که مستقل نیستند – مانند همین مثال – هم صادق است.)

اکنون با استفاده از نامساوی (۵-۹) کرانی برای طول بلندترین دوره‌ی موفقیت می‌یابیم. فرض کنید برای $n$ ,..., ۲, ۱, ۰ = $j$ پیشامد $L_j$ نشان‌دهنده‌ی این باشد که طول بلندترین دوره‌ی موفقیت دقیقاً برابر باشد با $j$. همچنین فرض کنید $L$ طول بلندترین دوره‌ی موفقیت باشد. طبق تعریف امیدریاضی،

$$E[L] = \sum_{j=0}^{n} j \Pr\{L_j\} \qquad (۵-۱۰)$$

مانند محاسبه‌ی نامساوی (۵-۹)، می‌توانیم سعی کنیم با استفاده از کران‌های بالا بر روی هر یک از $\{L_j\}$ها این مجموع را ارزیابی کنیم. متأسفانه با این روش به کران‌های ضعیفی دست خواهیم یافت. روش بهتر برای یافتن کرانی خوب برای این سری، الهام از تحلیل بالا است. به صورت شهودی، مشاهده می‌کنیم که برای هیچ جمله‌ای در مجموع تساوی (۵-۱۰)، هر دو فاکتور $j$ و $\Pr\{L_j\}$ بزرگ نیستند. چرا؟ وقتی $j \geq ۲\lceil \lg n \rceil$ مقدار $\Pr\{L_j\}$ بسیار کوچک است، و وقتی $j < ۲\lceil \lg n \rceil$ آن گاه خود $j$ کوچک است. به صورت رسمی‌تر می‌بینیم که پیشامدهای $L_j$ برای $n$ ,..., ۱, ۰ = $j$ گسسته هستند، و بنابراین احتمال این که یک دوره‌ی موفقیت با طول حداقل $۲\lceil \lg n \rceil$ در هرجایی شروع شود برابر خواهد بود با $\sum_{j=۲\lceil \lg n \rceil}^{n} \Pr\{L_j\}$. طبق نامساوی (۵-۹) داریم $\sum_{j=۲\lceil \lg n \rceil}^{n} \Pr\{L_j\} < \frac{۱}{n}$. همچنین با توجه به این که $\sum_{j=۰}^{n} \Pr\{L_j\} = ۱$، داریم $\sum_{j=۰}^{۲\lceil \lg n \rceil - ۱} \Pr\{L_j\} \leq ۱$. از این رو به دست می‌آوریم:

$$\begin{aligned}
E[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{۲\lceil \lg n \rceil - ۱} j \Pr\{L_j\} + \sum_{j=۲\lceil \lg n \rceil}^{n} j \Pr\{L_j\} \\
&< \sum_{j=0}^{۲\lceil \lg n \rceil - ۱} \left(۲\lceil \lg n \rceil\right) \Pr\{L_j\} + \sum_{j=۲\lceil \lg n \rceil}^{n} n \Pr\{L_j\} \\
&= ۲\lceil \lg n \rceil \sum_{j=0}^{۲\lceil \lg n \rceil - ۱} \Pr\{L_j\} + n \sum_{j=۲\lceil \lg n \rceil}^{n} \Pr\{L_j\} \\
&< ۲\lceil \lg n \rceil \cdot ۱ + n \cdot \left(\frac{۱}{n}\right) \\
&= O\left(\lg n\right)
\end{aligned}$$

شانس این که دوره‌ای از موفقیت‌ها در پرتاب سکه از $r\lceil \lg n \rceil$ فراتر رود، با رشد $r$ به شدت

کاهش می‌یابد. برای $r \geq ۱$، احتمال این که یک دوره‌ی موفقیت با طول $r\lceil \lg n \rceil$ در مکان $i$ام شروع شود برابر است با:

$$\Pr\left\{A_{i,r\lceil \lg n\rceil}\right\} = \frac{۱}{۲^{r\lceil \lg n\rceil}} \leq \frac{۱}{n^r}$$

بنابراین احتمال این که طول بلندترین دوره‌ی موفقیت حداقل $r\lceil \lg n \rceil$ باشد حداکثر برابر است با $n/n^r = ۱/n^{r-۱}$. به طور معادل، حداقل با احتمال $۱-۱/n^{r-۱}$ طول بلندترین دوره‌ی موفقیت کم‌تر از $r\lceil \lg n \rceil$ است.

به عنوان مثال، برای $n = ۱۰۰۰$ پرتاب سکه احتمال داشتن یک دوره‌ی موفقیت با طول حداقل $۲\lceil \lg n \rceil = ۲۰$ حداکثر برابر است با $\frac{۱}{n} = \frac{۱}{۱۰۰۰}$، و احتمال داشتن یک دوره‌ی موفقیت با طول حداقل $۳\lceil \lg n \rceil = ۳۰$ حداکثر برابر است با $\frac{۱}{n^۲} = \frac{۱}{۱,۰۰۰,۰۰۰}$.

اکنون یک کران پایین مکمل خواهیم یافت: امیدریاضی طول بلندترین دوره‌ی موفقیت در پرتاب $n$ سکه $\Omega(\lg n)$ است. برای اثبات این کران نگاهی خواهیم کرد به دوره‌های موفقیت با طول $s$، بدین صورت که $n$ پرتاب را تقریباً به $n/s$ گروه $s$تایی تقسیم خواهیم کرد. با انتخاب $s = \lfloor (\lg n)/۲ \rfloor$ می‌توانیم نشان دهیم احتمالاً حداقل یکی از این گروه‌ها تماماً شیر خواهد بود، و بنابراین می‌توانیم انتظار داشته باشیم که طول بلندترین دوره‌ی موفقیت حداقل $s = \Omega(\lg n)$ باشد. سپس نشان خواهیم داد که امیدریاضی طول بلندترین دوره‌ی موفقیت $\Omega(\lg n)$ است.

$n$ پرتاب سکه را به حداقل $\lfloor n/\lfloor (\lg n)/۲ \rfloor \rfloor$ گروه هریک شامل $\lfloor (\lg n)/۲ \rfloor$ پرتاب متوالی تقسیم می‌کنیم، و برای احتمال این که هیچ یک از گروه‌ها تماماً شیر باشند، یک کران پایین محاسبه می‌کنیم. طبق تساوی (۵-۹) احتمال شیر بودن تمام پرتاب‌های گروهی که از مکان $i$ام آغاز می‌شود برابر است با:

$$\Pr\left\{A_{i,\lfloor (\lg n)/۲ \rfloor}\right\} = \frac{۱}{۲^{\lfloor (\lg n)/۲ \rfloor}} \geq \frac{۱}{\sqrt{n}}$$

بنابراین احتمال این که یک دوره‌ی موفقیت با طول حداقل $\lfloor (\lg n)/۲ \rfloor$ در مکان $i$ام شروع نشود حداکثر $۱-\frac{۱}{\sqrt{n}}$ است. از آن جایی که $\lfloor n/\lfloor (\lg n)/۲ \rfloor \rfloor$ گروه از پرتاب‌های دوبه‌دو مجزا و مستقل ساخته شده‌اند، احتمال این که هیچ *کدام* از این گروه‌ها، دوره‌ای از موفقیت با طول $\lfloor (\lg n)/۲ \rfloor$ نباشند حداکثر برابر است با:

$$\begin{aligned}\left(1-\frac{1}{\sqrt{n}}\right)^{\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor} &\le \left(1-\frac{1}{\sqrt{n}}\right)^{\frac{n}{\lfloor(\lg n)/2\rfloor}-1} \\ &\le \left(1-\frac{1}{\sqrt{n}}\right)^{\frac{2n}{\lg n}-1} \\ &\le e^{-(2n/\lg n-1)/\sqrt{n}} \\ &= O\left(e^{-\lg n}\right) \\ &= O(1/n)\end{aligned}$$

برای این بحث از نامساوی (۳-۱۱) استفاده کردیم، که می‌گوید $1+x \le e^x$. همچنین از این واقعیت که برای $n$های به اندازه‌ی کافی بزرگ داریم $(2n/\lg n-1)/\sqrt{n} \ge \lg n$ (شاید بد نباشد که این عبارت را برای خود اثبات کنید).

بنابراین احتمال این که طول بلندترین دوره‌ی موفقیت از $\lfloor(\lg n)/2\rfloor$ فراتر رود برابر است با:

$$\sum_{j=\lfloor \lg n/2\rfloor+1}^{n} \Pr\{L_j\} \ge 1-O(1/n) \qquad (۵-۱۱)$$

اکنون می‌توانیم یک کران پایین برای طول بلندترین دوره‌ی موفقیت محاسبه کنیم. این کار را با استفاده از تساوی (۵-۱۱) و عملکردی مشابه تحلیل کران بالا شروع می‌کنیم:

$$\begin{aligned}E[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\ &= \sum_{k=0}^{\lfloor(\lg n)/2\rfloor} j \Pr\{L_j\} + \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} j \Pr\{L_j\} \\ &\ge \sum_{j=0}^{\lfloor(\lg n)/2\rfloor} 0.\Pr\{L_j\} + \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} \lfloor(\lg n)/2\rfloor \Pr\{L_j\} \\ &= 0.\sum_{j=0}^{\lfloor(\lg n)/2\rfloor} \Pr\{L_j\} + \lfloor(\lg n)/2\rfloor \sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} \Pr\{L_j\} \\ &\ge 0 + \lfloor(\lg n)/2\rfloor\left(1-O\left(\frac{1}{n}\right)\right) \qquad \text{(طبق نامساوی (۵-۱۱))} \\ &= \Omega(\lg n)\end{aligned}$$

مشابه پارادوکس تولد، می‌توانیم با استفاده از متغیرهای تصادفی شاخص، تحلیلی ساده‌تر ولی تقریبی به دست بیاوریم. فرض کنید $X_{ik} = I\{A_{ik}\}$ متغیر تصادفی شاخص مربوط به دوره‌ی موفقیت با طول حداقل $k$ باشد که با پرتاب $i$ام شروع می‌شود. برای شمارش تمام دوره‌های مانند

این، تعریف می‌کنیم:

$$X = \sum_{i=1}^{n-k+1} X_{ik}$$

با امیدریاضی گرفتن از دو طرف (طبق خطی بودن امیدریاضی) خواهیم داشت:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n-k+1} X_{ik}\right] \\ &= \sum_{i=1}^{n-k+1} E[X_{ik}] \\ &= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\} \\ &= \sum_{i=1}^{n-k+1} \frac{1}{2^k} \\ &= \frac{n-k+1}{2^k} \end{aligned}$$

می‌توانیم با قرار دادن مقادیر مختلف برای $k$، امیدریاضی تعداد دوره‌های موفقیت با طول‌های مختلف را محاسبه کنیم. اگر نتیجه‌ی این محاسبه بزرگ باشد (بسیار بزرگ‌تر از ۱)، انتظار خواهیم داشت که تعداد زیادی دوره‌ی موفقیت با طول $k$ اتفاق بیافتد، و احتمال این که حداقل یکی از آن‌ها داشته باشیم بسیار زیاد است. اگر این عدد کوچک باشد (بسیار کوچک‌تر از ۱)، باید انتظار داشته باشیم که تعداد کمی دوره‌ی موفقیت با طول $k$ اتفاق بیافتد، و احتمال این که حداقل یکی از آن‌ها داشته باشیم کم خواهد بود. اگر برای یک ثابت مثبت $c$ قرار دهیم $k = c \lg n$ به دست می‌آوریم:

$$\begin{aligned} E[X] &= \frac{n - c\lg n + 1}{2^{c\lg n}} \\ &= \frac{n - c\lg n + 1}{n^c} \\ &= \frac{1}{n^{c-1}} - \frac{(c\lg n - 1)/n}{n^{c-1}} \\ &= \theta\left(\frac{1}{n^{c-1}}\right) \end{aligned}$$

در صورتی که $c$ بزرگ باشد امیدریاضی تعداد دوره‌های موفقیت با طول $c\lg n$ بسیار پایین خواهد بود، و نتیجه می‌گیریم که احتمال وقوع آن‌ها نیز کم است. از سوی دیگر اگر قرار دهیم $c = 1/2$ به دست می‌آوریم $E[X] = \theta(1/n^{1/2-1}) = \theta(n^{1/2})$، و می‌توانیم انتظار داشته باشیم که تعداد دوره‌های موفقیت با طول $(1/2)\lg n$ زیاد باشد، و در نتیجه بسیار محتمل است که حداقل یک دوره‌ی موفقیت با این طول رخ دهد. از این برآوردهای تقریبی می‌توانیم نتیجه بگیریم که طول

بلندترین دوره‌ی موفقیت $\theta(\lg n)$ است.

## ۵-۴-۴ مسئله‌ی استخدام آنلاین

به عنوان آخرین مثال نسخه‌ای دیگر از مسئله‌ی استخدام را در نظر می‌گیریم. فرض کنید که نمی‌خواهیم برای یافتن بهترین متقاضی با تمامی متقاضیان مصاحبه کنیم. همچنین نمی‌خواهیم مکرراً با یافتن متقاضیان بهتر، منشی فعلی خود را اخراج کنیم. در این روش متقاضی‌ای را استخدام خواهیم کرد که به بهترین نزدیک است، و در مقابل فقط یک نفر را استخدام می‌کنیم. به علاوه قانونی در شرکت وجود دارد که باید از آن پیروی کنیم: پس از هر مصاحبه، باید بی‌درنگ تصمیم خود (استخدام یا رد) را به متقاضی اعلام کنیم. در این حالت تعادل میان کمینه کردن تعداد مصاحبه‌ها و بیشینه کردن کیفیت متقاضی استخدام شده چگونه خواهد بود؟

می‌توان مسئله را به صورت زیر مدل کرد. پس از هر مصاحبه قادر خواهیم بود به متقاضی مربوطه یک امتیاز بدهیم؛ فرض کنید $score(i)$ امتیاز داده شده به متقاضی $i$ام باشد، و فرض کنید که هیچ دو متقاضی امتیاز یکسان دریافت نخواهند کرد. پس از مصاحبه با $j$ متقاضی، می‌دانیم کدام یک از آن‌ها بالاترین امتیاز را دارد. تصمیم می‌گیریم که از استراتژی زیر استفاده کنیم: ابتدا یک عدد $k < n$ انتخاب می‌کنیم، سپس با $k$ متقاضی اول مصاحبه کرده و آن‌ها را رد می‌کنیم، و در نهایت اولین متقاضی بعد از آن که را از تمام متقاضیان قبل بهتر است، انتخاب می‌کنیم. اگر متوجه شویم که بهترین متقاضی در میان $k$ متقاضی اول بوده، آن گاه متقاضی $n$ام را استخدام می‌کنیم. این استراتژی در رویه‌ی ONLINE-MAXIMUM$(k,n)$ پیاده‌سازی شده است، که در زیر آن را می‌بینیم. مقدار بازگشتی این رویه اندیس کاندیدایی است که باید استخدام کنیم.

```
ON-LINE-MAXIMUM(k, n)
1  bestscore = -∞
2  for i = 1 to k
3      if score(i) > bestscore
4          bestscore = score(i)
5  for i = k + 1 to n
6      if score(i) > bestscore
7          return i
8  return n
```

می‌خواهیم برای هر مقدار ممکن برای $k$، احتمال استخدام بهترین متقاضی را بدست آوریم. در این صورت می‌توانیم بهترین $k$ را انتخاب و استراتژی را با آن پیاده‌سازی کنیم. فعلاً فرض کنید $k$ ثابت است. فرض کنید $M(j) = \max_{1 \le i \le j} \{score(i)\}$ نشان دهنده‌ی بالاترین امتیاز میان متقاضیان ۱ تا $j$ باشد. همچنین $S$ را پیشامد پیروزی در استخدام بهترین متقاضی در نظر بگیرید، و $S_i$ را پیشامد پیروزی در استخدام بهترین متقاضی در حالتی که بهترین متقاضی $i$ باشد. از آن‌جایی که $S_i$ها با هم ناسازگار هستند، داریم $\Pr\{S\} = \sum_{i=1}^{n} \Pr\{S_i\}$. اگر بهترین متقاضی میان $k$ نفر اول باشد، هیچ گاه پیروز نخواهیم شد، بنابراین برای $i = 1, 2, ..., k$ داریم $\Pr\{S_i\} = 0$. به دست می‌آوریم:

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\} \qquad (۵-۱۲)$$

اکنون $\Pr\{S_i\}$ را محاسبه می‌کنیم. اگر متقاضی $i$ام بهترین متقاضی باشد، برای موفق شدن دو چیز باید اتفاق بیافتد. اول، بهترین متقاضی باید در مکان $i$ام باشد، پیشامدی که ما آن را با $B_i$ نشان می‌دهیم. دوم، الگوریتم نباید هیچ یک از متقاضیان $k+1$ تا $i-1$ را انتخاب کند. دومی فقط زمانی اتفاق می‌افتد که برای هر $j$ که $k+1 \le j \le i-1$، عبارت شرطی در خط ۶ درست نباشد (یعنی $score(j) < bestscore$)، که در آن به دلیل یکتا بودن امتیازها از حالت مساوی صرف‌نظر کرده‌ایم). به عبارت دیگر باید تمام مقادیر $score(k+1)$ تا $score(i-1)$ کمتر از $M(k)$ باشند؛ اگر یکی از آن‌ها بزرگ‌تر از $M(k)$ باشد، اندیس متناظر آن توسط رویه بازگردانده خواهد شد. پیشامد عدم انتخاب متقاضیان $k+1$ تا $i-1$ را با $O_i$ نشان می‌دهیم. خوشبختانه پیشامدهای $B_i$ و $O_i$ مستقل هستند. پیشامد $O_i$ فقط به ترتیب نسبی متقاضیان در مکان‌های ۱ تا $i-1$ بستگی دارد، در حالی که $B_i$ بستگی به این دارد که آیا مقدار $i$ از تمامی مقادیر بزرگ‌تر است یا خیر. ترتیب مکان‌ها در میان متقاضیان ۱ تا $i-1$ تأثیری بر روی این بیشینه بودن مقدار $i$ام ندارد، و همچنین مقدار $i$ بر روی ترتیب مکان‌های ۱ تا $i-1$ نخواهد گذاشت. پس می‌توانیم با استفاده از تساوی (پ-۱۵) به دست آوریم

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\}\Pr\{O_i\}$$

احتمال $\Pr\{B_i\}$ بدیهتاً $1/n$ است، چرا که مقدار بیشینه با احتمال برابر در هر یک از مکان‌های ۱ تا $n$ قرار می‌گیرد. برای رخ‌داد پیشامد $O_i$، باید مقدار بیشینه در میان مکان‌های ۱ تا $i-1$ در یکی از $k$ مکان اول قرار گیرد، و البته این مقدار بیشینه با احتمال برابر در هر یک از $i-1$ مکان اول قرار خواهد داشت. در نتیجه $\Pr\{O_i\} = k/(i-1)$ و $\Pr\{S_i\} = k/(n(i-1))$. با استفاده از تساوی (۵-۱۲) خواهیم داشت

$$\begin{aligned}
\Pr\{S\} &= \sum_{i=k+1}^{n} \Pr\{S_i\} \\
&= \sum_{i=k+1}^{n} \frac{k}{n(i-1)} \\
&= \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1} \\
&= \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i}
\end{aligned}$$

از تقریب به کمک انتگرال استفاده کرده و این مجموع را از بالا و پایین محدود می‌کنیم. طبق تساوی (الف-۱۲) داریم:

$$\int_k^n \frac{1}{x}dx \le \sum_{i=k}^{n-1} \frac{1}{i} \le \int_{k-1}^{n-1} \frac{1}{x}dx$$

محاسبه‌ی این انتگرال‌های معین کران‌های زیر را به دست می‌دهد:

$$\frac{k}{n}(\ln n - \ln k) \le \Pr\{S\} \le \frac{k}{n}\big(\ln(n-1) - \ln(k-1)\big)$$

که کرانی نزدیک‌تر برای $\Pr\{S\}$ است. از آن‌جایی که می‌خواهیم احتمال موفقیت خود را بیشتر کنیم، اجازه دهید بر روی انتخاب مقداری برای $k$ تمرکز کنیم که کران پایین $\Pr\{S\}$ را بیشینه می‌کند. (گذشته از این، بیشینه کردن عبارت کران پایین ساده‌تر از کران بالا است.) با مشتق گرفتن از عبارت $(n/k)(\ln n - \ln k)$ نسبت به $k$ خواهیم داشت:

$$\frac{1}{n}(\ln n - \ln k - 1)$$

پس از ریشه گرفتن از این مشتق، می‌بینیم که کران پایین زمانی بیشینه می‌شود که داشته باشیم $\ln k = \ln n - 1 = \ln(n/e)$، یا به طور مشابه وقتی $k = n/e$. پس اگر استراتژی خود را با $k = n/e$ پیاده‌سازی کنیم، حداقل با احتمال $1/e$ در استخدام بهترین متقاضی موفق خواهیم شد.

## تمرین‌ها

۵-۴-۱ حداقل چند نفر باید در یک اتاق باشند تا با احتمال ۱/۲ روز تولد یکی از آن‌ها با شما یکی باشد؟ چند نفر باید در اتاق باشند تا با احتمال حداقل ۱/۲، دو نفر در ۴ جولای به دنیا آمده باشند؟

۵-۴-۲ فرض کنید تعدادی توپ را در $b$ سبد پرتاب می‌کنیم. پرتاب‌ها مستقل هستند، و در پرتاب احتمال قرار گرفتن توپ در تمام سبدها یکسان است. امیدریاضی تعداد پرتاب‌ها قبل از این که حداقل یک سبد حاوی دو توپ باشد چقدر است؟

۵-۴-۳★ در تحلیل پارادوکس تولد آیا باید تمام روزهای تولد مستقل از یکدیگر باشند، یا استقلال دوبه‌دو کافی است؟ جواب خود را توجیه کنید.

۵-۴-۴★ چند نفر باید به یک میهمانی دعوت شده باشند تا انتظار داشته باشیم که سه نفر با روز تولد یکسان در میهمانی باشند؟

۵-۴-۵★ احتمال این که رشته‌ای $k$ تایی روی مجموعه‌ای با اندازه‌ی $n$، یک جایگشت $k$ تایی باشد چقدر است؟ این سؤال چگونه به پارادوکس تولد مربوط می‌شود؟

۵-۴-۶★ فرض کنید $n$ توپ در $n$ سبد پرتاب می‌شوند، که در آن پرتاب‌ها مستقل از یکدیگرند و توپ‌ها با احتمال یکسان در هر یک از سبدها خواهند افتاد. امیدریاضی سبدهای خالی چقدر است؟ امیدریاضی سبدهای حاوی دقیقاً یک توپ چقدر است؟

۵-۴-۷★ در مسئله‌ی دوره‌ی پیروزی، یک کران پایین نزدیک‌تر برای طول دوره بیابید. برای این کار نشان دهید که در پرتاب $n$ بار سکه‌ی متقارن، احتمال این که هیچ دوره‌ی موفقیتی با طول بزرگ‌تر از $\lg n - 2\lg\lg n$ اتفاق نیافتد کمتر از $1/n$ است.

# مسائل



## ۵-۱ *شمارش احتمالاتی*

با یک شمارنده‌ی $b$ بیتی به طور معمول فقط می‌توانیم تا $2^b - 1$ بشماریم. به کمک روش *شمارش احتمالاتی* R. Morris، در ازای کمی از دست دادن دقت می‌توان تا مقادیر بسیار بیشتری شمرد.

یک شمارنده‌ی $i$ در نظر می‌گیریم که اندیس مقدار فعلی شمارش ($n_i$) است، و $i = 0,1,\ldots,2^b - 1$. به عبارت دیگر برای هر مقدار $i$ فرض می‌کنیم تا کنون تا $n_i$ شمرده‌ایم. $n_i$ها دنباله‌ای صعودی از مقادیر نامنفی را تشکیل می‌دهند. فرض می‌کنیم مقدار اولیه‌ی اندیس شمارنده ۰ باشد، نماینده‌ی مقدار شمارش $n_0 = 0$. عملگر INCREMENT بر روی شمارنده‌ی با مقدار $i$ به صورت احتمالاتی عمل می‌کند. اگر $i = 2^b - 1$، آن گاه یک خطای سرریز (overflow) رخ می‌دهد. در غیر این صورت شمارنده با احتمال $1/(n_{i+1} - n_i)$ به اندازه‌ی یک واحد افزایش خواهد یافت، و با احتمال $1 - 1/(n_{i+1} - n_i)$ بدون تغییر باقی خواهد ماند.

اگر دنباله‌ی $n_i = i$ برای هر $i \geq 0$ انتخاب کنیم، یک شمارنده‌ی معمولی خواهیم داشت. با انتخاب مقادیری دیگر، مثلاً $n_i = 2^{i-1}$ برای $i > 0$، و یا $n_i = F_i$ (عدد فیبوناچی $i$ام – بخش ۳-۲ را ببینید)، نتایج بسیار جذاب‌تری خواهیم گرفت.

در این مسئله فرض می‌کنیم $n_{2^b - 1}$ به اندازه‌ی کافی بزرگ هست که احتمال وقوع یک خطای سرریز قابل چشم پوشی باشد.

I. نشان دهید که امیدریاضی مقدار نمایش داده شده توسط شمارنده پس از $n$ عملگر INCREMENT، دقیقاً برابر است با $n$.

II. تحلیل واریانس عدد نمایش داده شده توسط شمارنده به دنباله‌ی $n_i$ وابسته است. اجازه دهید حالت خاص $n_i = 100i$ را برای تمام $i$های بزرگتر یا مساوی صفر را در نظر بگیریم. واریانس مقدار نمایش داده شده توسط شمارنده را پس از $n$ عملگر INCREMENT تخمین بزنید.

## ۵-۲ *جستجو در یک آریه‌ی مرتب نشده*

این مسئله سه الگوریتم را برای جستجوی یک مقدار $x$ در یک آرایه‌ی مرتب نشده‌ی $A$ حاوی $n$ عنصر آزمایش می‌کند.

استراتژی تصادفی زیر را در نظر بگیرید: یک اندیس تصادفی $i$ در $A$ انتخاب می‌کنیم. اگر $A[i] = x$، الگوریتم به پایان می‌رسد؛ در غیر این صورت جستجو را با انتخاب یک اندیس تصادفی دیگر در $A$ ادامه می‌دهیم. همین طور با انتخاب اندیس‌های تصادفی ادامه می‌دهیم تا این که اندیس $j$ را بیابیم طوری که داشته باشیم $A[j] = x$، و یا همه‌ی عناصر $A$ را بررسی کرده باشیم. توجه کنید که هر بار اندیس تصادفی را از میان تمام عناصر انتخاب می‌کنیم، و

بنابراین احتمال بررسی یک اندیس بیش از یک بار وجود دارد.

I. شبه‌کدی برای رویه‌ی RANDOM-SEARCH بنویسید که استراتژی بالا را پیاده‌سازی می‌کند. اطمینان حاصل کنید که با آزمایش تمام اندیس‌ها در $A$، الگوریتم شما پایان می‌یابد.

II. فرض کنید دقیقاً یک اندیس $i$ وجود دارد به طوری که $A[i]=x$. امیدریاضی تعداد اندیس‌هایی که در $A$ باید امتحان شوند تا $x$ پیدا شود و الگوریتم پایان یابد، چیست؟

III. جواب خود به قسمت II را گسترش دهید، بدین صورت که فرض کنید تعداد $k \geq ۱$ اندیس $i$ وجود دارد به طوری که $A[i]=x$. امیدریاضی تعداد اندیس‌هایی که باید امتحان شوند تا $x$ پیدا شود و الگوریتم پایان یابد، چیست؟ جواب شما باید تابعی از $n$ و $k$ باشد.

IV. فرض کنید مقدار $x$ در آرایه‌ی $A$ وجود ندارد. امیدریاضی تعداد اندیس‌هایی که باید آزمایش شوند تا الگوریتم به صورت ناموفق پایان یابد، چیست؟

اکنون یک الگوریتم جستجوی خطی قطعی را در نظر بگیرید، که آن را DETERMINISTIC-SEARCH می‌نامیم. مخصوصاً، این الگوریتم آرایه‌ی $A$ را به ترتیب برای یافتن $x$ می‌گردد، یعنی عناصر را به ترتیب $A[۱], A[۲], A[۳], ..., A[n]$ بررسی می‌کند تا این که یا $A[i]=x$ پیدا شود و یا به انتهای آرایه برسیم. فرض کنید احتمال وقوع تمام جایگشت‌های آرایه‌ی ورودی با هم برابر است.

V. فرض کنید دقیقاً یک اندیس وجود دارد که $A[i]=x$. امیدریاضی زمان اجرای DETERMINISTIC-SEARCH چیست؟ بدترین حالت زمان اجرای DETERMINISTIC-SEARCH چقدر است؟

VI. جواب خود به قسمت V را گسترش دهید، بدین صورت که فرض کنید تعداد $k \geq ۱$ اندیس $i$ وجود دارد که $A[i]=x$. امیدریاضی زمان اجرای DETERMINISTIC-SEARCH چیست؟ بدترین حالت زمان اجرای DETERMINISTIC-SEARCH چقدر است؟ جواب شما باید تابعی از $n$ و $k$ باشد.

VII. فرض کنید مقدار $x$ در آرایه‌ی $A$ وجود ندارد. امیدریاضی زمان اجرای DETERMINISTIC-SEARCH چیست؟ بدترین حالت زمان اجرای DETERMINISTIC-SEARCH چقدر است؟

نهایتاً الگوریتم تصادفی SCRAMBLE-SEARCH را در نظر بگیرید، که ابتدا آرایه‌ی ورودی را به صورت تصادفی مرتب کرده و سپس جستجوی خطی قطعی را روی آرایه‌ی حاصل اجرا می‌کند.

VIII. اگر $k$ تعداد اندیس‌های $i$ باشد که $A[i]=x$، بدترین حالت زمان اجرا و امیدریاضی زمان اجرای SCRAMBLE-SEARCH را برای حالت‌های $k=۰$ و $k=۱$ بیابید. جواب خود را گسترش کنید تا برای حالت‌های $k \geq ۱$ هم صدق کند.

IX. کدام‌یک از این سه الگوریتم را برای جستجو پیشنهاد می‌کنید؟ جواب خود را توجیه کنید.

# بخش دوم
# مرتب سازی و آمار ترتیبی

**شامل فصل‌های :**

مرتب‌سازی هرمی
مرتب‌سازی سریع
مرتب‌سازی در زمان خطی
میانه‌ها و شاخص‌های ترتیبی

## مقدمه

در این بخش، الگوریتم‌های مختلفی برای *مسئله‌ی مرتب‌سازی* زیر ارائه خواهد شد:

- ورودی: دنباله‌ای از $n$ عدد: $\langle a_1, a_2, ..., a_n \rangle$.
- خروجی: یک جایگشت (مرتب‌سازی) $\langle a'_1, a'_2, ..., a'_n \rangle$ از دنباله‌ی ورودی به طوری که $a'_1 \leq a'_2 \leq ... \leq a'_n$.

دنباله‌ی ورودی معمولا یک آرایه‌ی $n$ عضوی است، ولی بعضی مواقع به صورت‌های دیگری ارائه خواهد شد، مانند یک لیست پیوندی.

### ساختار داده‌ها

در عمل به ندرت اعدادی که باید مرتب شوند، مقادیر تنها هستند. معمولا هر کدام از آن‌ها قسمتی از مجموعه‌ای از داده‌ها، به نام *رکورد* (record) هستند. هر رکورد شامل یک *کلید* (key) است، که همان مقداری است که داده‌ها باید براساس آن مرتب شوند، و بقیه‌ی رکورد عبارت است از *داده‌های پیرو* (satellite data) که در مرتب‌سازی همراه کلیدها جابه‌جا می‌شوند. در عمل، وقتی یک الگوریتم مرتب‌سازی کلیدها را مرتب می‌کند، باید داده‌های پیرو را هم مرتب کند. اگر هر رکورد حاوی حجم زیادی از داده‌های پیرو باشد، برای مینیمم کردن داده‌هایی که باید جابه‌جا شوند، به جای خود داده‌ها اشاره‌گرهایی به داده‌ها را مرتب می‌کنند.

تا حدودی، همین جزئیات پیاده‌سازی است که یک برنامه‌ی کامل را از یک الگوریتم متمایز می‌کند. چه ما خود داده‌ها را جابه‌جا کنیم، و چه اشاره‌گرهایی به داده‌ها را، بر روی *روش* مرتب‌سازی مورد استفاده تأثیری نخواهد داشت. بنابراین وقتی بر روی مسئله‌ی مرتب‌سازی تمرکز می‌کنیم، فرض می‌کنیم که ورودی فقط شامل اعداد است. تبدیل یک الگوریتم مرتب‌سازی اعداد به یک برنامه برای مرتب‌سازی رکوردها کاری سرراست است. با این حال، باز هم در بعضی موقعیت‌های مهندسی ممکن است ظرافت‌هایی وجود داشته باشد که تبدیل الگوریتم به برنامه، خود چالشی جداگانه باشد.

## چرا مرتب‌سازی؟

بسیاری از دانشمندان علم کامپیوتر، مسئله‌ی مرتب‌سازی را اساسی‌ترین مسئله در یادگیری الگوریتم‌ها می‌دانند. دلیل‌های زیادی برای این امر وجود دارد:

- بعضی مواقع احتیاج به مرتب‌سازی داده‌ها ذاتی کاربرد مورد نظر است. به عنوان مثال، برای آماده سازی صورت حساب مشتریان، بانک احتیاج دارد که چک‌ها را برحسب شماره‌ی چک مرتب‌سازی کند.
- معمولا الگوریتم‌ها از مرتب‌سازی به عنوان یک زیرروال کلیدی استفاده می‌کنند. مثلا، یک برنامه که اشیاء گرافیکی سه بعدی را که روی هم قرار دارند روی صفحه نمایش چاپ می‌کند، ممکن است مجبور باشد که اشیاء را برحسب رابطه‌ی «روی هم بودن» مرتب کند تا بتواند آن‌ها را به ترتیب از پایین به بالا بکشد. در این کتاب، الگوریتم‌های بسیاری را خواهیم دید که از مرتب‌سازی به عنوان یک زیرروال استفاده می‌کنند.
- الگوریتم‌های مرتب‌سازی بسیار متنوعی وجود دارد، که از مجموعه‌ای غنی از تکنیک‌ها استفاده می‌کنند. در واقع، بسیاری از تکنیک‌های مهمی که در طراحی الگوریتم‌ها از آن‌ها استفاده می‌شود، در طی سال‌ها و در بدنه‌ی الگوریتم‌های مرتب‌سازی معرفی شده‌اند. می‌بینید که مرتب‌سازی پیشینه‌ی تاریخی بلندی دارد.
- مرتب‌سازی، مسئله‌ای است که می‌توان برای آن یک کران پایین قابل توجه اثبات کرد (در فصل ۸ این کار را انجام می‌دهیم). بهترین کران‌های بالایی که برای الگوریتم‌های مرتب‌سازی وجود دارد، به صورت حدی با این کران پایین یکی هستند، و بنابراین می‌توانیم مطمئن باشیم که الگوریتم‌های مرتب‌سازی موجود به صورت حدی بهینه هستند. به علاوه، می‌توانیم از کران پایین مرتب‌سازی برای اثبات کران پایین بسیاری از الگوریتم‌های دیگر استفاده کنیم.
- بسیاری از مسائل مهندسی هنگام پیاده‌سازی الگوریتم‌های مرتب‌سازی پیش می‌آیند. سریع‌ترین برنامه‌ی مرتب‌سازی برای یک موقعیت خاص، ممکن است به فاکتورهای بسیاری بستگی داشته باشد، مانند دانش قبلی در مورد کلیدها و داده‌های پیرو، سلسله مراتب حافظه (memory hirarchy) (حافظه‌ی مجازی و کش) در اکثر کامپیوترها و محیط نرم‌افزاری. بسیاری از این مسائل بهتر است در سطح الگوریتم حل شوند، نه با «سرو کله زدن» با کد.

## الگوریتم‌های مرتب‌سازی

در فصل ۲، دو الگوریتم معرفی کردیم که اعداد حقیقی را مرتب می‌کنند. مرتب‌سازی درجی در بدترین حالت در زمان $\theta(n^2)$ مرتب‌سازی را انجام می‌دهد. با این حال، از آن جایی که حلقه‌ی داخلی آن کار اضافی انجام نمی‌دهد، این الگوریتم برای ورودی‌های با اندازه‌ی کوچک سریع است، و به علاوه عناصر را درجا مرتب می‌کند. (به یاد بیاورید که الگوریتمی *درجا* مرتب می‌کند که تعداد عناصری که خارج از آرایه ذخیره می‌شوند، ثابت باشد.) مرتب‌سازی ادغامی زمان اجرای حدی بهتری دارد، $\theta(n \lg n)$، ولی رویه‌ی MERGE درجا مرتب‌سازی را انجام نمی‌دهد.

در این بخش، دو الگوریتم دیگر را معرفی می‌کنیم که تعدادی عدد حقیقی را مرتب می‌کنند. مرتب‌سازی هرمی، که در فصل ۶ معرفی خواهد شد، $n$ عدد را درجا و در زمان $O(n \lg n)$ مرتب می‌کند. این الگوریتم برای این کار از یک ساختار داده‌ی مهم به نام هرم استفاده می‌کند، که به کمک آن می‌توانیم صف‌های اولویت را هم پیاده‌سازی کنیم.

مرتب‌سازی سریع هم در فصل ۷، $n$ عدد را درجا مرتب می‌کند، ولی بدترین حالت زمان اجرای آن $\theta(n^2)$ است. با این وجود، زمان اجرای متوسط آن $\theta(n \lg n)$ است، و معمولا در عمل کارایی آن از مرتب‌سازی هرمی بهتر می‌باشد. مانند مرتب‌سازی درجی، مرتب‌سازی سریع هم کد کوتاهی دارد، و به همین دلیل ضریب ثابت زمان اجرای آن کوچک است. این الگوریتم، الگوریتمی معروف برای مرتب‌سازی آرایه‌های ورودی با اندازه‌ی بزرگ است.

مرتب‌سازی درجی، مرتب‌سازی ادغامی، مرتب‌سازی هرمی و مرتب‌سازی سریع، همه الگوریتم‌های مرتب‌سازی مقایسه‌ای هستند: یعنی ترتیب عناصر را به کمک مقایسه‌ی آن‌ها انجام می‌دهند. فصل ۸ با معرفی مدل درخت تصمیم‌گیری (decision tree) برای بررسی محدودیت‌های مرتب‌سازی‌های مقایسه‌ای آغاز می‌شود. با استفاده از این مدل، اثبات خواهیم کرد که کران پایین بدترین حالت زمان اجرای تمام الگوریتم‌های مقایسه‌ای که $n$ ورودی را مرتب می‌کنند، $\Omega(n \lg n)$ است، که نشان می‌دهد مرتب‌سازی هرمی و مرتب‌سازی ادغامی به صورت حدی، کاراترین الگوریتم‌های مرتب‌سازی مقایسه‌ای هستند.

سپس، نشان خواهیم داد که اگر بتوانیم به روشی غیر از مقایسه اطلاعاتی در مورد ترتیب عناصر به دست آوریم، می‌توانیم این کران پایین را بشکنیم. به عنوان مثال، الگوریتم مرتب‌سازی شمارشی، فرض می‌کند که اعداد ورودی در مجموعه‌ی $\{1,2,\ldots,k\}$ هستند. مرتب‌سازی شمارشی ، با استفاده از اندیس آرایه‌ها به عنوان یک وسیله برای تشخیص ترتیب عناصر، می‌تواند $n$ عدد را در زمان $\theta(k+n)$ مرتب کند. بنابراین، وقتی $k = O(n)$، این الگوریتم در زمان خطی نسبت به اندازه‌ی ورودی اجرا می‌شود. با استفاده از یک الگوریتم مشابه به نام مرتب‌سازی مبنایی، می‌توان دامنه‌ی مرتب‌سازی شمارشی را افزایش داد. اگر $n$ عدد برای مرتب‌سازی داشته باشیم، و هر عدد $d$ رقم داشته باشد، و هر رقم در مجموعه‌ی $\{1,2,\ldots,k\}$ باشد، آن گاه مرتب‌سازی مبنایی در زمان

$\theta(d(n+k))$ عداد را مرتب می‌کند. اگر $d$ ثابت و $k$ از مرتبه‌ی $O(n)$ باشد، مرتب‌سازی مبنایی در زمان خطی اجرا می‌شود. الگوریتم سوم یعنی مرتب‌سازی سطلی، احتیاج به اطلاعاتی در مورد توزیع احتمالاتی اعداد در آرایه‌ی ورودی دارد. این الگوریتم می‌تواند $n$ عدد حقیقی در بازه‌ی نیمه باز $[0,1)$ را که به صورت یکنواخت توزیع شده‌اند، در زمان متوسط $O(n)$ مرتب کند.

زمان‌های اجرای مربوط به الگوریتم‌های مرتب‌سازی ارائه شده در فصل‌های ۲ و ۶-۸ در جدول زیر خلاصه شده‌اند. مانند همیشه، $n$ نشان دهنده‌ی تعداد عناصری است که باید مرتب شوند. برای مرتب‌سازی شمارشی، عناصری که باید مرتب شوند، اعدادی هستند از مجموعه‌ی $\{0,1,...,k\}$. برای مرتب‌سازی مبنایی، هر عنصر یک عدد $d$رقمی است، که در آن هر رقم یکی از $k$ مقدار ممکن را به خود می‌گیرد. برای مرتب‌سازی سطلی، فرض می‌کنیم که کلیدها، اعداد حقیقی هستند که به صورت یکنواخت در بازه‌ی نیمه‌باز $[0,1)$ پراکنده شده‌اند. آخرین ستون سمت چپ نشان‌دهنده‌ی حالت متوسط، یا امیدریاضی زمان اجرا است، که وقتی که با بدترین حالت زمان اجرا متفاوت است، در جدول دقیقا مشخص شده است که منظور کدام است. از حالت متوسط زمان اجرای مرتب‌سازی هرمی صرف نظر می‌کنیم، چرا که در این کتاب تحلیلی بر روی آن صورت نمی‌گیرد.

| الگوریتم | بدترین حالت زمان اجرا | حالت متوسط/امیدریاضی زمان اجرا |
|---|---|---|
| *مرتب‌سازی درجی* | $\theta(n^2)$. | $\theta(n^2)$. |
| مرتب‌سازی ادغامی | $\theta(n \lg n)$. | $\theta(n \lg n)$. |
| مرتب‌سازی هرمی | $O(n \lg n)$. | – |
| مرتب‌سازی سریع | $\theta(n^2)$. | $\theta(n \lg n)$ (امیدریاضی). |
| مرتب‌سازی شمارشی | $\theta(k+n)$. | $\theta(k+n)$. |
| مرتب‌سازی مبنایی | $\theta(d(n+k))$. | $\theta(d(n+k))$. |
| مرتب‌سازی سطلی | $\theta(n^2)$. | $\theta(n)$ (حالت متوسط). |

## شاخص‌های ترتیبی

$i$امین شاخص ترتیبی در مجموعه‌ای از $n$ عدد، $i$امین عدد کوچک در آن مجموعه است. مسلما به راحتی می‌توان اعداد یک مجموعه را مرتب کرد، و سپس $i$امین عدد را به عنوان خروجی بازگرداند. بدون هیچ فرضی در مورد توزیع اعداد، این متد در زمان $\Omega(n \lg n)$ اجرا می‌شود، که کران پایین اثبات شده در فصل ۸ آن را تأیید می‌کند.

در فصل ۹، نشان می‌دهیم که می‌توانیم $i$امین عنصر کوچک یک مجموعه را در زمان $O(n)$ پیدا کنیم، حتی اگر عناصر مجموعه، اعداد حقیقی دلخواه باشند. یک الگوریتم تصادفی با سودوکد کوتاه ارائه خواهیم کرد که بدترین حالت زمان اجرای آن $\theta(n^2)$ است، ولی امیدریاضی زمان اجرای آن $O(n)$ است. همچنین، یک الگوریتم پیچیده‌تر ارائه خواهیم کرد که در بدترین حالت در زمان $O(n)$ اجرا می‌شود.

## زمینه‌ی مورد نیاز

با این که اکثر این بخش به ریاضیات پیچیده وابسته نیست، ولی در بعضی از قسمت‌ها به زمینه‌ی ریاضیاتی نسبتا سخت احتیاج پیدا خواهیم کرد. به طور کلی، در تحلیل مرتب‌سازی سریع، مرتب‌سازی سطلی و الگوریتم شاخص آماری ترتیبی از احتمالات استفاده خواهیم کرد، که آن را در پیوست پ مرور خواهیم کرد. همچنین مباحثی در تحلیل احتمالاتی و الگوریتم‌های تصادفی در فصل ۵ بررسی شدند. ریاضیات مورد نیاز برای تحلیل بدترین حالت زمان اجرای الگوریتم شاخص‌های آماری ترتیبی که در زمان خطی اجرا می‌شود، تا حدودی پیچیده‌تر از بقیه‌ی تحلیل‌های بدترین حالت زمان اجرا در این بخش است.

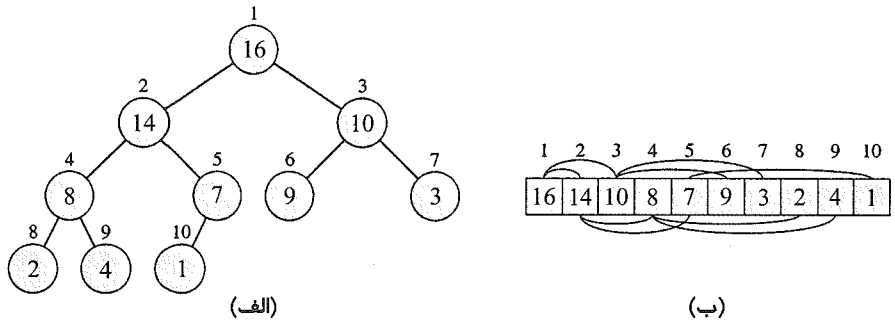

# مرتب‌سازی هرمی

## ۶-۰ مقدمه

در این فصل الگوریتمی دیگر برای مسئله‌ی مرتب‌سازی معرفی خواهیم کرد: مرتب‌سازی هرمی. مانند مرتب‌سازی ادغامی، و بر خلاف مرتب‌سازی درجی، زمان اجرای این الگوریتم $O(n \lg n)$ است. مانند مرتب‌سازی درجی، و برخلاف مرتب‌سازی ادغامی، الگوریتم مرتب‌سازی هرمی داده‌ها را در جا مرتب می‌کند: غیر از آرایه‌ی ورودی، مقدار حافظه‌ی مورد استفاده ثابت است. بنابراین خصوصیات خوب هر دو الگوریتم مرتب‌سازی قبلی در مرتب‌سازی هرمی وجود دارد.

مرتب‌سازی هرمی تکنیک طراحی دیگری هم معرفی می‌کند: استفاده از یک ساختمان داده برای مدیریت اطلاعات در طول اجرای الگوریتم. ساختمان داده‌ی مورد استفاده در این الگوریتم «هرم» (heap) نام دارد. این ساختمان داده نه تنها برای الگوریتم مرتب‌سازی هرمی مفید است، بلکه به عنوان یک صف اولویت کارآمد نیز قابل استفاده است. در فصل‌های بعد الگوریتم‌های زیادی را مشاهده خواهیم کرد که از این ساختمان داده استفاده می‌کنند.

واژه‌ی «هرم» در ابتدا در مفهوم الگوریتم مرتب‌سازی هرمی معرفی شد، ولی بعد از آن در بعضی از زبان‌های برنامه‌نویسی (مانند Lisp و Java) در معنی «حافظه‌ی بی‌مصرف جمع‌آوری شده» (garbage-collected storage) به کار رفته است. در این جا ساختمان داده‌ی هرم حافظه‌ی جمع‌آوری شده نیست، و هرجا در این کتاب صحبت از هرم می‌شود منظور ساختمان داده‌ی تعریف شده در این فصل است، و ربطی به جمع‌آوری حافظه‌ی بی‌مصرف ندارد.




**شکل ۶-۱** یک هرم بیشینه که به صورت (الف) یک درخت دودویی، و (ب) یک آرایه دیده می‌شود. عدد درون هر گره نشان‌دهنده‌ی مقدار ذخیره شده در آن عنصر است، و عدد بالای گره، اندیس مربوط به گره در آرایه را نشان می‌دهد. خطوط بالا و پایین عناصر آرایه نشان‌دهنده‌ی روابط پدر و فرزندی است؛ هر گره‌ی پدر در آرایه در سمت چپ گره‌ی فرزند است. ارتفاع درخت سه است؛ گره‌ی مربوط به عنصر چهارم آرایه (با مقدار ۸) دارای ارتفاع دو است.

## ۶-۱ هرم‌ها

ساختمان داده‌ی *هرم (دودویی)*، یک آرایه از اشیاء است که می‌توان آن را به چشم یک درخت دودویی (بخش ب-۵-۳) تقریباً کامل دید، همان طور که در شکل ۶-۱ نشان داده شده است. هر گره از درخت معادل یک عنصر در آرایه است و مقدار عنصر در آن گره ذخیره شده است. تمام سطوح درخت کاملاً پر هستند، احتمالاً غیر از سطح آخر، که این سطح هم از سمت چپ تا جای ممکن پر خواهد بود. آرایه‌ی $A$ که نشان‌دهنده‌ی یک هرم است، شیئی است با دو خصوصیت: $A.length$ که تعداد عناصر ذخیره شده در آرایه را نشان می‌دهد، و $A.heap\text{-}size$ که نشان دهنده‌ی تعداد عناصر هرم است (که در $A$ ذخیره شده‌اند). یعنی با این که ممکن است تمام عناصر $A$ اعداد معتبر باشند، ولی هیچ عنصری بعد از $A[A.heap\text{-}size]$، عنصری از هرم نیست (داریم $A.heap\text{-}size \le A.length$). ریشه‌ی درخت $A[1]$ است، و با داشتن اندیس یک گره در آرایه، مثلاً $i$، می‌توان به راحتی اندیس پدر آن گره در درخت، $\text{PARENT}(i)$، اندیس فرزند چپ، $\text{LEFT}(i)$، و اندیس فرزند راست آن، $\text{RIGHT}(i)$، را محاسبه کرد:

```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

در اکثر کامپیوترها، رویه‌ی LEFT می‌تواند به راحتی و با شیفت به چپ نمانش دودویی $i$ به

اندازه‌ی یک بیت، $2i$ را با یک دستورالعمل محاسبه کند. به طور مشابه رویه‌ی RIGHT می‌تواند به سرعت $2i+1$ را با یک شیفت و یک جمع محاسبه کند. همچنین رویه‌ی PARENT می‌تواند $\lfloor i/2 \rfloor$ را با یک شیفت به سمت راست بیابد. در پیاده‌سازی‌های خوب از مرتب‌سازی هرمی، این رویه‌ها معمولاً به صورت «ماکرو» و یا توابع «دورن خطی» (in-line) نوشته می‌شوند.

دو نوع هرم وجود دارد: هرم بیشینه و هرم کمینه. در هر دو نوع، مقادیر درون گره‌ها دارای خاصیتی به نام خصوصیت هرم (heap property) هستند، که جزئیات آن به نوع هرم مربوط می‌شود. در یک هرم بیشینه، خصوصیت هرم بیشینه این است که به ازای هر گره غیر از ریشه داریم

$$A[PARENT(i)] \geq A[i]$$

یعنی مقدار هر گره حداکثر برابر است با مقدار گره‌ی پدر خود. بنابراین بزرگ‌ترین عنصر در یک هرم بیشینه عنصر ریشه است، و زیردرخت هر گره، شامل مقادیری است که کوچک‌تر یا مساوی مقدار همان گره هستند. خصوصیت هرم کمینه دقیقاً برعکس است؛ برای هر گره‌ی $i$ غیر از ریشه،

$$A[PARENT(i)] \leq A[i]$$

کوچک‌ترین عنصر در یک هرم کمینه در ریشه‌ی آن است.

برای الگوریتم مرتب‌سازی هرمی از هرم‌های بیشینه استفاده می‌کنیم. معمولاً در صف‌های اولویت از هرم‌های کمینه استفاده می‌شود، که در بخش ۶-۵ در مورد آن‌ها بحث خواهد شد. در مورد تعیین این که در هر الگوریتم خاص به هرم کمینه احتیاج داریم و یا به هرم بیشینه، دقیق خواهیم بود، و اگر زمانی هم خصوصیات هرم کمینه کاربرد داشته باشد و هم خصوصیات هرم بیشینه، فقط از عبارت «هرم» استفاده خواهیم کرد.

اگر به هرم به صورت درخت نگاه کنیم، می‌توانیم ارتفاع هر گره را به صورت تعداد یال‌ها در طولانی‌ترین مسیر ساده‌ی پایینی از آن گره به یک برگ تعریف کنیم، و ارتفاع هرم برابر است با ارتفاع ریشه‌ی درخت. از آن جایی که یک هرم با $n$ عنصر بر مبنای یک درخت دودویی کامل است، ارتفاع آن برابر است با $\theta(\lg n)$ (تمرین ۶-۱-۲ را ببینید). خواهیم دید که عملیات روی هرم‌ها در زمان حداکثر خطی نسبت به ارتفاع درخت، یعنی $O(\lg n)$، اجرا می‌شوند. در ادامه‌ی این فصل پنج رویه‌ی اصلی معرفی خواهند شد، و نشان خواهیم داد که چطور می‌توان از آن‌ها در یک الگوریتم مرتب‌سازی و یک ساختمان داده‌ی صف اولویت استفاده کرد.

- رویه‌ی MAX-HEAPIFY که در زمان $O(\lg n)$ اجرا می‌شود، کلید حفظ خصوصیت هرم بیشینه است.
- رویه‌ی BUILD-MAX-HEAP که در زمان خطی اجرا می‌شود، از یک آرایه‌ی ورودی نامرتب، یک هرم بیشینه می‌سازد.
- رویه‌ی HEAPSORT که در زمان $O(n \lg n)$اجرا می‌شود، یک آرایه را به صورت در جا مرتب می‌کند.
- رویه‌های MAX-HEAP-INSERT، HEAP-EXTRACT-MAX، HEAP-INCREASE-KEY، و

HEAP-MAXIMUM، که همه در زمان $O(\lg n)$ اجرا می‌شوند، به ساختمان داده‌ی هرم اجازه می‌دهند که در یک صف اولویت مورد استفاده قرار گیرد.

## تمرین‌ها

۶-۱-۱ حداقل و حداکثر تعداد عناصر در یک هرم با ارتفاع $h$ چقدر است؟

۶-۱-۲ نشان دهید که ارتفاع یک هرم با $n$ عنصر برابر است با $\lfloor \lg n \rfloor$.

۶-۱-۳ نشان دهید که در هر زیردرختی از یک هرم بیشینه، ریشه‌ی زیردرخت حاوی بزرگ‌ترین عنصر در تمام آن زیردرخت است.

۶-۱-۴ با فرض این که تمام عناصر در یک هرم بیشینه یکتا هستند، کوچک‌ترین عنصر آن در کجا ممکن است قرار گیرد؟

۶-۱-۵ آیا یک آرایه‌ی مرتب شده، یک هرم کمینه است؟

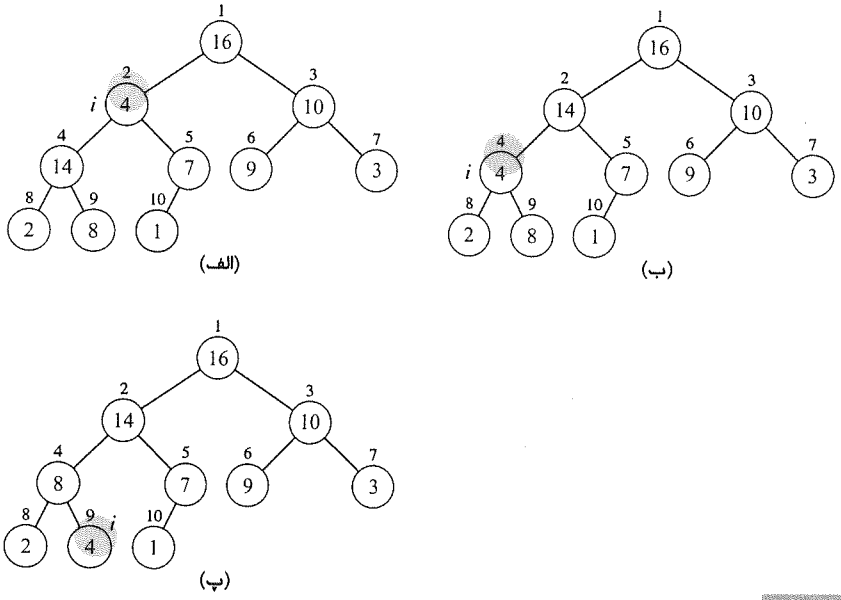
۶-۱-۶ آیا دنباله‌ی $\langle ۲۳,۱۷,۱۴,۶,۱۳,۱۰,۱,۵,۷,۱۲ \rangle$ یک هرم بیشینه است؟

۶-۱-۷ نشان دهید که اگر یک هرم را به صورت یک آرایه نشان دهیم، عناصر با اندیس‌های $\lfloor n/۲ \rfloor + ۲, \ldots, n$ حاوی برگ‌ها خواهند بود.

## ۶-۲ حفظ خصوصیت هرم

برای حفظ خصوصیت هرم‌های بیشینه از رویه‌ی MAX-HEAPIFY استفاده می‌کنیم. ورودی‌های این رویه عبارتند از یک آرایه‌ی $A$ و یک اندیس $i$ درون آرایه. هنگام فراخوانی MAX-HEAPIFY فرض می‌شود که درخت‌های دودویی با ریشه‌های $\text{LEFT}(i)$ و $\text{RIGHT}(i)$ هرم بیشینه هستند، ولی ممکن است مقدار $A[i]$ از فرزندان خود کوچک‌تر باشد، و بنابراین خصوصیت هرم بیشینه در تمام درخت برقرار نیست. کارکرد MAX-HEAPIFY بدین صورت است که مقدار $A[i]$ را در درخت به سمت پایین حرکت می‌دهد تا جایی که زیردرخت با ریشه‌ی $i$ تبدیل به یک هرم بیشینه شود.

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4     largest = l
5     else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7     largest = r
8  if largest ≠ i
9     exchange A[i] with A[largest]
10    MAX-HEAPIFY(A, largest)
```




**شکل ۶-۲** عملیات MAX-HEAPIFY $(A,2)$، که در آن $A.heap\text{-}size = 10$. (الف) ساختار اولیه، که در آن $A[2]$ در گره‌ی $i = 2$ از خصویت هرم‌های بیشینه پیروی نمی‌کند، چرا که مقدار آن از مقدار هردو فرزند خود بیشتر نیست. (ب) خصوصیت هرم‌های بیشینه برای گره‌ی ۲ با عوض کردن $A[2]$ با $A[4]$ برقرار می‌شود، ولی بعد از آن خصوصیت هرم‌های بیشینه برای گره‌ی $A[4]$ از بین برود. اکنون در فراخوانی بازگشتی MAX-HEAPIFY $(A,4)$، مقدار $i$ برابر است با ۴. پس از جابه‌جایی $A[4]$ با $A[9]$، همان طور که در (پ) نشان داده شده است، مشکل گره‌ی ۴ حل خواهد شد، و فراخوانی بازگشتی MAX-HEAPIFY $(A,9)$ تغییر دیگری در ساختار درخت نمی‌دهد.

شکل ۶-۲ عملیات MAX-HEAPIFY را نشان می‌دهد. در هر مرحله، بزرگ‌ترین عنصر از میان عناصر $A[i]$، $A[LEFT(i)]$، و $A[RIGHT(i)]$ تعیین شده و اندیس آن در *largest* ذخیره می‌شود. اگر $A[i]$ بزرگ‌ترین عنصر باشد، آن گاه زیردرخت با ریشه‌ی $i$ یک هرم بیشینه است، و رویه پایان می‌یابد. در غیر این صورت یکی از دو فرزند بزرگ‌ترین عنصر هستند، و جای $A[i]$ با $A[largest]$ عوض می‌شود.این کار باعث می‌شود عنصر $i$ و دو فرزندش خصوصیت هرم بیشینه را ارضا کنند. با این حال اکنون عنصر *largest* حاوی مقدار اولیه‌ی $A[i]$ است، و بنابراین زیردرخت با ریشه‌ی *largest* ممکن است از خصوصیت هرم‌های بیشینه پیروی نکند. در نتیجه MAX-HEAPIFY باید به صورت بازگشتی بر روی آن زیردرخت فراخوانی شود.

زمان اجرای MAX-HEAPIFY برروی یک زیردرخت با اندازه‌ی $n$ (که ریشه‌ی آن گره‌ی $i$ است) برابر است با $\theta(1)$ برای درست کردن رابطه‌ی عناصر $A[i]$، $A[LEFT(i)]$، و $A[RIGHT(i)]$، به علاوه‌ی زمان اجرای MAX-HEAPIFY برروی زیردرختی که ریشه‌ی آن یکی از فرزندان گره‌ی $i$ است (با فرض این که این فراخوانی بازگشتی رخ می‌دهد). اندازه‌ی زیردرخت‌های مربوط به دو فرزند حداکثر برابر است با $2n/3$ - بدترین حالت زمانی اتفاق می‌افتد که دقیقاً نصف ردیف آخر

درخت پر باشد – و بنابراین زمان اجرای MAX-HEAPIFY را می‌توان به صورت رابطه‌ی بازگشتی زیر نشان داد:

$$T(n) \le T(2n/3) + \theta(1)$$

بنابر حالت دو از قضیه‌ی اصلی (قضیه‌ی ۴-۱)، جواب این رابطه‌ی بازگشتی برابر است با $T(n) = O(\lg n)$. به عبارت دیگر می‌توان زمان اجرای MAX-HEAPIFY را روی یک گره با ارتفاع $h$ به صورت $O(h)$ توصیف کرد.

## تمرین‌ها

۶-۲-۱ با استفاده از شکل ۶-۲ به عنوان یک مدل، عملیات MAX-HEAPIFY$(A, 3)$ را روی آرایه‌ی $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$ نشان دهید.

۶-۲-۲ با استفاده از رویه‌ی MAX-HEAPIFY، شبه‌کدی برای رویه‌ی MIN-HEAPIFY$(A, i)$ بنویسید که عملیات مربوط را بر روی یک هرم کمینه انجام می‌دهد. سپس زمان اجرای MIN-HEAPIFY را با MAX-HEAPIFY مقایسه کنید.

۶-۲-۳ اگر $A[i]$ از فرزندان خود بزرگ‌تر باشد، تأثیر فراخوانی MAX-HEAPIFY$(A, i)$ چیست؟

۶-۲-۴ تأثیر فراخوانی MAX-HEAPIFY$(A, i)$ برای $i > A.heap\text{-}size/2$ چیست؟

۶-۲-۵ کد مربوط به MAX-HEAPIFY از نظر اندازه‌ی ضرایب ثابت کاملاً کارامد است، غیر از احتمالاً فراخوانی بازگشتی در خط ۱۰، که ممکن است باعث شود بعضی کامپایلرها کد ناکارامد (inefficient) تولید کنند. یک MAX-HEAPIFY کارامد بنویسید که از یک ساختار کنترلی تکراری (حلقه) به جای ساختار بازگشتی استفاده می‌کند.

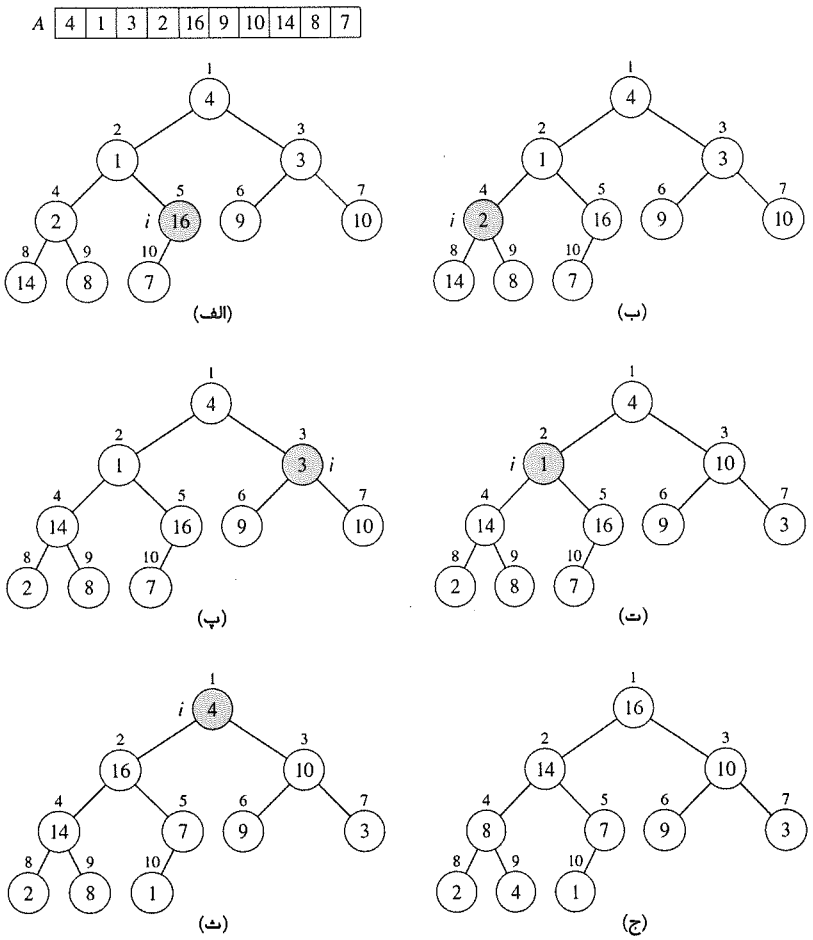
۶-۲-۶ نشان دهید که بدترین حالت زمان اجرای MAX-HEAPIFY بر روی یک هرم با اندازه‌ی $n$ از مرتبه‌ی $\Omega(\lg n)$ است. (*راهنمایی:* مقادیری به گره‌های یک هرم با اندازه‌ی $n$ بدهید که باعث شود MAX-HEAPIFY به صورت بازگشتی بر روی تمام گره‌ها در یک مسیر ساده از ریشه تا یک برگ فراخوانی شود.)

## ۶-۳ ساختن هرم

برای تبدیل آرایه‌ی $A[1 .. n]$، که در آن $n = A.length$، به یک هرم بیشینه می‌توانیم از رویه‌ی MAX-HEAPIFY به صورت پایین به بالا استفاده کنیم. طبق تمرین ۶-۱-۷ عناصر درون زیرآرایه‌ی $A[(\lfloor n/2 \rfloor + 1) .. n]$ برگ‌های درخت هستند، و بنابراین هر کدام یک هرم با یک عنصر هستند که می‌توانیم از آن‌ها شروع کنیم. رویه‌ی BUILD-MAX-HEAP از روی بقیه‌ی گره‌های درخت عبور کرده و MAX-HEAP را بر روی هر یک اجرا می‌کند.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

شکل ۶-۳ مثالی از عملیات BUILD-MAX-HEAP را نشان می‌دهد.




**شکل ۶-۳** عملیات BUILD-MAX-HEAP، که ساختمان داده را قبل از فراخوانی MAX-HEAPIFY در خط ۳ نشان می‌دهد. (الف) یک آرایه‌ی ۱۰ عنصری $A$ و درخت دودویی معادل آن. در شکل مشخص شده است که اندیس $i$ قبل از فراخوانی MAX-HEAPIFY $(A, i)$ به گره‌ی پنجم اشاره می‌کند. (ب) ساختمان داده پس از فراخوانی اول. در تکرار بعدی، اندیس $i$ به گره‌ی ۴ اشاره می‌کند. (پ)-(ث) تکرارهای بعدی حلقه‌ی for در MAX-HEAPIFY. دقت کنید که هر زمان MAX-HEAPIFY بر روی یک گره فراخوانی می‌شود، هر دو زیردرخت چپ و راست آن گره هرم بیشینه هستند. (ج) هرم بیشینه پس از آن که BUILD-MAX-HEAP پایان می‌یابد.

برای این که نشان دهیم BUILD-MAX-HEAP به درستی کار می‌کند، از ثابت حلقه‌ی زیر استفاده می‌کنیم:

- در شروع هر تکرار حلقه‌ی for در خطوط ۲-۳، هر یک از گره‌های $i+1, i+2, \ldots, n$ ریشه‌ی یک هرم بیشینه است.

باید نشان دهیم که این ثابت حلقه قبل از اولین تکرار حلقه برقرار است، هر یک از تکرارهای حلقه آن را حفظ می‌کند، و پس از اتمام حلقه، این ثابت حلقه خصوصیتی مفید برای اثبات درستی به ما می‌دهد.

- شروع: قبل از شروع اولین تکرار حلقه داریم $i = \lfloor n/2 \rfloor$. هر یک از گره‌های $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ یک برگ است و بنابراین می‌تواند ریشه‌ی یک هرم بیشینه باشد.
- ادامه: برای این که نشان دهیم هر تکرار حلقه، ثابت حلقه را برقرار نگه می‌دارد، مشاهده کنید که فرزندان گره‌ی $i$، شماره‌ای بزرگ‌تر از $i$ دارند. بنابراین طبق ثابت حلقه هر دو هرم بیشینه هستند. این دقیقاً شرط لازم برای این است که MAX-HEAPIFY$(A,i)$، گره‌ی $i$ را تبدیل به ریشه‌ی یک هرم بیشینه کند. به علاوه فراخوانی MAX-HEAPIFY خصوصیت هرم‌های بیشینه را روی گره‌های $i+1$ و $i+2$ هم حفظ می‌کند. افزایش $i$ توسط حلقه‌ی for، ثابت حلقه را برای تکرار بعدی حلقه آماده می‌کند.
- پایان: پس از اتمام حلقه داریم $i = 0$. طبق ثابت حلقه هر یک از گره‌های $1, 2, \ldots, n$ ریشه‌ی یک هرم بیشینه هستند، و به خصوص، گره‌ی ۱.

به صورت زیر می‌توانیم یک کران بالای ساده برای زمان اجرای BUILD-MAX-HEAP محاسبه کنیم. هر فراخوانی MAX-HEAPIFY به زمان $O(\lg n)$ احتیاج دارد، و در BUILD-MAX-HEAP این فراخوانی $O(n)$ بار انجام می‌شود. بنابراین زمان اجرا از مرتبه‌ی $O(n \lg n)$ است. این کران بالا، با این که درست است ولی به صورت حدی نزدیک نیست.

برای به دست آوردن یک کران بالای نزدیک تر، مشاهده می‌کنیم که زمان اجرای MAX-HEAPIFY بر روی یک گره به ارتفاع آن گره بستگی دارد، و ارتفاع اکثر گره‌ها بسیار کم است. این تحلیل جدید، بر مبنای این خصوصیت است که ارتفاع یک هرم با $n$ عنصر، برابر است با $\lfloor \lg n \rfloor$ (تمرین ۶-۱-۲ را ببینید)، و حداکثر $\lceil n/2^{h+1} \rceil$ گره ارتفاع $h$ دارند (تمرین ۶-۳-۳ را ببینید).

زمان مورد نیاز برای اجرای MAX-HEAPIFY بر روی یک گره با ارتفاع $h$ برابر است با $O(h)$، و بنابراین می‌توانیم کران بالای هزینه‌ی کلی اجرای BUILD-MAX-HEAP را به صورت زیر تعریف کنیم:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left( n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

مجموع آخر را می‌توان با جایگذاری $x = ۱/۲$ در فرمول (الف-۸) به دست آورد، که می‌دهد:

$$\sum_{h=۰}^{\infty} \frac{h}{۲^h} = \frac{۱/۲}{(۱-۱/۲)^۲} = ۲$$

بنابراین، کران بالای زمان اجرای BUILD-MAX-HEAP برابر است با:

$$O\left(n \sum_{h=۰}^{\lfloor \lg n \rfloor} \frac{h}{۲^n}\right) = O\left(n \sum_{h=۰}^{\infty} \frac{h}{۲^h}\right) = O(n)$$

از این رو می‌توانیم در زمان خطی، یک آرایه‌ی نامرتب را به هرم بیشینه تبدیل کنیم.

برای ساختن یک هرم کمینه می‌توانیم از رویه‌ی BUILD-MIN-HEAP استفاده کنیم، که مشابه BUILD-MAX-HEAP است، و فقط در خط ۳، فراخوانی MAX-HEAPIFY با فراخوانی MIN-HEAPIFY جایگزین شده است (تمرین ۶-۲-۲ را ببینید). رویه‌ی BUILD-MIN-HEAP از یک آرایه‌ی نامرتب در زمان خطی یک هرم کمینه می‌سازد.

## تمرین‌ها

۶-۳-۱ با استفاده از شکل ۶-۳ به عنوان یک مدل، عملیات BUILD-MAX-HEAP را بر روی آرایه‌ی $A = \langle ۵,۳,۱۷,۱۰,۸۴,۱۹,۶,۲۲,۹ \rangle$ نشان دهید.

۶-۳-۲ چرا در خط ۲ از رویه‌ی BUILD-MAX-HEAP، اندیس $i$ به جای این که از ۱ تا $\lfloor A.length/۲ \rfloor$ افزایش یابد، از $\lfloor A.length/۲ \rfloor$ تا ۱ کاهش می‌یابد؟

۶-۳-۳ نشان دهید که در هر هرم با $n$ عنصر، حداکثر $\lceil n/۲^{h+۱} \rceil$ گره با ارتفاع $h$ وجود دارد.

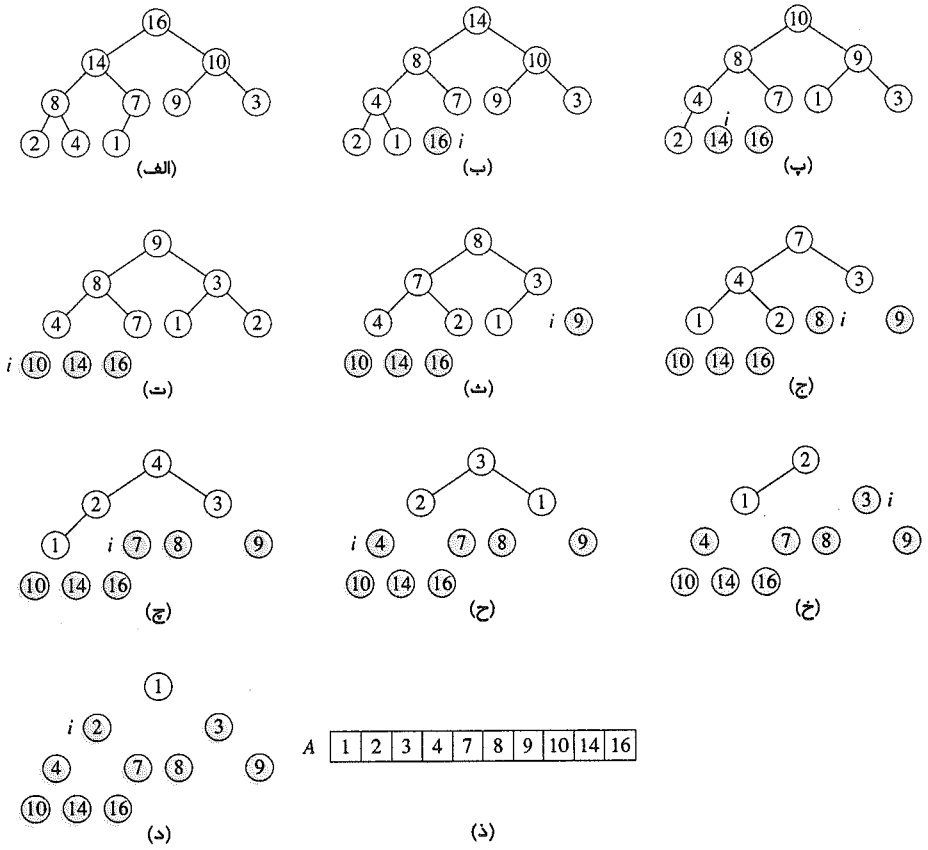
## ۶-۴ الگوریتم مرتب‌سازی هرمی

الگوریتم مرتب‌سازی هرمی با ساختن یک هرم بیشینه از آرایه‌ی ورودی $A[۱ .. n]$ به وسیله‌ی BUILD-MAX-HEAP شروع می‌کند، که در آن $n = A.length$. از آن جایی که عنصر بیشینه‌ی هرم در ریشه‌ی آن، یعنی $A[۱]$ ذخیره شده است، می‌توان با عوض کردن جای عناصر $A[۱]$ و $A[n]$ آن را در جای نهایی خود قرار داد. اگر اکنون (با کاهش $A.heap\text{-}size$) گره‌ی $n$ را از هرم «حذف» کنیم، مشاهده می‌کنیم که فرزندان ریشه هرم بیشینه باقی می‌مانند، ولی ریشه‌ی جدید ممکن است خصوصیت هرم‌های بیشینه را نقض کند. با این حال تنها کار مورد نیاز برای برقرار کردن دوباره‌ی خصوصیت هرم‌های بیشینه فراخوانی MAX-HEAPIFY $(A, ۱)$ است، که تمام زیرآرایه‌ی $A[۱ .. (n-۱)]$

را به یک هرم بیشینه تبدیل می‌کند. سپس این فرایند برای هرم بیشینه با اندازه‌ی $n-1$ تکرار می‌شود تا زمانی که یک هرم با اندازه‌ی ۲ باقی بماند. (برای یک ثابت حلقه‌ی دقیق، تمرین ۶-۴-۲ را ببینید.)

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size – 1
5      MAX-HEAPIFY(A, 1)
```

شکل ۶-۴ مثالی از عملیات مرتب‌سازی هرمی را پس از ساخت اولیه‌ی هرم بیشینه از آرایه نشان می‌دهد. هر هرم بیشینه در ابتدای حلقه‌ی for در خطوط ۲-۵ نشان داده شده است.




شکل ۶-۴ عملیات HEAPSORT. (الف) ساختمان داده‌ی هرم بیشینه دقیقاً پس از ساخته شدن توسط رویه‌ی BUILD-MAX-HEAP. (ب)-(د) هرم بیشینه دقیقاً پس از فراخوانی MAX-HEAPIFY در خط ۵. مقدار $i$ در هر شکل نشان داده شده است. فقط گره‌هایی که سایه‌ی روشن دارند هنوز در هرم هستند. (ذ) آرایه‌ی مرتب شده‌ی نهایی.

اجرای رویه‌ی HEAPSORT به زمان $O(n \lg n)$ نیاز دارد، چرا که فراخوانی BUILD-MAX-HEAP در زمان $O(n)$ انجام می‌شود، و هر یک از $n-1$ فراخوانی MAX-HEAPIFY در زمان $O(\lg n)$.

## تمرین‌ها

۶-۴-۱ با استفاده از شکل ۶-۴ به عنوان یک مدل، عملیات HEAPSORT را بر روی آرایه‌ی $A = \langle 5,13,2,25,7,17,20,8,4 \rangle$ نشان دهید.

۶-۴-۲ با استفاده از ثابت حلقه‌ی زیر، درستی HEAPSORT را اثبات کنید:

- در شروع هر تکرار حلقه‌ی for در خطوط ۲-۵، زیرآرایه‌ی $A[1..i]$ یک هرم بیشینه شامل $i$ عنصر کوچک $A[1..n]$ است، و زیرآرایه‌ی $A[i+1..n]$ حاوی $n-i$ عنصر بزرگ $A[1..n]$، به صورت مرتب شده.

۶-۴-۳ زمان اجرای مرتب‌سازی هرمی بر روی یک آرایه‌ی $A$ با طول $n$ که به صورت صعودی مرتب شده است، چقدر است؟ اگر به صورت نزولی مرتب شده باشد چطور؟

۶-۴-۴ نشان دهید که بدترین حالت زمان اجرای مرتب‌سازی هرمی از مرتبه‌ی $\Omega(n \lg n)$ است.

۶-۴-۵★ نشان دهید که وقتی تمام عناصر یکتا باشند، بهترین حالت زمان اجرای مرتب‌سازی هرمی از مرتبه‌ی $\Omega(n \lg n)$ است.

## ۶-۵ صف‌های اولویت

مرتب‌سازی هرمی یک الگوریتم بسیار مناسب است، ولی یک پیاده‌سازی خوب از مرتب‌سازی سریع، که در فصل ۷ معرفی می‌شود، معمولاً در عمل کارامدتر است. با این حال خود ساختمان داده‌ی هرم دارای کاربردهای فراوانی است. در این بخش یکی از مهم‌ترین کاربردهای هرم را معرفی می‌کنیم: استفاده به عنوان یک صف اولویت کارامد. مانند هرم‌ها دو نوع صف اولویت وجود دارد: صف‌های اولویت بیشینه و صف‌های اولویت کمینه. در این جا بر روی پیاده‌سازی صف‌های اولویت بیشینه تمرکز می‌کنیم، که بر پایه‌ی هرم‌های بیشینه بنا شده‌اند؛ تمرین ۶-۵-۳ از شما می‌خواهد که یک رویه برای صف‌های اولویت کمینه بنویسید.

*صف اولویت* (priority queue) ساختمان داده‌ای است برای نگه داری $S$ عنصر، که هر کدام یک مقدار مربوطه به نام *کلید* (key) دارند. یک *صف اولویت بیشینه* از عملیات زیر پشتیبانی می‌کند.

- INSERT$(S,x)$ عنصر $x$ را در مجموعه‌ی $S$ درج (insert) می‌کند. این عملیات را می‌توان به صورت $S = S \cup \{x\}$ نوشت.
- MAXIMUM$(S,x)$ عنصری از $S$ را که بزرگ‌ترین کلید را دارد، بازمی‌گرداند.
- EXTRAXT-MAX$(S)$ عنصری از $S$ را که بزرگ‌ترین کلید را دارد، بازگردانده و آن را از $S$ حذف می‌کند.

INCREASE-KEY $(S, x, k)$ مقدار کلید عنصر $x$ را به مقدار جدید $k$ افزایش می‌دهد، که فرض می‌شود مقدار $k$ حداقل به بزرگی مقدار فعلی کلید $x$ است.

یکی از کاربردهای صف‌های اولویت بیشینه، زمان‌بندی کارها (job scheduling) بر روی یک کامپیوتر مشترک می‌باشد. کارهایی که باید انجام شوند به همراه اولویت مربوط به آن‌ها در صف اولویت بیشینه ذخیره می‌شوند. وقتی یک کار تمام و یا قطع می‌شود، با اولویت‌ترین کار به وسیله‌ی EXTRACT-MAX از میان متقاضیان انتخاب می‌شود. یک کار جدید، در هر زمانی با استفاده از INSERT می‌تواند به لیست (صف) کارها افزوده شود.

به طور مشابه، یک صف اولویت کمینه از عملیات INSERT، MINIMUM، EXTRACT-MIN، و DECREASE-KEY پشتیبانی می‌کند. از صف اولویت کمینه می‌توان در یک شبیه‌ساز رویداد (event-driven simulator) استفاده کرد. عناصر درون صف، رویدادهایی هستند که باید شبیه‌سازی شوند، و هر کدام یک زمان وقوع دارند که به عنوان کلید در صف ذخیره می‌شود. رویدادها باید به ترتیب زمان وقوع شبیه‌سازی شوند، چرا که شبیه‌سازی یک رویداد می‌تواند باعث شود بقیه‌ی رویدادها در آینده اتفاع بیفتند (شبیه‌سازی شوند). در هر مرحله برنامه‌ی شبیه‌ساز برای انتخاب رویداد بعدی برای شبیه‌سازی از EXTRACT-MIN استفاده می‌کند. وقتی رویدادهای جدید ساخته می‌شوند، با استفاده از INESRT در صف اولویت کمینه درج می‌شوند. در فصل ۲۳ و ۲۴ استفاده‌های دیگری از صف‌های اولویت کمینه را خواهیم دید که به عملیات DECREASE-KEY هم نیاز دارند.

همان طور که انتظار می‌رود، می‌توانیم برای ساختن صف اولویت از هرم‌ها استفاده کنیم. در یک کاربرد خاص، مانند زمان‌بندی کارها و یا شبیه‌سازی رویدادها، عناصر درون صف اولویت متناظر با اشیاء مربوط به عملیات هستند. معمولاً مهم است که بدانیم هر کدام از اشیاء درون کاربرد متناظر با کدام یک از عناصر درون صف است، و بالعکس. بنابراین وقتی از یک هرم برای پیاده‌سازی یک صف اولویت استفاده می‌کنیم، معمولاً نیاز داریم یک دستگیره (handle) که به شیئ مربوطه در کاربرد اشاره می‌کند در هر یک از عناصر صف ذخیره کنیم. نوع دقیق دستگیره (اشاره گر، عدد صحیح و ...) به آن کاربرد خاص بستگی دارد. به طور مشابه، باید یک دستگیره که به عناصر مربوطه در هرم اشاره می‌کند، در اشیاء ذخیره کنیم. در این جا، این دستگیره می‌تواند یک اندیس آرایه باشد. از آن جایی که عناصر هرم هنگام انجام عملیات روی هرم جای خود را تغییر می‌دهند، در یک کاربرد واقعی با هر بار جابه‌جایی یک عنصر در هرم باید اندیس آرایه در شیئ مربوطه نیز به روز رسانی شود. از آن جایی که جزئیات دسترسی به اشیاء به شدت به خود عملیات و نوع پیاده‌سازی آن وابسته است، در این جا بیشتر از این آن را دنبال نمی‌کنیم. فقط باید بدانیم که در عمل باید از دستگیره‌ها به طور صحیح محافظت شود.

اکنون در مورد روش پیاده‌سازی عملیات بر روی یک صف اولویت بیشینه بحث خواهیم کرد. رویه‌ی HEAP-MAXIMUM، عملیات MAXIMUM را در زمان $\theta(1)$ پیاده‌سازی می‌کند.

```
HEAP-MAXIMUM(A)
1  return A[1]
```

رویه‌ی HEAP-EXTRACT-MAX عملیات EXTRACT-MAX را پیاده‌سازی می‌کند. این رویه مشابه بدنه‌ی حلقه‌ی for (خطوط ۳-۵) در رویه‌ی HEAPSORT است.

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2      error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size - 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

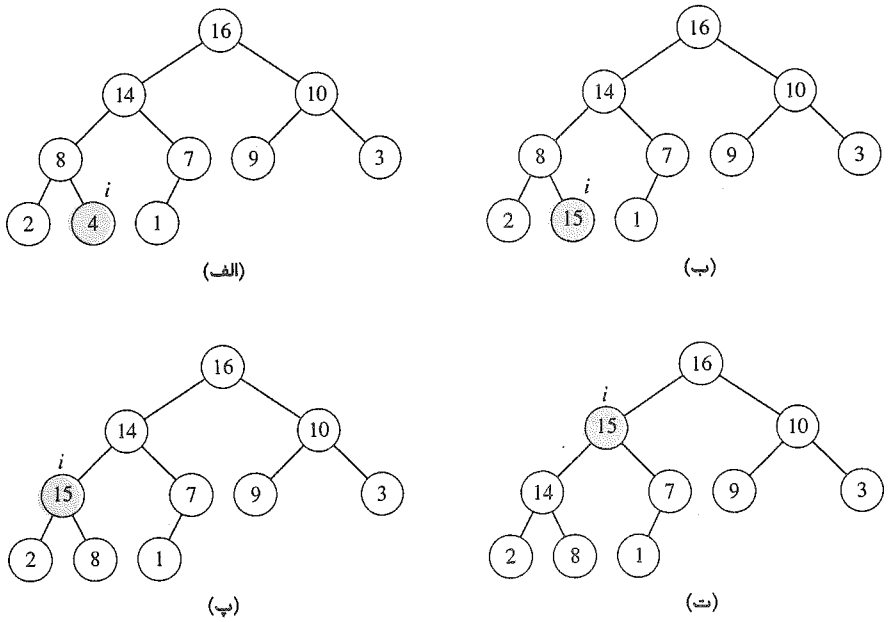
زمان اجرای HEAP-EXTRACT-MAX از مرتبه‌ی $O(\lg n)$ است، زیرا فقط تعداد ثابتی عملیات قبل از MAX-HEAPIFY انجام می‌دهد، که آن هم به زمان $O(\lg n)$ نیاز دارد.

رویه‌ی HEAP-INSERT-KEY عملیات INSERT-KEY را پیاده‌سازی می‌کند. عنصری از صف اولویت که باید افزایش یابد با اندیس $i$ درون آرایه مشخص می‌شود. این رویه ابتدا مقدار عنصر $A[i]$ را به مقدار جدید خود افزایش می‌دهد. سپس از آن جایی که افزایش کلید $A[i]$ ممکن است خصوصیت هرم‌های بیشینه را از بین ببرد، رویه به شکلی که یادآور حلقه‌ی درج (خطوط ۵-۷) از رویه‌ی INSERTION-SORT است، در یک مسیر از گره تا ریشه به دنبال مکان مناسب برای کلید افزایش یافته می‌گردد. در هنگام این جستجو، عنصر مربوطه مرتباً با پدر خود مقایسه می‌شود، که در صورتی که کلید عنصر فرزند بزرگ‌تر بود، جای عناصر پدر و فرزند عوض شده و جستجو ادامه می‌یابد، و اگر کلید عصر فرزند کوچک‌تر بود، جستجو خاتمه می‌یابد، چرا که در این صورت خصوصیت هرم‌های بیشینه حفظ شده است. (تمرین ۶-۵-۵ را برای یک ثابت حلقه‌ی دقیق ببینید.)

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

شکل ۶-۵ یک مثال از اجرای عملیات HEAP-INCREASE-KEY را نشان می‌دهد. زمان اجرای HEAP-INCREASE-KEY روی یک هرم با $n$ عنصر برابر است با $O(\lg n)$، چرا که طول مسیر طی شده از گره‌ی با مقدار جدید در خط ۳ تا ریشه $O(\lg n)$ است.

رویه‌ی MAX-HEAP-INSERT عملیات INSERT را پیاده‌سازی می‌کند. ورودی رویه کلیدی است که باید در هرم بیشینه‌ی $A$ درج شود. این رویه ابتدا با اضافه کردن یک گره با کلید $-\infty$، هرم بیشینه را گسترش می‌دهد. سپس HEAP-INCREASE-KEY را بر روی آن گره فراخوانی می‌کند تا مقدار مورد نظر را به آن اختصاص دهد و خصوصیت هرم‌های بیشینه را نیز برقرار سازد.




شکل ۶-۵ عملیات HEAP-INCREASE-KEY. (الف) هرم بیشینه‌ی شکل ۶-۴(الف) با یک گره که اندیس $i$ آن به صورت پر رنگ سایه زده شده است. (ب) مقدار کلید این گره به ۱۵ افزایش یافته است. (پ) پس از یک بار تکرار حلقه‌ی while در خطوط ۴-۶، جای کلیدهای این گره و پدر آن عوض می‌شود و اندیس $i$ به سمت بالا حرکت می‌کند. (ت) هرم بیشینه پس از یک تکرار دیگر از حلقه‌ی while. در این لحظه، $A[PARENT(i)] \geq A[i]$. اکنون، خصوصیت هرم‌های بیشینه برقرار است و رویه پایان می‌یابد.

MAX-HEAP-INSERT($A$, $key$)
1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY($A$, $A.heap\text{-}size$, $key$)

MAX-HEAP-INSERT بر روی یک هرم با $n$ عنصر در زمان $O(\lg n)$ اجرا می‌شود.

در مجموع یک هرم می‌تواند هر یک از اعمال مورد نیاز یک صف اولویت با اندازه‌ی $n$ را در زمان $O(\lg n)$ انجام دهد.

## تمرین‌ها

۶-۵-۱ عملیات HEAP-EXTRACT-MAX را بر روی $A = \langle ۱۵,۱۳,۹,۵,۱۲,۸,۷,۴,۰,۶,۲,۱ \rangle$ مشخص کنید.

۶-۵-۲ عملیات MAX-HEAP-INSERT$(A, ۱۰)$ را بر روی $A = \langle ۱۵,۱۳,۹,۵,۱۲,۸,۷,۴,۰,۶,۲,۱ \rangle$ نشان دهید.

۶-۵-۳ برای رویه‌های HEAP-MINIMUM، HEAP-EXTRACT-MIN، HEAP-DECREASE-KEY و MIN-HEAP-INSERT که اعمال یک صف اولیت کمینه را روی یک هرم کمینه پیاده‌سازی می‌کنند، شبه‌کد بنویسید.

۶-۵-۴ در خط ۲ از MAX-HEAP-INSERT، چرا به خود زحمت داده و ابتدا گره را با $-\infty$ مقداردهی می‌کنیم، در حالی که کار بعدی که می‌کنیم، افزایش آن به مقدار مورد نظر است؟

۶-۵-۵ با استفاده از ثابت حلقه‌ی زیر درستی HEAP-INCREASE-KEY را اثبات کنید:

در آغاز هر بار تکرار حلقه‌ی while در خطوط ۴-۶، آرایه‌ی $A[1 .. A.heap\text{-}size]$ خصوصیت هرم‌های بیشینه را دارد، غیر از این که ممکن است یک جا این خصوصیت نقض شود: ممکن است $A[i]$ از $A[PARENT(i)]$ بزرگ‌تر باشد.
می‌توانید فرض کنید که زیرآرایه‌ی $A[1 .. A.heap\text{-}size]$ در زمان فراخوانی HEAP-INCREASE-KEY خصوصیات هرم‌های بیشینه را حفظ می‌کند.

۶-۵-۶ هر عملیات جابه‌جایی در خط ۵ از رویه‌ی HEAP-INCREASE-KEY معمولاً به سه مقداردهی نیاز دارد. نشان دهید چطور می‌توان با استفاده از ایده‌ی حلقه‌ی داخلی INSERTION-SORT، این کار را فقط با استفاده از یک مقداردهی انجام داد.

۶-۵-۷ نشان دهید چطور می‌توان به وسیله‌ی یک صف اولویت، یک صف FIFO ساخت. همچنین نشان دهید که چگونه می‌توان به وسیله‌ی یک صف اولویت یک پشته ساخت. (صف‌های FIFO و پشته‌ها در بخش ۱۰-۱ معرفی می‌شوند.)

۶-۵-۸ عملیات HEAP-DELETE $(A, i)$ عنصر درون گره‌ی $i$ را از هرم $A$ حذف می‌کند. یک پیاده‌سازی از HEAP-DELETE ارائه کنید که برای یک هرم بیشینه با $n$ عنصر در زمان $O(\lg n)$ اجرا می‌شود.

۶-۵-۹ یک الگوریتم با زمان اجرای $O(n \lg k)$ بدهید که $k$ لیست مرتب شده را درون یک لیست مرتب شده ادغام می‌کند، که در آن $n$ برابر است با مجموع عناصر در تمام لیست‌های ورودی. (*راهنمایی:* از یک هرم کمینه برای ادغام $k$ تایی استفاده کنید.)

## مسائل

۶-۱ ***ساختن یک هرم با استفاده از درج***

رویه‌ی BUILD-MAX-HEAP در بخش ۶-۳ را می‌توان به وسیله‌ی استفاده‌ی مکرر از MAX-HEAP-INSERT برای درج عناصر در هرم پیاده‌سازی کرد. پیاده‌سازی زیر را در نظر بگیرید:

```
BUILD-MAX-HEAP'(A)
1  A.heap-size = 1
2  for i = 2 to A.length
3      MAX-HEAP-INSERT(A, A[i])
```

I. آیا رویه‌های BUILD-MAX-HEAP و BUILD-MIN-HEAP وقتی بر روی یک آرایه‌ی ورودی خاص اجرا می‌شوند، همیشه خروجی یکسانی دارند؟ اثبات کنید که خروجی آن‌ها یکسان است، و یا یک مثال نقض برای آن بیاورید.

II. نشان دهید که BUILD-MAX-HEAP برای ساختن یک هرم با $n$ عنصر، در بدترین حالت به زمان $\theta(n \lg n)$ نیاز دارد.

### ۶-۲ تحلیل هرم‌های $d$تایی

یک هرم *$d$تایی* مانند یک هرم دودویی است، فقط (با یک استثنای احتمالی) گره‌های غیر برگ دارای $d$ فرزند به جای ۲ فرزند هستند.

I. چگونه می‌توان یک هرم $d$تایی را در یک آرایه نمایش داد؟

II. ارتفاع یک هرم $d$تایی با $n$ عنصر بر حسب $n$ و $d$ چقدر است؟

III. یک پیاده‌سازی کارامد از EXTRACT-MAX برای یک هرم بیشینه‌ی $d$تایی بدهید. سپس زمان اجرای آن را بر حسب $d$ و $n$ تحلیل کنید.

IV. یک پیاده‌سازی کارامد از INSERT برای یک هرم بیشینه‌ی $d$تایی بدهید. سپس زمان اجرای آن را بر حسب $d$ و $n$ تحلیل کنید.

V. یک پیاده‌سازی کارامد از INCREASE-KEY$(A,i,k)$ بدهید، که اگر $k < A[i]$ بود پیغام خطا می‌دهد، و در غیر این صورت قرار می‌دهد $A[i] = k$، و سپس ساختار هرم بیشینه‌ی $d$تایی را به شکل مناسب به روز رسانی می‌کند. زمان اجرای آن را بر حسب $d$ و $n$ تحلیل کنید.

### ۶-۳ جداول یانگ (Young)

یک *جدول یانگ* $m \times n$، یک ماتریس $m \times n$ است به طوری که ورودی‌های هر یک از ردیف‌ها از چپ به راست، و ورودی‌های هر یک از ستون‌ها از بالا به پایین مرتب شده‌اند. بعضی از ورودی‌های یک جدول یانگ ممکن است $\infty$ باشند، که فرض می‌کنیم آن عناصر وجود ندارند. بنابراین از یک جدول یانگ می‌توان برای نگه داری $r \le mn$ عدد کران دار استفاده کرد.

I. یک جدول ۴×۴ یانگ بکشید که حاوی عناصر $\{9,16,3,2,4,8,5,14,12\}$ باشد.

II. نشان دهید که یک جدول یانگ $m \times n$ در صورتی خالی است که $Y[1,1] = \infty$، و در صورتی پر است (حاوی $mn$ عنصر است) که $Y[m,n] < \infty$.

III. یک الگوریتم برای پیاده‌سازی EXTRACT-MIN بر روی یک جدول یانگ $m \times n$ ناتهی بدهید که در زمان $O(m+n)$ اجرا می‌شود. الگوریتم شما باید از یک زیرروال استفاده کند که برای حل یک مسئله‌ی $m \times n$ به صورت بازگشتی یا یک مسئله‌ی $(m-1) \times n$ و یا یک مسئله‌ی $m \times (n-1)$ را حل می‌کند. (*راهنمایی:* MAX-HEAPIFY را به یاد بیاورید.) $T(p)$ را

تعیین کنید، که در آن $p = m + n$، حداکثر زمان اجرای EXTRACT-MIN روی هر جدول یانگ $m \times n$ است. یک رابطه‌ی بازگشتی برای $T(p)$ ارائه و آن را حل کنید. سپس کران $O(m+n)$ را بر روی آن به دست آورید.

IV. نشان دهید که چگونه می‌توان یک عنصر جدید را در یک جدول یانگ $m \times n$ ناپر در زمان $O(m+n)$ درج کرد.

V. بدون استفاده از متدی جدا برای مرتب‌سازی، نشان دهید که چگونه می‌توان برای مرتب‌سازی $n^2$ عدد در زمان $O(n^3)$، از یک جدول یانگ $n \times n$ استفاده کرد.

VI. یک الگوریتم با زمان اجرای $O(m+n)$ بدهید که تشخیص می‌دهد آیا یک عدد خاص در یک جدول یانگ داده شده وجود دارد آیا خیر.


# مرتب‌سازی سریع

مرتب‌سازی سریع یک الگوریتم مرتب‌سازی است که بدترین حالت زمان اجرای آن بر روی یک آرایه‌ی ورودی با اندازه‌ی $n$ از مرتبه‌ی $\theta(n^2)$ است. بر خلاف این بدترین حالت زمان اجرای کند، این الگوریتم معمولاً در عمل بهترین انتخاب برای مرتب‌سازی است، چرا که در حالت متوسط بسیار کارامد است: امیدریاضی زمان اجرای آن برابر است با $\theta(n \lg n)$، و ضرایب ثابت مخفی آن بسیار کوچک هستند. همچنین این الگوریتم دارای مزیت مرتب‌سازی در جا است (بخش ۲-۱ را ببینید)، و حتی در محیط‌های با حافظه‌ی مجازی به خوبی کار می‌کند.

بخش ۷-۱ الگوریتم را توضیح می‌دهد، به علاوه‌ی یک زیرروال مهم که مرتب‌سازی سریع از آن برای تقسیم‌بندی استفاده می‌کند. از آن جایی که رفتار مرتب‌سازی سریع پیچیده است، در بخش ۷-۱ با یک بحث شهودی در مورد کارایی آن شروع می‌کنیم، و تحلیل دقیق آن را به پایان فصل موکول می‌کنیم. در بخش ۷-۳ نسخه‌ای از مرتب‌سازی سریع معرفی می‌شود که از نمونه‌گیری تصادفی استفاده می‌کند. این الگوریتم زمان اجرای متوسط بسیار خوبی دارد، و هیچ ورودی خاصی باعث وقوع بدترین حالت زمان اجرای آن نمی‌شود. تحلیل این الگوریتم تصادفی را در بخش ۷-۴ خواهیم دید، و در آن جا نشان می‌دهیم که زمان اجرای آن در بدترین حالت $\theta(n^2)$ است، با امیدریاضی $O(n \lg n)$.

## ۷-۱ تعریف مرتب‌سازی سریع

مرتب‌سازی سریع همانند مرتب‌سازی ادغامی بر پایه‌ی الگوی تقسیم و حل، که در بخش ۲-۳-۱ معرفی شد، بنا شده است. در زیر فرایند سه مرحله‌ای تقسیم و حل را برای مرتب‌سازی یک زیرآرایه‌ی معمولی $A[p \,..\, r]$ می‌بینیم.

- **تقسیم:** آرایه‌ی $A[p \,..\, r]$ را به دو زیرآرایه‌ی (احتمالاً تهی) $A[p \,..\, q-1]$ و $A[q+1 \,..\, r]$ تقسیم (بازآرایی) کنید، به طوری که هر یک از عناصر $A[p \,..\, q-1]$ کوچک‌تر یا مساوی $A[q]$ باشند، که به نوبه‌ی خود کوچک‌تر یا مساوی هر یک از عناصر $A[q+1 \,..\, r]$ است. اندیس $q$ را هم به عنوان قسمتی از رویه‌ی تقسیم‌بندی محاسبه کنید.
- **حل:** دو زیرآرایه‌ی $A[p \,..\, q-1]$ و $A[q+1 \,..\, r]$ را با فراخوانی بازگشتی مرتب‌سازی سریع، مرتب کنید.
- **ادغام:** از آن جایی که زیرآرایه‌ها درجا مرتب می‌شوند، هیچ کاری برای ادغام آن‌ها نیاز نیست: اکنون تمام آرایه مرتب شده است.

رویه‌ی زیر، مرتب‌سازی سریع را پیاده‌سازی می‌کند.

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

برای مرتب‌سازی تمام آرایه‌ی $A$، فراخوانی اولیه QUICKSORT$(A, 1, A.length)$ خواهد بود.

## تقسیم‌بندی آرایه

کلید این الگوریتم رویه‌ی PARTITION است، که آرایه‌ی $A[p \,..\, r]$ را به صورت درجا بازآرایی می‌کند.

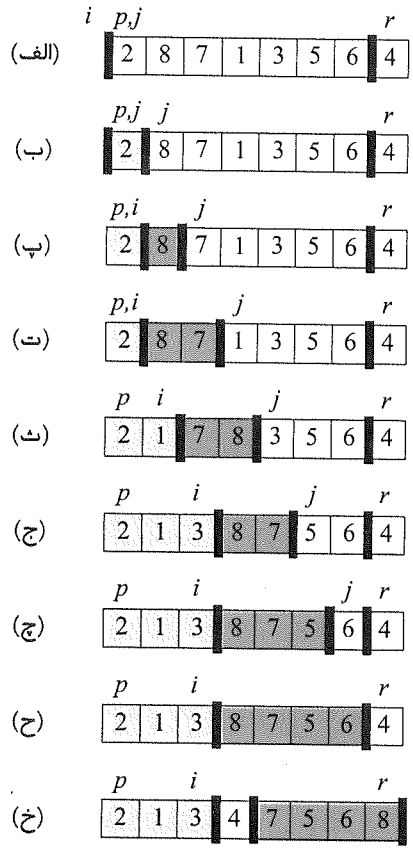
```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

شکل ۷-۱ عملیات PARTITION را بر روی یک آرایه‌ی ۸ عنصری نشان می‌دهد. این رویه همیشه یک عنصر $x = A[r]$ را به عنوان محور (pivot) انتخاب کرده، زیرآرایه‌ی $A[p \,..\, r]$ را حول آن بازآرایی می‌کند. پس از اجرای رویه، آرایه به چهار بخش (احتمالا تهی) تقسیم می‌شود. در شروع هر بار تکرار حلقه‌ی for در خطوط ۳-۶، هر منطقه دارای خصوصیاتی است که در شکل ۷-۲ نشان داده شده‌اند. این خصوصیات را در یک ثابت حلقه نشان می‌دهیم:

- در آغاز هر بار تکرار حلقه در خطوط ۳-۶، برای هر اندیس آرایه‌ی $k$،

۱. اگر $p \le k \le i$ آن گاه $A[k] \le x$.

۲. اگر $i+1 \le k \le j-1$ آن گاه $A[k] > x$.




**شکل ۷-۱** عملیات PARTITION در یک آرایه‌ی نمونه. عنصر $A[r]$، عنصر محور $x$ است. همه‌ی عناصری که به صورت کم رنگ سایه زده شده‌اند، در بخش اول هستند که مقادیر آن‌ها کم‌تر از $x$ است. عناصر پررنگ در بخش دوم هستند که مقادیر آن‌ها از $x$ بزرگ‌تر است. عناصر رنگ نشده هنوز در یکی از دو بخش قرار نگرفته‌اند، و عنصر سفید نهایی، عنصر محور است. (الف) آرایه‌ی اولیه و مقادیر متغیرها. هنوز هیچ یک از عناصر در یکی از دو بخش قرار نگرفته‌اند. (ب) جای مقدار ۲ «با خود عوض شده است» و در بخش مقادیر کوچک قرار گرفته است. (پ)-(ت) مقادیر ۷ و ۸ به بخش مقادیر بزرگ اضافه شده‌اند. (ث) جای مقادیر ۱ و ۸ عوض شده است، و بخش مقادیر کوچک رشد کرده است. (ج) جای مقادیر ۳ و ۸ عوض شده است، و بخش مقادیر کوچک رشد کرده است. (چ)-(ح) مقادیر ۵ و ۶ به بخش مقادیر بزرگ اضافه شده و حلقه پایان می‌یابد. (خ) در خطوط ۷-۸ جای عنصر محور عوض شده تا بین دو بخش قرار گیرد.

۳. اگر $k = r$ آن گاه $A[k] = x$.

اندیس‌های بین $j$ و $r-1$ توسط هیچ یک از سه حالت بالا پوشش داده نمی‌شوند، و مقادیر درون آن‌ها رابطه‌ی مشخصی با عنصر محوری $x$ ندارند.

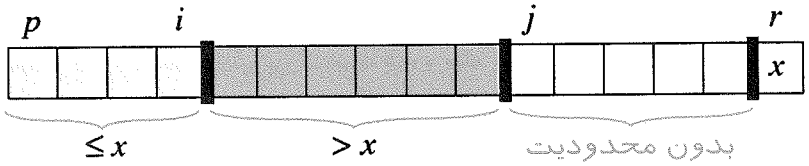
باید نشان دهیم که این ثابت حلقه قبل از شروع اولین تکرار حلقه برقرار است، با هر بار تکرار

حلقه برقرار می‌ماند، و پس از پایان اجرای حلقه، این ثابت حلقه خصوصیتی مفید برای اثبات درستی الگوریتم به ما می‌دهد.

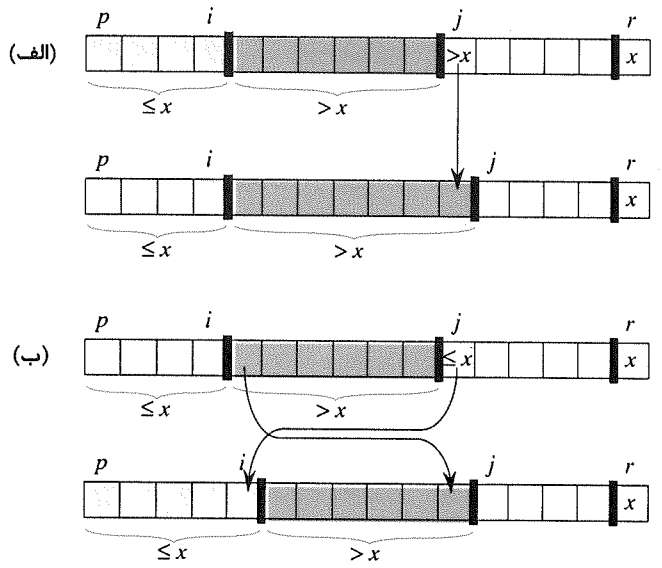
- آغاز: قبل از شروع اولین تکرار حلقه داریم $i = p-1$ و $j = p$. هیچ مقداری بین $p$ و $i$، و همچنین بین $i+1$ و $j-1$ وجود ندارد، و بنابراین دو شرط اول ثابت حلقه برقرار هستند. مقداردهی خط ۱ شرط سوم را برقرار می‌سازد.
- ادامه: همان طور که شکل ۷-۳ نشان می‌دهد، بسته به خروجی تست در خط ۴، باید دو حالت را در نظر بگیریم. شکل ۷-۳(الف) حالتی را نشان می‌دهد که $A[j] > x$؛ تنها کاری که در حلقه انجام می‌شود افزایش $j$ است. پس از این که $j$ افزایش یافت، شرط ۲ برای $A[j-1]$ برقرار خواهد بود، و تمام عناصر دیگر بدون تغییر باقی می‌مانند. شکل ۷-۳(ب) حالتی را نشان می‌دهد که $A[j] \le x$؛ $i$ افزایش می‌یابد، جای $A[i]$ و $A[j]$ عوض می‌شود، و سپس $j$ افزایش می‌یابد. اکنون در نتیجه‌ی جابه‌جایی داریم $A[i] \le x$، و شرط ۱ برقرار است. به طور مشابه داریم $A[j-1] > x$، چرا که عنصری که با $A[j-1]$ عوض شد، طبق ثابت حلقه از $x$ بزرگ‌تر است.
- پایان: در پایان داریم $j = r$. بنابراین تمام عناصر آرایه عضو یکی از سه مجموعه‌ای هستند که در ثابت حلقه تعریف شد. یعنی مقادیر درون آرایه به سه بخش تقسیم شده‌اند: مقادیری که کوچک‌تر یا مساوی $x$ هستند، مقادیری که بزرگ‌تر از $x$ هستند، و مجموعه‌ای تکی شامل مقدار $x$.

دو خط آخر PARTITION با جابه‌جا کردن عنصر محور با چپ‌ترین عنصر مجموعه‌ی عناصر بزرگ‌تر، آن را به مکان نهایی خود یعنی بین دو مجموعه منتقل می‌کند، و سپس اندیس جدید محور را بازمی‌گرداند. اکنون خروجی PARTITION خصوصیات مشخص شده در مرحله‌ی تقسیم را ارضا می‌کند. در واقع شرطی که خروجی این رویه ارضا می‌کند، کمی قوی‌تر است: پس از خط ۲ از QUICKSORT مقدار $A[q]$ اکیداً کوچک‌تر از تمام عناصر $A[q+1 \,..\, r]$ است.

زمان اجرای PARTITION بر روی زیرآرایه‌ی $A[p \,..\, r]$ برابر است با $\theta(n)$، که در آن $n = r - p + 1$ (تمرین ۷-۱-۳ را ببینید).




**شکل ۷-۳** چهار بخش تشکیل شده توسط رویه‌ی PARTITION بر روی زیرآرایه‌ی $A[p \,..\, r]$. تمام مقادیر درون $A[p \,..\, i]$ کوچک‌تر یا مساوی $x$ هستند، تمام مقادیر درون $A[i+1 \,..\, j-1]$ بزرگ‌تر از $x$ هستند، و $A[r] = x$. عناصر $A[j \,..\, r-1]$ می‌توانند هر مقداری داشته باشند.




**شکل ۷-۳** دو حالت برای یک تکرار از رویه‌ی PARTITION. (الف) اگر $A[j] > x$ تنها کاری که باید انجام دهیم افزایش $j$ است، که ثابت حلقه را برقرار نگه می‌دارد. (ب) اگر $A[j] \leq x$ اندیس $i$ افزایش می‌یابد، جای $A[i]$ و $A[j]$ عوض می‌شود، و آن گاه $j$ افزایش می‌یابد. دوباره ثابت حلقه برقرار است.

## تمرین‌ها

۷-۱-۱ با استفاده از شکل ۷-۱ به عنوان یک مدل، عملیات PARTITION را بر روی آرایه‌ی $A = \langle 13,19,9,5,12,8,7,4,11,2,6,21 \rangle$ مشخص کنید.

۷-۱-۲ اگر تمام عناصر آرایه‌ی $A[p \,..\, r]$ مقدار یکسانی داشته باشند، PARTITION چه مقداری را به عنوان $q$ بازمی‌گرداند؟ PARTITION را طوری اصلاح کنید که در چنین شرایطی طوری عمل کند که رابطه‌ی $q = (p+r)/2$ برقرار باشد.

۷-۱-۳ یک بحث مختصر بکنید که زمان اجرای PARTITION بر روی یک زیرآرایه با اندازه‌ی $n$ از مرتبه‌ی $\theta(n)$ است.

۷-۱-۴ اگر بخواهید QUICKSORT به صورت نزولی مرتب‌سازی را انجام دهد، آن را چگونه اصلاح می‌کنید؟

## ۷-۲ کارایی مرتب‌سازی سریع

زمان اجرای مرتب‌سازی سریع به متوازن بودن یا نبودن تقسیم‌بندی بستگی دارد، و این خود به عنصری بستگی دارد که به عنوان محور انتخاب می‌کنیم. اگر تقسیم‌بندی متوازن باشد، این الگوریتم به صورت حدی با سرعت مرتب‌سازی ادغامی اجرا می‌شود. ولی اگر تقسیم‌بندی نامتوازن باشد، ممکن است الگوریتم به صورت حدی به کندی مرتب‌سازی درجی اجرا شود. در این بخش به صورت غیررسمی عملکرد مرتب‌سازی سریع را در صورت متوازن بودن و یا نبودن تقسیم‌بندی بررسی خواهیم کرد.

### تقسیم‌بندی در بدترین حالت

بدترین حالت رفتار مرتب‌سازی سریع زمانی اتفاق می‌افتد که زیرروال تقسیم‌بندی، دو مسئله تولید کند یکی با اندازه‌ی $n-1$ و یکی با اندازه‌ی ۰. (این ادعا در بخش ۷-۴-۱ اثبات خواهد شد.) اجازه دهید فرض کنیم این نوع تقسیم‌بندی نامتوازن در تمام فراخوانی‌های بازگشتی اتفاق می‌افتد. هزینه‌ی تقسیم‌بندی $\theta(n)$ است. از آن جایی که فراخوانی بازگشتی بر روی یک آرایه با اندازه‌ی ۰ در همان جا پایان می‌یابد، $T(0) = \theta(1)$، و رابطه‌ی بازگشتی برای زمان اجرای کل رویه برابر است با:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \theta(n) \\ &= T(n-1) + \theta(n) \end{aligned}$$

به صورت شهودی، اگر هزینه‌ی تمام سطوح را با هم جمع کنیم، یک سری حسابی خواهیم داشت (تساوی (الف-۲)) که نتیجه‌ی آن برابر است با $\theta(n^2)$. در واقع به راحتی می‌توانیم از روش جانشین‌سازی استفاده کرده و نشان دهیم که جواب رابطه‌ی بازگشتی $T(n) = T(n-1) + \theta(n)$ برابر با $\theta(n^2)$ است. (تمرین ۷-۲-۱ را ببینید.)

بنابراین اگر در هر مرحله از فراخوانی بازگشتی در الگوریتم، تقسیم‌بندی در نامتوازن‌ترین حالت ممکن باشد، زمان اجرا $\theta(n^2)$ خواهد بود. از این رو بدترین حالت زمان اجرای مرتب‌سازی سریع از مرتب‌سازی درجی بهتر نخواهد بود. به علاوه زمان اجرای $\theta(n^2)$ زمانی اتفاق می‌افتد که آرایه‌ی ورودی از قبل مرتب شده باشد-موقعیتی معروف که در آن مرتب‌سازی درجی در زمان $\theta(n)$ اجرا می‌شود.

### تقسیم‌بندی در بهترین حالت

در بهترین تقسیم‌بندی ممکن، PARTITION دو زیرمسئله تولید می‌کند که اندازه‌ی هیچ کدام از $n/2$ بیشتر نیست، چرا که اندازه‌ی یکی $\lfloor n/2 \rfloor$ و اندازه‌ی دیگری $\lceil n/2 \rceil$ است. در این حالت مرتب‌سازی سریع بسیار سریع‌تر اجرا می‌شود. در این صورت رابطه‌ی بازگشتی برای زمان اجرا عبارت است از:

$$T(n) \le 2T(n/2) + \theta(n)$$

که در آن با در نظر نگرفتن کف و سقف و کم کردن ۱، مقداری بی‌دقتی داریم که از آن صرف نظر

می‌کنیم. طبق حالت ۲ از قضیه‌ی اصلی (قضیه‌ی ۴-۱) جواب این رابطه برابر است با $O(n \lg n)$. بنابراین توازن میان دو بخش در تقسیم‌بندی در هر سطح فراخوانی بازگشتی باعث می‌شود که الگوریتم به صورت حدی سریع‌تر اجرا شود.

## تقسیم‌بندی متوازن

زمان اجرای متوسط مرتب‌سازی سریع به بهترین حالت بسیار نزدیک‌تر است تا به بدترین حالت، همان طور که در تحلیل بخش ۷-۴ نشان داده خواهد شد. کلید درک این مسئله این است که بفهمیم چگونه توازن در تقسیم‌بندی در رابطه‌ی بازگشتی که زمان اجرا را توصیف می‌کند، نمود پیدا می‌کند.

برای مثال فرض کنید الگوریتم تقسیم‌بندی همیشه دو بخش با نسبت ۹ به ۱ تولید کند، که در نگاه اول بسیار نامتوازن به نظر می‌رسد. در این صورت رابطه‌ی بازگشتی زیر را برای زمان اجرای مرتب‌سازی سریع خواهیم داشت:

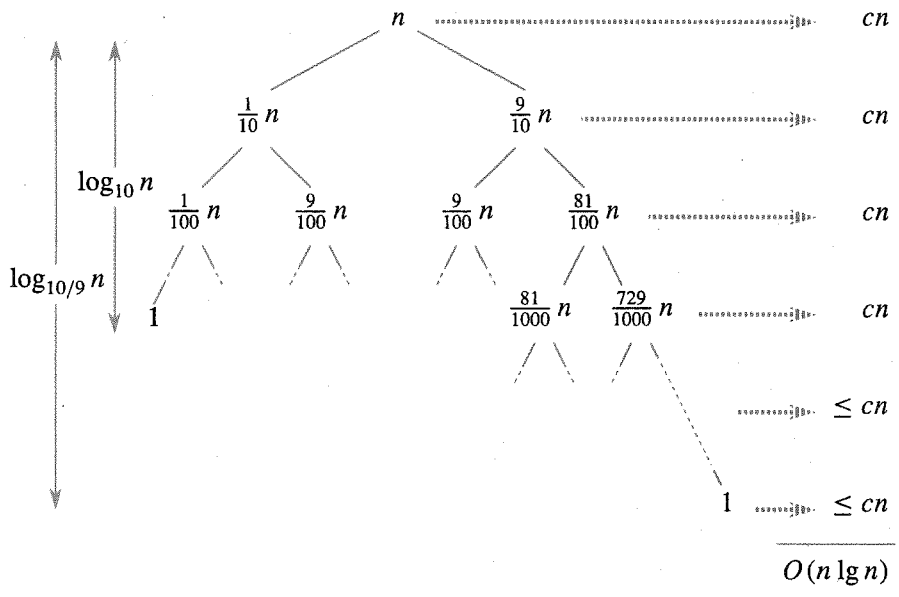
$$T(n) \le T(9n/10) + T(n/10) + cn$$

که در آن صریحاً ثابت مخفی $c$ را در جمله‌ی $\theta(n)$ ذکر کرده‌ایم. شکل ۷-۴ درخت بازگشتی این رابطه را نشان می‌دهد. توجه کنید که هر سطح از درخت دارای هزینه‌ی $cn$ است، تا جایی که یک حالت مرزی در عمق $\lg_{10} n = \theta(\lg n)$ اتفاق می‌افتد، و پس از آن هزینه‌ی هر سطح حداکثر برابر خواهد بود با $cn$. بازگشت در عمق $\lg_{10/9} n = \theta(\lg n)$ پایان می‌یابد. بنابراین هزینه‌ی کلی مرتب‌سازی سریع برابر خواهد بود با $O(n \lg n)$. پس با تقسیم‌بندی با نسبت ۹ به ۱ در هر سطح بازگشت، که ظاهراً به نظر نامتوازن می‌رسد، مرتب‌سازی سریع در زمان $O(n \lg n)$ اجرا می‌شود – هزینه‌ای که به صورت حدی با تقسیم‌بندی کاملاً متوازن برابر است. در واقع حتی تقسیم‌بندی با نسبت ۹۹ به ۱ هم منجر زمان اجرایی برابر با $O(n \lg n)$ خواهد شد. دلیل آن این است که هر نوع تقسیم‌بندی با نسبت *ثابت* باعث می‌شود که عمق درخت بازگشتی $\theta(\lg n)$ باشد، که در آن هزینه‌ی هر سطح $O(n)$ است. بنابراین اگر تقسیم‌بندی نسبت ثابت داشته باشد، زمان اجرا $O(n \lg n)$ خواهد بود.

## شهود در مورد حالت متوسط

برای داشتن درک درست از رفتار مرتب‌سازی سریع تصادفی، باید فرضی داشته باشیم از این که چه انتظاری از مقدار ورودی‌ها داریم. رفتار مرتب‌سازی سریع به ترتیب عناصر آرایه‌ی ورودی نسبت به هم بستگی دارد، و نه به مقدار خود عناصر. مانند تحلیل احتمالاتی در مسئله‌ی استخدام در بخش ۵-۲، از این به بعد فرض خواهیم کرد که احتمال وقوع هر یک از جایگشت‌های آرایه‌ی ورودی با بقیه یکسان است.

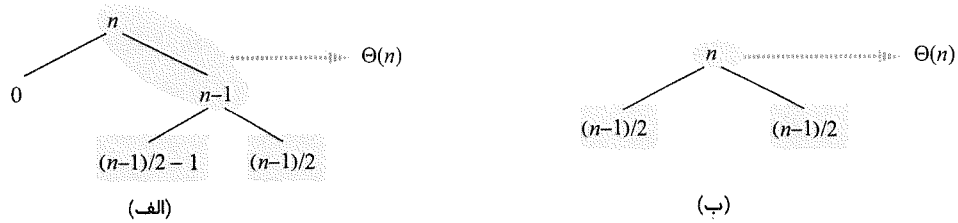
وقتی مرتب‌سازی سریع را روی یک آرایه‌ی ورودی تصادفی اجرا می‌کنیم، ممکن است تقسیم‌بندی همیشه به یک صورت اتفاق بیافتد، همان طور که در تحلیل غیررسمی خود فرض کردیم. در کل انتظار داریم که بعضی از تقسیم‌بندی‌ها به شکل معقولی متوازن باشند، و بعضی به شدت نامتوازن.




**شکل ۷-٤** یک درخت بازگشتی برای QUICKSORT در حالتی که PARTITION همیشه دو قسمت با نسبت ۹ به ۱ می‌سازد، که نتیجه‌ی آن زمان اجرای $O(n \lg n)$ است. گره‌ها اندازه‌ی زیرمسئله‌ها را نشان می‌دهند، به همراه هزینه‌ی هر سطح در سمت راست. هزینه‌ی سطوح شامل ثابت مخفی $c$ در جمله‌ی $\theta(n)$ می‌باشد.

به عنوان مثال در تمرین ۷-۲-۶ از شما خواسته می‌شود که نشان دهید حدود ۸۰ درصد از تقسیم‌بندی‌های PARTITION، بیش از نسبت ۹ به ۱ متوازن هستند، و حدود ۲۰ درصد از آن‌ها کمتر از نسبت ۹ به ۱.

در حالت متوسط PARTITION مخلوطی از تقسیم‌بندی‌های «خوب» و «بد» ارائه می‌دهد. در یک درخت بازگشتی برای اجرای حالت متوسط PARTITION، تقسیم‌بندی‌های خوب و بد به صورت تصادفی در سرتاسر درخت توزیع شده‌اند. با این حال برای درک شهودی فرض کنید که تقسیم‌بندی‌های خوب و بد به صورت یک سطح در میان در درخت توزیع شده‌اند، و تقسیم‌بندی‌های خوب، بهترین حالت تقسیم‌بندی، و تقسیم‌بندی‌های بد، بدترین حالت تقسیم‌بندی هستند. در شکل ۷-۵ (الف) تقسیم‌بندی در دو سطح متوالی نشان داده شده است. در ریشه‌ی درخت هزینه‌ی تقسیم‌بندی $n$ است، و زیرآرایه‌های تولید شده دارای اندازه‌های $n-1$ و ۰ هستند: بدترین حالت. در سطح بعدی زیرآرایه‌ی با اندازه‌ی $n-1$ به بهترین شکل تقسیم‌بندی شده است: دو زیرآرایه با اندازه‌های $(n-1)/2-1$ و $(n-1)/2$. اجازه دهید فرض کنیم که هزینه‌ی حالت مرزی برای زیرآرایه با اندازه‌ی ۰ برابر با ۱ است.




**شکل ۷-۵** (الف) دو سطح از یک درخت بازگشتی برای QUICKSORT. هزینه‌ی تقسیم‌بندی برای ریشه $n$ است و یک تقسیم‌بندی «بد» تولید می‌شود: دو زیرآرایه با اندازه‌های ۰ و $n-1$. تقسیم‌بندی زیرآرایه با اندازه‌ی $n-1$، هزینه‌ای برابر با $n-1$ دارد و یک تقسیم‌بندی «خوب» تولید می‌کند: زیرآرایه‌هایی با اندازه‌های $(n-1)/2-1$ و $(n-1)/2$. (ب) یک سطح تنها از درخت بازگشتی که در نهایت توازن است. در هر دو قسمت، هزینه‌ی تقسیم‌بندی برای زیرمسئله‌هایی که با سایه‌ی بیضی شکل مشخص شده‌اند، $\theta(n)$ است. با این حال زیرمسئله‌های باقی مانده در (الف)، که با سایه‌ی مستطیلی مشخص شده‌اند، از زیرمسئله‌های مشابه در (ب) بزرگ‌تر نیستند.

ترکیب یک تقسیم‌بندی بد و به دنبال آن یک تقسیم‌بندی خوب، سه زیرآرایه با اندازه‌های ۰، $(n-1)/2-1$، و $(n-1)/2$ تولید می‌کند که هزینه‌ی کل تقسیم‌بندی برابر است با $\theta(n)+\theta(n-1)=\theta(n)$. مسلماً این موقعیت بدتر از موقعیت به وجود آمده در شکل ۷-۵(ب) نیست، که در آن با یک مرحله تقسیم‌بندی، دو زیرآرایه با اندازه‌ی $(n-1)/2$ و با هزینه‌ی $\theta(n)$ ساخته می‌شود، با این که حالت دوم یک حالت متوازن است! به طور شهودی هزینه‌ی $\theta(n-1)$ مربوط به تقسیم‌بندی بد را می‌توان با هزینه‌ی $\theta(n)$ تقسیم‌بندی خوب یکی در نظر گرفت، و نتیجه‌ی کلی یک تقسیم‌بندی خوب است. بنابراین زمان اجرای مرتب‌سازی سریع در حالتی که تقسیم‌بندی‌های خوب و بد به صورت یک در میان اتفاق می‌افتند، مانند حالتی است که تمام تقسیم‌بندی‌ها خوب باشند، یعنی $O(n \lg n)$، ولی با ثابت مخفی بزرگ‌تر. در بخش ۷-۴-۲ تحلیلی دقیق‌تر از امیدریاضی زمان اجرای مرتب‌سازی سریع در حالت تصادفی خواهیم داشت.

## تمرین‌ها

۷-۲-۱ با استفاده از روش جانشین‌سازی اثبات کنید که جواب رابطه‌ی بازگشتی $T(n)=T(n-1)+\theta(n)$ برابر است با $T(n)=\theta(n^2)$، همان طور که در ابتدای بخش ۷-۲ گفته شد.

۷-۲-۲ اگر تمام عناصر آرایه‌ی $A$ مقدار برابر داشته باشند، زمان اجرای QUICKSORT چقدر خواهد بود؟

۷-۲-۳ اثبات کنید که اگر عناصر آرایه‌ی $A$ مقادیر متفاوت داشته باشند و به صورت نزولی مرتب شده باشند، زمان اجرای QUICKSORT برابر خواهد بود با $\theta(n^2)$.

۷-۲-۴ بانک‌ها معمولاً تراکنش‌های مالی در یک حساب را به ترتیب زمان وقوع تراکنش ثبت می‌کنند، ولی بسیاری از مردم می‌خواهند صورت حساب بانکی چک‌های خود را به ترتیب شماره‌ی چک دریافت کنند. مردم معمولاً چک‌های خود را به ترتیب شماره‌ی چک می‌نویسند، و معمولاً دریافت کننده‌های چک آن‌ها را به سرعت نقد می‌کنند. بنابراین مسئله‌ی تبدیل ترتیب برحسب زمان تراکنش به ترتیب برحسب شماره‌ی چک، مسئله‌ی مرتب‌سازی یک ورودی تقریباً مرتب شده است. بحث کنید که رویه‌ی INSERTION-SORT به احتمال زیاد این مسئله را با سرعت بیشتری از QUICKSORT حل می‌کند.

۷-۲-۵ فرض کنید که تقسیم‌بندی در هر مرحله از مرتب‌سازی سریع به نسبت $\alpha-1$ به $\alpha$ انجام می‌شود، که در آن $0 < \alpha \leq 1/2$ یک ثابت است. نشان دهید که حداقل عمق یک برگ در درخت بازگشتی تقریباً برابر است با $-\lg n/\lg\alpha$ و حداکثر عمق یک برگ تقریباً برابر است با $-\lg n/\lg(1-\alpha)$. (نگران گرد کردن اعداد نا صحیح نباشید.)

۷-۲-۶★ نشان دهید که برای هر ثابت $0 < \alpha \leq 1/2$، احتمال این که رویه‌ی PARTITION بر روی یک ورودی تصادفی، یک تقسیم‌بندی متوازن‌تر از $\alpha-1$ به $\alpha$ انجام دهد، $2\alpha-1$ است.

## ۷-۳ یک نسخه‌ی تصادفی از مرتب‌سازی سریع

در تعیین رفتار حالت متوسط مرتب‌سازی سریع، فرضی کردیم مبنی بر این که احتمال وقوع تمام جایگشت‌های ورودی با هم برابر است. با این حال در موقعیت‌های واقعی، همیشه نمی‌توانیم این فرض را داشته باشیم. (تمرین ۷-۲-۴ را ببینید.) همان‌طور که در بخش ۵-۳ دیدیم، بعضی مواقع می‌توانیم الگوریتم را تا حدودی تصادفی کنیم تا یک کارایی حالت متوسط خوب برای تمام ورودی‌ها به دست آوریم. اکثراً استفاده از نسخه‌ی تصادفی شده‌ی مرتب‌سازی سریع برای ورودی‌های به اندازه‌ی کافی بزرگ به نسخه‌ی غیر تصادفی آن ترجیح دارد.

در بخش ۵-۳ برای تصادفی کردن الگوریتم خود ترتیب آن را صریحاً تغییر دادیم. برای مرتب‌سازی سریع هم می‌توانیم همین کار را انجام دهیم، ولی تکنیک دیگری به نام *نمونه برداری تصادفی* وجود دارد که تحلیل را بسیار ساده‌تر می‌کند. به جای این که همیشه از $A[r]$ به عنوان محور استفاده کنیم، از یک عنصر تصادفی از زیرآرایه‌ی $A[p \mathinner{..} r]$ استفاده می‌کنیم. این کار را با جابه‌جا کردن عنصر $A[r]$ با یک عنصر از $A[p \mathinner{..} r]$ که به صورت تصادفی انتخاب شده است، انجام می‌دهیم. این اصلاح در الگوریتم باعث می‌شود که مطمئن شویم عنصر محور $x = A[r]$ با احتمال برابر می‌تواند هر یک از $r-p+1$ عنصر در زیرآرایه باشد. از آن جایی که عنصر محور به صورت تصادفی انتخاب شده است، انتظار داریم که تقسیم‌بندی آرایه‌ی ورودی در حالت متوسط در حد معقولی متوازن باشد.

تغییرات ایجاد شده در PARTITION و QUICKSORT جزئی هستند. در رویه‌ی تقسیم‌بندی جدید، فقط باید قبل از تقسیم‌بندی یک جابه‌جایی انجام دهیم:

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

مرتب‌سازی سریع جدید، به جای PARTITION رویه‌ی RANDOMIZED-PARTITION را فراخوانی می‌کند:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

این الگوریتم در بخش بعد تحلیل خواهد شد.

## تمرین‌ها

۷-۳-۱ چرا برای الگوریتم‌های تصادفی امیدریاضی زمان اجرا را تحلیل می‌کنیم، و نه بدترین حالت را؟

۷-۳-۲ حین اجرای رویه‌ی RANDOMIZED-QUICKSORT، در بدترین حالت تابع تولید کننده‌ی اعداد تصادفی (RANDOM) چند بار فراخوانی خواهد شد؟ در بهترین حالت چطور؟ جواب‌های خود را با استفاده از نماد $\theta$ ارائه کنید.

## ۷-۴ تحلیل مرتب‌سازی سریع

در بخش ۷-۲ به صورت شهودی در مورد بدترین حالت رفتار مرتب‌سازی سریع بحث کردیم، همین‌طور این که چرا انتظار داریم این الگوریتم با سرعت خوبی اجرا شود. در این بخش رفتار مرتب‌سازی سریع را به صورت دقیق‌تر بررسی می‌کنیم. با تحلیل بدترین حالت شروع می‌کنیم، که هم برای QUICKSORT و هم برای RANDOMIZED-QUICKSORT کاربرد دارد، و با تحلیل حالت متوسط RANDOMIZED-QUICKSORT کار را پایان می‌دهیم.

### ۷-۴-۱ تحلیل بدترین حالت

در بخش ۷-۲ دیدیم که بدترین حالت تقسیم در تمام سطوح درخت بازگشتی مرتب‌سازی سریع، زمان اجرای $\theta(n^2)$ را نتیجه خواهد داد، که به صورت شهودی بدترین حالت زمان اجرای الگوریتم است. اکنون این ادعا را اثبات می‌کنیم.

با استفاده از روش جانشین‌سازی (بخش ۴-۳) می‌توانیم نشان دهیم که زمان اجرای مرتب‌سازی سریع $O(n^2)$ است. فرض کنید $T(n)$ بدترین حالت زمان اجرای رویه‌ی QUICKSORT بر روی یک ورودی با اندازه‌ی $n$ باشد. رابطه‌ی 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 در آن پارامتر $q$ از ۰ تا $n-1$ تغییر می‌کند، چرا که رویه‌ی PARTITION دو زیرمسئله با مجموع اندازه‌ی $n-1$ تولید می‌کند. حدس می‌زنیم که به ازای ثابتی مانند $c$ داشته باشیم $T(n) \le cn^2$. با جایگذاری این حدس در بازگشت (۷-۱) خواهیم داشت:

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + (c(n-q-1)^2) + \theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \theta(n) \end{aligned}$$

عبارت $q^2 + (n-q-1)^2$ زمانی بیشینه‌ی مقدار خود را نسبت به پارامتر $0 \le q \le n-1$ اختیار می‌کند که $q$ در یکی از نقاط پایانی دامنه باشد، چرا که مشتق دوم این عبارت نسبت به $q$ مثبت است (تمرین ۷-۴-۳ را ببینید). این مشاهدات کران $\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$ را به ما می‌دهد. با استفاده از این کران خواهیم داشت

$$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \theta(n) \\ &\le cn^2 \end{aligned}$$

چرا که می‌توانیم ثابت $c$ را به حد کافی بزرگ انتخاب کنیم تا جمله‌ی $c(2n-1)$ عبارت $\theta(n)$ را خنثی کند. بنابراین، $T(n) = O(n^2)$. در بخش ۷-۲ حالت خاصی را دیدیم که در آن مرتب‌سازی سریع در زمان $\theta(n^2)$ اجرا می‌شود: زمانی که تقسیم‌بندی نامتوازن است. همچنین در تمرین ۷-۴-۱ از شما خواسته می‌شود که اثبات کنید جواب رابطه‌ی بازگشتی (۷-۱) برابر است با $T(n) = \theta(n^2)$. بنابراین (بدترین حالت) زمان اجرای مرتب‌سازی سریع برابر است با $\theta(n^2)$.

## ۷-۴-۲ امیدریاضی زمان اجرا

قبلاً به صورت شهودی بحث کردیم که چرا حالت متوسط زمان اجرای RANDOMIZED-QUICKSORT برابر با $O(n \lg n)$ است: اگر در هر سطح از بازگشت تابع RANDOMIZED-PARTITION هر تعداد ثابتی از عناصر را در یک سمت بگذارد، عمق درخت بازگشتی $\theta(\lg n)$ خواهد بود، و در هر سطح هم هزینه‌ی $O(n)$ صرف خواهد شد. حتی اگر در میان این سطوح، سطوحی با نامتوازن‌ترین شکل ممکن قرار دهیم، باز هم زمان اجرای کلی $O(n \lg n)$ خواهد بود. زمانی می‌توانیم امیدریاضی زمان اجرای RANDOMIZED-QUICKSORT را با دقت تعیین کنیم که نحوه‌ی کار رویه‌ی تقسیم‌بندی را درک کنیم، و سپس با استفاده از آن می‌توانیم کران $\theta(n \lg n)$ را بر روی امیدریاضی زمان اجرای RANDOMIZED-QUICKSORT قرار دهیم. این کران بالا بر روی

امیدریاضی زمان اجرا، به علاوه‌ی کران $O(n \lg n)$ بر روی بهترین زمان اجرا برای الگوریتم‌های مرتب‌سازی، که در بخش ۷-۲ دیدیم، امیدریاضی زمان اجرای $\theta(n \lg n)$ را نتیجه می‌دهد. فرض خواهیم کرد مقادیر عناصری که مرتب می‌شوند یکتا هستند.

## زمان اجرا و مقایسه‌ها

رویه‌های QUICKSORT و RANDOMIZED-QUICKSORT فقط در نحوه‌ی انتخاب عناصر محور با یکدیگر تفاوت دارند، و در تمام جنبه‌های دیگر یکسان عمل می‌کنند. بنابراین با بررسی رویه‌های QUICKSORT و PARTITION می‌توانیم رویه‌ی RANDOMIZED-QUICKSORT را تحلیل کنیم، ولی با این فرض که عناصر محور به صورت تصادفی از میان عناصر زیرآرایه‌ی ارسال شده به RANDOMIZED-PARTITION انتخاب می‌شوند.

زمان اجرای QUICKSORT به شدت تحت تأثیر زمان صرف شده در رویه‌ی PARTITION است. در هر فراخوانی رویه‌ی PARTITION یک عنصر محور انتخاب می‌شود، و این عنصر هیچ وقت در فراخوانی‌های بعدی QUICKSORT و PARTITION ظاهر نخواهد شد. بنابراین در تمام مدت اجرای الگوریتم مرتب‌سازی سریع، رویه‌ی PARTITION حداکثر می‌تواند $n$ بار فراخوانی شود. هر بار فراخوانی PARTITION به زمان $O(1)$ نیاز دارد، به علاوه‌ی مقداری زمان که متناسب است با تعداد تکرارهای حلقه‌ی for در خطوط ۳-۶. هر بار تکرار این حلقه یک عملیات مقایسه در خط ۴ انجام می‌دهد، یعنی مقایسه‌ی عنصر محور با یکی دیگر از عناصر آرایه‌ی $A$. بنابراین اگر بتوانیم تعداد کل دفعاتی که خط ۴ اجرا می‌شود را بشماریم، می‌توانیم کرانی برای کل زمانی که در حلقه‌ی for در کل اجرای QUICKSORT صرف می‌شود تعیین کنیم.

**لم ۷-۱** فرض کنید $X$ تعداد کل دفعاتی باشد که خط ۴ از PARTITION در طول اجرای QUICKSORT روی یک آرایه‌ی $n$ عنصری اجرا می‌شود. در این صورت زمان اجرای QUICKSORT برابر خواهد بود با $O(n+X)$.

*اثبات* طبق بحث بالا PARTITION حداکثر $n$ بار فراخوانی می‌شود، که هرکدام مقدار ثابتی زمان مصرف می‌کنند و سپس حلقه‌ی for را به تعداد خاصی اجرا می‌کنند. هر بار تکرار حلقه‌ی for یک بار خط ۴ را اجرا می‌کند. ■

بنابراین هدف ما محاسبه‌ی $X$ خواهد بود، که تعداد کل مقایسه‌هایی است که در تمام فراخوانی‌های PARTITION انجام می‌شود. در این جا سعی نخواهیم کرد تعداد مقایسه‌ها را در هر یک از فراخوانی‌های PARTITION تحلیل کنیم. در عوض یک کران کلی بر روی تعداد کل مقایسه‌ها به دست خواهیم آورد. برای انجام این کار باید بفهمیم که الگوریتم کی دو عنصر را با یکدیگر مقایسه می‌کند و کی نمی‌کند. برای راحتی تحلیل، عناصر آرایه‌ی $A$ را به صورت $z_1, z_2, \ldots, z_n$ نام گذاری می‌کنیم، که در آن $z_i$ نشان‌دهنده‌ی $i$امین عنصر کوچک آرایه است. همچنین مجموعه‌ی

$Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ را تعریف می‌کنیم که شامل عناصر بین $z_i$ و $z_j$، به علاوه‌ی خود این عناصر است.

الگوریتم در چه صورت عناصر $z_i$ و $z_j$ را مقایسه می‌کند؟ برای جواب دادن به این سؤال، ابتدا دقت می‌کنیم که هر جفت از عناصر حداکثر یک بار با هم مقایسه می‌شوند. چرا؟ عناصر فقط با عنصر محور مقایسه می‌شوند، و بعد از این که یک فراخوانی خاص پایان یافت، عنصر محور استفاده شده در آن فراخوانی هرگز دوباره با عناصر دیگر مقایسه نخواهد شد.

در این تحلیل از متغیرهای تصادفی شاخص استفاده خواهیم کرد (بخش ۵-۲ را ببینید). تعریف می‌کنیم

$$X_{ij} = I\{\text{مقایسه شود } z_j \text{ با } z_i\}$$

که در آن فرض می‌کنیم این مقایسه ممکن است در هر جایی از اجرای الگوریتم اتفاق بیافتد، و نه فقط در یک تکرار و یا یک فراخوانی PARTITION. از آن جایی که هر جفت از عناصر حداکثر یک بار مقایسه می‌شوند، به راحتی می‌توانیم تعداد کل مقایسه‌های انجام شده در الگوریتم را توصیف کنیم:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

با امیدریاضی گرفتن از دو طرف و استفاده از خطی بودن امیدریاضی و لم ۵-۱، به دست می‌آوریم:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{\text{مقایسه شود } z_j \text{ با } z_i\} \end{aligned} \qquad (۲-۷)$$

هنوز محاسبه‌ی $\Pr\{\text{مقایسه شود } z_j \text{ با } z_i\}$ باقی می‌ماند. در این تحلیل فرض می‌کنیم رویه‌ی RANDOMIZED-PARTITION هر عنصر محور را به صورت جداگانه و مستقل انتخاب می‌کند.

اجازه دهید به این فکر کنیم که در چه صورت دو عنصر مقایسه *نمی‌شوند*. یک ورودی مرتب‌سازی سریع را در نظر بگیرید که شامل اعداد ۱ تا ۱۰ باشد (به هر ترتیبی)، و فرض کنید که اولین عنصر محور ۷ باشد. آن گاه اولین فراخوانی PARTITION عناصر را به دو مجموعه‌ی $\{1,2,3,4,5,6\}$ و $\{8,9,10\}$ تقسیم می‌کند. برای این کار عنصر ۷ با تمام عناصر مقایسه می‌شود، ولی هیچ یک از عناصر مجموعه‌ی اول (مثلاً ۲)، هیچ وقت با هیچ یک از عناصر مجموعه‌ی دوم (مثلاً ۹) مقایسه نمی‌شود.

به طور کلی، از آن جایی که فرض کردیم مقدار عناصر یکتا است، وقتی یک عنصر محوری $x$، که $z_i < x < z_j$، انتخاب می‌شود، می‌دانیم که عناصر $z_i$ و $z_j$ هیچ گاه در طول اجرای الگوریتم با

یکدیگر مقایسه نخواهند شد. از سوی دیگر اگر $z_i$ قبل از تمام عناصر $Z_{ij}$ به عنوان محور انتخاب شود، آن گاه $z_i$ با تمام عناصر $Z_{ij}$ (غیر از خودش) مقایسه می‌شود. به طور مشابه، اگر $z_j$ قبل از تمام عناصر $Z_{ij}$ به عنوان محور انتخاب شود، $z_j$ با تمام عناصر $Z_{ij}$ (غیر از خودش) مقایسه می‌شود. در مثال ما، مقدار ۷ با ۹ مقایسه می‌شود چرا که ۷ قبل از تمام اعضای $Z_{۷,۹}$ به عنوان محور انتخاب می‌شود. در مقابل، ۲ و ۹ هیچ گاه با یکدیگر مقایسه نمی‌شوند، چرا که اولین عنصری که از $Z_{۲,۹}$ به عنوان محور انتخاب می‌شود ۷ است. بنابراین، $z_i$ و $z_j$ با هم مقایسه می‌شوند اگر و فقط اگر اولین عنصری که از $Z_{ij}$ به عنوان محور انتخاب می‌شود، یا $z_i$ باشد و یا $z_j$.

اکنون احتمال وقوع این رخداد را بررسی می‌کنیم. قبل از این که یک عنصر از $Z_{ij}$ به عنوان محور انتخاب شود، تمام مجموعه‌ی $Z_{ij}$ در یک قسمت قرار دارد. بنابراین احتمال این که هر یک از عناصر $Z_{ij}$ به عنوان محور انتخاب شود، برابر است. از آن جایی که مجموعه‌ی $Z_{ij}$ دارای $j-i+۱$ عنصر است، و از آن جایی که محورها به صورت جداگانه و مستقل انتخاب می‌شوند، احتمال این که هر یک از عناصر اول انتخاب شود برابر است با $۱/(j-i+۱)$. بنابراین داریم:

$$\begin{aligned}\{z_i \text{ با } z_j \text{ مقایسه شود}\} &= \Pr\{z_i \text{ یا } z_j \text{ اولین محور انتخاب شده از } Z_{ij} \text{ باشند}\} \\ &= \Pr\{z_i \text{ اولین محور انتخاب شده از } Z_{ij} \text{ باشد}\} \\ &\quad + \Pr\{z_j \text{ اولین محور انتخاب شده از } Z_{ij} \text{ باشد}\} \\ &= \frac{۱}{j-i+۱} + \frac{۱}{j-i+۱} \\ &= \frac{۲}{j-i+۱}\end{aligned} \tag{۳-۷}$$

خط دوم از این جا نتیجه می‌شود که دو واقعه با یکدیگر ناسازگار هستند. با ترکیب تساوی‌های (۲-۷) و (۳-۷) خواهیم داشت:

$$E[X] = \sum_{i=۱}^{n-۱} \sum_{j=i+۱}^{n} \frac{۲}{j-i+۱}$$

این سری را می‌توانیم با استفاده از یک تغییر متغیر ($k = j-i$) و کران سری‌های هارمونیک در تساوی (الف-۷) ارزیابی کنیم:

$$\begin{aligned}E[X] &= \sum_{i=۱}^{n-۱} \sum_{j=i+۱}^{n} \frac{۲}{j-i+۱} \\ &= \sum_{i=۱}^{n-۱} \sum_{k=۱}^{n-i} \frac{۲}{k+۱} \\ &< \sum_{i=۱}^{n-۱} \sum_{k=۱}^{n} \frac{۲}{k} \\ &= \sum_{i=۱}^{n-۱} O(\lg n) \\ &= O(n \lg n)\end{aligned} \tag{۴-۷}$$

بنابراین نتیجه می‌گیریم که با استفاده از RANDOMIZED-PARTITION، اگر مقادیر عناصر یکتا باشد، امیدریاضی زمان اجرای مرتب‌سازی سریع برابر است با $O(n \lg n)$.

## تمرین‌ها

۷-۴-۱ نشان دهید که جواب رابطه‌ی بازگشتی

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \theta(n)$$

برابر است با $T(n) = \Omega(n^2)$.

۷-۴-۲ نشان دهید که بهترین حالت زمان اجرای مرتب‌سازی سریع از مرتبه‌ی $\Omega(n \lg n)$ است.

۷-۴-۳ نشان دهید که $q^2 + (n-q-1)^2$ زمانی به مقدار بیشینه‌ی خود بر روی $q = 0, 1, \ldots, n-1$ می‌رسد که $q = 0$ یا $q = n-1$.

۷-۴-۴ نشان دهید که امیدریاضی زمان اجرای RANDOMIZED-QUICKSORT برابر است با $\Omega(n \lg n)$.

۷-۴-۵ با استفاده از زمان اجرای سریع مرتب‌سازی درجی بر روی ورودی‌های «تقریباً» مرتب، می‌توان زمان اجرای مرتب‌سازی سریع را در عمل افزایش داد. اجازه دهید مرتب‌سازی سریع را طوری اصلاح کنیم که وقتی بر روی یک زیرآرایه با کم‌تر از $k$ عنصر فراخوانی می‌شود، بدون انجام هیچ کاری بازگشت کند. وقتی بالاترین سطح فراخوانی مرتب‌سازی سریع پایان یافت، مرتب‌سازی درجی را روی تمام آرایه فراخوانی می‌کنیم. نشان دهید که امیدریاضی زمان اجرای این الگوریتم مرتب‌سازی $O(nk + n \lg(n/k))$ است. در تئوری و در عمل، چگونه باید $k$ را انتخاب کنیم؟

۷-۴-۶ ★ فرض کنید رویه‌ی PARTITION را بدین صورت اصلاح می‌کنیم که به صورت تصادفی سه عنصر از آرایه‌ی $A$ انتخاب کند و حول میانه‌ی آن‌ها (عنصر وسطی در میان سه عنصر) تقسیم‌بندی را انجام دهد. احتمال این که در بدترین حالت نسبت تقسیم‌بندی، $\alpha$ به $(1-\alpha)$ باشد را به صورت تابعی از $\alpha$ که در آن $0 < \alpha < 1$، محاسبه کنید.

## مسائل

۷-۱ *تصحیح تقسیم‌بندی به روش هور (Hoare)*

نسخه‌ی تقسیم‌بندی که در این فصل ارائه شد، نسخه‌ی اصلی الگوریتم تقسیم‌بندی نیست. در زیر نسخه‌ی اصلی تقسیم‌بندی را می‌بینیم، که متعلق به ت.هور (T.Hoare) است:

```
HOARE-PARTITION(A, p, r)
1  x = A[p]
```

```
2   i = p − 1
3   j = r + 1
4   while TRUE
5       repeat
6           j = j − 1
7       until A[j] ≤ x
8       repeat
9           i = i + 1
10      until A[i] ≥ x
11      if i < j
12          exchange A[i] with A[j]
13      else return j
```

I. عملیات HOAR-PARTITION را روی آرایه‌ی $A = \langle ۱۳,۱۹,۹,۵,۱۲,۸,۷,۴,۱۱,۲,۶,۲۱ \rangle$ مشخص کنید. همچنین مقدار عناصر آرایه و متغیرهای کمکی را پس از هر بار تکرار حلقه‌ی for در خطوط ۴-۱۳ نشان دهید.

در سه سؤال بعد از شما خواسته می‌شود که بحثی دقیق در مورد درستی HOAR-PARTITION ارائه دهید. با فرض این که زیرآرایه‌ی $A[p \,..\, r]$ حاوی حداقل دو عنصر است، عبارات زیر را اثبات کنید:

II. اندیس‌های $i$ و $j$ به گونه‌ای هستند که هیچ گاه به عنصری خارج از محدوده‌ی زیرآرایه‌ی $A[p \,..\, r]$ دسترسی پیدا نمی‌کنند.

III. وقتی HOAR-PARTITION پایان می‌یابد، مقداری مانند $j$ بازمی‌گرداند به طوری که $p \le j < r$.

IV. وقتی HAOR-PARTITION پایان می‌یابد، تمام عناصر $A[p \,..\, j]$ کمتر یا مساوی تمام عناصر $A[j+۱ \,..\, r]$ هستند.

رویه‌ی PARTITION در بخش ۷-۱، مقدار محور (که ابتدا در $A[r]$ قرار دارد) را از دو بخش ساخته شده جدا می‌کند. از سوی دیگر رویه‌ی HOAR-PARTITION همیشه عنصر محور (که ابتدا در $A[p]$ قرار دارد) را در یکی از دو بخش $A[p \,..\, j]$ و $A[j+۱ \,..\, r]$ قرار می‌دهد. از آن جایی که $p \le j < r$، این تقسیم‌بندی همیشه قابل استفاده است.

V. رویه‌ی QUICKSORT را طوری بازنویسی کنید که از HOAR-PARTITION استفاده کند.

## ۷-۲ *مرتب‌سازی سریع با عناصر تکراری*

در تحلیل امیدریاضی زمان اجرای مرتب‌سازی سریع تصادفی در بخش ۷-۴-۲ فرض کردیم که مقادیر عناصر متفاوت از یکدیگر هستند. در این مسئله بررسی می‌کنیم که اگر چنین نباشد، چه رخ خواهد داد.

I. فرض کنید مقدار تمام عناصر با هم برابر باشد. زمان اجرای مرتب‌سازی سریع تصادفی در این حالت چگونه خواهد بود؟

II. رویه‌ی PARTITION یک اندیس $q$ بازمی‌گرداند به طوری که هر عنصر $A[p\,..\,q-1]$ کوچک‌تر یا مساوی $A[q]$ باشد، و هر عنصر $A[q+1\,..\,r]$ بزرگ‌تر از $A[q]$. رویه‌ی PARTITION را اصلاح کرده و یک رویه‌ی PARTITION'$(A,p,r)$ بسازید، که عناصر $A[p\,..\,r]$ را بازآرایی کرده و دو اندیس $q$ و $t$ بازمی‌گرداند، به طوری که $p \le q \le t \le r$، و

- تمام عناصر $A[q\,..\,t]$ برابر هستند،
- هر عنصر $A[p\,..\,q-1]$ کوچک‌تر از $A[q]$ است، و
- هر عنصر $A[t+1\,..\,r]$ بزرگ‌تر از $A[q]$ است.

مانند PARTITION، رویه‌ی PARTITION' باید در زمان $\theta(r-p)$ اجرا شود.

III. رویه‌ی RANDOMIZED-QUICKSORT را طوری اصلاح کنید که از PARTITION' استفاده کند، و رویه‌ی جدید را RANDOMIZED-QUICKSORT' بنامید. سپس رویه‌ی QUICKSORT را تغییر دهید و یک رویه‌ی QUICKSORT' بسازید که RANDOMIZED-PARTITION' را فراخوانی می‌کند، و فقط روی بخش‌هایی از عناصر بازگشت را انجام می‌دهد که از مساوی بودن آن‌ها با یکدیگر اطمینان ندارد.

IV. با استفاده از QUICKSORT'، چگونه تحلیل بخش ۷-۴-۲ را اصلاح می‌کنید تا نیازی به فرض یکتا بودن عناصر نداشته باشید؟

## ۷-۳ *تحلیلی دیگر برای مرتب‌سازی سریع*

تحلیلی دیگر از زمان اجرای مرتب‌سازی سریع تصادفی شده، به جای تعداد مقایسه‌ها بر روی امیدریاضی زمان اجرای هر فراخوانی بازگشتی RANDOMIZED-QUICKSORT تمرکز می‌کند.

I. اثبات کنید که با داشتن یک آرایه با اندازه‌ی $n$، احتمال این که هر یک از عناصر به عنوان محور انتخاب شود برابر است با $1/n$. از این برای تعریف متغیر تصادفی شاخص

$X_i = I\{i$امین عنصر کوچک به عنوان محور انتخاب شود$\}$

استفاده کنید. $E[X_i]$ چقدر است؟

II. فرض کنید $T(n)$ یک متغیر تصادفی باشد که زمان اجرای مرتب‌سازی سریع را بر روی یک آرایه با اندازه‌ی $n$ نشان می‌دهد. نشان دهید:

$$E[T(n)] = E\left[\sum_{q=1}^{n} X_q (T(q-1) + T(n-q) + \theta(n))\right] \qquad (۷-۵)$$

III. نشان دهید که تساوی (۷-۵) به صورت زیر ساده می‌شود:

$$E[T(n)] = \frac{۲}{n}\sum_{q=۰}^{n-۱} E[T(q)] + \theta(n) \qquad (۷-۶)$$

مسائل

۷-۱ نشان دهید

$$\sum_{k=۱}^{n-۱} k \lg k \le \frac{۱}{۲} n^۲ \lg n - \frac{۱}{۸} n^۲ \qquad (۷-۷)$$

(*راهنمایی:* سری را به دو بخش تقسیم کنید، یکی برای $k = ۱, ۲, ..., \lceil n/۲ \rceil - ۱$ و دیگری برای $k = \lceil n/۲ \rceil, ..., n-۱$.)

۷-۲ با استفاده از کران تساوی (۷-۷)، نشان دهید که رابطه‌ی بازگشتی در تساوی (۷-۶) جوابی به شکل $E[T(n)] = \theta(n \lg n)$ دارد. (*راهنمایی:* به وسیله‌ی جانشین‌سازی، نشان دهید که به ازای $n$های به اندازه‌ی کافی بزرگ و ثابت مثبت $a$ داریم $E[T(n)] \le an \lg n$.)

## ۷-۴ عمق پشته برای مرتب‌سازی سریع

الگوریتم QUICKSORT در بخش ۷-۱ شامل دو فراخوانی بازگشتی از خود است. بعد از فراخوانی PARTITION، ابتدا زیرآرایه‌ی چپ و سپس زیرآرایه‌ی راست به صورت بازگشتی مرتب می‌شوند. در واقع دومین فراخوانی بازگشتی QUICKSORT ضروری نیست؛ به وسیله‌ی یک کنترل تکراری می‌توان از آن اجتناب کرد. این تکنیک که به *بازگشت دنباله‌ای* (tail recursion) معروف است، که به صورت خودکار توسط کامپایلرهای خوب فراهم می‌شود. نسخه‌ی زیر را از مرتب‌سازی سریع در نظر بگیرید که بازگشت دنباله‌ای را پیاده‌سازی می‌کند:

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
1  while p < r
2      // Partition and sort left subarray.
3      q = PARTITION(A, p, r)
4      TAIL-RECURSIVE-QUICKSORT(A, p, q – 1)
5      p = q + 1
```

۷-۴ نشان دهید که TAIL-RECURSIVE-QUICKSORT $(A, ۱, A.length)$ به درستی آرایه‌ی $A$ را مرتب می‌کند.

معمولاً کامپایلرها رویه‌های بازگشتی را با استفاده از پشته اجرا می‌کنند، که این پشته اطلاعات مورد نیاز فراخوانی‌های بازگشتی را در خود نگه می‌دارد، از جمله مقدار پارامترها برای هر فراخوانی بازگشتی. اطلاعات مربوط به آخرین فراخوانی در بالای پشته قرار دارد، و اطلاعات اولین فراخوانی در پایین پشته. وقتی یک رویه احضار می‌شود، اطلاعات آن در پشته *نشانده* (push) می‌شود؛ وقتی رویه پایان یافت، اطلاعات آن از پشته *بازیابی* (pop) می‌شود. از آن جایی که فرض می‌کنیم پارامترهای آرایه به صورت اشاره‌گر ذخیره می‌شوند، اطلاعات هر یک از فراخوانی‌های رویه به $O(۱)$ فضای پشته نیاز دارد. *عمق پشته* (stack depth) عبارت است از حداکثر فضایی از پشته که در کل زمان پردازش اشغال می‌شود.

II. سناریویی را شرح دهید که در آن عمق پشته‌ی TAIL-RECURSIVE-QUICKSORT برای یک آرایه‌ی ورودی با $n$ عنصر، $\theta(n)$ است.

III. کد TAIL-RECURSIVE-QUICKSORT را طوری اصلاح کنید که بدترین حالت عمق پشته $\theta(\lg n)$ باشد. این کد باید زمان اجرای $O(n \lg n)$ را حفظ کند.

## ۷-۵ تقسیم‌بندی میانه‌ی ۳

یک روش برای بهبود رویه‌ی RANDOMIZED-QUICKSORT این است که تقسیم‌بندی را حول محوری انجام دهیم که با دقتی بیشتر از یک عنصر تصادفی انتخاب شده باشد. یک روش معمول متد *میانه‌ی ۳* است: ۳ عنصر به صورت تصادفی انتخاب کرده و میانه (عنصر وسطی) آن‌ها را به عنوان محور در نظر می‌گیریم. (تمرین ۷-۴-۶ را ببینید.) برای این مسئله اجازه دهید فرض کنیم عناصر آرایه‌ی $A[1..n]$ یکتا هستند، و $n \geq 3$. آرایه‌ی خروجی مرتب شده را با $A'[1..n]$ نشان می‌دهیم. با استفاده از متد میانه‌ی ۳ برای انتخاب عنصر محور $x$، تعریف می‌کنیم $p_i = \Pr\{x = A'[i]\}$.

I. یک فرمول دقیق برای $p_i$ به صورت تابعی از $n$ و $i$ برای $i = 2,3,...,n-1$ ارائه کنید. (دقت کنید که $p_1 = p_n = 0$.)

II. در این روش احتمال این که عنصر $x = A'[\lfloor (n+1)/2 \rfloor]$، یعنی میانه‌ی $A[1..n]$، به عنوان محور انتخاب شود، نسبت به حالت معمولی چقدر افزایش می‌یابد؟ فرض کنید $n \to \infty$، و سپس حد نسبت این احتمالات را به دست آورید.

III. اگر تقسیم‌بندی «خوب» را بدین صورت تعریف کنیم که عنصر محور $x = A'[i]$ باشد، که $n/3 \leq i \leq 2n/3$، چقدر احتمال یک تقسیم‌بندی خوب را نسبت به پیاده‌سازی معمولی افزایش داده‌ایم؟ (*راهنمایی:* مجموع را به کمک یک انتگرال تقریب بزنید.)

IV. بحث کنید که در زمان اجرای $\Omega(n \lg n)$ برای مرتب‌سازی سریع، متد میانه‌ی ۳ فقط بر روی ضرایب ثابت تأثیر می‌گذارد.

## ۷-۶ مرتب‌سازی فازی (Fuzzy sorting) بازه‌ها

یک مسئله‌ی مرتب‌سازی را در نظر بگیرید که در آن اطلاع دقیقی از اعداد در دست نیست. در عوض برای هر عدد، یک بازه روی اعداد حقیقی داریم که می‌دانیم آن عدد به آن بازه تعلق دارد. یعنی $n$ بازه‌ی بسته به صورت $[a_i, b_i]$ داریم، که در آن $a_i \leq b_i$. هدف این است که این بازه‌ها را به صورت *فازی* مرتب کنیم، یعنی یک جایگشت $\langle i_1, i_2, ..., i_n \rangle$ از بازه‌ها بسازیم به طوری که برای $j = 1,2,...,n$ مقادیر $c_j \in [a_{i_j}, b_{i_j}]$ موجود باشند که در نامساوی $c_1 \leq c_2 \leq ... \leq c_n$ صدق کنند.

I. یک الگوریتم تصادفی برای مرتب‌سازی فازی $n$ بازه طراحی کنید. شکل کلی الگوریتم شما باید بدین صورت باشد که ابتدا به کمک مرتب‌سازی سریع، نقاط شروع بازه‌ها ($a_i$ها) را

مسائل

مرتب کند، ولی از مزیت بازه‌هایی که همپوشانی دارند برای بهبود زمان اجرا استفاده کند. (هر چه بازه‌ها بیشتر همپوشانی داشته باشند، مسئله‌ی مرتب‌سازی فازی آن‌ها ساده‌تر می‌شود. الگوریتم شما باید تا حد ممکن از این مسئله استفاده کند.)

اثبات کنید که امیدریاضی زمان اجرای الگوریتم شما در حالت کلی $\theta(n \lg n)$ است، ولی زمانی که تمام بازه‌ها همپوشانی دارند (یعنی مقداری مانند $x$ وجود دارد که برای تمام $i$ها $x \in [a_i, b_i]$)، امیدریاضی زمان اجرای آن به $\theta(n)$ کاهش می‌یابد. الگوریتم شما نباید صریحاً ورودی را برای این حالت چک کند؛ در عوض کارایی آن باید به صورت طبیعی با افزایش همپوشانی‌ها افزایش یابد.


# مرتب‌سازی در زمان خطی

## ۸-۰ مقدمه

تا کنون الگوریتم‌های مختلفی معرفی کرده‌ایم که $n$ عدد را در زمان $O(n \lg n)$ مرتب می‌کنند. مرتب‌سازی ادغامی و مرتب‌سازی هرمی این کران بالا را در بدترین حالت دارند؛ مرتب‌سازی سریع در حالت متوسط به این کران دست می‌یابد.

به علاوه برای هر کدام از این الگوریتم‌ها، می‌توانیم یک دنباله از $n$ عدد ورودی پیدا کنیم که باعث می‌شوند الگوریتم در زمان $\theta(n \lg n)$ اجرا شود.

این الگوریتم‌ها دارای یک خصوصیت مشترک جالب توجه هستند: *ترتیبی که آن‌ها تولید می‌کنند فقط بر مبنای مقایسه بین عناصر ورودی است.* این نوع الگوریتم‌های مرتب‌سازی، *مرتب‌سازی‌های مقایسه‌ای* نام دارند. تمام الگوریتم‌های مرتب‌سازی که تا به حال معرفی شده‌اند، مرتب‌سازی‌های مقایسه‌ای هستند.

در بخش ۸-۱ اثبات خواهیم کرد که تمام الگوریتم‌های مرتب‌سازی مقایسه‌ای، برای مرتب‌سازی $n$ عنصر در بدترین حالت باید $\theta(n \lg n)$ مقایسه انجام دهند. بنابراین مرتب‌سازی ادغامی و مرتب‌سازی هرمی به صورت حدی کارامد هستند، و هیچ الگوریتم مرتب‌سازی وجود ندارد که بیش از یک ضریب ثابت از آن‌ها سریع‌تر باشد.

در بخش‌های ۸-۲، ۸-۳ و ۸-۴، سه الگوریتم مرتب‌سازی– مرتب‌سازی شمارشی، مرتب‌سازی مبنایی و مرتب‌سازی سطلی – معرفی خواهد شد که در زمان خطی اجرا می‌شوند. نیازی به گفتن نیست که این الگوریتم‌ها از عملیاتی غیر از مقایسه برای مرتب‌سازی استفاده می‌کنند. بنابراین کران پایین $\theta(n \lg n)$ در مورد آن‌ها صادق نیست.

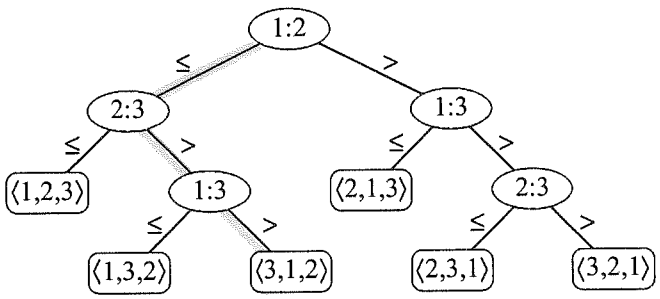
## ۸-۱ کران‌های پایین برای مرتب‌سازی

در یک مرتب‌سازی مقایسه‌ای برای دریافت اطلاعات در مورد ترتیب دنباله‌ی ورودی $\langle a_1, a_2, ..., a_n \rangle$، فقط از مقایسه بین عناصر استفاده می‌شود. یعنی مثلاً در مورد دو عنصر $a_i$ و $a_j$، یکی از تست‌های $a_i < a_j$، $a_i \leq a_j$، $a_i = a_j$، $a_i \geq a_j$، و $a_i > a_j$ را انجام می‌دهیم که ترتیب نسبی آن‌ها را بفهمیم. در مورد مقدار عناصر هیچ گونه مشاهده‌ای انجام نمی‌دهیم، و یا از هیچ روش دیگری برای دریافت اطلاعات در مورد ترتیب عناصر استفاده نمی‌کنیم.

در این بخش بدون از دست دادن کلیت مسئله فرض می‌کنیم که تمام عناصر یکتا هستند. با این فرض مقایسه‌هایی به شکل $a_i = a_j$ بی‌استفاده هستند، و بنابراین فرض می‌کنیم هیچ مقایسه‌ای بدین شکل انجام نمی‌شود. همچنین دقت می‌کنیم که مقایسه‌های $a_i \leq a_j$، $a_i \geq a_j$، $a_i > a_j$، و $a_i < a_j$ یکسان هستند، چرا که اطلاعات مشابهی در مورد ترتیب نسبی $a_i$ و $a_j$ به دست می‌دهند. از این رو فرض می‌کنیم تمام مقایسه‌ها به شکل $a_i \leq a_j$ هستند.

### مدل درخت تصمیم

مرتب‌سازی‌های مقایسه‌ای را به صورت انتزاعی می‌توان به چشم *درخت‌های تصمیم* (decision tree) دید. درخت تصمیم، یک درخت دودویی کامل است که نماینده‌ی مقایسه‌هایی است که یک الگوریتم مرتب‌سازی خاص روی عناصر یک ورودی انجام می‌دهد. از کنترل، جابه‌جایی اطلاعات، و تمام جنبه‌های دیگر الگوریتم صرف نظر شده است. شکل ۸-۱ درخت تصمیم مربوط به الگوریتم مرتب‌سازی درجی (بخش ۲-۱) را نشان می‌دهد که روی یک آرایه‌ی ورودی با سه عنصر عمل می‌کند.




**شکل ۸-۱** درخت تصمیم مرتب‌سازی درجی که بر روی سه عنصر عمل می‌کند. یک گره‌ی ورودی که با $i : j$ نشان داده‌شده است، نماینده‌ی یک مقایسه بین $a_i$ و $a_j$ است. یک برگ که با جایگشت $\langle \pi(1), \pi(2), ..., \pi(n) \rangle$ نشان داده‌شده است، ترتیب $a_{\pi(1)} \leq a_{\pi(2)} \leq ... \leq a_{\pi(n)}$ را نشان می‌دهد. مسیر سایه‌دار نشان‌دهنده‌ی تصمیم‌های اتخاذ شده برای حالتی است که الگوریتم بر روی دنباله‌ی ورودی $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$ اجرا می‌شود. جایگشت $\langle 3,1,2 \rangle$ در برگ مربوط به این ورودی، نشان می‌دهد که ترتیب نهایی عبارت است از $a_3 = 5, a_1 = 6, a_2 = 8$. در کل $3!$ جایگشت ممکن از ورودی‌ها وجود دارد، و بنابراین درخت تصمیم باید حداقل ۶ برگ داشته باشد.

در یک درخت تصمیم، به ازای $1 \le i, j \le n$ هر گره‌ی داخلی با $i : j$ نشان داده می‌شود، که در آن $n$ تعداد عناصر در دنباله‌ی ورودی است. هر برگ با یک جایگشت $\langle \pi(1), \pi(2), ..., \pi(n) \rangle$ نشان داده می‌شود. (برای پیش زمینه‌ای بر روی جایگشت‌ها، بخش پ-۱ را ببینید.) اجرای الگوریتم مرتب‌سازی متناظر است با دنبال کردن یک مسیر ساده از ریشه‌ی درخت تصمیم به یک برگ. در هر گره‌ی داخلی، یک مقایسه‌ی $a_i \le a_j$ انجام می‌شود. در این صورت درخت سمت چپ تعیین کننده‌ی مقایسه‌های بعدی برای حالتی است که می‌دانیم $a_i \le a_j$، و درخت سمت راست تعیین کننده‌ی مقایسه‌های بعدی برای حالتی که $a_i > a_j$. وقتی به یک برگ می‌رسیم، الگوریتم مرتب‌سازی ترتیب $a_{\pi(1)} \le a_{\pi(2)} \le ... \le a_{\pi(n)}$ را تشخیص داده است. از آن جایی که هر الگوریتم مرتب‌سازی صحیح باید قادر باشد که تمام جایگشت‌های مختلف دنباله‌ی ورودی را بسازد، یک شرط لازم برای صحیح بودن یک الگوریتم مرتب‌سازی این است که تمام $n!$ جایگشت مختلف دنباله‌ی ورودی در میان برگ‌های درخت تصمیم آن الگوریتم موجود، و از ریشه توسط یک مسیر که نشان‌دهنده‌ی یک اجرا از الگوریتم است قابل دسترس باشند. (به این برگ‌ها، برگ‌های «قابل دسترس» می‌گوییم.) بنابراین فقط درخت‌های تصمیمی را در نظر می‌گیریم که هر یک از جایگشت‌ها به صورت یک برگ قابل دسترس در آن ظاهر شوند.

## یک کران پایین برای بدترین حالت

طول بلندترین مسیر از ریشه‌ی یک درخت تصمیم به هر یک از برگ‌های آن درخت، نشان دهنده‌ی بیشترین تعداد مقایسه است که آن الگوریتم مرتب‌سازی باید (در بدترین حالت) انجام دهد. در نتیجه بیشترین تعداد مقایسه برای یک الگوریتم مرتب‌سازی مقایسه‌ای برابر است با ارتفاع درخت تصمیم آن. بنابراین یک کران پایین بر روی تمام درخت‌های تصمیمی که تمام جایگشت‌های دنباله‌ی ورودی به صورت یک برگ قابل دسترس در آن حضور دارند، برابر است با کران پایین زمان اجرای تمام الگوریتم‌های مرتب‌سازی مقایسه‌ای. قضیه‌ی زیر این کران پایین را تعیین می‌کند.

**قضیه‌ی ۸-۱** هر الگوریتم مرتب‌سازی مقایسه‌ای، در بدترین حالت به $\Omega(n \lg n)$ مقایسه نیاز دارد.

*اثبات* در نتیجه‌ی بحث بالا، کافی است که ارتفاع یک درخت تصمیم را که برگ‌های قابل دسترس آن حاوی تمام جایگشت‌های آرایه‌ی ورودی است، تعیین کنیم. یک درخت تصمیم مربوط به مرتب‌سازی مقایسه‌ای $n$ عنصر را، با ارتفاع $h$ و $l$ برگ قابل دسترس در نظر بگیرید. از آن جایی که هر یک از $n!$ جایگشت آرایه‌ی ورودی در یک برگ ظاهر می‌شود، داریم $n! \le l$. چون یک درخت دودویی با ارتفاع $h$، نمی‌تواند بیشتر از $2^h$ برگ داشته باشد، داریم:

$$n! \le l 2^h$$

که با لگاریتم گرفتن از دو طرف، به دست می‌آوریم

$$h \le \lg(n!) \qquad \text{(چرا که تابع لگاریتم صعودی اکید است)}$$
$$= \Omega(n \lg n) \qquad \text{(طبق تساوی (۳-۱۹))}$$

**نتیجه‌ی ۸-۲** بین الگوریتم‌های مرتب‌سازی مقایسه، مرتب‌سازی هرمی و مرتب‌سازی ادغامی به صورت حدی کارامد هستند.

*اثبات* کران بالای $O(n \lg n)$ بر روی زمان اجرای مرتب‌سازی هرمی و مرتب‌سازی ادغامی با کران پایین بدترین حالت $\Omega(n \lg n)$ از قضیه‌ی ۸-۱ مطابقت می‌کند. ■

## تمرین‌ها

۸-۱-۱ کوتاه‌ترین عمق ممکن برای یک برگ در درخت تصمیم یک مرتب‌سازی مقایسه‌ای چقدر است؟

۸-۱-۲ بدون استفاده از تقریب استرلینگ (Stirling's approximation)، کران‌های حدی نزدیک برای $\lg(n!)$ بیابید. برای این کار، با استفاده از تکنیک‌های بخش الف-۲ سری $\sum_{k=1}^{n} \lg k$ را ارزیابی کنید.

۸-۱-۳ نشان دهید که هیچ مرتب‌سازی مقایسه‌ای وجود ندارد که زمان اجرای آن حداقل برای نیمی از $n!$ ورودی با اندازه‌ی $n$، خطی باشد. آیا این قضیه در مورد کسر $1/n$ از ورودی‌ها نیز صحیح است؟ در مورد کسر $1/2^n$ چطور؟

۸-۱-۴ یک دنباله‌ی $n$تایی از اعداد داریم که باید مرتب شوند. دنباله‌ی ورودی شامل $n/k$ زیردنباله است، هر کدام حاوی $k$عنصر. همه‌ی عناصر درون یک زیردنباله از عناصر زیردنباله‌ی قبلی کوچک‌تر، و از عناصر زیردنباله‌ی بعدی بزرگ‌تر هستند. بنابراین تنها کاری که برای مرتب‌سازی تمام دنباله باید انجام دهیم، مرتب‌سازی عناصر درون هر یک از $n/k$ زیردنباله است. نشان دهید که کران پایین تعداد مقایسه‌های مورد نیاز برای این مرتب‌سازی، $\Omega(n \lg k)$ است. (*راهنمایی:* ترکیب کردن کران‌های پایین برای تک‌تک زیردنباله‌ها کار دشواری نیست!)

## ۸-۲ مرتب‌سازی شمارشی

*مرتب‌سازی شمارشی* (counting sort) فرض می‌کند که هر یک از $n$ عنصر آرایه‌ی ورودی یک عدد صحیح بین ۰ و $k$ است، برای یک عدد صحیح $k$. وقتی $k = O(n)$، مرتب‌سازی در زمان $\theta(n)$ اجرا می‌شود.

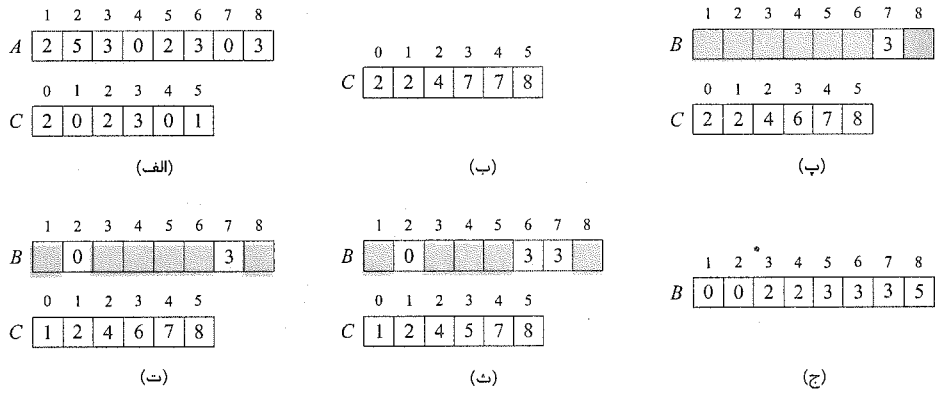
ایده‌ی اصلی مرتب‌سازی شمارشی این است که برای هر عنصر $x$، تعداد عناصر کمتر از $x$ را تعیین کنیم. می‌توان از این اطلاعات استفاده کرد و عنصر $x$ را مستقیماً در محل نهایی خود در آرایه‌ی خروجی قرار داد. برای مثال اگر ۱۷ عنصر کمتر از $x$ وجود داشته باشد، آن گاه $x$ متعلق به مکان خروجی ۱۸ است. برای حالت‌هایی که چندین عنصر با یک مقدار وجود دارند این طرح باید مقداری تغییر کند، چرا که نمی‌خواهیم چند عنصر را در یک مکان قرار دهیم.

در کد مرتب‌سازی شمارشی، فرض می‌کنیم که ورودی یک آرایه‌ی $A[1..n]$ است، و بنابراین $A.length = n$. به دو آرایه‌ی دیگر نیز احتیاج داریم: آرایه‌ی $B[1..n]$ خروجی مرتب شده را ذخیره می‌کند، و آرایه‌ی $C[0..k]$ فراهم کننده‌ی فضای موقت برای انجام عملیات است.

```
COUNTING-SORT(A, B, k)
1   let C[0..k] be a new array
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to A.length
5       C[A[j]] = C[A[j]] + 1
6       // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i – 1]
9       // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] – 1
```

شکل ۸–۲ نشان‌دهنده‌ی مرتب‌سازی شمارشی است. بعد از مقداردهی اولیه در حلقه‌ی for در خطوط ۲–۳، در حلقه‌ی for خطوط ۴–۵ هر یک از عناصر را بررسی می‌کنیم. اگر مقدار یک عنصر از آرایه‌ی ورودی $i$ باشد، $C[i]$ را افزایش می‌دهیم. بنابراین بعد از خط ۵، برای هر عدد صحیح $i = 0,1,...,k$، عنصر $C[i]$ شامل تعداد عناصری در $A$ است که مقدار $i$ دارند. در خطوط ۷–۸ با نگه داشتن یک مجموع پویا در آرایه‌ی $C$ برای هر $i = 0,1,...,k$ تعیین می‌کنیم که چند عنصر کمتر یا مساوی $i$ وجود دارد.

نهایتاً در حلقه‌ی for خطوط ۱۰–۱۲ عنصر $A[j]$ را در محل صحیح در آرایه‌ی خروجی $B$ قرار می‌دهیم. اگر تمام $n$ عنصر مجزا باشند، وقتی اولین بار وارد خط ۱۰ می‌شویم برای هر $A[j]$ مقدار $C[A[j]]$ موقعیت صحیح نهایی $A[j]$ در آرایه‌ی خروجی است، چرا که $C[A[j]]$ عنصر کوچک‌تر یا مساوی $A[j]$ وجود دارد. از آن جایی که ممکن است عناصر یکتا نباشند، هر بار که یک عنصر $A[j]$ را در آرایه‌ی $B$ قرار می‌دهیم $C[A[j]]$ را هم کاهش می‌دهیم. کاهش $C[A[j]]$ باعث می‌شود که عنصر ورودی بعدی با مقدار $A[j]$، در صورت وجود در مکان دقیقاً قبل از $A[j]$ در آرایه‌ی خروجی قرار گیرد.




شکل ۸-۲ عملیات COUNTING-SORT بر روی آرایه‌ی $A[1..8]$، که در آن هر عنصر $A$ یک عدد صحیح نامنفی نابیشتر از $k = 5$ است. (الف) آرایه‌ی $A$ و آرایه‌ی کمکی $C$ بعد از خط ۵. (ب) آرایه‌ی $C$ بعد از خط ۸ (پ)-(ت) آرایه‌ی خروجی $B$ و آرایه‌ی کمکی $C$ به ترتیب بعد از یک، دو، و سه تکرار از حلقه‌ی خطوط ۱۰-۱۲. فقط عناصر با سایه‌ی کمرنگ در $B$ پر شده‌اند. (ث) آرایه‌ی خروجی نهایی مرتب شده‌ی $B$.

مرتب‌سازی شمارشی به چه مقدار زمان نیاز دارد؟ حلقه‌ی for خطوط ۲-۳ در زمان $\theta(k)$ اجرا می‌شود، حلقه‌ی for خطوط ۴-۵ در زمان $\theta(n)$، حلقه‌ی for خطوط ۷-۸ در زمان $\theta(k)$، و حلقه‌ی for خطوط ۱۰-۱۲ در زمان $\theta(n)$. بنابراین زمان کل برابر است با $\theta(k+n)$. در عمل معمولاً زمانی از مرتب‌سازی شمارشی استفاده می‌کنیم که $k = O(n)$ عنصر داشته باشیم، که در این صورت زمان اجرا $\theta(n)$ خواهد بود.

مرتب‌سازی شمارشی کران پایین زمان اجرای $\Omega(n \lg n)$ را که در بخش ۸-۱ اثبات شد، می‌شکند، چرا که این الگوریتم، مرتب‌سازی را بر مبنای مقایسه‌ای انجام نمی‌دهد. در واقع در هیچ کجای کد، هیچ مقایسه‌ای بین عناصر آرایه‌ی ورودی انجام نمی‌شود. در عوض مرتب‌سازی شمارشی از مقادیر واقعی عناصر برای اندیس آن‌ها در یک آرایه استفاده می‌کند. زمانی که از مدل مقایسه‌ای برای مرتب‌سازی جدا می‌شویم، کران پایین $\theta(n \lg n)$ دیگر کاربردی نخواهد داشت.

یک خصوصیت مهم مرتب‌سازی شمارشی این است که این الگوریتم پایدار (stable) است: اعداد با مقدار یکسان، در آرایه‌ی خروجی همان ترتیبی را خواهند داشت که آرایه‌ی ورودی داشتند. یعنی تساوی بین دو عدد با این قانون شکسته می‌شود که هر عددی که در آرایه‌ی ورودی ابتدا ظاهر شده باشد، در آرایه‌ی خروجی نیز باید اول ظاهر شود. معمولاً خصوصیت پایداری فقط زمانی مهم است که داده‌های پیرو به وسیله‌ی عناصر آرایه حمل شوند. پایداری مرتب‌سازی شمارشی به دلیل دیگری مهم است: از این الگوریتم معمولاً به عنوان یک زیرروال در مرتب‌سازی مبنایی استفاده می‌شود. همان طور که در بخش بعد خواهیم دید، پایداری مرتب‌سازی شمارشی برای درستی مرتب‌سازی مبنایی حیاتی است.

### تمرین‌ها

۸-۲-۱ با استفاده از شکل ۸-۲ به عنوان یک مدل، عملیات COUNTING-SORT را روی آرایه‌ی $A = \langle 6,0,2,0,1,3,4,6,1,3,2 \rangle$ مشخص کنید.

۸-۲-۲ اثبات کنید که COUNTING-SORT پایدار است.

۸-۲-۳ فرض کنید سرآیند حلقه‌ی for در خط ۱۰ از رویه‌ی COUNTING-SORT به صورت زیر نوشته شود:

10 for $j = 1$ to $A.length$

نشان دهید که الگوریتم باز هم درست کار می‌کند. آیا این الگوریتم اصلاح شده پایدار است؟

۸-۲-۴ یک الگوریتم طراحی کنید که، با دریافت $n$ عدد ورودی بین ۰ و $k$، ابتدا یک پیش‌پردازش روی ورودی انجام داده و سپس به هر سؤال در مورد تعداد اعداد ووردی در بازه‌ی $[a\,..\,b]$ در زمان $O(1)$ پاسخ می‌دهد. پیش‌پردازش الگوریتم شما باید در زمان $\theta(n+k)$ انجام شود.

## ۸-۳ مرتب‌سازی مبنایی

مرتب‌سازی مبنایی الگوریتمی است که ماشین‌های مرتب‌سازی کارت‌ها از آن استفاده می‌کردند، که اکنون این ماشین‌ها را فقط در موزه‌های کامپیوتر می‌توانید پیدا کنید. کارت‌ها ۸۰ ستون دارند، و می‌توان در هر ستون یک سوراخ در یکی از ۱۲ جای ممکن ایجاد کرد. مرتب‌ساز می‌تواند به صورت مکانیکی «برنامه‌ریزی» شود که یک ستون از کارت‌ها را بررسی کرده و بسته به محل سوراخ، کارت‌ها را در یکی از ۱۲ مجموعه قرار دهد. سپس، اپراتور می‌تواند دسته‌های کارت را جمع کرده و طوری آن‌ها را روی هم قرار دهد که کارت‌هایی که محل اول آن‌ها سوراخ شده بر روی کارت‌هایی قرار گیرند که محل دوم آن‌ها سوراخ شده، و الی آخر.

برای ارقام دهدهی، در هر ستون فقط از یکی از ۱۰ مکان استفاده می‌شود. (از دو محل دیگر برای کدگذاری کاراکترهای غیرعددی استفاده می‌شود.) در این صورت یک عدد $d$ رقمی، $d$ ستون را اشغال خواهد کرد. از آن جایی که مرتب‌ساز کارت‌ها در هر زمان فقط می‌تواند یکی از ستون‌ها را بررسی کند، مسئله‌ی مرتب کردن $n$ کارت با اعداد $d$ رقمی، احتیاج به یک الگوریتم مرتب‌سازی دارد.

به طور شهودی یک راه حل این است که اعداد را طبق *با ارزشترین* رقم آن‌ها مرتب کنیم، به طور بازگشتی دسته‌های حاصل را مرتب کنیم، و سپس دسته‌های حاصل را به ترتیب با هم ادغام کنیم. متأسفانه از آن جایی که برای مرتب کردن هر دسته باید ۹ تای دیگر آن‌ها را کنار بگذاریم، این رویه

دسته‌های بسیاری از کارت‌ها تولید می‌کند که باید اطلاعات آن‌ها را نگه داریم. (تمرین ۸-۳-۵ را ببینید.)

بر خلاف شهود، مرتب‌سازی مبنایی برای مرتب‌سازی ابتدا *کم‌ارزشترین* رقم را مرتب می‌کند. سپس کارت‌ها در یک دسته ترکیب می‌شوند، بدین صورت که ابتدا کارت‌های دسته‌ی ۰ در دسته‌ی جدید قرار می‌گیرند، سپس کارت‌های دسته‌ی ۱، سپس دسته‌ی ۲ و الی آخر. سپس کارت‌ها از روی دومین رقم کم‌ارزش مرتب می‌شوند و مانند قبل دوباره با هم ترکیب می‌شوند. این روند ادامه می‌یابد تا کارت‌ها از روی تمام $d$ رقم مرتب شوند. بنابراین برای مرتب‌سازی فقط به $d$ عبور از روی کل دسته نیاز داریم. شکل ۸-۳ نحوه‌ی عمل مرتب‌سازی مبنایی بر روی یک «دسته»ی هفت‌تایی از اعداد سه رقمی را نشان می‌دهد.

در این الگوریتم پایدار بودن روش مرتب‌سازی ارقام حیاتی است. مرتب‌سازی انجام شده توسط ماشین مرتب‌سازی کارت‌ها پایدار است، ولی اپراتور باید هشیار باشد که ترتیب کارت‌هایی را که در یک دسته قرار می‌گیرند از بین نبرد، حتی با این که کارت‌های درون یک دسته در مکان مورد نظر رقم یکسانی دارند.

در یک کامپیوتر معمولی، که یک ماشین ترتیبی با دسترسی تصادفی (sequential random-access machine) است، بعضی مواقع از مرتب‌سازی مبنایی برای مرتب‌سازی رکوردهایی از اطلاعات که دارای فیلدهای مختلفی به عنوان کلید هستند، استفاده می‌شود. مثلاً ممکن است بخواهیم تاریخ‌ها را طبق سه کلید مرتب کنیم: سال، ماه، و روز. می‌توانیم از یک الگوریتم مرتب‌سازی با یک تابع مقایسه استفاده کنیم، که این تابع مقایسه بدین صورت است که دو تاریخ به عنوان ورودی دریافت کرده، و ابتدا سال‌های آن‌ها را مقایسه می‌کند، اگر مساوی بودند، ماه‌ها را مقایسه می‌کند، و اگر آن‌ها هم مساوی بودند، روزها را مقایسه می‌کند. به جای آن می‌توانیم این اطلاعات را سه بار با یک الگوریتم مرتب‌سازی پایدار مرتب کنیم: ابتدا روی روزها، سپس روی ماه‌ها، و در نهایت روی سال‌ها.

کد مرتب‌سازی مبنایی بسیار ساده است. رویه‌ی زیر فرض می‌کند که هر عنصر در آرایه‌ی $n$ عنصری $A$ یک عدد $d$ رقمی است، که در آن رقم ۱ کم‌ارزش‌ترین، و رقم $d$ پرارزش‌ترین رقم است.

| | | | |
|---|---|---|---|
| 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 عملیات مرتب‌سازی مبنایی بر روی یک لیست از هفت عدد ۳ رقمی. ستون سمت چپ لیست ورودی است. ستون‌های بعدی نشان‌دهنده‌ی لیست پس از مرتب‌سازی نسبت به مکان ارقام، از کم‌ارزش به پرارزش هستند. ستون سایه‌دار مکانی است که لیست قبل نسبت به آن مرتب شده تا لیست فعلی را بسازد.

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

**لم ۸-۳** با $n$ عدد $d$رقمی، که در آن هر رقم می‌تواند $k$ مقدار ممکن داشته باشد، RADIX-SORT ورودی را به درستی در زمان $\theta(d(n+k))$ مرتب می‌کند، اگر مرتب‌سازی پایداری که استفاده می‌کند در زمان $\theta(n+k)$ اجرا شود.

*اثبات* درستی مرتب‌سازی مبنایی از طریق استقرا روی ستونی که مرتب می‌شود، قابل اثبات است (تمرین ۸-۳-۳ را ببینید). تحلیل زمان اجرای مرتب‌سازی مبنایی به روش مرتب‌سازی پایداری بستگی دارد که به عنوان الگوریتم مرتب‌سازی واسطه استفاده می‌شود. وقتی هر رقم در فاصله‌ی ۰ تا $k-1$ باشد (یعنی بتواند $k$ مقدار ممکن را اختیار کند) و $k$ زیاد بزرگ نباشد، مرتب‌سازی شمارشی انتخاب بدیهی خواهد بود. هر عبور از روی $n$ عدد $d$ رقمی به زمان $\theta(n+k)$ احتیاج دارد. $d$ عبور انجام خواهد شد، و بنابراین زمان کلی اجرای مرتب‌سازی مبنایی عبارت است از $\theta(d(n+k))$. ■

وقتی $d$ ثابت باشد و $k=O(n)$، مرتب‌سازی مبنایی در زمان خطی اجرا می‌شود. به طور کلی‌تر در انتخاب دامنه‌ی مقدار ارقام خود مقداری انعطاف پذیری داریم.

**لم ۸-۴** به ازای هر $n$ عدد $b$ بیتی و هر عدد صحیح $r \le b$، رویه‌ی RADIX-SORT این اعداد را در زمان $\theta((b/r)(n+2^r))$ به درستی مرتب می‌کند، اگر مرتب‌سازی پایداری که استفاده می‌کند، برای ورودی‌های بین ۰ تا $k$ در زمان $\theta(n+k)$ اجرا شود.

*اثبات* برای یک مقدار $r \le b$، فرض می‌کنیم هر کلید $d = \lceil b/r \rceil$ رقم دارد، و هر رقم $r$ بیت فضا مصرف می‌کند. هر رقم یک عدد صحیح بین ۰ و $2^r-1$ است، بنابراین می‌توانیم از مرتب‌سازی شمارشی با $k = 2^r-1$ استفاده کنیم. (مثلاً می‌توانیم یک کلمه‌ی ۳۲ بیتی را به صورت ۴ رقم ۸ بیتی ببینیم، یعنی $b=32$، $r=8$، $k=2^r-1=255$، و $d=b/r=4$.) هر عبور مرتب‌سازی شمارشی در زمان $\theta(n+k)=\theta(n+2^r)$ انجام می‌شود، و در کل $d$ عبور داریم، که زمان اجرای کلی آن برابر خواهد بود با $\theta(d(n+2^r))=\theta((b/r)(n+2^r))$. ■

برای مقادیر داده شده‌ی $n$ و $b$ می‌خواهیم $r \le b$ را طوری انتخاب کنیم که عبارت $(b/r)(n+2^r)$ را کمینه کند. اگر $b < \lfloor \lg n \rfloor$، آن گاه برای هر مقدار $r$ داریم $(n+2^r)=\theta(n)$. بنابراین انتخاب $r=b$ زمان اجرای $(b/b)(n+2^b)=\theta(n)$ را نتیجه می‌دهد، که به صورت حدی بهینه است. اگر $b \ge \lfloor \lg n \rfloor$، در این صورت انتخاب $r=\lfloor \lg n \rfloor$ بهترین زمان اجرا (به صورت حدی) را می‌دهد: زمان اجرای $\theta(bn/\lg n)$. اگر $r$ را بالای $\lfloor \lg n \rfloor$ افزایش دهیم، جمله‌ی $2^r$ در صورت کسر سریع‌تر از جمله‌ی $r$

در مخرج رشد می‌کند، و بنابراین افزایش $r$ بالای $\lfloor \lg n \rfloor$ زمان اجرای $\Omega(bn/\lg n)$ را نتیجه می‌دهد. در عوض اگر $r$ را زیر $\lfloor \lg n \rfloor$ کاهش دهیم آن گاه جمله‌ی $b/r$ افزایش می‌یابد، و جمله‌ی $n + 2^r$ از مرتبه‌ی $\theta(n)$ باقی می‌ماند.

آیا مرتب‌سازی مبنایی به الگوریتم‌های مرتب‌سازی مقایسه‌ای، مانند مرتب‌سازی سریع ترجیح دارد؟ اگر $b = O(\lg n)$، همان‌گونه که اکثراً همین‌طور است، و انتخاب $r \approx \lg n$ را داشته باشیم، زمان اجرای مرتب‌سازی مبنایی از مرتبه‌ی $\theta(n)$ خواهد بود، که به نظر بهتر از امیدریاضی زمان اجرای $\theta(n \lg n)$ برای مرتب‌سازی سریع است. با این حال ضرایب ثابت مخفی در نماد $\theta$ با هم متفاوتند. با این که ممکن است مرتب‌سازی مبنایی نسبت به مرتب‌سازی سریع عبورهای کمتری از روی $n$ کلید داشته باشد، هر عبور مرتب‌سازی مبنایی ممکن است زمان بسیار بیشتری بگیرد. این که کدام الگوریتم ترجیح دارد به مشخصات پیاده‌سازی‌ها، ماشین مربوطه (مرتب‌سازی سریع معمولاً از حافظه‌های کش سخت‌افزاری بسیار مؤثرتر از مرتب‌سازی مبنایی استفاده می‌کند)، و داده‌های ورودی وابسته است. به علاوه نسخه‌ای از مرتب‌سازی مبنایی که از مرتب‌سازی شمارشی پایدار به عنوان مرتب‌سازی واسطه استفاده می‌کند، مرتب‌سازی را درجا انجام نمی‌دهد، کاری که بسیاری از الگوریتم‌های مقایسه‌ای با زمان $\theta(n \lg n)$ انجام می‌دهند. بنابراین وقتی حافظه‌ی اصلی بسیار باارزش است، ممکن است یک الگوریتم مرتب‌سازی درجا ترجیح داشته باشد.

## تمرین‌ها

۸-۳-۱ با استفاده از شکل ۸-۳ به عنوان یک مدل، عملیات RADIX-SORT را روی لیست کلمات زیر مشخص کنید: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

۸-۳-۲ کدام یک از الگوریتم‌های مرتب‌سازی زیر پایدار هستند: مرتب‌سازی درجی، مرتب‌سازی ادغامی، مرتب‌سازی هرمی، و مرتب‌سازی سریع؟ یک طرح ساده بدهید که هر الگوریتم مرتب‌سازی را پایدار می‌کند. طرح شما چه مقدار زمان و حافظه‌ی اضافی مصرف می‌کند؟

۸-۳-۳ با استفاده از استقرا نشان دهید که مرتب‌سازی مبنایی به درستی کار می‌کند. کجای اثبات شما به این فرض احتیاج دارد که مرتب‌سازی واسطه پایدار باشد؟

۸-۳-۴ نشان دهید که چطور می‌توان $n$ عدد در بازه‌ی ۰ و $n^2 - 1$ را در زمان $O(n)$ مرتب کرد.

★ ۸-۳-۵ در اولین الگوریتم مرتب‌سازی کارت‌ها در این بخش، در بدترین حالت دقیقاً چند بار عبور و مرتب‌سازی نیاز است تا اعداد دهدهی $d$ رقمی مرتب شوند؟ در بدترین حالت، اپراتور باید اطلاعات چند دسته کارت را نگه دارد؟

## ۸-۴ مرتب‌سازی سطلی

زمانی که ورودی توزیع یکنواخت داشته باشد، *مرتب‌سازی سطلی* (bucket sort) در زمان خطی اجرا می‌شود. مانند مرتب‌سازی شمارشی، مرتب‌سازی سطلی سریع است چون که فرضی در مورد ورودی‌های داده شده می‌کند. مرتب‌سازی شمارشی فرض می‌کند که ورودی شامل اعداد صحیحی در یک بازه‌ی کوچک است، در حالی که مرتب‌سازی سطلی فرض می‌کند که ورودی توسط یک رویه‌ی تصادفی تولید شده که عناصر را به صورت یکنواخت در بازه‌ی $[0,1)$ توزیع می‌کند. (برای تعریف توزیع یکنواخت به بخش پ-۲ مراجعه کنید.)

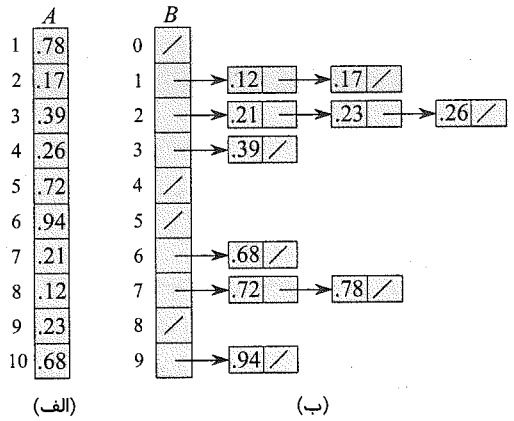
ایده‌ی اصلی مرتب‌سازی سطلی این است که بازه‌ی $[0,1)$ را به $n$ زیربازه با اندازه‌ی یکسان تقسیم، یا *سطل‌بندی*، و سپس $n$ عدد ورودی را درون سطل‌ها پخش کند. از آن جایی که ورودی‌ها به صورت یکنواخت در بازه‌ی $[0,1)$ توزیع شده‌اند، انتظار نداریم که اعداد زیادی در یک سطل قرار گیرند. برای تولید خروجی، به سادگی اعداد درون هر سطل را مرتب می‌کنیم و سپس به ترتیب اعداد سطل‌ها را در لیست خروجی قرار می‌دهیم.

کد زیر برای مرتب‌سازی سطلی فرض می‌کند که ورودی یک آرایه‌ی $A$ با $n$ عنصر است و هر $A[i]$ در رابطه‌ی $0 \le A[i] < 1$ صدق می‌کند. این کد احتیاج به یک آرایه‌ی کمکی $B[0 .. n-1]$ از لیست‌های پیوندی (سطل‌ها) دارد، و فرض می‌کند که مکانیزمی برای استفاده از این لیست‌ها در اختیار داریم. (بخش ۱۰-۲ طرز پیاده‌سازی عملیات پایه‌ای بر روی لیست‌های پیوندی را توضیح می‌دهد.)

```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4      make B[i] an empty list
5  for i = 1 to n
6      insert A[i] into list B[⌊n A[i]⌋]
7  for i = 0 to n − 1
8      sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

شکل ۸-۴ عملیات مرتب‌سازی سطلی را بر روی یک آرایه‌ی ورودی با ۱۰ عنصر را نشان می‌دهد.

برای این که ببینیم این الگوریتم به درستی کار می‌کند، دو عنصر $A[i]$ و $A[j]$ را در نظر بگیرید. بدون از دست دادن کلیت مسئله فرض کنید $A[i] \le A[j]$. از آن جایی که $\lfloor nA[i] \rfloor \le \lfloor nA[j] \rfloor$، عنصر $A[i]$ یا در همان سطل عنصر $A[j]$ قرار می‌گیرد، و یا در یک سطل با اندیس پایین‌تر. اگر $A[i]$ و $A[j]$ در یک سطل باشند، آن گاه حلقه‌ی for در خطوط ۷-۸ ترتیب آن‌ها را درست می‌کند. اگر $A[i]$ و $A[j]$ در سطل‌های متفاوت باشند، آن گاه خط ۹ آن‌ها را به ترتیب درست قرار می‌دهد. بنابراین مرتب‌سازی سطلی به درستی کار می‌کند.




**شکل ۸-٤** عملیات BUCKET-SORT. (الف) آرایه‌ی ورودی $A[1..10]$. (ب) آرایه‌ی $B[0..9]$ از لیست‌های پیوندی (سطل‌های) مرتب شده بعد از خط ۸ از الگوریتم. سطل $i$ شامل مقادیر بازه‌ی نیمه باز $[i/10,(i+1)/10)$ است. خروجی مرتب شده متشکل است از لیست‌های $B[0],B[1],...,B[9]$ که به ترتیب به یکدیگر متصل شده‌اند.

برای تحلیل زمان اجرا، مشاهده کنید که تمام خطوط به غیر از خط ۸ در بدترین حالت در زمان $O(n)$ اجرا می‌شوند. فقط این باقی می‌ماند که زمان کلی صرف شده توسط $n$ فراخوانی مرتب‌سازی درجی در خط ۸ را متعادل کنیم.

برای تحلیل هزینه‌ی فراخوانی‌های مرتب‌سازی درجی، فرض کنید $n_i$ متغیر تصادفی نشان دهنده‌ی تعداد عناصر در سطل $B[i]$ باشد. از آن جایی که زمان اجرای مرتب‌سازی درجی از درجه‌ی دو است (بخش ۲-۲ را ببینید)، زمان اجرای مرتب‌سازی سطلی برابر است با:

$$T(n)=\theta(n)+\sum_{i=0}^{n-1}O(n_i^2)$$

اکنون حالت متوسط زمان اجرای مرتب‌سازی سطلی را تحلیل می‌کنیم. این کار را با محاسبه‌ی امیدریاضی زمان اجرا بر روی توزیع ورودی‌ها انجام می‌دهیم. با امیدریاضی گرفتن از هر دو طرف و استفاده از خطی بودن امیدریاضی، داریم:

$$\begin{aligned}E[T(n)] &= E\left[\theta(n)+\sum_{i=0}^{n-1}O(n_i^2)\right]\\ &= \theta(n)+\sum_{i=0}^{n-1}E[O(n_i^2)] \qquad \text{(طبق خطی بودن امیدریاضی)} \qquad (8\text{-}1)\\ &= \theta(n)+\sum_{i=0}^{n-1}O(E[n_i^2]) \qquad \text{(طبق تساوی (پ-۲۲))}\end{aligned}$$

ادعا می‌کنیم که

$$E[n_i^2]=2-1/n \qquad (8\text{-}2)$$

برای $i = ۰,۱,...,n-۱$. اصلاً عجیب نیست که تمام سطل‌ها با اندیس $i$ مقدار $E[n_i^۲]$ یکسانی داشته باشند، چرا که هر مقدار در آرایه‌ی $A$ با احتمال برابر در هر یک از سطل‌ها قرار می‌گیرد. برای اثبات تساوی (۸-۲)، متغیرهای تصادفی شاخص زیر را تعریف می‌کنیم:

$$X_{ij} = I\{A[j] \text{ در سطل } i \text{ قرار گیرد}\}$$

برای $i = ۰,۱,...,n-۱$ و $j = ۱,۲,...,n$. بنابراین

$$n_i = \sum_{j=۱}^{n} X_{ij}$$

برای محاسبه‌ی $E[n_i^۲]$، توان دو را باز کرده و جمله‌ها را دوباره گروه‌بندی می‌کنیم:

$$\begin{aligned} E[n_i^۲] &= E\left[\left(\sum_{i=۱}^{n} X_{ij}\right)^۲\right] \\ &= E\left[\sum_{j=۱}^{n}\sum_{k=۱}^{n} X_{ij}X_{ik}\right] \\ &= E\left[\sum_{j=۱}^{n} X_{ij}^۲ + \sum_{۱\le j\le n}\ \sum_{\substack{۱\le k\le n \\ k\ne j}} X_{ij}X_{ik}\right] \\ &= \sum_{j=۱}^{n} E[X_{ij}^۲] + \sum_{۱\le j\le n}\ \sum_{\substack{۱\le k\le n \\ k\ne j}} E[X_{ij}X_{ik}] \end{aligned} \qquad (۸-۳)$$

که در آن خط آخر از خطی بودن امیدریاضی نتیجه می‌شود. دو سری را به صورت جداگانه ارزیابی می‌کنیم. متغیر تصادفی شاخص $X_{ij}$ با احتمال $۱/n$ برابر با ۱ و در غیر این صورت برابر با ۰ است، و بنابراین:

$$\begin{aligned} E[X_{ij}^۲] &= ۱\cdot\frac{۱}{n} + ۰\cdot\left(۱-\frac{۱}{n}\right) \\ &= \frac{۱}{n} \end{aligned}$$

وقتی $k \ne j$، متغیرهای $X_{ij}$ و $X_{ik}$ از یکدیگر مستقل‌اند، و از این رو:

$$\begin{aligned} E[X_{ij}X_{ik}] &= E[X_{ij}]E[X_{ik}] \\ &= \frac{۱}{n}\cdot\frac{۱}{n} \\ &= \frac{۱}{n^۲} \end{aligned}$$

با جایگذاری این دو مقدار امیدریاضی در تساوی (۸-۳) به دست می‌آوریم:

$$
\begin{aligned}
E[n_i^2] &= \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \ne j}} \frac{1}{n^2} \\
&= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 2 - \frac{1}{n}
\end{aligned}
$$

که تساوی (۸-۲) را اثبات می‌کند.

با استفاده از امیدریاضی در تساوی (۸-۱)، نتیجه می‌گیریم که حالت متوسط زمان اجرا برای مرتب‌سازی سطلی $\theta(n) + n \cdot O(2 - 1/n) = \theta(n)$ است.

حتی اگر ورودی به صورت یکنواخت توزیع نشده باشد، ممکن است مرتب‌سازی سطلی باز هم در زمان خطی اجرا شود. تا وقتی که ورودی این خاصیت را داشته باشد که مجموع مربعات اندازه‌ی سطل‌ها نسبت به تعداد کل عناصر خطی باشد، تساوی (۸-۱) به ما می‌گوید که مرتب‌سازی سطلی در زمان خطی اجرا می‌شود.

## تمرین‌ها

۸-۴-۱ با استفاده از شکل ۸-۴ به عنوان یک مدل، عملیات BUCKET-SORT را روی آرایه‌ی $A = \langle 0{,}79, 0{,}13, 0{,}16, 0{,}64, 0{,}39, 0{,}20, 0{,}89, 0{,}53, 0{,}71, 0{,}42 \rangle$ مشخص کنید.

۸-۴-۲ بدترین حالت زمان اجرای الگوریتم مرتب‌سازی سطلی چیست؟ چه تغییر کوچکی می‌تواند با حفظ امیدریاضی زمان اجرای خطی، بدترین حالت زمان اجرای این الگوریتم را به $O(n \lg n)$ تغییر دهد؟

۸-۴-۳ فرض کنید $X$ متغیر تصادفی تعداد شیرها در دو پرتاب یک سکه‌ی متقارن باشد. مقدار $E[X^2]$ چقدر است؟ مقدار $E^2[X]$ چطور؟

۸-۴-۴★ $n$ نقطه‌ی $p_i = (x_i, y_i)$ در دایره‌ی واحد داریم، به طوری که برای $i = 1, 2, \ldots, n$ داریم $0 < x_i^2 + y_i^2 \le 1$. فرض کنید نقاط به صورت یکنواخت پراکنده شده‌اند؛ یعنی احتمال یافتن یک نقطه در هر منطقه‌ای از دایره متناسب است با مساحت آن منطقه. یک الگوریتم با زمان اجرای متوسط $\theta(n)$ طراحی کنید که $n$ نقطه را به ترتیب فاصله‌ی آن‌ها از مرکز دایره، یعنی $d_i = \sqrt{x_i^2 + y_i^2}$ مرتب می‌کند. (*راهنمایی:* اندازه‌ی سطل‌ها در BUCKET-SORT را طوری طراحی کنید که بازتاب توزیع یکنواخت نقاط در دایره باشد.)

۸-۴-۵★ یک *تابع توزی احتمال* $P(x)$ برای یک متغیر تصادفی $X$ به صورت

$P(x) = \Pr\{X \leq x\}$ تعریف می‌شود. فرض کنید یک لیست از $n$ متغیر تصادفی $X_1, X_2, \ldots, X_n$ از یک تابع توزیع احتمال $P$ پیروی می‌کند، که در زمان $O(1)$ قابل محاسبه است. الگوریتمی ارائه کنید که این اعداد را در زمان متوسط خطی مرتب می‌کند.

## مسائل

### ۸-۱ *کران پایین احتمالاتی بر روی مرتب‌سازی‌های مقایسه‌ای*

در این مسئله، یک کران پایین احتمالاتی $\Omega(n \lg n)$ بر روی زمان اجرای هر مرتب‌سازی مقایسه‌ای قطعی یا تصادفی برای مرتب‌سازی $n$ عنصر مجزا اثبات می‌کنیم. با بررسی یک مرتب‌سازی مقایسه‌ای قطعی $A$ با درخت تصمیم $T_A$ شروع می‌کنیم. فرض می‌کنیم احتمال بروز هر یک از جایگشت‌های ورودی $A$ برابر است.

I. فرض کنید احتمال این که هر برگ $T_A$ با یک ورودی تصادفی قابل دسترس باشد، بر روی برگ نوشته شده است. اثبات کنید که دقیقاً بر روی $n!$ برگ احتمال $1/n!$ نوشته شده است، و احتمال بقیه‌ی برگ‌ها ۰ است.

II. فرض کنید $D(T)$ نشان دهنده‌ی طول مسیر خارجی درخت تصمیم $T$ باشد؛ یعنی $D(T)$ مجموع عمق تمام برگ‌های $T$ است. فرض کنید $T$ یک درخت تصمیم با $k > 1$ برگ باشد، و $LT$ و $RT$ به ترتیب زیردرخت‌های سمت چپ و راست $T$ باشند. نشان دهید که $D(T) = D(LT) + D(RT) + k$.

III. فرض کنید $d(k)$ مقدار کمینه‌ی $D(T)$ روی تمام درخت‌های تصمیم $T$ با $k > 1$ برگ باشد. نشان دهید که $d(k) = \min_{1 \leq i \leq k-1}\{d(i) + d(k-i) + k\}$. (*راهنمایی:* فرض کنید یک درخت تصمیم $T$ با $k$ برگ به این مقدار کمینه دست یافته است. فرض کنید $i_0$ تعداد برگ‌ها در $LT$ باشد، و $k - i_0$ تعداد برگ‌ها در $RT$.)

IV. اثبات کنید که برای مقادیر $k > 1$ و $i$ در فاصله‌ی $1 \leq i \leq k-1$، تابع $i \lg i + (k-i)\lg(k-i)$ در $i = k/2$ کمینه می‌شود. نتیجه بگیرید که $d(k) = \theta(k \lg k)$.

V. نشان دهید $D(T_A) = \theta(n! \lg(n!))$، و نتیجه بگیرید که زمان متوسط مرتب‌سازی $n$ عنصر $\theta(n \lg n)$ است.

اکنون یک مرتب‌سازی مقایسه‌ای *تصادفی* $B$ را در نظر بگیرید. می‌توانیم با گسترش مدل درخت تصمیم آن را طوری اصلاح کنیم که بتواند حالت‌های تصادفی را هم اداره کند، بدین صورت که دو نوع گره داشته باشیم: گره‌های مقایسه‌ای معمولی و گره‌های «تصادفی». گره‌های تصادفی، انتخاب‌های تصادفی به شکل $\text{RANDOM}(1, r)$ را که توسط الگوریتم $B$ انجام شده‌اند، مدل می‌کند؛ گره، $r$ فرزند دارد، و احتمال انتخاب شدن هریک از آن‌ها طی اجرای الگوریتم برابر است.

VI. نشان دهید که برای هر الگوریتم مرتب‌سازی مقایسه‌ای تصادفی $B$، یک مرتب‌سازی مقایسه‌ای قطعی $A$ وجود دارد که امیدریاضی تعداد مقایسه‌های آن بیشتر از $B$ نیست.

### ۸-۲ *مرتب‌سازی درجا در زمان خطی*

فرض کنید یک آرایه از $n$ رکورد داده شده برای مرتب‌سازی داریم، و کلید هر رکورد مقدار ۰ یا ۱ دارد. یک الگوریتم برای مرتب‌سازی چنین رکوردی ممکن است زیرمجموعه‌ای از سه خصوصیت مطلوب زیر را داشته باشد:

۱. الگوریتم در زمان $O(n)$ اجرا می‌شود.

۲. الگوریتم پایدار است.

۳. الگوریتم درجا مرتب می‌کند، و غیر از آرایه‌ی اصلی فقط به مقدار ثابتی فضای حافظه نیاز دارد.

I. الگوریتمی طراحی کنید که معیار ۱ و ۲ بالا را ارضا کند.

II. الگوریتمی طراحی کنید که معیار ۱ و ۳ بالا را ارضا کند.

III. الگوریتمی طراحی کنید که معیار ۲ و ۳ بالا را ارضا کند.

IV. آیا می‌توانید از هیچ یک از الگوریتم‌های مرتب‌سازی خود در بخش‌های I–III به عنوان متد مرتب‌سازی در خط ۲ رویه‌ی RADIX-SORT استفاده کنید، به طوری که RADIX-SORT مرتب‌سازی $n$ رکورد با کلیدهای $b$ بیتی را در زمان $O(bn)$ انجام دهد؟ توضیح دهید چطور، و یا بگویید چرا نه.

V. فرض کنید $n$ رکورد، کلیدهایی در فاصله‌ی ۱ تا $k$ دارند. نشان دهید چطور می‌توان مرتب‌سازی شمارشی را اصلاح کرد به شکلی که رکوردها را درجا و در زمان $O(n+k)$ مرتب کند. می‌توانید از $O(k)$ حافظه خارج از آرایه‌ی ورودی استفاده کنید. آیا الگوریتم شما پایدار است؟ (*راهنمایی:* چطور این کار را برای $k = ۳$ انجام می‌دهید؟)

### ۸-۳ *مرتب‌سازی اشیائی با طول متغیر*

I. یک آرایه از اعداد صحیح داریم، که ممکن است تعداد ارقام اعداد مختلف متفاوت باشد، ولی مجموع کل ارقام در *تمام* اعداد درون آرایه، $n$ است. نشان دهید که چطور می‌توان آرایه را در زمان $O(n)$ مرتب کرد.

II. یک آرایه از رشته‌ها داریم، که ممکن است تعداد کاراکترهای رشته‌های مختلف متفاوت باشد، ولی مجموع کل کاراکترها در *تمام* رشته‌های آرایه، $n$ است. نشان دهید که چطور می‌توان این رشته‌ها را در زمان $O(n)$ مرتب کرد.

(توجه کنید که ترتیب مورد نظر ترتیب استاندارد الفبا است؛ به عنوان مثال، $a < ab < b$.)

### ۸-۴ *کوزه‌های آب*

فرض کنید $n$ کوزه‌ی قرمز و $n$ کوزه‌ی آبی داریم که شکل و اندازه‌ی تمام آن‌ها متفاوت

است. ظرفیت تمام کوزه‌های قرمز با یکدیگر متفاوت است، و همین طور است ظرفیت کوزه‌های آبی. به علاوه برای هر کوزه‌ی قرمز یک کوزه‌ی آبی با همان ظرفیت وجود دارد، و بالعکس.

وظیفه‌ی شما این است که یک گروه‌بندی از کوزه‌ها به صورت جفت‌های قرمز و آبی بیابید که ظرفیت آن‌ها برابر باشد. برای انجام این کار می‌توانید عملیات زیر را انجام دهید: یک جفت کوزه انتخاب کنید که یکی از آن‌ها قرمز و دیگری آبی است، کوزه‌ی قرمز را با آب پر کنید، و سپس آب را در کوزه‌ی آبی بریزید. این عمل می‌تواند به شما بگوید که آیا ظرفیت دو کوزه برابر است، و یا کدام یک از آن‌ها ظرفیت بیشتری دارد. فرض کنید چنین مقایسه‌ای به یک واحد زمان نیاز دارد. هدف شما این است که الگوریتمی برای این گروه‌بندی بیابید که کم‌ترین تعداد مقایسه‌ها را انجام می‌دهد. به خاطر داشته باشید که نمی‌توانید دو کوزه‌ی هم‌رنگ را مستقیماً با یکدیگر مقایسه کنید.

I. یک الگوریتم قطعی طراحی کنید که با استفاده از $\theta(n^2)$ مقایسه، گروه‌بندی را انجام می‌دهد.

II. اثبات کنید که کران پایین تعداد مقایسه‌های مورد نیاز در هر الگوریتمی که این مسئله را حل می‌کند، $\Omega(n \lg n)$ است.

III. یک الگوریتم تصادفی بدهید که امیدریاضی تعداد مقایسه‌های آن $O(n \lg n)$ است، و اثبات کنید که این کران صحیح است. بدترین حالت تعداد مقایسه‌ها در الگوریتم شما چقدر است؟

## ۸-۵ *مرتب‌سازی تقریبی*

فرض کنید به جای مرتب‌سازی یک آرایه، فقط می‌خواهیم عناصر به طور متوسط افزایش یابند. به صورت دقیق‌تر به یک آرایه‌ی $A$ با $n$ عنصر، $k$-مرتب می‌گوییم اگر برای هر $i = 1,2,\ldots,n-k$ نامساوی زیر برقرار باشد:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \le \frac{\sum_{j=i+1}^{i+k} A[j]}{k}$$

I. ۱- مرتب بودن یک آرایه به چه معنی است؟

II. یک جایگشت از اعداد $1,2,\ldots,10$ بدهید که ۲- مرتب باشد، ولی مرتب نباشد.

III. نشان دهید که یک آرایه‌ی $n$ عنصری $k$- مرتب است اگر و تنها اگر $A[i] \le A[i+k]$ برای $i = 1,2,\ldots,n-k$.

IV. الگوریتمی ارائه کنید که در زمان $O(n \lg(n/k))$ یک آرایه‌ی $n$ عنصری را $k$-مرتب می‌کند. همچنین می‌توانیم برای $k$-مرتب‌سازی یک آرایه، زمانی که $k$ یک ثابت است، یک کران پایین بیابیم.

V. نشان دهید که یک آرایه‌ی $k$-مرتب با طول $n$ را می‌توان در زمان $O(n \lg k)$ مرتب کرد. (*راهنمایی:* از جواب تمرین ۶-۵-۹ استفاده کنید.)

VI. نشان دهید که وقتی $k$ یک ثابت است، برای $k$-مرتب‌سازی یک آرایه‌ی $n$ عنصری به $\theta(n \lg n)$ زمان احتیاج داریم. (*راهنمایی:* از جواب قسمت قبل به همراه کران پایین مرتب‌سازی‌های مقایسه‌ای استفاده کنید.)

## ۸-۶ *کران پایین بر روی ادغام لیست‌های مرتب*

مسئله‌ی ادغام دو لیست مرتب شده به کرات پیش می‌آید. از این کار به عنوان یک زیرروال در MERGE-SORT استفاده می‌شود، و رویه‌ی ادغام دو لیست مرتب با نام MERGE در بخش ۲-۳-۱ داده شده است. در این مسئله نشان می‌دهیم که کران پایین $2n - 1$ روی بیشترین تعداد مقایسه‌های مورد نیاز برای ادغام دو لیست مرتب که هر یک $n$ عنصر دارند، وجود دارد.

ابتدا کران پایین $2n - o(n)$ مقایسه را با استفاده از یک درخت تصمیم نشان می‌دهیم.

I. به چند طریق می‌توان $2n$ عدد را به دو لیست مرتب، هر یک با $n$ عدد تقسیم کرد؟

II. با استفاده از یک درخت تصمیم و جواب خود به قسمت قبل، نشان دهید که هر الگوریتم که به درستی دو لیست مرتب را ادغام می‌کند، حداقل باید $2n - o(n)$ مقایسه انجام دهد.

اکنون کران نزدیک‌تر $2n - 1$ را اثبات می‌کنیم.

III. نشان دهید که اگر دو عدد در ترتیب نهایی پشت سر هم باشند ولی در دو لیست مختلف قرار داشته باشند، باید با هم مقایسه شوند.

IV. از جواب قسمت قبل خود استفاده کنید و نشان دهید که کران پایین تعداد مقایسه‌ها برای ادغام دو لیست مرتب، $2n - 1$ است.

## ۸-۷ *لم مرتب‌سازی ۰-۱ و مرتب‌سازی ستونی*

یک عملیات *مقایسه-تعویض* (compare-exchange) بر روی دو عنصر آرایه‌ی $A[i]$ و $A[j]$، که در آن $i < j$، به صورت زیر است:

```
COMPARE-EXCHANGE(A, i, j)
1  if A[i] > A[j]
2      exchange A[i] with A[j]
```

پس از عملیات مقایسه-تعویض، می‌دانیم که $A[i] \le A[j]$.

یک *الگوریتم مقایسه-تعویض بی‌توجه* (oblivious compare-exchange algorithm) صرفا دنباله‌ای است از اعمال مقایسه-تعویض. اندیس مکان‌هایی که با یکدیگر مقایسه می‌شوند باید از قبل مشخص شده باشد، و با این که این اندیس‌ها می‌توانند به طول آرایه وابسته باشند، نه می‌توانند به مقدار عناصر آرایه بستگی داشته باشند، و نه به نتیجه‌ی هیچ یک از مقایسه‌های قبلی. به عنوان مثال در زیر مرتب‌سازی درجی را می‌بینیم که به صورت یک الگوریتم مقایسه-تعویض بی‌توجه ارائه شده است:

```
INSERTION-SORT(A)
1  for j = 2 to A.length
```

مسائل

```
2       for i = j – 1 downto 1
3           COMPARE-EXCHANGE(A, i, i+1)
```

*لم مرتب‌سازی ۰-۱* (0-1 sorting lemma) روشی بسیار قدرتمند ارائه می‌کند برای اثبات این که یک الگوریتم مقایسه-تعویض بی‌توجه، یک نتیجه‌ی مرتب شده به دست می‌دهد. این لم می‌گوید که اگر یک الگوریتم مقایسه-تعویض بی‌توجه به درستی تمام دنباله‌های ورودی حاوی اعداد ۰ و ۱ را مرتب کند، آن گاه تمام ورودی‌ها حاوی اعداد دلخواه را هم مرتب می‌کند.

برای اثبات لم مرتب‌سازی ۰-۱، عکس نقیض آن را اثبات خواهیم کرد: اگر یک الگوریتم مقایسه-تعویض بی‌توجه نتواند یک ورودی با اعداد دلخواه را مرتب کند، آن گاه یک ورودی ۰-۱ هم وجود دارد که نمی‌تواند مرتب کند. فرض کنید یک الگوریتم مقایسه-تعویض بی‌توجه X نمی‌تواند به درستی یک آرایه‌ی $A[1..n]$ را مرتب کند. فرض کنید $A[p]$ کوچک‌ترین مقدار در $A$ باشد که الگوریتم X در مکان اشتباه قرار می‌دهد، و فرض کنید $A[q]$ مقداری باشد که الگوریتم X اشتباهاً در مکان $A[p]$ قرار می‌دهد. یک آرایه‌ی $B[1..n]$ از ۰ ها و ۱ها تعریف می‌کنیم به صورت زیر:

$$B[i] = \begin{cases} 0 & \text{اگر } A[i] \le A[p] \\ 1 & \text{اگر } A[i] > A[p] \end{cases}$$

I. بحث کنید که $A[q] > A[p]$، و بنابراین $B[p] = 0$ و $B[q] = 1$.

II. برای تکمیل اثبات لم مرتب‌سازی ۰-۱، اثبات کنید که الگوریتم X نمی‌تواند آرایه‌ی $B$ را به درستی مرتب کند.

اکنون از لم مرتب‌سازی ۰-۱ استفاده کرده و نشان می‌دهیم که یک الگوریتم مرتب‌سازی خاص به درستی کار می‌کند. الگوریتم *مرتب‌سازی ستونی* (columnsort) بر روی یک آرایه‌ی مستطیلی با $n$ عنصر کار می‌کند. این آرایه $r$ سطر و $s$ ستون دارد (یعنی $n = rs$)، با سه محدودیت:

- $r$ باید زوج باشد،
- $s$ باید مقسوم‌علیهی از $r$ باشد، و
- $r \ge 2s^2$.

وقتی مرتب‌سازی ستونی پایان می‌یابد، آرایه به صورت *ستون-محور* (column-major order) مرتب شده است: وقتی ستون‌ها را از بالا به پایین و از چپ به راست می‌خوانیم، عناصر به صورت صعودی خواهند بود.

مرتب‌سازی ستونی در هشت مرحله انجام می‌شود، مستقل از مقدار $n$. مراحل فرد همگی یکسان هستند: مرتب‌سازی هر ستون به صورت جداگانه. هر مرحله‌ی زوج، یک بازآرایی ثابت

است. این مراحل به صورت زیر هستند:

۱. مرتب‌سازی تمام ستون‌ها.
۲. تبدیل آرایه به ترانهاده‌ی خود، ولی بازآرایی آن به صورتی که دوباره $r$ سطر و $s$ ستون داشته باشد. به عبارت دیگر، تبدیل چپ‌ترین ستون به $r/s$ سطر بالایی، به ترتیب؛ تبدیل ستون بعدی به $r/s$ ستون بعدی، به ترتیب؛ و الی آخر.
۳. مرتب‌سازی تمام ستون‌ها.
٤. انجام عکس بازآرایی انجام شده در مرحله‌ی ۲.
۵. مرتب‌سازی تمام ستون‌ها.
۶. انتقال نیمه‌ی بالایی هر ستون به نیمه‌ی پایینی همان ستون، و نیمه‌ی پایینی هر ستون به نیمه‌ی بالایی ستون بعدی در سمت راست. خالی گذاشتن نیمه‌ی بالایی چپ‌ترین ستون، و انتقال نیمه‌ی پایینی آخرین ستون به نیمه‌ی بالایی یک ستون جدید در سمت راست، و خالی گذاشتن نیمه‌ی پایینی ستون جدید.
۷. مرتب‌سازی تمام ستون‌ها.
۸. انجام عکس بازآرایی انجام شده در مرحله‌ی ۶.

شکل ۸-۵ یک نمونه از مراحل مرتب‌سازی ستونی را با $r = ۶$ و $s = ۳$ نشان می‌دهد. (با این که این مثال محدودیت $r \geq ۲s^۲$ را نقض می‌کند، ولی اتفاقاً الگوریتم به درستی کار می‌کند.)

| | | |
|---|---|---|
| 10 | 14 | 5 |
| 8 | 7 | 17 |
| 12 | 1 | 6 |
| 16 | 9 | 11 |
| 4 | 15 | 2 |
| 18 | 3 | 13 |

(الف)

| | | |
|---|---|---|
| 4 | 1 | 2 |
| 8 | 3 | 5 |
| 10 | 7 | 6 |
| 12 | 9 | 11 |
| 16 | 14 | 13 |
| 18 | 15 | 17 |

(ب)

| | | |
|---|---|---|
| 4 | 8 | 10 |
| 12 | 16 | 18 |
| 1 | 3 | 7 |
| 9 | 14 | 15 |
| 2 | 5 | 6 |
| 11 | 13 | 17 |

(پ)

| | | |
|---|---|---|
| 1 | 3 | 6 |
| 2 | 5 | 7 |
| 4 | 8 | 10 |
| 9 | 13 | 15 |
| 11 | 14 | 17 |
| 12 | 16 | 18 |

(ت)

| | | |
|---|---|---|
| 1 | 4 | 11 |
| 3 | 8 | 14 |
| 6 | 10 | 17 |
| 2 | 9 | 12 |
| 5 | 13 | 16 |
| 7 | 15 | 18 |

(ج)

| | | |
|---|---|---|
| 1 | 4 | 11 |
| 2 | 8 | 12 |
| 3 | 9 | 14 |
| 5 | 10 | 16 |
| 6 | 13 | 17 |
| 7 | 15 | 18 |

(چ)

| | | |
|---|---|---|
| | 5 | 10 | 16 |
| | 6 | 13 | 17 |
| | 7 | 15 | 18 |
| 1 | 4 | 11 | |
| 2 | 8 | 12 | |
| 3 | 9 | 14 | |

(ح)

| | | |
|---|---|---|
| | 4 | 10 | 16 |
| | 5 | 11 | 17 |
| | 6 | 12 | 18 |
| 1 | 7 | 13 | |
| 2 | 8 | 14 | |
| 3 | 9 | 15 | |

(خ)

| | | |
|---|---|---|
| 1 | 7 | 13 |
| 2 | 8 | 14 |
| 3 | 9 | 15 |
| 4 | 10 | 16 |
| 5 | 11 | 17 |
| 6 | 12 | 18 |

(د)

**شکل ۸-۵** مراحل مرتب‌سازی ستونی. (الف) آرایه‌ی ورودی با ۶ سطر و ۳ ستون. (ب) پس از مرتب‌سازی ستون‌ها در مرحله‌ی ۱. (پ) پس از ترانهاده گرفتن و تغییر شکل در مرحله‌ی ۲. (ت) پس از مرتب‌سازی ستون‌ها در مرحله‌ی ۳. (ث) پس از انجام مرحله‌ی ۴، که بازآرایی مرحله‌ی ۲ را برعکس می‌کند. (ج) پس از مرتب‌سازی ستون‌ها در مرحله‌ی ۵. (چ) پس از جابه‌جایی نیم‌ستون‌ها در مرحله‌ی ۶. (ح) پس از مرتب‌سازی ستون‌ها در مرحله‌ی ۷. (خ) پس از انجام مرحله‌ی ۸، که بازآرایی مرحله‌ی ۶ را برعکس می‌کند. اکنون آرایه به صورت ستون-محور مرتب شده است.

مسائل

III. بحث کنید که می‌توانیم مرتب‌سازی ستونی را به صورت یک الگوریتم مقایسه-تعویض بی‌توجه در نظر بگیریم، حتی اگر ندانیم که برای مرتب‌سازی در مراحل فرد از چه الگوریتمی استفاده می‌شود.

با این که باور این که مرتب‌سازی ستونی به درستی کار می‌کند، سخت به نظر می‌رسد، در این جا از لم مرتب‌سازی ۰-۱ استفاده می‌کنیم تا درستی آن را اثبات کنیم. لم مرتب‌سازی ۰-۱ در این جا به کار می‌رود چرا که می‌توانیم مرتب‌سازی ستونی را به صورت یک الگوریتم مقایسه-تعویض بی‌توجه در نظر بگیریم. چند تعریف به شما کمک خواهد کرد که از لم مرتب‌سازی ۰-۱ استفاده کنید. می‌گوییم ناحیه‌ای از یک آرایه *پاک* (clean) است اگر بدانیم که تماماً یا حاوی ۰ها است و یا حاوی ۱ها. در غیر این صورت ممکن است یک ناحیه هم حاوی ۰ باشد و هم ۱، که می‌گوییم *ناپاک* (dirty) است. از این پس فرض کنید که آرایه‌ی ورودی فقط حاوی ۰ها و ۱ها است، و می‌توانیم آن را به صورت یک آرایه با $r$ سطر و $s$ ستون در نظر بگیریم.

IV. اثبات کنید که پس از مراحل ۱-۳، آرایه حاوی چند سطر پاک از ۰ها در بالا، چند سطر پاک از ۱ها در پایین، و حداکثر $s$ سطر ناپاک در وسط است.

V. اثبات کنید که پس از مرحله‌ی ۴، اگر آرایه به صورت ستون-محور خوانده شود، با یک ناحیه‌ی پاک از ۰ها آغاز شده و با یک ناحیه‌ی پاک از ۱ها پایان می‌یابد، و ناحیه‌ای ناپاک از حداکثر $s^2$ عنصر در میانه دارد.

VI. اثبات کنید که مراحل ۵-۸ یک خروجی کاملاً مرتب شده‌ی ۰-۱ تولید می‌کنند. نتیجه بگیرید که مرتب‌سازی ستونی تمام ورودی‌ها با اعداد دلخواه را به درستی مرتب می‌کند.

VII. اکنون فرض کنید $s$ مقسوم‌علیه $r$ نیست. اثبات کنید که پس از مراحل ۱-۳، آرایه حاوی چند سطر پاک از ۰ها در بالا، چند سطر پاک از ۱ها در پایین، و حداکثر $2s-1$ سطر ناپاک در میانه است. برای این که در این حالت مرتب‌سازی ستونی به درستی کار کند، $r$ نسبت به $s$ چقدر باید بزرگ باشد؟

VIII. یک تغییر ساده در مرحله‌ی ۱ پیشنهاد کنید که به ما اجازه می‌دهد با حفظ محدودیت $r \geq 2s^2$، محدودیت بخش‌پذیری $r$ بر $s$ را حذف کنیم، و مرتب‌سازی ستونی همچنان به درستی کار کند.


# میانه‌ها و شاخص‌های ترتیبی

## ۹ـ۰ مقدمه

$i$امین *شاخص ترتیبی* (order statistic) در یک مجموعه‌ی $n$ عنصری، $i$امین عنصر کوچک آن آرایه است. به عنوان مثال، *کمینه‌ی* یک مجموعه از عناصر معادل است با اولین شاخص ترتیبی آن مجموعه ($i=1$)، و *بیشینه* معادل است با $n$امین شاخص ترتیبی ($i=n$). *میانه* به صورت غیر رسمی «نقطه‌ی وسطی» مجموعه است. وقتی $n$ فرد باشد میانه یکتا است، که همان شاخص ترتیبی $i=(n+1)/2$ می‌باشد. اگر $n$ زوج باشد دو میانه داریم، یکی با $i=n/2$ و دیگری با $i=n/2+1$. بنابراین صرف نظر از زوج یا فرد بودن $n$، میانه‌ها روی $i=\lfloor (n+1)/2 \rfloor$ (*میانه‌ی پایین*) و $i=\lceil (n+1)/2 \rceil$ (*میانه‌ی بالا*) هستند. در این کتاب برای راحتی، از عبارت «میانه» برای مشخص کردن میانه‌ی پایین استفاده می‌کنیم.

در این فصل مسئله‌ی انتخاب $i$امین شاخص ترتیبی در یک مجموعه از $n$ عدد مجزا مطرح می‌شود. برای راحتی فرض می‌کنیم که اعداد مجموعه یکتا هستند، با این که تقریباً هرکاری که انجام می‌دهیم برای مجموعه‌های با اعداد تکراری هم گسترش می‌یابد. *مسئله‌ی انتخاب* را می‌توان به صورت رسمی به شکل زیر تعریف کرد:

*ورودی:* مجموعه‌ی $A$ از $n$ عدد (مجزا) و یک عدد $i$، که $1 \le i \le n$.

*خروجی:* عنصر $x \in A$ که دقیقاً از $i-1$ عنصر دیگر $A$ بزرگ‌تر است.

مسئله‌ی انتخاب را می‌توان در زمان $O(n \lg n)$ حل کرد، چرا که می‌توانیم اعداد را با استفاده از مرتب‌سازی هرمی و یا مرتب‌سازی ادغامی مرتب کرده و به سادگی عنصر با اندیس $i$ را به خروجی

بدهیم. در این فصل الگوریتم‌های سریع‌تری برای این کار ارائه خواهد شد.

در بخش ۹-۱ مسئله‌ی انتخاب کمینه و بیشینه‌ی مجموعه‌ای از عناصر را بررسی می‌کنیم. مسئله‌ی انتخاب کلی بسیار جذاب‌تر است، که در دو بخش بعدی به آن رسیدگی می‌شود. در بخش ۹-۲ یک الگوریتم کاربردی بررسی می‌شود که به امیدریاضی زمان اجرای $O(n)$ دست می‌یابد. در بخش ۹-۳ یک الگوریتم با بار تئوری بیشتر معرفی می‌شود که زمان اجرای آن در بدترین حالت $O(n)$ است.

## ۹-۱ مقادیر کمینه و بیشینه

چند مقایسه برای یافتن مقدار کمینه‌ی یک مجموعه از $n$ عنصر لازم است؟ می‌توانیم به راحتی به یک کران بالای $n-1$ مقایسه دست یابیم: هر عنصر در مجموعه را بررسی می‌کنیم و کوچک‌ترین عنصری را که تا به حال یافته‌ایم، در حافظه نگه می‌داریم. در رویه‌ی زیر فرض می‌کنیم که این مجموعه در آرایه‌ی $A$ ذخیره شده است، که در آن $A.length = n$.

```
MINIMUM(A)
1  min = A[1]
2  for i = 2 to A.length
3      if min > A[i]
4          min = A[i]
5  return min
```

مسلماً یافتن بیشینه هم به طور مشابه می‌تواند با $n-1$ مقایسه انجام شود.

آیا این بهترین الگوریتم ممکن است؟ بله، چرا که می‌توانیم یک کران پایین $n-1$ مقایسه هم برای مسئله‌ی یافتن کمینه بیابیم. به الگوریتم تعیین کمینه‌ی یک مجموعه به شکل یک تورنومنت میان عناصر نگاه کنید. هر مقایسه معادل است با یک مسابقه در تورنومنت که در آن عنصر کوچک‌تر پیروز می‌شود. مشاهده‌ی کلیدی در این‌جا این است که هر عنصر، غیر از کوچک‌ترین عنصر، باید حداقل در یک مسابقه شکست بخورد. بنابراین $n-1$ مقایسه برای یافتن کمینه ضروری است، و الگوریتم MINIMUM نسبت به تعداد مقایسه‌های انجام شده بهینه می‌باشد.

### کمینه و بیشینه‌ی هم‌زمان

در بعضی کاربردها باید هر دو مقدار کمینه و بیشینه‌ی یک مجموعه‌ی $n$ عنصری را بیابیم. مثلاً ممکن است یک برنامه‌ی گرافیکی بخواهد یک مجموعه از داده‌های $(x, y)$ را بر روی یک صفحه‌ی نمایش مستطیلی و یا هر خروجی گرافیکی دیگر مقیاس کند. برای انجام این کار، برنامه ابتدا باید کمینه و بیشینه را روی هر محور تشخیص دهد.

طراحی الگوریتمی که هم زمان مقادیر کمینه و بیشینه‌ی $n$ عنصر را با $\theta(n)$ مقایسه بیابد اصلاً سخت نیست، که به صورت حدی هم بهینه است. فقط کافی است که به سادگی کمینه و بیشینه را به صورت جداگانه بیابیم، هر کدام با استفاده از $n-1$ مقایسه، و مجموع $2n-2$ مقایسه.

در واقع، حداکثر $\lfloor n/2 \rfloor$ ۳ مقایسه برای یافتن مقادیر کمینه و بیشینه کافی است. استراتژی این است که عناصر کمینه و بیشینه‌ای که تا به حال دیده شده‌اند را به صورت هم‌زمان نگه داریم. به جای بررسی عناصر به صورت مقایسه‌ی آن‌ها با هر دوی کمینه و بیشینه با هزینه‌ی ۲ مقایسه برای هر عنصر، عناصر را به صورت جفتی بررسی می‌کنیم. ابتدا یک جفت از عناصر را *با یکدیگر* مقایسه می‌کنیم، و سپس عنصر بزرگ‌تر را با بیشینه‌ی فعلی، و عنصر کوچک‌تر را با کمینه‌ی فعلی مقایسه می‌کنیم، که هزینه‌ی آن ۳ مقایسه برای ۲ عنصر است.

تعیین مقادیر اولیه برای کمینه و بیشینه‌ی فعلی بستگی به فرد یا زوج بودن $n$ دارد. اگر $n$ فرد باشد، هر دوی کمینه و بیشینه را برابر با عنصر اول مجموعه قرار می‌دهیم، و سپس بقیه‌ی عناصر را به صورت جفتی بررسی می‌کنیم. اگر $n$ زوج باشد، یک مقایسه برای دو عنصر اول انجام می‌دهیم تا مقادیر اولیه‌ی کمینه و بیشینه را تعیین کنیم، و سپس بقیه‌ی عناصر را به صورت جفتی بررسی می‌کنیم، مانند حالت فرد بودن $n$.

اجازه دهید تعداد کل مقایسه‌های لازم را تحلیل کنیم. اگر $n$ فرد باشد، $\lfloor n/2 \rfloor$ ۳ مقایسه انجام می‌دهیم. اگر $n$ زوج باشد، ۱ مقایسه‌ی اولیه، به علاوه‌ی $3(n-2)/2$ مقایسه خواهیم داشت، که برابر است با $3n/2-2$. بنابراین در هر دو حالت، تعداد کل مقایسه‌ها حداکثر برابر است با $\lfloor n/2 \rfloor$ ۳.

### تمرین‌ها

۹-۱-۱ نشان دهید که از میان $n$ عنصر، دومین عنصر کوچک را می‌توان با $n+\lceil \lg n \rceil -2$ مقایسه در بدترین حالت پیدا کرد. (*راهنمایی:* کوچک‌ترین عنصر را هم بیابید.)

۹-۱-۲★ نشان دهید که در بدترین حالت، $\lceil 3n/2 \rceil -2$ مقایسه برای یافتن بیشینه و کمینه‌ی $n$ عدد ضروری است. (*راهنمایی:* تعداد اعدادی که ممکن است بیشینه یا کمینه باشند را در نظر بگیرید، و تعیین کنید که یک مقایسه چگونه بر روی این اعداد تأثیر می‌گذارد.)

## ۹-۲ انتخاب با زمان اجرای مورد انتظار خطی

به نظر می‌رسد مسئله‌ی کلی انتخاب مشکل‌تر از مسئله‌ی ساده‌ی یافتن مقدار کمینه باشد. با این حال به طرزی شگفت‌آور، زمان اجرای حدی هر دو مسئله یکی است: $\theta(n)$. در این بخش یک الگوریتم تقسیم و حل برای مسئله‌ی انتخاب معرفی می‌کنیم. الگوریتم RANDOMIZED-SELECT از مدل الگوریتم مرتب‌سازی سریع در فصل ۷ استفاده می‌کند. مانند مرتب‌سازی سریع، ایده این است که آرایه‌ی ورودی را به صورت بازگشتی قسمت‌بندی کنیم. ولی برخلاف مرتب‌سازی سریع که به صورت بازگشتی هر دو قسمت را بررسی می‌کند، RANDOMIZED-SELECT فقط بر روی یک قسمت کار می‌کند. این اختلاف در تحلیل زمان اجرا خود را نشان می‌دهد: در حالی که امیدریاضی

زمان اجرای مرتب‌سازی سریع $\theta(n \lg n)$ است، امیدریاضی زمان اجرای RANDOMIZED-SELECT برابر است با $\theta(n)$، با فرض این که مقدار عناصر یکتا باشد.

RANDOMIZED-SELECT از رویه‌ی RANDOMIZED-PARTITION که در بخش ۷-۳ معرفی شد استفاده می‌کند. بنابراین مانند RANDOMIZED-QUICKSORT یک الگوریتم تصادفی است، چرا که قسمتی از رفتار آن توسط خروجی یک تولید کننده‌ی اعداد تصادفی تعیین می‌شود. رویه‌ی RANDOMIZED-SELECT زیر، $i$امین عنصر کوچک آرایه‌ی $A[p \,..\, r]$ را بازمی‌گرداند.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p = r
2      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      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

رویه‌ی RANDOMIZED-SELECT به صورت زیر کار می‌کند. خط ۱ حالت پایه‌ی بازگشت را چک می‌کند، که در آن زیرآرایه‌ی $A[p \,..\, r]$ فقط حاوی یک عنصر است. در این حالت $i$ باید برابر با ۱ باشد، و در خط ۲ به سادگی $A[p]$ را به عنوان $i$امین عنصر کوچک بازمی‌گردانیم. در غیر این صورت در خط ۳ رویه‌ی RANDOMIZED-PARTITION فراخوانی می‌شود، که آرایه‌ی $A[p \,..\, r]$ را به دو زیرآرایه‌ی (احتمالاً تهی) $A[p \,..\, q-1]$ و $A[q+1 \,..\, r]$ تقسیم می‌کند، به طوری که هر عنصر $A[p \,..\, q-1]$ کوچک‌تر یا مساوی $A[q]$ است، که خود از هر یک از عناصر $A[q+1 \,..\, r]$ کوچک‌تر است. مانند مرتب‌سازی سریع، $A[q]$ را با نام عنصر محور می‌شناسیم. خط ۴ رویه‌ی RANDOMIZED-SELECT عدد $k$ را محاسبه می‌کند، که تعداد عناصر درون زیرآرایه‌ی $A[p \,..\, q]$ (عناصر پایین آرایه) است، به علاوه‌ی یکی برای عنصر محور. سپس خط ۵ چک می‌کند که آیا $A[q]$، $i$امین عنصر کوچک آرایه هست یا خیر. اگر این طور باشد $A[q]$ در خط ۶ بازگردانده می‌شود. در غیر این صورت الگوریتم تشخیص می‌دهد که $i$امین عنصر کوچک در کدام یک از دو زیرآرایه‌ی $A[p \,..\, q-1]$ و $A[q+1 \,..\, r]$ قرار دارد. اگر $i < k$، آن گاه عنصر مورد نظر در قسمت پایین قرار دارد و این عنصر به صورت بازگشتی در خط ۸ از زیرآرایه‌ی مربوطه انتخاب می‌شود. از طرف دیگر اگر $i > k$، آن گاه عنصر مورد نظر در قسمت بالا قرار دارد. از آن جایی که در این لحظه $k$ مقدار کوچک‌تر از $i$امین عنصر کوچک $A[p \,..\, r]$ می‌شناسیم — عناصر $A[p \,..\, q]$ — عنصر مورد نظر $(i-k)$امین عنصر کوچک زیرآرایه‌ی $A[q+1 \,..\, r]$ است، که به صورت بازگشتی در خط ۹ محاسبه می‌شود. به نظر می‌آید که در این کد به زیرآرایه‌های با ۰ عنصر هم اجازه‌ی فراخوانی بازگشتی داده می‌شود، ولی در تمرین ۹-۲-۱ از شما خواسته می‌شود که نشان دهید چنین چیزی اتفاق نخواهد افتاد.

بدترین حالت زمان اجرای RANDOMIZED-SELECT برابر است با $\theta(n^2)$، حتی برای یافتن کمینه، چرا که ممکن است در نهایت بدشانسی باشیم و همیشه تقسیم‌بندی نسبت به بزرگ‌ترین عنصر موجود انجام شود، و تقسیم‌بندی هم به زمان $\theta(n)$ نیاز دارد. با این حال خواهیم دید که امیدریاضی زمان اجرای این الگوریتم خطی است، و از آن جایی که تصادفی است، هیچ ورودی خاصی بدترین حالت زمان اجرا را بر آن تحمیل نمی‌کند.

امیدریاضی زمان اجرای RANDOMIZED-SELECT بر روی یک آرایه‌ی ورودی $A[p\,..\,r]$ با $n$ عنصر، یک متغیر تصادفی است که آن را با $T(n)$ مشخص می‌کنیم، و می‌توانیم به صورت زیر یک کران بالا بر روی $E[T(n)]$ تعیین کنیم. رویه‌ی RANDOMIZED-PARTITION با احتمال برابر هر یک از عناصر را به عنوان محور بازمی‌گرداند. بنابراین برای هر $k$ به طوری که $1 \le k \le n$، زیرآرایه‌ی $A[p\,..\,q]$ با احتمال $1/n$ حاوی $k$ عنصر است (که تمام آن‌ها کوچک‌تر یا مساوی محور هستند). برای $k = 1,2,...,n$، متغیر تصادفی شاخص $X_k$ را بدین صورت تعریف می‌کنیم:

$$X_k = I\{\text{زیرآرایه‌ی } A[p\,..\,q] \text{ دقیقاً } k \text{ عنصر داشته باشد}\}$$

و بنابراین، با فرض این که عناصر یکتا هستند، داریم:

$$E[X_k] = 1/n \qquad (۹-۱)$$

وقتی RANDOMIZED-SELECT را فراخوانی و $A[q]$ را به عنوان عنصر محور انتخاب می‌کنیم، هنوز نمی‌دانیم که آیا در همان جا با جواب درست رویه را خاتمه می‌دهیم، روی زیرآرایه‌ی $A[p\,..\,q-1]$ فراخوانی بازگشتی را انجام می‌دهیم، و یا روی زیرآرایه‌ی $A[q+1\,..\,r]$ رویه را فراخوانی می‌کنیم. این تصمیم بستگی به این دارد که $i$امین عنصر کوچک نسبت به $A[q]$ در کجا قرار داشته باشد. با فرض این که $T(n)$ صعودی اکید است، می‌توانیم زمان مورد نیاز برای فراخوانی بازگشتی را به کمک زمان مورد نیاز برای فراخوانی بازگشتی بر روی بزرگ‌ترین ورودی ممکن کران‌دار کنیم. به صورت دیگر، برای یافتن یک کران بالا فرض می‌کنیم که $i$امین عنصر کوچک همیشه در زیرآرایه‌ی بزرگ‌تر باشد. برای یک فراخوانی مشخص از RANDOMIZED-SELECT، متغیر تصادفی شاخص $X_k$ دقیقاً برای یک مقدار از $k$ مقدار ۱ دارد، و برای بقیه‌ی مقادیر $k$ برابر با ۰ است. وقتی $X_k = 1$، اندازه‌ی دو زیرآرایه‌ای که باید بر روی آن‌ها بازگشت را انجام دهیم $k-1$ و $n-k$ است. بنابراین رابطه‌ی 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 امیدریاضی گرفتن از دو طرف، داریم

$$
\begin{aligned}
&E[T(n)] \\
&\le E\left[\sum_{i=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\
&= \sum_{k=1}^{n} E[X_k \cdot T(\max(k-1, n-k))] + O(n) && \text{(طبق خطی بودن امیدریاضی)} \\
&= \sum_{k=1}^{n} E[X_k] \cdot E[T(\max(k-1, n-k))] + O(n) && \text{(طبق تساوی (پ-۲۴))} \\
&= \sum_{k=1}^{n} \frac{1}{n} E[T(\max(k-1, n-k))] + O(n) && \text{(طبق تساوی (۹-۱))}
\end{aligned}
$$

برای به کار بردن تساوی (پ-۲۴)، قبول می‌کنیم که $X_k$ و $T(\max(k-1, n-k))$ متغیرهای تصادفی مستقل هستند. تمرین ۹-۲-۲ از شما می‌خواهد که این حکم را اثبات کنید.

اجازه دهید عبارت $\max(k-1, n-k)$ را در نظر بگیریم. داریم:

$$
\max(k-1, n-k) = \begin{cases} k-1 & \text{اگر } k > \lfloor k/2 \rfloor \\ n-k & \text{اگر } k \le \lceil n/2 \rceil \end{cases}
$$

اگر $n$ زوج باشد، هر جمله از $T(\lceil n/2 \rceil)$ تا $T(n-1)$ دقیقاً دو بار در سری ظاهر می‌شود، و اگر $n$ فرد باشد، تمام این جملات دو بار ظاهر می‌شوند، و فقط $T(\lfloor n/2 \rfloor)$ یک بار ظاهر می‌شود. بنابراین داریم

$$
E[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n)
$$

با استفاده از روش جانشین‌سازی نشان می‌دهیم که $E[T(n)] = O(n)$. فرض کنید $T(n) \le cn$ برای یک ثابت $c$ که شرط اولیه‌ی بازگشت را ارضا می‌کند. فرض می‌کنیم که $T(n) = O(1)$ برای $n$های کوچک‌تر از یک ثابت خاص؛ این ثابت را بعداً تعیین می‌کنیم. همچنین یک ثابت $a$ در نظر می‌گیریم به طوری که تابع نشان داده شده توسط جمله‌ی $O(n)$ در بالا (که زمان اجرای قسمت غیربازگشتی الگوریتم را نشان می‌دهد) برای تمام $n > 0$ از بالا توسط $an$ محدود شده باشد. با استفاده از این فرض استقرا داریم:

$$
\begin{aligned}
E[T(n)] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k\right) + an \\
&= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2}\right) + an
\end{aligned}
$$

$$
\begin{aligned}
&\le \frac{۲c}{n}\left(\frac{(n-۱)n}{۲} - \frac{(n/۲-۲)(n/۲-۱)}{۲}\right) + an \\
&= \frac{۲c}{n}\left(\frac{n^۲-n}{۲} - \frac{n^۲/۴ - ۳n/۲ - ۲}{۲}\right) + an \\
&= \frac{c}{n}\left(\frac{۳n^۲}{۴} + \frac{n}{۲} - ۲\right) + an \\
&= c\left(\frac{۳n}{۴} + \frac{۱}{۲} - \frac{۲}{n}\right) + an \\
&\le \frac{۳cn}{۴} + \frac{c}{۲} + an \\
&= cn - \left(\frac{cn}{۴} - \frac{c}{۲} - an\right)
\end{aligned}
$$

برای تکمیل اثبات باید نشان دهیم که برای $n$های به اندازه‌ی کافی بزرگ، این عبارت آخر حداکثر برابر با $cn$ است، و یا به طور مشابه $cn/۴ - c/۲ - an \ge ۰$. اگر $c/۲$ را به هر دو طرف اضافه کنیم و از $n$ فاکتور بگیریم، بدست می‌آوریم $n(c/۴ - a) \ge c/۲$. اگر ثابت $c$ را طوری انتخاب کنیم که $c/۴ - a > ۰$، یعنی $c > ۴a$، می‌توانیم هر دو طرف را بر $c/۴ - a$ تقسیم کنیم، که به دست می‌دهد:

$$n \ge \frac{c/۲}{c/۴ - a} = \frac{۲c}{c - ۴a}$$

بنابراین اگر فرض کنیم برای $n < ۲c/(c-۴a)$ داریم $T(n) = O(۱)$، آن گاه $E[T(n)] = O(n)$. نتیجه می‌گیریم که می‌توان هر شاخص ترتیبی، و به طور خاص میانه، را در زمان متوسط خطی یافت.

## تمرین‌ها

۹-۲-۱ نشان دهید که در RANDOMIZED-SELECT هیچ وقت یک فراخوانی بازگشتی بر روی یک آرایه با طول ۰ اتفاق نمی‌افتد.

۹-۲-۲ نشان دهید که متغیر تصادفی شاخص $X_k$ و مقدار $T(\max(k-۱, n-k))$ از یکدیگر مستقل‌اند.

۹-۲-۳ یک نسخه‌ی بازگشتی از RANDOMIZED-SELECT بنویسید.

۹-۲-۴ فرض کنید از رویه‌ی RANDOMIZED-SELECT برای انتخاب عنصر کمینه‌ی آرایه‌ی $A = \langle ۳,۲,۹,۰,۷,۵,۴,۸,۶,۱ \rangle$ استفاده می‌کنیم. یک دنباله از تقسیم‌بندی‌ها ارائه کنید که به کارایی بدترین حالت RANDOMIZED-SELECT منجر شود.

## ۹-۳ انتخاب در بدترین حالت زمان اجرای خطی

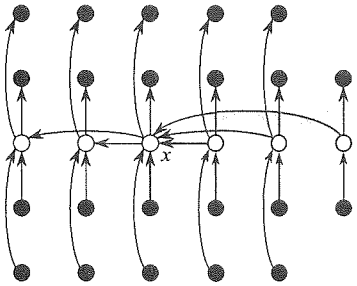
اکنون یک الگوریتم انتخاب ارائه می‌کنیم که بدترین حالت زمان اجرای آن $O(n)$ است. مانند RANDOMIZED-SELECT، الگوریتم SELECT عنصر مورد نظر را با تقسیم‌بندی آرایه‌ی ورودی به صورت بازگشتی می‌یابد. با این حال ایده‌ی پشت این الگوریتم این است که *تضمین* کنیم تقسیم‌بندی آرایه‌ی ورودی خوب انجام می‌شود. SELECT از الگوریتم تقسیم‌بندی قطعی استفاده شده در مرتب‌سازی سریع استفاده می‌کند (بخش ۷-۱ را ببینید)، که بدین صورت اصلاح شده است که عنصر محور را به عنوان پارامتر ورودی می‌گیرد.

الگوریتم SELECT، $i$امین عنصر کوچک یک آرایه‌ی ورودی با $n > ۱$ عنصر را با انجام مراحل زیر پیدا می‌کند. (اگر $n = ۱$، آن گاه SELECT به سادگی تنها عنصر ورودی خود را به عنوان $i$امین عنصر کوچک بازمی‌گرداند.)

۱. تقسیم $n$ عنصر آرایه‌ی ورودی به $\lfloor n/۵ \rfloor$ گروه ۵ عنصری، و حداکثر یک گروه متشکل از $n \bmod ۵$ عنصر باقی مانده.
۲. یافتن میانه‌ی هر یک از $\lceil n/۵ \rceil$ گروه، با مرتب‌سازی گروه‌ها (با استفاده از مرتب‌سازی درجی) و سپس انتخاب میانه‌ی آن‌ها از لیست‌های مرتب شده.
۳. استفاده از SELECT به صورت بازگشتی برای یافتن عنصر $x$، میانه‌ی این $\lceil n/۵ \rceil$ میانه‌ی یافته شده در مرحله‌ی ۲. (اگر تعداد میانه‌ها زوج باشد، آن گاه طبق قرارداد اول بخش $x$ میانه‌ی پایین خواهد بود.)
۴. تقسیم‌بندی آرایه‌ی ورودی نسبت به $x$ (میانه‌ی میانه‌ها) با استفاده از نسخه‌ی اصلاح شده‌ی PARTITION. فرض کنید $k$ یکی بیشتر از تعداد عناصر در قسمت پایین باشد، یعنی $x$ همان $k$امین عنصر کوچک آرایه است و $n-k$ عنصر در قسمت بالا قرار دارند.
۵. بازگرداندن $x$ در صورت برقراری $i = k$. در غیر این صورت، استفاده از SELECT به صورت بازگشتی برای یافتن $i$امین عنصر کوچک در قسمت پایین در صورتی که $i < k$، و یا $(i-k)$امین عنصر کوچک در قسمت بالا در صورتی که $i > k$.

برای تحلیل زمان اجرای SELECT، ابتدا یک کران پایین بر روی تعداد عناصری که بزرگ‌تر از عنصر محور $x$ هستند تعیین می‌کنیم. شکل ۹-۱ برای تصور این قضیه مفید است. حداقل نیمی از میانه‌های یافت شده در مرحله‌ی دو بزرگ‌تر[۱] از میانه‌ی میانه‌ها هستند. بنابراین حداقل نیمی از $\lceil n/۵ \rceil$ گروه حاوی سه عنصر هستند که از $x$ بزرگ‌تر است، غیر از یک گروه که کم‌تر از ۵ عنصر دارد (در صورتی که $n$ بر ۵ بخش‌پذیر نباشد) و همان گروهی که $x$ در آن قرار دارد. با حذف این دو گروه، نتیجه می‌شود که تعداد عناصری که از $x$ بزرگ‌ترند حداقل برابر است با:

[۱] به خاطر این که فرض کرده‌ایم اعداد یکتا هستند، می‌توانیم بگوییم «بزرگ‌تر از» و «کوچک‌تر از» بدون این که نگران حالت تساوی باشیم.




تحلیل الگوریتم SELECT. در شکل $n$ عنصر با دایره‌های کوچک مشخص شده‌اند، و هر گروه یک ستون را اشغال کرده است. میانه‌های گروه‌ها سفید شده‌اند، و میانه‌ی میانه‌ها با $x$ مشخص شده است. (وقتی می‌خواهیم میانه‌ی یک گروه با تعداد زوجی عنصر را بیابیم، از میانه‌ی پایینی استفاده می‌کنیم.) فلش‌ها از عناصر بزرگ‌تر به عناصر کوچک‌تر کشیده شده‌اند، که به کمک آن می‌توان دید که سه تا از ۵ عنصر هر یک از گروه‌های کامل سمت راست $x$ از $x$ بزرگ‌ترند، و ۳ تا از ۵ عنصر گروه‌های سمت چپ $x$ از $x$ کوچک‌ترند. عناصر بزرگ‌تر از $x$ با زمینه‌ی سایه‌دار مشخص شده‌اند.

$$۳\left(\left\lceil\frac{۱}{۲}\left\lceil\frac{n}{۵}\right\rceil\right\rceil-۲\right)\geq\frac{۳n}{۱۰}-۶$$

به طور مشابه تعداد عناصری که از $x$ کوچک‌ترند هم حداقل $۳n/۱۰-۶$ است. بنابراین در بدترین حالت، در مرحله‌ی ۵، رویه‌ی SELECT حداکثر بر روی $۷n/۱۰+۶$ عنصر به صورت بازگشتی فراخوانی می‌شود.

اکنون می‌توانیم یک رابطه‌ی بازگشتی $T(n)$ برای بدترین حالت زمان اجرای الگوریتم SELECT بیابیم. مراحل ۱، ۲، و ۴ در زمان $O(n)$ اجرا می‌شوند. (مرحله‌ی ۲ شامل $O(n)$ فراخوانی مرتب‌سازی درجی روی مجموعه‌هایی با اندازه‌ی $O(۱)$ است.) زمان اجرای مرحله‌ی ۳، برابر با $T(\lceil n/۵\rceil)$ است، و مرحله‌ی ۵ حداکثر $T(۷n/۱۰+۶)$ زمان مصرف می‌کند، با فرض این که $T$ اکیداً صعودی است. با این که در ابتدا به نظر هیچ سودی ندارد، فرض می‌کنیم که زمان اجرا روی هر ورودی با ۱۴۰ عنصر و یا کم‌تر برابر $O(۱)$ است؛ هدف انتخاب این ثابت جادویی ۱۴۰ به زودی مشخص خواهد شد. در این صورت می‌توانیم رابطه‌ی بازگشتی زیر را به دست آوریم:

$$T(n)\leq\begin{cases}\theta(۱) & \text{اگر } n\leq ۱۴۰\\ T(\lceil n/۵\rceil)+T(۷n/۱۰+۶)+O(n) & \text{اگر } n>۱۴۰\end{cases}$$

به کمک جانشین‌سازی نشان می‌دهیم که این زمان اجرا خطی است. به طور دقیق‌تر، نشان خواهیم داد که برای یک ثابت به اندازه‌ی کافی بزرگ $c$ و هر $n>۰$ داریم $T(n)\leq cn$. با این فرض شروع می‌کنیم که برای یک ثابت به اندازه‌ی کافی بزرگ $c$ و $n\leq ۱۴۰$ داریم $T(n)\leq cn$؛ در صورت بزرگ بودن $c$ به اندازه‌ی کافی، این فرض درست خواهد بود. همچنین یک ثابت $a$ انتخاب می‌کنیم به طوری که تابع توصیف شده توسط جمله‌ی $O(n)$ در بالا (که مؤلفه‌ی غیر بازگشتی زمان اجرای الگوریتم را نشان می‌دهد) برای هر $n>۰$ از بالا با $an$ محدود شده باشد. با جایگذاری این فرض

استقرا در سمت راست رابطه‌ی بازگشتی به دست می‌آوریم:

$$\begin{aligned} T(n) &\le c\lceil n/5\rceil + c(7n/10+6) + an \\ &\le cn/5 + c + 7n/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

که حداکثر برابر با $cn$ است اگر

$$-cn/10 + 7c + an \le 0 \qquad (۹-۲)$$

نامساوی (۹-۲) معادل است با نامساوی $c \ge 10a(n/(n-70))$ که در آن $n > 70$. از آن جایی که فرض کردیم $n \ge 140$، داریم $n/(n-70) \le 2$، و انتخاب $c \ge 20a$، نامساوی (۹-۲) را ارضا خواهد کرد. (توجه کنید که هیچ چیز خاصی در مورد ثابت ۱۴۰ وجود ندارد؛ می‌توانیم آن را با هر ثابتی بزرگ‌تر از ۷۰ جایگزین، و سپس $c$ را متناسب با آن انتخاب کنیم.) بنابراین بدترین حالت زمان اجرای SELECT خطی است.

مانند مرتب‌سازی‌های مقایسه‌ای (بخش ۸-۱ را ببینید)، SELECT و RANDOMIZED-SELECT اطلاعات خود را در مورد ترتیب نسبی عناصر فقط با استفاده از مقایسه به دست می‌آورند. از فصل ۸ به یاد بیاورید که در مدل مقایسه‌ای، مرتب‌سازی به زمان $\Omega(n \lg n)$ نیاز دارد، حتی در حالت متوسط (مسئله‌ی ۸-۱ را ببینید). الگوریتم‌های مرتب‌سازی در زمان خطی در فصل ۸ فرض‌هایی در مورد عناصر ورودی می‌کنند. در مقابل الگوریتم‌های انتخاب در زمان خطی در این فصل هیچ نیازی به فرض در مورد ورودی ندارند. کران پایین $\Omega(n \lg n)$ در مورد آن‌ها صادق نیست، چرا که آن‌ها مسئله‌ی انتخاب را بدون مرتب‌سازی حل می‌کنند. بنابراین حل مسئله‌ی انتخاب با استفاده از مرتب‌سازی و اندیس‌گذاری، همان طور که در مقدمه‌ی این فصل گفته شد، به صورت حدی ناکارامد است.

## تمرین‌ها

۹-۳-۱ در الگوریتم SELECT عناصر ورودی به گروه‌های ۵تایی تقسیم شدند. اگر آن‌ها به گروه‌های ۷تایی تقسیم شوند، باز هم الگوریتم در زمان خطی اجرا می‌شود؟ بحث کنید که اگر از گروه‌های ۳تایی استفاده شود SELECT در زمان خطی اجرا نمی‌شود.

۹-۳-۲ SELECT را تحلیل کرده و نشان دهید که اگر $n \ge 140$، آن گاه حداقل $\lceil n/4\rceil$ عناصر بزرگ‌تر از میانه‌ی میانه‌ها، و حداقل $\lceil n/4\rceil \lceil \lceil n/4\rceil$ عناصر کوچک‌تر از آن هستند.

۹-۳-۳ نشان دهید می‌توان مرتب‌سازی سریع را طوری تغییر داد که در بدترین حالت در زمان $O(n \lg n)$ اجرا شود، با فرض این که تمام عناصر یکتا هستند.

۹-۳-۴★ فرض کنید یک الگوریتم فقط از مقایسه برای یافتن $i$امین عنصر کوچک در یک مجموعه‌ی $n$ عنصری استفاده می‌کند. نشان دهید که این الگوریتم می‌تواند $i-1$ عنصر کوچک و $n-i$ عنصر بزرگ آرایه را هم بدون انجام مقایسه‌های بیشتر بیابد.

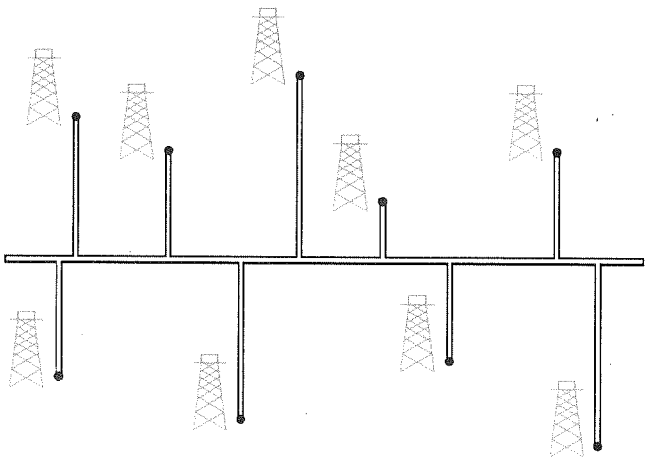
۹-۳-۵ فرض کنید یک زیرروال برای یافتن میانه با بدترین حالت زمان اجرای خطی به صورت «جعبه‌ی سیاه» دارید. یک الگوریتم ساده ارائه کنید که مسئله‌ی انتخاب را برای یک شاخص ترتیبی دلخواه در زمان خطی حل می‌کند.

۹-۳-۶ *چارک* (quantile)های $k$ام یک مجموعه‌ی $n$ عنصری، $k-1$ شاخص ترتیبی هستند که مجموعه‌ی مرتب شده را به $k$ زیرمجموعه با اندازه‌ی ثابت (با اختلاف حداکثر ۱) تقسیم می‌کنند. یک الگوریتم با زمان $O(n \lg k)$ بدهید که چارک‌های $k$ام یک مجموعه را لیست می‌کند.

۹-۳-۷ یک الگوریتم با زمان $O(n)$ ارائه کنید که یک مجموعه‌ی $S$ از $n$ عدد یکتا و یک عدد صحیح مثبت $k \le n$ را به عنوان ورودی دریافت کرده، و $k$ عدد در $S$ پیدا می‌کند که از بقیه‌ی عناصر به میانه‌ی $S$ نزدیک‌ترند.

۹-۳-۸ فرض کنید $X[1..n]$ و $Y[1..n]$ دو آرایه باشند، هر یک شامل $n$ عدد که از قبل مرتب شده‌اند. یک الگوریتم با زمان $O(\lg n)$ ارائه کنید که میانه‌ی تمام عناصر آرایه‌های $X$ و $Y$ را می‌یابد.

۹-۳-۹ پروفسور Olay مشاور یک شرکت نفتی است، که در حال ساخت یک خط لوله از شرق به غرب در یک زمین نفت‌خیز با $n$ چاه نفت است. از هر چاه باید یک لوله‌ی انشعابی با کوتاه‌ترین مسیر (شمالی یا جنوبی) مستقیماً به خط لوله‌ی اصلی متصل باشد، همان طور که در شکل ۹-۲ می‌بینید. با داشتن مختصات $x$ و $y$ چاه‌ها، پروفسور چگونه باید مکان بهینه‌ی خط لوله‌ی اصلی را بیابد (طوری که مجموع طول لوله‌های انشعابی کمینه شود)؟ نشان دهید که این مکان بهینه را می‌توان در زمان خطی تعیین کرد.



**شکل ۹-۲** پروفسور Olay می‌خواهد مکان لوله‌ی نفت شرقی-غربی را طوری تعیین کند که کل طول انشعاب‌های شمالی- جنوبی کمینه شود.

## مسائل

### i عدد بزرگ به صورت مرتب شده

می‌خواهیم با استفاده از یک الگوریتم مقایسه‌ای، $i$ عنصر بزرگ یک مجموعه‌ی $n$تایی از اعداد را به ترتیب بیابیم. این الگوریتم را با استفاده از هر یک از متدهای زیر بیابید به طوری که بهترین زمان حدی را در بدترین حالت داشته باشد، و سپس زمان اجرای الگوریتم‌ها را نسبت به $n$ و $i$ تحلیل کنید.

I. مرتب‌سازی اعداد، و لیست کردن $i$ عدد بزرگ.

II. ساختن یک صف اولویت بیشینه از اعداد، و $i$ بار فراخوانی EXTRACT-MAX.

III. استفاده از یک الگوریتم شاخص ترتیبی و یافتن $i$امین عنصر بزرگ، تقسیم‌بندی نسبت به آن عدد، و مرتب‌سازی $i$ عدد بزرگ.

### ۹-۲ میانه‌ی وزن دار

برای $n$ عنصر مجزای $x_1, x_2, ..., x_n$ با وزن‌های مثبت $w_1, w_2, ..., w_n$، به طوری که $\sum_{i=1}^{n} w_i = 1$، میانه‌ی (بابیش) وزن‌دار، عنصر $x_k$ است که شرایط زیر را ارضا می‌کند:

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

و

$$\sum_{x_i > x_k} w_i \leq \frac{1}{2}$$

برای مثال، اگر عناصر عبارت باشند از ۰/۲، ۰/۰۵، ۰/۱۵، ۰/۱، ۰/۰۵، ۰/۳۵، ۰/۱ و هر عنصر برابر با وزن خود باشد (یعنی $w_i = x_i$ برای $i = 1,2,...,7$)، آن گاه میانه برابر است با ۰/۱، ولی میانه‌ی وزن‌دار عبارت است از ۰/۲.

I. بحث کنید که میانه‌ی $x_1, x_2, ..., x_n$، میانه‌ی وزن‌دار $x_i$ با وزن‌های $w_i = 1/n$ برای $i = 1,2,...,n$ است.

II. نشان دهید که چطور می‌توان با استفاده از مرتب‌سازی، میانه‌ی وزن‌دار $n$ عنصر را در زمان $O(n \lg n)$ پیدا کرد.

III. نشان دهید که چطور می‌توان با استفاده از یک الگوریتم میانه‌ی با زمان خطی مانند SELECT در بخش ۹-۳، میانه‌ی وزن‌دار را در بدترین حالت در زمان $\theta(n)$ پیدا کرد.

*مسئله‌ی* مکان انباری‌ی پست به صورت زیر تعریف می‌شود: $n$ نقطه‌ی $p_1, p_2, ..., p_n$ با وزن‌های $w_1, w_2, ..., w_n$ داریم. می‌خواهیم یک نقطه‌ی $p$ (که لزوماً یکی از نقاط ورودی نیست) بیابیم به طوری که مجموع $\sum_{i=1}^{n} w_i d(p, p_i)$ را کمینه می‌کند، که در آن $d(a,b)$ فاصله‌ی بین نقاط $a$ و $b$ است.

۱۷ بحث کنید که میانه‌ی وزن‌دار بهترین جواب برای مسئله‌ی مکان اداره‌ی پست یک بعدی است، که در آن نقاط فقط اعداد حقیقی هستند، و فاصله‌ی بین نقاط $a$ و $b$ عبارت است از $d(a,b)=|a-b|$.

۷ بهترین جواب برای مسئله‌ی مکان اداره‌ی پست دو بعدی را بیابید، که در آن نقاط جفت مختصات $(x,y)$ هستند، و فاصله‌ی بین نقاط $a=(x_1,y_1)$ و $b=(x_2,y_2)$ عبارت است از *فاصله‌ی منهتن* (Manhattan distance) که به صورت $d(a,b)=|x_1-x_2|+|y_1-y_2|$ تعریف می‌شود.

## ۹-۳ *شاخص‌های ترتیبی کوچک*

نشان داده شد که بیشترین تعداد مقایسه‌های انجام شده توسط SELECT برای انتخاب $i$امین شاخص ترتیبی از $n$ عدد (که آن را با $T(n)$ نشان می‌دهیم) در رابطه‌ی $T(n)=\theta(n)$ صدق می‌کند، ولی در عوض ثابت مخفی درون نماد $\theta$ بزرگ است. وقتی $i$ نسبت به $n$ کوچک باشد، می‌توانیم رویه‌ای متفاوت پیاده‌سازی کنیم که از SELECT به عنوان یک زیرروال استفاده می‌کند ولی در بدترین حالت مقایسه‌های کمتری انجام می‌دهد.

۱ الگوریتمی شرح دهید که از $U_i(n)$ مقایسه برای یافتن $i$امین عنصر کوچک در میان $n$ عنصر استفاده می‌کند، که در آن

$$U_i(n)=\begin{cases}T(n) & \text{اگر } i\geq n/2\\ \lfloor n/2\rfloor+U_i(\lceil n/2\rceil+T(2i)) & \text{در غیر این صورت}\end{cases}$$

(*راهنمایی:* با $\lfloor n/2\rfloor$ مقایسه‌ی جفتی مجزا شروع کنید، و در هر جفت یک فراخوانی بازگشتی روی مجموعه‌ای انجام دهید که شامل عناصر کوچک‌تر است.)

II نشان دهید که اگر $i<n/2$، آن گاه $U_i(n)=n+O(T(2i))\lg(n/i)$.

III نشان دهید که اگر $i$ یک ثابت کوچک‌تر از $n/2$ باشد، آن گاه $U_i(n)=n+O(\lg n)$.

IV نشان دهید که اگر $i=n/k$ برای $k\geq 2$، آن گاه $U_i(n)=n+O(T(2n/k)\lg k)$.

## ۹-۴ *تحلیلی متفاوت از انتخاب تصادفی*

در این مسئله از متغیرهای تصادفی شاخص استفاده می‌کنیم تا رویه‌ی RANDOMIZED-SELECT را به روشی مشابه تحلیل RANDOMIZED-QUICKSORT در بخش ۷-۴-۲ بررسی کنیم.

مانند تحلیل مرتب‌سازی سریع فرض می‌کنیم تمام عناصر یکتا هستند، و عناصر آرایه‌ی ورودی $A$ را به صورت $z_1, z_2, \ldots, z_n$ نام‌گذاری می‌کنیم، که در آن $z_i$ همان $i$امین عنصر کوچک است. بنابراین فراخوانی RANDOMIZED-SELECT$(A,1,n,k)$ عنصر $z_k$ را بازمی‌گرداند.

برای $1 \le i < j \le n$ فرض می‌کنیم

$X_{ijk} = I\{$در حین اجرای الگوریتم برای یافتن $z_k$، $z_i$ با $z_j$ مقایسه شده باشد$\}$

I. یک عبارت دقیق برای $E[X_{ijk}]$ بدهید. (*راهنمایی:* مقدار عبارت شما ممکن است بسته به مقادیر $i$، $j$، و $k$ تغییر کند.)

II. فرض کنید $X_k$ نشان دهنده‌ی تعداد کل مقایسه‌های انجام شده بین عناصر $A$ هنگام یافتن $z_k$ باشد. نشان دهید که

$$E[X_k] \le 2\left(\sum_{i=1}^{k}\sum_{j=k}^{n}\frac{1}{j-i+1} + \sum_{j=k+1}^{n}\frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2}\frac{k-i-1}{k-i+1}\right)$$

III. نشان دهید که $E[x_k] \le 4n$.

IV. نتیجه بگیرید که، با فرض این که تمام عناصر آرایه‌ی $A$ یکتا باشند، امیدریاضی زمان اجرای RANDOMIZED-SELECT برابر است با $O(n)$.

# بخش سوم ساختمان‌های داده

شامل فصل‌های :

مجموعه‌ها، همان قدر که برای ریاضیات اساسی هستند برای علوم کامپیوتر هم اهمیت دارند. در حالی که مجموعه‌های ریاضی غیر قابل تغییرند، مجموعه‌های اداره شده توسط الگوریتم‌ها می‌توانند رشد کنند، کوچک شوند، و یا در طول زمان تغییرات مختلف بکنند. به چنین مجموعه‌هایی پویا (dynamic) می‌گوییم. در پنج فصل بعد چندین تکنیک پایه‌ای برای نمایش مجموعه‌های متناهی پویا و اداره کردن آن‌ها معرفی خواهد شد.

الگوریتم‌ها ممکن است به انواع مختلفی از عملیات بر روی مجموعه‌ها نیاز داشته باشند. به عنوان مثال، بسیاری از الگوریتم‌ها فقط احتیاج دارند که بتوانند عناصر را در مجموعه‌ها درج کنند، آن‌ها را حذف کنند، و عضویت در یک مجموعه را تشخیص دهند. یک مجموعه‌ی پویا که از این عملیات پشتیبانی می‌کند، یک دیکشنری (dictionary) نام دارد. الگوریتم‌های ما به عملیات پیچیده‌تری نیاز دارند. به عنوان مثال صف‌های اولویت مینیمم، که در فصل ۶ در میان متن ساختمان داده‌ی هرم معرفی شدند، از عملیات درج یک عنصر در مجموعه و حذف کوچک‌ترین عنصر مجموعه پشتیبانی می‌کنند. بهترین راه پیاده‌سازی یک مجموعه‌ی پویا به عملیاتی که باید پشتیبانی شود، بستگی دارد.

## عناصر یک مجموعه‌ی پویا

در یک پیاده‌سازی معمولی یک مجموعه‌ی پویا، هر عنصر به صورت یک شیئ نشان داده می‌شود، که اگر یک اشاره‌گر به آن داشته باشیم، می‌توانیم فیلدهای آن را بررسی و اداره کنیم. (در بخش ۱۰-۳ پیاده‌سازی اشیا و اشاره‌گرها را در زبان‌هایی که از آن‌ها به عنوان انواع داده‌ی اصلی پشتیبانی نمی‌کنند،

بررسی خواهد شد.) برخی از انواع مجموعه‌های پویا فرض می‌کنند که یکی از فیلدهای شیئ، یک *کلید* برای تعیین هویت است. اگر تمام کلیدها متفاوت باشند، می‌توانیم به مجموعه‌ی پویا به صورت یک مجموعه از مقادیر کلید نگاه کنیم. این شیئ ممکن است *داده‌ی پیرو* هم داشته باشد، که به همراه فیلدهای شیئ حمل می‌شوند، ولی در پیاده‌سازی مجموعه از آن‌ها استفاده‌ای نمی‌شود. همچنین ممکن است فیلدهایی وجود داشته باشند که توسط عملیات مجموعه اداره می‌شوند: این فیلدها ممکن است داده و یا اشاره‌گرهایی به اشیای دیگر مجموعه داشته باشند.

بعضی از مجموعه‌های پویا فرض می‌کنند که کلیدها از یک مجموعه‌ی کاملا مرتب (totally ordered) هستند، مثلا مجموعه‌ی اعداد حقیقی، و یا مجموعه‌ی تمام کلمه‌های متشکل از الفبای معمولی. یک ترتیب عام به ما اجازه می‌دهد که، مثلا، مینیمم یک مجموعه را تعریف کنیم، و یا از عنصر بزرگ‌تر دقیقا بعد از یک عنصر در یک مجموعه صحبت کنیم.

## عملیات بر روی مجموعه‌های پویا

عملیات بر روی یک مجموعه‌ی پویا می‌تواند به دو نوع تقسیم شود: *جستجوها* (queries)، که فقط اطلاعاتی راجع به مجموعه بازمی‌گردانند، و *عملیات اصلاحاتی*، که مجموعه را تغییر می‌دهند. در این جا یک لیست از عملیات معمول را می‌بینیم. معمولا هر کاربرد خاص فقط نیاز دارد که تعداد کمی از این عملیات پیاده‌سازی شوند.

SEARCH$(S, k)$

- یک جستجو که یک مجموعه‌ی $S$ و یک مقدار کلید $k$ را گرفته، و یک اشاره‌گر $x$ به یک عنصر در $S$ بازمی‌گرداند، به طوری که $x.key = k$، و یا $NIL$ در صورتی که چنین عنصری در $S$ وجود نداشته باشد.

INSERT$(S, k)$

- یک عملیات اصلاحی که یک عنصر که $x$ به آن اشاره می‌کند را به $S$ اضافه می‌کند. معمولا فرض می‌کنیم که هر فیلدی در عنصر $x$ که پیاده‌سازی به آن نیاز دارد قبلا مقداردهی شده است.

DELETE$(S, k)$

- یک عملیات اصلاحی که یک اشاره‌گر $x$ به یکی از عناصر $S$ دریافت کرده و آن را از $S$ حذف می‌کند. (توجه کنید که این عملیات از یک اشاره‌گر به یک عنصر $x$ استفاده می‌کند، و نه یک مقدار کلید.)

MINIMUM$(S)$

- یک جستجو بر روی یک مجموعه‌ی کاملا مرتب $S$، که یک اشاره‌گر به یک عنصر در $S$ با کوچک‌ترین کلید بازمی‌گرداند.

MAXIMUM$(S)$

- یک جستجو بر روی یک مجموعه‌ی کاملا مرتب $S$، که یک اشاره‌گر به یک عنصر در $S$ با بزرگ‌ترین کلید بازمی‌گرداند.

SUCCESSOR $(S, x)$

- یک جستجو که با دریافت یک عنصر $x$ که کلید آن عضو یک مجموعه‌ی کاملا مرتب است، یک اشاره‌گر به عنصر بزرگ‌تر بعدی در $S$ بازمی‌گرداند، و یا $NIL$ در صورتی که $x$ عنصر ماکزیمم باشد.

PREDECESSOR $(S, x)$

- یک جستجو که با دریافت یک عنصر $x$ که کلید آن عضو یک مجموعه‌ی کاملا مرتب است، یک اشاره‌گر به عنصر کوچک‌تر بعدی در $S$ بازمی‌گرداند، و یا $NIL$ در صورتی که $x$ عنصر مینیمم باشد.

جستجوهای SUCCESSOR و PREDECESSOR را می‌توان برای کار با مجموعه‌های با عناصر غیر یکتا هم گسترش داد. برای یک مجموعه از $n$ کلید، فرض معمول این است که فراخوانی MINIMUM و به دنبال آن $n-1$ بار فراخوانی SUCCESSOR، تمام عناصر مجموعه را به ترتیب می‌شمارد.

زمان مورد نیاز برای اجرای یک عملیات مجموعه معمولا نسبت به اندازه‌ی مجموعه‌ی داده شده سنجیده می‌شود. مثلا در فصل ۱۳ یک ساختمان داده بررسی می‌شود که می‌تواند هر یک از عملیات لیست شده در بالا را بر روی یک مجموعه با $n$ عضو، در زمان $O(\lg n)$ انجام دهد.

## خلاصه‌ی بخش سه

در فصل‌های ۱۰–۱۴ ساختمان‌های داده‌ی مختلفی توضیح داده خواهد شد که می‌توان از آن‌ها برای پیاده‌سازی مجموعه‌های پویا استفاده کرد؛ بعدا از بسیاری از آن‌ها برای ساختن الگوریتم‌های کارا برای مسائل مختلف استفاده خواهد شد. ساختمان داده‌ی مهم دیگری–هرم–قبلا در فصل ۶ معرفی شد.

فصل ۱۰ نکات اساسی کار با ساختمان‌های داده‌ی ساده مانند پشته، صف، لیست پیوندی، و درخت‌های ریشه‌دار را توضیح خواهد داد. همچنین نحوه‌ی پیاده‌سازی اشیا و اشاره‌گرها در محیط‌های برنامه‌نویسی که از آن‌ها پشتیبانی نمی‌کنند در این فصل بحث خواهد شد. بسیاری از موضوع‌ها برای کسانی که یک درس برنامه‌نویسی مقدماتی را گذرانده‌اند، آشنا خواهد بود.

در فصل ۱۱، جداول درهم معرفی خواهند شد، که از عملیات دیکشنری INSERT، DELETE، و SEARCH پشتیبانی می‌کنند. در بدترین حالت، یک جدول درهم نیاز به زمان $\theta(n)$ برای انجام یک عملیات SEARCH دارد، ولی امیدریاضی عملیات جداول درهم، $O(1)$ است. تحلیل جداول درهم بر پایه‌ی احتمالات است، ولی بیشتر مطالب فصل به هیچ پیش‌زمینه‌ای در این مورد نیاز ندارد.

درخت‌های جستجوی دودویی، که در فصل ۱۲ به آن‌ها خواهیم پرداخت، از تمام عملیات مجموعه‌های پویا که در بالا معرفی شد پشتیبانی می‌کنند. بر روی یک درخت جستجوی دودویی با $n$ عنصر، در بدترین حالت هر عملیات $\theta(n)$ زمان خواهد برد، ولی بر روی یک درخت جستجوی دودویی به صورت تصادفی ساخته شده، امیدریاضی زمان اجرا برای هر عملیات $O(\lg n)$ است. درختان جستجوی دودویی زیربنای بسیاری از ساختمان‌های داده‌ی دیگر هستند.

درختان قرمز- سیاه، نسخه‌ای از درختان جستجوی دودویی، در فصل ۱۳ معرفی می‌شوند. برخلاف درخت‌های جسجتوی دودویی معمولی، کارایی درخت‌های قرمز- سیاه تضمین شده است: عملیات در بدترین حالت در زمان $O(\lg n)$ انجام می‌شوند. یک درخت قرمز-سیاه، یک درخت جستجوی دودویی متوازن (balanced) است: در فصل ۱۸ در بخش پنجم، نوع دیگری از درخت‌های متوازن، به نام B-درخت‌ها معرفی خواهد شد. با این که مکانیک ساختاری یک درخت قرمز- سیاه تا حدودی پیچیده است، شما می‌توانید اکثر اطلاعات فصل را بدون آموختن جزئیات این ساختار درک کنید. با این همه مرور کردن کدها می‌تواند کاملا آموزنده باشد.

در فصل ۱۴، نشان خواهیم داد که چگونه می‌توان درخت‌های قرمز-سیاه را طوری گسترش داد که از عملیاتی غیر از عملیات اصلی که در بالا معرفی شد، پشتیبانی کنند. ابتدا، آن‌ها را گسترش می‌دهیم تا بتوانیم به صورت پویا، شاخص‌های ترتیبی یک مجموعه از کلیدها را داشته باشیم. سپس، آن‌ها را به شکل دیگری گسترش می‌دهیم تا بتوانیم بازه‌هایی از اعداد صحیح را در آن‌ها نگه داری کنیم.

فصل

# ۱۰

# ساختمان‌های داده‌ی مقدماتی

در این فصل روش‌های نمایش مجموعه‌های پویا به وسیله‌ی ساختمان‌های داده‌ی ساده (که از اشاره‌گرها استفاده می‌کنند) را بررسی خواهیم کرد. با این که با استفاده از اشاره‌گرها می‌توان ساختمان‌های داده‌ی پیچیده‌ای ساخت، در این جا فقط ساختمان‌های داده‌ی اساسی را معرفی خواهیم کرد: پشته، صف، لیست پیوندی، و درخت‌های ریشه‌دار. همچنین روشی خواهیم دید برای ساخت اشیا و اشاره‌گرها را از آرایه‌ها.

## ۱۰-۱ پشته و صف

پشته‌ها و صف‌ها مجموعه‌های پویایی هستند که در آن‌ها هنگام عملیات DELETE، خود مجموعه تعیین می‌کند کدام عنصر حذف می‌شود، نه کاربر. در یک پشته (stack)، عنصری که از مجموعه حذف می‌شود آخرین عنصری است که به مجموعه اضافه شده است: پشته سیاست *آخرین ورودی، اولین خروجی* (last-in first-out, LIFO) را پیاده‌سازی می‌کند. به طور مشابه در یک صف (queue)، عنصری که حذف می‌شود همیشه عنصری است که بیشتر از بقیه در مجموعه بوده است: صف سیاست *اولین ورودی، اولین خروجی* (first-in first-out, FIFO) را پیاده‌سازی می‌کند. روش‌های کارامد مختلفی برای پیاده‌سازی صف‌ها و پشته‌ها در کامپیوتر وجود دارد. در این بخش خواهیم دید که چگونه از یک آرایه‌ی ساده برای پیاده‌سازی هر دوی آن‌ها استفاده کنیم.

### پشته‌ها

عملیات INSERT بر روی یک پشته معمولاً PUSH نامیده می‌شود، و عملیات DELETE، که هیچ عنصری به عنوان آرگومان ورودی نمی‌گیرد، POP نامیده می‌شود. این اسامی به پشته‌های واقعی اشاره

دارند، مانند پشته‌های فنری بشقاب که در بعضی رستوران‌ها از آن‌ها استفاده می‌شود. ترتیبی که بشقاب‌ها از پشته خارج می‌شوند برعکس ترتیبی است که در پشته قرار گرفته‌اند، چرا که فقط بشقاب بالایی در دسترس است.

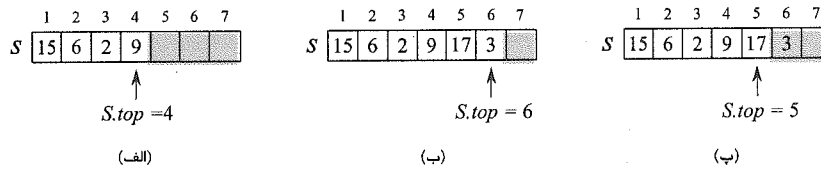
همان طور که در شکل ۱۰-۱ نشان داده شده است، می‌توانیم یک پشته با حداکثر $n$ عنصر را با یک آرایه‌ی $S[1 .. n]$ پیاده‌سازی کنیم. آرایه یک خصیصه‌ی $S.top$ دارد که اندیس آخرین عنصر درج شده در آرایه را نگه می‌دارد. پشته حاوی عناصر $S[1 .. S.top]$ است، که در آن $S[1]$ عنصر انتهایی پشته و $S[S.top]$ عنصر بالای پشته است.

وقتی $S.top = 0$ هیچ عنصری در پشته وجود ندارد، و می‌گوییم پشته *تهی* (empty) است. تهی بودن پشته را می‌توان به کمک عملیات جستجوی STACK-EMPTY چک کرد. اگر از یک پشته‌ی تهی عنصری بازیابی شود، می‌گوییم پشته *پاریز* (underflow) می‌شود، که معمولاً یک خطا است. اگر $S.top$ از $n$ فراتر رود، پشته *سرریز* (overflow) شده است. (ما در پیاده‌سازی شبه‌کد در زیر نگران سرریز پشته نخواهیم بود.)

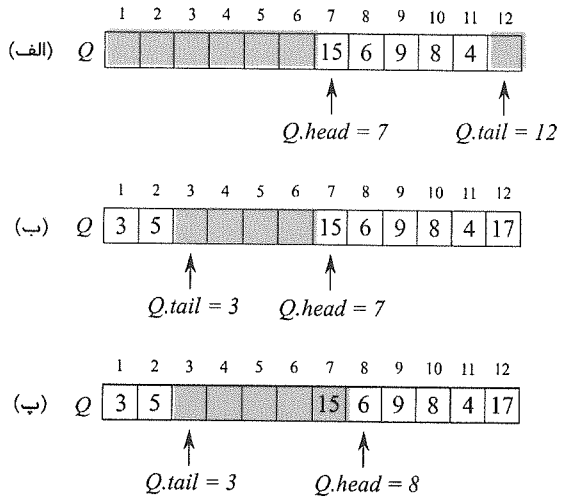
```
STACK-EMPTY(S)
1  if S.top == 0
2      return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2      error "underflow"
3  else S.top = S.top - 1
4      return S[S.top + 1]
```




**شکل ۱۰-۱** یک پیاده‌سازی پشته‌ی $S$ به وسیله‌ی آرایه. عناصر پشته آن‌هایی هستند که با سایه‌ی کم‌رنگ مشخص شده‌اند. (الف) پشته‌ی $S$ حاوی ۴ عنصر است، با عنصر بالایی با مقدار ۹. (ب) پشته‌ی $S$ پس از فراخوانی PUSH$(S, ۱۷)$ و PUSH$(S, ۳)$. (پ) پشته‌ی $S$ بعد از این که فراخوانی POP$(S)$ عنصر ۳ را بازمی‌گرداند. این همان عنصری است که آخر از همه درج شده است. با این که عنصر ۳ هنوز در آرایه وجود دارد، ولی در پشته نیست؛ عنصر بالایی ۱۷ است.




**شکل ۱۰-۲** پیاده‌سازی صف با استفاده از آرایه‌ی $Q[1..12]$. مکان‌های کم‌رنگ عناصر صف را نشان می‌دهند. (الف) صف ۵ عنصر دارد که در مکان‌های $Q[7..11]$ قرار دارند. (ب) آرایش صف بعد از فراخوانی ENQUEUE$(Q,17)$، ENQUEUE$(Q,3)$ و ENQUEUE$(Q,5)$. (پ) آرایش صف بعد از این که فراخوانی DEQUEUE$(Q)$، مقدار کلید ۱۵ را که قبلا در ابتدای صف بوده بازمی‌گرداند. مقدار کلید سر جدید صف برابر است با ۶.

شکل ۱۰-۱ تأثیر اعمال اصلاحی PUSH و POP را نشان می‌دهد. هر سه عملیات پشته در زمان $O(1)$ اجرا می‌شوند.

## صف‌ها

به عملیات INSERT در یک صف ENQUEUE، و به عملیات DELETE در آن DEQUEUE گفته می‌شود؛ مانند عملیات POP در پشته، DEQUEUE هیچ عنصری به عنوان آرگومان ورودی نمی‌گیرد. خصوصیت FIFO در یک صف باعث می‌شود که مانند یک صف از انسان‌ها در یک دفتر ثبت نام عمل کند. صف یک *ابتدا* (head) و یک *انتها* (tail) دارد. وقتی یک عنصر در صف درج می‌شود در انتهای صف قرار می‌گیرد، درست مانند یک دانش‌آموز تازه رسیده که به انتهای صف می‌رود. عنصر حذف شده همیشه از سر صف حذف می‌شود، مانند دانش‌آموز سر صف که بیشتر از همه در صف انتظار کشیده است. (خوشبختانه نیازی نیست که نگران جلو زدن عناصر محاسباتی در صف باشیم!)

شکل ۱۰-۲ روشی ارائه می‌کند برای پیاده‌سازی یک صف با حداکثر $n-1$ عنصر با استفاده از یک آرایه‌ی $Q[1..n]$. صف یک خصوصیت $Q.head$ دارد که اندیس سر صف را نگه می‌دارد، یا به سر صف اشاره می‌کند. خصوصیت $Q.tail$ اندیس مکانی را نشان می‌دهد که عنصر جدیدی که به صف اضافه می‌شود، در آن مکان قرار می‌گیرد. عناصر درون صف در مکان‌های $Q.head$، $Q.head+1$، ...، $Q.tail-1$ قرار دارند. در این جا ابتدا و انتهای آرایه را به هم متصل می‌کنیم تا مکان ۱ دقیقاً بعد از مکان $n$ به صورت دایره‌ای بیاید. وقتی $Q.head = Q.tail$، صف تهی است. در ابتدا داریم

$Q.head = Q.tail = 1$. وقتی صف تهی است تلاش برای حذف یک عنصر باعث پاریز صف می‌شود. وقتی $Q.head = Q.tail + 1$، صف پر است و تلاش برای درج یک عنصر در صف باعث سرریز صف می‌شود.

در رویه‌های ENQUEUE و DEQUEUE در زیر از چک کردن برای سرریز و پاریز صرف‌نظر شده است. (تمرین ۱۰-۱-۴ از شما می‌خواهد کدی ارائه کنید که وجود این دو خطا را هم چک می‌کند.) در شبه‌کدهای زیر فرض شده است که $n = Q.length$.

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3       Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

شکل ۱۰-۲ تأثیر عملیات ENQUEUE و DEQUEUE را نشان می‌دهد. هر عملیات در زمان $O(1)$ اجرا می‌شود.

## تمرین‌ها

۱۰-۱-۱ با استفاده از شکل ۱۰-۱ به عنوان یک مدل، نتیجه‌ی هر یک از عملیات دنباله‌ی $\text{PUSH}(S,4)$، $\text{PUSH}(S,1)$، $\text{PUSH}(S,3)$، $\text{POP}(S)$، $\text{PUSH}(S,8)$، و $\text{POP}(S)$ را روی یک پشته‌ی تهی $S$ که در آرایه‌ی $S[1..6]$ ذخیره شده است نشان دهید.

۱۰-۱-۲ توضیح دهید که چطور می‌توان دو پشته را در یک آرایه‌ی $A[1..n]$ به صورت هم‌زمان پیاده‌سازی کرد به طوری که هیچ یک از آن‌ها سرریز نشود، مگر این که کل تعداد عناصر در دو پشته از $n$ فراتر رود. عملیات PUSH و POP باید در زمان $O(1)$ انجام شوند.

۱۰-۱-۳ با استفاده از شکل ۱۰-۲ به عنوان یک مدل، نتیجه‌ی هر یک از عملیات در دنباله‌ی $\text{ENQUEUE}(Q,4)$، $\text{ENQUEUE}(Q,1)$، $\text{ENQUEUE}(Q,3)$، $\text{DEQUEUE}(Q)$، $\text{ENQUEUE}(Q,8)$، $\text{DEQUEUE}(Q)$ را روی یک صف تهی $Q$ که در آرایه‌ی $Q[1..6]$ ذخیره شده است نشان دهید.

۱۰-۱-۴ ENQUEUE و DEQUEUE را طوری بازنویسی کنید که خطاهای سرریز و پاریز در صف را تشخیص دهند.

۱۰-۱-۵ در حالی که یک پشته فقط اجازه‌ی درج و حذف عناصر را در یک سر می‌دهد، و صف

اجازه‌ی درج در یک سر و حذف از سر دیگر را می‌دهد، یک *صف دوطرفه* (deque) اجازه‌ی درج و حذف در دو طرف را می‌دهد. چهار رویه‌ای با زمان $O(1)$ برای درج و حذف عناصر در هر دو سر یک صف دوطرفه که از یک آرایه ساخته شده است، بنویسید.

۱۰-۱-۶ نشان دهید چطور می‌توان با استفاده از دو پشته، یک صف ساخت. زمان اجرای اعمال این صف را تحلیل کنید.

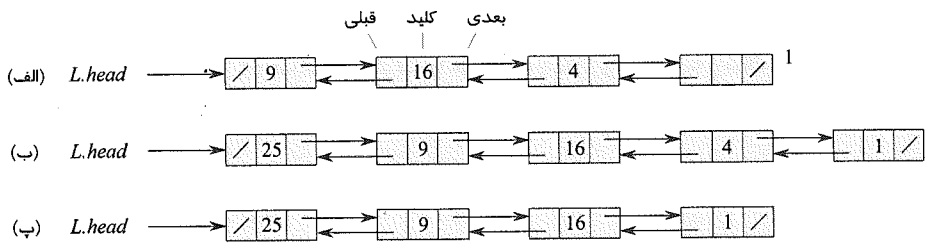
۱۰-۱-۷ نشان دهید چطور می‌توان با استفاده از دو صف، یک پشته ساخت. زمان اجرای اعمال این پشته را تحلیل کنید.

## ۱۰-۲ لیست‌های پیوندی

*لیست پیوندی* (linked list) ساختمان داده‌ای است که در آن اشیا به صورت خطی مرتب شده‌اند. با این حال برخلاف آرایه‌ها که این ترتیب خطی توسط اندیس‌ها تعیین می‌شود، ترتیب در لیست‌های پیوندی توسط اشاره‌گرهای درون هر شیئ تعیین می‌شود. لیست‌های پیوندی نمایشی ساده و انعطاف‌پذیر برای مجموعه‌های پویا هستند، که از تمام اعمال لیست شده در بخش معرفی قسمت III (نه لزوماً به صورت بهینه) پشتیبانی می‌کنند.

همان طور که در شکل ۱۰-۳ نشان داده شده است، هر عنصر از یک *لیست پیوندی دو طرفه* (doubly linked list) $L$، شیئی است حاوی یک فیلد $key$ و دو فیلد اشاره‌گر دیگر: $next$ و $prev$. این شیئ ممکن است داده‌های پیرو دیگری هم داشته باشد. با داشتن یک عنصر $x$ در لیست، $x.next$ به عنصر مابعد (successor) در لیست پیوندی و $x.prev$ به عنصر ماقبل (predecessor) در لیست پیوندی اشاره می‌کند. اگر $x.prev = \text{NIL}$، عنصر $x$ هیچ عنصری ماقبل خود ندارد و بنابراین عنصر اول لیست، و یا *سر* (head) است. اگر $x.next = \text{NIL}$، آن‌گاه $x$ هیچ عنصری مابعد خود ندارد و بنابراین عنصر آخر لیست، و یا *انتها* (tail) است. خصیصه‌ی $L.head$ به اولین عنصر لیست اشاره می‌کند. اگر $L.head = \text{NIL}$، آن گاه لیست تهی است.

یک لیست ممکن است یکی از اشکال مختلف ممکن را داشته باشد. ممکن است پیوندها (اشاره‌گرها)ی آن به صورت یک طرفه و یا دو طرفه باشند، ممکن است مرتب‌شده باشد یا نباشد، ممکن است دایره‌ای باشد یا نباشد. اگر یک لیست، *پیوندی یک طرفه* (singly linked) باشد، اشاره‌گر $prev$ در عناصر وجود نخواهد داشت. اگر *مرتب‌شده* (sorted) باشد، ترتیب خطی لیست مطابق است با ترتیب خطی کلیدها که در عناصر لیست مرتب شده‌اند. اگر *نامرتب* (unsorted) باشد، عناصر می‌توانند به هر ترتیبی ظاهر شوند. در یک *لیست دایره‌ای* (circular list) اشاره‌گر $prev$ در عنصر سر لیست به انتهای لیست اشاره می‌کند، و اشاره‌گر $next$ در انتهای لیست، به ابتدای لیست اشاره می‌کند. بنابراین می‌توان به لیست به چشم یک حلقه از عناصر نگاه کرد. در ادامه‌ی این بخش فرض می‌کنیم لیست‌هایی که با آن‌ها کار می‌کنیم، نامرتب و پیوندی دوطرفه هستند.




**شکل ۱۰-۳** (الف) یک لیست پیوندی دو طرفه‌ی $L$ که نشان‌دهنده‌ی مجموعه‌ی پویای $\{۱,۴,۹,۱۶\}$ است. هر عنصر در لیست، یک شیئ است با فیلدهایی برای کلید و اشاره‌گرهایی (که با فلش نمایش داده شده‌اند) به عناصر قبلی و بعدی. فیلد *next* از عنصر انتها و فیلد *prev* از عنصر ابتدا، NIL هستند، که با یک ممیز قطری نمایش داده شده‌اند. خصیصه‌ی $L.head$ به سر لیست اشاره می‌کند. (ب) بعد از اجرای LIST-INSERT$(L,x)$، که در آن $x.key = ۲۵$، لیست پیوندی یک شیئ جدید با کلید ۲۵ به عنوان سر لیست دارد. این عنصر جدید به سر قدیمی لیست با کلید ۹ اشاره می‌کند. (پ) نتیجه‌ی فراخوانی LIST-DELETE$(L,x)$، که در آن $x$ به عنصر با کلید ۴ اشاره می‌کند.

## جستجو در یک لیست پیوندی

رویه‌ی LIST-SEARCH$(L,k)$ به کمک جستجوی ساده‌ی خطی اولین عنصر با کلید $k$ را در لیست $L$ یافته و یک اشاره‌گر به آن عنصر بازمی‌گرداند. اگر هیچ عنصری با کلید $k$ در لیست موجود نباشد، NIL بازگردانده می‌شود. برای لیست پیوندی شکل ۱۰-۳(الف)، فراخوانی LIST-SEARCH$(L,۴)$ یک اشاره‌گر به عنصر سوم بازمی‌گرداند، و فراخوانی LIST-SEARCH$(L,۷)$ مقدار NIL را باز خواهد گرداند.

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

برای جستجو در یک لیست با $n$ عنصر، رویه‌ی LIST-SEARCH در بدترین حالت به $\theta(n)$ زمان نیاز دارد، چرا که ممکن است مجبور باشد تمام لیست را جستجو کند.

## درج در یک لیست پیوندی

با داشتن یک عنصر $x$ که فیلد $k$ در آن از قبل مقداردهی شده، رویه‌ی LIST-INSERT عنصر $x$ را به ابتدای لیست «پیوند می‌زند»، همان طور که در شکل ۱۰-۳(ب) نشان داده شده است.

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

زمان اجرای LIST-INSERT در یک لیست با $n$ عنصر عبارت است از $O(1)$.

### حذف کردن از یک لیست پیوندی

رویه‌ی LIST-DELETE، یک عنصر $x$ را از یک لیست پیوندی $L$ حذف می‌کند. برای این کار باید اشاره‌گری به عنصر $x$ به رویه بدهیم. در این صورت رویه با اصلاح اشاره‌گرها پیوند $x$ را از لیست جدا می‌کند. اگر بخواهیم یک عنصر با کلیدی خاص را از لیست حذف کنیم، باید ابتدا LIST-SEARCH را فراخوانی کنیم تا یک اشاره‌گر به عنصر مورد نظر به ما بازگرداند.

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

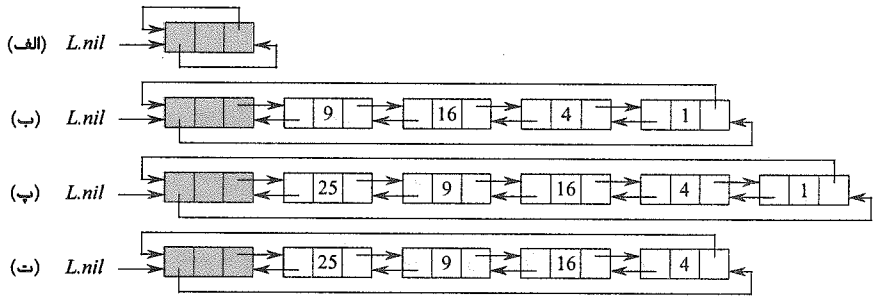
شکل ۱۰-۳ (پ) نشان می‌دهد که چگونه یک عنصر از یک لیست پیوندی حذف می‌شود. LIST-DELETE در زمان $O(1)$ اجرا می‌شود، ولی اگر بخواهیم یک عنصر با یک کلید دلخواه را حذف کنیم، در بدترین حالت به زمان $\theta(n)$ نیاز داریم، چرا که باید ابتدا برای یافتن عنصر LIST-SEARCH را فراخوانی کنیم.

### نگهبان‌ها

اگر بتوانیم حالت‌های مرزی ابتدا و انتهای لیست را از کد LIST-DELETE حذف کنیم، کد آن بسیار ساده‌تر خواهد شد.

```
LIST-DELET' (L, x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

*نگهبان* (sentinel)، یک عنصر اضافه است که به ما اجازه می‌دهد حالت‌های مرزی را ساده کنیم. برای مثال فرض کنید به همراه یک لیست $L$، یک شیئ $L.nil$ هم فراهم می‌کنیم که نشان‌دهنده‌ی NIL است، ولی تمام فیلدهای دیگر عناصر لیست را دارد. هر جا که یک اشاره‌گر به NIL در کد داریم، آن را با اشاره‌گری به عنصر نگهبان $L.nil$ جایگزین می‌کنیم. همان طور که در شکل ۱۰-۴ نشان داده شده است، این تغییر یک لیست پیوندی دوطرفه‌ی معمولی را به یک *لیست پیوندی دوطرفه‌ی دایره‌ای با یک نگهبان* تبدیل می‌کند، که در آن نگهبان $L.nil$ بین ابتدا و انتهای لیست قرار داده می‌شود؛ فیلد $L.nil.next$ به سر لیست، و فیلد $L.nil.prev$ به انتهای لیست اشاره می‌کند. به طور مشابه، فیلد $next$ عنصر انتهایی و فیلد $prev$ عنصر سر لیست به $L.nil$ اشاره می‌کنند. از آن‌جایی که $L.nil.next$ به ابتدای لیست اشاره می‌کند می‌توانیم به کلی خصوصیت $L.head$ را حذف، و تمام مراجعی که به آن اشاره می‌کنند را با مراجعی به $L.nil.next$ جایگزین کنیم. شکل ۱۰-۴ نشان می‌دهد که یک لیست تهی فقط شامل نگهبان است، و هر دوی $L.nil.next$ و $L.nil.prev$ به $L.nil$ اشاره می‌کنند.




**شکل ۱۰-۴** یک لیست پیوندی دوطرفه‌ی دایره‌ای با نگهبان. نگهبان $L.nil$ بین ابتدا و انتهای صف قرار دارد. دیگر به خصوصیت $L.head$ نیازی نیست، چرا که می‌توانیم به کمک $L.nil.next$ به ابتدای صف دسترسی داشته باشیم. (الف) یک لیست تهی. (ب) لیست پیوندی شکل ۱۰-۳(الف)، با کلید ۹ در ابتدا و کلید ۱ در انتها. (پ) لیست پس از اجرای LIST-INSERT′$(L, x)$، که در آن $x.key = ۲۵$. عنصر جدید تبدیل به ابتدای لیست می‌شود. (ت) لیست پس از حذف شیئ با کلید ۱. انتهای جدید، عنصر با کلید ۴ است.

کد LIST-SEARCH به صورت سابق باقی می‌ماند، ولی مراجع NIL و $L.head$ به صورت گفته شده در بالا تغییر می‌کنند.

```
LIST-SEARC′(L, k)
1  x = L.nil.next
2  while x ≠ L.nil and x.key ≠ k
3      x = x.next
4  return x
```

از رویه‌ی دو خطی LIST-DELETE′ برای حذف یک عنصر از لیست استفاده می‌کنیم. همچنین از رویه‌ی زیر برای درج یک عنصر در لیست استفاده خواهیم کرد.

```
LIST-INSER′ (L, x)
1  x.next = L.nil.next
2  L.nil.next.prev = x
3  L.nil.next = x
4  x.prev = L.nil
```

شکل ۱۰-۴، تأثیر LIST-INSERT′ و LIST-DELETE′ را روی یک لیست ساده نشان می‌دهند. نگهبان‌ها به ندرت کران‌های حدی زمان اجرای اعمال روی ساختمان‌های داده را کاهش می‌دهند، ولی می‌توانند ضرایب ثابت را به خوبی کاهش دهند. فایده‌ی استفاده از نگهبان در حلقه‌ها معمولاً تمیزی کد است، نه سرعت؛ به عنوان مثال کد لیست پیوندی با استفاده از نگهبان ساده‌تر شده است، ولی در رویه‌های LIST-INSERT′ و LIST-DELETE′ فقط به اندازه‌ی $O(۱)$ در زمان صرفه جویی کرده‌ایم. با این حال در موقعیت‌های دیگر، استفاده از نگهبان‌ها به ما کمک می‌کند که کد حلقه‌ها را کوچک‌تر کنیم، که باعث می‌شود ضرایب ثابت، مثلاً برای جمله‌های $n$ یا $n^۲$ در زمان اجرا کاهش یابند.

نباید از نگهبان‌ها به صورت مبهم استفاده کرد. اگر تعداد زیادی لیست کوچک داشته باشیم، حافظه‌ی اضافی مصرف شده برای نگهبان‌های آن‌ها می‌تواند فضای قابل توجهی اشغال کند. در این کتاب، فقط زمانی از نگهبان‌ها استفاده خواهیم کرد که واقعاً کد را ساده‌تر کنند.

## تمرین‌ها

۱۰-۲-۱ آیا می‌توان عملیات INSERT را روی یک لیست پیوندی یک طرفه با زمان $O(1)$ پیاده‌سازی کرد؟ عملیات DELETE را چطور؟

۱۰-۲-۲ با استفاده از یک لیست پیوندی یک طرفه‌ی $L$، یک پشته پیاده‌سازی کنید. عملیات PUSH و POP باید همچنان در زمان $O(1)$ اجرا شوند.

۱۰-۲-۳ با استفاده از یک لیست پیوندی یک طرفه‌ی $L$، یک صف پیاده‌سازی کنید. عملیات ENQUEUE و DEQUEUE باید همچنان در زمان $O(1)$ اجرا شوند.

۱۰-۲-۴ همان‌طور که گفته شد، هر تکرار حلقه در رویه‌ی LIST-SEARCH' به دو تست نیاز دارد: یکی برای $x \neq L.nil$ و دیگری برای $x.key \neq k$. نشان دهید که چطور می‌توان $x \neq L.nil$ را از تکرارها حذف کرد.

۱۰-۲-۵ عملیات دیکشنری INSERT، SEARCH، و DELETE را با استفاده از لیست‌های پیوندی یک طرفه‌ی دایره‌ای پیاده‌سازی کنید. زمان اجرای رویه‌های شما چقدر است؟

۱۰-۲-۶ عملیات UNION روی مجموعه‌های پویا، دو مجموعه‌ی گسسته‌ی $S_1$ و $S_2$ را به عنوان ورودی می‌گیرد و یک مجموعه‌ی $S = S_1 \cup S_2$، شامل تمام عناصر $S_1$ و $S_2$ را به عنوان خروجی بازمی‌گرداند. معمولاً مجموعه‌های $S_1$ و $S_2$ توسط این عملیات نابود می‌شوند. نشان دهید چگونه می‌توان با استفاده از لیست مناسب به عنوان ساختمان داده، عملیات UNION را در زمان $O(1)$ پیاده‌سازی کرد.

۱۰-۲-۷ رویه‌ای غیر بازگشتی با زمان $\theta(n)$ ارائه کنید که یک لیست پیوندی یک طرفه با $n$ عنصر را معکوس می‌کند. این رویه نباید بیشتر از مقدار ثابتی حافظه، غیر از مقداری که برای لیست نیاز است، مصرف کند.

★ ۱۰-۲-۸ توضیح دهید که چگونه می‌توان یک لیست پیوندی دو طرفه را با استفاده از فقط یک فیلد اشاره‌گر $x.np$، به جای دو فیلد $next$ و $prev$ در هر عنصر، پیاده‌سازی کرد. تمام مقادیر اشاره‌گرها را اعداد صحیح $k$ بیتی در نظر بگیرید، و $x.np$ را به صورت $x.np = x.next \ XOR \ x.prev$ تعریف کنید، که این عملگر، «یای انحصاری» $k$ بیتی $x.next$ و $x.prev$ را بازمی‌گرداند. (مقدار NIL با ۰ نشان داده می‌شود.) تمام اطلاعاتی که برای دسترسی به ابتدای صف لازم است را توصیف کنید. نشان دهید چطور می‌توان

عملیات INSERT، SEARCH، و DELETE را روی چنین لیستی پیاده‌سازی کرد. همچنین نشان دهید چطور می‌توان چنین لیستی را در زمان $O(1)$ معکوس کرد.

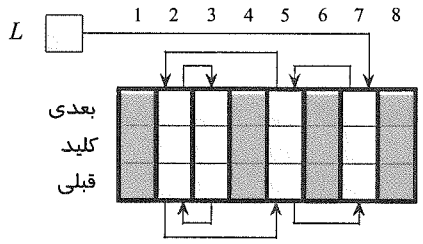
## ۱۰-۳ پیاده‌سازی اشاره‌گرها و اشیا

چطور می‌توان اشاره‌گرها و اشیا را در زبان‌هایی که این امکانات را فراهم نمی‌کنند، پیاده‌سازی کرد؟ در این بخش روش‌هایی برای پیاده‌سازی ساختمان‌های داده‌ی پیوندی بدون داشتن یک نوع داده‌ی اشاره‌گر خواهیم دید. اشاره‌گرها و اشیا را از آرایه‌ها و اندیس آن‌ها خواهیم ساخت.

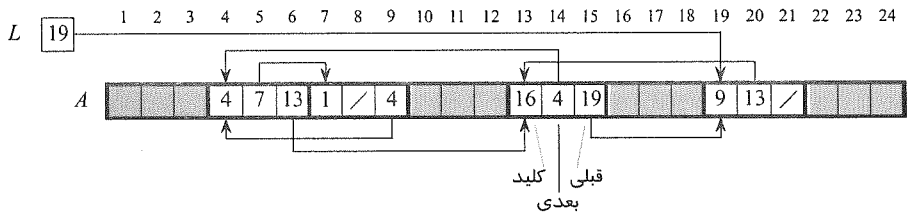
### نمایشی چند آرایه‌ای از اشیا

می‌توان مجموعه‌ای از اشیا که فیلدهای یکسانی دارند را به صورت یک آرایه برای هر یک از فیلدها نمایش داد. به عنوان مثال، شکل ۱۰-۵ نشان می‌دهد که چطور می‌توان لیست پیوندی شکل ۱۰-۳(الف) را با سه آرایه پیاده‌سازی کرد. آرایه‌ی *key*، مقادیر کلیدهای درون مجموعه‌ی پویا را نشان می‌دهد، و اشاره‌گرها در آرایه‌های *next* و *prev* ذخیره شده‌اند. برای یک اندیس $x$، ورودی‌های $key[x]$، $next[x]$، و $prev[x]$ نشان دهنده‌ی یک شیئ در لیست پیوندی هستند. با این تفاسیر، اشاره‌گر $x$ فقط یک اندیس معمولی در آرایه‌های *key*، *next*، و *prev* است.

در شکل ۱۰-۳ (الف)، شیئ با کلید ۴ در لیست بعد از شیئ با کلید ۱۶ می‌آید. در شکل ۱۰-۵، کلید ۴ در $key[۲]$ ذخیره شده است، و کلید ۱۶ در $key[۵]$، بنابراین داریم $next[۵] = ۲$ و $prev[۲] = ۵$. با این که ثابت NIL در فیلد *next* در عنصر انتها و فیلد *prev* در عنصر ابتدا دیده می‌شود، معمولاً از یک عدد صحیح (مثلاً ۰ یا ۱-) که نمی‌تواند یک اندیس واقعی در آرایه را نشان دهند، استفاده می‌کنیم. متغیر $L$ اندیس ابتدای لیست را نگه می‌دارد.




**شکل ۱۰-۵** لیست پیوندی شکل ۱۰-۳ (الف) که توسط آرایه‌های *key*، *next*، و *prev* نمایش داده شده است. هر تکه‌ی عمودی از آرایه‌ها نشان دهنده‌ی یک شیئ است. اندیس‌های آرایه‌ها که در اشاره‌گرها ذخیره می‌شوند، در بالا نمایش داده شده است؛ فلش‌ها نشان می‌دهند که چگونه باید آن‌ها را تفسیر کرد. مکان‌های با سایه‌ی کم‌رنگ اشیای درون لیست هستند. متغیر $L$ اندیس ابتدای لیست را نگه می‌دارد.




**شکل ۱۰-۶** لیست پیوندی شکل‌های ۱۰-۳ (الف) و ۱۰-۵ که در یک آرایه‌ی تنهای $A$ ذخیره است. هر عنصر لیست، یک شیئ است که یک زیرآرایه‌ی پیوسته با طول ۳ را در آرایه اشغال می‌کند. سه فیلد *key*، *next*، و *prev* به ترتیب دارای آفست‌های ۰، ۱، و ۲ در هر شیئ هستند. یک اشاره‌گر به شیئ، اندیسی است که به اولین عنصر مربوط به شیئ اشاره می‌کند. اشیا حاوی عناصر لیست با سایه‌ی کم‌رنگ مشخص شده‌اند، و فلش‌ها نشان دهنده‌ی ترتیب در لیست هستند.

## نمایشی تک آرایه‌ای از اشیا

معمولاً کلمه‌ها در حافظه‌ی کامپیوتر با اعداد صحیح ۰ تا $M-1$ آدرس‌دهی می‌شوند، که در آن $M$ یک عدد به اندازه‌ی کافی بزرگ است. در بسیاری از زبان‌های برنامه‌نویسی، یک شیئ مجموعه‌ای از مکان‌ها را در حافظه‌ی کامپیوتر اشغال می‌کند. یک اشاره‌گر، فقط آدرس اولین مکان حافظه‌ی مربوط به شیئ است، و بقیه‌ی مکان‌های حافظه مربوط به شیئ را می‌توان با اضافه کردن آفست (offset) به اشاره‌گر به دست آورد.

می‌توانیم از استراتژی یکسانی برای پیاده‌سازی اشیا در محیط‌های برنامه‌نویسی که از نوع داده‌ی اشاره‌گر پشتیبانی نمی‌کنند، استفاده کنیم. مثلاً شکل ۱۰-۶ نشان می‌دهد که چطور می‌توان از یک آرایه‌ی $A$ برای ذخیره‌ی لیست پیوندی شکل‌های ۱۰-۳(الف) و ۱۰-۵ استفاده کرد. یک شیئ، زیرآرایه‌ی پیوسته‌ی $A[j \mathinner{..} k]$ را اشغال می‌کند. هر فیلد از شیئ متناظر است با یک آفست در محدوده‌ی ۰ تا $k-1$، و اندیس $j$ یک اشاره‌گر به شیئ است. در شکل ۱۰-۶، آفست مربوط به *key*، *next*، و *prev* به ترتیب ۰، ۱، و ۲ است. برای خواندن مقدار $prev[i]$ با داشتن اشاره‌گر $i$، می‌توانیم مقدار آفست ۲ را با اشاره‌گر جمع کنیم، و حافظه‌ی $A[i+2]$ را بخوانیم.

نمایش تک آرایه‌ای از این جهت انعطاف پذیر است که اجازه‌ی ذخیره‌ی اشیا با طول‌های مختلف را می‌دهد. مشکلات اداره کردن چنین مجموعه‌ی ناهمگنی از اشیا بیشتر از اداره کردن یک مجموعه‌ی ناهمگن در حالتی است که تمام اشیا، فیلدهای یکسانی دارند. از آن‌جایی که اکثر ساختمان‌های داده‌ای که با آن‌ها برخورد خواهیم کرد از عناصر همگنٌ تشکیل شده‌اند، برای هدف ما بهتر است که از نمایش چند آرایه‌ای اشیا استفاده کنیم.

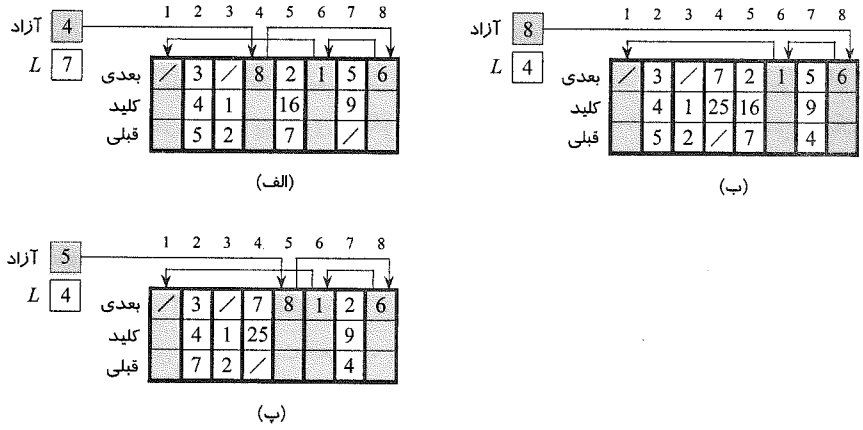
## تخصیص دادن و آزاد کردن اشیا

برای درج یک کلید در یک مجموعه‌ی پویا که به صورت لیست پیوندی دو طرفه نمایش داده شده است، باید یک اشاره‌گر به یک شیئ بی‌استفاده در لیست پیوندی اختصاص دهیم. بنابراین مناسب

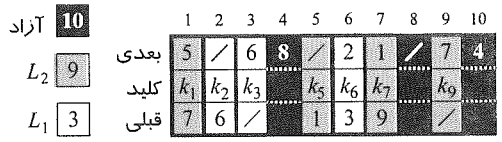
است که حافظه‌ی مربوط به اشیای بی‌استفاده‌ی لیست پیوندی را اداره کنیم، تا بتوانیم بعداً از این حافظه‌ها برای تخصیص اشیای جدید استفاده کنیم. در بعضی سیستم‌ها، یک *زباله‌روب* (garbage collector) مسئول تعیین این است که کدام یک از اشیا بی‌استفاده‌اند. با این حال بسیاری از کاربردها به حد کافی ساده هستند که بتوانند مسئولیت آزاد کردن اشیای بی‌استفاده را بر عهده بگیرند. اکنون مسئله‌ی اختصاص دادن و آزاد کردن اشیای ناهمگن را با استفاده از مثال یک لیست پیوندی دو طرفه که به صورت چند آرایه‌ای نمایش داده شده است، بررسی می‌کنیم.

فرض کنید آرایه‌ها در نمایش چند آرایه‌ای، طول $m$ دارند و در لحظه‌ای خاص، مجموعه‌ی پویا حاوی $n \le m$ عنصر است. در این صورت، این $n$ شیئ نشان‌دهنده‌ی عناصر فعلی درون مجموعه‌ی پویا هستند، و بقیه‌ی $m-n$ شیئ، *آزاد* هستند؛ از اشیای آزاد می‌توان برای نمایش عناصری که بعداً در مجموعه‌ی پویا درج می‌شوند استفاده کرد.

اشیای آزاد را در یک لیست پیوندی یک طرفه، که آن را *لیست آزاد* می‌نامیم، نگه خواهیم داشت. این لیست آزاد فقط از آرایه‌ی *next* استفاده می‌کند، که اشاره‌گرهای *next* درون لیست را ذخیره می‌کند. ابتدای لیست آزاد در یک متغیر خارجی (global variable) با نام *free* نگه‌داری می‌شود. وقتی لیست پیوندی $L$ ناتهی است، لیست آزاد می‌تواند با لیست $L$ در آمیخته شود، همان طور که در شکل ۱۰-۷ نشان داده شده است. توجه کنید که هر شیئ یا در لیست $L$ است و یا در لیست آزاد، ولی نمی‌تواند در هر دو باشد.




**شکل ۱۰-۷** تأثیر رویه‌های ALLOCATE-OBJECT و FREE-OBJECT. (الف) لیست شکل ۱۰-۵ (با سایه‌ی کم‌رنگ) و لیست آزاد (با سایه‌ی پررنگ). فلش‌ها نشان دهنده‌ی ساختار لیست آزاد هستند. (ب) نتیجه‌ی فراخوانی ALLOCATE-OBJECT () (که اندیس ۴ را بازمی‌گرداند)، مقداردهی *key*[۴] با ۲۵، و فراخوانی LIST-INSERT $(L, 4)$. ابتدای جدید لیست آزاد، شیئ ۸ است، که قبلا در لیست آزاد، *next*[۴] بوده است. (پ) پس از اجرای LIST-DELETE $(L, 5)$ رویه‌ی FREE-OBJECT (۵) را فراخوانی می‌کنیم. شیئ ۵، ابتدای جدید لیست آزاد می‌شود، که شیئ ۸ در لیست آزاد پس از آن قرار دارد.




**شکل ۱۰-۸** دو لیست پیوندی، $L_1$ (سایه‌ی کم‌رنگ) و $L_2$ (سایه‌ی پررنگ)، و یک لیست آزاد (تیره) که با هم آمیخته شده‌اند.

لیست آزاد یک پشته است؛ شیئ اختصاص داده شده‌ی بعدی، آخرین شیئی است که آزاد شده بود. می‌توانیم از پیاده‌سازی لیستی عملیات پشته‌ی PUSH و POP به ترتیب برای پیاده‌سازی رویه‌های تخصیص و آزاد کردن اشیا استفاده کنیم. فرض می‌کنیم متغیر خارجی *free* که در رویه‌ی زیر از آن استفاده شده است به اولین عنصر لیست آزاد اشاره می‌کند.

```
ALLOCATE-OBJECT()
1  if free == NIL
2      error "out of space"
3  else x = free
4      free = x.next
5      return x

FREE-OBJECT(x)
1  x.next = free
2  free = x
```

لیست آزاد در ابتدا حاوی تمام $n$ شیئ آزاد است. وقتی تمام لیست آزاد مصرف شد، فراخوانی رویه‌ی ALLOCATE-OBJECT منجر به یک پیغام خطا خواهد شد. معمول است که فقط 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 در آرایه‌های *key*، *next*، و *prev* با هم آمیخته شده‌اند.

دو رویه در زمان $O(1)$ اجرا می‌شوند، که آن‌ها را در عمل بسیار کاربردی می‌کند. می‌توان آن‌ها را طوری اصلاح کرد که با هر مجموعه‌ی ناهمگنی از اشیا کار کنند، بدین صورت که می‌توان از هرکدام از فیلدهای شیئ به عنوان فیلد *next* در لیست آزاد استفاده کرد.

## تمرین‌ها

۱-۳-۱۰ با استفاده از نمایش چند آرایه‌ای، شکلی از یک لیست پیوندی دو طرفه بکشید که دنباله‌ی $\langle 13,4,8,19,5,11 \rangle$ در آن ذخیره شده است. همین کار را برای نمایش تک آرایه‌ای انجام دهید.

۲-۳-۱۰ رویه‌های ALLOCATE-OBJECT و FREE-OBJECT را برای یک مجموعه‌ی ناهمگن از اشیا که به صورت نمایش تک آرایه‌ای پیاده‌سازی شده است، بنویسید.

۱۰-۳-۴ چرا در پیاده‌سازی رویه‌های ALLOCATE-OBJECT و FREE-OBJECT، نیازی به مقداردهی و یا بازنشانی فیلدهای *prev* در اشیا نداریم؟

۱۰-۳-۴ معمولاً مطلوب است که تمام عناصر یک لیست پیوندی دو طرفه را در یک حافظه‌ی فشرده نگه داریم، مثلاً با استفاده از اولین $m$ مکان در نمایش چند آرایه‌ای. (این حالتی است که در محیط‌های محاسباتی صفحه بندی شده با حافظه‌ی مجازی وجود دارد.) توضیح دهید که چطور می‌توان رویه‌های ALLOCATE-OBJECT و FREE-OBJECT را طوری پیاده‌سازی کرد که نمایش لیست، فشرده باشد. فرض کنید هیچ اشاره‌گری به عناصر لیست پیوندی بیرون لیست وجود ندارد. (*راهنمایی:* از پیاده‌سازی آرایه‌ای پشته استفاده کنید.)

۱۰-۳-۵ فرض کنید $L$ یک لیست پیوندی دو طرفه با طول $m$ است که در آرایه‌های *key*، *next*، و *prev* با طول $n$ نگه داری شده می‌شود. فرض کنید این آرایه‌ها به کمک رویه‌های ALLOCATE-OBJECT و FREE-OBJECT اداره می‌شوند تا یک لیست آزاد دو طرفه‌ی $F$ را نگه دارند. باز هم فرض کنید که از $n$ عنصر، دقیقاً $m$ تای آن‌ها در لیست $L$ و $n-m$ تای دیگر در لیست آزاد هستند. یک رویه‌ی COMPACTIFY-LIST$(L, F)$ بنویسید که با داشتن لیست $L$ و لیست آزاد $F$، عناصر درون $L$ را طوری جابه‌جا می‌کند که مکان‌های $1, 2, \ldots, m$ را اشغال کنند، و لیست آزاد $F$ را طوری اصلاح می‌کند که همچنان معتبر باشد، و مکان‌های آرایه‌ی $m+1, m+2, \ldots, n$ را اشغال کند. زمان اجرای رویه‌ی شما باید $\theta(m)$ باشد، و فقط باید از مقدار ثابتی حافظه‌ی اضافی استفاده کند. بحثی دقیق در مورد درستی رویه‌ی خود ارائه کنید.

## ۱۰-۴ نمایش درخت‌های ریشه‌دار

متدهای نمایش لیست که در بخش قبل ارائه شد، برای بسیاری از ساختمان‌های داده‌ی ناهمگن قابل استفاده است. در این بخش به طور خاص به مسئله‌ی نمایش درخت‌های ریشه‌دار به کمک لیست‌های پیوندی خواهیم پرداخت. ابتدا درخت‌های دودویی را بررسی خواهیم کرد، و سپس یک متد برای درخت‌های ریشه‌دار ارائه می‌کنیم که در آن گره‌ها می‌توانند تعداد دلخواهی فرزند داشته باشند.

هر گره‌ی درخت را به صورت یک شیئ نمایش خواهیم داد. مانند لیست‌های پیوندی، فرض می‌کنیم هر گره حاوی یک فیلد *key* است. فیلدهای مهم دیگر، اشاره‌گرهایی به گره‌های دیگرند، و بسته به نوع درخت متفاوت خواهند بود.

### درخت‌های دودویی

همان‌طور که در شکل ۱۰-۹ نشان داده شده است، از فیلدهای *p*، *left*، و *right* برای ذخیره‌ی

اشاره‌گرهایی به پدر، فرزند سمت چپ، و فرزند سمت راست هر گره در درخت دودویی $T$ استفاده می‌کنیم. اگر $x.p = \text{NIL}$، آن گاه $x$ ریشه‌ی درخت است. اگر $x$ فرزند سمت چپ نداشته باشد، آن گاه $x.left = \text{NIL}$، و همین طور است برای فرزند سمت راست. ریشه‌ی کل درخت $T$ به وسیله‌ی خصوصیت $T.root$ تعیین می‌شود. اگر $T.root = \text{NIL}$، آن گاه درخت تهی است.

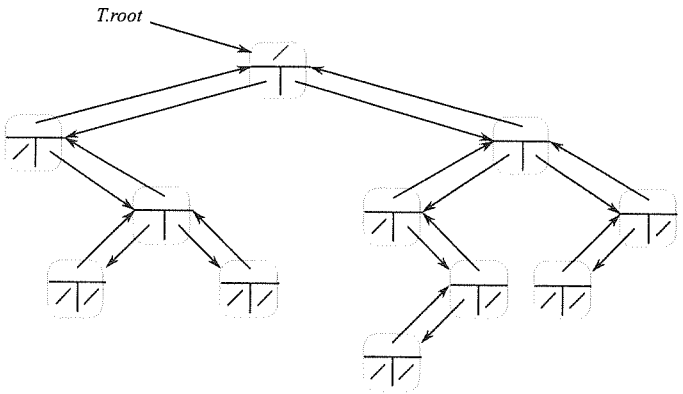
## درخت‌های ریشه‌دار با شاخه‌های نامحدود

روشی که برای نمایش درخت‌های دودویی به کار بردیم می‌تواند برای هر نوع درختی که تعداد فرزندان گره‌ها در آن حداکثر $k$ است، گسترش یابد: فیلدهای $left$ و $right$ با فیلدهای $child_1$، $child_2$، ...، $child_k$ جایگزین می‌شوند. این طرح برای وقتی که تعداد فرزندان نامحدود است، قابل استفاده نیست، چرا که نمی‌دانیم باید چند فیلد (آرایه در نمایش چند آرایه‌ای) به هر شیئ اختصاص دهیم. به علاوه حتی اگر تعداد فرزندان با $k$ محدود شده باشد، ولی این عدد ثابتی بزرگ باشد و اکثر گره‌ها تعداد کمی فرزند داشته باشند، مقدار زیادی حافظه هدر خواهد رفت.

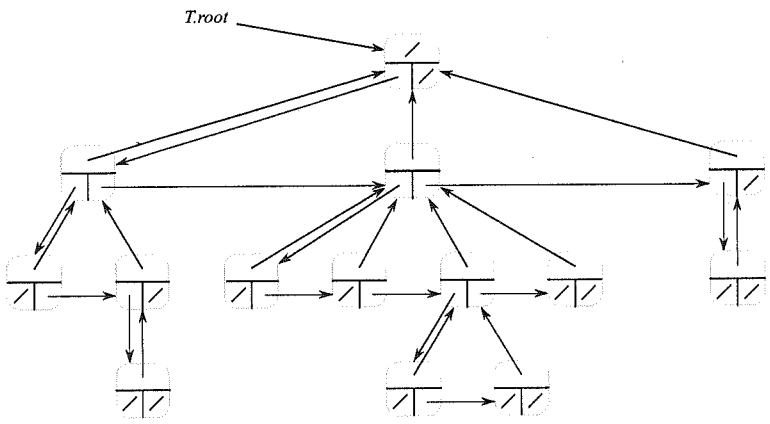
خوشبختانه روش هوشمندانه‌ای برای نمایش درخت‌های با شاخه‌های نامحدود وجود دارد. این طرح، این مزیت را دارد که برای نمایش یک درخت ریشه‌دار با $n$ گره، فقط $O(n)$ حافظه مصرف می‌کند. نمایش *فرزند چپ، برادر راست* (left-child, right-sibling representation) در شکل ۱۰-۱۰ نشان داده شده است. مانند قبل، هر گره حاوی یک اشاره‌گر $p$ به پدر خود است، و $T.root$ به ریشه‌ی درخت $T$ اشاره می‌کند. با این حال هر گره به جای این که یک اشاره‌گر به هر یک از فرزندان خود داشته باشد، فقط دو اشاره‌گر دارد:

۱. $x.left\text{-}child$ به چپ‌ترین فرزند گره‌ی $x$ اشاره می‌کند.
۲. $x.right\text{-}sibling$ به برادر گره‌ی $x$ در سمت راست اشاره می‌کند.

اگر گره‌ی $x$ هیچ فرزندی نداشته باشد، آن گاه $x.left\text{-}child = \text{NIL}$، و اگر $x$ راست‌ترین فرزند پدر خود باشد، آن گاه $x.right\text{-}sibling = \text{NIL}$.




شکل ۱۰-۹ نمایش یک درخت دودویی $T$. هر گره‌ی $x$ دارای فیلدهای $x.p$ (بالا)، $x.left$ (پایین و چپ)، و $x.right$ (پایین و راست) است. فیلدهای $key$ نشان داده نشده‌اند.




شکل ۱۰-۹ نمایش فرزند چپ، برادر راست برای درخت $x$. هر گره‌ی $x$ حاوی فیلدهای $x.p$ (بالا)، $x.left\text{-}child$ (پایین و چپ)، و $x.right\text{-}sibling$ (پایین و راست) است. کلیدها نشان داده نشده‌اند.

## نمایش‌های دیگر درخت‌ها

بعضی مواقع درخت‌های ریشه‌دار را به روش‌های دیگری نمایش می‌دهیم. مثلاً در فصل ۶، هرم‌ها را معرفی کردیم، که بر پایه‌ی یک درخت دودویی کامل است، و به وسیله‌ی یک آرایه و یک اندیس نمایش داده می‌شود. درخت‌هایی که در فصل ۲۱ خواهیم دید، فقط به سمت ریشه پیمایش خواهند شد، و بنابراین فقط اشاره‌گرهای پدر وجود خواهند داشت؛ هیچ اشاره‌گری به فرزندان در آن‌ها وجود ندارد. روش‌های بسیار دیگری هم وجود دارد. این که کدام روش بهتر است به کاربرد مربوطه بستگی دارد.

## تمرین‌ها

۱۰-۳-۱ درخت دودویی که اطلاعات فیلدهای آن در زیر آمده و ریشه‌ی آن در اندیس ۶ است را بکشید.

| اندیس | کلید | چپ | راست |
|---|---|---|---|
| ۱ | ۱۲ | ۷ | ۳ |
| ۲ | ۱۵ | ۸ | *NIL* |
| ۳ | ۴ | ۱۰ | *NIL* |
| ۴ | ۱۰ | ۵ | ۹ |
| ۵ | ۲ | *NIL* | *NIL* |
| ۶ | ۱۸ | ۱ | ۴ |
| ۷ | ۷ | *NIL* | *NIL* |
| ۸ | ۱۴ | ۶ | ۲ |
| ۹ | ۲۱ | *NIL* | *NIL* |
| ۱۰ | ۵ | *NIL* | *NIL* |

۱۰-۴-۲ یک رویه‌ی بازگشتی با زمان اجرای $O(n)$ بنویسید که یک درخت دودویی با $n$ گره را دریافت کرده و کلید هر گره در درخت را در صفحه نمایش چاپ می‌کند.

۱۰-۴-۳ یک رویه‌ی غیر بازگشتی با زمان اجرای $O(n)$ بنویسید که یک درخت دودویی با $n$ گره را دریافت کرده و کلید هر گره در درخت را در صفحه نمایش چاپ می‌کند. از یک پشته به عنوان ساختمان داده‌ی کمکی استفاده کنید.

۱۰-۴-۴ یک رویه با زمان اجرای $O(n)$ بنویسید که تمام کلیدهای یک درخت دلخواه ریشه‌دار با $n$ گره را چاپ می‌کند، که در آن درخت به روش فرزند چپ، برادر راست ذخیره شده است.

★ ۱۰-۴-۵ یک رویه‌ی غیر بازگشتی با زمان $O(n)$ بنویسید که یک درخت دودویی با $n$ گره را دریافت کرده و کلید هر گره را نمایش می‌دهد. فقط می‌توانید از مقدار ثابتی حافظه خارج از خود درخت استفاده کنید، و نباید درخت را تغییر دهید، حتی به طور موقت در طول اجرای رویه.

★ ۱۰-۴-۶ نمایش فرزند چپ، برادر راست برای درخت‌های دلخواه ریشه‌دار از سه اشاره‌گر در هر گره استفاده می‌کند: *left-child*، *right-sibling*، و *parent*. می‌توان از هر گره به پدر آن گره در زمان ثابت دسترسی پیدا کرد، و همچنین دسترسی به هر یک از فرزندان آن گره در زمان خطی نسبت به تعداد فرزندان انجام می‌شود. نشان دهید که چطور می‌توان با استفاده از فقط دو اشاره‌گر و یک مقدار بولین در هر گره، به پدر و تمام فرزندان گره در زمان خطی نسبت به تعداد فرزندان دسترسی پیدا کرد.

## مسائل

۱۰-۱ ***مقایسه‌ی لیست‌ها***

برای هر یک از چهار نوع لیست در جدول زیر، زمان اجرای حدی هر یک از اعمال مجموعه‌های پویا که در زیر آمده‌اند، چقدر است؟

| | مرتب نشده، یک طرفه | مرتب شده، یک طرفه | مرتب نشده، دو طرفه | مرتب شده، دو طرفه |
|---|---|---|---|---|
| SEARCH($L$, $k$) | | | | |
| INSERT($L$,$x$) | | | | |
| DELETE($L$, $x$) | | | | |
| SUCCESSOR($L$, $x$) | | | | |
| PREDECESSOR($L$, $x$) | | | | |
| MINIMUM($L$) | | | | |
| MAXIMUM($L$) | | | | |

### ۱۰-۲ هرم‌های قابل ادغام با استفاده از لیست‌های پیوندی

یک *هرم قابل ادغام* (mergeable heap) از عملیات زیر پشتیبانی می‌کند: MAKE-HEAP (که یک هرم قابل ادغام تهی تولید می‌کند) INSERT، MINIMUM، EXTRACT-MIN، و UNION.[1] در هر یک از حالت‌های زیر، نشان دهید که چگونه می‌توان با استفاده از یک لیست پیوندی، هرم‌های قابل ادغام را پیاده‌سازی کرد. سعی کنید هر یک از اعمال در حد ممکن بهینه باشد. زمان اجرای هر یک از اعمال را نسبت به اندازه‌ی مجموعه(ها)ی پویا تحلیل کنید.

I. لیست‌ها مرتب شده‌اند.

II. لیست‌ها نامرتب‌اند.

III. لیست‌ها نامرتب، و مجموعه‌های پویایی که باید ادغام شوند، گسسته‌اند.

### ۱۰-۳ جستجو در یک لیست مرتب فشرده

در تمرین ۱۰-۳-۴ از شما خواسته شد که توضیح دهید چگونه می‌توان یک لیست $n$ عنصری را در $n$ مکان اول یک آرایه نگه داشت. فرض خواهیم کرد که تمام کلیدها یکتا هستند و لیست فشرده، مرتب شده است، یعنی برای $i = 1, 2, ..., n$ به طوری که $next[i] \neq \text{NIL}$ داریم $key[i] < key[next[i]]$. همچنین فرض می‌کنیم یک متغیر $L$ داریم حاوی اولین عنصر لیست. تحت این فرض‌ها، شما باید نشان دهید که از الگوریتم تصادفی زیر می‌توان برای جستجو در این لیست استفاده کرد، و امیدریاضی زمان اجرای آن $O(\sqrt{n})$ است.

```
COMPACT-LIST-SEARCH(L, n, k)
1   i = L
2   while i ≠ NIL and key[i] < k
3       j = RANDOM(1, n)
4       if key[i] < key[j] and key[j] ≤ k
5           i = j
6           if key[i] = k
7               return i
8       i = next[i]
9   if i = NIL or key[i] > k
10      return NIL
11  else return i
```

اگر از خطوط ۳-۷ رویه‌ی بالا صرف نظر کنیم، یک الگوریتم معمولی برای جستجو در یک لیست مرتب شده خواهیم داشت، که در آن اندیس $i$ به هر یک از مکان‌های لیست اشاره می‌کند. جستجو زمانی پایان می‌یابد که اندیس $i$ از انتهای لیست «بیرون بیفتد»، و یا زمانی که

---

[1] چون یک هرم قابل ادغام را طوری تعریف کرده‌ایم که از MINIMUM و EXTRACT-MIN پشتیبانی کند، می‌توانیم آن را *هرم کمینه قابل ادغام* هم بنامیم. از سوی دیگر اگر هرم از MAXIMUM و EXTRACT-MAX پشتیبانی کند، هرم یک *هرم بیشینه قابل ادغام* خواهیم داشت.

$key[i] \geq k$. در حالت دوم، اگر $key[i] = k$ مسلماً یک کلید با مقدار $k$ پیدا کرده‌ایم. با این حال اگر $key[i] > k$، در این صورت هیچ وقت یک کلید با مقدار $k$ پیدا نخواهیم کرد، و بنابراین پایان دادن به جستجو کار درستی است.

خطوط ۳-۷ سعی می‌کنند که به سمت یک مکان تصادفی $j$ پرش کنند. چنین پرشی زمانی مفید است که $key[j]$ بزرگ‌تر از $key[i]$ باشد، ولی از $k$ بزرگ‌تر نباشد؛ در چنین حالتی، $j$ مکانی در آرایه است که در یک جستجوی معمولی $i$ باید از روی آن عبور کند. از آن‌جایی که لیست فشرده است، مطمئن خواهیم بود که هر انتخابی برای اندیس $j$ بین ۱ و $n$ به یک شیئ در لیست اشاره می‌کند، و نه یک مکان خالی در لیست آزاد.

به جای تحلیل کارایی COMPACT-LIST-SEARCH به صورت مستقیم، یک الگوریتم مشابه، یعنی COMPACT-LIST-SERACH′ را تحلیل خواهیم کرد، که دو حلقه‌ی جداگانه را اجرا می‌کند. این الگوریتم یک پارامتر اضافه‌ی $t$ می‌گیرد، که کرانی بالا بر روی تعداد تکرارهای حلقه‌ی اول تعیین می‌کند.

```
COMPACT-LIST-SEARC′ (L, n, k, t)
1  i = L
2  for q = 1 to t
3      j = RANDOM(1, n)
4      if key[i] < key[j] and key[j] ≤ k
5          i = j
6          if key[i] == k
7              return i
8  while i ≠ NIL and key[i] < k
9      i = next[i]
10 if i == NIL or key[i] > k
11     return NIL
12 else return i
```

برای مقایسه‌ی اجرای الگوریتم‌های COMPACT-LIST-SEARCH $(L, k)$ و COMPACT-LIST-SEARCH′ $(L, k, t)$، فرض کنید دنباله‌ی اعداد صحیحی که توسط فراخوانی‌های RANDOM $(1, n)$ بازگردانده می‌شود، برای هر دو الگوریتم یکسان است.

فرض کنید در COMPACT-LIST-SEARCH $(L, k)$، حلقه‌ی while در خطوط ۲-۸ به تعداد $t$ بار تکرار می‌شود. بحث کنید که COMPACT-LIST-SEARCH′ $(L, k, t)$ جوابی یکسان بازمی‌گرداند و تعداد کل تکرارهای هر دو حلقه‌ی 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 فاصله‌ی مکان $i$ تا کلید مورد نظر $k$ را در لیست پیوندی (از میان زنجیره‌ی اشاره‌گرهای $next$) پس از $t$ تکرار از حلقه‌ی for در خطوط ۲-۷ توصیف می‌کند.

II. بحث کنید که امیدریاضی زمان اجرای COMPACT-LIST-SEARCH$'(L,k,t)$ برابر است با $O(t+E[X_t])$.

III. نشان دهید که $E[X_t] \le \sum_{r=1}^{n}(1-r/n)^t$. (*راهنمایی:* از تساوی (پ-۲۵) استفاده کنید.)

IV. نشان دهید که $\sum_{r=0}^{n-1} r^t \le n^{t+1}/(t+1)$.

V. اثبات کنید که $E[X_t] \le n/(t+1)$.

VI. نشان دهید که امیدریاضی زمان اجرای COMPACT-LIST-SEARCH$'(L,k,t)$ عبارت است از $O(t+n/t)$.

VII. نتیجه بگیرید که امیدریاضی زمان اجرای COMPACT-LIST-SEARCH برابر است با $O(\sqrt{n})$.

VIII. چرا در COMPACT-LIST-SEARCH فرض کردیم که تمام کلیدها یکتا هستند؟ بحث کنید که زمانی که لیست حاوی کلیدهای تکراری است، پرش‌های تصادفی به بهبود زمان اجرای حدی کمکی نمی‌کنند.

فصل

# ۱۱

# جداول درهم

## ۱۱-۰ مقدمه

در بسیاری از کاربردها به مجموعه‌ی پویایی نیاز داریم که فقط از عملیات دیکشنری INSERT، SEARCH، و DELETE پشتیبانی کند. مثلاً کامپایلری که یک زبان برنامه‌نویسی را ترجمه می‌کند دارای یک جدول نمادها است، که در آن کلید عناصر، رشته‌های دلخواهی از کاراکترها هستند که متناظرند با شناسه‌هایی در زبان. یک جدول درهم ساختمان داده‌ای مؤثر برای پیاده‌سازی دیکشنری‌ها است. با این که جستجوی یک عنصر در جدول درهم می‌تواند به اندازه‌ی جستجو در لیست پیوندی طول بکشد- زمان $\theta(n)$ در بدترین حالت- در عمل، استفاده از درهم‌سازی بسیار خوب عمل می‌کند. تحت فرض‌های معقول، امیدریاضی زمان جستجوی عناصر در یک جدول درهم $O(1)$ است.

جدول درهم تعمیمی از مفهوم ساده‌تر یک آرایه‌ی معمولی است. آدرس‌دهی مستقیم در آرایه‌های معمولی باعث می‌شود بتوانیم یک مکان دلخواه در آرایه را در زمان $O(1)$ بررسی کنیم. در بخش ۱۱-۱ آدرس‌دهی مستقیم با جزئیات بیشتر مورد بحث قرار خواهد گرفت. آدرس‌دهی مستقیم زمانی قابل استفاده است که آن قدر استطاعت داشته باشیم که یک مکان در آرایه برای هر کلید ممکن در نظر بگیریم.

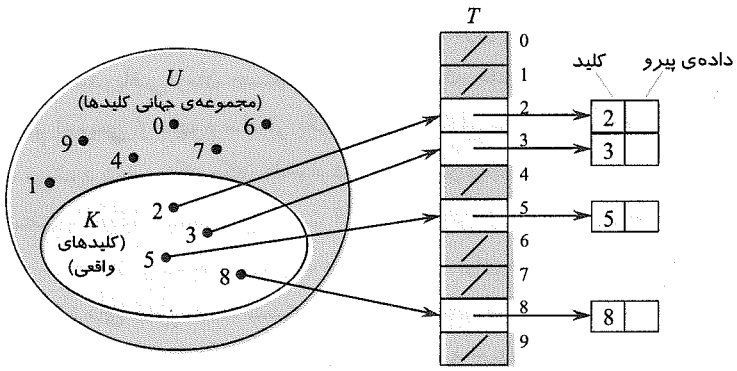
وقتی تعداد کلیدهای ذخیره شده نسبت به تعداد کلیدهای ممکن کم باشد، جدول درهم جانشینی مؤثر برای آدرس‌دهی مستقیم در آرایه خواهد بود، چرا که جداول درهم معمولاً از یک آرایه با اندازه‌ای متناسب با تعداد کلیدهای ذخیره شده استفاده می‌کنند. به جای استفاده از کلید به صورت مستقیم به عنوان اندیس آرایه، اندیس آرایه از روی کلید *محاسبه* می‌شود. در بخش ۱۱-۲ ایده‌های اصلی بررسی می‌شود، با تأکید بر روی «زنجیره‌ای کردن» (chaining) به عنوان یک روش برای مدیریت «تصادم‌ها» (collision) در صورتی که بیشتر از یک کلید به یک اندیس در آرایه نگاشت (map) شوند.

در بخش ۱۱-۳ توضیح داده خواهد شد که چگونه می‌توان با استفاده از توابع درهم‌ساز، اندیس‌های آرایه را از روی کلیدها محاسبه کرد. حالت‌های مختلفی از این موضوع پایه‌ای ارائه و تحلیل خواهد شد. در بخش ۱۱-۴ نگاهی خواهیم داشت به «آدرس‌دهی باز» (open addressing)، که راه دیگری است برای مقابله با تصادم‌ها. سخن آخر این است که درهم‌سازی تکنیکی است بسیار کاربردی و مؤثر: عملیات دیکشنری پایه‌ای در حالت متوسط فقط به زمان $O(1)$ نیاز دارند. در بخش ۱۱-۵ خواهیم دید که چگونه به کمک «درهم‌سازی کامل» (perfect hashing) می‌توان جستجوها را در *بدترین حالت* در زمان $O(1)$ انجام داد، البته در صورتی که مجموعه‌ی کلیدهای مورد استفاده ایستا باشد (یعنی حالتی که کلیدها پس از ذخیره شدن هرگز تغییر نمی‌کنند).

## ۱۱-۱ جداول آدرس مستقیم

آدرس‌دهی مستقیم، تکنیکی ساده است که وقتی مجموعه‌ی جهانی (universe) $U$ مربوط به کلیدها کوچک باشد، خوب کار می‌کند. فرض کنید یک کاربرد به یک مجموعه‌ی پویا نیاز دارد که در آن هر عنصر کلیدی دارد از مجموعه‌ی $U = \{0,1,\ldots,m-1\}$، که در آن $m$ چندان بزرگ نیست. فرض خواهیم کرد که هیچ دو عنصری کلید برابر ندارند.

برای نشان دادن این مجموعه‌ی پویا، از یک آرایه، یا یک *جدول آدرس مستقیم* (direct-address table) استفاده می‌کنیم که با $T[0 \ldots m-1]$ نمایش داده می‌شود، که در آن هر مکان متناسب است با یک کلید در مجموعه‌ی جهانی $U$. شکل ۱۱-۱ این روش را مشخص می‌کند؛ مکان $k$ به یک عنصر در مجموعه با کلید $k$ اشاره می‌کند. اگر مجموعه شامل هیچ عنصری با کلید $k$ نباشد، آن گاه $T[k] = \text{NIL}$.




**شکل ۱۱-۱** پیاده‌سازی یک مجموعه‌ی پویا با جدول آدرس مستقیم $T$. هر کلید در مجموعه‌ی جهانی $U = \{0,1,\ldots,9\}$ متناظر است با یک اندیس در جدول. مجموعه‌ی $K = \{2,3,5,8\}$ از کلیدهای واقعی تعیین کننده‌ی مکان‌هایی در جدول است که حاوی اشاره‌گرهایی به عناصر می‌باشند. مکان‌های دیگر، که با سایه‌ی پررنگ مشخص شده‌اند، حاوی NIL هستند.

عملیات دیکشنری در این پیاده‌سازی بدیهی هستند.

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

تمام این عملیات سریع هستند: فقط زمان $O(1)$ برای اجرای آن‌ها مورد نیاز است.

برای بعضی کاربردها، عناصر مجموعه‌ی پویا می‌توانند در خود جدول آدرس مستقیم ذخیره شوند. یعنی به جای این که کلید عنصر و داده‌های پیرو را در یک شیئ خارج از جدول آدرس مستقیم ذخیره کنیم و یک اشاره‌گر به آن شیئ در یکی از مکان‌های جدول قرار دهیم، می‌توانیم با ذخیره‌ی خود شیئ در آن مکان در حافظه صرفه‌جویی کنیم. برای نشان دادن یک مکان خالی در یک شیئ از یک کلید ویژه استفاده خواهیم کرد. به علاوه معمولاً نیازی نیست که فیلد کلید شیئ را ذخیره کنیم، چرا که اگر اندیس شیئ را در جدول داشته باشیم، کلید آن را هم داریم. با این حال، اگر کلیدها ذخیره نشوند باید روشی داشته باشیم که بتوانیم خالی بودن یک مکان را تشخیص دهیم.

## تمرین‌ها

۱۱-۱-۱ فرض کنید مجموعه‌ی پویای $S$ به کمک یک جدول آدرس مستقیم $T$ با طول $m$ نشان داده شده است. یک رویه تعریف کنید که عنصر بیشینه‌ی $S$ را پیدا می‌کند. کارایی رویه‌ی شما در بدترین حالت چقدر است؟

۱۱-۱-۲ یک *بردار بیتی* (bit vector) یک آرایه‌ی ساده از بیت‌ها (۰ها و ۱ها) است. یک بردار بیتی با طول $m$ بسیار کم‌تر از یک آرایه از $m$ اشاره‌گر فضا اشغال می‌کند. توضیح دهید که چطور می‌توان از یک بردار بیتی برای نشان دادن یک مجموعه‌ی پویا از عناصر یکتا بدون داده‌های پیرو استفاده کرد. عملیات دیکشنری باید در زمان $O(1)$ اجرا شوند.

۱۱-۱-۳ یک روش پیشنهاد کنید برای پیاده‌سازی یک جدول آدرس مستقیم که در آن لزومی ندارد کلید عناصر ذخیره شده یکتا باشند، و عناصر می‌توانند داده‌ی پیرو داشته باشند. هر سه عملیات دیکشنری (INSERT، DELETE، و SEARCH) باید در زمان $O(1)$ اجرا شوند. (فراموش نکنید که DELETE یک اشاره‌گر به عنصری که باید حذف شود را به عنوان آرگومان می‌گیرد، و نه یک کلید.)

۱۱-۱-۴ ★ می‌خواهیم با استفاده از آدرس‌دهی مستقیم بر روی یک آرایه‌ی بزرگ، یک دیکشنری پیاده‌سازی کنیم. در آغاز ممکن است مکان‌های آرایه حاوی مقادیر نامربوط (garbage) باشند، و مقداردهی تمام آرایه به خاطر اندازه‌ی آن کاربردی نیست. یک روش برای پیاده‌سازی این دیکشنری آدرس مستقیم بر روی یک آرایه‌ی بزرگ پیشنهاد کنید. هر عنصر

ذخیره شد باید به اندازه‌ی $O(1)$ حافظه اشغال کند؛ عملیات SEARCH، INSERT، و DELETE هر یک باید در زمان $O(1)$ اجرا شوند؛ و مقداردهی اولیه‌ی ساختمان داده باید در زمان $O(1)$ انجام شود. (*راهنمایی:* از یک پشته‌ی کمکی استفاده کنید، که اندازه‌ی آن تعداد کلیدهایی است که در دیکشنری ذخیره شده‌اند، و به کمک آن تشخیص دهید که آیا یک مکان خاص در این آرایه‌ی بزرگ حاوی مقداری معتبر است یا خیر.)

## ۱۱-۲ جداول درهم

مشکل آدرس‌دهی مستقیم واضح است: اگر مجموعه‌ی جهانی $U$ بزرگ باشد، ذخیره‌ی یک جدول با اندازه‌ی $|U|$ ممکن است غیر کاربردی، و یا حتی بر روی حافظه‌ی یک کامپیوتر معمولی ناممکن باشد. به علاوه $K$، مجموعه‌ی کلیدهایی که *واقعاً* ذخیره شده‌اند، ممکن است نسبت به $U$ بسیار کوچک باشد، به طوری که اکثر فضایی که به $T$ اختصاص داده شده است به هدر رود.

وقتی مجموعه‌ی کلیدهای ذخیره شده در دیکشنری بسیار کوچک‌تر از مجموعه‌ی جهانی تمام کلیدهای ممکن باشد، یک جدول درهم نسبت به یک جدول آدرس‌دهی مستقیم بسیار حافظه‌ی کمتری مصرف می‌کند. به طور خاص، می‌توان حافظه‌ی مصرفی را به $\theta(|K|)$ کاهش داد، در حالی که همچنان مزیت جستجو در زمان $O(1)$ را خواهیم داشت. تنها مسئله این است که این کران برای *زمان متوسط* است، در حالی که در آدرس‌دهی مستقیم این زمان در *بدترین حالت* برقرار است.

در آدرس‌دهی مستقیم، یک عنصر با کلید $k$ در مکان $k$ ذخیره می‌شود. در جداول درهم این عنصر در مکان $h(k)$ ذخیره می‌شود؛ یعنی از یک *تابع درهم‌ساز* (hash function) $h$ برای محاسبه‌ی مکانی که کلید $k$ باید در آن قرار گیرد استفاده می‌کنیم. در این جا، $h$ مجموعه‌ی جهانی $U$ را به مکان‌ها در *جدول درهم* $T[0..m-1]$ نگاشت می‌کند:

$$h : U \to \{0,1,\ldots,m-1\}$$

که در آن اندازه‌ی جدول درهم، $m$، معمولاً بسیار کوچک‌تر از $|U|$ است. می‌گوییم یک عنصر با کلید $k$ به مکان $h(k)$ *درهم‌ساخته* می‌شود؛ همچنین می‌گوییم $h(k)$ *مقدار درهم‌سازی شده‌ی* کلید $k$ است. شکل ۱۱-۲ ایده‌ی اولیه را مشخص می‌کند. نکته‌ی تابع درهم‌ساز این است که دامنه‌ی اندیس‌های آرایه را کوچک کنیم. به جای $|U|$، اندازه‌ی آرایه می‌تواند فقط $m$ باشد.

در این میان یک مسئله وجود دارد: ممکن است دو کلید به یک مکان نگاشت شوند. به این موقعیت یک *تصادم* (collision) می‌گوییم. خوشبختانه روش‌های کارامدی وجود دارند که به کمک آن‌ها می‌توان مشکل تصادم‌ها را حل کرد.

مسلماً راه حل ایده‌آل این است که کلا از تصادم جلوگیری کنیم. ممکن است بخواهیم با انتخاب یک تابع درهم‌ساز مناسب $h$ به این هدف دست یابیم. یک ایده این است که $h$ تقریباً «تصادفی» باشد، که بدین صورت از تصادم‌ها جلوگیری می‌شود، و یا تعداد آن‌ها به حداقل می‌رسد. اصطلاح

«درهم‌سازی» این ایده را الهام می‌کند. (مسلماً تابع درهم‌ساز $h$ باید قطعی باشد تا یک ورودی خاص $k$ همیشه یک مقدار ثابت $h(k)$ تولید کند.) با این حال از آن جایی که $|U| > m$، باید حداقل دو کلید وجود داشته باشند که مقدار درهم‌سازی شده‌ی برابری دارند؛ بنابراین جلوگیری از تصادم‌ها غیرممکن است. پس با این که یک تابع درهم‌ساز که خوب طراحی شده و به نظر «تصادفی» است می‌تواند تعداد تصادم‌ها را کمینه کند، باز هم به یک روش برای حل مسئله‌ی تصادم‌ها نیاز خواهیم داشت.

در ادامه‌ی این بخش به ساده‌ترین تکنیک حل مسئله‌ی تصادم‌ها می‌پردازیم، که زنجیره‌ای کردن (chaining) نام دارد. در بخش ۱۱-۴، یک روش دیگر برای این مسئله با نام آدرس‌دهی باز (open addressing) معرفی خواهد شد.

## حل مسئله‌ی تصادم به کمک زنجیره‌ای سازی

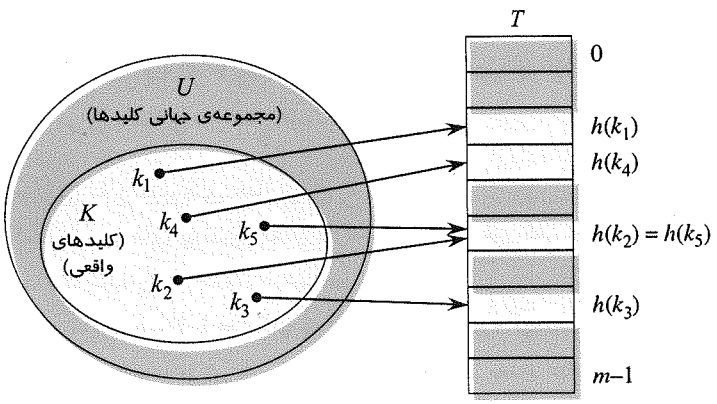
در *زنجیره‌ای سازی*، تمام عناصری را که به یک مکان نگاشت می‌شوند را در یک لیست پیوندی ذخیره می‌کنیم، همان طور که در شکل ۱۱-۳ نشان داده شده است. مکان $j$ حاوی اشاره‌گری به ابتدای یک لیست است که شامل تمام عناصری است که کلید آن‌ها به $j$ نگاشت می‌شود؛ اگر چنین عناصری وجود نداشته باشند، مکان $j$ حاوی NIL خواهد بود.

وقتی که تصادم‌ها با زنجیره‌ای سازی حل شوند، پیاده‌سازی عملیات دیکشنری بر روی یک جدول درهم $T$ بسیار ساده خواهد بود.

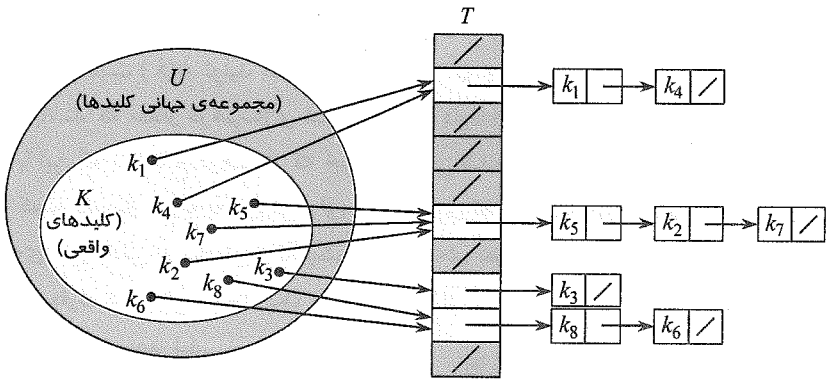
```
CHAINED-HASH-INSERT(T, x)
1  insert x at the head of list T[h(x.key)]

CHAINED-HASH-SEARCH(T, k)
1  search for an element with key k
2  in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
1  delete x from the list T[h(x.key)]
```




**شکل ۱۱-۲** استفاده از تابع درهم‌ساز $h$ برای نگاشت کلیدها به مکان‌های جدول درهم. کلیدهای $k_2$ و $k_5$ به یک مکان نگاشت می‌شوند، و بین آن‌ها یک تصادم پیش می‌آید.




**شکل ۱۱-۳** حل مسئله‌ی تصادم به کمک زنجیره‌ای سازی. هر مکان جدول درهم، $T[j]$، حاوی یک لیست پیوندی از تمام کلیدهایی است که مقدار درهم‌سازی شده‌ی آن‌ها $j$ است. مثلاً، $h(k_1) = h(k_4)$ و $h(k_5) = h(k_7) = h(k_2)$. لیست‌ها می‌توانند یک‌طرفه یا دوطرفه باشند؛ در این جا آن‌ها را به صورت دوطرفه نشان می‌دهیم، چرا که حذف در این حالت بسیار سریع‌تر است.

بدترین حالت زمان اجرای درج عناصر $O(1)$ است. رویه‌ی درج تا حدودی سریع است چرا که فرض می‌کند عنصر $x$ که قرار است درج شود، از قبل در جدول وجود ندارد؛ در صورت لزوم می‌توان این فرض را، با انجام یک جستجو (با هزینه‌ی اضافی) برای عنصری با کلید $x.key$ قبل از درج چک کرد. برای جستجو، بدترین حالت زمان اجرا متناسب است با طول لیست؛ در ادامه این عملیات را دقیق‌تر بررسی می‌کنیم. اگر لیست‌ها دوطرفه باشند، حذف یک عنصر می‌تواند در زمان $O(1)$ انجام شود. (توجه کنید که CHAINED-HASH-DELETE عنصر $x$ را به عنوان ورودی می‌گیرد، و نه کلید آن را، و بنابراین نیازی به جستجوی عنصر $x$ نداریم. اگر جدول درهم از حذف هم پشتیبانی کند، آن گاه لیست پیوندی آن باید دوطرفه باشد تا بتوانیم به سرعت عناصر مورد نظر را حذف کنیم. اگر لیست‌ها یک طرفه باشند، دریافت خود عنصر $x$ به جای کلید آن به عنوان ورودی کمک چندانی نخواهد کرد. همچنان مجبور خواهیم بود که $x$ را در لیست $T[h(x.key)]$ بیابیم تا بتوانیم اشاره‌گر $next$ عنصر قبل از $x$ را به درستی مقداردهی کنیم. در لیست‌های یک طرفه، جستجو و حذف ذاتاً زمان اجرای یکسانی خواهند داشت.)

## تحلیل درهم‌سازی با زنجیره‌ای سازی

درهم‌سازی به کمک لیست‌های پیوندی چقدر کارامد است؟ به طور کلی چقدر طول می‌کشد که یک عنصر با یک کلید داده شده را در جدول درهم جستجو کنیم؟

با داشتن یک جدول درهم $T$ با $m$ مکان که $n$ عنصر را ذخیره کرده‌اند، *فاکتور بار* (load factor) جدول $T$ را به صورت $\alpha = n/m$ تعریف می‌کنیم، یعنی متوسط تعداد عناصر ذخیره شده در هر لیست پیوندی. تحلیل ما نسبت به $\alpha$ خواهد بود، که ممکن است کوچک‌تر، مساوی، و یا بزرگ‌تر از ۱ باشد.

بدترین حالت رفتار درهم‌سازی با لیست‌های پیوندی وحشتناک خواهد بود: تمام $n$ کلید به یک

مکان نگاشت می‌شوند و یک لیست پیوندی با طول $n$ می‌سازند. بنابراین بدترین حالت جستجو برابر است با $\theta(n)$ به علاوه‌ی زمان محاسبه‌ی تابع درهم‌ساز – بدتر از زمانی که از یک لیست پیوندی برای ذخیره‌ی تمام عناصر استفاده می‌کنیم. به وضوح، از جداول درهم برای بدترین حالت زمان اجرای آن‌ها استفاده نمی‌شود. (با این حال درهم سازی کامل که در بخش ۱۱-۵ توضیح داده خواهد شد، در حالتی که مجموعه‌ی کلیدها ایستا باشد بدترین حالت زمان اجرای خوبی دارد.)

کارایی متوسط درهم‌سازی به این بستگی دارد که تابع درهم‌ساز $h$ چقدر خوب عناصر را در میان $m$ مکان توزیع می‌کند. در بخش ۱۱-۳ در این مورد بحث خواهد شد، ولی در این جا فرض می‌کنیم که هر عنصر داده شده با احتمال یکسان به هرکدام از $m$ مکان موجود در جدول درهم نگاشت می‌شود، مستقل از این که عناصر دیگر به کجا نگاشت شده‌اند. این فرض را، فرض *درهم‌سازی ساده‌ی متوازن* می‌نامیم.

اجازه دهید برای $j = 0,1,\ldots,m-1$، طول لیست $T[j]$ را با $n_j$ نشان دهیم. بنابراین

$$n = n_0 + n_1 + \ldots + n_{m-1} \qquad (۱۱-۱)$$

و مقدار متوسط $n_j$ برابر است با $E[n_j] = \alpha = n/m$.

فرض می‌کنیم می‌توان مقدار درهم‌سازی شده‌ی $h(k)$ را در زمان $O(1)$ محاسبه کرد، و بنابراین زمان مورد نیاز برای جستجوی یک عنصر با کلید $k$ به صورت خطی به $n_{h(k)}$ برای لیست $T[h(k)]$ وابسته است. با کنار گذاشتن زمان $O(1)$ برای محاسبه‌ی مقدار درهم‌سازی شده و همچنین دسترسی به مکان $h(k)$، اجازه دهید که امیدریاضی تعداد عناصر آزمایش شده توسط الگوریتم جستجو را بررسی کنیم، یعنی تعداد عناصری در لیست $T[h(k)]$ که مقدار کلید آن‌ها بررسی می‌شود تا کلیدی با مقدار $k$ یافت شود. دو حالت را در نظر خواهیم گرفت. در حالت اول جستجو ناموفق است: کلید هیچ عنصری برابر $k$ نیست. در حالت دوم با موفقیت یک عنصر با کلید $k$ یافت می‌شود.

**قضیه ۱۱-۱** در یک جدول درهم که در آن تصادم‌ها به کمک زنجیره‌ای سازی حل می‌شوند، امیدریاضی زمان یک جستجوی ناموفق $\theta(1+\alpha)$ است، با فرض درهم‌سازی ساده‌ی متوازن.

*اثبات* با فرض درهم‌سازی ساده‌ی متوازن، هر کلید $k$ که در جدول ذخیره نشده است با احتمال برابر به هر یک از $m$ مکان نگاشت می‌شود. امیدریاضی زمان اجرای یک جستجوی ناموفق برابر است با امیدریاضی زمان یک جستجو تا انتهای لیست $T[h(k)]$، که امیدریاضی طول آن برابر است با $E[n_{h(k)}] = \alpha$. بنابراین، امیدریاضی تعداد عناصر چک شده در یک جستجوی ناموفق $\alpha$، و کل زمان (به همراه زمان محاسبه‌ی $h(k)$) برابر است با $\theta(1+\alpha)$. ■

در یک جستجوی موفق شرایط مقداری متفاوت است، چرا که احتمال جستجوی تمام لیست‌ها با یکدیگر برابر نیست. در عوض احتمال جستجو شدن یک لیست متناسب است با تعداد عناصری که در

آن ذخیره شده است. با این حال امیدریاضی زمان جستجو $\theta(1+\alpha)$ است.

**قضیه‌ی ۱۱-۲** در یک جدول درهم که در آن تصادم‌ها به کمک زنجیره‌ای سازی حل می‌شوند، امیدریاضی زمان یک جستجوی موفق $\theta(1+\alpha)$ است، با فرض درهم‌سازی ساده‌ی متوازن.

*اثبات* فرض می‌کنیم عنصری که به دنبال یافتن آن هستیم با احتمال برابر هر یک از $n$ عنصر ذخیره شده در جدول است. تعداد عناصری که برای یافتن $x$ چک می‌شوند یکی بیشتر است از تعداد عناصری که در لیست $x$ قبل از $x$ قرار دارند. چون عناصر جدید در ابتدای لیست درج می‌شوند، عناصر قبل از $x$ در لیست، همگی بعد از درج $x$ در جدول درج شده‌اند. برای یافتن امیدریاضی تعداد عناصری که چک می‌شوند، برای هر کدام از $n$ عنصر جدول، امیدریاضی تعداد عناصری که بعد از آن عنصر به لیست آن اضافه شده‌اند به علاوه‌ی ۱ را محاسبه کرده و از آن‌ها میانگین می‌گیریم. فرض کنید $x_i$ نشان‌دهنده‌ی $i$امین عنصر درج شده در جدول باشد، که $i = 1,2,...,n$، و فرض کنید $k_i = x_i.key$. برای کلیدهای $k_i$ و $k_j$، متغیر تصادفی شاخص $X_{ij} = I\{h(k_i) = h(k_j)\}$ را تعریف می‌کنیم. تحت فرض درهم‌سازی ساده‌ی متوازن، داریم $\Pr\{h(k_i) = h(k_j)\} = 1/m$، و بنابراین طبق لم ۵-۱ داریم $E[X_{ij}] = 1/m$. پس امیدریاضی تعداد عناصر چک شده در یک جستجوی موفق برابر است با

$$
\begin{aligned}
&E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] \\
&\quad=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) && \text{(طبق خطی بودن امیدریاضی)} \\
&\quad=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&\quad=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&\quad=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&\quad=1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) && \text{(طبق تساوی (الف-۱))} \\
&\quad=1+\frac{n-1}{2m} \\
&\quad=1+\frac{\alpha}{2}-\frac{\alpha}{2n}
\end{aligned}
$$

بنابراین، کل زمان مورد نیاز برای یک جستجوی موفق (با احتساب زمان محاسبه‌ی تابع درهم‌ساز)

برابر است با $\theta(2+\alpha/2-\alpha/2n)=\theta(1+\alpha)$.

معنای این تحلیل چیست؟ اگر تعداد مکان‌ها در جدول درهم حداقل متناسب با تعداد عناصر درون جدول باشد، داریم $n=O(m)$، و در نتیجه $\alpha=n/m=O(m)/m=O(1)$. بنابراین جستجو در حالت متوسط به زمان ثابت نیاز دارد. از آن جایی که درج در بدترین حالت در زمان $O(1)$، و اگر لیست‌ها دو طرفه باشند حذف در بدترین حالت در زمان $O(1)$ انجام می‌شود، تمام عملیات دیکشنری در حالت متوسط در زمان $O(1)$ اجرا خواهند شد.

## تمرین‌ها

۱۱-۲-۱ فرض کنید یک تابع درهم‌ساز $h$ برای درهم‌سازی $n$ کلید یکتا در یک آرایه‌ی $T$ با طول $m$ داریم. با فرض درهم‌سازی ساده‌ی متوازن، امیدریاضی تعداد تصادم‌ها چقدر است؟ به طور دقیق‌تر، امیدریاضی تعداد اعضای مجموعه‌ی $\{\{k,l\}: k \neq l \text{ and } h(k)=h(l)\}$ چقدر است؟

۱۱-۲-۲ درج عناصر ۵، ۲۸، ۱۹، ۱۵، ۲۰، ۳۳، ۱۲، ۱۷، ۱۰ را در یک جدول درهم زنجیره‌ای نشان دهید. فرض کنید که جدول ۹ مکان دارد، و تابع درهم‌ساز عبارت است از $h(k)=k \bmod 9$.

۱۱-۲-۳ پروفسور مارلی (Marley) ادعا می‌کند که اگر در جدول درهم زنجیره‌ای، عناصر درون لیست‌ها را به صورت مرتب شده نگه داریم، می‌توانیم کارایی کلی این روش را افزایش دهیم. تحلیل کنید که این اصلاح در روش زنجیره‌ای سازی چگونه بر روی زمان اجرای جستجوهای موفق، جستجوهای ناموفق، درج، و حذف اثر می‌گذارد.

۱۱-۲-۴ یک روش پیشنهاد کنید که به کمک آن بتوان اختصاص حافظه به عناصر و آزاد کردن حافظه را درون خود جدول درهم، و به وسیله‌ی پیوند دادن تمام جاهای خالی در جدول درون یک لیست آزاد انجام داد. فرض کنید هر مکان خالی می‌تواند یک پرچم (flag) به علاوه‌ی یک اشاره‌گر و یک عنصر، و یا دو اشاره‌گر را در خود ذخیره کند. تمام اعمال دیکشنری و لیست آزاد باید با امیدریاضی زمان اجرای $O(1)$ انجام شوند. آیا لیست آزاد باید دوطرفه باشد، و یا یک لیست آزاد پیوندی یک طرفه کافی است؟

۱۱-۲-۵ فرض کنید مجموعه‌ای از $n$ کلید را در یک جدول درهم با اندازه‌ی $m$ ذخیره می‌کنیم. نشان دهید که اگر کلیدها از یک مجموعه‌ی جهانی $U$ استخراج شوند، به طوری که $|U| \geq nm$، یک زیرمجموعه از $U$ با اندازه‌ی $n$ وجود دارد که حاوی کلیدهایی است که همه به یک مکان در جدول درهم نگاشت می‌شوند، به طوری که بدترین حالت زمان

جستجو برای جدول درهم زنجیره‌ای $\theta(n)$ است.

۱۱-۲-۶ فرض کنید مجموعه‌ای از $n$ کلید را در یک جدول درهم با اندازه‌ی $m$ ذخیره کرده‌ایم، که در آن تصادم‌ها با زنجیره‌ای سازی حل می‌شوند. به علاوه طول هر زنجیره را می‌دانیم، همچنین $L$، طول بزرگ‌ترین زنجیره را. رویه‌ای با امیدریاضی زمان اجرای $O(L\cdot(1+1/\alpha))$ توصیف کنید که به صورت یکنواخت یک کلید از میان کلیدهای جدول درهم انتخاب می‌کند.

## ۱۱-۳ توابع درهم‌ساز

در این بخش، در مورد مسائل مربوط به طراحی توابع درهم‌ساز خوب بحث خواهیم کرد، و سپس سه روش برای ساختن این توابع معرفی می‌کنیم. دو تا از این روش‌ها، درهم‌سازی تقسیمی (hashing by division) و درهم‌سازی ضربی (hashing by multiplication)، به طور طبیعی مکاشفه‌ای (heuristic) هستند، در حالی که روش سوم، درهم‌سازی جهانی (universal hashing)، از تصادفی‌سازی برای دست‌یابی به کارایی خوب و قابل اثبات استفاده می‌کند.

### چه چیز باعث می‌شود که یک تابع درهم‌ساز خوب باشد؟

یک تابع درهم‌ساز خوب (تقریباً) فرض درهم‌سازی ساده‌ی متوازن را ارضا می‌کند: هر کلید با احتمال یکسان به هر یک از $m$ مکان نگاشت می‌شود، مستقل از این که کلیدهای دیگر به کجا نگاشت شده‌اند. متأسفانه، معمولاً ممکن نیست که چنین چیزی را چک کنیم، چرا که احتمال خیلی کمی دارد که توزیع احتمالاتی کلیدها را بدانیم، و حتی ممکن است کلیدها مستقل از یکدیگر نباشند.

بعضی مواقع نحوه‌ی توزیع کلیدها را می‌دانیم. مثلاً اگر بدانیم که کلیدهای $k$، اعداد حقیقی تصادفی هستند که به صورت مستقل و یکنواخت در بازه‌ی $0 \le k < 1$ توزیع شده‌اند، آن گاه تابع درهم‌ساز

$$h(k) = \lfloor km \rfloor$$

فرض درهم‌سازی ساده‌ی یکنواخت را ارضا می‌کند.

در عمل، معمولاً می‌توان از تکنیک‌های مکاشفه‌ای برای ساختن یک تابع درهم‌ساز خوب استفاده کرد. اطلاعات کیفی در مورد توزیع کلیدها می‌توانند در این طراحی مفید باشند. مثلاً جدول نماد (symbol table) یک کامپایلر را در نظر بگیرید، که در آن کلیدها رشته‌های کاراکتری هستند که نشان دهنده‌ی شناسه‌ها در یک برنامه می‌باشند. معمولاً در یک برنامه از نمادهای متشابه، مانند pt و pts زیاد استفاده می‌شود. یک تابع درهم‌ساز خوب احتمال این که چنین نمادهایی به یک مکان نگاشت شوند را به حداقل می‌رساند.

یک رویکرد خوب این است که مقدار درهم‌سازی شده را طوری تعریف کنیم که از تمام الگوهایی که ممکن است در داده وجود داشته باشد، مستقل باشد. مثلاً «روش تقسیمی» (که در بخش ۱۱-۳-۱ توضیح داده خواهد شد)، مقدار درهم را به صورت باقی‌مانده‌ی کلید بر یک عدد اول خاص تعریف

می‌کند. این روش معمولاً نتیجه‌ی خوبی دارد، با فرض این که عدد اول طوری انتخاب شود که از هر الگویی در توزیع کلیدها مستقل باشد.

نهایتاً، توجه می‌کنیم که بعضی از کاربردهای توابع درهم‌ساز ممکن است به فرض‌هایی قوی‌تر از درهم‌سازی ساده‌ی یکنواخت نیاز داشته باشند. مثلاً ممکن است بخواهیم کلیدهایی که به یکدیگر «نزدیک» هستند، مقادیر درهم دوری داشته باشند. (به طور خاص این خصوصیت هنگام استفاده از کاوش خطی مفید است، که در بخش ۱۱-۴ آن را تعریف خواهیم کرد.) درهم‌سازی جهانی که در بخش ۱۱-۳-۳ توضیح داده خواهد شد، معمولاً خصوصیات مورد نظر را فراهم می‌کند.

## تفسیر کلیدها به صورت اعداد طبیعی

اکثر توابع درهم‌ساز فرض می‌کنند که مجموعه‌ی جهانی کلیدها، مجموعه‌ی اعداد طبیعی $\mathbb{N} = \{0,1,2,\ldots\}$ است. بنابراین اگر کلیدها اعداد طبیعی نباشند، باید با روشی آن‌ها را به صورت اعداد طبیعی تفسیر کنیم. مثلاً یک رشته‌ی کاراکتری را می‌توان به کمک نشان‌گذاری مبنایی (radix notation) مناسب به یک عدد طبیعی تبدیل کرد. بنابراین ممکن است شناسه‌ی pt به صورت جفت اعداد دهدهی (۱۱۲,۱۱۶) تفسیر شود، چرا که در مجموعه‌ی کاراکترهای اسکی (ASCII) داریم $p = 112$ و $t = 116$؛ آن گاه به صورت یک عدد مبنای ۱۲۸ رشته‌ی pt تبدیل شود به $116 + (128 \times 112) = 14452$. این کار بسیار معمول است که در یک کاربرد، یک روش برای ترجمه‌ی کلیدها به اعداد طبیعی (احتمالاً بزرگ) ابداع کنیم. در ادامه فرض می‌کنیم کلیدها اعداد طبیعی هستند.

## ۱۱-۳-۱ روش تقسیمی

در *روش تقسیمی* (division method) برای ساختن توابع درهم‌ساز، با گرفتن باقی‌مانده‌ی $k$ بر $m$، هر کلید $k$ را به یکی از $m$ مکان ممکن نگاشت می‌کنیم. یعنی، تابع درهم‌ساز عبارت است از

$$h(k) = k \bmod m$$

برای مثال اگر اندازه‌ی جدول درهم $m = 12$ و کلید $k = 100$ باشد، آن گاه $h(k) = 4$. از آن جایی که این تابع فقط به یک عملیات تقسیم نیاز دارد، این روش بسیار سریع است.

وقتی از روش تقسیمی استفاده می‌کنیم معمولاً از مقادیر خاص $m$ دوری می‌کنیم. مثلاً $m$ نباید توانی از ۲ باشد، چرا که اگر $m = 2^p$، آن گاه $h(k)$ به سادگی برابر است با $p$ بیت کم‌ارزش $k$. غیر از حالتی که می‌دانیم احتمال وقوع تمام الگوهای $p$ بیتی کم‌ارزش برابر است، بهتر است تابع درهم‌ساز را طوری تعریف کنیم که به تمام بیت‌های کلید بستگی داشته باشد. همان طور که تمرین ۱۱-۳-۳ از شما می‌خواهد که نشان دهید، انتخاب $m = 2^p - 1$ وقتی $k$ یک رشته‌ی کاراکتری تفسیر شده به صورت عدد مبنای $2^p$ است، انتخاب ضعیفی است، چرا که تغییر جایگشت کاراکترهای $k$ مقدار درهم‌سازی شده‌ی آن را تغییر نمی‌دهد.

یک عدد اول که به توان ۲ نزدیک نباشد معمولاً انتخاب خوبی برای $m$ است. مثلاً فرض کنید

می‌خواهیم یک جدول درهم پیوندی برای نگهداری رشته‌های تقریباً $n = ۲۰۰۰$ کاراکتری اختصاص دهیم، که در آن هر کاراکتر ۸ بیت دارد. این که در هر جستجوی ناموفق به طور متوسط ۳ عنصر را آزمایش کنیم، هزینه‌ی چندان زیادی ندارد، بنابراین یک جدول درهم با اندازه‌ی $m = ۷۰۱$ تخصیص می‌دهیم. عدد ۷۰۱ برای این انتخاب شده است که یک عدد اول نزدیک ۲۰۰۰/۳ است، ولی نزدیک توانی از ۲ نیست. با در نظر گرفتن کلیدها به صورت اعداد صحیح، تابع درهم‌ساز ما عبارت خواهد بود از

$$h(k) = k \bmod ۷۰۱$$

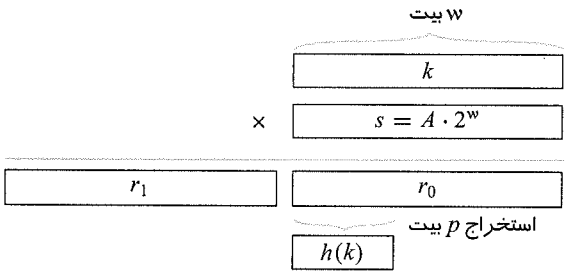
### ۱۱-۳-۲ روش ضربی

روش ضربی (multiplication method) برای ساختن توابع درهم‌ساز در دو مرحله عمل می‌کند. اول کلید $k$ را در یک عدد ثابت $A$ در فاصله‌ی $۰ < A < ۱$ ضرب می‌کنیم و قسمت صحیح $kA$ را حذف می‌کنیم (یعنی فقط قسمت اعشاری را نگه می‌داریم). سپس این مقدار را در $m$ ضرب می‌کنیم و از حاصل جزء صحیح می‌گیریم. به طور خلاصه، تابع درهم عبارت است از

$$h(k) = \lfloor m(kA \bmod ۱) \rfloor$$

که در آن $kA \bmod ۱$ یعنی قسمت اعشاری $kA$، یعنی $kA - \lfloor kA \rfloor$.

یک مزیت روش ضربی این است که مقدار $m$ در آن مهم نیست. در این روش، معمولاً $m$ را به صورت توانی از ۲ در نظر می‌گیریم ($m = ۲^p$ برای یک عدد صحیح $p$) چرا که در این صورت می‌توانیم تابع را بر روی اکثر کامپیوترها به سادگی و به صورت زیر پیاده‌سازی کنیم. فرض کنید اندازه‌ی هر کلمه‌ی ماشین $w$ بیت است، و $k$ در یک کلمه جا می‌شود. $A$ را بدین صورت محدود می‌کنیم که باید کسری به شکل $s/۲^w$ باشد، که در آن $s$ یک عدد صحیح در فاصله‌ی $۰ < s < ۲^w$ است. با توجه به شکل ۱۱-۴، ابتدا $k$ را در عدد صحیح $w$ بیتی $s = A \cdot ۲^w$ ضرب می‌کنیم. نتیجه یک عدد $۲w$ بیتی $r_1 ۲^w + r_0$ خواهد بود، که در آن $r_1$ کلمه‌ی سمت راست حاصل ضرب، و $r_0$ کلمه‌ی سمت چپ آن است. مقدار درهم‌سازی شده‌ی مورد نظر، $p$ بیت پر ارزش $r_0$ خواهد بود.




**شکل ۱۱-٤** روش ضربی برای تابع درهم‌ساز. نمایش $w$ بیتی کلید $k$ در مقدار $w$ بیتی $s = A \cdot ۲^w$ ضرب می‌شود. $p$ بیت پر ارزش نیمه‌ی پایینی بیت‌های حاصل ضرب، مقدار مورد نظر تابع درهم‌ساز $h(k)$ خواهد بود.

با این حال که این روش برای هر مقداری از ثابت $A$ کار می‌کند، ولی با بعضی مقادیر بهتر از بقیه جواب می‌دهد. انتخاب بهینه به خصوصیات داده‌ای که باید درهم‌سازی شود بستگی دارد. Knuth پیشنهاد می‌کند که

$$A \approx (\sqrt{5}-1)/2 = 0{,}6180339887... \qquad (2-11)$$

در حد معقولی خوب خواهد بود.

به‌عنوان یک مثال، فرض کنید داریم $k = 123456$، $p = 14$، $m = 2^{14} = 16384$، و $w = 32$. با در پیش گرفتن پیشنهاد Knuth مقدار $A$ را طوری انتخاب می‌کنیم که کسری به شکل $s/2^{32}$ و نزدیک به $(\sqrt{5}-1)/2$ باشد، یعنی $A = 2654435769/2^{32}$. آن‌گاه $k \cdot s = 327706022297664 = (76300 \times 2^{32}) + 1761282864$، و بنابراین $r_1 = 76300$ و $r_0 = 1761282864$. ۱۴ بیت پرارزش $r_0$ مقدار $h(k) = 67$ را به ما می‌دهند.

### ★ ۱۱-۳-۳ درهم‌سازی جهانی

اگر یک دشمن بدخواه بخواهد اعدادی را که باید با یک تابع ثابت درهم‌سازی شوند انتخاب کند، می‌تواند $n$ کلید را طوری انتخاب کند که همه به یک مکان نگاشت شوند، که در این صورت زمان متوسط جستجو $\theta(n)$ خواهد شد. هر تابع درهم‌ساز ثابت این اشکال بزرگ را دارد؛ تنها راه مؤثر برای بهبود این وضعیت این است که تابع درهم‌ساز را به صورت *تصادفی* طوری انتخاب کنیم که *مستقل* از کلیدهایی باشد که ذخیره می‌شوند. این رویکرد، که درهم‌سازی جهانی (universal hashing) نامیده می‌شود، کارایی خوب قابل اثباتی را در حالت متوسط نتیجه می‌دهد، بدون توجه به این که دشمن بدخواه چه کلیدهایی انتخاب می‌کند.

ایده‌ی اصلی پشت درهم‌سازی جهانی این است که در ابتدای اجرای برنامه، تابع درهم‌ساز را به صورت تصادفی از میان تعدادی تابع درهم‌ساز به دقت طراحی شده انتخاب کنیم. مانند مرتب‌سازی سریع، تصادفی‌سازی تضمین می‌کند که هیچ ورودی خاصی نمی‌تواند همیشه باعث بدترین حالت رفتار شود. به خاطر تصادفی‌سازی، الگوریتم می‌تواند در هر بار اجرا رفتار متفاوتی داشته باشد، حتی برای ورودی‌های ثابت، که باعث می‌شود کارایی در حالت متوسط برای هر ورودی بالا باشد. با بازگشت به مثال جدول نماد کامپایلر متوجه می‌شویم که اکنون انتخاب شناسه‌های برنامه‌نویس نمی‌تواند باعث افت کارایی درهم‌ساز شود. کارایی ضعیف فقط زمانی اتفاق می‌افتد که کامپایلر به صورت تصادفی تابع درهم‌سازی را انتخاب کند که باعث شود مجموعه‌ی شناسه‌ها به صورت ضعیف نگاشت شوند، ولی احتمال رخ دادن چنین حالتی کم است، و برای تمام مجموعه‌های شناسه‌ی با اندازه‌ی برابر یکسان است.

فرض کنید H یک مجموعه‌ی متناهی از توابع درهم‌ساز باشد که یک مجموعه‌ی جهانی داده شده‌ی $U$ از کلیدها را به مجموعه‌ی $\{0,1,...,m-1\}$ می‌نگارد. گفته می‌شود چنین مجموعه‌ای جهانی است اگر برای هر جفت از کلیدهای مجزای $k,l \in U$، تعداد توابع درهم‌ساز $h \in H$ که $h(k) = h(l)$، حداکثر برابر با $H/m$ باشد. به عبارت دیگر، با یک تابع درهم‌ساز که به صورت تصادفی از H انتخاب

شده است، احتمال یک تصادم میان کلیدهای مجزای $k$ و $l$ بیشتر از $1/m$ نیست، که برابر است با احتمال یک تصادم میان $h(k)$ و $h(l)$ که به صورت تصادفی و مستقل از مجموعه‌ی $\{0,1,...,m-1\}$ انتخاب شده‌اند.

قضیه‌ی زیر نشان می‌دهد که یک مجموعه‌ی جهانی از توابع درهم‌ساز، رفتار حالت متوسط خوبی دارد. به یاد بیاورید که $n_i$ نشان‌دهنده‌ی طول لیست $T[i]$ است.

**قضیه‌ی ۱۱-۳**

فرض کنید یک تابع درهم‌ساز $h$ به صورت تصادفی از یک مجموعه‌ی جهانی توابع درهم‌ساز انتخاب شده است، و از آن برای درهم‌سازی $n$ کلید به یک جدول $T$ با اندازه‌ی $m$ استفاده شده است. همچنین فرض کنید از زنجیره‌ای سازی برای حل مسئله‌ی تصادم‌ها استفاده می‌شود. اگر کلید $k$ در جدول نباشد، در این صورت امیدریاضی طول لیستی که کلید $k$ به آن نگاشت می‌شود، $E[n_{h(k)}]$، حداکثر برابر است با فاکتور بار $\alpha = n/m$. اگر کلید $k$ در جدول باشد، در این صورت امیدریاضی طول لیست حاوی $k$، یعنی $E[n_{h(k)}]$، حداکثر برابر $1+\alpha$ است.

*اثبات* توجه می‌کنیم که در این جا امیدریاضی بر روی انتخاب‌های مختلف تابع درهم‌ساز گرفته می‌شود، و به هیچ فرضی در مورد توزیع کلیدها وابسته نیست. برای هر جفت $k$ و $l$ از کلیدهای مجزا، یک متغیر تصادفی شاخص به صورت $X_{kl} = I\{h(k) = h(l)\}$ تعریف می‌کنیم. از آن جایی که طبق تعریف یک مجموعه‌ی جهانی از توابع درهم، یک جفت از کلیدها حداکثر با احتمال $1/m$ تصادم می‌کنند، داریم $\Pr\{h(k) = h(l)\} \le 1/m$، و بنابراین لم ۵-۱ ایجاب می‌کند که $E[X_{kl}] \le 1/m$.

سپس برای هر کلید $k$ متغیر تصادفی شاخص $Y_k$ را تعریف می‌کنیم که برابر است با تعداد کلیدهایی غیر از $k$ که به همان مکانی نگاشت می‌شوند که $k$ نگاشت می‌شود، به طوری که

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}$$

بنابراین داریم

$$\begin{aligned} E[Y_k] &= E\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] \\ &= \sum_{\substack{l \in T \\ l \neq k}} [X_{kl}] \qquad \text{(طبق خطی بودن امیدریاضی)} \\ &\le \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m} \end{aligned}$$

ادامه‌ی اثبات به این بستگی دارد که آیا $k$ در جدول $T$ وجود دارد یا خیر.

- اگر $k \notin T$، آن گاه $n_{h(k)} = Y_k$ و $n = |\{l : l \in T \text{ and } l \neq k\}|$. بنابراین $E[n_{h(k)}] = E[Y_k] \le n/m = \alpha$.

• اگر $k \in k$، آن گاه از آن جایی که $k$ در لیست $T[h(k)]$ وجود دارد و شمارنده‌ی $Y_k$ شامل کلید $k$ نمی‌شود، داریم $n_{h(k)} = Y_k + ۱$ و $\left|\{l : l \in T \text{ and } l \neq k\}\right| = n - ۱$. بنابراین $E[n_{h(k)}] = E[Y_k] + ۱$. ■

نتیجه‌ی زیر می‌گوید که درهم‌سازی جهانی بازدهی مورد نظر را فراهم می‌کند: اکنون برای یک دشمن بدخواه غیر ممکن است که بتواند دنباله‌ای از اعمال را اجرا کند که باعث شود بدترین حالت زمان اجرا رخ دهد. با انتخاب هوشمندانه و تصادفی توابع درهم‌ساز در هنگام اجرا، تضمین خواهیم کرد که امیدریاضی زمان اجرای هر دنباله‌ای از اعمال خوب باشد.

**نتیجه‌ی ۱۱-۴** با استفاده از درهم‌سازی جهانی و زنجیره‌ای سازی در یک جدول با $m$ مکان، امیدریاضی زمان اجرای هر دنباله‌ای از $n$ عملیات INSERT، SEARCH، و DELETE که شامل $O(m)$ عملیات INSERT است، $\theta(n)$ است.

*اثبات* از آن جایی که تعداد درج‌ها $O(m)$ است، داریم $n = O(m)$ و بنابراین $\alpha = O(۱)$. اعمال INSERT و DELETE در زمان ثابت اجرا می‌شوند، و طبق قضیه‌ی ۱۱-۳، امیدریاضی زمان اجرا برای هر عملیات SEARCH برابر $O(۱)$ است. بنابراین، چون امیدریاضی خطی است، امیدریاضی زمان اجرای کل دنباله $O(n)$ است. از آن جایی که هر عملیات $\Omega(۱)$ زمان می‌گیرد، کران $\theta(n)$ برقرار است. ■

## طراحی یک مجموعه‌ی جهانی از توابع درهم‌ساز

به طور کلی طراحی یک مجموعه‌ی جهانی از توابع درهم‌ساز کار ساده‌ای است، که با مقدار کمی نظریه‌ی اعداد می‌توان آن را اثبات کرد. اگر با نظریه‌ی اعداد ناآشنا هستید می‌توانید به فصل ۳۱ مراجعه کنید.

با انتخاب یک عدد اول $p$ به اندازه‌ی کافی بزرگ شروع می‌کنیم، به طوری که تمام کلیدهای ممکن $k$ در فاصله‌ی بسته‌ی ۰ تا $p-۱$ باشند، به علاوه فرض کنید $Z_p$ نشان دهنده‌ی مجموعه‌ی $\{۰,۱,...,p-۱\}$، و $Z_p^*$ نشان دهنده‌ی $\{۱,۲,...,p-۱\}$. از آن جایی که $p$ اول است، می‌توانیم معادلات به پیمانه‌ی $p$ را با تکنیک‌های فصل ۳۱ حل کنیم. چون فرض کردیم اندازه‌ی مجموعه‌ی جهانی کلیدها از تعداد مکان‌های جدول درهم بیشتر است، داریم $p > m$.

اکنون با استفاده از یک تبدیل خطی و سپس ساده سازی به پیمانه‌ی $p$، و سپس به پیمانه‌ی $m$، تابع درهم‌ساز $h_{a,b}$ را برای هر $a \in Z_p^*$ و هر $b \in Z_p$ تعریف می‌کنیم:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m \qquad (۱۱-۳)$$

برای مثال، با $p = ۱۷$ و $m = ۶$ داریم $h_{۳,۴}(۸) = ۵$. خانواده‌ی تمام این توابع درهم‌ساز عبارت است از

$$H_{p,m} = \{h_{a,b} : a \in Z_p^* \text{ and } b \in Z_p\} \qquad (۱۱-۴)$$

هر تابع درهم‌ساز $h_{a,b}$ مقدار $Z_p$ را به $Z_m$ نگاشت می‌کند. مجموعه‌ی توابع درهم‌ساز این

خصوصیت خوب را دارد که اندازه‌ی $m$، یعنی محدوده‌ی خروجی، دلخواه است – و نه لزوماً عدد اول – خصوصیتی که از آن در بخش ۱۱-۵ استفاده خواهیم کرد. از آن جایی که $p-1$ انتخاب برای $a$ و $p$ انتخاب برای $b$ وجود دارد، در کل $p(p-1)$ تابع درهم‌ساز در $H_{p,m}$ وجود خواهد داشت.

**قضیه‌ی ۱۱-۵** مجموعه‌ی $H_{p,m}$ از توابع درهم‌ساز تعریف شده توسط تساوی‌های (۱۱-۳) و (۱۱-۴) جهانی است.

**اثبات** دو کلید مجزای $k$ و $l$ را از $Z_p$ در نظر بگیرید، به طوری که $k \neq l$. برای یک تابع درهم‌ساز داده شده‌ی $h_{a,b}$ تعریف می‌کنیم

$$r = (ak + b) \bmod p,$$
$$s = (al + b) \bmod p$$

ابتدا دقت می‌کنیم که $r \neq s$. چرا؟ مشاهده کنید که

$$r - s \equiv a(k - l) \pmod{p}$$

این نتیجه می‌دهد که $r \neq s$ چون $p$ یک عدد اول است و هر دوی $a$ و $(k-l)$ به پیمانه‌ی $p$ ناصفر هستند، و بنابراین نتیجه‌ی این حاصل ضرب هم باید (طبق قضیه‌ی ۳۱-۶) به پیمانه‌ی $p$ ناصفر باشد. بنابراین، حین محاسبه‌ی هر $h_{a,b}$ای در $H_{p,m}$، ورودی‌های مجزای $k$ و $l$ به مقادیر مجزای $r$ و $s$ به پیمانه‌ی $p$ نگاشت می‌شوند؛ تا «مرحله‌ی به پیمانه‌ی $p$» هنوز تصادمی نداریم. به علاوه، هر کدام از $p(p-1)$ انتخاب ممکن برای جفت $(a,b)$ با $a \neq 0$ به یک جفت متفاوت $(r,s)$ با $r \neq s$ منجر می‌شود، چرا که می‌توانیم با داشتن $a$ و $b$، مقادیر $r$ و $s$ را به دست آوریم:

$$a = ((r - s)((k - l)^{-1} \bmod p)) \bmod p,$$
$$b = (r - ak) \bmod p$$

که در آن $((k-l)^{-1} \bmod p)$ نشان‌دهنده‌ی وارون ضربی $k-l$ به پیمانه‌ی $p$ است. از آن جایی که فقط $p(p-1)$ جفت ممکن $(r,s)$ با $r \neq s$ وجود دارد، یک تناظر یک به یک بین جفت‌های $(a,b)$ با $a \neq 0$ و جفت‌های $(r,s)$ با $r \neq s$ وجود دارد. بنابراین برای هر جفت از ورودی‌های $k$ و $l$، اگر $(a,b)$ را به صورت تصادفی و یکنواخت از $Z_p^* \times Z_p$ انتخاب کنیم، جفت مرتب حاصل $(r,s)$ با احتمال برابر می‌تواند هر جفت از مقادیر متفاوت به پیمانه‌ی $p$ باشد.

این سپس نتیجه می‌دهد احتمال این که کلیدهای متفاوت $k$ و $l$ تصادم داشته باشند، برابر است با احتمال این که $r \equiv s \pmod{m}$، وقتی که $r$ و $s$ به صورت تصادفی و به شکل دو عدد نابرابر به پیمانه‌ی $p$ انتخاب شده‌اند. برای یک مقدار داده شده برای $r$، از $p-1$ مقدار ممکن باقی مانده برای $s$، تعداد مقادیری برای $s$ به طوری که $r \neq s$ و $s \equiv r \pmod{m}$ حداکثر برابر است با

$$\lceil p/m \rceil - 1 \le ((p+m-1)/m) - 1 \quad \text{(طبق تساوی (۳-۷))}$$
$$= (p-1)/m .$$

احتمال این که $s$ و $r$، وقتی به پیمانه‌ی $p$ محاسبه می‌شوند با هم تصادم کنند حداکثر برابر است با $((p-1)/m)/(p-1) = 1/m$.

بنابراین برای هر جفت مقادیر متفاوت $k, l \in Z_p$،

$$\Pr\{h_{a,b}(k) = h_{a,b}(l)\} \le 1/m$$

و از این رو $H_{p,m}$ جهانی است.

## تمرین‌ها

۱-۳-۱۱ فرض کنید می‌خواهیم در یک لیست پیوندی با طول $n$ جستجو کنیم، که در آن عناصر حاوی یک کلید $k$ با یک مقدار درهم $h(k)$ هستند. هر کلید یک رشته‌ی کاراکتری طولانی است. هنگام جستجوی یک عنصر با کلیدی خاص، چگونه می‌توانیم از مقادیر درهم کلیدها استفاده کنیم؟

۲-۳-۱۱ فرض کنید یک رشته از $r$ کاراکتر، با درنظر گرفتن به صورت اعداد مبنای ۱۲۸ و استفاده از روش تقسیمی به $m$ مکان درهم‌سازی شده است. عدد $m$ به سادگی به صورت یک عدد در یک کلمه‌ی ۳۲ بیتی در کامپیوتر نشان داده‌شده است، ولی رشته‌ی $r$ کاراکتری، که به صورت یک عدد مبنای ۱۲۸ در نظر گرفته می‌شود، تعداد زیادی کلمه در کامپیوتر اشغال می‌کند. چطور می‌توانیم با استفاده از تعداد ثابتی کلمه در حافظه، غیر از خود رشته، از روش تقسیمی برای محاسبه‌ی مقدار درهم رشته‌ی کاراکتری استفاده کنیم؟

۳-۳-۱۱ نسخه‌ای از روش تقسیمی را در نظر بگیرید که در آن $h(k) = k \bmod m$، که $m = 2^p - 1$ و $k$ یک رشته‌ی کاراکتری است که به صورت یک عدد مبنای $2^p$ در نظر گرفته می‌شود. نشان دهید که اگر رشته‌ی $x$ را بتوان با تغییر جایگشت کاراکترهای رشته‌ی $y$ به دست آورد، آن گاه مقدار درهم $x$ و $y$ برابر خواهد بود. مثالی از یک کاربرد ارائه کنید که در آن چنین خصوصیتی نامطلوب است.

۴-۳-۱۱ یک جدول درهم با اندازه‌ی $m = 1000$ و یک تابع درهم‌ساز متناظر به صورت $h(k) = \lfloor m(kA \bmod 1) \rfloor$ را در نظر بگیرید، که در آن $A = (\sqrt{5} - 1)/2$. مکان نگاشت کلیدهای ۶۱، ۶۲، ۶۳، ۶۴، و ۶۵ را محاسبه کنید.

۵-۳-۱۱ ★ یک خانواده‌ی H از توابع درهم‌ساز، از یک مجموعه‌ی متناهی $U$ به یک مجموعه‌ی متناهی $B$ را $\varepsilon$–جهانی تعریف می‌کنیم اگر برای تمام جفت‌هایی از عناصر متفاوت $k$ و $l$ در $U$ داشته باشیم

$$\Pr\{h(k) = h(l)\} \le \varepsilon$$

که در آن احتمال برای توابع درهم $h$ محاسبه می‌شود که به صورت تصادفی از خانواده‌ی H انتخاب می‌شوند. نشان دهید در که یک خانواده‌ی $\varepsilon$-جهانی از توابع درهم‌ساز باید داشته باشیم

$$\varepsilon \geq \frac{1}{|B|} - \frac{1}{|U|}$$

★ ۱۱-۳-۶ فرض کنید $U$ مجموعه‌ی $n$ تایی‌هایی باشد که مقادیر آن عضو $Z_p$ باشند، و همچنین $B = Z_p$، که در آن $p$ یک عدد اول است. تابع درهم‌ساز $h_b : U \to B$ را برای $b \in Z_p$ روی یک ورودی $n$ تایی $\langle a_0, a_1, ..., a_{n-1} \rangle$ از $U$ به صورت زیر تعریف می‌کنیم

$$h_b(\langle a_0, a_1, ..., a_{n-1} \rangle) = \left( \sum_{j=0}^{n-1} a_j b^j \right) \bmod p$$

و فرض کنید $H = \{h_b : b \in Z_p\}$. بحث کنید که طبق تعریف مجموعه‌های $\varepsilon$-جهانی، H یک مجموعه‌ی $((n-1)/p)$-جهانی است. (*راهنمایی:* تمرین ۳۱-۴-۴ را ببینید.)

## ۱۱-۴ آدرس‌دهی باز

در *آدرس‌دهی باز* (open addressing)، تمام عناصر در خود جدول درهم ذخیره می‌شوند، یعنی هر ورودی جدول یا حاوی یک عنصر از مجموعه‌ی پویا، و یا حاوی NIL است. وقتی برای یک عنصر جستجو می‌کنیم، مشخصاً مکان‌های جدول را آزمایش می‌کنیم تا عنصر مورد نظر یافت شود، و یا مشخص شود که این عنصر در جدول موجود نیست. برخلاف روش زنجیره‌ای سازی هیچ لیست و یا عنصری خارج از جدول وجود ندارد. بنابراین در آدرس‌دهی باز جدول درهم می‌تواند «پر شود» به طوری که دیگر نتوانیم هیچ درجی انجام دهیم؛ یک نتیجه این است که فاکتور بار $\alpha$ نمی‌تواند از ۱ فراتر رود.

مسلماً می‌توانستیم لیست‌های پیوندی مورد استفاده برای زنجیره‌ای سازی را درون خود جدول درهم، در مکان‌های خالی ذخیره کنیم (تمرین ۱۱-۲-۴ را ببینید)، ولی مزیت آدرس‌دهی باز این است که اصلاً از اشاره‌گرها استفاده نمی‌کند. به جای دنبال کردن اشاره‌گرها، ما دنباله‌ی مکان‌هایی که باید چک شوند را *محاسبه* می‌کنیم. حافظه‌ی اضافی آزاد شده به خاطر عدم ذخیره‌ی اشاره‌گرها در جدول باعث می‌شود که جدول درهم مکان‌های بیشتری برای ذخیره‌ی عناصر داشته باشد، که آن هم باعث می‌شود تعداد تصادم‌ها کم شده و زمان بازیابی کاهش یابد.

برای انجام عملیات درج با استفاده از آدرس‌دهی باز، به صورت پی‌درپی جدول درهم را آزمایش، یا *کاوش* (probe) می‌کنیم تا یک مکان خالی پیدا کنیم و کلید را در آن قرار دهیم. به جای دنباله‌ی از قبل تعیین شده‌ی $0, 1, ..., m-1$ (که به زمان $\theta(n)$ نیاز دارد)، دنباله‌ی مکان‌هایی که کاوش می‌شوند به *کلیدی که می‌خواهیم درج کنیم بستگی دارد*. برای تعیین این که کدام مکان‌ها باید کاوش شوند، تابع

درهم‌ساز را طوری اصلاح می‌کنیم که شامل عدد کاوش (با شروع از ۰) به عنوان ورودی دوم هم باشد. بنابراین تابع درهم‌ساز عبارت خواهد بود از

$$h : U \times \{0,1,\ldots,m-1\} \to \{0,1,\ldots,m-1\}$$

با آدرس‌دهی باز، برای هر کلید $k$ نیاز داریم که *دنباله‌ی کاوش*

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

یک جایگشت از $\langle 0,1,\ldots,m-1 \rangle$ باشد، تا حین پر شدن جدول، تمام مکان‌های جدول درهم درنهایت به عنوان یک مکان خالی برای یک کلید جدید در نظر گرفته شوند. در شبه‌کد زیر، فرض می‌کنیم عناصر در جدول $T$ کلیدهایی هستند بدون داده‌های پیرو؛ کلید $k$ دقیقاً برابر است با عنصری که حاوی کلید $k$ است. هر مکان یا حاوی یک کلید، و یا حاوی NIL است (اگر آن مکان خالی باشد). رویه‌ی HASH-INSERT به عنوان ورودی یک جدول درهم $T$ و یک کلید $k$ دریافت می‌کند. خروجی این رویه یا شماره‌ی مکان ذخیره‌ی کلید $k$ است و یا پیغام خطا در صورتی که جدول درهم پر باشد.

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == NIL
5          T[j] = k
6          return j
7      else i = i + 1
8  until i == m
9  error "hash table overflow"
```

الگوریتم جستجوی کلید $k$ همان دنباله‌ای را کاوش می‌کند که الگوریتم درج هنگام درج کلید $k$ آزمایش کرده بود. بنابراین، وقتی به یک مکان خالی برمی‌خوریم می‌توانیم به جستجو خاتمه دهیم (به صورت ناموفق)، چرا که اگر کلید $k$ در جدول درج شده بود، باید در همان مکان درج می‌شد و نه در مکان‌های بعدی. (در این بحث فرض می‌شود که کلیدها از جدول درهم حذف نمی‌شوند.) رویه‌ی HASH-SEARCH یک جدول درهم $T$ و یک کلید $k$ را به عنوان ورودی دریافت می‌کند، و در صورت یافتن کلید $k$ در مکان $j$، عدد $j$ را بازمی‌گرداند، و در غیر این صورت NIL را.

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

حذف از یک جدول درهم با آدرس‌دهی باز دشوار است. وقتی یک کلید را از مکان $i$ حذف می‌کنیم، نمی‌توانیم به سادگی NIL را در آن ذخیره کرده و آن را به صورت یک مکان خالی جدول در نظر بگیریم. چنین کاری باعث می‌شود نتوانیم کلیدهایی را جستجو کنیم که هنگام درج آن‌ها، مکان $i$ را کاوش کرده بودیم (در آن زمان مکان $i$ پر بوده است). یک راه حل این است که به جای NIL، چنین مکانی را با مقدار خاص DELETED علامت‌گذاری کنیم. سپس رویه‌ی HASH-INSERT را طوری اصلاح می‌کنیم که چنین مکانی را به عنوان یک مکان خالی در نظر بگیرد، و کلیدهای جدید را در آن درج کند. نیازی به اصلاح رویه‌ی HASH-INSERT نیست، چرا که این رویه هنگام جستجو از روی مقادیر DELETED عبور می‌کند. با این حال وقتی از مقدار خاص DELETED استفاده می‌کنیم، دیگر زمان جستجو به فاکتور بار $\alpha$ وابسته نیست، و به همین دلیل وقتی نیاز به حذف عناصر از جدول درهم داریم، معمولاً زنجیره‌ای سازی راه حل بهتری برای تصادم‌ها خواهد بود.

در تحلیل خود فرض درهم‌سازی یکنواخت را خواهیم داشت: فرض می‌کنیم که هر کلید با احتمال برابر هر یک از $m!$ جایگشت $\langle ۰,۱,...,m-۱\rangle$ را به عنوان دنباله‌ی کاوش دریافت خواهد کرد. درهم‌سازی یکنواخت، حالت کلی‌تر ایده‌ی درهم‌سازی ساده‌ی یکنواخت است، برای حالتی که توابع درهم‌ساز فقط یک عدد تولید نمی‌کنند، بلکه یک دنباله‌ی کاوش تولید می‌کنند. با این حال پیاده‌سازی درهم‌سازی یکنواخت کار دشواری است، و در عمل از تقریب‌های مناسب (مانند درهم‌سازی دوگانه، که در زیر تعریف شده است) استفاده می‌شود.

معمولاً از سه تکنیک برای محاسبه‌ی دنباله‌ی کاوش مورد نیاز برای آدرس‌دهی باز استفاده می‌شود: کاوش خطی، کاوش درجه دو، و درهم‌سازی دوگانه. تمام این تکنیک‌ها تضمین می‌کنند که برای هر کلید $k$ جایگشت $\langle h(k,۰),h(k,۱),...,h(k,m-۱)\rangle$ یک جایگشت از $\langle ۰,۱,...,m-۱\rangle$ است. با این حال هیچ کدام از این تکنیک‌ها فرض درهم‌سازی یکنواخت را ارضا نمی‌کنند، چرا که هیچ کدام از آن‌ها نمی‌توانند بیشتر از $m^۲$ دنباله‌ی کاوش مجزا تولید کنند (به جای $m!$ که درهم‌سازی یکنواخت به آن نیاز دارد). ولی درهم‌سازی دوگانه بیشترین تعداد دنباله‌های کاوش را دارد، و به نظر می‌آید که بهترین نتیجه را بدهد.

### کاوش خطی

با داشتن یک تابع درهم‌ساز معمولی $h':U\to\{۰,۱,...,m-۱\}$، که آن را یک تابع درهم‌ساز کمکی می‌نامیم، متد کاوش خطی (linear probing) از تابع درهم‌ساز

$$h(k,i)=(h'(k)+i)\bmod m$$

استفاده می‌کند، برای $i=۰,۱,...,m-۱$. با داشتن کلید $k$ اولین مکانی که کاوش می‌شود $T[h'(k)]$ است، یعنی مکانی که تابع درهم‌ساز کمکی تولید می‌کند. سپس مکان $T[h'(k)+۱]$ را کاوش می‌کنیم، و همین‌طور ادامه می‌دهیم تا مکان $T[m-۱]$. آن گاه به اول جدول بازگشته و مکان‌های $T[۰]$، $T[۱]$، ... را کاوش می‌کنیم تا به مکان $T[h'(k)-۱]$ برسیم. از آن جایی که مکان اولیه‌ی کاوش تمام دنباله‌ی کاوش را تعیین می‌کند، فقط $m$ دنباله‌ی کاوش مختلف وجود دارد.

پیاده‌سازی کاوش خطی ساده است، ولی این روش از مشکلی به نام دسته‌بندی اولیه (primary clustering) رنج می‌برد. دنباله‌های طولانی مکان‌های پر شده، که زمان متوسط جستجو را افزایش می‌دهد. دسته‌ها از این رو به وجود می‌آیند که یک مکان خالی که قبل از آن $i$ مکان پر وجود دارد، با احتمال $(i+1)/m$ در مرحله‌ی بعد پر می‌شود. این دنباله‌های طولانی از مکان‌های پر شده، همین طور طولانی‌تر می‌شوند، و زمان متوسط جستجو افزایش می‌یابد.

## کاوش درجه دو

کاوش درجه دو (quadric probing) از یک تابع درهم‌ساز به صورت

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \qquad (۵-۱۱)$$

استفاده می‌کند، که در آن $h'$ یک تابع درهم‌ساز کمکی است، $c_1$ و $c_2 \neq 0$ ثابت‌های کمکی هستند، و $i = 0,1,\ldots,m-1$. اولین مکانی که کاوش می‌شود $T[h'(k)]$ است؛ مکان‌های بعدی چندین مکان جلوتر از مکان اولیه هستند، که این تعداد به رفتار درجه دوی عدد کاوش $i$ بستگی دارد. این روش بسیار بهتر از کاوش خطی کار می‌کند، ولی برای استفاده از تمام جدول درهم، مقادیر $c_1$، $c_2$، و $m$ باید محدود شوند. مسئله‌ی ۱۱-۳ یک روش برای انتخاب مقدار این پارامترها نشان می‌دهد. همچنین اگر دو کلید مکان کاوش اولیه‌ی یکسانی داشته باشند، دنباله‌ی کاوش آن‌ها یکسان خواهد بود، چرا که $h(k_1,0) = h(k_2,0)$ نتیجه می‌دهد $h(k_1,i) = h(k_2,i)$. این خصوصیت باعث ایجاد یک دسته بندی خفیف می‌شود، با نام دسته بندی ثانویه. مانند کاوش خطی، مکان اولیه‌ی کاوش تمام دنباله را تعیین می‌کند، و بنابراین فقط $m$ دنباله‌ی کاوش متفاوت وجود دارد.

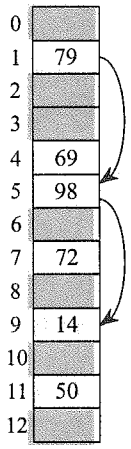
## درهم‌سازی دوگانه

درهم‌سازی دوگانه یکی از بهترین متدهای موجود برای آدرس‌دهی باز است، چرا که در آن جایگشت‌های تولید شده بسیاری از خصوصیت‌های جایگشت‌های تصادفی را دارند. درهم‌سازی دوگانه از یک تابع درهم‌ساز به صورت

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m$$

استفاده می‌کند، که در آن $h_1$ و $h_2$ توابع درهم‌ساز کمکی هستند. مکان اولیه‌ی کاوش $T[h_1(k)]$ است؛ مکان‌های بعدی به اندازه‌ی $h_2(k) \bmod m$ جلوتر از مکان قبلی هستند. بنابراین برخلاف حالت کاوش خطی یا درجه دو، در این جا دنباله‌ی کاوش از دو جهت به کلید $k$ بستگی دارد، چرا که ممکن است مکان اولیه‌ی کاوش و یا تعداد جابه‌جایی و یا هر دوی آن‌ها تغییر کنند. شکل ۱۱-۵ مثالی از درج با استفاده از درهم‌سازی دوگانه را نشان می‌دهد.

مقدار $h_2(k)$ باید نسبت به اندازه‌ی جدول درهم، اول باشد تا تمام جدول درهم جستجو شود. (تمرین ۱۱-۴-۳ را ببینید.) یک روش مناسب برای اطمینان از این مسئله این است که $m$ را توانی از ۲ در نظر بگیریم، و $h_2$ را طوری طراحی کنیم که همیشه یک عدد فرد تولید کند. روش دیگر این است




**شکل ۱۱-۵** درج به وسیله‌ی درهم‌سازی دوگانه. در این جا یک جدول درهم با اندازه‌ی ۱۳ داریم، با $h_1(k) = k \bmod ۱۳$ و $h_2(k) = ۱ + (k \bmod ۱۱)$. از آن جایی که $۱۴ \equiv ۱ \pmod{۱۳}$ و $۱۴ \equiv ۳ \pmod{۱۱}$، کلید ۱۴، بعد از این که مکان‌های پر ۱ و ۵ آزمایش شدند، در مکان خالی ۹ درج می‌شود.

که $m$ اول باشد و $h_2$ طوری طراحی شود که همیشه یک عدد مثبت کوچک‌تر از $m$ بازگرداند. مثلاً، می‌توانیم $m$ را اول انتخاب کنیم و

$$h_1(k) = k \bmod m$$
$$h_2(k) = ۱ + (k \bmod m')$$

که در آن $m'$ طوری انتخاب شده است که مقدار آن کمی از $m$ کوچک‌تر باشد (مثلاً $m - ۱$). به عنوان مثال، اگر $k = ۱۲۳۴۵۶$، $m = ۷۰۱$، و $m' = ۷۰۰$، داریم $h_1(k) = ۸۰$ و $h_2(k) = ۲۵۷$، و مکان اول کاوش مکان ۸۰ است، و بعد از آن هر ۲۵۷ امین مکان (به پیمانه‌ی $m$) کاوش می‌شود تا یا کلید یافت شود و یا تمام مکان‌ها بررسی شوند.

وقتی $m$ فرد یا توانی از ۲ باشد، مزیت درهم‌سازی دوگانه نسبت به کاوش خطی و یا درجه دو این است که به جای $\theta(m)$ دنباله، از $\theta(m^2)$ دنباله‌ی کاوش استفاده می‌شود، چرا که هر جفت ممکن $(h_1(k), h_2(k))$ یک دنباله‌ی کاوش متفاوت را نتیجه می‌دهد. در نتیجه کارایی درهم‌سازی دوگانه بسیار نزدیک به کارایی طرح درهم‌سازی یکنواخت «ایده‌آل» است.

## تحلیل درهم‌سازی آدرس باز

تحلیل ما از آدرس‌دهی باز، مانند تحلیل زنجیره‌ای سازی بر مبنای فاکتور بار جدول درهم ($\alpha = n/m$) است. مسلماً در آدرس‌دهی باز حداکثر یک عنصردر هر مکان داریم، و بنابراین $n \le m$، که نتیجه می‌دهد $\alpha \le ۱$.

فرض می‌کنیم از درهم‌سازی متوازن استفاده شده است. در این طرح ایده‌آل، دنباله‌ی کاوش $\langle h(k,۰), h(k,۱), \ldots, h(k,m-۱) \rangle$ استفاده شده برای جستجوی هر کلید $k$، با احتمال برابر هر یک از

جایگشت‌های $\langle ۰,۱,...,m-۱\rangle$ خواهد بود. مسلماً هر کلید یک دنباله‌ی کاوش ثابت تعیین شده خواهد داشت؛ منظور این است که با توجه به توزیع احتمالاتی بر روی فضای کلیدها و عملیات تابع درهم‌ساز بر روی آن‌ها، احتمال وقوع هر یک از دنباله‌های کاوش برابر است.

اکنون امیدریاضی تعداد کاوش‌ها را برای آدرس‌دهی باز با فرض درهم‌سازی متوازن تحلیل می‌کنیم. با تحلیل تعداد کاوش‌ها در یک جستجوی ناموفق آغاز می‌کنیم.

**قضیه‌ی ۱۱-۶** با داشتن یک جدول درهم آدرس‌باز با فاکتور بار $\alpha = n/m < ۱$، امیدریاضی تعداد کاوش‌ها در یک جستجوی ناموفق، با فرض درهم‌سازی متوازن حداکثر برابر است با $۱/(۱-\alpha)$.

*اثبات* در یک جستجوی ناموفق، تمام کاوش‌ها غیر از آخرین کاوش به یک مکان پر که حاوی کلید مورد نظر نیست دسترسی پیدا می‌کنند، و آخرین مکان کاوش شده خالی است. فرض کنید متغیر تصادفی $X$ تعداد کاوش‌های انجام شده در یک جستجوی ناموفق باشد، همچنین برای $i = ۱,۲,...$ پیشامد $A_i$ نشان دهنده‌ی انجام $i$امین کاوش باشد که مکان مربوط به آن کاوش پر است. در این صورت پیشامد $\{X \geq i\}$ برابر است با اشتراک پیشامدهای $A_۱ \cap A_۲ \cap ... \cap A_{i-۱}$. برای تعیین کران $\Pr\{X \geq i\}$، کران $\Pr\{A_۱ \cap A_۲ \cap ... \cap A_{i-۱}\}$ را تعیین می‌کنیم. طبق تمرین پ-۲-۵،

$$\Pr\{A_۱ \cap A_۲ \cap ... \cap A_{i-۱}\} = \Pr\{A_۱\} \cdot \Pr\{A_۲ \mid A_۱\} \cdot \Pr\{A_۳ \mid A_۱ \cap A_۲\} \cdot \\ \Pr\{A_{i-۱} \mid A_۱ \cap A_۲ \cap A_{i-۲}\}$$

از آن جایی که $n$ عنصر و $m$ مکان وجود دارد، $\Pr\{A_۱\} = n/m$. برای $j > ۱$، احتمال این که $j$امین کاوش بر روی یک مکان پر انجام شود، با فرض این که $j-۱$ کاوش قبل بر روی مکان‌های پر انجام شده باشند، برابر است با $(n-j+۱)/(m-j+۱)$. این احتمال از این جا ناشی می‌شود که یکی از $(n-(j-۱))$ عنصر باقی مانده را در یکی از $(m-(j-۱))$ مکان کاوش نشده خواهیم یافت، و با فرض درهم‌سازی متوازن، این احتمال برابر است با نسبت این دو عبارت. با توجه به این که برای $۰ \leq j < m$، نامساوی $n < m$ نتیجه می‌دهد $(n-j)/(m-j) \leq n/m$، برای تمام $i$هایی که $۱ \leq i \leq m$ داریم:

$$\begin{aligned}\Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-۱}{m-۱} \cdot \frac{n-۱}{m-۲} \cdots \frac{n-i+۲}{m-i+۲} \\ &\leq \left(\frac{n}{m}\right)^{i-۱} \\ &= \alpha^{i-۱}\end{aligned}$$

اکنون از تساوی (پ-۲۵) برای تعیین کران امیدریاضی تعداد کاوش‌ها استفاده می‌کنیم:

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\
&\leq \sum_{i=0}^{\infty} \alpha^{i-1} \\
&= \sum_{i=0}^{\infty} \alpha^{i} \\
&= \frac{1}{1-\alpha}
\end{aligned}
$$

کران $1/(1+\alpha) = 1+\alpha+\alpha^2+\alpha^3+\cdots$ یک تفسیر بدیهی دارد: همیشه حداقل یک کاوش انجام می‌شود. با احتمال تقریباً $\alpha$، مکان بررسی شده در اولین کاوش پر خواهد بود، و باید کاوش دوم را انجام دهیم. با احتمال تقریباً $\alpha^2$، مکان‌های بررسی شده در دو کاوش اول پر خواهند بود، و باید کاوش سوم را انجام دهیم، و الی آخر.

اگر $\alpha$ ثابت باشد قضیه‌ی ۱۱-۶ پیش‌بینی می‌کند که یک جستجوی ناموفق در زمان $O(1)$ انجام می‌شود. به عنوان مثال اگر جدول درهم نیمه پر باشد، متوسط تعداد کاوش‌ها در یک جستجوی ناموفق حداکثر $1/(1-0/5) = 2$ خواهد بود، و اگر جدول ۹۰ درصد پر باشد، متوسط تعداد کاوش‌ها $1/(1-0/9) = 10$ خواهد بود.

از روی قضیه‌ی ۱۱-۶ می‌توان به راحتی کارایی رویه‌ی HASH-INSERT را تعیین کرد.

**نتیجه‌ی ۱۱-۷** درج یک عنصر در یک جدول درهم آدرس‌باز با فاکتور بار $\alpha$، با فرض درهم‌سازی متوازن، در حالت متوسط حداکثر به $1/(1-\alpha)$ کاوش نیاز دارد.

*اثبات* یک عنصر در صورتی می‌تواند درج شود که در جدول جای خالی وجود داشته باشد، و بنابراین $\alpha < 1$. برای درج یک کلید، ابتدا باید یک جستجوی ناموفق انجام شود، و به دنبال آن قرار دادن کلید در اولین مکان خالی پیدا شده. بنابراین امیدریاضی تعداد کاوش‌ها برای درج حداکثر $1/(1-\alpha)$ است.

تعیین امیدریاضی تعداد کاوش‌ها در یک جستجوی موفق نیاز به محاسبات بیشتری دارد.

**قضیه‌ی ۱۱-۸** با داشتن یک جدول درهم آدرس‌باز با فاکتور بار $\alpha < 1$، و با فرض درهم‌سازی متوازن امیدریاضی تعداد کاوش‌ها در یک جستجوی موفق حداکثر برابر است با

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

با فرض این که جستجو با احتمال برابر برای هر یک از کلیدهای درون جدول انجام می‌شود.

*اثبات* در جستجو برای کلید $k$، از همان دنباله‌ی کاوشی استفاده می‌شود که در درج کلید $k$ استفاده شده بود. طبق نتیجه‌ی ۱۱-۷، اگر $k$، $(i+1)$امین کلیدی باشد که در جدول درج شده است، امیدریاضی تعداد کاوش‌های انجام شده در جستجوی $k$ حداکثر برابر است با $1/(1-i/m) = m/(m-i)$. میانگین گرفتن روی تمام $n$ کلید درون جدول درهم، متوسط تعداد کاوش‌ها را در یک جستجوی موفق به ما می‌دهد:

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m} 1/k \\
&\le \frac{1}{\alpha}\int_{m-n}^{m}(1/x)\,dx \qquad \text{(طبق تساوی (الف-۱۲))} \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}
\end{aligned}$$

که عبارت آخر کرانی برای امیدریاضی تعداد کاوش‌ها در یک جستجوی موفق است. ■

اگر جدول درهم نیمه پر باشد، امیدریاضی تعداد کاوش‌ها در یک جستجوی موفق کم‌تر از ۱٫۳۸۷ است، و اگر جدول ۹۰ درصد پر باشد، امیدریاضی تعداد کاوش‌ها در یک جستجوی موفق کم‌تر از ۲٫۵۵۹ خواهد بود.

## تمرین‌ها

۱-۴-۱۱ درج کلیدهای ۱۰، ۲۲، ۳۱، ۴، ۱۵، ۲۸، ۱۷ و ۵۹ را در یک جدول درهم با طول $m = 11$، با استفاده از آدرس‌دهی باز با تابع درهم‌ساز اصلی $h'(k) = k$ در نظر بگیرید. نتیجه‌ی این درج‌ها را در سه حالت استفاده از کاوش خطی، استفاده از کاوش درجه دو با $c_1 = 1$ و $c_2 = 3$، و استفاده از درهم‌سازی دوگانه با $h_2(k) = 1 + (k \bmod (m-1))$ تعیین کنید.

۲-۴-۱۱ یک شبه‌کد برای HASH-DELETE همان طور که در متن مشخص شده است بنویسید، و HASH-INSERT را طوری اصلاح کنید که مقدار خاص DELETED را هم در نظر بگیرد.

۳-۴-۱۱ یک جدول درهم آدرس‌باز با درهم‌سازی متوازن را در نظر بگیرید. یک کران بالا برای تعداد کاوش‌ها در یک جستجوی ناموفق و همچنین در یک جستجوی موفق بدهید، در حالت‌هایی که فاکتور بار ۳/۴ و ۷/۸ باشد.

★ ۴-۴-۱۱ فرض کنید که از درهم‌سازی دوگانه برای حل تصادم‌ها استفاده می‌کنیم؛ یعنی از تابع

درهم‌ساز $h(k,i)=(h_1(k)+ih_2(k)) \bmod m$ استفاده می‌کنیم. نشان دهید که اگر بزرگ‌ترین مقسوم‌علیه مشترک $m$ و $h_2(k)$ برای کلید $k$ برابر ۱ $\geq d$ باشد، آن گاه در یک جستجوی ناموفق برای کلید $k$ تعداد $(1/d)$ از مکان‌های جدول قبل از بازگشت به مکان $h_1(k)$ بررسی می‌شود. بنابراین وقتی $d=1$، یعنی وقتی که $m$ و $h_2(k)$ نسبت به هم اول هستند، ممکن است در جستجو تمام عناصر جدول بررسی شوند. (*راهنمایی:* فصل ۳۱ را ببینید.)

★ ۱۱-۳-۵ یک جدول درهم آدرس‌باز با فاکتور بار $\alpha$ را در نظر بگیرید. یک مقدار غیر صفر برای $\alpha$ پیدا کنید که برای آن امیدریاضی تعداد کاوش‌ها در یک جستجوی ناموفق، دو برابر تعداد کاوش‌ها در یک جستجوی موفق باشد. از کران‌های بالای داده شده در قضیه‌های ۱۱-۶ و ۱۱-۸ برای امیدریاضی این کاوش‌ها استفاده کنید.

## ۱۱-۵ درهم‌سازی کامل

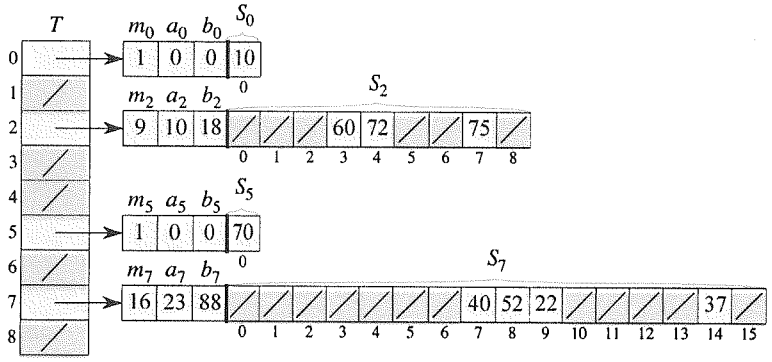
با این که معمولاً از درهم‌سازی برای دستیابی به یک کارایی بالا در حالت متوسط استفاده می‌شود، اگر مجموعه‌ی کلیدها ایستا (static) باشد، می‌توان از آن برای دستیابی به یک کارایی بالا در *بدترین حالت* استفاده کرد: وقتی کلیدها در جدول ذخیره شدند، دیگر تغییر نمی‌کنند. مجموعه‌ی کلیدها در بعضی از کاربردها به صورت طبیعی ایستا است: مجموعه‌ی کلمات رزرو شده در یک زبان برنامه‌نویسی، و یا مجموعه‌ی نام فایل‌ها در یک CD-ROM. به یک تکنیک درهم‌سازی، *درهم‌سازی کامل* می‌گوییم اگر بدترین حالت تعداد دسترسی به حافظه برای انجام یک جستجو $O(1)$ باشد.

ایده‌ی اصلی طراحی یک درهم‌سازی کامل ساده است. از یک طرح درهم‌سازی دو سطحی با درهم‌سازی جهانی در هر سطح استفاده می‌کنیم. شکل ۱۱-۶ این رویکرد را نشان می‌دهد.

مرحله‌ی اول به طور کلی مشابه درهم‌سازی زنجیره‌ای است: $n$ کلید به $m$ مکان با استفاده از تابع درهم‌ساز $h$ نگاشت می‌شوند، که این تابع با دقت از میان خانواده‌ای از توابع درهم‌ساز جهانی انتخاب شده است.

با این حال به جای ساختن یک لیست از کلیدهایی که به مکان $j$ نگاشت می‌شوند، از یک *جدول درهم دوم* کوچک $S_j$ با تابع درهم‌ساز استفاده می‌کنیم. با انتخاب مناسب تابع درهم‌ساز $h_j$، می‌توانیم تضمین کنیم که هیچ تصادمی در سطح دوم اتفاق نمی‌افتد.

با این حال برای این که بتوانیم تضمین کنیم هیچ تصادمی در سطح دوم اتفاق نمی‌افتد، باید $m_j$ (اندازه‌ی جدول درهم $S_j$) مربع $n_j$ (تعداد کلیدهای درهم‌سازی شده به مکان $j$) باشد. با این که به نظر می‌رسد این وابستگی درجه دوی $m_j$ به $n_j$ باعث شود که حافظه‌ی مصرفی به شدت بالا رود، نشان خواهیم داد که با انتخاب مناسب تابع درهم‌ساز سطح اول، امیدریاضی کل حافظه‌ی مصرفی همچنان $O(n)$ خواهد بود.




**شکل ۱۱-۶** استفاده از درهم‌سازی کامل برای ذخیره‌ی مجموعه‌ی $K = \{۱۰,۲۲,۳۷,۴۰,۶۰,۷۰,۷۵\}$. تابع درهم‌ساز خارجی $h(k) = ((ak + b) \bmod p) \bmod m$ است، که در آن $a = ۳$، $b = ۴۲$، $p = ۱۰۱$، و $m = ۹$. برای مثال $h(۷۵) = ۲$، پس کلید ۷۵ به مکان ۲ از جدول $T$ نگاشت می‌شود. جدول درهم دوم، $S_j$، تمام کلیدهای نگاشت شده به مکان $j$ را ذخیره می‌کند. اندازه‌ی جدول درهم $S_j$ برابر $m_j$، و تابع درهم‌ساز مربوطه $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$ است. از آن جایی که $h_۲(۷۵) = ۱$، کلید ۷۵ در مکان ۱ از جدول درهم دوم ذخیره می‌شود. هیچ تصادمی در هیچ یک از جداول درهم دوم وجود ندارد، و بنابراین جستجو در بدترین حالت در زمان ثابت انجام می‌شود.

توابع درهم‌ساز را از مجموعه‌ی توابع جهانی بخش ۱۱-۳-۳ انتخاب می‌کنیم. تابع درهم‌ساز سطح اول از مجموعه‌ی $H_{p,m}$ انتخاب می‌شود، که مانند بخش ۱۱-۳-۳، $p$ یک عدد اول بزرگتر از مقدار تمام کلیدها است. کلیدهای نگاشت شده به مکان $j$، دوباره با استفاده از تابع $h_j$ که از مجموعه‌ی $H_{p,m}$ انتخاب شده است، به مکانی در یک جدول درهم دوم $S_j$ با اندازه‌ی $m_j$ نگاشت می‌شوند.[۱]

کار خود را در دو مرحله ادامه می‌دهیم. ابتدا مشخص می‌کنیم که چطور می‌توان اطمینان حاصل کرد که در جداول درهم دوم تصادم پیش نمی‌آید. سپس نشان خواهیم داد که امیدریاضی حافظه‌ی مصرف شده — برای جدول درهم اول و تمام جداول درهم دوم — $O(n)$ است.

**قضیه‌ی ۱۱-۹** اگر $n$ کلید را در یک جدول درهم با اندازه‌ی $m = n^۲$ و با استفاده از یک تابع درهم‌ساز که به صورت تصادفی از یک مجموعه‌ی جهانی از توابع درهم‌ساز انتخاب شده است، ذخیره کنیم، احتمال این که حداقل یک تصادم وجود داشته باشد کم‌تر از ۱/۲ است.

*اثبات* $\binom{n}{۲}$ جفت کلید وجود دارد که ممکن است بین آن‌ها تصادم پیش آید؛ اگر تابع درهم‌ساز $h$ از مجموعه‌ی $H_{p,m}$ به صورت تصادفی انتخاب شود، با احتمال $۱/m$ بین هر یک از این جفت‌ها تصادم پیش می‌آید. فرض کنید $X$ یک متغیر تصادفی باشد که تعداد تصادم‌ها را می‌شمارد. وقتی $m = n^۲$، امیدریاضی تعداد تصادم‌ها برابر است با

[۱] وقتی $n_j = m_j = ۱$، در واقع نیازی به یک تابع درهم‌ساز برای مکان $j$ نداریم؛ وقتی از یک تابع درهم‌ساز $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m_j$ برای چنین مکانی استفاده می‌کنیم، به سادگی انتخاب می‌کنیم $a = b = ۰$.

$$E[X] = \binom{n}{2} \cdot \frac{1}{n^2}$$
$$= \frac{n^2 - n}{2} \cdot \frac{1}{n^2}$$
$$< 1/2$$

(دقت داشته باشید که این تحلیل مانند تحلیل پارادوکس تولد در بخش ۵-۴-۱ است.) با استفاده از نامساوی مارکوف (Markov's inequality) (پ-۳۰) داریم $\Pr\{X \geq r\} \leq E[X]/t$، که با $t = 1$ اثبات را کامل می‌کند. ■

در حالت توصیف شده در قضیه‌ی ۱۱-۹، که در آن $m = n^2$، با یک تابع درهم‌ساز که به صورت تصادفی از $H_{p,m}$ انتخاب شده باشد، احتمال این که هیچ تصادمی نداشته باشیم بیشتر از آن است که تصادم داشته باشیم. بنابراین با داشتن مجموعه‌ی $K$ که $n$ کلیدی که نگاشت می‌شوند عضو آن هستند (به یاد بیاورید که $K$ ایستا است)، با چند بار سعی کردن می‌توان یک تابع درهم‌ساز بدون تصادم پیدا کرد.

با این حال وقتی $n$ بزرگ باشد، داشتن یک جدول درهم با اندازه‌ی $m = n^2$ امکان‌پذیر نیست. بنابراین طرح درهم‌سازی دو سطحی را اجرا کرده، و از رویکرد قضیه‌ی ۱۱-۹ فقط برای نگاشت ورودی‌های درون هر یک از مکان‌ها استفاده می‌کنیم. از یک تابع درهم‌ساز خارجی (یا اولیه) برای نگاشت کلیدها به $m = n$ مکان استفاده می‌کنیم. سپس اگر $n_j$ کلید به مکان $j$ نگاشت شده باشند، از یک جدول درهم دوم $S_j$ با اندازه‌ی $m_j = n^2$ برای نگاشت بدون تصادم با جستجوی با زمان ثابت استفاده می‌کنیم.

اکنون به مسئله‌ی حافظه‌ی $O(n)$ بازمی‌گردیم. از آن جایی که $m_j$، اندازه‌ی $j$ امین جدول درهم دوم، نسبت به تعداد کلیدهای ذخیره شده از درجه‌ی دو رشد می‌کند، یک ریسک وجود دارد که کل حافظه‌ی مصرفی هزینه‌ی بسیار بالایی داشته باشد.

اگر اندازه‌ی جدول اول $m = n$ باشد، آن گاه حافظه‌ی مصرفی برای جدول درهم اول، برای حافظه‌های با اندازه‌ی $m_j$ در جدول‌های درهم دوم، و برای حافظه‌ی پارامترهای $a_j$ و $b_j$ که توابع درهم‌ساز دوم انتخاب شده از $H_{p,m}$ را تعریف می‌کنند (بخش ۱۱-۳-۳)، همگی $O(n)$ است (به جز حالتی که $n_j = 1$ که از $a = b = 0$ استفاده می‌کنیم.). به کمک قضیه‌ی زیر و یک نتیجه گیری، کرانی برای امیدریاضی اندازه‌ی تمام جداول درهم دوم تعیین می‌کنیم. سپس در نتیجه‌ی دوم، کرانی برای احتمال این که مجموع اندازه‌های تمام جداول درهم دوم بیشتر از خطی باشد تعیین می‌کنیم.

**قضیه‌ی ۱۱-۱۰**

اگر با استفاده از یک تابع درهم‌ساز $h$ که به صورت تصادفی از یک مجموعه‌ی جهانی از توابع درهم‌ساز انتخاب شده است، $n$ کلید را در یک جدول درهم با اندازه‌ی $m = n$ ذخیره کنیم، آن گاه

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$$

که در آن $n_j$ تعداد کلیدهای نگاشت شده به مکان $j$ است.

*اثبات* با اتحاد زیر شروع می‌کنیم، که برای تمام اعداد صحیح غیر منفی $a$ برقرار است:

$$a^۲ = a + ۲\binom{a}{۲} \qquad (۱۱-۶)$$

داریم

$$\begin{aligned}
&E\left[\sum_{j=۰}^{m-۱} n_j^۲\right] \\
&\quad = E\left[\sum_{j=۰}^{m-۱} \left(n_j + ۲\binom{n_j}{۲}\right)\right] && \text{(طبق تساوی (۱۱-۶))} \\
&\quad = E\left[\sum_{j=۰}^{m-۱} n_j\right] + ۲E\left[\sum_{j=۰}^{m-۱}\binom{n_j}{۲}\right] && \text{(طبق خطی بودن امیدریاضی)} \\
&\quad = E[n] + ۲E\left[\sum_{j=۰}^{m-۱}\binom{n_j}{۲}\right] && \text{(طبق تساوی (۱۱-۱))} \\
&\quad = n + ۲E\left[\sum_{j=۰}^{m-۱}\binom{n_j}{۲}\right] && \text{(چرا که } n \text{ یک متغیر تصادفی نیست)}
\end{aligned}$$

برای ارزیابی سری $\sum_{j=۰}^{m-۱}\binom{n_j}{۲}$، مشاهده می‌کنیم که این سری به سادگی برابر است با کل تعداد تصادم‌ها. طبق خصوصیات درهم‌سازی جهانی، امیدریاضی مقدار این سری حداکثر برابر است با

$$\binom{n}{۲}\frac{۱}{m} = \frac{n(n-۱)}{۲m} = \frac{n-۱}{۲}$$

چرا که $m = n$. بنابراین،

$$\begin{aligned}
۲E\left[\sum_{j=۰}^{m-۱} n_j^۲\right] &\le n + ۲\frac{n-۱}{۲} \\
&= ۲n - ۱ \\
&< ۲n
\end{aligned}$$

**نتیجه‌ی ۱۱-۱۱** اگر با استفاده از یک تابع درهم‌ساز $h$ که به صورت تصادفی از یک مجموعه جهانی از توابع درهم‌ساز انتخاب شده است، $n$ کلید را به یک جدول درهم با اندازه‌ی $m = n$ نگاشت کنیم، و برای $j = ۰,۱,۲,\ldots,m-۱$، اندازه‌ی هر یک از جداول درهم دوم را $m_j = n_j^۲$ قرار دهیم، آن گاه امیدریاضی مقدار حافظه‌ی مورد نیاز برای تمام جداول درهم دوم در یک درهم‌سازی کامل کم‌تر از $۲n$ است.

*اثبات* از آن جایی که برای $j = ۰,۱,۲,...,m-۱$ داریم $m_j = n_j^۲$، قضیه‌ی ۱۱-۱۰ می‌دهد

$$E\left[\sum_{j=۰}^{m-۱} m_j\right] = E\left[\sum_{j=۰}^{m-۱} n_j^۲\right] < ۲n \qquad (۷-۱۱)$$

که اثبات را کامل می‌کند.

**نتیجه‌ی ۱۲-۱۱** اگر با استفاده از یک تابع درهم‌ساز $h$ که به صورت تصادفی از یک مجموعه جهانی از توابع درهم‌ساز انتخاب شده است، $n$ کلید را به یک جدول درهم با اندازه‌ی $m = n$ نگاشت کنیم، و برای $j = ۰,۱,۲,...,m-۱$، اندازه‌ی هر یک از جداول درهم دوم را $m_j = n_j^۲$ قرار دهیم، آن گاه احتمال این که اندازه‌ی کل جداول درهم دوم از $۴n$ فراتر رود، ۱/۲ است.

*اثبات* دوباره از نامساوی مارکوف (پ-۲۹)، $\Pr\{X \geq r\} \leq E[x]/t$، استفاده می‌کنیم، ولی این بار بر روی نامساوی (۷-۱۱)، با $X = \sum_{j=۰}^{m-۱} m_j$ و $t = ۴n$:

$$\begin{aligned}\Pr\left\{\sum_{j=۰}^{m-۱} m_j \geq ۴n\right\} &\leq \frac{E[\sum_{j=۰}^{m-۱} m_j}{۴n} \\ &< \frac{۲n}{۴n} \\ &= ۱/۲\end{aligned}$$

طبق نتیجه‌ی ۱۱-۱۲، می‌بینیم که با آزمایش چند تابع درهم‌ساز که به صورت تصادفی از یک مجموعه‌ی جهانی از توابع درهم‌ساز انتخاب شده‌اند، می‌توان به سرعت تابعی پیدا کرد که حافظه‌ی مصرفی آن معقول است.

## تمرین‌ها

★ ۱-۵-۱۱ فرض کنید با استفاده از آدرس‌دهی باز و درهم‌سازی متوازن، $n$ کلید را به یک جدول درهم با اندازه‌ی $m$ نگاشت می‌کنیم. همچنین فرض کنید $p(n,m)$ احتمال این باشد که هیچ تصادمی رخ ندهد. نشان دهید که $p(n,m) \leq e^{-n(n-۱)/۲m}$. (*راهنمایی:* تساوی (۳-۱۱) را ببینید.) بحث کنید که وقتی $n$ از $\sqrt{m}$ فراتر می‌رود، احتمال جلوگیری از تصادم به سرعت به صفر میل می‌کند.

## مسائل

### ۱۱-۱ کران بلندترین کاوش برای درهم‌سازی

از یک جدول درهم آدرس‌باز با اندازه‌ی $m$ برای ذخیره‌ی $n < m/2$ عنصر استفاده شده است.

I. با فرض درهم‌سازی متوازن، نشان دهید که برای $i = 1,2,...,n$، احتمال این که $i$امین درج به بیش از $k$ کاوش نیاز داشته باشد حداکثر $2^{-k}$ است.

II. نشان دهید که برای $i = 1,2,...,n$، احتمال این که $i$امین درج به بیش از $2\lg n$ کاوش نیاز داشته باشد $O(1/n^2)$ است.

فرض کنید متغیر تصادفی $X_i$ نشان‌دهنده‌ی تعداد کاوش‌های مورد نیاز برای $i$امین درج باشد. در قسمت II نشان دادید که $\Pr\{X_i > 2\lg n\} \le 1/n^2$. فرض کنید متغیر تصادفی $X = \max_{1 \le j \le n} X_i$ نشان‌دهنده‌ی بیشینه‌ی تعداد کاوش‌های مورد نیاز برای هر یک از $n$ درج باشد.

III. نشان دهید که $\Pr\{X > 2\lg n\} = O(1/n)$.

IV. نشان دهید که $E[X]$، امیدریاضی طول بلندترین کاوش، $O(\lg n)$ است.

### ۱۱-۲ کران اندازه‌ی هر مکان برای زنجیره‌ای سازی

فرض کنید یک جدول درهم با $n$ مکان داریم، که در آن تصادم‌ها به کمک زنجیره‌ای سازی رفع شده‌اند، و فرض کنید $n$ کلید به این جدول نگاشت شده‌اند. هر کلید با احتمال برابر به هر یک از مکان‌ها نگاشت می‌شود. فرض کنید بعد از این که تمام درج‌ها انجام شد، $M$ بیشینه‌ی تعداد کلیدها در هر یک از مکان‌ها باشد. وظیفه‌ی شما این است که کران بالای $O(\lg n/\lg\lg n)$ را برای $E[M]$، امیدریاضی $M$، اثبات کنید.

I. بحث کنید که احتمال $Q_k$ که دقیقاً $k$ کلید به یک مکان خاص نگاشت شوند برابر است با

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}$$

II. فرض کنید $P_k$ احتمال این باشد که $M = k$، یعنی این که مکانی که حاوی بیشترین کلیدها است، $k$ کلید داشته باشد. نشان دهید که $P_k \le nQ_k$.

III. با استفاده از تقریب استرلینگ (تساوی (۳-۱۷)) نشان دهید که $Q_k < e^k/k^k$.

IV. نشان دهید که یک ثابت $c > 1$ وجود دارد که برای $k_0 = c\lg n/\lg\lg n$ داشته باشیم $Q_{k_0} < 1/n^3$. نتیجه بگیرید که برای $k \ge k_0 = c\lg n/\lg\lg n$ داریم $P_k < 1/n^2$.

V. بحث کنید که

$$E[M] \le \Pr\left\{M > \frac{c\lg n}{\lg\lg n}\right\} \cdot n + \Pr\left\{M \le \frac{c\lg n}{\lg\lg n}\right\} \cdot \frac{c\lg n}{\lg\lg n}$$

نتیجه بگیرید که $E[M]=O(\lg n/\lg\lg n)$.

## ۱۱-۳ کاوش درجه دو

فرض کنید می‌خواهیم در یک جدول درهم با مکان‌های $0,1,...,m-1$، برای کلید $k$ جستجو کنیم، و فرض کنید یک تابع درهم‌ساز داریم که کلیدها را به مجموعه‌ی $\{0,1,...,m-1\}$ نگاشت می‌کند. روش جستجو به صورت زیر است.

۱. محاسبه‌ی مقدار $i=h(k)$، و مقداردهی $j=0$.

۲. کاوش در مکان $i$ برای کلید $k$. پایان جستجو اگر کلید مورد نظر یافت شد، و یا اگر مکان کاوش شده خالی بود.

۳. قرار دادن $i=i+1$. اگر اکنون $i$ برابر با $m$ باشد، جدول پر است، پس جستجو باید پایان یابد. در غیر این صورت انجام $j=(i+j) \bmod m$ و بازگشت به مرحله‌ی ۲.

فرض کنید $m$ توانی از ۲ است.

I. با ارائه‌ی ثابت‌های مناسب $c_1$ و $c_2$ برای تساوی (۱۱-۵)، نشان دهید که این رویکرد حالتی از رویکرد کلی «کاوش درجه دو» است.

II. اثبات کنید که در این الگوریتم در بدترین حالت تمام مکان‌های جدول بررسی می‌شوند.

## ۱۱-۴ درهم‌سازی و تعیین هویت (authentication)

فرض کنید $H=\{h\}$ یک مجموعه از توابع درهم‌ساز باشد که در آن هر $h$ مجموعه‌ی جهانی کلیدها $(U)$ را به $\{0,1,...,m-1\}$ نگاشت می‌کند. می‌گوییم H، k-جهانی است اگر برای هر دنباله‌ی ثابت از $k$ کلید متفاوت $\langle x^{(1)},x^{(2)},...,x^{(k)}\rangle$، و برای هر $h$ که به صورت تصادفی از H انتخاب شده است، دنباله‌ی $\langle h(x^{(1)}),h(x^{(2)}),...,h(x^{(k)})\rangle$ با احتمال برابر هر یک از $m^k$ دنباله‌ی با طول $k$ از عناصر مجموعه‌ی $\{0,1,...,m-1\}$ باشد.

I. نشان دهید که اگر H، ۲-جهانی باشد، جهانی هم هست.

II. فرض کنید $U$ مجموعه‌ی تمام $n$ تایی‌های $Z_p=\{0,1,...,p-1\}$ باشد، که $p$ یک عدد اول است. یک عنصر $x=\langle x_0,x_1,...,x_{n-1}\rangle\in U$ را در نظر بگیرید. برای هر $n$ تایی $a=\langle a_0,a_1,...,a_{n-1}\rangle\in U$، تابع درهم‌ساز $h_a$ را به صورت زیر تعریف می‌کنیم:

$$h_a(x)=\left(\sum_{j=0}^{n-1}a_j x_j\right) \bmod p$$

فرض کنید $H=\{h_a\}$. نشان دهید که H جهانی است، ولی ۲- جهانی نیست. (*راهنمایی:* کلیدی بیابید برای آن تمام توابع درهم‌ساز درون H یک مقدار تولید می‌کنند.)

III. فرض کنید H را از بخش II کمی تغییر می‌دهیم: برای هر $a\in U$ و برای هر $b\in Z_p$ تعریف می‌کنیم

مسائل

$$h'_{ab}(x) = \left( \sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$

و فرض کنید $H' = \{h'_{ab}\}$. بحث کنید که H'، ۲–جهانی است. (*راهنمایی:* $n$ تایی‌های ثابت $x \in U$ و $y \in U$ را در نظر بگیرید، به صورتی که $x_i \neq y_i$ برای یک $i$. وقتی $a_i$ و $b$ روی $Z_p$ تغییر می‌کنند، برای $h'_{ab}(x)$ و $h'_{ab}(y)$ چه رخ می‌دهد؟)

۱۷. فرض کنید که آلیس و باب به صورت خصوصی بر روی یک تابع درهم‌ساز $h$ از H که یک خانواده‌ی ۲– جهانی از توابع درهم‌ساز است، توافق کرده‌اند. سپس آلیس یک پیام $m$ را از طریق اینترنت برای باب می‌فرستد، که $m \in U$. سپس به دنبال آن یک ضمیمه‌ی $t = h(m)$ را برای تعیین هویت این پیغام ارسال می‌کند، و باب پس از دریافت جفت $(m,t)$، چک می‌کند که آیا این جفت در تساوی $t = h(m)$ صدق می‌کنند یا خیر. فرض کنید که یک دشمن، $(m,t)$ را در مسیر استراق سمع می‌کند و سعی می‌کند با جایگزینی این جفت با جفت $(m',t')$، باب را فریب دهد. بحث کنید که احتمال این که این دشمن در فریب باب موفق شود و باب جفت $(m',t')$ را بپذیرد حداکثر $1/p$ است، مستقل از این که از چه مقدار قدرت محاسبه برای انتخاب $(m',t')$ استفاده می‌شود، حتی اگر دشمن از خانواده‌ی H مورد استفاده به عنوان توابع درهم‌ساز اطلاع داشته باشد.

فصل

# ۱۲

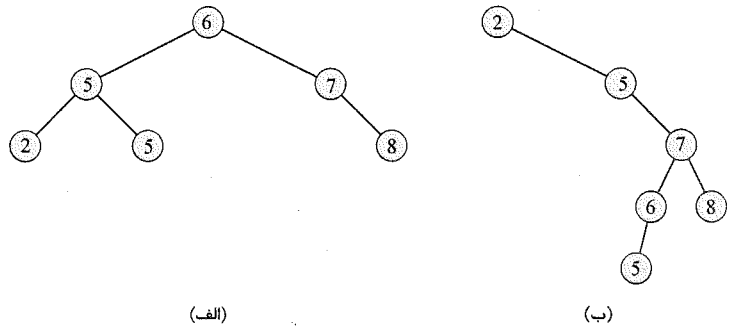
# درخت‌های جستجوی دودویی

## ۱۲-۰ مقدمه

درخت‌های جستجو ساختمان‌های داده‌ای هستند که از بسیاری از اعمال مجموعه‌های پویا پشتیبانی می‌کنند، از جمله SEARCH، MINIMUM، MAXIMUM، PREDECESSOR، SUCCESSOR، INSERT، و DELETE. بنابراین از یک درخت جستجو هم می‌توان به عنوان یک دیکشنری و هم به عنوان یک صف اولویت استفاده کرد.

اعمال اصلی بر روی یک درخت جستجوی دودویی متناسب است با ارتفاع درخت. برای یک درخت جستجوی دودویی کامل با $n$ گره، این اعمال در بدترین حالت در زمان $\theta(\lg n)$ اجرا می‌شوند. با این حال اگر درخت یک زنجیره‌ی خطی شامل $n$ گره باشد، همان اعمال در بدترین حالت در زمان $\theta(n)$ اجرا می‌شوند. در بخش ۱۲-۴ خواهیم دید که امیدریاضی ارتفاع یک درخت جستجوی دودویی که به صورت تصادفی ساخته می‌شود $O(\lg n)$ است، بنابراین اعمال اصلی برروی چنین درختی در حالت متوسط در زمان $\theta(\lg n)$ اجرا خواهند شد.

در عمل نمی‌توانیم همیشه تضمین کنیم که درخت‌های جستجوی دودویی به صورت تصادفی ساخته می‌شوند، ولی می‌توانیم انواعی از درخت‌های جستجوی دودویی طراحی کنیم که کارایی بدترین حالت زمان اجرای آن‌ها برای عملیات اصلی تضمین شده است. در فصل ۱۳ یکی از این انواع، به نام درخت‌های قرمز-سیاه ارائه می‌شود، که ارتفاع آن $O(\lg n)$ است. در فصل ۱۸ B-درخت‌های معرفی می‌شوند، که به طور خاص برای نگه داری پایگاه‌های داده بر روی حافظه‌های ثانویه با دسترسی تصادفی (دیسک) مناسب هستند.




**شکل ۱۲-۱** درخت‌های جستجوی دودویی. برای هر گره‌ی $x$، کلیدهای درون زیردرخت سمت چپ $x$ حداکثر برابرند با $x.key$، و کلیدهای درون زیردرخت سمت راست $x$ حداقل برابرند با $x.key$. درخت‌های جستجوی دودویی متفاوتی می‌توانند حاوی یک مجموعه‌ی خاص از مقادیر باشند. بدترین حالت زمان اجرای اکثر اعمال درخت جستجو متناسب است با ارتفاع درخت. (الف) یک درخت جستجوی دودویی برای ۶ گره با ارتفاع ۲. (ب) یک درخت جستجوی دودویی نسبتا ناکارامد با ارتفاع ۴ و همان مجموعه‌ی گره‌ها.

پس از معرفی خصوصیات اصلی درخت‌های جستجوی دودویی، خواهیم دید که چطور بر روی یک درخت جستجوی دودویی حرکت کرده و مقادیر آن را به صورت مرتب شده چاپ کنیم، چطور در یک درخت جستجوی دودویی برای یک مقدار جستجو کنیم، چطور عناصر با مقدار بیشینه یا کمینه را پیدا کنیم، چطور عناصر ماقبل یا مابعد یک عنصر را بیابیم، و چطور عملیات درج و یا حذف را در یک درخت جستجوی دودویی انجام دهیم.

## ۱-۱۲ درخت جستجوی دودویی چیست؟

در درخت‌های جستجوی دودویی، همان طور که از نام آن‌ها پیدا است، عناصر به صورت یک درخت دودویی سازمان‌دهی شده‌اند، مانند شکل ۱۲-۱. چنین درختی را می‌توان به صورت یک ساختمان داده‌ی پیوندی نمایش داد که در آن هر گره یک شیئ است. علاوه بر یک فیلد *key* و داده‌های پیرو، هر گره حاولی فیلدهای *left*، *right*، و *p* است که به ترتیب به گره‌های فرزند سمت چپ، فرزند سمت راست، و پدر اشاره می‌کنند. اگر یکی از فرزندها و یا پدر گره وجود نداشته باشد، فیلد مربوطه حاوی مقدار NIL خواهد بود. ریشه‌ی درخت، تنها گره‌ای است که فیلد پدر آن NIL است.

کلیدهای درون یک درخت جستجوی دودویی همیشه طوری طراحی شده‌اند که *خصوصیت درخت‌های جستجوی دودویی* را ارضا کنند:

- فرض کنید $x$ یک گره در یک درخت جستجوی دودویی باشد. اگر $y$ یک گره در زیردرخت سمت چپ $x$ باشد، آن گاه $y.key \le x.key$، و اگر $y$ یک گره در زیردرخت سمت راست $x$ باشد، آن گاه $y.key \ge x.key$.

همان طور که در شکل ۱۲-۱(الف) می‌بینید، کلید ریشه ۶ است، کلیدهای ۲، ۵ و ۵ در زیردرخت سمت راست بزرگ‌تر از ۶ نیستند، و کلیدهای ۷ و ۸ در زیردرخت سمت چپ کوچک‌تر از ۶ نیستند. چنین خصوصیتی برای تمام گره‌ها در درخت برقرار است. به عنوان مثال، کلید ۵ مربوط به فرزند سمت چپ ریشه کوچک‌تر از کلید ۲ در زیردرخت سمت چپ آن، و همچنین بزرگ‌تر از کلید ۵ در زیردرخت سمت راست آن نیست.

خصوصیت درخت‌های جستجوی دودویی به ما اجازه می‌دهد که تمام کلیدها در یک درخت جستجوی دودویی را به صورت مرتب شده چاپ کنیم، به کمک یک الگوریتم بازگشتی ساده به نام *پیمایش میان‌ترتیبی درخت* (inorder tree walk). این الگوریتم این گونه نام گذاری شده است زیرا کلید ریشه بین کلیدهای زیردرخت سمت راست و زیردرخت سمت چپ چاپ می‌شود. (به طور مشابه، در *پیمایش پیش‌ترتیبی درخت* (preorder tree walk) ریشه قبل از تمام مقادیر در زیردرخت خود چاپ می‌شود، و در *پیمایش پس‌ترتیبی درخت* (postorder tree walk)، ریشه بعد از تمام مقادیر در زیردرخت خود چاپ می‌شود.) برای استفاده از رویه‌ی زیر برای چاپ تمام عناصر در یک درخت جستجوی دودویی $T$ از فراخوانی INORDER-TREE-WAKL ($T.root$) استفاده می‌کنیم.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

به عنوان یک مثال، پیمایش میان‌ترتیبی درخت تمام کلیدها در هر یک از دو درخت جستجوی دودویی شکل ۱۲-۱ را به صورت ۲، ۳، ۵، ۵، ۷، ۸ چاپ می‌کند. درستی الگوریتم را می‌توان به کمک استقرا و مستقیماً با استفاده از خصوصیت درخت‌های جستجوی دودویی اثبات کرد.

پیمایش یک درخت جستجوی دودویی با $n$ گره به زمان $\theta(n)$ نیاز دارد، چرا که بعد از فراخوانی اولیه، رویه دقیقاً دو بار برای هر یک از گره‌های درخت فراخوانی می‌شود — یک بار برای فرزند چپ و یک بار برای فرزند راست. قضیه‌ی زیر یک اثبات رسمی برای خطی بودن زمان اجرای پیمایش میان‌ترتیبی می‌دهد.

**قضیه‌ی ۱-۱۲** اگر $x$ ریشه‌ی یک زیردرخت با $n$ گره باشد، فراخوانی INORDER-TREE-WALK به زمان $\theta(n)$ نیاز دارد.

*اثبات* فرض کنید $T(n)$ زمان اجرای INORDER-TREE-WALK باشد، وقتی که بر روی ریشه‌ی یک زیردرخت با $n$ گره فراخوانی می‌شود. چون INORDER-TREE-WALK تمام $n$ گره‌ی زیردرخت را پیمایش می‌کند، داریم $T(n) = \Omega(n)$. این می‌ماند که نشان دهیم $T(n) = O(n)$.

از آن جایی که INORDER-TREE-WALK بر روی یک زیردرخت تهی به زمان ثابت کوتاهی نیاز دارد (برای تست $x \neq$ NIL)، پس $T(0) = c$ برای یک ثابت مثبت $c$.

برای $n > 0$، فرض کنید INORDER-TREE-WALK بر روی گره‌ی $x$ فراخوانی می‌شود، که زیردرخت سمت چپ آن $k$ گره و زیردرخت سمت راست آن $n-k-1$ گره دارد. زمان اجرای INORDER-TREE-WALK برابر است با $T(n)=T(k)+T(n-k-1)+d$، که $d$ ثابتی است نشان دهنده‌ی زمان اجرای INORDER-TREE-WALK(x) جدا از زمان فراخوانی‌های بازگشتی.

از روش جانشین‌سازی برای نشان دادن $T(n)=\theta(n)$ استفاده می‌کنیم، و اثبات می‌کنیم که $T(n)=(c+d)n+c$. برای $n=0$ داریم $T(0)=c=0+c=(c+d).0+c$. 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

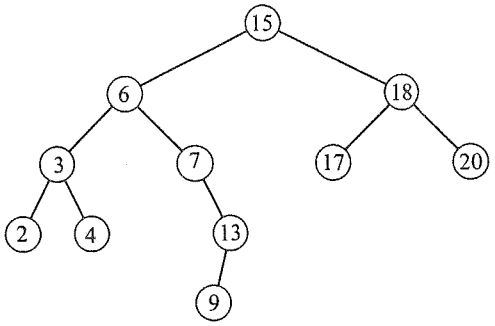
۱-۱-۱۲ برای مجموعه‌ی کلیدهای $\{1,4,5,10,16,17,21\}$ درخت‌های جستجوی دودویی با ارتفاع ۲، ۳، ۴، ۵، و ۶ بکشید.

۲-۱-۱۲ تفاوت میان خصوصیت درخت‌های جستجوی دودویی و خصوصیت هرم کمینه چیست (بخش ۶-۱ را ببینید)؟ آیا می‌توان از خصوصیت هرم کمینه برای چاپ کلیدهای یک درخت با $n$ گره به ترتیب در زمان $O(n)$ استفاده کرد؟ توضیح دهید چگونه، و یا چرا نه.

۳-۱-۱۲ یک الگوریتم غیر بازگشتی برای انجام پیمایش میان‌ترتیبی ارائه کنید. (*راهنمایی:* یک راه حل ساده وجود دارد که در آن از یک پشته به عنوان ساختمان داده‌ی کمکی استفاده می‌شود، و همچنین یک راه حل نسبتاً پیچیده ولی با حافظه‌ی مصرفی کم‌تر وجود دارد که از پشته استفاده نمی‌کند، ولی فرض می‌کند که می‌توان تساوی دو اشاره‌گر را بررسی کرد.)

۴-۱-۱۲ دو الگوریتم بازگشتی برای انجام پیمایش‌های پیش‌ترتیبی و پس‌ترتیبی ارائه کنید که برای یک درخت با $n$ گره در زمان $\theta(n)$ اجرا شوند.

۵-۱-۱۲ بحث کنید که از آن‌جایی که مرتب‌سازی $n$ عنصر در مدل مقایسه‌ای در بدترین حالت در زمان $\Omega(n \lg n)$ اجرا می‌شود، هر الگوریتم بر مبنای مقایسه برای ساختن یک درخت جستجوی دودویی از یک لیست دلخواه از $n$ عنصر در بدترین حالت به زمان $\Omega(n \lg n)$ نیاز دارد.




**شکل ۱۲-۲** جستجوهای مختلف بر روی یک درخت جستجوی دودویی. برای جستجو به دنبال کلید ۱۳ در درخت، مسیر ۱۳ ← ۷ ← ۶ ← ۱۵ را از ریشه دنبال می‌کنیم. کلید کمینه در درخت ۲ است، که می‌توان با دنبال کردن اشاره‌گرهای *left* از ریشه آن را پیدا کرد. کلید ماکزیمم ۲۰ است که با دنبال کردن اشاره‌گرهای *right* از ریشه می‌توان به آن رسید. گره‌ی مابعد گره‌ی با کلید ۱۵، گره‌ی با کلید ۱۷ است، چرا که این کلید کمینه کلیدهای زیردرخت سمت راست آن است. گره‌ی با کلید ۱۳ زیردرخت سمت راست ندارد، و بنابراین گره‌ی مابعد آن پایین‌ترین جد آن است که گره‌ی مربوطه در زیردرخت سمت چپ آن باشد. در این جا، گره‌ی با کلید ۱۵ گره‌ی مابعد گره‌ی با کلید ۱۳ است.

## ۱۲-۲ جستجوهای مختلف در درخت‌های جستجوی دودویی

یک عملیات معمول بر روی یک درخت جستجوی دودویی، جستجو برای یک کلید ذخیره شده در آن است. علاوه بر عملیات جستجو، درخت‌های جستجوی دودویی می‌توانند از اعمال دیگری مانند MINIMUM، MAXIMUM، SUCCESSOR، و PREDECESSOR پشتیبانی کنند. در این بخش این اعمال را بررسی خواهیم کرد و نشان خواهیم داد که هر کدام از آن‌ها را می‌توان طوری پیاده‌سازی کرد که برای یک درخت با ارتفاع $h$ در زمان $O(h)$ اجرا شوند.

### جستجو

از رویه‌ی زیر برای جستجو به دنبال گره‌ای با یک کلید خاص در یک درخت جستجوی دودویی استفاده می‌کنیم. با داشتن یک اشاره‌گر به ریشه‌ی درخت و یک کلید $k$، رویه‌ی TREE-SEARCH در صورت وجود یک اشاره‌گر به یک گره با کلید $k$ بازمی‌گرداند، و در غیر این صورت NIL را بازمی‌گرداند.

```
TREE-SEARCH (x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

رویه جستجوی خود را از ریشه آغاز می‌کند، و همان‌طور که در شکل ۱۲-۲ مشخص شده است، مسیر خود را به سمت پایین ادامه می‌دهد. برای هر گره‌ی $x$ که به آن برخورد می‌کند، کلید $k$ را با $x.key$ مقایسه می‌کند. اگر دو کلید برابر باشند جستجو پایان می‌یابد. اگر $k$ کوچک‌تر از $x.key$ باشد، جستجو در زیردرخت سمت چپ $x$ ادامه خواهد یافت، چرا که خصوصیت درخت‌های جستجوی دودویی ایجاب می‌کند که $k$ نمی‌تواند در زیردرخت سمت راست باشد. به طور مشابه اگر $k$ بزرگ‌تر از $x.key$ باشد، جستجو در زیردرخت سمت راست $x$ ادامه خواهد یافت. گره‌های بررسی شده طی این رویه‌ی بازگشتی یک مسیر از ریشه به سمت پایین تشکیل می‌دهند، و بنابراین زمان اجرای TREE-SEARCH عبارت است از $O(h)$، که در آن $h$ ارتفاع درخت است.

با «باز کردن» بازگشت و تبدیل آن به یک حلقه‌ی while، می‌توان این رویه‌ی بازگشتی را به یک رویه‌ی تکراری تبدیل کرد. در اکثر کامپیوترها این نسخه بهینه‌تر است.

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

## بیشینه و کمینه

همان‌طور که در شکل ۱۲-۲ نشان داده شده است، همیشه می‌توان عنصر کمینه را در یک درخت جستجوی دودویی با دنبال کردن اشاره‌گرهای *left* از ریشه، تا برخورد به مقدار NIL، پیدا کرد. رویه‌ی زیر یک اشاره‌گر به عنصر کمینه در زیردرخت با ریشه‌ی $x$ بازمی‌گرداند، که فرض می‌کنیم $x$ NIL نیست.

```
TREE-MINIMUM (x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

خصوصیت درخت‌های جستجوی دودویی تضمین می‌کند که TREE-MINIMUM صحیح است. اگر یک گره‌ی $x$ زیردرخت سمت چپ نداشته باشد، آن گاه از آن جایی که تمام کلیدها در زیردرخت سمت راست $x$ حداقل برابرند با $x.key$، کلید کمینه در زیردرخت با ریشه‌ی $x$ خود $x.key$ است. اگر گره‌ی $x$ زیردرخت سمت چپ داشته باشد، آن گاه از آن جایی که هیچ کلیدی در زیردرخت سمت راست از $x.key$ کوچک‌تر نیست و هیچ کلیدی در زیردرخت سمت چپ بزرگ‌تر از $x.key$ نیست، کلید کمینه زیردرخت با ریشه‌ی $x$ را می‌توان در زیردرخت با ریشه‌ی $x.left$ جستجو کرد.

رویه‌ی TREE-MAXIMUM به صورت مشابه نوشته می‌شود.

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
```

```
2      x = x.right
3  return x
```

هر دوی این رویه‌ها برای یک درخت با ارتفاع $h$ در زمان $O(h)$ اجرا می‌شوند، چرا که مانند TREE-SEARCH، دنباله‌ی گره‌های بررسی شده یک مسیر از ریشه به سمت پایین تشکیل می‌دهند.

## عناصر ماقبل و مابعد

وقتی یک گره در یک درخت جستجوی دودویی داریم، بعضی مواقع مهم است که بتوانیم عناصر ماقبل و مابعد آن را (در ترتیب نزولی کلیدها) که در پیمایش میان‌ترتیبی درخت تعیین می‌شوند، بیابیم. اگر تمام کلیدها متمایز باشند، عنصر مابعد $x$، گره‌ی با کوچک‌ترین کلید بزرگ‌تر از $x.key$ است. ساختار یک درخت جستجوی دودویی به ما اجازه می‌دهد که عنصر مابعد یک گره را بدون هیچ مقایسه‌ای بین کلیدها بیابیم. رویه‌ی زیر عنصر مابعد یک گره‌ی $x$ را در یک درخت جستجوی دودویی در صورت وجود بازمی‌گرداند، و اگر $x$ عنصر با بزرگ‌ترین کلید در درخت باشد، NIL را بازمی‌گرداند.

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM (x.right)
3  y = x.p
4  while y ≠ NIL and x = y.right
5      x = y
6      y = y.p
7  return y
```

کد TREE-SUCCESSOR به دو حالت تقسیم می‌شود. اگر زیردرخت سمت راست گره‌ی $x$ ناتهی باشد، در این صورت عنصر مابعد $x$ به سادگی برابر است با چپ‌ترین عنصر زیردرخت سمت راست $x$، که در خط ۲ با فراخوانی TREE-MINIMUM ($x.right$) پیدا می‌شود. به عنوان مثال، عنصر مابعد گره‌ی با کلید ۱۵ در شکل ۱۲-۲، گره‌ی با کلید ۱۷ است.

در حالت دیگر، همان‌طور که در تمرین ۱۲-۲-۶ از شما خواسته می‌شود نشان دهید، اگر زیردرخت سمت راست $x$ تهی باشد و $x$ یک عنصر مابعد $y$ داشته باشد، آن گاه $y$ پایین‌ترین جد $x$ است که $x$ در زیردرخت سمت چپ آن است. در شکل ۱۲-۲، عنصر مابعد گره‌ی با کلید ۱۳، گره‌ی با کلید ۱۵ است. برای یافتن $y$، به سادگی آن قدر به سمت بالا می‌رویم تا به گره‌ای برخورد کنیم که فرزند سمت چپ پدر خود باشد؛ این کار در خطوط ۳-۷ رویه‌ی TREE-SUCCESSOR انجام می‌شود.

زمان اجرای TREE-SUCCESSOR بر روی یک درخت با ارتفاع $h$ برابر است با $O(h)$، چرا که یا یک مسیر به سمت بالا را دنبال می‌کنیم و یا یک مسیر به سمت پایین. رویه‌ی TREE-PREDECESSOR هم، که مشابه TREE-SUCCESSOR است، در زمان $O(h)$ اجرا می‌شود.

حتی اگر کلیدها متمایز نباشند، عناصر ماقبل و مابعد هر گره را به صورت گره‌ی بازگشتی توسط

رویه‌های TREE-SUCCESSOR $(x)$ و TREE-PREDECESSOR $(x)$ تعریف می‌کنیم.
به طور خلاصه در بالا، قضیه‌ی زیر را اثبات کردیم.

**قضیه‌ی ۱۲-۲** اعمال مجموعه‌های پویای SEARCH، MINIMUM، MAXIMUM، SUCCESSOR، و PREDECESSOR را می‌توان بر روی یک درخت جستجوی دودویی با ارتفاع $h$ در زمان $O(h)$ پیاده‌سازی کرد.

## تمرین‌ها

۱-۲-۱۲ فرض کنید اعدادی بین ۱ و ۱۰۰۰ در یک درخت جستجوی دودویی داریم، و می‌خواهیم برای عدد ۳۶۳ جستجو کنیم. کدام یک از دنباله‌های زیر نمی‌توانند دنباله‌ی کلیدهای جستجو شده باشند؟

I. ۲، ۲۵۲، ۴۰۱، ۳۹۸، ۳۳۰، ۳۹۷، ۳۶۳.
II. ۹۲۴، ۲۲۰، ۹۹۱، ۲۴۴، ۸۹۸، ۲۵۸، ۳۶۲، ۳۶۳.
III. ۹۲۵، ۲۰۲، ۹۱۱، ۲۴۰، ۹۱۲، ۲۴۵، ۳۶۳.
IV. ۲، ۳۹۹، ۳۸۷، ۲۱۹، ۲۶۶، ۳۸۲، ۳۸۱، ۲۷۸، ۳۶۳.
V. ۹۳۵، ۲۷۸، ۳۴۷، ۶۲۱، ۲۹۹، ۳۹۲، ۳۵۶، ۳۶۳.

۲-۲-۱۲ نسخه‌ی بازگشتی رویه‌های TREE-MINIMUM و TREE-MAXIMUM را بنویسید.

۳-۲-۱۲ رویه‌ی TREE-PREDECESSOR را بنویسید.

۴-۲-۱۲ پروفسور بانین (Bunyan) فکر می‌کند یک خصوصیت قابل توجه درخت‌های جستجوی دودویی را کشف کرده است. فرض کنید که جستجو برای یک کلید $k$ در یک درخت جستجوی دودویی در یک برگ تمام می‌شود. سه مجموعه‌ی زیر را در نظر بگیرید: $A$، کلیدهایی که در سمت چپ مسیر جستجو قرار دارند؛ $B$، کلیدهای روی مسیر جستجو؛ و $C$، کلیدهایی که در سمت راست مسیر جستجو قرار دارند. پروفسور بانین ادعا می‌کند که هر سه کلید $a \in A$، $b \in B$، و $c \in C$ باید در نامساوی $a \le b \le c$ صدق کنند. کوچک‌ترین مثال نقض ممکن را برای ادعای پروفسور بیابید.

۵-۲-۱۲ نشان دهید که اگر یک گره در یک درخت جستجوی دودویی دو فرزند داشته باشد، آن گاه عنصر مابعد آن فرزند سمت چپ، و عنصر ماقبل آن فرزند سمت راست ندارد.

۶-۲-۱۲ یک درخت جستجوی دودویی $T$ را در نظر بگیرید که کلیدهای آن متمایز هستند. نشان دهید که اگر زیردرخت سمت راست گره‌ی $x$ در درخت $T$ تهی باشد و $x$ یک عنصر مابعد مانند $y$ داشته باشد، آن گاه $y$ پایین‌ترین جد $x$ است که فرزند سمت چپ آن نیز یک جد $x$ باشد. (به یاد بیاورید که هر عنصر، جد خود نیز محسوب می‌شود.)

۱۲-۲-۷ یک پیمایش میان‌ترتیبی روی یک درخت جستجوی دودویی با $n$ گره را می‌توان به وسیله‌ی یافتن عنصر کمینه‌ی درخت (به کمک TREE-MIMIMUM)، و سپس $n-1$ بار فراخوانی TREE-SUCCESSOR انجام داد. نشان دهید که زمان اجرای این الگوریتم $\theta(n)$ است. (به خاطر داشته باشید که هر گره، جد خود نیز محسوب می‌شود.)

۱۲-۲-۸ اثبات کنید که مستقل از این که از کدام گره در یک درخت جستجوی دودویی با ارتفاع $h$ شروع می‌کنیم، $k$ بار فراخوانی متوالی TREE-SUCCESSOR به زمان $O(k+h)$ نیاز دارد.

۱۲-۲-۹ فرض کنید $T$ یک درخت جستجوی دودویی باشد که کلیدهای آن متمایز هستند، و $x$ یک برگ، و $y$ پدر آن است. نشان دهید که $y.key$ یا کوچک‌ترین عنصر بزرگ‌تر از $x.key$ در $T$ است، و یا بزرگ‌ترین عنصر کوچک‌تر از $x.key$ در $T$.

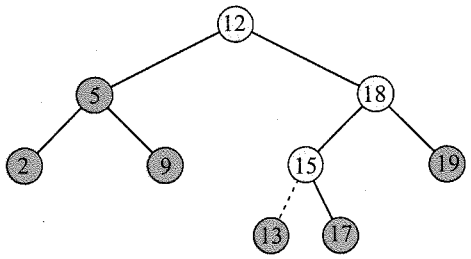
## ۱۲-۳ درج و حذف

اعمال درج و حذف باعث می‌شود که مجموعه‌ی پویای ساخته شده توسط درخت جستجوی دودویی تغییر کند. ساختمان داده باید طوری اصلاح شود که بازتاب این تغییر را نشان دهد، ولی طوری که خصوصیت درخت‌های جستجوی دودویی همچنان حفظ شود. همان طور که خواهیم دید، اصلاح درخت برای درج یک عنصر نسبتا کار ساده‌ای است، ولی برای حذف کار کمی پیچیده‌تر می‌شود.

### درج

برای درج یک مقدار جدید $v$ درون یک درخت جستجوی دودویی $T$ از رویه‌ی TREE-INSERT استفاده می‌کنیم. یک گره‌ی $z$ به رویه‌ی ارسال می‌شود، به طوری که $z.key = v$، $z.left = \text{NIL}$، و $z.right = \text{NIL}$. این رویه، $T$ و بعضی فیلدهای $z$ را طوری تغییر می‌دهد که $z$ در یک مکان مناسب در درخت درج شود.

```
TREE-INSERT(T, z)
1   y = NIL
2   x = T.root
3   while x ≠ NIL
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y
9   if y == NIL
10      T.root = z          // Tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```




**شکل ۱۲-۳** درج یک عنصر با کلید ۱۳ درون یک درخت جستجوی دودویی. گره‌های کم‌رنگ مشخص‌کننده‌ی مسیر طی شده از ریشه تا محل درج عنصر جدید هستند. خط نقطه‌چین پیوندی را مشخص می‌کند که به درخت اضافه شده است تا عنصر جدید درج شود.

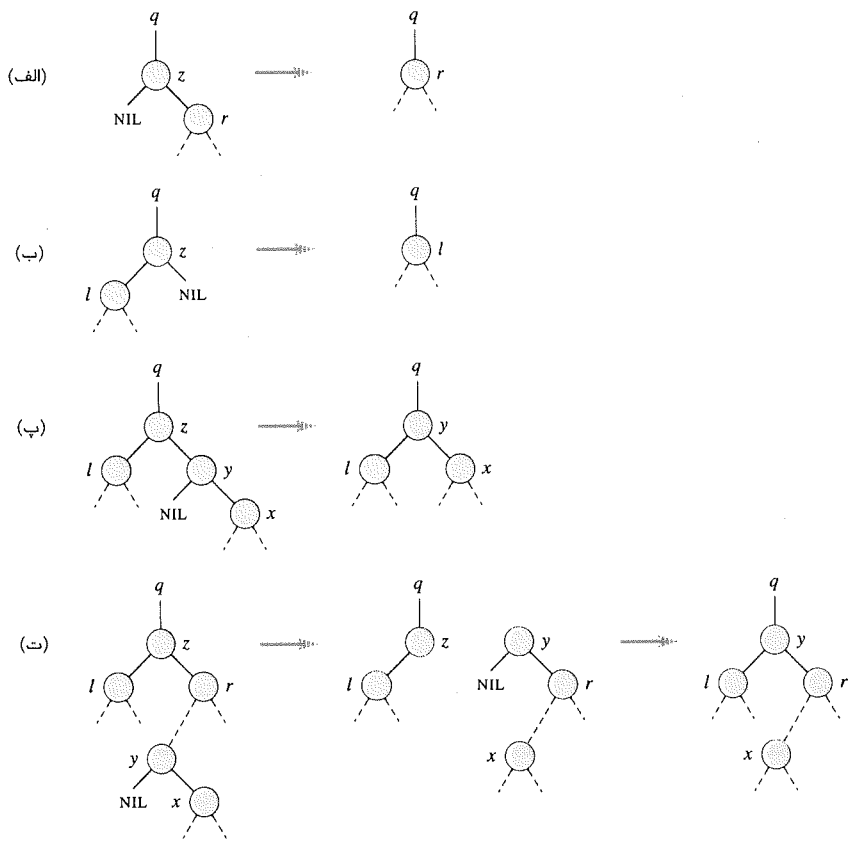
شکل ۱۲-۳ نشان می‌دهد که TREE-INSERT چگونه کار می‌کند. درست مانند رویه‌های TREE-SEARCH و ITERATICVE-TREE-SEARCH، رویه‌ی TREE-INSERT از ریشه‌ی درخت شروع و یک مسیر ساده به سمت پایین را دنبال می‌کند. اشاره‌گر $x$ مسیر را دنبال می‌کند، و اشاره‌گر $y$ نشان دهنده‌ی پدر $x$ است. بعد از مقداردهی اولیه، حلقه‌ی **while** در خطوط ۳-۷ باعث می‌شود که دو اشاره‌گر به سمت پایین درخت حرکت کنند، و در این حرکت بسته به نتیجه‌ی مقایسه‌ی $z.key$ با $x.key$ مسیر خود را به سمت چپ یا راست ادامه دهند، تا وقتی که $x$ مقدار NIL به خود بگیرد. این NIL همان جایی است که می‌خواهیم ورودی $z$ را قرار دهیم. خطوط ۸-۱۳ مقداردهی‌های نهایی اشاره‌گرها را برای درج $z$ انجام می‌دهند.

مانند بقیه‌ی اعمال اصلی بر روی درخت‌های جستجو، رویه‌ی TREE-INSERT بر روی یک درخت با ارتفاع $h$ در زمان $O(h)$ اجرا می‌شود.

## حذف

رویه‌ی حذف یک گره‌ی داده شده‌ی $z$ از یک درخت جستجوی دودویی، یک اشاره‌گر به $z$ را به عنوان آرگومان ورودی دریافت می‌کند. رویه سه حالت نشان داده شده در شکل ۱۲-۴ را در نظر می‌گیرد.

- اگر $z$ هیچ فرزندی نداشته باشد، پدر آن ($z.p$) را طوری اصلاح می‌کنیم که $z$ با NIL جایگزین شود.
- اگر گره فقط یک فرزند داشته باشد، آن گاه با اصلاح پدر $z$، فرزند آن را جایگزین آن می‌کنیم.
- نهایتاً اگر گره دو فرزند داشته باشد، عنصر مابعد $z$ یعنی $y$ را – که باید در زیردرخت سمت راست $z$ باشد– پیدا کرده و آن را جایگزین $z$ در درخت می‌کنیم. باقی زیردرخت سمت راست $z$ تبدیل به زیردرخت راست $y$ می‌شود، و زیردرخت چپ $z$ تبدیل به زیردرخت چپ $y$. این حالت کمی پیچیده است، چرا که همان طور که خواهیم دید، اگر $y$ فرزند سمت راست $z$ باشد تفاوت‌هایی به وجود خواهد آمد.




**شکل ۱۲-٤** حذف یک گره‌ی $z$ از یک درخت جستجوی دودویی. گره‌ی $z$ ممکن است ریشه باشد، فرزند سمت چپ $q$ باشد، و یا فرزند سمت راست آن. (الف) گره‌ی $z$ فرزند سمت چپ ندارد. $z$ را با فرزند سمت راست آن، $r$، جایگزین می‌کنیم، که ممکن است NIL باشد یا نباشد. (ب) گره‌ی $z$ یک فرزند سمت چپ $l$ دارد، ولی فرزند سمت راست ندارد. $z$ را با $l$ جایگزین می‌کنیم. (پ) گره‌ی $z$ دو فرزند دارد؛ فرزند سمت چپ آن گره‌ی $l$ است، فرزند سمت راست آن همان عنصر مابعد آن، یعنی $y$ است، و فرزند سمت راست $y$ گره‌ی $x$ است. $z$ را با $y$ جایگزین می‌کنیم، $l$ فرزند سمت چپ $y$ خواهد شد، و فرزند سمت راست $y$ همان $x$ باقی خواهد ماند. (ت) گره‌ی $z$ دو فرزند دارد ($l$ فرزند سمت چپ، و $r$ فرزند سمت راست)، و عنصر مابعد آن $y \neq r$ درون زیردرخت با ریشه‌ی $r$ قرار دارد. $y$ را با فرزند سمت راست خودش یعنی $x$ جایگزین می‌کنیم، و $y$ پدر $r$ خواهد شد. سپس مقداردهی‌های لازم را انجام می‌دهیم تا $y$ فرزند $q$ و پدر $l$ هم بشود.

رویه‌ی حذف یک گره‌ی داده شده‌ی $z$ از یک درخت جستجوی دودویی $T$، اشاره‌گرهایی به $T$ و $z$ را به عنوان آرگومان ورودی دریافت می‌کند. در این رویه، اداره‌ی موقعیت‌های مختلف کمی متفاوت از از سه حالت ارائه شده در بالا انجام می‌شود، یعنی به صورت چهار حالتی که در شکل ۱۲-۴ نشان داده شده است.

- اگر $z$ فرزند سمت چپ نداشته باشد (قسمت (الف) از شکل)، آن گاه $z$ را با فرزند سمت راست آن جایگزین می‌کنیم، که ممکن است NIL باشد یا نباشد. وقتی فرزند سمت راست $z$ برابر NIL باشد، این حالت معادل است با زمانی که $z$ هیچ فرزندی ندارد، و وقتی فرزند سمت راست $z$ غیر NIL باشد، این حالت معادل وضعیتی است که در آن $z$ فقط یک فرزند دارد، و آن فرزند سمت راست است.
- اگر $z$ فقط یک فرزند داشته باشد، و آن فرزند سمت چپ باشد (قسمت (ب) از شکل)، آن گاه $z$ را با فرزند سمت راست آن جایگزین می‌کنیم.
- در غیر این صورت، $z$ هم فرزند سمت چپ دارد و هم فرزند سمت راست. در این وضعیت گره‌ی مابعد $z$ یعنی $y$ را می‌یابیم، که در زیردرخت سمت راست $z$ قرار دارد و هیچ فرزندی ندارد (تمرین ۱۲-۲-۵ را ببینید). می‌خواهیم $y$ را از مکان فعلی خود بیرون کشیده و آن را جایگزین $z$ کنیم.
  - اگر $y$ فرزند سمت راست $z$ باشد (قسمت (پ))، آن گاه $z$ را با $y$ جایگزین می‌کنیم، و کاری به فرزند سمت راست $y$ نداریم.
  - در غیر این صورت، $y$ در زیردرخت سمت راست $z$ است، ولی فرزند سمت راست آن نیست (قسمت (ت)). در این حالت، ابتدا $y$ را با فرزند سمت راست خودش، و سپس $z$ را با $y$ جایگزین می‌کنیم.

برای جابه‌جا کردن زیردرخت‌ها درون درخت جستجوی دودویی، یک زیرروال TRANSPLANT تعریف می‌کنیم، که یک زیردرخت را با زیردرخت دیگر جایگزین می‌کند. وقتی TRANSPLANT زیردرخت با ریشه‌ی $u$ را با زیردرخت با ریشه‌ی $v$ جایگزین می‌کند، پدر گره‌ی $u$ تبدیل به پدر گره‌ی $v$ می‌شود، و $v$ فرزند جدید پدر گره‌ی $u$ خواهد شد.

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

خطوط ۱-۲ حالتی را اداره می‌کنند که در آن $u$ ریشه‌ی $T$ است. در غیر این صورت، $u$ یا فرزند سمت چپ پدر خود است و یا فرزند سمت راست پدر خود. در صورتی که $u$ فرزند سمت چپ باشد، خطوط ۳-۴ به روزرسانی $u.p.left$ را برعهده دارند، و در صورتی که فرزند سمت راست باشد، خط ۵، $u.p.right$ را به روزرسانی می‌کند. اجازه می‌دهیم که $v$ مقدار NIL داشته باشد، و اگر غیر NIL باشد، خطوط ۶-۷ اشاره‌گر $v.p$ را به روزرسانی می‌کنند. توجه کنید که TRANSPLANT سعی نمی‌کند که $v.left$ و $v.right$ را به روزرسانی کند؛ انجام یا عدم انجام این کار وظیفه‌ی فراخواننده‌ی

TRANSPLANT است.

با در دست داشتن رویه‌ی TRANSPLANT، رویه‌ی زیر گره‌ی $z$ را از درخت جستجوی دودویی $T$ حذف می‌کند:

```
TREE-DELETE(T, z)
1  if z.left == NIL
2      TRANSPLANT(T, z, z.right)
3  elseif z.right == NIL
4      TRANSPLANT(T, z, z.left)
5  else y = TREE-MINIMUM(z.right)
6      if y.p ≠ z
7          TRANSPLANT(T, y, y.right)
8          y.right = z.right
9          y.right.p = y
10     TRANSPLANT(T, z, y)
11     y.left = z.left
12     y.left.p = y
```

رویه‌ی TREE-DELETE چهار حالت گفته شده را به صورت زیر اجرا می‌کند. خطوط ۱-۲ حالتی را اداره می‌کنند که در آن گره‌ی $z$ هیچ فرزندی ندارد، و خطوط ۳-۴ حالتی را که $z$ فرزند چپ دارد ولی فرزند راست ندارد. خطوط ۵-۱۲ با دو حالت باقی‌مانده سروکار دارند، که در آن‌ها $z$ دو فرزند دارد. خط ۵ گره‌ی $y$ را پیدا می‌کند، که عنصر مابعد $z$ است. چون زیردرخت سمت راست $z$ تهی است، عنصر مابعد آن باید گره‌ای در زیردرخت سمت راست باشد که کوچک‌ترین کلید را دارد؛ فراخوانی TREE-MINIMUM$(z.right)$ به این دلیل است. همان طور که قبلا گفتیم، $y$ فرزند سمت راست ندارد. می‌خواهیم $y$ را از مکان خود در آورده و جایگزین $z$ کنیم. اگر $y$ فرزند سمت راست $z$ باشد، خطوط ۱۰-۱۲ گره‌ی $z$ را با $y$، و فرزند سمت چپ $y$ را با فرزند سمت چپ $z$ جایگزین می‌کنند. اگر $y$ فرزند سمت راست $z$ نباشد، خطوط ۷-۹ گره‌ی $y$ را با فرزند سمت راست آن جایگزین، و فرزند سمت راست $z$ را تبدیل به فرزند سمت راست $y$ می‌کنند، و سپس خطوط ۱۰-۱۲ گره‌ی $z$ را با $y$، و فرزند سمت چپ $y$ را با فرزند سمت چپ $z$ جایگزین می‌کنند.

تمام خطوط TREE-DELETE، از جمله فراخوانی‌های TRANSPLANT، در زمان ثابت اجرا می‌شوند، غیر از فراخوانی TREE-MINIMUM در خط ۵. بنابراین رویه‌ی TREE-DELETE بر روی یک درخت با ارتفاع $h$ در زمان $O(h)$ اجرا می‌شود.

به طور خلاصه، در بالا قضیه‌ی زیر را اثبات کرده‌ایم.

**قضیه‌ی ۱۲-۳** اعمال مجموعه‌های پویای INSERT و DELETE را می‌توان بر روی یک درخت با ارتفاع $h$ در زمان $O(h)$ پیاده‌سازی کرد.

## تمرین‌ها

۱-۳-۱۲ یک نسخه‌ی بازگشتی از رویه‌ی TREE-INSERT ارائه کنید.

۲-۳-۱۲ فرض کنید یک درخت جستجوی دودویی با درج پی‌درپی عناصر متمایز ساخته شده است. بحث کنید که تعداد گره‌های بررسی شده برای جستجوی یک مقدار در درخت برابر است با یک به علاوه‌ی تعداد گره‌های بررسی شده هنگام درج همان عنصر در درخت.

۳-۳-۱۲ برای مرتب‌سازی یک مجموعه از $n$ عدد می‌توان ابتدا یک درخت جستجوی دودویی حاوی این اعداد ساخت (با استفاده‌ی پی‌درپی از TREE-INSERT برای درج تمام اعداد در درخت) و سپس به کمک پیمایش میان‌ترتیبی اعداد را به ترتیب چاپ کرد. زمان اجرای بهترین حالت و بدترین حالت برای این الگوریتم چیست؟

۴-۳-۱۲ آیا عملیات حذف «جابه‌جایی پذیر» است؟ یعنی آیا حذف $x$ و سپس حذف $y$ همان درختی را می‌سازد که حذف $y$ و سپس حذف $x$ می‌سازد؟ اثبات کنید که چرا، و یا یک مثال نقض بیاورید.

۵-۳-۱۲ فرض کنید هر گره‌ی $x$ به جای این که حاوی یک خصیصه‌ی $x.p$ باشد که به پدر آن اشاره می‌کند، حاوی یک خصیصه‌ی $x.succ$ باشد که به عنصر مابعد آن اشاره می‌کند. با استفاده از این روش نمایش، برای SEARCH، INSERT، و DELETE بر روی یک درخت جستجوی دودویی $T$ شبه‌کد ارائه کنید. این رویه‌ها باید در زمان $O(h)$ اجرا شوند، که در آن $h$ ارتفاع درخت $T$ است. (*راهنمایی:* ممکن است بخواهید زیرروالی برای یافتن پدر یک گره پیاده‌سازی کنید.)

۶-۳-۱۲ وقتی گره‌ی $z$ در TREE-DELETE دو فرزند دارد، می‌توانیم به جای عنصر مابعد آن، عنصر ماقبل آن را از درخت بیرون بکشیم. بحث‌هایی وجود دارد که می‌گوید یک استراتژی عادلانه که به عنصر مابعد و عنصر ماقبل اولویت یکسان بدهد، منجر به کارایی بیشتری خواهد شد. چگونه می‌توان TREE-DELETE را تغییر داد که چنین استراتژی عادلانه‌ای را پیاده‌سازی کند؟

## ۱۲-۴ درخت‌های جستجوی دودویی به صورت تصادفی ساخته شده

نشان دادیم که تمام اعمال اولیه بر روی یک درخت جستجوی دودویی در زمان $O(h)$ اجرا می‌شوند، که $h$ ارتفاع درخت است. ولی با درج و حذف عناصر در یک درخت جستجوی دودویی، ارتفاع آن تغییر می‌کند. مثلاً اگر $n$ عنصر به ترتیب صعودی اکید در درخت درج شوند، درخت یک زنجیر با ارتفاع $n-1$ خواهد بود. از طرف دیگر تمرین ب-۵-۴ نشان می‌دهد که $h \geq \lfloor \lg n \rfloor$. با این حال،

مانند مرتب‌سازی سریع می‌توانیم نشان دهیم که رفتار در حالت متوسط تا حدود زیادی نزدیک به بهترین حالت خواهد بود تا به بدترین حالت.

متأسفانه وقتی از اعمال درج و حذف در ساختن یک درخت جستجوی دودویی استفاده می‌شود، اطلاعات زیادی در مورد ارتفاع متوسط درخت نخواهیم داشت. اگر درخت فقط با درج ساخته شده باشد، تحلیل تا حدود زیادی ساده‌تر خواهد شد. اجازه دهید یک *درخت جستجوی دودویی به صورت تصادفی* ساخته شده بر روی $n$ کلید را به صورت درختی تعریف کنیم که با درج پی‌درپی عناصر در یک درخت خالی با ترتیب تصادفی ساخته شده است، که در آن احتمال وقوع هر یک از $n!$ جایگشت ممکن کلیدها برابر است. (تمرین ۱۲-۴-۳ از شما می‌خواهد نشان دهید که این فرض با فرض این که احتمال وقوع هر یک از درخت‌های جستجوی دودویی ممکن برابر است، تفاوت دارد.) در این بخش قضیه‌ی زیر را اثبات خواهیم کرد.

**قضیه‌ی ۱۲-۳** امیدریاضی ارتفاع یک درخت جستجوی دودویی به صورت تصادفی ساخته شده بر روی $n$ کلید، $O(\lg n)$ است.

*اثبات* با تعریف سه متغیر تصادفی شروع می‌کنیم که به ما کمک می‌کنند ارتفاع یک درخت جستجوی دودویی به صورت تصادفی ساخته شده را اندازه بگیریم. ارتفاع درخت جستجوی دودویی به صورت تصادفی ساخته شده بر روی $n$ کلید را با $X_n$ نشان می‌دهیم، و *ارتفاع توانی* را به صورت $Y_n = 2^{X_n}$ تعریف می‌کنیم. وقتی یک درخت جستجوی دودویی را از روی $n$ کلید می‌سازیم، یک کلید را به عنوان ریشه انتخاب می‌کنیم، و فرض می‌کنیم $R_n$ نشان دهنده‌ی متغیر تصادفی باشد که جایگاه این کلید را در ترتیب کلیدها نشان می‌دهد. مقدار $R_n$ با احتمال برابر می‌تواند هر یک از مقادیر مجموعه‌ی $\{1,2,...,n\}$ باشد. اگر $R_n = i$، آن گاه زیردرخت سمت چپ ریشه، یک درخت جستجوی دودویی به صورت تصادفی ساخته شده با $i-1$ کلید، و زیردرخت سمت راست آن یک درخت جستجوی دودویی به صورت تصادفی ساخته شده با $n-i$ کلید است. از آن جایی که ارتفاع یک درخت جستجوی دودویی، یکی بیشتر است از بیشینه‌ی ارتفاع دو زیردرخت سمت چپ و سمت راست آن، ارتفاع توانی آن برابر است با دو برابر بیشینه‌ی ارتفاع دو زیردرخت آن. اگر بدانیم که $R_n = i$، آن گاه داریم

$$Y_n = 2.\max(Y_{i-1}, Y_{n-i})$$

به عنوان حالت‌های پایه داریم $Y_1 = 1$، چرا که ارتفاع توانی یک درخت با یک گره برابر است با $2^0 = 1$، و همچنین برای راحتی تعریف می‌کنیم $Y_0 = 0$.

سپس متغیرهای تصادفی شاخص $Z_{n,1}, Z_{n,2}, ..., Z_{n,n}$ را به صورت زیر تعریف می‌کنیم:

$$Z_{n,i} = I\{R_n = i\}$$

از آن جایی که $R_n$ با احتمال برابر می‌تواند هر یک از عناصر مجموعه‌ی $\{1,2,...,n\}$ باشد، برای

$i = ۱,۲,...,n$ داریم $\Pr\{R_n = i\} = ۱/n$، و بنابراین طبق لم ۵-۱،

$$E[Z_{n,i}] = ۱/n \qquad (۱-۱۲)$$

برای $i = ۱,۲,...,n$. چون دقیقاً یکی از مقادیر $Z_{n,i}$ برابر ۱ است و بقیه ۰ هستند، داریم

$$Y_n = \sum_{i=۱}^{n} Z_{n,i} (۲ \cdot \max(Y_{i-۱}, Y_{n-i}))$$

نشان خواهیم داد که $E[Y_n]$ نسبت به $n$ از درجه‌ی چندجمله‌ای است، که در نهایت نتیجه می‌دهد $E[X_n] = O(\lg n)$.

ادعا می‌کنیم که متغیر تصادفی شاخص $Z_{n,i} = I\{R_n = i\}$ مستقل از مقادیر $Y_{i-۱}$ و $Y_{n-i}$ است. با انتخاب $R_n = i$، زیردرخت سمت چپ (که ارتفاع توانی آن $Y_{i-۱}$ است) به صورت تصادفی بر روی $i-۱$ کلید ساخته شده است که جایگاه آن‌ها پایین‌تر از $i$ است. این زیردرخت درست مانند هر درخت جستجوی دودویی به صورت تصادفی ساخته شده‌ی دیگر بر روی $i-۱$ کلید است. غیر از تعداد کلیدهایی که در این درخت وجود دارد، انتخاب $R_n = i$ به طور کلی تأثیر دیگری بر روی ساختار درخت نمی‌گذارد، و بنابراین متغیرهای تصادفی $Y_{i-۱}$ و $Z_{n,i}$ مستقل از یکدیگرند. به طور مشابه زیردرخت سمت راست که ارتفاع توانی آن $Y_{n-i}$ است، به صورت تصادفی بر روی $n-i$ کلید ساخته می‌شود که جایگاه آن‌ها بالاتر از $i$ است. ساختار این درخت مستقل از مقدار $R_n$ است، و بنابراین متغیرهای تصادفی $Y_{n-i}$ و $Z_{n,i}$ مستقل از یکدیگرند. از این رو داریم

$$\begin{aligned}
E[Y_n] &= E\left[\sum_{i=۱}^{n} Z_{n,i} (۲.\max(Y_{i-۱}, Y_{n-i}))\right] \\
&= \sum_{i=۱}^{n} E[Z_{n,i} (۲.\max(Y_{i-۱}, Y_{n-i}))] && \text{(طبق خطی بودن امیدریاضی)} \\
&= \sum_{i=۱}^{n} E[Z_{n,i}] E[۲.\max(Y_{i-۱}, Y_{n-i})] && \text{(طبق استقلال)} \\
&= \sum_{i=۱}^{n} \frac{۱}{n} . E[۲.\max(Y_{i-۱}, Y_{n-i})] && \text{(طبق تساوی (۱-۱۲))} \\
&= \frac{۲}{n} \sum_{i=۱}^{n} E[\max(Y_{i-۱}, Y_{n-i})] && \text{(طبق تساوی (پ-۲۲))} \\
&\le \frac{۲}{n} \sum_{i=۱}^{n} (E[Y_{i-۱}] + E[Y_{n-i}]) && \text{(طبق تمرین پ-۳-۴)}
\end{aligned}$$

چون هر یک از عبارت‌های $E[Y_۰], E[Y_۱], ..., E[Y_{n-۱}]$ دو بار در مجموع آخر ظاهر می‌شوند، یک بار به صورت $E[Y_{i-۱}]$ و یک بار به صورت $E[Y_{n-i}]$، رابطه‌ی بازگشتی زیر را خواهیم داشت:

$$E[Y_n] \le \frac{۴}{n}\sum_{i=۰}^{n-۱} E[Y_i] \qquad (۲-۱۲)$$

با استفاده از متد جانشین سازی نشان خواهیم داد که برای تمام مقادیر مثبت $n$، رابطه‌ی بازگشتی (۲-۱۲) جوابی به شکل

$$E[Y_n] \le \frac{۱}{۴}\binom{n+۳}{۳}$$

خواهد داشت. برای این کار از اتحاد

$$\sum_{i=۰}^{n-۱}\binom{i+۳}{۳} = \binom{n+۳}{۴} \qquad (۳-۱۲)$$

استفاده خواهیم کرد. (تمرین ۱۲-۴-۱ از شما می‌خواهد که این اتحاد را اثبات کنید.)

برای حالت پایه، می‌بینیم که کران‌های $Y_۰ = E[Y_۰] \le (۱/۴)\binom{۳}{۳} = ۱/۴$ و $Y_۰ = ۰$ و $Y_۱ = E[Y_۱] \le \frac{۱}{۴}\binom{۱+۳}{۳} = ۱$ برقرارند. برای گام استقرا داریم

$$\begin{aligned}
E[Y_n] &\le \frac{۴}{n}\sum_{i=۰}^{n-۱} E[Y_i] \\
&= \frac{۴}{n}\sum_{i=۰}^{n-۱}\frac{۱}{۴}\binom{i+۳}{۳} && \text{(طبق فرض استقرا)} \\
&= \frac{۱}{n}\sum_{i=۰}^{n-۱}\binom{i+۳}{۳} \\
&= \frac{۱}{n}\binom{n+۳}{۴} && \text{(طبق تساوی (۳-۱۲))} \\
&= \frac{۱}{n}\cdot\frac{(n+۳)!}{۴!(n-۱)!} \\
&= \frac{۱}{۴}\cdot\frac{(n+۳)!}{۳!n!} \\
&= \frac{۱}{۴}\binom{n+۳}{۳}
\end{aligned}$$

اکنون کران $E[Y_n]$ را تعیین کرده‌ایم، ولی هدف نهایی ما این است که برای $E[X_n]$ کرانی بیابیم. همان طور که در تمرین ۱۲-۴-۴ از شما خواسته می‌شود نشان دهید، تابع $f(x) = ۲^x$ محدب است (بخش پ-۳ را ببینید). بنابراین می‌توانیم از نامساوی ینسن (Jensen's inequality) (پ-۲۶) استفاده کنیم، که می‌گوید

$$۲^{E[X_n]} \le E[۲^{X_n}] = E[Y_n]$$

و به دست بیاوریم

$$
\begin{aligned}
۲^{E[X_n]} &\le \frac{۱}{۴}\binom{n+۳}{۳} \\
&= \frac{۱}{۴}\cdot\frac{(n+۳)(n+۲)(n+۱)}{۶} \\
&= \frac{n^۳+۶n^۲+۱۱n+۶}{۲۴}
\end{aligned}
$$

با لگاریتم گرفتن از دو طرف خواهیم داشت $E[X_n] = O(\lg n)$.

## تمرین‌ها

۱-۴-۱۲ تساوی (۱۲-۳) را اثبات کنید.

۲-۴-۱۲ یک درخت جستجوی دودویی با $n$ گره توصیف کنید که میانگین عمق گره‌ها در آن $\theta(\lg n)$، ولی ارتفاع درخت $\omega(\lg n)$ باشد. یک کران بالای حدی برای یک درخت جستجوی دودویی بدهید که میانگین عمق گره‌ها در آن $\theta(\lg n)$ باشد.

۳-۴-۱۲ نشان دهید که عبارت «یک درخت جستجوی دودویی به صورت تصادفی انتخاب شده از روی $n$ کلید، که در آن احتمال وقوع هر یک از درخت‌های ممکن با یکدیگر برابر است» با عبارت «یک درخت جستجوی دودویی به صورت تصادفی ساخته شده» که در این بخش تعریف شد، متفاوت است. (*راهنمایی:* حالت‌های ممکن را برای $n = ۳$ لیست کنید.)

۴-۴-۱۲ نشان دهید که تابع $f(x) = ۲^x$ محدب است.

★ ۵-۴-۱۲ عملیات RANOMIZED-QUICKSORT را بر روی دنباله‌ای از $n$ عدد ورودی متمایز در نظر بگیرید. اثبات کنید که برای هر ثابت $k > ۰$، تمام $n!$ جایگشت ممکن، به غیر از $O(\sqrt{n^k})$ تا از آن‌ها منجر به زمان اجرای $O(n \lg n)$ می‌شوند.

## مسائل

۱-۱۲ ***درخت‌های جستجوی دودویی با کلیدهای مساوی***

کلیدهای برابر در پیاده‌سازی درخت‌های جستجوی دودویی، مشکلاتی را پدید می‌آورند.

a. کارایی حدی TREE-INSERT، وقتی که از آن برای درج $n$ عنصر با کلیدهای برابر در یک درخت خالی استفاده می‌شود چقدر است؟

سعی می‌کنیم با بررسی $z.key = x.key$ قبل از خط ۵ و بررسی $z.key = y.key$ قبل از خط

۱۱ رویه‌ی TREE-INSERT را بهبود بخشیم. اگر حالت تساوی برقرار بود یکی از استراتژی‌های زیر را پیاده‌سازی می‌کنیم. برای هر یک از استراتژی‌ها، کارایی حدی درج $n$ عنصر با کلیدهای یکسان را در یک درخت جستجوی دودویی خالی بیابید. (استراتژی‌ها برای خط ۵ توصیف شده‌اند که در آن کلیدهای $z$ و $x$ را مقایسه می‌کنیم. $x$ را با $y$ جایگزین کنید تا استراتژی مربوط به خط ۱۱ به دست آید.)

I. یک پرچم بولین $x.b$ برای گره‌ی $x$ نگه دارید، و بسته به مقدار $x.b$، مقدار $x$ را برابر با یکی از مقادیر $x.left$ یا $x.right$ قرار دهید. مقدار $x.b$ با هر بار ملاقات $x$ بین مقادیر TRUE و FALSE تغییر می‌کند.

II. لیستی از گره‌های برابر در $x$ نگه دارید، و $z$ را به این لیست اضافه کنید.

III. به صورت تصادفی $x$ را با یکی از مقادیر $x.left$ یا $x.right$ مقداردهی کنید. (کارایی بدترین حالت را به دست آورید، و به صورت غیر رسمی کارایی حالت متوسط را از آن استخراج کنید.)

## ۱۲-۲ درخت‌های مبنایی (Radix trees)

با داشتن دو رشته‌ی $a=a_0a_1...a_p$ و $b=b_0b_1...b_q$، که هر یک از $a_i$ها و $b_j$ها عضو یک مجموعه‌ی مرتب از کاراکترها هستند، می‌گوییم رشته‌ی $a$ *به صورت الفبایی* (lexicographically) *کوچک‌تر* از رشته‌ی $b$ است اگر یکی از دو حالت زیر برقرار باشد:

۱. عدد $j$ موجود باشد به طوری که $0 \le j \le \min(p,q)$، و برای $i=0,1,...,j-1$ داشته باشیم $a_i=b_i$ و $a_j<b_j$، یا

۲. $p<q$ باشد و برای $i=0,1,...,p$ داشته باشیم $a_i=b_i$.

برای مثال اگر $a$ و $b$ رشته‌های بیتی باشد، آن‌گاه طبق قانون اول ۱۰۱۱۰<۱۰۱۰۰ (با قرار دادن $j=3$) و طبق قانون دوم ۱۰۱۰۰۰<۱۰۱۰۰. این مرتب‌سازی مانند ترتیب استفاده شده در دیکشنری‌های زبان انگلیسی است.

ساختمان داده‌ی *درخت مبنایی* نشان داده شده در شکل ۱۲-۵، رشته بیت‌های ۱۰۱۱، ۱۰، ۰۱۱، ۱۰۰، و ۰ را ذخیره می‌کند. وقتی برای یک کلید $a=a_0a_1...a_p$ جستجو می‌کنیم، در یک گره با ارتفاع $i$ به سمت چپ می‌رویم اگر $a_i=0$، و به سمت راست می‌رویم اگر $a_i=1$. فرض کنید $S$ مجموعه‌ای از رشته‌های دودویی متمایزی باشد که مجموع طول آن‌ها برابر است با $n$. نشان دهید که چگونه می‌توان از یک درخت مبنایی برای مرتب‌سازی الفبایی $S$ در زمان $\theta(n)$ استفاده کرد. برای مثال شکل ۱۲-۵، خروجی مرتب‌سازی باید دنباله‌ی ۰, ۰۱۱, ۱۰, ۱۰۰, ۱۰۱۱ باشد.

## ۱۲-۳ میانگین عمق گره‌ها در یک درخت جستجوی دودویی به صورت تصادفی ساخته شده

در این مسئله اثبات می‌کنیم که میانگین عمق یک گره در یک درخت جستجوی دودویی به صورت تصادفی ساخته شده با $n$ گره، $O(\lg n)$ است. با این که این نتیجه از قضیه‌ی ۱۲-۴

ضعیف‌تر است، تکنیک استفاده شده در آن یک تشابه جالب را میان ساختن یک درخت جستجوی دودویی و اجرای RANDOMIZED-QUICKSORT از بخش ۷-۳ آشکار می‌کند.

*مجموع طول مسیر* $P(T)$ را در یک درخت دودویی $T$ به صورت مجموع عمق تمام گره‌های $x$ در درخت $T$ تعریف می‌کنیم، و عمق هر گره‌ی $x$ را با $d(x,T)$ نمایش می‌دهیم.

I. بحث کنید که متوسط عمق یک گره در درخت $T$ برابر است با

$$\frac{1}{n}\sum_{x\in T} d(x,T) = \frac{1}{n}P(T)$$

بنابراین، می‌خواهیم نشان دهیم که امیدریاضی مقدار $P(T)$ برابر است با $O(n\lg n)$.

II. فرض کنید $T_L$ و $T_R$ به ترتیب نشان‌دهنده‌ی زیردرخت‌های سمت چپ و سمت راست $T$ باشند. بحث کنید که اگر $T$ دارای $n$ گره باشد، آن گاه

$$P(T) = P(T_L) + P(T_R) + n - 1$$

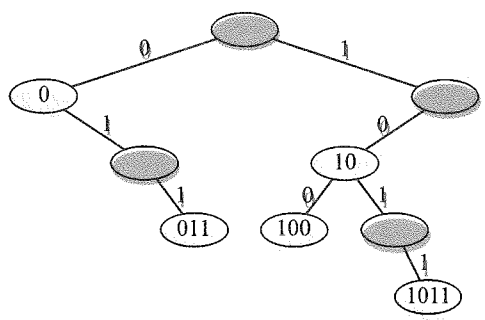
III. فرض کنید $P(n)$ نشان‌دهنده‌ی مجموع طول مسیر متوسط برای یک درخت جستجوی دودویی با $n$ گره باشد. نشان دهید که

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1)$$

IV. نشان دهید که $P(n)$ را می‌توان به صورت زیر بازنویسی کرد:

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \theta(n)$$

V. با یادآوری تحلیل جایگزینی که برای نسخه‌ی تصادفی مرتب‌سازی تصادفی در مسئله‌ی ۷-۲ انجام شد، نتیجه بگیرید که $P(n) = O(n\lg n)$.




**شکل ۱۲-۵** یک درخت مبنایی که رشته بیت‌های ۱۰۱۱، ۱۰، ۰۱۱، ۱۰۰، و ۰ را ذخیره کرده است. کلید هر گره را می‌توان با دنبال کردن مسیر از گره تا آن ریشه تعیین کرد. بنابراین احتیاجی به ذخیره‌ی کلیدها در گره‌ها نیست؛ در این جا کلیدها فقط برای راحتی نمایش داده شده‌اند. گره‌های با سایه‌ی پررنگ نشان‌دهنده‌ی گره‌هایی هستند که کلید آن‌ها در درخت نیست؛ چنین گره‌هایی فقط برای این وجود دارند که مسیر گره‌های دیگر تا ریشه را کامل کنند.

در هر یک از احضارهای بازگشتی مرتب‌سازی تصادفی، یک عنصر محور تصادفی برای تقسیم‌بندی عناصر انتخاب می‌کردیم. به طور مشابه هر گره‌ی یک درخت جستجوی دودویی، مجموعه‌ی عناصری که درون زیردرخت مربوط به همان عنصر قرار دارند را به دو بخش تقسیم می‌کند.

VI. یک پیاده‌سازی از مرتب‌سازی تصادفی ارائه کنید که در آن مقایسه‌هایی که برای مرتب‌سازی عناصر انجام می‌شوند دقیقاً مشابه مقایسه‌هایی باشند که برای درج عناصر در یک درخت جستجوی دودویی انجام می‌شود. (ترتیب انجام مقایسه‌ها ممکن است متفاوت باشد، ولی مجموعه‌ی مقایسه‌ها باید دقیقاً یکسان باشد.)

## ۴-۱۲ تعداد درخت‌های دودویی مختلف

فرض کنید $b_n$ نشان دهنده‌ی تعداد درخت‌های دودویی مختلفی باشد که با $n$ گره می‌توان ساخت. در این مسئله، یک فرمول برای $b_n$ و یک تقریب حدی برای آن پیدا خواهیم کرد.

I. نشان دهید که $b_0 = 1$، و برای $n \geq 1$،

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}$$

II. با ارجاع به مسئله‌ی ۴-۵ 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 که $B(x) = xB(x)^2 + 1$، و بنابراین یک راه برای نشان دادن $B(x)$ به صورت زیر است:

$$B(x) = \frac{1}{2x}(1 - \sqrt{1-4x})$$

*بسط تیلور* (Taylor expansion) تابع $f(x)$ حول نقطه‌ی $x = a$ به صورت زیر تعریف می‌شود:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^x$$

که در آن $f^{(k)}(x)$ نشان دهنده‌ی $k$امین مشتق تابع $f$ در نقطه‌ی $x = a$ است.

III. استفاده از بسط تیلور $\sqrt{1-4x}$ حول نقطه‌ی $x = 0$، نشان دهید که

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

که همان عدد $n$ام کاتالان (Catalan number) است. (اگر می‌خواهید، می‌توانید به جای از بسط تیلور، از تعمیم بسط دوجمله‌ای (پ-۴) به نماهای ناصحیح $n$ استفاده کنید، که در آن برای هر عدد حقیقی $n$ و هر عدد صحیح $k$، اگر $k \geq 0$، مقدار $\binom{n}{k}$ را به صورت

$n(n-1)(n-k+1)/k!$، و در غیر این صورت ۰ تعریف می‌کنیم.)

IV. نشان دهید

$$b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}}(1+O(1/n))$$


# درختان قرمز- سیاه

در فصل ۱۲ نشان دادیم که می‌توان تمام اعمال اصلی مجموعه‌های پویا – مانند SEARCH، PREDECESSOR، SUCCESSOR، MINIMUM، MAXIMUM، INSERT، و DELETE – را بر روی یک درخت جستجوی دودویی با ارتفاع $h$ در زمان $O(h)$ پیاده‌سازی کرد. پس این اعمال در صورتی سریع هستند که ارتفاع درخت کم باشد. ولی اگر ارتفاع درخت زیاد باشد کارایی آن بهتر از یک لیست پیوندی نخواهد بود. درختان قرمز-سیاه یکی از رویکردهایی است که به کمک آن می‌توان درختان جستجو را «متوازن» کرد، که در این صورت تضمین می‌شود اعمال اصلی مجموعه‌های پویا در بدترین حالت در زمان $O(\lg n)$ اجرا می‌شوند.

## ۱-۱۳ خصوصیات درختان قرمز- سیاه

یک درخت *قرمز-سیاه*، یک درخت جستجوی دودویی است با یک بیت حافظه‌ی اضافی در هر گره: *رنگ* آن گره، که یا می‌تواند قرمز (RED) باشد و یا سیاه (BLACK). درختان قرمز-سیاه با محدود کردن رنگ گره‌ها در هر مسیر از ریشه تا برگ‌ها تضمین می‌کنند که هیچ مسیری وجود ندارد که طول آن بیشتر از دو برابر مسیری دیگر باشد، و بدین صورت درخت تقریباً *متوازن* خواهد بود.

اکنون هر گره‌ی درخت حاوی فیلدهای *color*، *key*، *left*، *right*، و $p$ خواهد بود. اگر یک فرزند یا پدر یک گره وجود نداشته باشد، اشاره‌گر مربوطه در آن فیلد حاوی مقدار NIL است. با این مقادیر NIL به صورت اشاره‌گرهایی به گره‌های خارجی درخت (برگ‌ها) برخورد می‌کنیم، و گره‌های معمولی درخت، گره‌های داخلی خواهند بود.

یک درخت قرمز- سیاه، یک درخت جستجوی دودویی است که خصوصیات زیر را که

*خصوصیات درختان قرمز- سیاه* هستند، ارضا می‌کند:

۱. هر گره یا قرمز است و یا سیاه.
۲. ریشه سیاه است.
۳. تمام برگ ( NIL )ها سیاه هستند.
٤. اگر یک گره قرمز باشد آن گاه هر دو فرزند آن سیاه هستند.
۵. برای هر گره، تمام مسیرها از آن گره تا برگ‌های نوه‌ی آن گره حاوی تعداد مساوی گره‌ی سیاه است.

شکل ۱۳-۱(الف) مثالی از یک درخت قرمز- سیاه را نشان می‌دهد.

برای سادگی در برخورد با مقادیر مرزی در کد درختان قرمز-سیاه، از یک مقدار نگهبان برای نمایش NIL استفاده خواهیم کرد (بخش ۱۰-۲ را ببینید). برای یک درخت قرمز-سیاه $T$، مقدار نگهبان $T.nil$ یک شیئ با همان فیلدهایی است که یک گره‌ی معمولی درخت دارد. مقدار فیلد *color* آن BLACK است، و فیلدهای دیگر آن — *left*، *right*، $p$، و *key* — می‌توانند حاوی مقادیر دلخواه باشد. همان طور که شکل ۱۳-۱(ب) نشان می‌دهد، تمام اشاره‌گرهای NIL با اشاره‌گرهایی به مقدار نگهبان $T.nil$ جایگزین شده‌اند.

از مقدار نگهبان استفاده می‌کنیم تا بتوانیم با یک فرزند NIL از یک گره‌ی $x$ به صورت یک گره‌ی معمولی که پدر آن $x$ است برخورد کنیم. با این که می‌توانستیم برای هر NIL یک گره‌ی نگهبان جداگانه تعریف کنیم تا پدر هر NIL مشخص باشد، برای جلوگیری از اتلاف حافظه از این کار صرف نظر می‌کنیم. به جای این کار، از یک نگهبان $T.nil$ برای نشان دادن تمام NIL ها — برگ‌ها و پدر ریشه — استفاده می‌کنیم. مقدار فیلدهای $p$، *left*، *right*، و *key* گره‌ی نگهبان مهم نیستند، ولی با این حال ممکن است هنگام اجرای یک رویه برای راحتی به آن‌ها مقدار دهیم.

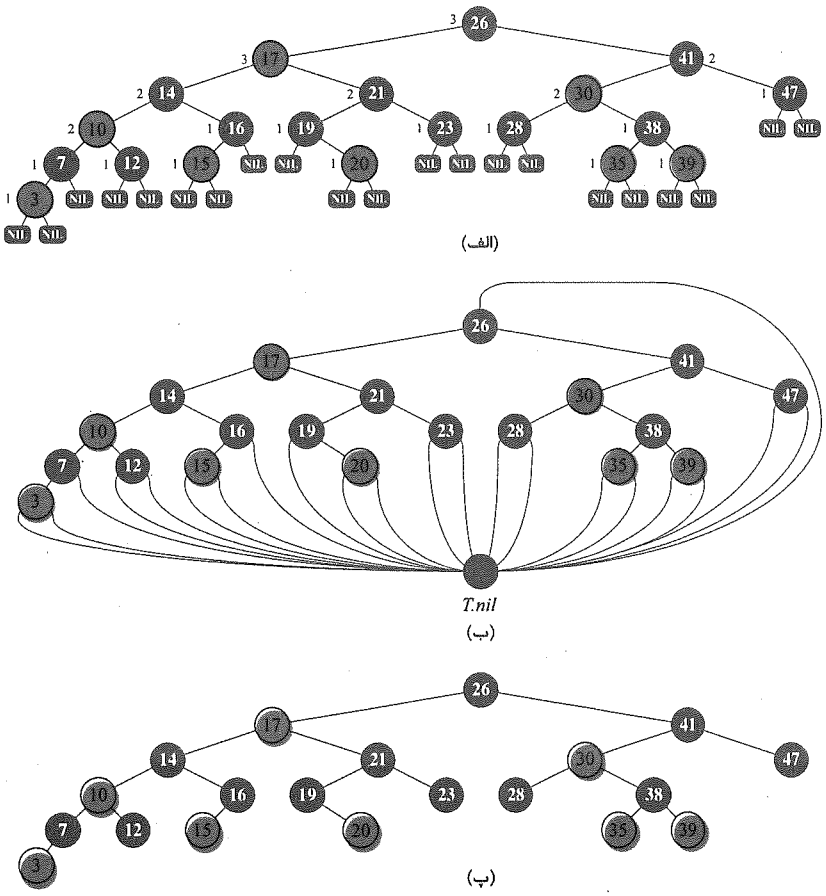
به طور کلی توجه خود را بر روی گره‌های داخلی یک درخت قرمز-سیاه متمرکز می‌کنیم، چرا که آن‌ها حاوی مقادیر کلید هستند. در ادامه‌ی این فصل، همان طور که در شکل ۱۳-۱(ج) نشان داده شد، از کشیدن برگ‌ها در درخت‌های قرمز-سیاه صرف نظر می‌کنیم.

به تعداد گره‌های سیاه در هر مسیری از یک گره‌ی $x$ (که شامل خود گره نمی‌شود) تا یک برگ در زیردرخت آن گره، *ارتفاع سیاه* آن گره گفته و آن را با $bh(x)$ نشان می‌دهیم. طبق خصوصیت ۵ ارتفاع سیاه یک گره خوش تعریف است، چرا که تمام مسیرهای پایینی از گره تعداد ثابتی گره‌ی سیاه دارند. ارتفاع سیاه یک درخت قرمز-سیاه را به صورت ارتفاع سیاه ریشه‌ی آن تعریف می‌کنیم.

لم زیر نشان می‌دهد که چرا درختان قرمز-سیاه، درختان جستجوی دودویی خوبی هستند.

**لم ۱-۱۳** ارتفاع یک درخت قرمز-سیاه با $n$ گره‌ی داخلی حداکثر $2\lg(n+1)$ است.

*اثبات* با نشان دادن این که زیردرخت گره‌ی $x$ حداقل $2^{bh(x)}-1$ گره‌ی داخلی دارد شروع می‌کنیم. این کار را به کمک استقرا بر روی ارتفاع $x$ انجام می‌دهیم. اگر ارتفاع $x$ برابر ۰ باشد، آن گاه $x$ یک




**شکل ۱۳-۱** یک درخت قرمز- سیاه که در آن گره‌های سیاه با رنگ سیاه و گره‌های قرمز با رنگ قرمز مشخص شده‌اند. هر گره در یک درخت قرمز-سیاه یا قرمز است و یا سیاه، هر دو فرزند یک گره‌ی قرمز، سیاه هستند، و تمام مسیرهای ساده از یک گره تا هر یک از برگ‌های نوه‌ی آن گره حاوی تعداد مساوی گره‌های سیاه است. (الف) تمام برگ‌ها، که به صورت NIL نشان داده شده‌اند، سیاه هستند. هر گره‌ی غیر برگ با ارتفاع سیاه آن علامت‌گذاری شده است؛ ارتفاع سیاه NIL ها ۰ است. (ب) همان درخت قرمز-سیاه که در آن تمام NIL ها با یک مقدار نگهبان $T.nil$ جایگزین شده‌اند که همیشه سیاه است، و همچنین ارتفاع سیاه گره‌ها حذف شده است. پدر ریشه هم همان مقدار نگهبان است. (پ) همان درخت قرمز-سیاه که در آن برگ‌ها و پدر ریشه حذف شده‌اند. از این شکل نمایش در ادامه‌ی این فصل استفاده خواهیم کرد.

برگ ( $T.nil$) بوده و زیردرخت $x$ حاوی $۲^{bh(x)}-۱=۲^{۰}-۱=۰$ گره‌ی داخلی است. برای گام استقرا یک گره‌ی $x$ را در نظر بگیرید که یک گره‌ی داخلی است با دو فرزند و ارتفاع مثبت. هر فرزند دارای ارتفاع سیاه $bh(x)$ و یا $bh(x)-۱$ است، بسته به این که رنگ آن به ترتیب قرمز باشد یا سیاه. از آن جایی که ارتفاع یک فرزند $x$ از ارتفاع خود $x$ کم‌تر است، می‌توانیم از فرض استقرا استفاده کنیم تا

نتیجه بگیریم که هر فرزند حداقل $2^{bh(x)}-1$ گره‌ی داخلی دارد. بنابراین، زیردرخت گره‌ی $x$ حداقل شامل $2^{bh(x)}-1=1+(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)$ گره‌ی داخلی است، که ادعای بالا را ثابت می‌کند.

برای کامل کردن اثبات لم فرض کنید $h$ ارتفاع درخت باشد. طبق خصوصیت ۴ حداقل نیمی از گره‌ها در هر مسیر ساده‌ای از ریشه تا یک برگ، غیر از خود ریشه، باید سیاه باشند. در نتیجه ارتفاع سیاه ریشه باید حداقل $h/2$ باشد؛ بنابراین،

$$n \geq 2^{h/2}-1$$

با بردن ۱ به سمت چپ نامساوی و لگاریتم گرفتن از هر دو طرف، نتیجه می‌گیریم که $\lg(n+1) \geq h/2$، یا $h \geq 2\lg(n+1)$. ■

یک نتیجه‌ی مستقیم این لم این است که اعمال مجموعه‌های پویای SEARCH، MINIMUM، MAXIMUM، SUCCESSOR، و PREDECESSOR را می‌توان در زمان $O(\lg n)$ بر روی درخت‌های قرمز-سیاه پیاده‌سازی کرد، چرا که آن‌ها بر روی یک درخت جستجوی دودویی با ارتفاع $h$ در زمان $O(h)$ اجرا می‌شوند (همان طور که در فصل ۱۲ نشان داده شد)، و هر درخت قرمز-سیاه با $n$ گره، یک درخت جستجو با ارتفاع $O(\lg n)$ است. (البته در الگوریتم‌های فصل ۱۲، اشاره‌گرهای NIL باید با $T.nil$ جایگزین شوند.) با این که الگوریتم‌های TREE-INSERT و TREE-DELETE از فصل ۱۲، وقتی یک درخت قرمز-سیاه را به عنوان ورودی دریافت کنند، در زمان $O(\lg n)$ اجرا می‌شوند، آن‌ها همان اعمال درج و حذف مورد نظر ما بر روی یک درخت قرمز-سیاه نیستند، چرا که تضمین نمی‌کنند که درخت جستجوی دودویی اصلاح شده همچنان یک درخت قرمز-سیاه باشد. با این حال در بخش‌های ۱۳-۳ و ۱۳-۴ خواهیم دید که می‌توان این اعمال را هم بر روی یک درخت قرمز-سیاه در زمان $O(\lg n)$ پیاده‌سازی کرد.

## تمرین‌ها

۱۳-۱-۱ به سبک شکل ۱۳-۱(الف)، درخت جستجوی دودویی کامل با ارتفاع ۳ را بر روی کلیدهای $\{1,2,\ldots,15\}$ بکشید. برگ‌های NIL و رنگ گره‌ها را به سه شکل مختلف طوری اضافه کنید که ارتفاع سیاه درخت‌های قرمز-سیاه به وجود آمده ۲، ۳، و ۴ باشد.

۱۳-۱-۲ درخت قرمز-سیاهی را بکشید که پس از فراخوانی TREE-INSERT بر روی درخت شکل ۱۳-۱ با کلید ۳۶ پدید می‌آید. اگر گره‌ی درج شده با قرمز رنگ‌آمیزی شود، درخت به وجود آمده یک درخت قرمز-سیاه است؟ اگر با سیاه رنگ‌آمیزی شود چطور؟

۱۳-۱-۳ فرض کنید یک *درخت قرمز-سیاه نااکید* (relaxed red-black tree)، یک درخت جستجوی دودویی باشد که خصوصیات ۱، ۳، ۴، و ۵ درختان قرمز-سیاه را ارضا می‌کند. یعنی ریشه ممکن است قرمز باشد یا سیاه. یک درخت قرمز-سیاه نااکید $T$ را در نظر بگیرید که ریشه‌ی آن قرمز است. اگر ریشه‌ی $T$ را با سیاه رنگ‌آمیزی کنیم، ولی هیچ تغییر دیگری

در آن ایجاد نکنیم، آیا درخت حاصل یک درخت قرمز-سیاه خواهد بود؟

۴-۱-۱۳ فرض کنید در یک درخت قرمز- سیاه، تمام گره‌های قرمز را در پدر سیاه آن‌ها «جذب» (absorb) کنیم، به طوری که فرزندان گره‌ی قرمز، تبدیل شوند به فرزندان پدر سیاه آن‌ها. (از مقدار کلیدهای این گره‌ها صرف نظر می‌کنیم.) درجه‌های ممکن یک گره‌ی سیاه بعد از این که تمام فرزندان قرمز آن جذب شدند چیست؟ در مورد عمق برگ‌ها در مورد درخت حاصل چه می‌توان گفت؟

۵-۱-۱۳ نشان دهید که بلندترین مسیر ساده‌ی ممکن از یک گره‌ی $x$ در یک درخت قرمز-سیاه تا یک برگ نوه‌ی همان گره حداکثر برابر است با دو برابر کوتاه‌ترین مسیر ساده از گره‌ی $x$ تا یک برگ نوه‌ی $x$.

۶-۱-۱۳ بیشترین تعداد ممکن گره‌های داخلی در یک درخت قرمز-سیاه با ارتفاع سیاه $k$ چقدر است؟ کم‌ترین تعداد ممکن چقدر است؟

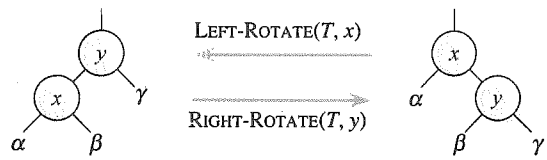
۷-۱-۱۳ یک درخت قرمز- سیاه با $n$ گره را توصیف کنید که بیشترین نسبت ممکن گره‌های قرمز داخلی به گره‌های سیاه داخلی را دارد. این نسبت چند است؟ چه درختی کم‌ترین نسبت ممکن را دارد، و این نسبت چند است؟

## ۱۳-۲ دوران‌ها

اعمال درختان جستجوی جستجوی TREE-INSERT و TREE-DELETE، وقتی بر روی یک درخت قرمز- سیاه با $n$ کلید اجرا می‌شوند، به زمان $O(\lg n)$ نیاز دارند. از آن جایی که آن‌ها تغییراتی در درخت ایجاد می‌کنند، ممکن است درخت حاصل خصوصیات درختان قرمز-سیاه را که در بخش ۱۳-۱ توصیف شد، نقض کنند. برای بدست آوردن دوباره‌ی این خصوصیات باید در رنگ بعضی از گره‌ها در درخت، و در ساختار بعضی از اشاره‌گرها تغییراتی ایجاد کنیم.

ساختار اشاره‌گرها را به کمک *دوران* (rotation) تغییر می‌دهیم، که یک عملیات محلی در یک درخت جستجوی دودویی است که خصوصیت درختان جستجوی دودویی را حفظ می‌کند. شکل ۱۳-۲ دو نوع دوران را نشان می‌دهد: دوران چپ و دوران راست. وقتی یک دوران چپ بر روی یک گره‌ی $x$ در درخت انجام می‌دهیم، فرض می‌کنیم که فرزند سمت راست آن، $y$، برابر با $T.nil$ نیست؛ $x$ می‌تواند هر گره‌ای در درخت باشد که فرزند سمت راست آن $T.nil$ نیست. دوران چپ، یک چرخش «حول» پیوند بین $x$ و $y$ ایجاد می‌کند. در این عمل $y$ ریشه‌ی جدید زیردرخت می‌شود، $x$ فرزند سمت راست $y$، و فرزند سمت چپ $y$، فرزند سمت راست $x$ خواهد شد.

در شبه‌کد LEFT-ROTATE فرض می‌شود که $x.right \neq T.nil$، و همچنین پدر ریشه $T.nil$ است.




**شکل ۱۳-۲** عملیات دوران بر روی یک درخت جستجوی دودویی. عملیات LEFT-ROTATE $(T, x)$ با تغییر تعداد ثابتی از اشاره‌گرها، ساختار دو گره‌ی سمت راست را تبدیل می‌کند به ساختار دو گره‌ی سمت چپ. ساختار سمت راست را می‌توان به کمک عملیات معکوس، یعنی RIGHT-ROTATE $(T, y)$، به ساختار سمت چپ تبدیل کرد. حروف $\alpha$، $\beta$، و $\gamma$ نشان‌دهنده‌ی زیردرخت‌های دلخواه هستند. یک عملیات دوران خصوصیات درختان جستجوی دودویی را حفظ می‌کند: کلیدهای درون $\alpha$ بزرگ‌تر از $x.key$ هستند، و $x.key$ از کلیدهای درون $\beta$، و کلیدهای درون $\beta$ از $y.key$، و $y.key$ از کلیدهای درون $\lambda$ بزرگ‌تر است.

```
LEFT-ROTATE(T, x)
1  y = x.right            // Set y.
2  x.right = y.left       // Turn y's left subtree into x's right subtree.
3  if y.left ≠ T.nil
4      y.left.p = x
5  y.p = x.p              // Link x's parent to y.
6  if x.p == T.nil
7      T.root = y
8  elseif x == x.p.left
9      x.p.left = y
10 else x.p.right == y
11 y.left = x             // Put x on y's left.
12 x.p = y.
```

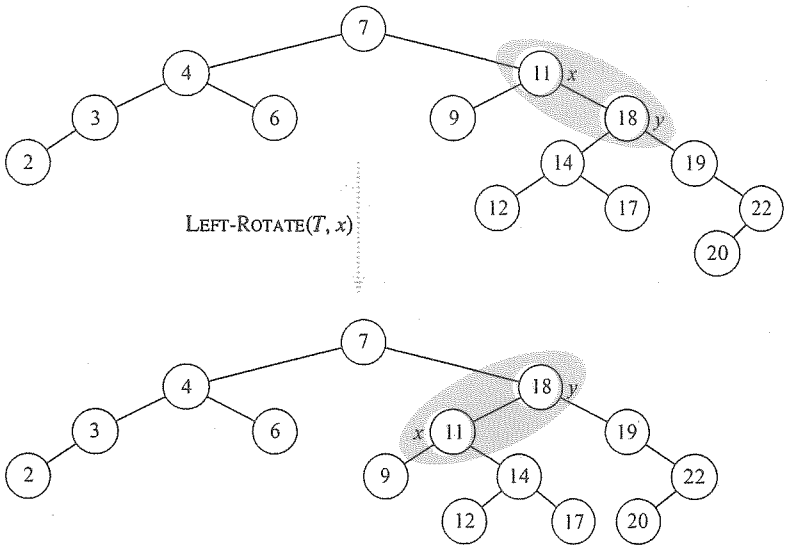
شکل ۱۳-۳ مثالی از تغییرات انجام شده توسط LEFT-ROTATE بر روی یک درخت جستجوی دودویی را نشان می‌دهد. کد RIGHT-ROTATE هم به صورت مشابه پیاده‌سازی می‌شود. LEFT-ROTATE و RIGHT-ROTATE در زمان $O(1)$ اجرا می‌شوند. در یک دوران، فقط اشاره‌گرها تغییر می‌کنند؛ تمام فیلدهای دیگر بدون تغییر باقی می‌مانند.

## تمرین‌ها

۱-۲-۱۳ برای RIGHT-ROTATE شبه‌کد بنویسید.

۲-۲-۱۳ بحث کنید که در هر درخت جستجوی دودویی با $n$ گره، دقیقاً $n-1$ دوران ممکن وجود دارد.

۳-۲-۱۳ در شکل ۱۳-۲ فرض کنید $a$، $b$، و $c$ گره‌هایی دلخواه به ترتیب در زیردرخت‌های $\alpha$، $\beta$، و $\lambda$ باشند. وقتی یک دوران چپ بر روی گره‌ی $x$ در شکل انجام می‌شود، عمق گره‌های $a$، $b$، و $c$ چگونه تغییر می‌کند؟




**شکل ۱۳-۳** یک مثال از تغییرات حاصل از LEFT-ROTATE$(T, x)$ بر روی یک درخت جستجوی دودویی. پیمایش میان‌ترتیبی بر روی درخت ورودی و درخت اصلاح شده، لیست یکسانی از کلیدها ایجاد می‌کنند.

۱۳-۲-۴ نشان دهید که هر درخت جستجوی دودویی دلخواه با $n$ گره را می‌توان با $O(n)$ دوران، به هر درخت جستجوی دودویی دلخواه دیگر با همان $n$ گره تبدیل کرد. (*راهنمایی:* ابتدا نشان دهید که برای تبدیل درخت به یک زنجیره‌ی متمایل به راست، $n-1$ دوران راست کافی است.)

★ ۱۳-۲-۵ می‌گوییم یک درخت جستجوی دودویی $T_1$ را می‌توان به یک درخت جستجوی دودویی $T_2$ *تبدیل راست* (right-convert) کرد اگر بتوان $T_2$ را از دنباله‌ای از RIGHT-ROTATEها بر روی $T_1$ به دست آورد. مثالی از درختان $T_1$ و $T_2$ بدهید که نتوان $T_1$ را تبدیل راست به $T_2$ کرد. سپس نشان دهید که اگر یک درخت $T_1$ را بتوان به درخت $T_2$ تبدیل راست کرد، این کار را می‌توان با $O(n^2)$ فراخوانی RIGHT-ROTATE انجام داد.

## ۱۳-۳ درج

درج یک گره در یک درخت قرمز-سیاه با $n$ گره را می‌توان در زمان $O(\lg n)$ انجام داد. از یک نسخه‌ی رویه‌ی TREE-INSERT (بخش ۱۲-۳) با مقداری تغییر برای درج گره‌ی $z$ در درخت $T$ استفاده می‌کنیم، و در این کار با درخت مانند یک درخت جستجوی دودویی معمولی برخورد خواهیم کرد. سپس $z$ را با قرمز رنگ آمیزی می‌کنیم. بعد از آن برای تضمین این که خصوصیات درختان

قرمز–سیاه از بین نمی‌روند، از یک رویه‌ی کمکی به نام RB-INSERT-FIXUP برای تغییر رنگ گره‌ها و انجام دوران‌های مورد نیاز استفاده می‌کنیم. فراخوانی رویه‌ی RB-INSERT ($T$, $z$)، گره‌ی $z$ را، با فرض این که فیلد $key$ آن قبلا مقداردهی شده است، درون درخت قرمز–سیاه $T$ درج می‌کند.

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == T.nil
10        T.root = z
11  elseif z.ket < y.key
12        y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

چهار تفاوت بین رویه‌های TREE-INSERT و RB-INSERT وجود دارد. اول، تمام NILها در TREE-INSERT با $T.nil$ جایگزین شده‌اند. دوم، $z.left$ و $z.right$ را در خطوط ۱۴–۱۵ رویه‌ی RB-INSERT با $T.nil$ مقداردهی می‌کنیم، برای این که ساختار مناسب درخت را حفظ کنیم. سوم، $z$ را در خط ۱۶ با قرمز رنگ‌آمیزی می‌کنیم. چهارم، از آن جایی که رنگ‌آمیزی $z$ با قرمز ممکن است باعث شود که یکی از خصوصیات درخت قرمز–سیاه نقض شود، در خط ۱۷ رویه‌ی RB-INSEERT، رویه‌ی RB-INSERT-FIXUP ($T$, $z$) را فراخوانی می‌کنیم تا خصوصیات درختان قرمز–سیاه را بازسازی کنیم.

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2        if z.p == z.p.p.left
 3              y = z.p.p.right
 4              if y.color == RED
 5                    z.p.color = BLACK          // Case 1
 6                    y.color = BLACK            // Case 1
 7                    z.p.p.color = RED          // Case 1
 8                    z = z.p.p                  // Case 1
 9              else if z == z.p.right
```

```
10            z = z.p                          // Case 2
11            LEFT-ROTATE(T, z)                // Case 2
12          z.p.color = BLACK                  // Case 3
13          z.p.p.color = RED                  // Case 3
14          RIGHT-ROTATE(T, z.p.p)             // Case 3
15    else (same as then clause with "right" and "left" exchanged)
16  T.root.color = BLACK
```

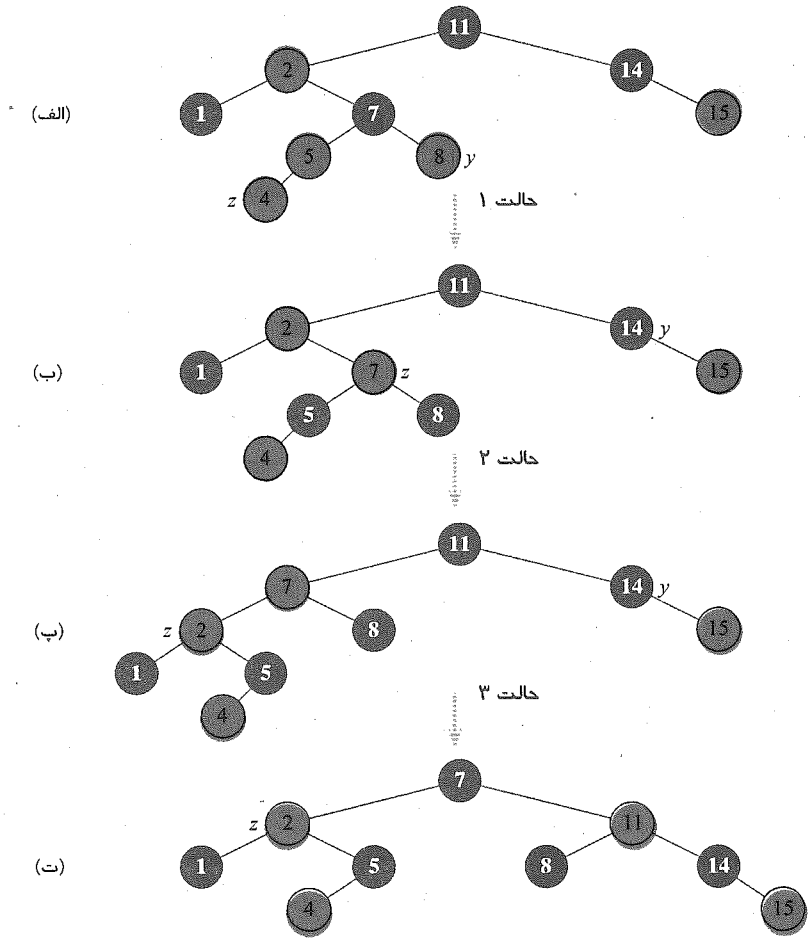
برای درک این که RB-INSERT-FIXUP چگونه کار می‌کند، بررسی کد را به سه مرحله‌ی اصلی تقسیم می‌کنیم. اول، تعیین می‌کنیم که وقتی در RB-INSERT، گره‌ی $z$ درج شده و با قرمز رنگ آمیزی می‌شود، چه خصوصیاتی از درختان قرمز- سیاه ممکن است نقض شود. دوم، هدف کلی حلقه‌ی while را در خطوط ۱-۱۵ بررسی می‌کنیم. در نهایت، هر یک از سه حالتی را که حلقه‌ی while به آن‌ها تقسیم شده است، بررسی می‌کنیم تا ببینیم آن‌ها چگونه رسیدن به هدف کلی حلقه‌ی while را میسر می‌کنند.[1] شکل ۱۳-۴ نحوه‌ی عملکرد RB-INSERT-FIXUP را بر روی یک درخت قرمز- سیاه نمونه نشان می‌دهد.

کدام یک از خصوصیات درختان قرمز-سیاه در فراخوانی RB-INSERT-FIXUP ممکن است نقض شوند؟ خصوصیت ۱ مطمئنا باقی خواهد ماند، همین طور خصوصیت ۳، چرا که هر دو فرزند گره‌ی قرمز تازه درج شده، مقدار نگهبان $T.nil$ هستند. خصوصیت ۵ هم که می‌گوید تعداد گره‌های سیاه در هر مسیری از یک گره باید برابر باشد، برقرار خواهد بود، زیرا گره‌ی $z$ جایگزین نگهبان (سیاه) می‌شود، و $z$ یک گره‌ی قرمز با فرزندان سیاه است. بنابراین، تنها خصوصیاتی که ممکن است نقض شوند، خصوصیت ۲ است که می‌گوید ریشه باشد سیاه باشد، و خصوصیت ۴ که می‌گوید یک گره‌ی قرمز نمی‌تواند فرزند قرمز داشته باشد. هر دو نقض احتمالی به خاطر این است که $z$ با قرمز رنگ آمیزی می‌شود. خصوصیت ۲ در صورتی نقض می‌شود که $z$ ریشه باشد، و خصوصیت ۴ در صورتی نقض می‌شود که پدر $z$ قرمز باشد. شکل ۱۳-۴(الف) یک حالت نقض خصوصیت ۴ را بعد از درج $z$ نشان می‌دهد.

حلقه‌ی while در خطوط ۱-۱۵ سه ثابت حلقه‌ی زیر را هنگام آغاز هر یک از تکرارهای حلقه حفظ می‌کند:

I. گره‌ی $z$ قرمز است.
II. اگر $z.p$ ریشه باشد، آن گاه $z.p$ سیاه است.
III. اگر نقضی از خصوصیات درختان قرمز-سیاه وجود داشته باشد، حداکثر یک نقض وجود خواهد داشت، و این نقض، یا نقض خصوصیت ۲ است و یا نقض خصوصیت ۴. اگر خصوصیت ۲ نقض شده باشد، به این خاطر است که $z$ ریشه، و رنگ آن قرمز است. اگر خصوصیت ۴ نقض شده باشد، بدین خاطر است که $z$ و $z.p$ هر دو قرمز هستند.

[1] حالت ۳ حالت ۲ را هم شامل می‌شود، و بنابراین این دو حالت کاملاً مستقل نیستند.




**شکل ۱۳-٤** عملیات RB-INSERT-FIXUP. (الف) یک گره‌ی $z$ بعد از درج شدن. از آن جایی که $z$ و پدر آن هر دو قرمز هستند، خصوصیت ۴ نقض می‌شود. چون عموی $z$، یعنی $y$ قرمز است، حالت ۱ در حلقه‌ی while رخ می‌دهد. گره‌ها دوباره رنگ آمیزی می‌شوند و $z$ در درخت به سمت بالا حرکت می‌کند، که ما را به درخت نشان داده شده در قسمت (ب) می‌رساند. دوباره $z$ و پدر آن، هر دو قرمز هستند، ولی عموی $z$ سیاه است. از آن جایی که $z$ فرزند سمت راست $z.p$ است، حالت ۲ اتفاق می‌افتد. یک دوران چپ انجام می‌شود و درخت حاصل در قسمت (پ) نشان داده شده است. اکنون $z$ فرزند سمت چپ پدر خود است و حالت ۳ را خواهیم داشت. یک دوران راست، درخت قسمت (ت) را نتیجه می‌دهد، که یک درخت قرمز-سیاه با خصوصیات لازم است.

هدف قسمت III، که با نقض خصوصیات درختان قرمز-سیاه سروکار دارد، بیشتر این است که نشان دهد RB-INSERT-FIXUP خصوصیات درختان قرمز-سیاه را بازسازی می‌کند، در حالی که قسمت‌های I و II برای این هستند که به ما کمک کنند موقعیت‌های مختلف درون کد را بهتر درک کنیم. از آن جایی که ما بر روی گره‌ی $z$ و گره‌های اطراف آن در درخت تمرکز می‌کنیم، خوب است

که از قسمت I بدانیم که $z$ قرمز است. از قسمت II برای این استفاده می‌کنیم که نشان دهیم $z.p.p$ وجود دارد، وقتی که در خطوط ۲، ۳، ۷، ۸، ۱۳، و ۱۴ به آن اشاره می‌کنیم.

به یاد بیاورید که باید نشان دهیم ثابت حلقه قبل از شروع اولین تکرار حلقه برقرار است، هر تکرار حلقه ثابت حلقه را برقرار نگه می‌دارد، و ثابت حلقه یک خصوصیت مفید بعد از پایان حلقه به ما می‌دهد.

با بحث‌های شروع و پایان حلقه آغاز می‌کنیم. سپس با جزئیات بیشتر بررسی می‌کنیم که بدنه‌ی حلقه چگونه کار می‌کند، و بحث می‌کنیم که ثابت حلقه در هر تکرار حلقه برقرار باقی می‌ماند. در عین حال، نشان خواهیم داد که دو نتیجه‌ی ممکن برای هر تکرار حلقه وجود دارد: اشاره‌گر $z$ در درخت به سمت بالا حرکت می‌کند، و یا دوران‌هایی انجام خواهد شد و حلقه پایان خواهد یافت.

- **آغاز:** قبل از اولین تکرار حلقه، با یک درخت قرمز-سیاه بدون هیچ نقضی شروع کردیم، و یک گره‌ی قرمز $z$ را به آن اضافه کردیم. نشان خواهیم داد که هنگامی که RB-INSERT-FIXUP فراخوانی می‌شود، تمام قسمت‌های ثابت حلقه برقرار است:
  - I. وقتی RB-INSERT-FIXUP فراخوانی می‌شود، $z$ گره‌ی قرمزی است که به درخت اضافه شده است.
  - II. اگر $z.p$ ریشه باشد، آن گاه $z$ سیاه بوده است و قبل از فراخوانی RB-INSERT-FIXUP تغییری نکرده است.
  - III. قبلا دیدیم که خصوصیات ۱، ۳، و ۵ هنگام فراخوانی RB-INSERT-FIXUP برقرار هستند.

  اگر خصوصیت ۲ نقض شده باشد، آن گاه ریشه باید گره‌ی تازه اضافه شده‌ی $z$ باشد، که تنها گره‌ی داخلی درخت است. از آن جایی که پدر و هر دو فرزند $z$ گره‌ی نگهبان است، که سیاه است، خصوصیت ۴ نقض نشده است. بنابراین این نقض خصوصیت ۲ تنها نقض خصوصیات درختان قرمز سیاه در کل درخت است.

اگر خصوصیت ۴ نقض شده باشد، آن گاه از آن جایی که فرزندان $z$ سیاه هستند و درخت قبل از اضافه شدن $z$ هیچ یک از خصوصیات درختان قرمز-سیاه را نقض نمی‌کرد، این نقض باید بدین خاطر باشد که $z$ و $z.p$ هر دو قرمز هستند. به علاوه هیچ نقض دیگری از خصوصیات درختان قرمز- سیاه وجود ندارد.

- **پایان:** وقتی حلقه پایان می‌یابد، بدین خاطر پایان می‌یابد که $z.p$ سیاه است. (اگر $z$ ریشه باشد، آن گاه $z.p$ مقدار نگهبان است، و سیاه.) بنابراین در پایان حلقه خصوصیت ۴ نقض نشده است. طبق ثابت حلقه، تنها خصوصیتی که ممکن است نقض شده باشد، خصوصیت ۲ است. ولی خط ۱۶ این خصوصیت را هم برقرار می‌سازد. پس وقتی RB-INSERT-FIXUP پایان می‌یابد، تمام خصوصیات درختان قرمز-سیاه برقرار هستند.

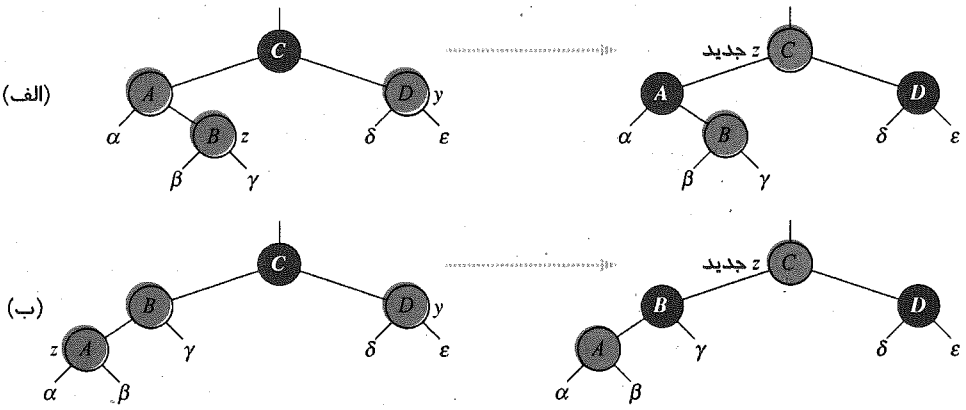
- **ادامه:** در واقع در حلقه‌ی while، شش حالت مختلف هست که باید بررسی شود، ولی سه تا از آن‌ها با سه تای دیگر متقارن هستند، بسته به این که $z.p$ فرزند سمت راست $z.p.p$ است یا

فرزند سمت چپ آن، که در خط ۲ تعیین می‌شود. در این جا فقط کد حالتی را که $z.p$ فرزند سمت چپ است نوشته‌ایم. گره‌ی $z.p.p$ وجود دارد، چرا که طبق قسمت II ثابت حلقه، اگر $z.p$ ریشه باشد، آن گاه $z.p$ سیاه است. از آن جایی که فقط در حالتی وارد تکرار حلقه می‌شویم که $z.p$ قرمز باشد، پس $z.p$ نمی‌تواند ریشه باشد. بنابراین $z.p.p$ وجود دارد.

حالت ۱ به وسیله‌ی رنگ برادر پدر $z$، و یا «عموی» $z$ از حالت‌های ۲ و ۳ جدا می‌شود. خط ۳ اشاره‌گر $y$ را برابر عموی $z$، یا $z.p.p.right$ قرار می‌دهد، و یک تست در خط ۴ انجام می‌شود. اگر $y$ قرمز باشد حالت ۱ اجرا می‌شود. در غیر این صورت کنترل به حالت ۲ و ۳ می‌رود. در هر سه حالت پدر بزرگ $z$، یعنی $z.p.p$ سیاه است، چرا که پدر آن یعنی $z.p$ قرمز است، و خصوصیت ۴ فقط بین $z$ و $z.p$ نقض می‌شود.

### حالت ۱: عموی $z$ قرمز است.

شکل ۱۳-۵ حالت ۱ را نشان می‌دهد (خطوط ۵-۸). حالت ۱ زمانی اجرا می‌شود که $z.p$ و $y$ هر دو قرمز باشند. از آن جایی که $z.p.p$ سیاه است، می‌توانیم $z.p$ و $y$ را سیاه کنیم، که مشکل قرمز بودن $z$ و $z.p$ حل می‌شود، و می‌توانیم $z.p.p$ را قرمز کنیم و خصوصیت ۵ را حفظ کنیم. سپس حلقه‌ی while را با $z.p.p$ به عنوان گره‌ی جدید $z$ تکرار می‌کنیم. اشاره‌گر $z$ دو سطح در درخت به سمت بالا حرکت می‌کند.




**شکل ۱۳-۵** حالت ۱ از رویه‌ی RB-INSERT-FIXUP. خصوصیت ۴ نقض می‌شود، چرا که $z$ و پدر آن $z.p$ هر دو قرمز هستند. در هر دو حالتی که (الف) $z$ فرزند سمت راست باشد یا (ب) فرزند سمت چپ، کار یکسانی انجام می‌شود. هر یک از زیردرخت‌های $\alpha$، $\beta$، $\gamma$، $\delta$، و $\varepsilon$ ریشه‌ی سیاه دارند، و ارتفاع سیاه همه‌ی آن‌ها با هم برابر است. کد حالت ۱ رنگ بعضی از گره‌ها را تغییر می‌دهد، و بدین صورت خصوصیت ۵ را حفظ می‌کند: تمام مسیرهای پایینی از یک گره تا یک برگ دارای تعداد مساوی گره‌های سیاه هستند. در ادامه‌ی حلقه‌ی while پدر بزرگ $z$ گره‌ی جدید $z$ خواهد بود. هر گونه نقض خصوصیت ۴ اکنون می‌تواند فقط میان $z$ جدید، که قرمز است، و پدر آن که ممکن است قرمز باشد، اتفاق بیافتد.

اکنون نشان می‌دهیم که حالت ۱ در آغاز تکرار بعدی ثابت حلقه را برقرار نگه می‌دارد. از $z$ برای نشان دادن گره‌ی $z$ در تکرار فعلی، و از $z' = z.p.p$ برای نشان دادن گره‌ای که در تست خط ۱ در تکرار بعدی آن را $z$ می‌نامیم، استفاده می‌کنیم.

I. چون این تکرار $z.p.p$ را با قرمز رنگ آمیزی می‌کند، در آغاز تکرار بعدی گره‌ی $z'$ قرمز است.

II. در این تکرار، $z'.p$ همان $z.p.p.p$ است و رنگ آن تغییری نمی‌کند. اگر این گره ریشه باشد، قبل از شروع این تکرار سیاه است، و بنابراین در آغاز تکرار بعدی سیاه باقی می‌ماند.

III. قبلا بحث کردیم که حالت ۱ خصوصیت ۵ را برقرار نگه می‌دارد، و به وضوح خصوصیات ۱ و یا ۳ را نقض نمی‌کند.

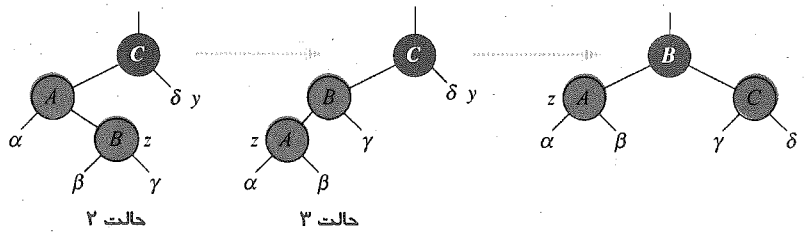
اگر گره‌ی $z'$ در آغاز تکرار بعد ریشه باشد، در این صورت حالت ۱ تنها نقض شدگی خصوصیت ۴ را در این تکرار تصحیح کرده است. از آن جایی که $z'$ ریشه و قرمز است، خصوصیت ۲ تنها خصوصیتی خواهد بود که نقض شده است، و این نقض به خاطر $z'$ است.

اگر در آغاز تکرار بعد $z'$ ریشه نباشد، در این صورت حالت ۱ خصوصیت ۲ را نقض نکرده است. حالت ۱ تنها نقض خصوصیت ۴ را که در آغاز تکرار وجود داشته رفع کرده است. سپس $z'$ را قرمز کرده و در $z'.p$ هم تغییری ایجاد نکرده است. اگر $z'.p$ سیاه بوده باشد، خصوصیت ۴ نقض نشده است. اگر $z'.p$ قرمز بوده باشد، قرمز کردن $z'$، خصوصیت ۴ را میان $z'$ و $z'.p$ نقض کرده است.

### حالت ۲: عموی $z'$ سیاه، و $z$ فرزند سمت راست است.

### حالت ۳: عموی $z'$ سیاه، و $z$ فرزند سمت چپ است.

در حالت‌های ۲ و ۳ رنگ عموی $z$ سیاه است. این دو حالت نسبت به فرزند چپ یا راست بودن $z$ از یکدیگر جدا می‌شوند. خطوط ۱۰-۱۱ مربوط به حالت ۲ می‌شوند، که در شکل ۱۳-۶، به همراه حالت ۳ نشان داده شده است. در حالت ۲ گره‌ی $z$ فرزند سمت راست پدر خود است. با انجام یک دوران چپ، حالت ۲ را به حالت ۳ تبدیل می‌کنیم (خطوط ۱۲-۱۴)، که در آن $z$ فرزند سمت چپ است. از آن جایی که هم $z$ و هم $z.p$ قرمز هستند، این دوران چپ نه بر روی ارتفاع سیاه تأثیر می‌گذارد و نه بر روی خصوصیت ۵. چه ما به صورت مستقیم وارد حالت ۳ شویم، و چه از طریق حالت ۲، عموی $z$ سیاه خواهد بود، چرا که در غیر این صورت باید وارد حالت ۱ می‌شدیم. به علاوه، گره‌ی $z.p.p$ وجود دارد، چرا که قبلا بحث کرده بودیم که این گره در زمان اجرای خطوط ۲و ۳ وجود داشته است، و پس از این که $z$ را در خط ۱۰ یک سطح به سمت بالا، و سپس در خط ۱۱ یک سطح به سمت پایین حرکت دادیم، هویت $z.p.p$ بدون تغییر باقی مانده است. در حالت ۳ چند تغییر رنگ و یک دوران راست انجام دادیم، که خصوصیت ۵ را حفظ می‌کند، و سپس، از آن جایی که دیگر دو گره‌ی قرمز پشت سر هم نداریم، کار ما تمام شده است. بدنه‌ی حلقه‌ی while دیگر اجرا نمی‌شود، زیرا اکنون $z.p$ سیاه است.




**شکل ۱۴-۶** حالت‌های ۲ و ۳ در رویه‌ی RB-INSERT. مانند حالت ۱، خصوصیت ۴ در هر دو حالت ۲ و ۳ نقض می‌شود، چرا که $z$ و پدر آن $z.p$ هر دو قرمز هستند. هر یک از زیردرختان $\alpha$، $\beta$، $\gamma$، و $\delta$ یک ریشه‌ی سیاه دارند ( $\alpha$، $\beta$ و $\gamma$ از خصوصیت ۴، و $\delta$ از این رو که در غیر این صورت باید در حالت ۱ باشیم)، و ارتفاع سیاه همه برابر است. حالت ۲ به کمک یک دوران چپ به حالت ۳ تبدیل می‌شود، که خصوصیت ۵ را برقرار نگه می‌دارد: تمام مسیرهای پایینی از یک گره به یک برگ دارای تعداد برابری گره‌ی سیاه هستند. در حالت ۳ چند تغییر رنگ و دوران انجام می‌شود که باز هم خصوصیت ۵ را حفظ می‌کنند. سپس حلقه‌ی while پایان می‌یابد، چرا که خصوصیت ۴ ارضا می‌شود: دیگر دو گره‌ی قرمز پشت سر هم قرار ندارند.

اکنون نشان می‌دهیم که حالت‌های ۲ و ۳ ثابت حلقه را برقرار نگه می‌دارند. (همان طور که بحث کردیم، $z.p$ در آزمایش بعدی در خط ۱ سیاه خواهد بود، و بدنه‌ی حلقه دیگر اجرا نمی‌شود.)

I. حالت ۲ باعث می‌شود که $z$ به $z.p$ اشاره کند، که قرمز است. تغییر دیگری بر روی $z$ و رنگ آن در حالت‌های ۲ و ۳ انجام نمی‌گیرد.

II. حالت ۳ گره‌ی $z.p$ را سیاه می‌کند، بنابراین اگر $z.p$ در آغاز تکرار بعد ریشه باشد، سیاه خواهد بود.

III. مانند حالت ۱، خصوصیات ۱، ۳، و ۵ در حالت‌های ۲ و ۳ حفظ می‌شوند.

از آن جایی که گره‌ی $z$ در حالت‌های ۲ و ۳ ریشه نیست، می‌دانیم که خصوصیت ۲ نقض نمی‌شود. حالت‌های ۲ و ۳ خصوصیت ۲ را نقض نمی‌کنند، چرا که تنها گره‌ای که قرمز شده است، به وسیله‌ی یک دوران در حالت ۳ تبدیل می‌شود به فرزند یک گره‌ی سیاه.

حالت‌های ۲ و ۳ تنها نقض خصوصیت ۴ را بر طرف می‌کنند، و همچنین هیچ یک از خصوصیات دیگر را نقض نمی‌کنند.

اکنون که نشان دادیم تمام تکرارهای حلقه ثابت حلقه را حفظ می‌کنند، نشان داده‌ایم که RB-INSERT-FIXUP به درستی خصوصیات درختان قرمز- سیاه را بازسازی می‌کند.

## تحلیل

زمان اجرای RB-INSERT چقدر است؟ از آن جایی که ارتفاع یک درخت قرمز سیاه با $n$ گره $O(\lg n)$ است، خطوط ۱-۱۶ رویه‌ی RB-INSERT در زمان $O(\lg n)$ اجرا می‌شوند. در RB-INSERT-FIXUP، حلقه‌ی **while** فقط در صورتی تکرار می‌شود که حالت ۱ اجرا شده باشد، و سپس اشاره‌گر $z$

به اندازه‌ی دو سطح در درخت به سمت بالا حرکت می‌کند. بنابراین کل تعداد دفعاتی که حلقه‌ی while می‌تواند تکرار شود $O(\lg n)$ است. پس کل زمانی که RB-INSERT می‌گیرد، $O(\lg n)$ است. جالب است بدانید که در این رویه به هیچ وجه تعداد دوران‌ها بیشتر از ۲ نخواهد بود، چرا که اگر حالت ۲ و یا ۳ اجرا شود، حلقه‌ی while پایان می‌یابد.

## تمرین‌ها

۱-۳-۱۳ در خط ۱۶ رویه‌ی RB-INSERT، گره‌ی تازه درج شده‌ی $z$ را با قرمز رنگ آمیزی کردیم. توجه کنید که اگر $z$ را سیاه می‌کردیم، خصوصیت ۴ درختان قرمز-سیاه نقض نمی‌شد. توضیح دهید که چرا $z$ را با سیاه رنگ آمیزی نکردیم؟

۲-۳-۱۳ درخت قرمز- سیاهی را نشان دهید که پس از درج کلیدهای ۴۱، ۳۸، ۳۱، ۱۲، ۱۹، و ۸ در یک درخت خالی ایجاد می‌شود.

۳-۳-۱۳ فرض کنید که ارتفاع سیاه هر یک از زیردرختان $\alpha$، $\beta$، $\gamma$، $\delta$، و $\varepsilon$ در شکل ۱۳-۵ و ۱۳-۶ برابر $k$ باشد. هر یک از گره‌ها را در هر شکل با ارتفاع سیاه آن‌ها علامت گذاری کنید تا نشان دهید که خصوصیت ۵ با تبدیل‌های انجام شده حفظ می‌شود.

۴-۳-۱۳ پروفسور تیچ (Teach) فکر می‌کند که RB-INSERT-FIXUP ممکن است در حالتی که تست خط ۱ باعث نشود که وقتی $z$ ریشه است حلقه پایان یابد، $T.nil.color$ را با RED مقدار دهی کند. نشان دهید که نگرانی پروفسور بی مورد است، و RB-INSERT-FIXUP هیچ وقت $T.nil.color$ را برابر با RED قرار نمی‌دهد.

۵-۳-۱۳ یک درخت قرمز-سیاه را در نظر بگیرید که با درج $n$ گره به وسیله‌ی RB-INSERT شکل می‌گیرد. بحث کنید که اگر $n > ۱$، درخت حداقل یک گره‌ی قرمز دارد.

۶-۳-۱۳ یک روش بهینه برای پیاده‌سازی RB-INSERT در حالتی پیشنهاد کنید که نمایش درختان قرمز-سیاه حافظه‌ای برای ذخیره‌ی اشاره‌گر به پدر گره‌ها ندارد.

## ۴-۱۳ حذف

حذف یک گره، مانند بقیه‌ی اعمال اصلی بر روی یک درخت قرمز- سیاه با $n$ گره، در زمان $O(\lg n)$ انجام می‌شود. حذف یک گره از یک درخت قرمز-سیاه مقداری پیچیده‌تر از درج یک گره در درخت قرمز- سیاه است.

رویه‌ی حذف یک گره از یک درخت قرمز-سیاه بر پایه‌ی رویه‌ی TREE-DELETE است (بخش ۱۲-۳ را ببینید). ابتدا باید زیرروال TRANSPLANT را که TREE-DELETE فراخوانی می‌کند، اصلاح کنیم تا بتوان آن را برای درختان قرمز-سیاه به کار برد:

```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

رویه‌ی RB-TRANSPLANT از دو جهت با TRANSPLANT تفاوت دارد. اول، خط ۱ به جای NIL از مقدار نگهبان $T.nil$ استفاده می‌کند. دوم، مقداردهی $v.p$ در خط ۶ بدون هیچ شرطی اجرا می‌شود: حتی در صورتی که $v$ به مقدار نگهبان اشاره کند هم می‌توانیم $v.p$ را مقداردهی کنیم. در واقع چون $v = T.nil$ می‌توانیم از قابلیت مقداردهی به $v.p$ استفاده کنیم.

رویه‌ی RB-DELETE مانند رویه‌ی TREE-DELETE است، فقط با چند خط اضافه‌ی شبه‌کد. بعضی از خطوط اضافه اطلاعات یک گره‌ی $y$ را نگه می‌دارند که ممکن است باعث نقض خصوصیات درختان قرمز- سیاه شود. وقتی می‌خواهیم گره‌ی $z$ را حذف کنیم، و $z$ کم‌تر از دو فرزند دارد، آن گاه $z$ از درخت حذف می‌شود، و $y$ را به جای $z$ قرار می‌دهیم. وقتی $z$ دو فرزند دارد، آن گاه $y$ باید عنصر مابعد $z$ باشد، و $y$ به مکان $z$ در درخت حرکت می‌کند. همچنین قبل از جابه‌جایی $y$ در درخت یا حذف آن از درخت، رنگ آن را به خاطر می‌سپاریم، و همچنین اطلاعات گره‌ی $x$ را که به مکان اولیه‌ی $y$ در درخت منتقل می‌شود، چرا که گره‌ی $x$ هم ممکن است خصوصیات درختان قرمز-سیاه را نقض کند. پس از حذف گره‌ی $z$، رویه‌ی RB-DELETE یک رویه‌ی کمکی RB-DELETE-FIXUP را فراخوانی می‌کند، که با چند تغییر رنگ و دوران خصوصیات درختان قرمز-سیاه را بازیابی می‌کند.

```
RB-DELETE(T, z)
1  y = z
2  y-original-color = y.color
3  if z.left == T.nil
4      x = z.right
5      RB-TRANSPLANT(T, z, z.right)
6  elseif z.right == T.nil
7      x = z.left
8      RB-TRANSPLANT(T, z, z.left)
9  else y = TREE-MINIMUM(z.right)
10     y-original-color = y.color
11     x = y.right
12     if y.p == z
13         x.p = y
14     else RB-TRANSPLANT(T, y, y.right)
15         y.right = z.right
16         y.right.p = y
```

```
17    RB-TRANSPLANT(T, z, y)
18    y.left = z.left
19    y.left.p = y
20    y.color = z.color
21  if y-original-color == BLACK
22    RB-DELETE-FIXUP(T, x)
```

با این که تعداد خطوط کد RB-DELETE تقریباً دوبرابر TREE-DELETE است، ساختار اصلی هر دو رویه یکی است. می‌توانید هر خط از RB-DELETE را در TREE-DELETE بیابید (با جایگزینی NIL با $T.nil$ و فراخوانی RB-TRANSPLANT به جای TRANSPLAN)، که تحت شرایط یکسان اجرا می‌شوند.

در زیر تفاوت‌های دیگر دو رویه را می‌بینیم:

- گره‌ی $y$ را به عنوان گره‌ای که از درخت حذف می‌شود، یا در درخت جابه‌جا می‌شود، نگه می‌داریم. در خط ۱، $y$ به $z$ اشاره می‌کند، برای حالتی که $z$ کم‌تر از دو فرزند دارد و باید از درخت حذف شود. وقتی $z$ دو فرزند دارد، در خط ۹، $y$ به عنصر مابعد $z$ اشاره می‌کند، درست مانند TREE-DELETE، و $y$ به مکان $z$ در درخت منتقل می‌شود.
- از آن جایی که ممکن است رنگ گره‌ی $y$ تغییر کند، متغیر $y$-*original-color* رنگ آن را قبل از هر تغییری ثبت می‌کند. خطوط ۲ و ۱۰ این متغیر را درست بعد از تغییر در $y$ مقداردهی می‌کنند. وقتی $z$ دو فرزند دارد، آن گاه $y \neq z$ و گره‌ی $y$ به مکان اولیه‌ی $z$ در درخت قرمز- سیاه منتقل می‌شود؛ خط ۲۰ رنگی مشابه رنگ $z$ به $y$ می‌دهد. باید رنگ اولیه‌ی $y$ را ذخیره کنیم تا بتوانیم در پایان RB-DELETE آن را تست کنیم؛ اگر این رنگ سیاه بود، آن گاه حذف یا جابه‌جایی $y$ می‌تواند خصوصیات درختان قرمز-سیاه را نقض کند.
- همان طور که بحث شد، اطلاعات گره‌ی $x$ را که به مکان گره‌ی $y$ منتقل می‌شود، نگه می‌داریم. مقداردهی‌های خطوط ۴، ۷، و ۱۱ گره‌ی $x$ را طوری مقداردهی می‌کنند که یا به تنها فرزند $y$ اشاره کند و یا، اگر $y$ هیچ فرزندی نداشته باشد، به مقدار نگهبان $T.nil$. (از بخش ۱۲-۳ به خاطر بیاورید که $y$ فرزند سمت چپ ندارد.)
- از آن جایی که گره‌ی $x$ به مکان اولیه‌ی گره‌ی $y$ منتقل می‌شود، خصیصه‌ی $x.p$ همیشه طوری مقداردهی می‌شود که به مکان اولیه‌ی پدر $y$ اشاره کند، حتی اگر $x$، در واقع $T.nil$ باشد. غیر از حالتی که $z$ پدر اولیه‌ی $y$ باشد (که فقط در صورتی رخ می‌دهد که $z$ دو فرزند داشته باشد، و عنصر مابعد آن، $y$، فرزند سمت راست آن باشد)، مقداردهی به $x.p$ در خط ۶ رویه‌ی RB-TRANSPLANT انجام می‌شود. (مشاهده کنید که وقتی RB-TRANSPLANT در خطوط ۵، ۸، یا ۱۴ فراخوانی می‌شود، دومین پارامتر ارسال شده همان $x$ است.)

ولی وقتی پدر اولیه‌ی $y$ گره‌ی $z$ باشد، نمی‌خواهیم $x.p$ به پدر اولیه‌ی $y$ اشاره کند، چرا که می‌خواهیم آن گره را از درخت حذف کنیم. از آن جایی که گره‌ی $y$ به بالا حرکت می‌کند تا جای

$z$ را در درخت بگیرد، مقداردهی $x.p$ با $y$ در خط ۱۳ باعث می‌شود که $x.p$ بـه مکـان اولیـه‌ی پدر $y$ اشاره کند، حتی اگر داشته باشیم $x = T.nil$.

- نهایتاً اگر گره‌ی $y$ سیاه باشد، ممکن است یک یا دو ناسازگاری با خصوصـیات درختـان قرمـز-سیاه به وجود آورده باشیم، و بنابراین RB-DELETE-FIXUP را در خط ۲۲ فراخوانی مـی‌کنـیم تـا این خصوصیات را بازیابی کنیم. اگر $y$ قرمز بود، خصوصیات درختان قرمـز-سـیاه بـا حـذف یـا جابه‌جایی $y$ همچنان برقرار می‌ماند، به دلایل زیر:

۱. ارتفاع سیاه در هیچ کجای درخت تغییر نمی‌کرد.
۲. هیچ دو گره‌ی قرمزی همسـایه نمـی‌شـدند. چـون $y$ جـای $z$ را (همچنـین رنـگ آن را) در درخت می‌گیرد، نمی‌توانیم در مکان جدید $y$ دو گره‌ی قرمز رنگ در همسـایگی هـم داشـته باشیم. به علاوه اگر $y$ فرزند سمت راست $z$ نبود، آن گاه فرزند راست اولیه‌ی $y$ جای $y$ را در درخت می‌گرفت. اگر $y$ قرمز باشد، آن گاه $x$باید سیاه باشد، و بنابراین جـایگزینی $y$ بـا $x$ نمی‌تواند باعث شود که دو گره‌ی قرمز همسایه شوند.
۳. از آن جایی که اگر $y$ قرمز باشد، نمی‌تواند ریشه باشد، ریشه سیاه باقی می‌ماند.

اگر گره‌ی $y$ سیاه باشد سه مشکل به وجود خواهد آمد، که فراخوانی RB-DELETE-FIXUP آن‌هـا را مرتفع خواهد کرد. اول، اگر $y$ ریشه باشد و یک فرزند قرمز آن تبدیل به ریشه شود، خصوصیت ۲ نقض خواهد شد. دوم، اگر هر دوی $x$ و $x.p$ قرمز باشند آن گاه خصوصیت ۴ نقض می‌شـود. سـوم، جابه‌جایی $y$ در درخت باعث می‌شود که هر مسیر ساده که قبلا حاوی $y$ بوده است، یک گره‌ی سـیاه کم‌تر داشته باشد. بنابراین اکنـون خصوصـیت ۵ توسـط تمـام اجـداد $y$ در درخـت نقـض مـی‌شـود. می‌توانیم نقض خصوصیت ۵ را بدین صورت رفع کنیم که فرض کنیم گره‌ی $x$، کـه اکنـون در مکـان اولیه‌ی $y$ قرار دارد، یک «سیاه اضافی» دارد. یعنی اگر یکی به تعداد گره‌های سیاه در تمـام مسـیرهای ساده‌ی حاوی $x$ اضافه کنیم، آن گاه تحت این تفسیر خصوصیت ۵ برقرار است. وقتی گره‌ی سـیاه $y$ را حذف یا جابه‌جا می‌کنیم، سیاهی آن را به $x$ «می‌دهیم». مشکل این جا است که اکنون گـره‌ی $x$ نـه سیاه است و نه قرمز، و خصوصیت ۱ نقض می‌شود. در عوض گره‌ی $x$ یا «سیاه دوگانـه» اسـت و یـا «قرمز و سیاه»، و به ترتیب دو تا یا یکی به تعداد گره‌های سیاه در مسیرهای سـاده‌ی حـاوی $x$ اضـافه می‌کند. خصیصه‌ی *color* متغیر $x$ اکنون یا RED خواهد بود (اگر $x$ قرمز و سیاه باشد) و یـا BLACK (اگر $x$ سیاه دوگانه باشد). به عبارت دیگر، سیاه اضافه بر روی گره روی مکان $x$ اثـر مـی‌کنـد و نـه روی خصیصه‌ی *color*.

اکنون می‌توانیم رویه‌ی RB-DELETE-FIXUP را ببینـیم و بررسـی کنـیم کـه چگونـه خصوصـیات درختان قرمز- سیاه را به درخت جستجو بازمی‌گرداند.

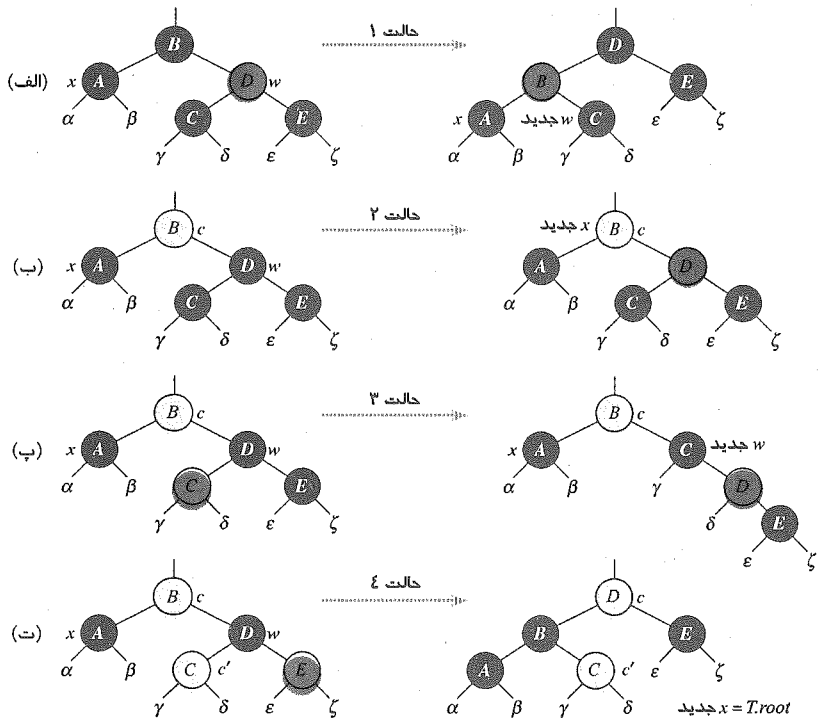
```
RB-DELETE-FIXUP(T, z)
1   while x ≠ T.root and x.color == BLACK
2       if x == x.p.left
3           w = x.p.right
```

```
4        if w.color == RED
5            w.color = BLACK                          // case 1
6            x.p.color = RED                          // case 1
7            LEFT-ROTATE(T, x.p)                      // case 1
8            w = x.p.right                            // case 1
9        if w.left.color == BLACK and w.right.color == BLACK
10           w.color = RED                            // case 2
11           x = x.p                                  // case 2
12       else if w.right.color == BLACK
13               w.left.color = BLACK                 // case 3
14               w.color = RED                        // case 3
15               RIGHT-ROTATE(T, w)                   // case 3
16               w = x.p.right                        // case 3
17           w.color = x.p.color                      // case 4
18           x.p.color = BLACK                        // case 4
19           w.right.color = BLACK                    // case 4
20           LEFT-ROTATE(T, x.p)                      // case 4
21           x = T.root                               // case 4
22   else (same as then clause with "right" and "left" exchanged)
23 x.color = BLACK
```

رویه‌ی RB-DELETE-FIXUP خصوصیات ۱، ۲، و ۴ را بازسازی می‌کند. تمرین ۱۳-۴-۱ و ۱۳-۴-۲ از شما می‌خواهد نشان دهید که این رویه خصوصیات ۲ و ۴ را بازسازی می‌کند، و بنابراین در ادامه‌ی این بخش بر روی خصوصیت ۱ تمرکز می‌کنیم. هدف حلقه‌ی while در خطوط ۱-۲۲ این است که سیاهی اضافی را در درخت به سمت بالا ببرد تا

۱. اشاره‌گر $x$ به یک گره‌ی قرمز و سیاه اشاره کند، که در این صورت در خط ۲۳ گره‌ی $x$ را (به صورت یگانه) با سیاه رنگ آمیزی می‌کنیم،
۲. $x$ به ریشه اشاره کند، که در این صورت این سیاهی اضافی می‌تواند به سادگی حذف شود، و یا
۳. بتوان رنگ آمیزی‌ها و دوران‌های مناسب را انجام داد.

در حلقه‌ی while متغیر $x$ همیشه به یک گره‌ی سیاه دوگانه که ریشه نیست اشاره می‌کند. در خط ۲ تشخیص می‌دهیم که آیا $x$ فرزند سمت چپ پدر خود $x.p$ است یا فرزند سمت راست. (در این جا کد حالتی که $x$ فرزند سمت چپ است را داده‌ایم؛ حالتی که در آن $x$ فرزند سمت راست است – خط ۲۲ – متقارن است.) همیشه یک اشاره‌گر $w$ به برادر $x$ نگه می‌داریم. از آن جایی که $x$ سیاه دوگانه است، گره‌ی $w$ نمی‌تواند $T.nil$ باشد، چرا که در غیر این صورت تعداد سیاه‌ها در مسیر $x.p$ به برگ (سیاه یگانه‌ی) $w$ کم‌تر از تعداد سیاه‌ها در مسیر $x.p$ به $x$ خواهد بود.




**شکل ۱۳-۷** حالت‌های حلقه‌ی **while** در رویه‌ی RB-DELETE-FIXUP. خصوصیت *color* گره‌های سیاه، BLACK است، گره‌های با رنگ قرمز گره‌های RED هستند، و خصوصیت *color* گره‌های با رنگ آبی، $c$ یا $c'$ است، که ممکن است RED باشد و یا BLACK. حروف $\alpha$، $\beta$، ...، $\zeta$ نشان‌دهنده‌ی زیردرختان دلخواه هستند. در هر حالت، ترکیب سمت راست با تغییراتی در رنگ‌ها و/یا انجام یک تبدیل به ترکیب سمت چپ تبدیل می‌شود. هر گره‌ای که به $x$ اشاره می‌کند، حاوی یک سیاهی اضافی است و، یا سیاه دوگانه است یا قرمز و سیاه. تنها حالتی که باعث می‌شود حلقه تکرار شود حالت ۲ است. (الف) حالت ۱، با تعویض رنگ گره‌های $B$ و $D$ و انجام یک دوران چپ به حالت‌های ۲، ۳ یا ۴ تبدیل می‌شود. (ب) در حالت ۲، سیاهی اضافی که با اشاره‌گر $x$ نشان داده می‌شود، با رنگ آمیزی گره‌ی $D$ با قرمز و قرار دادن اشاره‌گر $x$ به گره‌ی $B$، در درخت به سمت بالا حرکت می‌کند. اگر از طریق حالت ۱ وارد حالت ۲ شویم، حلقه‌ی **while** پایان خواهد یافت چرا که گره‌ی جدید $x$ قرمز و سیاه است، و بنابراین مقدار $c$ خصوصیت *color* آن، RED است. (پ) حالت ۳ با تعویض رنگ گره‌های $C$ و $D$ و انجام یک دوران راست به حالت ۴ تبدیل می‌شود. (ت) در حالت ۴، سیاهی اضافی نشان داده شده با $x$ را می‌توان با چند تغییر رنگ و یک دوران چپ (بدون نقض خصوصیات درختان قرمز-سیاه) حذف کرد، و حلقه پایان می‌یابد.

این چهار حالت[۲] در کد در شکل ۱۳-۷ مشخص شده‌اند. قبل از بررسی جزئیات هر حالت، اجازه

[۲] مانند RB-INSERT-FIXUP، حالت‌ها در RB-DELETE-FIXUP دو به دو مجزا نیستند.

دهید نگاهی کلی‌تر بیندازیم تا ببینیم چطور می‌توانیم تأیید کنیم که تبدیلات درون هر حالت خصوصیت ۵ را حفظ می‌کنند. ایده‌ی اصلی این است که در هر حالت تعداد گره‌های سیاه (شامل سیاهی اضافی $x$) از ($x$ شامل) ریشه‌ی زیردرخت نشان داده شده به هر یک از زیردرخت‌های $\alpha$، $\beta$، ...، $\xi$ با تبدیلات حفظ می‌شود. بنابراین، اگر خصوصیت ۵ قبل از تبدیلات برقرار باشد، در ادامه هم برقرار خواهد بود. به عنوان مثال در شکل ۱۳-۷(الف)، که نشان دهنده‌ی حالت ۱ است، تعداد گره‌های سیاه از ریشه به هر یک از زیردرخت‌های $\alpha$ و یا $\beta$ برابر ۳ است، هم قبل از تبدیلات و هم بعد از آن. (دوباره، به یاد بیاورید که گره‌ی $x$ حاوی یک سیاهی اضافی است.) به طور مشابه تعداد گره‌های سیاه از ریشه به هر یک از زیردرخت‌های $\gamma$، $\delta$، $\varepsilon$ و $\xi$ برابر ۲ است، هم قبل از تبدیلات و هم بعد از آن. در شکل ۱۳-۷(ب)، شمارش باید شامل مقدار $c$ از خصوصیت *color* ریشه‌ی زیردرخت نشان داده شده باشد، که می‌تواند RED یا BLACK باشد. اگر تعریف کنیم $count(RED)=0$ و $count(BLACK)=1$، در این صورت تعداد گره‌های سیاه از ریشه به $\alpha$ برابر است با $count(c)+2$، هم قبل از تبدیلات و هم بعد از آن. در این حالت بعد از تبدیلات، خصوصیت *color* گره‌ی جدید $x$ برابر $c$ است، ولی این گره در واقع یا قرمز و سیاه است (اگر $c=RED$) و یا سیاه دوگانه (اگر $c=BLACK$). حالت‌های دیگر را می‌توان به صورت مشابه بررسی کرد (تمرین ۱۳-۴-۵ را ببینید).

### حالت ۱: برادر $x$ قرمز است.

حالت ۱ (خطوط ۵-۸ رویه‌ی RB-DELETE-FIXUP و شکل ۱۳-۷(الف)) وقتی اتفاق می‌افتد که گره‌ی $w$، برادر گره‌ی $x$، قرمز باشد. از آن جایی که فرزندان $w$ باید سیاه باشند، می‌توانیم رنگ $w$ و $x.p$ را عوض کرده و سپس یک دوران چپ بر روی $x.p$ انجام دهیم بدون این که هیچ یک از خصوصیات درختان قرمز-سیاه نقض شوند. برادر جدید $x$، که یکی از فرزندان $w$ قبل از دوران است، اکنون سیاه است و بنابراین حالت ۱ را به حالت‌های ۲، ۳ یا ۴ تبدیل کرده‌ایم.

خالت‌های ۲، ۳، و ۴ وقتی اتفاق می‌افتند که گره‌ی $w$ سیاه است؛ این حالت‌ها به وسیله‌ی رنگ فرزندان $w$ از هم متمایز می‌شوند.

### حالت ۲: برادر $x$ و هر دو فرزند $w$ سیاه هستند.

در حالت ۲ (خطوط ۱۰-۱۱ رویه‌ی RB-DELETE-FIXUP و شکل ۱۳-۷-(ب)) هر دو فرزند $w$ سیاه هستند. از آن جایی که $w$ هم سیاه است، یک سیاهی از هر دو گره‌ی $x$ و $w$ می‌گیریم، که باعث می‌شود $x$ سیاه یگانه و $w$ قرمز شود. برای جبران این حذف سیاهی از $x$ و $w$، می‌خواهیم یک سیاهی اضافی به $x.p$ اضافه کنیم، که در ابتدا یا قرمز بود و یا سیاه. این کار را با تکرار حلقه‌ی **while** در حالتی که $x.p$ گره‌ی جدید $x$ است، انجام می‌دهیم. مشاهده کنید که اگر از طریق حالت ۱ به حالت ۲ وارد شویم، گره‌ی جدید $x$ قرمز و سیاه است، چرا که $x.p$ در ابتدا قرمز بود. بنابراین، مقدار $c$ از خصوصیت *color* گره‌ی جدید $x$ برابر RED است، و حلقه زمانی تمام می‌شود که این وضعیت

را چک کند. سپس، گره‌ی جدید $x$ در خط ۲۳ (به صورت یگانه) با سیاه رنگ آمیزی می‌شود.

**حالت ۳ : برادر $x$ سیاه، فرزند سمت چپ $w$ قرمز، و فرزند سمت راست $w$ سیاه است.**

حالت ۳ (خطوط ۱۳-۱۶ و شکل ۱۳-۷(پ)) زمانی اتفاق می‌افتد که $w$ سیاه، فرزند سمت چپ آن قرمز، و فرزند سمت راست آن سیاه باشد. می‌توانیم بدون نقض هیچ یک از خصوصیات درختان قرمز- سیاه، رنگ $w$ و فرزند سمت چپ آن را عوض کنیم، و سپس یک دوران چپ بر روی $w$ انجام دهیم. اکنون برادر جدید $x$ یک گره‌ی سیاه است که فرزند سمت راست آن قرمز است، و بنابراین حالت ۳ به حالت ۴ تبدیل شده است.

**حالت ۴: برادر $x$ سیاه، و فرزند سمت راست $w$ قرمز است.**

حالت ۴ (خطوط ۱۷-۲۱ و شکل ۱۳-۷(ت)) زمانی اتفاق می‌افتد که $w$ سیاه و فرزند سمت راست آن قرمز باشد. با انجام چند تغییر رنگ و یک دوران چپ بر روی $x.p$، می‌توانیم سیاهی اضافی روی $x$ را حذف کنیم، و آن را تبدیل به یک گره‌ی سیاه یگانه کنیم، بدون این که خصوصیات درختان قرمز- سیاه نقض شوند. تبدیل $x$ به ریشه باعث می‌شود که در هنگام تست شرط حلقه، حلقه‌ی **while** پایان یابد.

**تحلیل**

زمان اجرای RB-DELETE چقدر است؟ از آن جایی که ارتفاع یک درخت قرمز- سیاه با $n$ گره $O(\lg n)$ است، هزینه‌ی کلی رویه بدون فراخوانی RB-DELETE-FIXUP برابر است با $O(\lg n)$. در RB-DELETE-FIXUP، حالت‌های ۱، ۳، و ۴ هر یک پس از انجام تعداد ثابتی تغییر رنگ و حداکثر سه دوران پایان می‌یابند. حالت ۲ تنها حالتی است که در آن ممکن است حلقه‌ی **while** تکرار شود، و سپس اشاره‌گر $x$ حداکثر به $O(\lg n)$ بار در درخت به سمت بالا حرکت می‌کند، و هیچ دورانی انجام نمی‌شود. بنابراین، رویه‌ی RB-DELETE-FIXUP به زمان $O(\lg n)$ نیاز دارد و حداکثر سه دوران انجام می‌دهد، و از این رو زمان اجرای کلی RB-DELETE برابر است با $O(\lg n)$.

## تمرین‌ها

۱۳-۴-۱ نشان دهید که پس از اجرای RB-DELETE-FIXUP، ریشه‌ی درخت باید سیاه باشد.

۱۳-۴-۲ نشان دهید که اگر در RB-DELETE، هم $x$ و هم $y.p$ قرمز باشند، آن گاه خصوصیت ۴ با فراخوانی RB-DELETE-FIXUP $(T, x)$ بازسازی می‌شود.

۱۳-۴-۳ در تمرین ۱۳-۳-۲ یک درخت قرمز-سیاه دیدیم که از درج پشت سر هم کلیدهای ۴۱، ۳۸، ۳۱، ۱۹، ۱۲، و ۸ در یک درخت خالی حاصل شده بود. اکنون درختان قرمز-سیاهی را نشان دهید که از حذف کلیدهای ۸، ۱۲، ۱۹، ۳۱، ۳۸، و ۴۱ از این درخت حاصل می‌شوند.

۴-۴-۱۳ در کدام خطوط کد رویه‌ی RB-DELETE-FIXUP ممکن است $T.nil$ را بررسی و یا اصلاح کنیم؟

۵-۴-۱۳ در هر یک از حالت‌های شکل ۱۳-۷، تعداد گره‌های سیاه را از ریشه‌ی زیردرخت نشان داده شده به هر یک از زیردرخت‌های $\alpha$، $\beta$، ...، $\xi$ بدست آورید، و نشان دهید که این تعدادها پس از تبدیلات ثابت باقی می‌مانند. وقتی خصوصیت $color$ یک گره $c$ یا $c'$ است، در شمارش خود از نمادهای $count(c)$ یا $count(c')$ به صورت سمبلیک استفاده کنید.

۶-۴-۱۳ پروفسور اسکلتون (Skeleton) و بارون (Baron) فکر می‌کنند که در آغاز حالت ۱ از RB-DELETE-FIXUP، گره‌ی $x.p$ ممکن است سیاه نباشد. اگر این پروفسورها درست می‌گویند، در این صورت خطوط ۵-۶ غلط هستند. نشان دهید که در آغاز حالت ۱، گره‌ی $x.p$ باید سیاه باشد و نگرانی پروفسورها بی مورد است.

۷-۴-۱۳ فرض کنید یک گره‌ی $x$ به وسیله‌ی RB-INSERT در یک درخت قرمز-سیاه درج می‌شود، و سپس درست پس از درج شدن، به وسیله‌ی RB-DELETE حذف می‌شود. آیا درخت قرمز- سیاه حاصل مانند درخت قرمز- سیاه اولیه است؟ درستی جواب خود را اثبات کنید.

## مسائل

۱-۱۳ ***مجموعه‌های پویای پایدار***

بعضی مواقع در طول اجرای یک الگوریتم، متوجه می‌شویم که باید نسخه‌های قبلی یک مجموعه‌ی پویا را پس از تغییرات، در جایی ذخیره کنیم. به چنین مجموعه‌ای *پایدار* (persistent) گفته می‌شود. یک روش برای پیاده‌سازی یک مجموعه‌ی پایدار این است که هر گاه مجموعه اصلاح می‌شود، تمام آن را ذخیره کنیم. ولی این روش می‌تواند باعث کندی برنامه شود و همچنین فضای زیادی اشغال می‌کند. بعضی مواقع، می‌توانیم کار بسیار بهتری انجام دهیم.

یک مجموعه‌ی پایدار $S$ را با اعمال INSERT، DELETE، و SEARCH در نظر بگیرید، که آن را به کمک درختان جستجوی دودویی پیاده‌سازی می‌کنیم، مانند شکل ۱۳-۸(الف).

یک ریشه‌ی جداگانه برای هر نسخه از مجموعه نگه می‌داریم. برای درج کلید ۵ در مجموعه، یک گره‌ی جدید با کلید ۵ می‌سازیم. این گره فرزند چپ یک گره‌ی جدید با کلید ۷ خواهد شد، چرا که نمی‌توانیم گره‌ی موجود با کلید ۷ را تغییر دهیم. به طور مشابه گره‌ی جدید با کلید ۷، فرزند سمت چپ یک گره‌ی جدید با کلید ۸ خواهد شد که فرزند سمت راست گره‌ی موجود با کلید ۱۰ است. در عوض، گره‌ی جدید با کلید ۸ فرزند سمت راست

یک ریشه‌ی جدید $r'$ با کلید ۴ خواهد شد، که فرزند سمت چپ آن گره‌ی موجود با کلید ۳ است. بنابراین فقط قسمت‌هایی از درخت را کپی می‌کنیم، و بعضی از گره‌ها را با درخت اصلی به اشتراک می‌گذاریم، همان طور که در شکل ۱۳-۸(ب) نشان داده شده است.

فرض کنید که هر یک از گره‌های درخت حاوی فیلدهای *key*، *left*، و *right* است، ولی فیلدی برای پدر آن‌ها وجود ندارد. (همچنین تمرین ۱۳-۳-۶ را ببینید.)

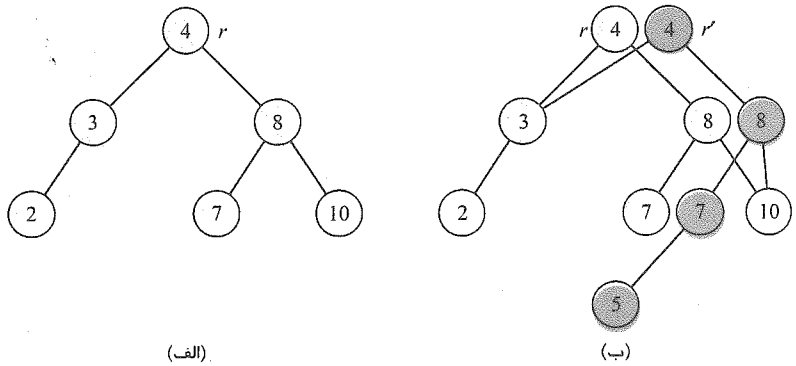
I. برای یک درخت جستجوی دودویی پایدار کلی، گره‌هایی را شناسایی کنید که برای درج یک کلید $k$ و یا حذف یک گره‌ی $y$ باید تغییر کنند.

II. یک رویه‌ی PERSISTENT-TREE-INSERT بنویسید که، با دریافت یک درخت پایدار $T$ و یک کلید $k$ برای درج، یک درخت پایدار جدید $T'$ بازمی‌گرداند که نتیجه‌ی درج کلید $k$ در درخت $T$ است.

III. اگر ارتفاع یک درخت جستجوی دودویی پایدار $T$ برابر $h$ باشد، مرتبه‌ی زمانی و حافظه‌ای پیاده‌سازی شما برای PERSISTENT-TREE-INSERT چقدر است؟ (حافظه‌ی مورد نیاز متناسب است با تعداد گره‌های جدید تخصیص یافته.)

IV. فرض کنید که فیلد پدر را هم در گره‌های درخت قرار می‌دادیم. در این حالت PERSISTENT-TREE-INSERT نیاز دارد که کپی‌های بیشتری انجام دهد. اثبات کنید که در این صورت PERSISTENT-TREE-INSERT نیاز به $\Omega(n)$ زمان و حافظه دارد، که در آن $n$ تعداد گره‌های درون درخت است.

V. نشان دهید که چطور می‌توان از درختان قرمز-سیاه استفاده کرد که بدترین حالت زمان اجرا و حافظه‌ی مصرفی برای درج و حذف از مرتبه‌ی $O(\lg n)$ شود.




**شکل ۱۳-۸** (الف) یک درخت جستجوی دودویی با کلیدهای ۲، ۳، ۴، ۷، ۸، و ۱۰. (ب) درخت جستجوی دودویی پایدار که از درج کلید ۵ حاصل می‌شود. آخرین نسخه‌ی مجموعه حاوی گره‌هایی است که از ریشه‌ی $r$ قابل دسترس هستند، و نسخه‌ی قبلی شامل گره‌هایی است که از $r'$ قابل دسترس هستند. گره‌های با سایه‌ی پررنگ، گره‌هایی هستند که با درج کلید ۵ به درخت اضافه شده‌اند.

مسائل

## ۱۳-۲ عملیات اتصال در درختان قرمز-سیاه

عملیات *اتصال* (join)، دو مجموعه‌ی پویای $S_1$ و $S_2$ و یک عنصر $x$ را دریافت می‌کند، به طوری که برای هر $x_1 \in S_1$ و $x_2 \in S_2$، داریم $x_1.key \leq x.key \leq x_2.key$. این عملیات یک مجموعه‌ی $S_1 \cup \{x\} \cup S_2$ بازمی‌گرداند. در این مسئله بررسی می‌کنیم که چطور می‌توان عملیات اتصال را برای درختان قرمز- سیاه پیاده‌سازی کرد.

I. با داشتن یک درخت قرمز-سیاه $T$، فرض کنید ارتفاع سیاه آن را به کمک فیلد $T.bh$ ذخیره می‌کنیم. نشان دهید که این فیلد را می‌توان با RB-INSERT و RB-DELETE، بدون نیاز به حافظه‌ی اضافی درون گره‌ها، و همچنین بدون افزایش زمان اجرای حدی نگه‌داری کرد. نشان دهید که هنگام پایین آمدن از $T$، می‌توانیم ارتفاع سیاه هر یک از گره‌هایی را که پیمایش می‌کنیم در زمان $O(1)$ برای هر گره، تعیین کنیم.

می‌خواهیم یک عملیات RB-JOIN$(T_1, x, T_2)$ پیاده‌سازی کنیم، که $T_1$ و $T_2$ را نابود کرده و یک درخت قرمزسیاه $T = T_1 \cup \{x\} \cup T_2$ بازمی‌گرداند. فرض کنید $n$ مجموع تعداد گره‌ها در $T_1$ و $T_2$ باشد.

II. فرض کنید $T_1.bh \geq T_2.bh$. یک الگوریتم با زمان $O(\lg n)$ ارائه کنید که یک گره‌ی سیاه $y$ در $T_1$ پیدا می‌کند که کلید آن، بزرگ‌ترین کلید میان گره‌های با ارتفاع سیاه $T_2.bh$ است.

III. فرض کنید $T_y$ زیردرخت با ریشه‌ی $y$ باشد. توضیح دهید که چطور $T_y \cup \{x\} \cup T_2$ می‌تواند در زمان $O(1)$ و بدون از بین رفتن خصوصیت درختان جستجوی دودویی جایگزین $T_y$ شود.

IV. $x$ باید چه رنگی باشد که خصوصیات ۱، ۳، و ۵ درختان قرمز-سیاه برقرار باقی بمانند؟ توضیح دهید که چگونه می‌توان خصوصیات ۲ و ۴ را در زمان $O(\lg n)$ برقرار کرد؟

V. بحث کنید که با فرض بخش II، هیچ کلیتی از دست نمی‌رود. حالت متقارنی را توضیح دهید که وقتی $T_1.bh = T_2.bh$ باشد، پیش می‌آید.

VI. نشان دهید که زمان اجرای RB-JOIN برابر $O(\lg n)$ است.

## ۱۳-۳ درختان AVL

یک *درخت AVL* یک درخت جستجوی دودویی است که ارتفاع آن متوازن است: برای هر گره‌ی $x$، ارتفاع زیردرختان سمت چپ و سمت راست این گره حداکثر یکی با هم فرق دارد. برای پیاده‌سازی یک درخت AVL، در هر گره یک فیلد اضافی نگه می‌داریم: $x.h$ که ارتفاع گره‌ی $x$ است. مانند هر درخت جستجوی دودویی دیگر، فرض می‌کنیم که $T.root$ به ریشه‌ی درخت اشاره می‌کند.

I. ثابت کنید که ارتفاع یک درخت AVL با $n$ گره، $O(\lg n)$ است. (*راهنمایی:* اثبات کنید که در یک درخت AVL با ارتفاع $h$، حداقل $F_h$ گره وجود دارد، که $F_h$ عدد فیبوناچی $h$ام است.)

II. برای درج در یک درخت AVL، ابتدا یک گره در مکان مناسب در درخت جستجوی دودویی قرار می‌گیرد. پس از این کار ممکن است که درخت دیگر متوازن نباشد. به خصوص، ممکن است تفاوت ارتفاع فرزندان سمت چپ و سمت راست یک گره، ۲ باشد. یک رویه‌ی BALANCE ($x$) ارائه کنید که یک زیردرخت با ریشه‌ی $x$ دریافت می‌کند که ارتفاع فرزندان سمت چپ و سمت راست آن متوازن است، و تفاوت ارتفاع در آن حداکثر ۲ است، یعنی $|x.right.h - x.left.h| \le 2$، و زیردرخت با ریشه‌ی $x$ را طوری اصلاح می‌کند که ارتفاع آن متوازن شود. (*راهنمایی:* از دوران استفاده کنید.)

III. با استفاده از قسمت II، یک رویه‌ی بازگشتی AVL-INSERT ($x, z$) ارائه کنید که یک گره‌ی $x$ درون درخت AVL و یک گره‌ی تازه ساخته شده‌ی $z$ (که کلید آن مقداردهی شده است) را دریافت می‌کند، و گره‌ی $z$ را به زیردرخت با ریشه‌ی $x$ اضافه می‌کند، و همچنین این خصوصیت را حفظ می‌کند که $x$ ریشه‌ی یک درخت AVL است. مانند TREE-INSERT از بخش ۱۲-۳، فرض کنید که $z.key$ قبلا مقدار دهی شده است و $z.left = \text{NIL}$ و $z.right = \text{NIL}$؛ همچنین فرض کنید که $z.h = 0$. بنابراین برای درج گره‌ی $z$ در درخت $T$، باید AVL-INSERT ($T.root, z$) را فراخوانی کنیم.

IV. نشان دهید که AVL-INSERT که بر روی یک درخت AVL با $n$ گره اجرا شده است، در زمان $O(\lg n)$ اجرا شده و $O(1)$ دوران انجام می‌دهد.

## ۱۳-۴ *Treap* ها

اگر مجموعه‌ای از $n$ عنصر را در یک درخت جستجوی دودویی درج کنیم، ممکن است درخت حاصل به شدت نامتوازن باشد، که باعث می‌شود زمان جستجو بسیار بالا باشد. با این حال همان طور که در بخش ۱۲-۴ دیدیم، درختان جستجوی دودویی به صورت تصادفی ساخته شده گرایش به توازن دارند. بنابراین یک استراتژی که در حالت متوسط از یک مجموعه‌ی ثابت از عناصر، یک درخت جستجوی دودویی متوازن می‌سازد، این است که به صورت تصادفی به عناصر یک جایگشت بدهیم و سپس آن‌ها را به ترتیب در درخت درج کنیم.

ولی اگر در ابتدا تمام عناصر را نداشته باشیم چطور؟ اگر عناصر را یکی یکی دریافت کنیم، آیا باز هم می‌توانیم از آن‌ها یک درخت جستجوی دودویی تصادفی بسازیم؟

در این جا یک ساختمان داده را بررسی می‌کنیم که به این سؤال جواب می‌دهد. یک *treap*، یک درخت جستجوی دودویی است که روش مرتب کردن گره‌ها در آن در آن اصلاح شده است. شکل ۱۳-۹ یک نمونه را نشان می‌دهد. مثل همیشه هر گره‌ی $x$ در درخت دارای یک کلید $x.key$ است. به علاوه، یک مقدار $x.priority$ برای هر گره در نظر می‌گیریم، که یک عدد تصادفی است که مستقلاً برای هر گره محاسبه شده است، و نشان دهنده‌ی اولویت هر گره است. فرض می‌کنیم که مقادیر تمام فیلدهای اولویت، و همچنین مقادیر تمام فیلدهای

کلید متمایز است. گره‌های treap طوری مرتب شده است که کلیدها از قانون درختان جستجوی دودویی پیروی کنند، و اولویت‌ها از خصوصیت هرم کمینه:

- اگر $v$ فرزند سمت چپ $u$ باشد، آن گاه $v.key < u.key$.
- اگر $v$ فرزند سمت راست $u$ باشد، آن گاه $v.key > u.key$.
- اگر $v$ فرزند $u$ باشد، آن گاه $v.priority > u.priority$.

(این درخت از این رو "treap" نام گذاری شده است که هم خصوصیات درختان جستجوی دودویی را دارد و هم خصوصیات هرم‌ها را.)

مفید است که به treapها از دید زیر نگاه کنیم. فرض کنید که گره‌های $x_1, x_2, \ldots, x_n$ را با کلیدهای مربوطه در یک treap درج می‌کنیم. آن گاه treap حاصل درختی است که در صورت درج کلیدها در یک درخت جستجوی دودویی معمولی با ترتیب اولویت (به صورت تصادفی انتخاب شده‌ی) آن‌ها شکل می‌گرفت، یعنی اگر $x_i.priority < x_j.priority$ آن گاه $x_i$ قبل از $x_j$ در درخت درج شده است.

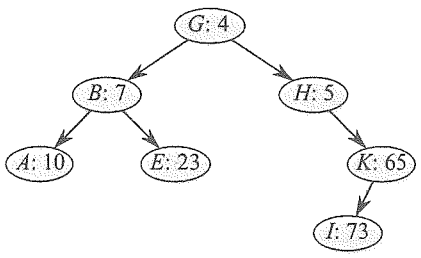
I. نشان دهید که برای یک مجموعه‌ی داده شده از گره‌های $x_1, x_2, \ldots, x_n$ با کلیدها و اولویت‌های مربوطه (که متمایز هستند)، یک treap یکتا وجود دارد.

II. نشان دهید که امیدریاضی ارتفاع یک treap برابر است با $\theta(\lg n)$، و بنابراین زمان جستجو برای یک مقدار در treap برابر است با $\theta(\lg n)$.

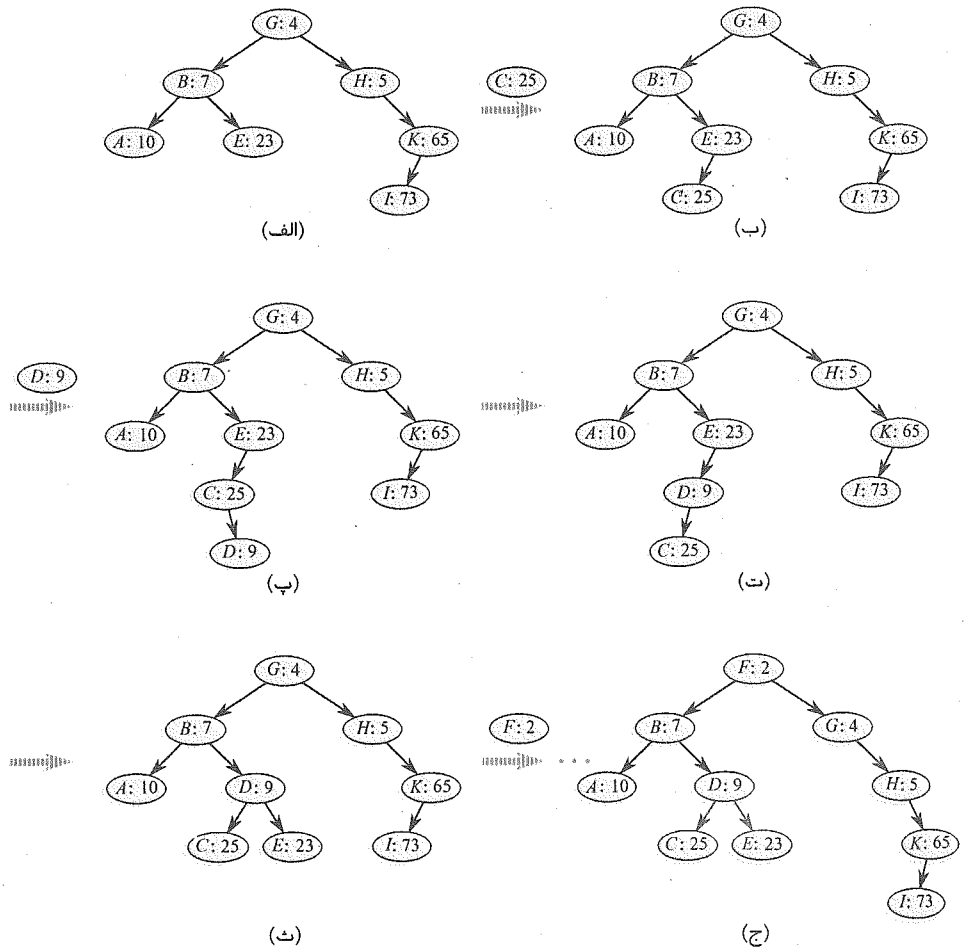
اجازه دهید ببینیم چطور می‌توان یک مقدار جدید را در یک treap درج کرد. اولین کاری که می‌کنیم این است که یک اولویت تصادفی به گره اختصاص می‌دهیم. سپس الگوریتم درج را، که آن را TREAP-INSERT می‌نامیم، فراخوانی می‌کنیم، که عملیات آن در شکل ۱۳-۱۰ مشخص شده است.

III. توضیح دهید که TREAP-INSERT چگونه کار می‌کند. ابتدا ایده را به صورت شفاهی توضیح داده و سپس برای آن شبه‌کد ارائه کنید. (*راهنمایی:* ابتدا درج معمولی درختان جستجوی دودویی را اجرا کرده، سپس با استفاده از دوران خصوصیت هرم کمینه را بازسازی کنید.)

IV. نشان دهید که امیدریاضی زمان اجرای TREAP-INSERT برابر است با $\theta(\lg n)$.

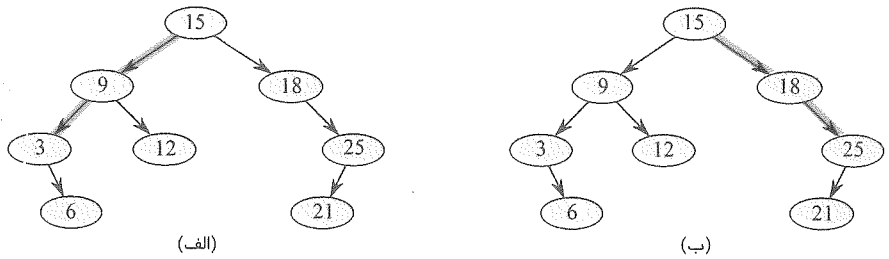



**شکل ۱۳-۹** یک treap. هر گره‌ی $x$ با $x.priority$ : $x.key$ علامت گذاری شده است. به عنوان مثال، کلید ریشه $G$ و اولویت آن ۴ است.




**شکل ۱۳-۱۰** عملیات TREAP-INSERT. (الف) treap اولیه، قبل از درج. (ب) treap بعد از درج یک گره‌ی با کلید $C$ و اولویت ۲۵. (پ)-(ت) مراحل میانی برای درج یک گره با کلید $D$ و اولویت ۹. (ث) treap بعد از این که درج در قسمت‌های (پ) و (ت) تمام شده است. (ج) treap بعد از درج یک گره با کلید $F$ و اولویت ۲.

TREAP-INSERT ابتدا یک جستجو و سپس دنباله‌ای از چندین دوران انجام می‌دهد. با این که این دو عملیات امیدریاضی زمان اجرای یکسانی دارند، در عمل هزینه‌ی آن‌ها متفاوت است. یک جستجو اطلاعات را بدون اصلاح کردن آن‌ها از treap می‌خوانند. در مقابل یک دوران اشاره‌گرهای پدر و فرزند را در treap تغییر می‌دهد. در اکثر کامپیوترها، اعمال خواندن بسیار سریع‌تر از اعمال نوشتن هستند. بنابراین خوب است که TREAP-INSERT دوران‌های کمی انجام دهد. نشان خواهیم داد که امیدریاضی تعداد دوران‌های انجام شده به یک عدد ثابت محدود می‌شود.




**شکل ۱۳-۱۱** دورهای یک درخت جستجوی دودویی. دور چپ در (الف) و دور راست در (ب) با سایه مشخص شده‌اند.

برای انجام این کار به چند تعریف نیاز داریم، که در شکل ۱۳-۱۱ نشان داده شده‌اند. *دور چپ* (left spin) یک درخت جستجوی دودویی $T$، مسیری است که از ریشه به سمت گره‌ی با کوچک‌ترین کلید در درخت می‌رود. به معنای دیگر، دور چپ مسیری از ریشه است که فقط شامل یال‌های به سمت چپ است. به طور مشابه *دور راست* درخت $T$، مسیری از ریشه است که فقط شامل یال‌های راست است. طول یک دور برابر است با تعداد گره‌های درون مسیر آن.

V. $T$ را که یک treap است دقیقاً بعد از درج $x$ با استفاده از TREAP-INSERT در آن در نظر بگیرید. فرض کنید $C$ طول دور راست زیردرخت چپ $x$، و $D$ طول دور چپ زیردرخت راست $x$ باشد. اثبات کنید که تعداد کل دوران‌هایی که حین درج $x$ انجام شده‌اند برابر است با $C+D$.

اکنون امیدریاضی مقادیر $C$ و $D$ را محاسبه می‌کنیم. بدون از دست دادن کلیت مسئله، فرض می‌کنیم که کلیدها عبارتند از ۱، ۲، ...، $n$، چرا که آن‌ها را فقط با یکدیگر مقایسه می‌کنیم.

برای گره‌های $x$ و $y$، که $x \neq y$، فرض کنید $k = x.key$ و $i = y.key$. متغیر تصادفی شاخص زیر را تعریف می‌کنیم:

$$X_{i,k} = I\{y \text{ در دور راست زیردرخت سمت چپ } x \text{ است (در } T)\}$$

VI. نشان دهید که $X_{i,k} = 1$ اگر و تنها اگر $y.priority > x.priority$، $y.key < x.key$، و برای هر $z$ که $y.key < z.key < x.key$، داشته باشیم $y.priority < z.priority$.

VII. نشان دهید که

$$\Pr\{X_{i,k} = 1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}$$

VIII. نشان دهید که

$$E[C] = \sum_{j=1}^{k-1} \frac{1}{j(j+1)} = 1 - \frac{1}{k}$$

IX. با بحثی مشابه نشان دهید که

$$E[D] = 1 - \frac{1}{n-k+1}$$

X. نتیجه بگیرید که امیدریاضی تعداد دوران‌های انجام شده هنگام درج یک گره در treap کم‌تر از ۲ است.


# ساختمان‌های داده‌ی تکمیلی

موقعیت‌هایی در دنیای مهندسی وجود دارند که در آن‌ها فقط به یک ساختمان داده‌ی «کتاب متنی» — مانند یک لیست پیوندی دوطرفه، یک جدول درهم، و یا یک درخت جستجوی دودویی — نیاز داریم، ولی بسیاری موقعیت‌های دیگر هستند که در آن‌ها باید مقداری خلاقیت به خرج دهیم. با این حال در شرایط بسیار خاص نیاز خواهید داشت که یک ساختمان داده‌ی کاملاً جدید طراحی کنید. بیشتر مواقع کافی است که با ذخیره‌ی اطلاعاتی اضافی در یک ساختمان داده‌ی کتاب متنی آن را تکمیل کنید. سپس می‌توانید اعمال جدیدی برای ساختمان داده طراحی کنید تا بتواند از کاربرد مورد نظر پشتیبانی کند. با این حال تکمیل یک ساختمان داده همیشه سرراست نیست، چرا که اطلاعات اضافه شده باید توسط اعمال معمولی بر روی ساختمان داده به هنگام سازی و حفظ شوند.

در این بخش دو ساختمان داده توصیف می‌شود که با تکمیل درختان قرمز-سیاه ساخته می‌شوند. بخش ۱۴-۱ ساختمان داده‌ای را توصیف می‌کند که از اعمال کلی شاخص‌های ترتیبی بر روی یک مجموعه‌ی پویا پشتیبانی می‌کند. به کمک این ساختمان داده می‌توانیم $i$امین عدد کوچک در یک مجموعه، و یا رتبه‌ی یک عنصر داده شده در ترتیب عام مجموعه را بیابیم. بخش ۱۴-۲ فرایند تکمیل یک ساختمان داده را به صورت تجریدی بیان می‌کند و یک قضیه فراهم می‌کند که به کمک آن تکمیل درختان قرمز- سیاه ساده‌تر می‌شود. بخش ۱۴-۳ از این قضیه برای کمک به طراحی یک ساختمان داده برای نگه داشتن یک مجموعه‌ی پویا از بازه‌ها، مانند بازه‌های زمانی استفاده می‌کند. آن گاه با داشتن یک بازه می‌توانیم به سرعت یک بازه در مجموعه بیابیم که با آن همپوشانی دارد.

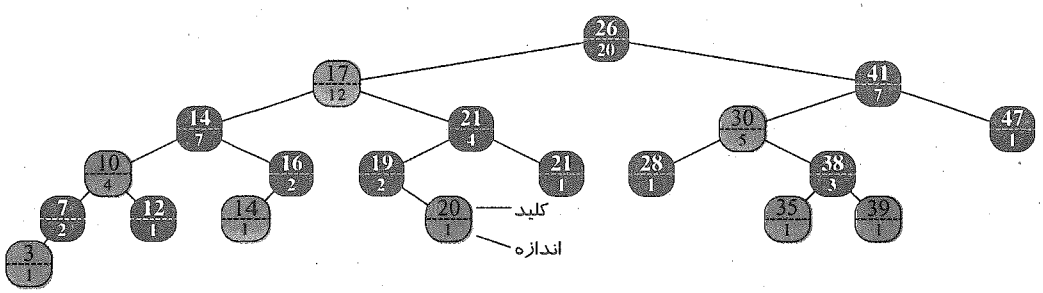
## ۱-۱۴ شاخص‌های ترتیبی پویا

در فصل ۹ مفهوم یک شاخص ترتیبی را معرفی کردیم. به خصوص، $i$امین شاخص ترتیبی یک مجموعه با $n$ عنصر، که در آن $i \in \{1,2,...,n\}$، به سادگی عنصری در مجموعه است که $i$امین کلید کوچک را دارد. دیدیم که هر شاخص ترتیبی را در یک مجموعه‌ی مرتب نشده می‌توان در زمان $O(n)$ یافت. در این بخش، خواهیم دید که چطور می‌توان درختان قرمز–سیاه را طوری اصلاح کرد که به کمک آن بتوان هر شاخص ترتیبی را در زمان $O(\lg n)$ دریافت کرد. همچنین خواهیم دید که چگونه می‌توان رتبه‌ی (rank) یک عنصر– مکان آن در ترتیب خطی مجموعه – را به همین ترتیب در زمان $O(\lg n)$ تعیین کرد.

یک ساختمان داده که می‌تواند از اعمال شاخص‌های ترتیبی با سرعت زیاد پشتیبانی کند در شکل ۱۴-۱ نشان داده شده است. یک *درخت شاخص ترتیبی* (order-statictic tree) $T$ یک درخت قرمز–سیاه است که فقط در هر گره‌ی آن اطلاعاتی اضافی ذخیره شده است. علاوه بر فیلدهای معمول درختان قرمز–سیاه $x.key$، $x.color$، $x.p$، $x.left$، و $x.right$ در یک گره‌ی $x$، یک فیلد دیگر $x.size$ هم داریم. این فیلد حاوی تعداد گره‌های (داخلی) در زیردرخت با ریشه‌ی $x$ (شامل خود $x$) است، یعنی اندازه‌ی زیردرخت. اگر اندازه‌ی نگهبان را به صورت ۰ تعریف کنیم، یعنی $T.nil.size$ را با ۰ مقداردهی کنیم، آن گاه تساوی زیر را خواهیم داشت:

$$x.size = x.left.size + x.right.size + 1$$

در یک درخت شاخص ترتیبی نیازی نیست که کلیدها مجزا باشند. (به عنوان مثال درخت شکل ۱۴-۱ دو کلید با مقدار ۱۴ و دو کلید با مقدار ۲۱ دارد.) در حضور کلیدهای مساوی، مفهوم بالا از رتبه خوش تعریف نیست. با تعریف رتبه‌ی یک عنصر به صورت مکان چاپ آن در یک پیمایش میان‌ترتیبی درخت، این ابهام در رتبه را رفع می‌کنیم. به عنوان مثال در شکل ۱۴-۱، رتبه‌ی کلید ۱۴ ذخیره شده در گره‌ی سیاه ۵، و رتبه‌ی کلید ۱۴ ذخیره شده در گره‌ی قرمز ۶ است.




**شکل ۱۴-۱** یک درخت شاخص ترتیبی، که یک درخت قرمز- سیاه تکمیل شده است. هر گره‌ی $x$ علاوه بر فیلدهای معمول خود یک فیلد $x.size$ دارد، که تعداد گره‌ها در زیردرخت با ریشه‌ی $x$ است.

## دریافت یک عنصر با یک رتبه‌ی داده شده

قبل از این که نشان دهیم چگونه می‌توان اطلاعات اندازه را حین درج و حذف حفظ کرد، اجازه دهید پیاده‌سازی دو جستجوی شاخص ترتیبی که از این اطلاعات استفاده می‌کنند را بررسی کنیم. ابتدا یک عملیات را بررسی می‌کنیم که یک عنصر با یک رتبه‌ی داده شده را بازمی‌گرداند. رویه‌ی OS-SELECT$(x, i)$ یک اشاره‌گر به گره‌ی حاوی $i$امین کلید کوچک در زیردرخت $x$ را بازمی‌گرداند. برای یافتن گره‌ی با $i$امین کلید کوچک در درخت شاخص ترتیبی $T$ رویه‌ی OS-SELECT $(T.root, i)$ را فراخوانی می‌کنیم.

```
OS-SELECT(x, i)
1  r = x.left.size + 1
2  if i == r
3      return x
4  elseif i < r
5      return OS-SELECT(x.left, i)
6  else return OS-SELECT(x.right, i − r)
```

در خط ۱رویه‌ی OS-SELECT، رتبه‌ی $x$ در زیردرخت با ریشه‌ی $x$ ($r$) را محاسبه می‌کنیم. مقدار $x.left.size$ تعداد گره‌هایی است که در پیمایش میان‌ترتیبی زیردرخت $x$ قبل از $x$ می‌آیند. بنابراین $x.left.size + 1$ رتبه‌ی $x$ در زیردرخت با ریشه‌ی $x$ است. اگر $i = r$، آن گاه گره‌ی $x$، $i$امین عنصر کوچک است، و بنابراین در خط ۳ بازگشت می‌کنیم. اگر $i < r$، آن گاه $i$امین عنصر کوچک در زیردرخت سمت چپ $x$ است، پس در خط ۵ بر روی $x.left$ بازگشت می‌کنیم. اگر $i > r$ آن گاه $i$امین عنصر کوچک در زیردرخت سمت راست $x$ قرار خواهد داشت. از آن جایی که $r$ عنصر در زیردرخت $x$ وجود دارد که در پیمایش میان‌ترتیبی قبل از زیردرخت سمت راست $x$ می‌آیند، $i$امین عنصر کوچک در زیردرخت $x$، $(i - r)$امین عنصر کوچک در زیردرخت با ریشه‌ی $x.right$ است. این عنصر به صورت بازگشتی در خط ۶ تعیین می‌شود.

برای این که ببینیم OS-SELECT چگونه عمل می‌کند، یک جستجو برای ۱۷امین عنصر کوچک در درخت شاخص ترتیبی شکل ۱۴-۱ را در نظر بگیرید. با $x$ به عنوان ریشه، که کلید آن ۲۶ است، و $i = ۱۷$ شروع می‌کنیم. چون اندازه‌ی زیردرخت سمت چپ گره‌ی با کلید ۲۶، گره‌ی با کلید ۱۲ است، رتبه‌ی آن ۱۳ خواهد بود. بنابراین می‌دانیم که گره‌ی با رتبه‌ی ۱۷، ۴ = ۱۳ – ۱۷امین عنصر کوچک در زیردرخت سمت راست ۲۶ است. پس از فراخوانی بازگشتی، $x$ گره‌ی با کلید ۴۱ است، و $i = ۴$. چون اندازه‌ی زیردرخت سمت چپ ۴۱، ۵ است، رتبه‌ی ان در زیردرخت ۶ است. بنابراین می‌دانیم که گره‌ی با رتبه‌ی ۴، ۴امین عنصر کوچک در زیردرخت چپ ۴۱ است. بعد از فراخوانی بازگشتی، $x$ گره‌ی با کلید ۳۰، و رتبه‌ی آن در زیردرخت ۲ است. بنابراین یک بار دیگر برای یافتن ۲ = ۲ – ۴امین عنصر کوچک در زیردرخت با ریشه‌ی گره‌ی با کلید ۳۸ بازگشت می‌کنیم. اکنون می‌بینیم که اندازه‌ی زیردرخت چپ آن ۱ است، که بدین معنی است که این گره دومین عنصر کوچک است. بنابراین یک اشاره‌گر به گره‌ی با کلید ۳۸ توسط رویه بازگردانده می‌شود.

چون هر فراخوانی بازگشتی یک سطح در درخت شاخص ترتیبی پایین می‌رود، کل زمان اجرای

OS-SELECT در بدترین حالت متناسب است با ارتفاع درخت. از آن جایی که درخت، یک درخت قرمز-سیاه است، ارتفاع آن برابر است با $O(\lg n)$، که در آن $n$ تعداد گره‌ها است. بنابراین زمان اجرای OS-SELECT برای یک مجموعه‌ی پویا با $n$ عنصر $O(\lg n)$ است.

## تعیین رتبه‌ی یک عنصر

با داشتن یک اشاره‌گر به یک گره‌ی $x$ در یک درخت شاخص ترتیبی $T$، رویه‌ی OS-RANK مکان $x$ در ترتیب خطی تعیین شده توسط یک پیمایش میان‌ترتیبی $T$ را بازمی‌گرداند.

```
OS-RANK(T, x)
1  r = x.left.size + 1
2  y = x
3  while y ≠ T.root
4      if y == y.p.right
5          r = r + y.p.left.size + 1
6          y = y.p
7  return r
```

رویه به صورت زیر کار می‌کند. رتبه‌ی $x$ را می‌توان به صورت تعداد گره‌های قبل از $x$ در پیمایش میان‌ترتیبی درخت، به علاوه‌ی ۱ برای خود $x$ دید. OS-RANK ثابت حلقه‌ی زیر را حفظ می‌کند.

- در آغاز هر تکرار حلقه‌ی while در خطوط ۳-۶، $r$ برابر است با رتبه‌ی $x.key$ در زیردرخت با ریشه‌ی $y$.

به صورت زیر از این ثابت حلقه استفاده می‌کنیم تا نشان دهیم که OS-RANK به درستی کار می‌کند:

- آغاز: قبل از اولین تکرار، خط ۱، $r$ را برابر با رتبه‌ی $x.key$ در زیردرخت با ریشه‌ی $x$ قرار می‌دهد. قرار دادن $y = x$ در خط ۲، صحت ثابت را در اولین باری که تست خط ۳ اجرا می‌شود، تضمین می‌کند.
- ادامه: در پایان هر تکرار حلقه‌ی while، قرار می‌دهیم $y = y.p$. بنابراین باید نشان دهیم که اگر در آغاز بدنه‌ی حلقه، $r$ رتبه‌ی $x.key$ در زیردرخت با ریشه‌ی $y$ باشد، آن گاه در پایان بدنه‌ی حلقه، $r$ رتبه‌ی $x.key$ در زیردرخت با ریشه‌ی $y.p$ است. در هر تکرار حلقه‌ی while، زیردرخت با ریشه‌ی $y.p$ را در نظر می‌گیریم. قبلاً تعداد گره‌هایی را که در پیمایش میان‌ترتیبی زیردرخت با ریشه‌ی $y$ قبل از $x$ می‌آیند محاسبه کرده‌ایم، بنابراین باید گره‌هایی را که در پیمایش میان‌ترتیبی در زیردرخت برادر $y$ قبل از $x$ می‌آیند به آن اضافه کنیم، به علاوه‌ی ۱ برای $y.p$، اگر آن هم قبل از $x$ می‌آید. اگر $y$ یک فرزند سمت چپ باشد، آن گاه نه $y.p$ و نه هیچ یک از گره‌های درون زیردرخت سمت راست $y$ قبل از $x$ نمی‌آیند، پس $r$ را تغییر نمی‌دهیم. در غیر این صورت، $y$ یک فرزند سمت راست است و تمام گره‌ها در زیردرخت سمت چپ $y.p$ قبل از $x$ می‌آیند، همین طور است خود $y.p$. بنابراین در خط ۵، $y.p.left.size + 1$ را به مقدار فعلی $r$ اضافه می‌کنیم.

• پایان: حلقه زمانی پایان می‌یابد که $y = T.root$، بنابراین زیردرخت با ریشه‌ی $y$ کل درخت است. پس مقدار $r$ برابر است با رتبه‌ی $x.key$ در کل درخت.

به عنوان یک مثال، وقتی OS-RANK را بر روی درخت شاخص ترتیبی شکل ۱۴-۱ برای یافتن رتبه‌ی گره‌ی با کلید ۳۸ اجرا می‌کنیم، دنباله‌ی زیر را برای مقادیر $y.key$ و $y$ در بالای حلقه‌ی **while** خواهیم داشت:

| تکرار | $y.key$ | $r$ |
|---|---|---|
| ۱ | ۳۸ | ۲ |
| ۲ | ۳۰ | ۴ |
| ۳ | ۴۱ | ۴ |
| ۴ | ۲۶ | ۱۷ |

رتبه‌ی ۱۷ بازگردانده می‌شود.

از آن جایی که هر تکرار حلقه‌ی **while** به زمان $O(1)$ نیاز دارد، و در هر تکرار $y$ یک سطح در درخت بالا می‌رود، در بدترین حالت زمان اجرای OS-RANK نسبت به ارتفاع درخت خطی است: $O(\lg n)$ بر روی یک درخت شاخص ترتیبی با $n$ گره.

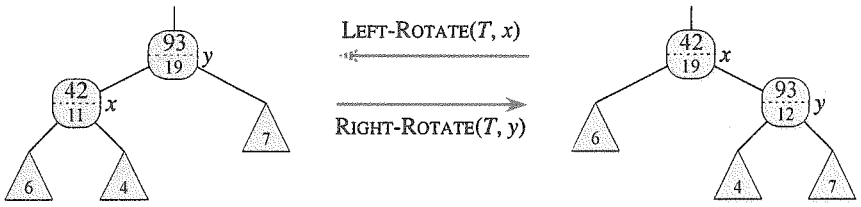
## نگه‌داری اندازه‌ی زیردرخت

با داشتن فیلد $size$ در هر گره، OS-SELECT و OS-RANK به سرعت می‌توانند اطلاعات شاخص ترتیبی را محاسبه کنند. ولی اگر نتوان این فیلدها را با اعمال اصلی که یک درخت قرمز-سیاه را اصلاح می‌کنند، به صورت کارامد حفظ کرد، کار ما بی‌ارزش خواهد بود. در این جا نشان خواهیم داد که اندازه‌ی زیردرخت‌ها را می‌توان با هر دو عملیات درج و حذف حفظ کرد، بدون این که زمان اجرای حدی آن‌ها تغییر کند.

در بخش ۱۳-۳ گفتیم که درج در یک درخت قرمز-سیاه از دو فاز تشکیل می‌شود. اولین فاز از ریشه به سمت پایین درخت حرکت، و گره‌ی جدید را به عنوان فرزند یک گره‌ی موجود اضافه می‌کند. فاز دوم در درخت به سمت بالا حرکت کرده و با تغییر رنگ‌ها و انجام دوران‌های مورد نیاز خصوصیات درختان قرمز-سیاه را حفظ می‌کند.

برای نگه داری اندازه‌ی زیردرخت، به سادگی در فاز اول $x.size$ را برای هر گره‌ی $x$ در مسیر طی شده از ریشه به برگ‌ها یکی افزایش می‌دهیم. مقدار فیلد $size$ گره‌ی جدید اضافه شده ۱ خواهد بود. از آن جایی که $O(\lg n)$ گره در مسیر طی شده وجود دارد، هزینه‌ی اضافی حفظ فیلد $size$ برابر $O(\lg n)$ است.

در فاز دوم تنها تغییرات ساختاری بر روی درخت قرمز-سیاه توسط دوران‌ها انجام می‌شوند، که حداکثر دو تا از این تغییرات خواهیم داشت. به علاوه، دوران یک عملیات محلی است: فقط فیلد $size$ دو گره نامعتبر می‌شود. اتصالی که این دوران حول آن انجام می‌شود با این دو گره مجاور است. به کد LEFT-ROTATE $(T, x)$ در بخش ۱۳-۲ خطوط زیر را اضافه می‌کنیم:




**شکل ۱٤-۲** به هنگام‌سازی اندازه‌ی زیردرخت‌ها در هنگام دوران. اتصالی که دوران حول آن انجام می‌شود با دو گره‌ای که باید فیلد *size* آن‌ها تغییر کند مجاور است. به هنگام‌سازی‌ها محلی هستند، که فقط به اطلاعات *size* ذخیره شده در $x$، $y$، و ریشه‌ی درخت‌های نشان داده شده به صورت مثلث نیاز دارند.

12 $y.size = x.size$
13 $x.size = x.left.size + x.right.size + 1$

شکل ۱۴-۲ نحوه‌ی به هنگام سازی فیلدها را نشان می‌دهد. تغییرات RIGHT-ROTATE به صورت مشابه انجام می‌شود.

از آن جایی که هنگام درج در یک درخت قرمز-سیاه حداکثر دو دوران انجام می‌شود، زمان اضافی صرف شده در به هنگام سازی فیلدهای *size* در فاز دوم فقط $O(1)$ است. بنابراین کل زمان مورد نیاز برای درج در یک درخت شاخص ترتیبی با $n$ گره $O(\lg n)$ است، که به صورت حدی معادل یک درخت قرمز- سیاه معمولی است.

حذف از یک درخت قرمز-سیاه هم از دو فاز تشکیل شده است: اولی بر روی درخت جستجوی زیربنایی عمل می‌کند، و دومی باعث می‌شود که حداکثر سه دوران رخ دهد، یا این که هیچ تغییر ساختاری بر روی درخت انجام نمی‌دهد. (بخش ۱۳-۴ را ببینید.) اولین فاز یا یک گره‌ی $y$ را از درخت حذف می‌کند، یا آن را در درخت به سمت بالا حرکت می‌دهد. برای به هنگام سازی اندازه‌ی زیردرخت‌ها، به سادگی یک مسیر از گره‌ی $y$ (که از مکان اولیه‌ی آن در درخت آغاز می‌شود) تا ریشه را پیموده، فیلد *size* هر گره بر روی مسیر را یکی کاهش می‌دهیم. از آن جایی که طول این مسیر در یک درخت قرمز-سیاه با $n$ گره $O(\lg n)$ است، زمان اضافی صرف شده برای حفظ فیلد *size* در فاز اول $O(\lg n)$ خواهد بود. $O(1)$ دوران انجام شده در فاز دوم حذف را می‌توان به روشی مشابه برای درج اداره کرد. بنابراین زمان اجرای هر دو عملیات درج و حذف، به همراه نگه داری فیلد *size*، برای یک درخت شاخص ترتیبی با $n$ گره $O(\lg n)$ است.

## تمرین‌ها

۱-۱-۱۴ نشان دهید که OS-SELECT $(T, ۱۰)$ بر روی درخت قرمز- سیاه شکل ۱۴-۱ چگونه عمل می‌کند.

۱-۱-۱۴ نشان دهید که OS-RANK $(T, x)$ بر روی درخت قرمز-سیاه شکل ۱۴-۱ و گره‌ی $x$ با $x.key = ۳۵$ چگونه عمل می‌کند.

۱۴-۱-۳ یک نسخه‌ی غیر بازگشتی از OS-SELECT بنویسید.

۱۴-۱-۴ یک رویه‌ی بازگشتی OS-KEY-RANK$(T, x)$ بنویسید که یک درخت شاخص ترتیبی $T$ و یک کلید $k$ را به عنوان ورودی دریافت کرده و رتبه‌ی $k$ را در مجموعه‌ی پویای نشان داده شده توسط $T$ بازمی‌گرداند. فرض کنید که کلیدهای $T$ یکتا هستند.

۱۴-۱-۵ با داشتن یک عنصر $x$ در یک درخت شاخص ترتیبی با $n$ گره و یک عدد طبیعی $i$، چگونه می‌توان $i$امین عنصر مابعد $x$ در ترتیب خطی درخت را در زمان $O(\lg n)$ تعیین کرد؟

۱۴-۱-۶ مشاهده کنید که هر گاه در OS-SELECT یا OS-RANK به فیلد *size* یک گره اشاره می‌شود، از آن فقط برای محاسبه‌ی یک رتبه استفاده می‌شود. بر این اساس، فرض کنید در هر گره رتبه‌ی آن در زیردرخت همان گره را ذخیره می‌کنیم. نشان دهید که چگونه می‌توان در اعمال درج و حذف این اطلاعات را حفظ کرد. (به خاطر داشته باشید که این دو عملیات می‌توانند باعث شوند که دوران رخ دهد.)

۱۴-۱-۷ نشان دهید که چگونه می‌توان از یک درخت شاخص ترتیبی برای شمردن تعداد وارونگی‌ها (مسئله‌ی ۲-۴ را ببینید) در یک آرایه با اندازه‌ی $n$ در زمان $O(n \lg n)$ استفاده کرد.

★ ۱۴-۱-۸ $n$ وتر را بر روی یک دایره در نظر بگیرید، که هر یک توسط نقاط پایانی آن تعریف شده است. یک الگوریتم با زمان $O(n \lg n)$ برای تعیین تعداد جفت‌هایی از وترها که درون دایره با یکدیگر برخورد می‌کنند ارائه کنید. (برای مثال، اگر $n$ وتر همگی قطرهایی هستند که در مرکز دایره با هم برخورد می‌کنند، جواب صحیح $\binom{n}{2}$ خواهد بود.) فرض کنید که هیچ دو وتری نقطه‌ی پایانی یکسان ندارند.

## ۱۴-۲ نحوه‌ی تکمیل یک ساختمان داده

فرآیند تکمیل یک ساختمان داده‌ی اولیه برای پشتیبانی از کاربردهای اضافی در طراحی الگوریتم‌ها زیاد رخ می‌دهد. از این فرآیند دوباره در بخش بعد برای طراحی یک ساختمان داده که بر روی بازه‌ها عمل می‌کند استفاده خواهیم کرد. در این بخش مراحل مورد نیاز در این تکمیل را بررسی می‌کنیم. همچنین یک قضیه اثبات می‌کنیم که به ما اجازه می‌دهد درختان قرمز–سیاه را به سادگی در حالت‌های بسیاری تکمیل کنیم.

تکمیل یک ساختمان داده را می‌توان به چهار مرحله تقسیم کرد:

۱. انتخاب یک ساختمان داده‌ی زیربنا.
۲. تعیین اطلاعات اضافی که باید در ساختمان داده‌ی زیربنا حفظ کرد.

۳. تحقیق این که می‌توان این اطلاعات اضافی را با اعمالی که ساختمان داده‌ی زیربنا را اصلاح می‌کنند، حفظ کرد.

٤. توسعه‌ی اعمال جدید.

مانند هر متد طراحی، نباید کورکورانه مراحل را به ترتیب داده شده دنبال کنید. اکثر کارهای طراحی به عنصر سعی و خطا نیاز دارند، و پیشرفت تمام مراحل معمولاً به صورت موازی انجام می‌شود. مثلاً تعیین اطلاعات اضافی و توسعه‌ی اعمال جدید (مراحل ۲ و ۴) بدون این که قادر باشیم اطلاعات اضافی را به صورت کارامد نگه داریم، فایده‌ای ندارد. با این حال این متد چهار مرحله‌ای یک تمرکز مناسب برای تلاش شما در تکمیل یک ساختمان داده فراهم می‌کند، و همچنین روشی است مناسب برای سازمان دهی مستندات یک ساختمان داده‌ی تکمیلی.

در بخش ۱۴-۱ هم برای طراحی درختان شاخص ترتیبی این چهار مرحله را دنبال کردیم. برای مرحله‌ی ۱، درختان قرمز- سیاه را به عنوان ساختمان داده‌ی زیربنا برگزیدیم. یک سرنخ از کارامد بودن درختان قرمز- سیاه از پشتیبانی بهینه‌ی آن‌ها از اعمال مجموعه‌های پویا بر روی یک ترتیب عام به دست می‌آید، اعمالی مانند MINIMUM، MAXIMUM، SUCCESSOR، و PREDECESSOR.

برای مرحله‌ی ۲ فیلد *size* را فراهم کردیم، که هر گره‌ی $x$ در آن اندازه‌ی زیردرخت با ریشه‌ی $x$ را ذخیره می‌کند. به طور کلی، اطلاعات اضافی اعمال را بهینه‌تر می‌کنند. مثلاً می‌توانستیم OS-SELECT و OS-RANK را فقط با استفاده از کلیدهای ذخیره شده در درخت پیاده‌سازی کنیم، ولی در آن صورت در زمان $O(\lg n)$ اجرا نمی‌شدند. بعضی مواقع اطلاعات اضافی، اطلاعات اشاره‌گری هستند، نه داده، مانند تمرین ۱۴-۲-۱.

برای مرحله‌ی ۳، اطمینان حاصل کردیم که درج و حذف می‌توانند اطلاعات فیلد *size* را نگه داشته و در عین حال در زمان $O(\lg n)$ اجرا شوند. به صورت ایده‌آل، برای حفظ اطلاعات اضافی فقط تعداد کمی از عناصر ساختمان داده باید نیاز به به هنگام سازی داشته باشند. مثلاً اگر در هر گره رتبه‌ی آن در درخت را ذخیره می‌کردیم، OS-SELECT و OS-RANK به سرعت اجرا می‌شدند، ولی درج یک عنصر کمینه‌ی جدید باعث می‌شد که اطلاعات تمام گره‌ها در درخت تغییر کند. وقتی به جای آن اندازه‌ی زیردرخت‌ها را نگه داریم، درج یک عنصر جدید باعث می‌شود که فقط اطلاعات $O(\lg n)$ گره تغییر کند.

برای مرحله‌ی ۴، عملیات OS-SELECT و OS-RANK را پیاده‌سازی کردیم. به هر حال از همان ابتدا نیاز به اعمال جدید بود که ما را به تلاش برای تکمیل ساختمان داده واداشت. به ندرت به جای توسعه‌ی اعمال جدید، از اطلاعات جدید برای تسریع اعمال موجود استفاده می‌کنیم، مانند تمرین ۱۴-۲-۱.

## تکمیل درختان قرمز- سیاه

وقتی درختان قرمز- سیاه زیربنای یک ساختمان داده هستند، می‌توانیم اثبات کنیم که انواع خاصی از

اطلاعات اضافی را همیشه می‌توان با اعمال درج و حذف حفظ کرد، که ۳ مرحله را بسیار ساده می‌کند. اثبات قضیه‌ی زیر مشابه بحث بخش ۱۴-۱ است که فیلد *size* را می‌توان برای درختان شاخص ترتیبی حفظ کرد.

**قضیه‌ی ۱۴-۱ (تکمیل یک درخت قرمز-سیاه)**

فرض کنید $f$ یک فیلد باشد که یک درخت قرمز-سیاه $T$ با $n$ گره را تکمیل می‌کند، و فرض کنید که محتوای $f$ را برای یک گره‌ی $x$ می‌توان فقط با استفاده از اطلاعات درون گره‌های $x$، $x.left$، و $x.right$، شامل $x.left.f$ و $x.right.f$ محاسبه کرد. در این صورت می‌توانیم در اعمال درج و حذف مقدار $f$ را در تمام گره‌های $T$ حفظ کنیم، بدون این که کارایی $O(\lg n)$ این اعمال به صورت حدی تغییری بکند.

*اثبات* ایده‌ی اصلی اثبات این است که تغییر یک فیلد $f$ در یک گره‌ی $x$ فقط بر روی اجداد $x$ در درخت منتشر می‌شود. یعنی ممکن است برای تغییر $x.f$ نیاز داشته باشیم $x.p.f$ را تغییر دهیم، و فقط همین؛ به هنگام سازی $x.p.f$ ممکن است نیاز به تغییر $x.p.p.f$ داشته باشد، و فقط همین؛ و به همین ترتیب تا بالای درخت. وقتی $T.root.f$ به هنگام سازی می‌شود، هیچ گره‌ی دیگری به این مقدار جدید وابسته نیست، و بنابراین فرآیند پایان می‌یابد. چون ارتفاع درخت قرمز-سیاه $O(\lg n)$ است، تغییر یک فیلد $f$ در یک گره هزینه‌ای برابر با $O(\lg n)$ برای به هنگام سازی بر روی گره‌های وابسته به آن تغییر در بر دارد.

درج یک گره‌ی $x$ در $T$ شامل دو فاز است. (بخش ۱۳-۳ را ببینید.) حین فاز اول، $x$ به عنوان فرزند یک گره‌ی موجود $x.p$ درج می‌شود. مقدار $x.f$ را می‌توان در زمان $O(1)$ محاسبه کرد، چرا که طبق فرض مقدار آن فقط به اطلاعات فیلدهای دیگر $x$ و اطلاعات فرزندان $x$ وابسته است، ولی هر دو فرزند $x$ مقادیر نگهبان $T.nil$ هستند. وقتی $x.f$ محاسبه شد، تغییر به سمت بالا در درخت منتشر می‌شود. بنابراین کل زمان مورد نیاز برای اولین فاز درج $O(\lg n)$ است. در طول فاز دوم، تنها تغییر ساختاری بر روی درخت در نتیجه‌ی دوران است. از آن جایی که در یک دوران فقط دو گره تغییر می‌کنند، کل زمان به هنگام سازی $f$ در یک دوران $O(\lg n)$ است. چون تعداد دوران‌ها در یک درج حداکثر دو است، کل زمان درج $O(\lg n)$ خواهد بود.

مانند درج، حذف هم دو فاز دارد. (بخش ۱۳-۴ را ببینید.) در فاز اول، زمانی در درخت تغییر رخ می‌دهد که عنصر حذف شده از درخت پاک شود. اگر گره‌ی حذف شده در هنگام حذف دو فرزند داشته باشد، آن گاه عنصر مابعد آن به مکان آن منتقل می‌شود. انتشار به هنگام سازی‌های $f$ حاصل از این تغییرات حداکثر $O(\lg n)$ هزینه خواهد داشت، چرا که این تغییرات درخت را به صورت محلی اصلاح می‌کنند. بازیابی درخت قرمز-سیاه در فاز دوم حداکثر به سه دوران نیاز دارد، و انتشار به هنگام سازی‌های $f$ در هر دوران حداکثر $O(\lg n)$ زمان خواهد برد. بنابراین کل زمان مورد نیاز برای حذف، مانند درج $O(\lg n)$ است. ■

در بسیاری از حالات، مانند حفظ فیلد *size* در درختان شاخص ترتیبی، هزینه‌ی به هنگام سازی بعد از یک دوران $O(1)$ است، نه $O(\lg n)$، که در اثبات قضیه‌ی ۱۴-۱ داده شد. تمرین ۱۴-۲-۴ یک مثال می‌دهد.

## تمرین‌ها

۱۴-۲-۱ نشان دهید که چگونه می‌توان از اعمال مجموعه‌های پویای MINIMUM، MAXIMUM، SUCCESSOR، و PREDECESSOR بر روی یک درخت شاخص ترتیبی در بدترین حالت در زمان $O(1)$ پشتیبانی کرد. کارایی حدی اعمال دیگر درختان شاخص ترتیبی نباید تغییر کند. (*راهنمایی:* اشاره‌گرهایی به گره‌ها اضافه کنید.)

۱۴-۲-۲ آیا می‌توان ارتفاع سیاه گره‌ها را در یک درخت قرمز-سیاه به صورت یک فیلد نگه داشت، بدون این که بر روی کارایی حدی اعمال درختان قرمز-سیاه تأثیری بگذارد؟ نشان دهید چگونه، یا بحث کنید که چرا نه. در مورد عمق گره‌ها چه می‌توان گفت؟

★ ۱۴-۲-۳ فرض کنید $\otimes$ یک عملگر دودویی شرکت پذیر باشد، و $a$ یک فیلد ذخیره شده در هر یک از گره‌های یک درخت قرمز-سیاه. فرض کنید می‌خواهیم یک فیلد جدید $f$ در هر یک از گره‌های $x$ قرار دهیم به طوری که $x.f = x_1.a \otimes x_2.a \otimes \ldots \otimes x_m.a$، که در آن $x_1, x_2, \ldots, x_m$ لیست گره‌های زیردرخت با ریشه‌ی $x$ در پیمایش میان‌ترتیبی است. نشان دهید که بعد از یک دوران می‌توان به درستی فیلد *size* را در زمان $O(1)$ به هنگام سازی کرد. بحث خود را کمی اصلاح کنید و آن را برای خصیصه‌ی *size* در درختان شاخص ترتیبی به کار ببرید.

★ ۱۴-۲-۵ می‌خواهیم یک درخت قرمز-سیاه را با یک عملیات RB-ENUMERATE$(x, a, b)$ تکمیل کنیم، که تمام کلیدهای $k$ در درخت قرمز-سیاه با ریشه‌ی $x$ را، که برای آن‌ها داریم $a \le k \le b$، در خروجی چاپ می‌کند. نشان دهید که چگونه می‌توان RB-ENUMERATE را در زمان $\theta(m + \lg n)$ پیاده‌سازی کرد، که در آن $m$ تعداد کلیدهایی است که چاپ می‌شوند، و $n$ تعداد گره‌های داخلی درخت است. (*راهنمایی:* نیازی به اضافه‌ی فیلدهای جدید به درخت قرمز- سیاه نیست.)

## ۱۴-۳ درختان بازه‌ای

در این بخش، درختان قرمز- سیاه را طوری تکمیل می‌کنیم که از اعمالی بر روی مجموعه‌های پویایی از بازه‌ها پشتیبانی کنند. یک *بازه‌ی بسته* (closed interval) یک جفت مرتب از اعداد حقیقی $[t_1, t_2]$ است، که در آن $t_1 \le t_2$. بازه‌ی $[t_1, t_2]$ نشان دهنده‌ی مجموعه‌ی $\{t \in \mathbb{R} : t_1 \le t \le t_2\}$ است. بازه‌های

*باز* (open) و *نیمه باز* (half-open) به ترتیب هر دو یا یکی از نقاط پایانی را از مجموعه حذف می‌کنند. در این بخش فرض می‌کنیم بازه‌ها بسته هستند؛ گسترش نتایج به بازه‌های باز و نیمه باز از نظر مفهومی کار سرراستی است.

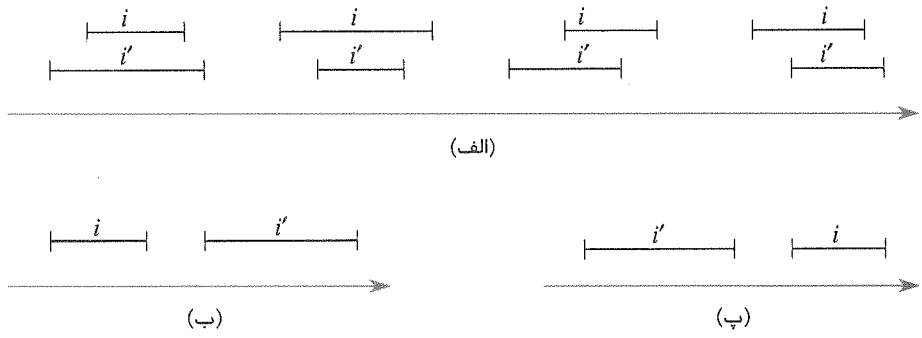
استفاده از بازه‌ها برای نمایش رخدادهایی که هر یک دوره‌ی پیوسته‌ای از زمان را اشغال می‌کنند، ساده است. برای مثال ممکن است بخواهیم پایگاه داده‌ای از بازه‌های زمانی را برای یافتن رخدادهایی که در یک بازه‌ی زمانی اتفاق افتاده‌اند، جستجو کنیم. ساختمان داده‌ی ارائه شده در این بخش یک ابزار کارامد برای حفظ چنین پایگاه داده‌ای فراهم می‌کند.

می‌توانیم یک بازه‌ی $[t_1, t_2]$ را به صورت یک شیئ $i$ با فیلدهای $i.low = t_1$ (انتهای پایینی) و $i.high = t_2$ (انتهای بالایی) نشان دهیم. می‌گوییم بازه‌های $i$ و $i'$ با یکدیگر *همپوشانی* (overlap) دارند اگر $i \cap i' \neq \emptyset$، یعنی، اگر $i.low \leq i'.high$ و $i'.low \leq i.high$. همان طور که شکل ۱۴-۳ نشان می‌دهد، هر دو بازه‌ی دلخواه $i$ و $i'$ *سه‌گانه‌ی بازه‌ها* (interval trichotomy) را ارضا می‌کنند؛ یعنی دقیقاً یکی از سه خصوصیت زیر برقرار است:

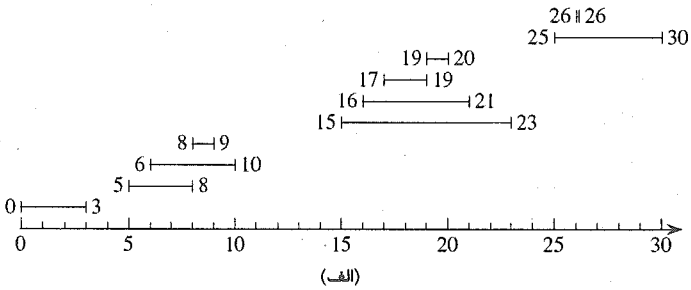
۱. $i$ و $i'$ همپوشانی دارند،
۲. $i$ در سمت چپ $i'$ است ($i.high < i'.low$)،
۳. $i$ در سمت راست $i'$ است ($i'.high < i.low$).

یک *درخت بازه‌ای* (interval tree) یک درخت قرمز– سیاه است که مجموعه‌ای پویا از عناصر را نگه می‌دارد، که در آن هر عنصر $x$ حاوی یک بازه‌ی $x.int$ است. درختان بازه‌ای از اعمال زیر پشتیبانی می‌کنند.

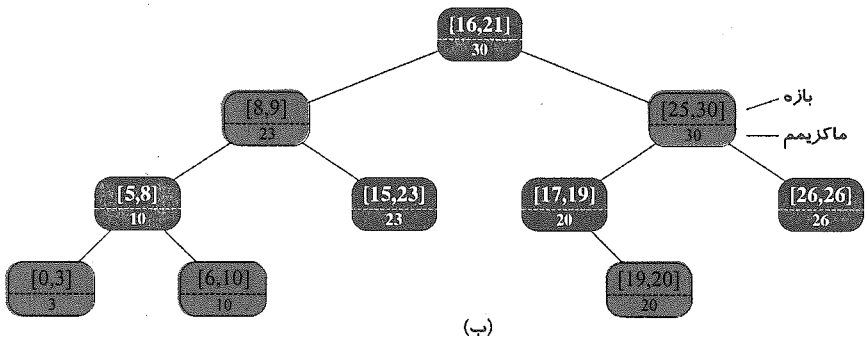
- INTERVAL-INSERT$(T, x)$ یک عنصر $x$ را به درخت بازه‌ای $T$ اضافه می‌کند، که فرض بر این است که فیلد $int$ آن حاوی یک بازه است.
- INTERVAL-DELETE$(T, x)$ عنصر $x$ را از درخت بازه‌ای $T$ حذف می‌کند.




**شکل ۱۴-۳** سه‌گانه‌ی بازه‌ها برای دو بازه‌ی بسته‌ی $i$ و $i'$. (الف) اگر $i$ و $i'$ همپوشانی داشته باشند، چهار موقعیت مختلف به وجود خواهد آمد؛ در هر یک از آن‌ها، $i.low \leq i'.high$ و $i'.low \leq i.high$. (ب) بازه‌ها همپوشانی ندارند، و $i.high < i'.low$. (پ) بازه‌ها همپوشانی ندارند، و $i'.high < i.low$.




(الف)




(ب)

**شکل ۱۴-۴** یک درخت بازه‌ای. (الف) یک مجموعه با ۱۰ بازه، که در شکل از پایین به بالا بر حسب نقاط پایانی چپ مرتب شده است. (ب) درخت بازه‌ای که این بازه‌ها را نشان می‌دهد. هر گره‌ی $x$ حاوی یک بازه است که بالای خط‌چین نشان داده شده است، و مقدار بیشینه‌ی هر نقطه‌ی انتهایی بازه در زیردرخت با ریشه‌ی $x$، که پایین خط‌چین نشان داده شده است. یک پیمایش میان‌ترتیبی بر روی درخت گره‌ها را به ترتیب نقاط پایانی چپ لیست می‌کند.

- INTERVAL-SEARCH$(T, x)$ یک اشاره‌گر به یک عنصر $x$ در درخت بازه‌ای $T$ بازمی‌گرداند به طوری که $x.int$ با بازه‌ی $i$ همپوشانی دارد، یا عنصر نگهبان $T.nil$ اگر چنین عنصری در مجموعه وجود نداشته باشد.

شکل ۱۴-۴ مشخص می‌کند که یک درخت بازه‌ای چگونه از یک مجموعه از بازه‌ها پشتیبانی می‌کند. متد چهار مرحله‌ای بخش ۱۴-۲ را که برای طراحی یک درخت بازه‌ای و اعمالی که بر روی آن اجرا می‌شوند به کار رفته، دنبال خواهیم کرد.

## مرحله‌ی ۱: ساختمان داده‌ی زیربنا

یک درخت قرمز- سیاه انتخاب می‌کنیم که در آن هر گره‌ی $x$ حاوی یک بازه‌ی $x.int$، و کلید $x$ انتهای پایانی بازه، یعنی $x.int.low$ است. بنابراین یک پیمایش میان‌ترتیبی بر روی ساختمان داده بازه‌ها را به ترتیب انتهای چپ لیست می‌کند.

### مرحله‌ی ۲: اطلاعات اضافی

علاوه بر خود بازه‌ها، هر گره‌ی $x$ حاوی یک مقدار $x.max$ است، که مقدار بیشینه در میان تمام نقاط پایانی بازه‌های ذخیره شده در زیردرخت $x$ است.

### مرحله‌ی ۳: حفظ اطلاعات

باید تحقیق کنیم که درج و حذف را می‌توان بر روی یک درخت بازه‌ای با $n$ گره در زمان $O(\lg n)$ انجام داد. می‌توانیم مقدار $x.max$ را با داشتن بازه‌ی $x.int$ و مقادیر $max$ فرزندان گره‌ی $x$ تعیین کنیم:

$$x.max = \max(x.int.high, x.left.max, x.right.max)$$

بنابراین طبق قضیه‌ی ۱۴-۱، درج و حذف در زمان $O(\lg n)$ اجرا می‌شوند. در واقع به هنگام سازی فیلد $max$ را بعد از یک دوران می‌توان در زمان $O(1)$ پیاده‌سازی کرد، همان طور که در تمرین‌های ۱۴-۲-۴ و ۱۴-۳-۱ نشان داده شده است.

### مرحله‌ی ۴: توسعه‌ی اعمال جدید

تنها عملیات جدیدی که نیاز داریم INTERVAL-SEARCH$(T, i)$ است، که یک گره در درخت $T$ می‌یابد که بازه‌ی آن با $i$ همپوشانی دارد. اگر هیچ بازه‌ای در درخت وجود نداشته باشد که با $i$ همپوشانی داشته باشد، یک اشاره‌گر به نگهبان $T.nil$ بازگردانده می‌شود.

```
INTERVAL-SEARCH(T, i)
1  x = T.root
2  while x ≠ T.nil and i does not overlap x.int
3      if x.left ≠ T.nil and x.left.max ≥ i.low
4          x = x.left
5      else x = x.right
6  return x
```

جستجو برای یک بازه که با $i$ همپوشانی دارد با $x$ به عنوان ریشه‌ی درخت آغاز می‌شود و به سمت پایین ادامه می‌یابد. این روند زمانی پایان می‌یابد که یا بازه‌ی مورد نظر یافت شود و یا این که $x$ به مقدار نگهبان $T.nil$ اشاره کند. چون هر تکرار حلقه‌ی اصلی $O(1)$ زمان می‌گیرد، و چون ارتفاع یک درخت قرمز- سیاه با $n$ گره $O(\lg n)$ است، رویه‌ی INTERVAL-SEARCH در زمان $O(\lg n)$ اجرا می‌شود.

قبل از این که ببینیم چرا INTERVAL-SEARCH به درستی کار می‌کند، اجازه دهید بررسی کنیم که بر روی درخت بازه‌ای شکل ۱۴-۴ چگونه عمل می‌کند. فرض کنید می‌خواهیم یک بازه پیدا کنیم که با بازه‌ی $i = [۲۲, ۲۵]$ همپوشانی دارد. با $x$ به عنوان ریشه‌ی درخت آغاز می‌کنیم، که حاوی $[۱۶, ۲۱]$ است و با $i$ همپوشانی ندارد. چون $x.left.max = ۲۳$ از $i.low = ۲۲$ بزرگ‌تر است، حلقه با $x$ به عنوان فرزند سمت چپ ریشه ادامه می‌یابد – گره‌ی حاوی $[۸, ۹]$، که آن هم با $i$ همپوشانی ندارد.

این بار $x.left.max = ۱۰$ کم‌تر از $i.low = ۲۲$ است، بنابراین حلقه با فرزند سمت راست $x$ به عنوان $x$ جدید ادامه می‌یابد. بازه‌ی [۱۵, ۲۳] ذخیره شده در این گره با $i$ همپوشانی دارد، بنابراین رویه این گره را بازمی‌گرداند.

به عنوان یک مثال از یک جستجوی ناموفق، فرض کنید می‌خواهیم در درخت بازه‌ای شکل ۱۴-۴ یک بازه بیابیم که با $i = [۱۴, ۱۱]$ همپوشانی دارد. یک بار دیگر با $x$ به عنوان ریشه‌ی درخت آغاز می‌کنیم. چون بازه‌ی ریشه، یعنی [۱۶, ۲۱] با $i$ همپوشانی ندارد، و چون $x.left.max = ۲۳$ بزرگ‌تر از $i.low = ۱۱$ است، به سمت راست و به گره‌ی حاوی بازه‌ی [۸, ۹] می‌رویم. بازه‌ی [۸, ۹] با $i$ همپوشانی ندارد، و $x.left.max = ۱۰$ کم‌تر از $i.low = ۱۱$ است، پس به سمت راست حرکت می‌کنیم. (توجه کنید که هیچ بازه‌ای در زیردرخت سمت راست با $i$ همپوشانی ندارد.) بازه‌ی [۱۵, ۲۳] با $i$ همپوشانی ندارد، و فرزند سمت چپ آن $T.nil$ است، پس دوباره به سمت راست می‌رویم، حلقه پایان می‌یابد، و مقدار نگهبان $T.nil$ بازگردانده می‌شود.

برای این که ببینیم چرا INTERVAL-SEARCH به درستی کار می‌کند، باید درک کنیم که چرا کافی است که یک مسیر از ریشه تا یک برگ را بررسی کنیم. ایده‌ی اصلی این است که در هر گره‌ی $x$، اگر $x.int$ با $i$ همپوشانی نداشته باشد، جستجو همیشه در مسیر ایمن ادامه می‌یابد: در صورت وجود، یک بازه‌ی همپوشانی دار را حتماً می‌توان در آن جا پیدا کرد. قضیه‌ی زیر این خصوصیت را به صورت دقیق‌تر توضیح می‌دهد.

**قضیه‌ی ۱۴-۲** هر اجرای INTERVAL-SEARCH$(T, i)$ یا یک گره بازمی‌گرداند که بازه‌ی آن با $i$ همپوشانی دارد، و یا $T.nil$ را بازمی‌گرداند و درخت $T$ هیچ گره‌ای ندارد که بازه‌ی آن با $i$ همپوشانی داشته باشد.

*اثبات* حلقه‌ی while خطوط ۲-۵ وقتی پایان می‌یابد که یا $x = T.nil$ و یا $i$ با $x.int$ همپوشانی داشته باشد. در حالت دوم، مطمئناً بازگرداندن $x$ کار درستی است. بنابراین بر روی حالت اول تمرکز می‌کنیم، که در آن حلقه‌ی while برای این پایان می‌یابد که $x = T.nil$.

از ثابت زیر برای حلقه‌ی while خطوط ۲-۵ استفاده می‌کنیم:

- اگر درخت $T$ حاوی یک بازه باشد که با $i$ همپوشانی داشته باشد، آن گاه چنین بازه‌ای در زیردرخت با ریشه‌ی $x$ قرار دارد.

از این ثابت حلقه به صورت زیر استفاده می‌کنیم:

- آغاز: قبل از اولین تکرار حلقه، خط ۱ اشاره‌گر $x$ را با ریشه‌ی درخت $T$ مقداردهی می‌کند، پس ثابت حلقه برقرار است.
- ادامه: در هر تکرار حلقه‌ی while، یا خط ۴ اجرا می‌شود و یا خط ۵. نشان می‌دهیم که در هر دو حالت ثابت حلقه حفظ می‌شود.

اگر خط ۵ اجرا شود، آن گاه به خاطر شرط انشعاب در خط ۳، داریم $x.left = T.nil$ یا

$x.left.max < i.low$. اگر $x.left = T.nil$، زیردرخت با ریشه‌ی $x.left$ مطمئناً هیچ بازه‌ای ندارد که با $i$ همپوشانی داشته باشد، و بنابراین مقداردهی $x$ با $x.right$ ثابت حلقه را حفظ می‌کند. بنابراین فرض کنید که $x.left \neq T.nil$ و $x.left.max < i.low$. همان طور که شکل ۱۴-۵ نشان می‌دهد، برای هر بازه‌ی $i'$ در زیردرخت سمت چپ $x$، داریم

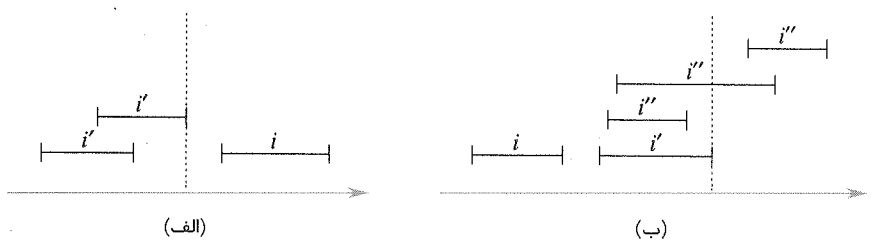
$$i'.high \leq x.left.max < i.low$$

بنابراین طبق سه‌گانه‌ی بازه‌ها، $i$ و $i'$ با هم همپوشانی ندارند. از این رو، زیردرخت سمت چپ $x$ حاوی هیچ بازه‌ای نیست که با $i$ همپوشانی داشته باشد، و مقداردهی $x$ با $x.right$ ثابت حلقه را حفظ می‌کند.

از طرف دیگر اگر خط ۴ اجرا شود، آن گاه نشان خواهیم داد که عکس نقیض ثابت حلقه برقرار است. یعنی اگر هیچ بازه‌ای که با $i$ همپوشانی داشته باشد در زیردرخت با ریشه‌ی $x.left$ موجود نباشد، آن گاه هیچ بازه‌ای که با $i$ همپوشانی دارد در هیچ کجای درخت موجود نخواهد بود. از آن جایی که خط ۴ اجرا شده است، به خاطر شرط انشعاب در خط ۳، داریم $x.left.max \geq i.low$. به علاوه طبق تعریف فیلد $max$، باید یک بازه‌ی $i'$ در زیردرخت سمت چپ $x$ موجود باشد به طوری که

$$i'.high = x.left.max \geq i.low$$

(شکل ۱۴-۵(ب) این حالت را نشان می‌دهد.) چون $i$ و $i'$ همپوشانی ندارند، و چون عبارت $i'.high < i.low$ درست نیست، از سه‌گانه‌ی بازه‌ها نتیجه می‌شود که $i'.low > i.high$. کلید درخت‌های بازه‌ای، انتهای پایینی بازه‌ها است، و بنابراین خصوصیت درخت‌های جستجو ایجاب می‌کند که برای هر بازه‌ی "$i$ در زیردرخت سمت راست $x$،




**شکل ۱۴-۵** بازه‌ها در اثبات قضیه‌ی ۱۴-۲. در هر حالت مقدار $x.left.max$ به صورت خط‌چین نشان داده شده است. (الف) جستجو به سمت راست می‌رود. هیچ بازه‌ی $i'$ی در زیردرخت سمت چپ $x$ نمی‌تواند با $i$ همپوشانی داشته باشد. (ب) جستجو به سمت چپ می‌رود. زیردرخت سمت چپ $x$ حاوی یک بازه است که با $i$ همپوشانی دارد (حالتی که نشان داده نشده است)، یا یک بازه‌ی $i'$ در زیردرخت سمت چپ $x$ وجود دارد که $i'.high = x.left.max$. چون $i$ با $i'$ همپوشانی ندارد، پس با هیچ بازه‌ی "$i$ی در زیردخت سمت راست $x$ هم همپوشانی ندارد، چرا که $i'.low \leq i".low$.

$$i.high < i'.low \\ \leq i''.low$$

طبق سه‌گانه‌ی بازه‌ها $i$ و $i''$ همپوشانی ندارند. نتیجه می‌گیریم که چه در زیردرخت سمت چپ $x$ بازه‌ای که با $i$ همپوشانی دارد، وجود داشته باشد یا نه، مقداردهی $x$ با $x.left$ ثابت حلقه را حفظ می‌کند.

- پایان: اگر حلقه زمانی که $x = T.nil$ است پایان یابد، هیچ بازه‌ای که با $i$ همپوشانی داشته باشد در زیردرخت با ریشه‌ی $x$ وجود ندارد. عکس نقیض ثابت حلقه ایجاب می‌کند که $T$ حاوی هیچ بازه‌ای نیست که با $i$ همپوشانی داشته باشد. بنابراین بازگرداندن $x = T.nil$ کار درستی است. ■

بنابراین رویه‌ی INTERVAL-SERACH به درستی کار می‌کند.

## تمرین‌ها

۱۴-۳-۱ برای LEFT-ROTATE شبه‌کدی بنویسید که بر روی گره‌ها در یک درخت بازه‌ای عمل و فیلد $max$ را در زمان $O(1)$ به هنگام سازی می‌کند.

۱۴-۳-۲ کد INTERVAL-SEARCH را طوری بازنویسی کنید که وقتی تمام بازه‌ها باز هستند به درستی کار کند.

۱۴-۳-۳ یک الگوریتم بهینه توصیف کنید که با دریافت بازه‌ی $i$، یک بازه بازمی‌گرداند که با $i$ همپوشانی داشته و انتهای پایینی آن کمینه است، و در صورت عدم وجود چنین بازه‌ای، $T.nil$ را بازمی‌گرداند.

۱۴-۳-۴ با دریافت یک درخت بازه‌ای $T$ و یک بازه‌ی $i$، توضیح دهید که چگونه می‌توان در زمان $O(\min(m, k \lg n))$ تمام بازه‌هایی در $T$ را که با $i$ همپوشانی دارند لیست کرد، که در آن $k$ تعداد بازه‌های لیست خروجی است. (*راهنمایی:* یک متد ساده چندین جستجوی انجام می‌دهد، و هنگام جستجوها، درخت را تغییر می‌دهد. یک متد کمی پیچیده‌تر اصلاً درخت را اصلاح نمی‌کند.)

۱۴-۳-۵ اصلاحاتی بر روی رویه‌های درختان بازه‌ای پیشنهاد کنید به طوری که از عملیات جدید INTERVAL-SEARCH-EXACTLY $(T, i)$ پشتیبانی کنند، که این رویه یک اشاره‌گر به یک گره‌ی $x$ در درخت بازه‌ای $T$ بازمی‌گرداند به طوری که $t.int.low = i.low$ و $t.int.high = i.high$، یا $T.nil$ اگر $T$ حاوی چنین گره‌ای نباشد. تمام اعمال، شامل INTERVAL-SEARCH-EXACLY، باید برای یک درخت با $n$ گره در زمان $O(\lg n)$ اجرا شوند.

۶-۳-۱۴ نشان دهید که چگونه می‌توان یک مجموعه‌ی پویای $Q$ از اعداد را که از عملیات MIN-GAP پشتیبانی می‌کند نگه داشت، که این عملیات اندازه‌ی اختلاف دو عدد با کم‌ترین اختلاف در $Q$ را به دست می‌دهد. به عنوان مثال اگر $Q = \{1,5,9,15,18,22\}$، آن گاه MIN-GAP$(Q)$ مقدار $۳ = ۱۵ - ۱۸$ را بازمی‌گرداند، چرا که ۱۵ و ۱۸ در میان اعداد $Q$ کم‌ترین اختلاف را دارند. اعمال INSERT، DELETE، SEARCH، و MIN-GAP را تا حد ممکن بهینه کنید، و زمان اجرای آن را تحلیل کنید.

★ ۷-۳-۱۴ پایگاه‌های داده‌ی VLSI معمولاً یک مدار را به صورت لیستی از مستطیل‌ها نشان می‌دهند. فرض کنید تمام مستطیل‌ها در خط مستقیم جهت داده شده‌اند (اضلاع موازی با محورهای $x$ و $y$)، به طوری که نمایش یک مستطیل تشکیل شده است از کمینه و بیشینه‌ی مختصات $x$ و $y$ آن مستطیل. یک الگوریتم با زمان $O(n \lg n)$ برای تعیین این که آیا در مجموعه‌ای از مستطیل‌ها دو مستطیل وجود دارند که با هم همپوشانی دارند یا خیر، ارائه کنید. نیازی نیست الگوریتم شما تمام نقاط برخورد را گزارش کند، ولی در صورتی که یک مستطیل وجود داشته باشد که به طور کامل دیر دیگری واقع شده باشد، باید وجود همپوشانی را گزارش کند، حتی اگر خطوط مرزی با هم برخورد نداشته باشند. (*راهنمایی:* یک خط «جاروب» از روی مجموعه‌ی مستطیل‌ها بگذرانید.)

## مسائل

### ۱-۱۴ *نقطه‌ی همپوشانی بیشینه*

فرض کنید می‌خواهیم اطلاعات یک نقطه‌ی همپوشانی بیشینه را در مجموعه‌ای از بازه‌ها نگه داریم – یک نقطه که با بیشترین بازه‌ها در پایگاه داده همپوشانی دارد.

I. نشان دهید که همیشه یک نقطه‌ی همپوشانی بیشینه وجود دارد که نقطه‌ی پایانی یکی از پاره‌خط‌ها است.

II. یک ساختمان داده طراحی کنید که به صورت بهینه از اعمال INTERVAL-INSERT، INTERVAL-DELETE، و FIND-POM پشتیبانی می‌کند، که این عملیات آخری یک نقطه‌ی همپوشانی بیشینه را بازمی‌گرداند. (*راهنمایی:* یک درخت قرمز-سیاه از تمام نقاط پایانی نگه دارید. یک مقدار ۱+ به هر نقطه‌ی پایانی چپ، و یک مقدار ۱- به هر نقطه‌ی پایانی راست نسبت دهید. هر گره‌ی درخت را با اطلاعاتی اضافه تکمیل کنید که نقطه‌ی همپوشانی بیشینه را نگه می‌دارد.)

### ۲-۱۴ *جایگشت جوزف*

*مسئله‌ی جوزف* (Josephus problem) به صورت زیر تعریف شده است. فرض کنید $n$ نفر در

یک دایره صف کشیده‌اند و به ما یک عدد مثبت $m \leq n$ داده شده است. با شروع از نفر اول تعیین شده، دور دایره حرکت کرده و مکرراً نفر $m$ام را حذف می‌کنیم. بعد از حذف هر نفر، شمارش دور دایره با تعداد افراد باقی مانده ادامه می‌یابد. این فرآیند ادامه می‌یابد تا تمام $n$ نفر حذف شوند. ترتیبی که افراد بر حسب آن از دایره حذف می‌شوند $(n,m)$- جایگشت جوزف ((n, m)-Josephus permutation) را برای اعداد $1,2,...,n$ تعیین می‌کند. به عنوان مثال، $(7,3)$ - جایگشت جوزف به صورت $\langle 3,6,2,7,5,1,4 \rangle$ است.

I. فرض کنید $m$ یک ثابت است. الگوریتمی با زمان $O(n)$ ارائه کنید که با دریافت یک عدد $n$، $(n,m)$-جایگشت جوزف را بازمی‌گرداند.

II. فرض کنید $m$ ثابت نیست. الگوریتمی با زمان $O(n \lg n)$ ارائه کنید که با دریافت اعداد $n$ و $m$، $(n,m)$ - جایگشت جوزف را بازمی‌گرداند.

# بخش چهارم

## تکنیک‌های پیشرفته‌ی تحلیل و طراحی

شامل فصل‌های :

۱۵ برنامه‌ریزی پویا

۱۶ الگوریتم‌های حریصانه

۱۷ تحلیل سرشکن

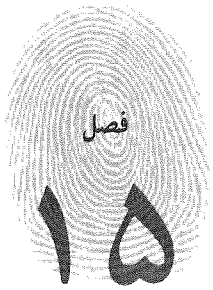
این بخش سه تکنیک مهم طراحی و تحلیل الگوریتم‌های کارا را پوشش می‌دهد: برنامه‌ریزی پویا (فصل ۱۵)، الگوریتم‌های حریصانه (فصل ۱۶)، و تحلیل سرشکن (فصل ۱۷). بخش‌های قبل تکنیک‌های پرکاربرد دیگری را معرفی کردند، مانند تقسیم و حل، تصادفی‌سازی، و جواب رابطه‌های بازگشتی. تکنیک‌های این بخش تا حدودی پیچیده‌تر هستند، ولی برای حل بهینه‌ی بسیاری از مسائل محاسباتی مفیدند. از مطالب معرفی شده در این بخش، بعدا در این کتاب استفاده خواهد شد.

برنامه‌ریزی پویا معمولا در مسائل بهینه‌سازی به کار می‌رود، که در آن‌ها باید مجموعه‌ای از انتخاب‌ها انجام گیرد تا یک جواب بهینه به دست آید. معمولا همین طور که انتخاب‌ها انجام می‌شوند، زیرمسئله‌هایی با شکل مشابه پیش می‌آیند. برنامه‌ریزی پویا زمانی مناسب است که یک زیرمسئله‌ی خاص ممکن است از بیش از یک مجموعه‌ی جزئی از انتخاب‌ها به وجود بیاید؛ تکنیک کلیدی این است که جواب هر زیرمسئله مانند این را ذخیره کرده، تا اگر دوباره به این زیرمسئله برخوردیم بتوانیم از آن استفاده کنیم. فصل ۱۵ نشان می‌دهد که چگونه این ایده‌ی ساده بعضی مواقع می‌تواند الگوریتم‌های با زمان نمایی را به الگوریتم‌های چند جمله‌ای تبدیل کند.

مانند الگوریتم‌های برنامه‌ریزی پویا، الگوریتم‌های حریصانه معمولا برای مسائل بهینه‌سازی قابل کاربرد هستند، که در آن‌ها باید مجموعه‌ای از انتخاب‌ها انجام شود تا به یک جواب بهینه برسیم. ایده‌ی یک الگوریتم حریصانه این است که هر انتخابی که انجام می‌دهیم، به صورت محلی بهینه باشد. یک مثال ساده، تبدیل سکه‌ها است: برای مینیمم کردن تعداد سکه‌های مورد نیاز برای خرد کردن یک

مقدار خاص، کافی است که مکررا بزرگ‌ترین سکه‌ی ممکن را که از مقدار باقی مانده بزرگ‌تر نیست انتخاب کنیم. بسیاری مسائل مانند این وجود دارد که در آن‌ها یک نگرش حریصانه بسیار سریع‌تر از رویکرد برنامه‌ریزی پویا به جواب بهینه ختم می‌شود. با این حال، همیشه تشخیص این که آیا رویکرد حریصانه به جواب بهینه می‌رسد، کار ساده‌ای نیست. فصل ۱۶ نظریه‌ی ماترویدها (matroid theory) را بازبینی می‌کند، که یک پایه‌ی ریاضی ارائه می‌کند که به ما کمک می‌کند نشان دهیم که یک الگوریتم حریصانه به جواب بهینه ختم می‌شود.

تحلیل سرشکن ابزاری است برای تحلیل الگوریتم‌هایی که در آن‌ها دنباله‌ای از اعمال مشابه انجام می‌شود. به جای این که برای تخمین هزینه‌ی دنباله‌ی این اعمال، کران هزینه‌ی هر یک از عملیات را به صورت جداگانه تعیین کنیم، با استفاده از تحلیل سرشکن می‌توانیم کران واقعی کل دنباله را تعیین کنیم. یکی از دلایلی که این ایده می‌تواند مؤثر باشد این است که ممکن است غیر ممکن باشد که همه‌ی اعمال دنباله در بدترین حالت زمان خود اجرا شوند. در حالی که بعضی از اعمال پرهزینه هستند، بسیاری از اعمال انجام شده می‌توانند کم هزینه باشند. با این حال، تحلیل سرشکن فقط یک ابزار برای تحلیل نیست، بلکه ابزاری است برای تفکر در مورد طراحی الگوریتم‌ها، چرا که طراحی یک الگوریتم و تحلیل زمان اجرای آن معمولا به شدت به هم وابسته هستند.




# برنامه‌ریزی پویا

## ۱۵-۰ مقدمه

برنامه‌ریزی پویا مانند متد تقسیم و حل، مسائل را به وسیله‌ی ترکیب جواب زیرمسئله‌ها حل می‌کند. (در این متن «برنامه‌ریزی» به متد برنامه‌ریزی جدولی اشاره دارد، و نه به برنامه‌نویسی در کامپیوتر.) همان طور که در فصل ۲ و ۴ دیدیم، الگوریتم‌های تقسیم و حل مسئله را به زیرمسئله‌های مستقل تقسیم می‌کنند، زیرمسئله‌ها را به صورت بازگشتی حل می‌کند، و سپس جواب زیرمسئله‌ها را با هم ترکیب می‌کند تا جواب مسئله‌ی کلی به دست آید. در مقابل برنامه‌ریزی پویا زمانی کاربرد دارد که زیرمسئله‌ها با هم همپوشانی دارد، یعنی وقتی زیرمسئله‌ها، خود زیرمسئله‌های مشترکی دارند. در چنین موقعیتی، یک الگوریتم تقسیم و حل بیشتر از مقدار مورد نیاز کار انجام می‌دهد، و زیرمسئله‌های مشترک را دوباره و دوباره حل می‌کند. یک الگوریتم برنامه‌ریزی پویا، هر زیرمسئله را فقط یک بار حل، و جواب آن را در یک جدول ذخیره می‌کند، و از محاسبه‌ی دوباره‌ی زیرمسئله، هر بار که زیرمسئله پیش می‌آید جلوگیری می‌کند.

برنامه‌ریزی پویا معمولاً برای حل *مسائل بهینه‌سازی* به کار می‌رود. در چنین مسائلی ممکن است جواب‌های زیادی وجود داشته باشد. هر جواب مقداری دارد، و هدف ما این است که جوابی با مقدار بهینه (بیشینه یا کمینه) بیابیم. به چنین جوابی، یک جواب بهینه برای مسئله می‌گوییم، در مقابل جواب بهینه‌ی یکتا، چرا که ممکن است چندین جواب برای مسئله موجود باشد که مقدار آن بهینه است.

توسعه‌ی یک الگوریتم برنامه‌ریزی پویا را می‌توان به چهار مرحله تقسیم کرد.

۱. توصیف ساختار یک جواب بهینه.
۲. تعیین مقدار جواب بهینه به صورت بازگشتی.
۳. محاسبه‌ی مقدار یک جواب بهینه به سبک پایین به بالا (bottom-up).
٤. ساختن یک جواب بهینه از اطلاعات محاسبه شده.

مراحل ۱-۳ پایه‌های یک جواب برنامه‌ریزی پویا به یک مسئله را تشکیل می‌دهند. اگر فقط مقدار یک جواب بهینه به مسئله نیاز است، و نه خود جواب، می‌توان از مرحله‌ی ۴ صرف نظر کرد. معمولاً وقتی می‌خواهیم مرحله‌ی ۴ را انجام دهیم، مقداری اطلاعات اضافی در مرحله‌ی ۳ نگه می‌داریم تا ساختن جواب در مرحله‌ی ۴ ساده‌تر شود.

در بخشی که در ادامه خواهد آمد، از متدهای برنامه‌ریزی پویا برای حل چند مسئله‌ی بهینه‌سازی استفاده خواهد شد. بخش ۱۵-۱ یک مسئله‌ی برش یک میله به میله‌های با طول کوتاه‌تر را بررسی می‌کند، به طوری که ارزش کل آن‌ها بیشینه شود. مسئله‌ی بخش ۱۵-۲، مسئله‌ی ضرب زنجیره‌ای از ماتریس‌ها است به طوری که تعداد ضرب‌های اسکالر انجام شده در کل کمینه شود. با داشتن این نمونه‌ها از برنامه‌ریزی پویا، در بخش ۱۵-۳ دو صفت کلیدی بررسی می‌شود که یک مسئله باید داشته باشد تا بتوان برای حل آن از تکنیک برنامه‌ریزی پویا استفاده کرد. سپس در بخش ۱۵-۴ خواهیم دید که چطور می‌توان با داشتن دو دنباله، بلندترین زیردنباله‌ی مشترک آن‌ها را پیدا کرد. نهایتاً در بخش ۱۵-۵ برای ساختن درختان جستجوی دودویی بهینه با داشتن توزیع خاصی از کلیدها استفاده خواهد شد.

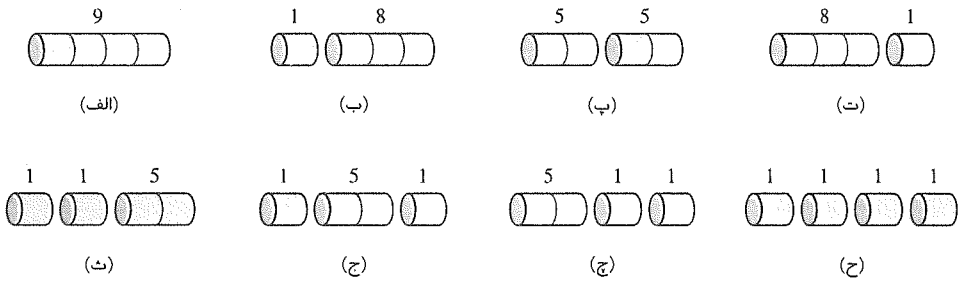
## ۱۵-۱ برش میله

اولین مثال، از برنامه‌ریزی پویا برای حل یک مسئله‌ی ساده استفاده می‌کند که در آن می‌خواهیم تصمیم بگیریم که یک میله‌ی فولادی را از کجا برش دهیم. مؤسسه‌ی Serling، میله‌های بلند فولاد را خریداری کرده، آن‌ها را برش می‌دهد، و سپس میله‌های کوچک‌تر را به فروش می‌رساند. هزینه‌ی برش‌ها ناچیز است. مدیریت مؤسسه‌ی Serling می‌خواهد بداند که بهترین روش برای برش میله‌ها چیست.

فرض کنید برای $i = 1,2,...$ می‌دانیم که مؤسسه‌ی Serling برای فروش میله‌ای به طول $i$ اینچ، $p_i$ دلار دریافت می‌کند. طول میله‌ها همیشه عددی صحیح (به اینچ) است. شکل ۱۵-۱ یک جدول قیمت نمونه را نشان می‌دهد.

| طول $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| قیمت $p_i$ | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 30 |

**شکل ۱۵-۱** یک جدول قیمت نمونه برای میله‌ها. هر میله‌ی با $i$ اینچ طول، درآمد $p_i$ دلار را برای کمپانی به همراه دارد.




**شکل ۱۵-۳** هشت روش ممکن برای برش یک میله به طول ۴. بالای هر قطعه، قیمت آن قطعه مشخص شده است، مبنی بر جدول قیمت نمونه در شکل ۱۵-۱. استراتژی بهینه، بخش (پ) است — برش میله به دو میله‌ی با طول ۲ — که قیمت کل را به ۱۰ می‌رساند.

می‌توانیم یک میله با طول $n$ را به $2^{n-1}$ روش مختلف برش دهیم، چرا که برای هر نقطه‌ای در فاصله‌ی $i$ از انتهای سمت چپ، برای $i = 1,2,...,n-1$، یک انتخاب مستقل مبنی بر برش یا عدم برش در آن نقطه داریم.[1] تقسیم به قطعه‌های کوچک‌تر را با استفاده از نماد جمع معمولی نشان داد،

بنابراین $7 = 2+2+3$ نشان دهنده‌ی میله‌ای با طول ۷ است که به سه قطعه تقسیم شده است— دو قطعه با طول ۲ و یکی با طول ۳. اگر یک جواب بهینه، میله را به $k$ قطعه تقسیم کند، برای یک $1 \le k \le n$، آن گاه تقسیم‌بندی بهینه‌ی میله به صورت

$$n = i_1 + i_2 + ... + i_k$$

به قطعه‌های با طول $i_1$، $i_2$، ...، $i_k$، بیشترین درآمد به صورت

$$r_n = p_{i_1} + p_{i_2} + ... + p_{i_k}$$

را به همراه خواهد داشت. برای مثال نمونه‌ی داده شده، می‌توانیم درآمدهای بهینه‌ی $r_i$، برای $i = 1,2,...,10$ را به صورت دستی تعیین کنیم، به همراه تقسیم‌بندی‌های بهینه‌ی مربوطه:

$r_1 = 1$ از جواب $1 = 1$ (بدون برش)،
$r_2 = 5$ از جواب $2 = 2$ (بدون برش)،
$r_3 = 8$ از جواب $3 = 3$ (بدون برش)،
$r_4 = 10$ از جواب $4 = 2+2$ ،
$r_5 = 13$ از جواب $5 = 2+3$،
$r_6 = 17$ از جواب $6 = 6$ (بدون برش)،

---

[1] اگر بخواهیم که اندازه‌ی قطعه‌ها به ترتیب صعودی باشد، تعداد روش‌های ممکن کم‌تر خواهد شد. برای $n = 4$، فقط ۵ روش برای این کار خواهیم داشت: بخش‌های (الف)، (ب)، (پ)، (ث)، و (ج) در شکل ۱۵-۲. تعداد این روش‌ها، *تابع تقسیم* (partition function) نام دارد، که تقریباً برابر است با $e^{\pi\sqrt{2n/3}} / 4n\sqrt{3}$. این مقدار کم‌تر از $2^{n-1}$ است، ولی همچنان بسیار بزرگ‌تر از تمام توابع چندجمله‌ای برحسب $n$. ولی بیشتر از این در این مورد بحثی نخواهد شد.

$r_۷ = ۱۸$ از جواب $۶+۱=۷$ یا $۳+۲+۲=۷$،
$r_۸ = ۲۲$ از جواب $۶+۲=۸$،
$r_۹ = ۲۵$ از جواب $۶+۳=۹$،
$r_{۱۰} = ۳۰$ از جواب $۱۰=۱۰$ (بدون برش)،

به طور کلی، می‌توانیم مقادیر $r_n$ را برای $n \geq ۱$ بر حسب درآمدهای بهینه‌ی میله‌های کوچک‌تر بنویسیم:

$$r_n = \max(p_n, r_۱ + r_{n-۱}, r_۲ + r_{n-۲}, \ldots, r_{n-۱} + r_۱) \qquad (۱-۱۵)$$

آرگومان اول، $p_n$، متناظر است با عدم برش، و فروش میله‌ی با طول $n$ به همان شکلی که هست. $n-۱$ آرگومان دیگر تابع max متناظرند با درآمد بیشینه‌ی به دست آمده از یک برش اولیه‌ی میله به دو قطعه با اندازه‌های $i$ و $n-i$، برای هر $i = ۱,۲,\ldots,n-۱$، و سپس برش بهینه‌ی هر یک از آن دو میله و به دست آوردن سود $r_i$ و $r_{n-i}$. از آن جایی که از قبل نمی‌دانیم کدام مقدار $i$ درآمد را بیشینه می‌کند، باید تمام مقادیر ممکن $i$ را در نظر بگیریم، و مقداری را انتخاب کنیم که بیشترین درآمد را به ما می‌دهد. همچنین، گزینه‌ی دیگری که در دست داریم این است که در صورتی که فروش میله به همان شکلی که هست، بدون برش، بیشترین درآمد را به دست می‌دهد، هیچ یک از $i$ها را انتخاب نکنیم.

توجه کنید که برای حل مسئله‌ی اصلی با اندازه‌ی $n$، مسئله‌های کوچک‌تری از همان نوع را حل می‌کنیم. وقتی اولین برش را انجام دادیم، می‌توانیم دو قطعه‌ی به دست آمده را نمونه‌های مستقلی از همان مسئله‌ی برش میله بدانیم. جواب بهینه‌ی کلی، جواب بهینه‌ی دو زیرمسئله‌ی مشابه را با هم ترکیب می‌کند، که در آن درآمد هر یک از آن دو قطعه بیشینه شده است. می‌گوییم مسئله‌ی برش میله دارای *زیرساختار بهینه* (optimal substructure) است: جواب‌های بهینه به یک مسئله از جواب‌های بهینه از زیرمسئله‌ها تشکیل شده‌اند، که آن‌ها را به صورت مستقل حل می‌کنیم.

در یک روش مشابه، ولی کمی ساده‌تر برای تشکیل یک ساختار بازگشتی برای مسئله‌ی برش میله، یک تجزیه را به صورت یک قطعه‌ی اولیه با طول $i$ که از سمت چپ بریده شده می‌بینیم، به همراه بقیه‌ی میله به طول $n-i$. فقط قطعه‌ی سمت راست، و نه قطعه‌ی اول، باید دوباره برش داده شود. می‌توانیم هر قطعه‌ای به طول $n$ را به این صورت ببینیم: به صورت یک قطعه‌ی اولیه و سپس تجزیه‌ی بقیه‌ی میله. در این روش، وقتی که جواب به صورت بدون برش است، می‌توانیم بگوییم که اندازه‌ی قطعه‌ی اول $i = n$ است، با درآمد $p_n$، و اندازه‌ی باقی میله ۰ است، با درآمد متناظر $r_۰ = ۰$. بنابراین نسخه‌ی ساده‌تر زیر را از تساوی (۱-۱۵) به دست می‌آوریم:

$$r_n = \max_{۱ \leq i \leq n} (p_i + r_{n-i}) \qquad (۲-۱۵)$$

در این فرمول‌بندی، یک جواب بهینه در خود فقط جواب *تنها* زیرمسئله‌ی مشابه – باقی میله – را دارد، و نه دو زیرمسئله.

## پیاده‌سازی بازگشتی پایین به بالا

رویه‌ی زیر، محاسبه‌ی ضمنی درون تساوی (۱۵-۲) را به شکل سرراست، بازگشتی، و بالا به پایین انجام می‌دهد.

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n-i))
6  return q
```

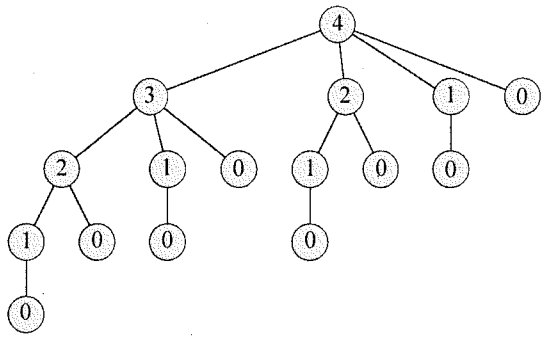
رویه‌ی CUT-ROD به عنوان ورودی یک آرایه‌ی $p[1..n]$ از قیمت‌ها را دریافت می‌کند، به همراه یک عدد صحیح $n$، و بیشینه‌ی درآمد ممکن برای یک میله با طول $n$ را بازمی‌گرداند. اگر $n = 0$، هیچ درآمدی نخواهیم داشت، و بنابراین CUT-ROD در خط ۲ مقدار ۰ را بازمی‌گرداند. در خط ۳ بیشینه‌ی درآمد $q$ با $-\infty$ مقداردهی اولیه می‌شود، به طوری که حلقه‌ی for در خطوط ۴-۵ بتواند به درستی مقدار $q = \max(q, p[i] + \text{CUT-ROD}(p, n-i))$ را محاسبه کند؛ سپس خط ۶ این مقدار را بازمی‌گرداند. یک استقرای ساده بر روی $n$، با استفاده از تساوی (۱۵-۲)، نشان می‌دهد که این جواب برابر است با همان جواب مورد نظر $r_n$.

اگر رویه‌ی CUT-ROD را به زبان برنامه‌نویسی دلخواه خود کد کرده و آن را روی کامپیوتر خود اجرا کنید، خواهید دید که برای ورودی‌های کمی بزرگ، برنامه‌ی شما به زمان بسیار زیادی برای اجرا نیاز دارد. برای $n = 40$، خواهید دید که اجرای برنامه‌ی شما حداقل چند دقیقه، و شاید حتی بیش از یک ساعت، به طول می‌انجامد. در واقع خواهید دید که هر بار که $n$ را یکی افزایش می‌دهید، زمان اجرای برنامه‌ی شما تقریباً دوبرابر می‌شود.

چرا CUT-ROD این قدر ناکارامد است؟ مشکل این جا است که CUT-ROD به صورت بازگشتی مکرراً خود را با پارامترهای یکسان فراخوانی می‌کند: حل چندباره‌ی زیرمسئله‌های تکراری. شکل ۱۵-۳ نشان می‌دهد که برای $n = 4$ چه رخ می‌دهد: CUT-ROD$(p, n)$ رویه‌ی CUT-ROD$(p, n-1)$ را برای $n = 1, 2, \ldots, n$ فراخوانی می‌کند. به طور مشابه، CUT-ROD$(p, n)$ رویه‌ی CUT-ROD$(p, j)$ را برای $j = 0, 1, \ldots, n-1$ فراخوانی می‌کند. وقتی این فرایند بازگشتی، باز می‌شود، مقدار کار انجام شده بر حسب تابعی از $n$ به صورت نمایی رشد می‌کند.

برای تحلیل زمان اجرای CUT-ROD، فرض کنید $T(n)$ نشان دهنده‌ی تعداد کل فراخوانی‌های انجام شده‌ی CUT-ROD در حالتی باشد که پارامتر دوم آن برابر $n$ است. این عبارت برابر است با تعداد گره‌های درون زیردرخت که برچسب ریشه‌ی آن‌ها در درخت بازگشت، $n$ است. بنابراین $T(0) = 1$ و

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \qquad (۱۵-۳)$$




**شکل ۱۵-۳** درخت بازگشت نشان دهنده‌ی فراخوانی‌های بازگشتی حاصل از فراخوانی CUT-ROD$(p,n)$ برای $n=۴$. برچسب هر گره، اندازه‌ی زیرمسئله‌ی متناظر را نشان می‌دهد، و یک یال از یک پدر با برچسب $s$ به یک فرزند با برچسب $t$ متناظر است با یک برش اولیه با اندازه‌ی $s-t$ و یک زیرمسئله‌ی باقی مانده با اندازه‌ی $t$. یک مسیر از ریشه به یک برگ متناظر است با یکی از $۲^{n-۱}$ روش برش یک میله با طول $n$. به طول کلی، این درخت بازگشتی حاوی $۲^n$ گره و $۲^{n-۱}$ برگ است.

۱ اولیه برای فراخوانی ریشه است، و عبارت $T(j)$ تعداد فراخوانی‌های (شامل فراخوانی‌های بازگشتی) حاصل از فراخوانی CUT-ROD$(p,n-i)$ را می‌شمارد، که در آن $j=n-i$. همان طور که تمرین ۱۵-۱-۱ از شما می‌خواهد نشان دهید،

$$T(n)=۲^n \tag{۱۵-۴}$$

و بنابراین زمان اجرای CUT-ROD بر حسب $n$ از مرتبه‌ی نمایی است.

طبق تجربیات گذشته، این زمان اجرای نمایی چندان دور از انتظار نیست. CUT-ROD صریحاً تمام $۲^{n-۱}$ حالت ممکن برای برش یک میله به طول $n$ را در نظر می‌گیرد. درخت فراخوانی‌های بازگشتی $۲^{n-۱}$ برگ دارد، یکی برای هر روش ممکن برش میله. برچسب‌های روی مسیر ساده‌ی از ریشه به یک برگ نشان دهنده‌ی اندازه‌ی هر قطعه‌ی باقی‌مانده در سمت راست قبل از هر برش است. یعنی این برچسب‌ها نقاط برش مربوطه را به دست می‌دهند، که از انتهای راست میله اندازه‌گیری شده‌اند.

## استفاده از برنامه‌ریزی پویا برای برش بهینه‌ی میله

اکنون نشان می‌دهیم که چطور می‌توان با استفاده از برنامه‌ریزی پویا، CUT-ROD را به یک برنامه‌ی بهینه تبدیل کرد.

متد برنامه‌ریزی پویا به صورت زیر کار می‌کند. وقتی مشاهده کردیم که یک راه حل بازگشتی ساده‌لوحانه ناکارامد است، چون زیرمسئله‌های یکسانی را چندین بار حل می‌کند، کاری می‌کنیم که هر زیرمسئله فقط یک *بار* حل شود، و سپس جواب آن‌ها را ذخیره می‌کنیم. اگر بعداً دوباره نیاز داشتیم که به جواب این زیرمسئله رجوع کنیم، به جای این که دوباره آن را محاسبه کنیم، به سادگی می‌توانیم جواب را از حافظه بازیابی کنیم. بنابراین برنامه‌ریزی پویا از حافظه‌ی اضافی برای صرفه‌جویی در زمان محاسبه استفاده می‌کند؛ نمونه‌ای از *سبک سنگین زمان و حافظه* (time-memory trade-off). این

صرفه‌جویی ممکن است بسیار قابل توجه باشد: یک جواب با زمان نمایی می‌تواند تبدیل به یک جواب با زمان چندجمله‌ای شود. یک رویکرد برنامه‌ریزی پویا در زمان چندجمله‌ای اجرا می‌شود اگر تعداد زیرمسئله‌های *مجزا*، نسبت به اندازه‌ی ورودی از مرتبه‌ی چندجمله‌ای باشد، و بتوانیم هر زیرمسئله را در زمان چندجمله‌ای حل کنیم.

معمولاً دو روش معادل برای پیاده‌سازی یک رویکرد برنامه‌ریزی پویا وجود دارد. در این جا هر دو را با مثال برش میله نشان خواهیم داد.

رویکرد اول عبارت است از رویکرد *بالا به پایین با به‌خاطرسپاری* (top-down with memoization[1]). در این روش، مانند روش معمولی رویه را به صورت بازگشتی می‌نویسیم، ولی آن را طوری اصلاح می‌کنیم که نتیجه‌ی هر زیرمسئله را (معمولاً در یک آرایه و یا یک جدول درهم) ذخیره کند. اکنون، رویه ابتدا چک می‌کند که آیا این زیرمسئله را قبلاً حل کرده است یا نه. اگر قبلاً حل شده بود، مقدار ذخیره شده را بازمی‌گرداند، و محاسبات انجام شده‌ی جدید را هم ذخیره می‌کند؛ در غیر این صورت، رویه مقدار مورد نظر را طبق معمول محاسبه می‌کند. می‌گوییم که رویه‌ی بازگشتی *خاطره‌دار* (memorized) شده است؛ یعنی نیتجه‌ی محاسباتی که قبلاً انجام داده است را «به خاطر می‌آورد».

رویکرد دوم، *متد پایین به بالا* (bottom-up method) است. این رویکرد معمولاً به یک مفهوم طبیعی از «اندازه»ی یک زیرمسئله وابسته است، به طوری که حل هر زیرمسئله‌ی خاص فقط به حل زیرمسئله‌های «کوچک‌تر» بستگی دارد. در این روش، زیرمسئله‌ها را بر حسب اندازه مرتب کرده و به ترتیب، با شروع از کوچک‌ترین، آن‌ها را حل می‌کنیم. زمانی که می‌خواهیم یک زیرمسئله را حل کنیم، تمام زیرمسئله‌های کوچک‌تری را که جواب این مسئله به آن‌ها بستگی دارد، قبلاً حل، و جواب آن‌ها را ذخیره کرده‌ایم. هر زیرمسئله را فقط یک بار حل می‌کنیم، و وقتی برای اولین بار آن را می‌بینیم، قبلاً تمام زیرمسئله‌های پیش‌نیاز آن را حل کرده‌ایم.

این دو رویکرد به الگوریتم‌هایی با زمان اجرای یکسان ختم می‌شوند، غیر از حالت‌های بسیار خاص که در آن‌ها رویکرد بالا به پایین، تمام زیرمسئله‌های ممکن را بررسی نمی‌کند. زمان اجرای رویکرد پایین به بالا معمولاً فاکتورهای ثابت بسیار بهتری دارد، چرا که سربار کم‌تری برای فراخوانی رویه‌ها دارد.

در زیر شبه‌کد رویه‌ی CUR-ROD به صورت بالا به پایین را می‌بینیم، که با استفاده از به خاطرسپاری نوشته شده است:

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

[1] این یک اشتباه تایپی نیست. کلمه‌ی مورد نظر واقعاً *memoization* است، نه *memorization*. کلمه‌ی *memoization* از ریشه‌ی memo می‌آید (به معنی یادداشت)، چرا که در این تکنیک ما یادداشت‌هایی را ذخیره و بعداً به آن‌ها رجوع کنیم.

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n-i, r))
8  r[n] = q
9  return q
```

ابتدا رویه‌ی اصلی MEMOIZED-CUT-ROD یک آرایه‌ی کمکی جدید $r[0 .. n]$ را با $-\infty$ مقداردهی می‌کند، که یک انتخاب مناسب است برای نشان دادن مقدار «نامعلوم». (مقادیر معلوم درآمد همیشه نامنفی هستند.) سپس رویه‌ی کمکی MEMOIZED-CUT-ROD-AUX فراخوانی می‌شود.

رویه‌ی MEMOIZED-CUT-ROD-AUX فقط یک نسخه‌ی خاطره‌دار از رویه‌ی قبلی CUT-ROD است. این رویه ابتدا در خط ۱ چک می‌کند که آیا مقدار مورد نظر قبلاً محاسبه شده است یا نه، و اگر این طور بود، در خط ۲ آن را بازمی‌گرداند. در غیر این صورت، خطوط ۳–۷ مقدار مورد نظر $q$ را به روش معمول محاسبه می‌کنند، خط ۸ آن را $r[n]$ ذخیره می‌کند، و خط ۹ آن را بازمی‌گرداند.

روش پایین به بالا از این هم ساده‌تر است:

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j-i])
7      r[j] = q
8  return r[n]
```

برای رویکرد پایین به بالای برنامه‌ریزی پویا، BOTTOM-UP-CUT-ROD از ترتیب طبیعی زیرمسئله‌ها استفاده می‌کند: یک مسئله با اندازه‌ی $i$ «کوچک‌تر» از یک زیرمسئله با اندازه‌ی $j$ است اگر $i < j$. بنابراین، رویه زیرمسئله‌های با اندازه‌ی $j = 0,1,...,n$ را به همین ترتیب حل می‌کند.

خط ۱ رویه‌ی BOTTOM-UP-CUT-ROD یک آرایه‌ی جدید $r[0 .. n]$ می‌سازد، که در آن نتیجه‌ی زیرمسئله‌ها ذخیره خواهد شد، و خط ۲، $r[0]$ را با ۰ مقداردهی اولیه می‌کند، چرا که از یک میله با طول ۰ هیچ درآمدی حاصل نخواهد شد. خطوط ۳–۶ تمام زیرمسئله‌های با اندازه‌ی $j$ را، برای $j = 1,2,...,n$، به ترتیب صعودی اندازه حل می‌کنند. رویکرد حل یک مسئله‌ی خاص با اندازه‌ی $j$ درست مانند روش استفاده شده در CUT-ROD است، با این تفاوت که اکنون خط ۶، به جای فراخوانی بازگشتی برای محاسبه‌ی جواب زیرمسئله با اندازه‌ی $j - i$، مستقیماً به ورودی $r[j - i]$ مراجعه می‌کند. خط ۷ جواب زیرمسئله با اندازه‌ی $j$ را در $r[j]$ ذخیره می‌کند. نهایتاً، خط ۸، $r[n]$ را

بازمی‌گرداند، که برابر است با مقدار بهینه‌ی $r_n$.

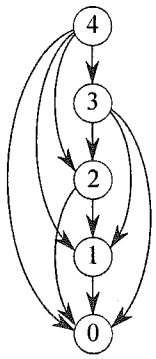
نسخه‌های پایین به بالا و بالا به پایین، زمان حدی یکسانی دارند. زمان اجرای رویه‌ی BOTTOM-UP-CUT-ROD برابر است با $\theta(n^2)$، به خاطر حلقه‌ی دوتایی تودرتو. تعداد تکرارهای حلقه‌ی for داخلی آن در خطوط ۵-۶، یک سری حسابی را تشکیل می‌دهد. زمان اجرای نسخه‌ی بالا به پایین، MEMOIZED-CUT-ROD هم $\theta(n^2)$ است، هرچند که درک این زمان اجرا ممکن است کمی سخت‌تر باشد. چون یک فراخوانی بازگشتی برای حل یک زیرمسئله‌ی قبلاً حل شده، بی‌درنگ بازمی‌گردد، MEMOIZED-CUT-ROD هر زیرمسئله را فقط یک بار حل می‌کند. این رویه، زیرمسئله‌های با اندازه‌های $0,1,\ldots,n$ را حل می‌کند، و برای حل هر زیرمسئله با اندازه‌ی $n$، حلقه‌ی for خطوط ۶-۷ تعداد $n$ بار تکرار می‌شود. بنابراین تعداد کل تکرارهای این حلقه‌ی for در تمام فراخوانی‌های بازگشتی MEMOIZED-CUT-ROD یک سری حسابی تشکیل می‌دهند با مجموع $\theta(n^2)$ تکرار، درست مانند حلقه‌ی for در رویه‌ی BOTTOM-UP-CUT-ROD. (در واقع در این جا از نوعی تحلیل سرشکن استفاده کردیم. تحلیل سرشکن را به طور مفصل در بخش ۱۷-۱ خواهیم دید.)

## گراف‌های زیرمسئله

وقتی درمورد یک مسئله‌ی برنامه‌ریزی پویا فکر می‌کنیم، باید مجموعه‌ی زیرمسئله‌های درگیر و نحوه‌ی ارتباط آن‌ها با یکدیگر را درک کنیم.

*گراف زیرمسئله* (subproblem graph) دقیقاً حاوی همین اطلاعات است. شکل ۱۵-۴ گراف زیرمسئله را برای مسئله‌ی برش میله با $n = 4$ نشان می‌دهد. گراف زیرمسئله یک یال جهت‌دار از رأس مربوط به زیرمسئله‌ی $x$ به رأس مربوط به زیرمسئله‌ی $y$ دارد اگر تعیین یک جواب بهینه برای $x$ مستقیماً نیاز به تعیین یک جواب بهینه برای مسئله‌ی $y$ داشته باشد. به عنوان مثال گراف زیرمسئله حاوی یک یال از $x$ به $y$ است اگر یک رویه‌ی بازگشتی بالا به پایین برای حل $x$، مستقیماً خود را برای حل $y$ فراخوانی کند. می‌توان گراف زیرمسئله را یک نسخه‌ی «کاهش داده شده» از درخت بازگشت برای متد بازگشتی بالا به پایین دانست، که در آن تمام گره‌هایی را که نشان دهنده‌ی یک زیرمسئله هستند، در یک گره می‌آمیزیم، و جهت تمام یال‌ها را از پدر به فرزند می‌کشیم.

متد پایین به بالا برای برنامه‌ریزی پویا، رأس‌های گراف زیرمسئله را به ترتیبی در نظر می‌گیرد که در آن زیرمسئله‌ی $y$ مجاور با زیرمسئله‌ی داده شده‌ی $x$ را قبل از $x$ حل می‌کنیم. (از بخش ب-۴ به خاطر بیاورید که رابطه‌ی مجاورت لزوماً متقارن نیست.) با استفاده از اصطلاحات فصل ۲۲، در یک الگوریتم برنامه‌ریزی پویای پایین به بالا، رأس‌های گراف زیرمسئله را به ترتیب «برعکس مرتب‌سازی توپولوژیکی»، یا «ترتیب توپولوژیکی ترانهاده» (بخش ۲۲-۴ را ببینید) در نظر می‌گیریم. به عبارت دیگر، هیچ زیرمسئله‌ای در نظر گرفته نمی‌شود تا این که تمام زیرمسئله‌هایی که آن زیرمسئله به آن‌ها بستگی دارد، حل شوند. به طور مشابه، با استفاده از مفاهیم همان فصل، می‌توانیم متد بالا به پایین (با به خاطر سپاری) را برای برنامه‌ریزی پویا به صورت یک «جستجوی عمق-اول» در گراف زیرمسئله ببینیم (بخش ۲۲-۳ را ببینید).




شکل ۱۵-۷ گراف زیرمسئله برای مسئله‌ی برش میله با $n = 4$. برچسب رأس‌ها نشان‌دهنده‌ی اندازه‌ی زیرمسئله‌های متناظر است. یک یال جهت‌دار $(x, y)$ نشان می‌دهد که هنگام حل زیرمسئله‌ی $x$، به جواب زیرمسئله‌ی $y$ نیاز داریم. این گراف یک نسخه‌ی کاهش داده شده از درخت شکل ۱۵-۳ است، که در آن تمام گره‌ها با برچسب یکسان در یک رأس آمیخته شده‌اند، و تمام یال‌ها از پدر به فرزند جهت گرفته‌اند.

اندازه‌ی گراف زیرمسئله‌ی $G = (V, E)$ می‌تواند به ما کمک کند که زمان اجرای الگوریتم برنامه‌ریزی پویا را تعیین کنیم. از آن جایی که هر زیرمسئله را فقط یک بار حل می‌کنیم، زمان اجرا برابر است با مجموع زمان‌های مورد نیاز برای حل زیرمسئله‌ها. معمولاً، زمان مورد نیاز برای محاسبه‌ی جواب یک زیرمسئله متناسب است با درجه‌ی (تعداد یال‌های خروجی) رأس متناظر در گراف زیرمسئله، و تعداد زیرمسئله‌ها برابر است با تعداد رأس‌های گراف زیرمسئله. در این حالت معمول، زمان اجرای الگوریتم برنامه‌ریزی پویا متناسب است با تعداد رأس‌ها و یال‌ها.

## بازسازی یک جواب

جواب‌های برنامه‌ریزی پویایی که در این جا برای مسئله‌ی برش میله ارائه شد، مقدار یک جواب بهینه را بازمی‌گردانند، ولی خود جواب (لیست اندازه‌ی قطعه‌ها) را نه. می‌توانیم روش برنامه‌ریزی پویا را طوری گسترش دهیم که علاوه بر *مقدار* بهینه‌ی محاسبه شده برای هر زیرمسئله، یک *انتخاب* را هم بازگردانند که به مقدار بهینه منتهی می‌شود. با این اطلاعات، به سادگی می‌توانیم یک جواب بهینه را در خروجی چاپ کنیم.

در زیر یک نسخه‌ی گسترش یافته از BOTTOM-UP-CUT-ROD را می‌بینیم که برای هر میله با اندازه‌ی $j$، علاوه بر درآمد $r_j$، یک $s_j$ را هم محاسبه می‌کند، که اندازه‌ی بهینه‌ی اولین قطعه‌ی بریده شده است:

```
EXTENDED-BOTTOM-UP-CUT-ROD (p, n)
1  let r[0..n] and s[0..n] be new arrays
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
```

```
5       for i = 1 to j
6           if q < p[i] + r[j-i]
7               q = p[i] + r[j-i]
8               s[j] = i
9       r[j] = q
10  return r and s
```

این رویه مشابه BOTTOM-UP-CUT-ROD است، با این تفاوت که در خط ۱، یک آرایه‌ی $s$ هم می‌سازد، و در خط ۸ هنگام حل زیرمسئله‌ی با اندازه‌ی $j$، مقدار $s[j]$ را به هنگام‌سازی می‌کند تا اندازه‌ی بهینه‌ی $i$ مربوط به اولین قطعه‌های بریده شده را نگه دارد.

رویه‌ی زیر یک جدول قیمت $p$ و یک اندازه‌ی میله‌ی $n$ را دریافت می‌کند، EXTENDED-BOTTOM-UP-CUT-ROD را برای محاسبه‌ی $s[1\,..\,n]$ مربوط به اندازه‌ی بهینه‌ی اولین قطعه‌های بریده شده فراخوانی می‌کند، و سپس لیست کامل اندازه‌ی قطعه‌ها را در یک برش بهینه برای میله‌ی به طول $n$ چاپ می‌کند:

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

در مثال برش میله، فراخوانی EXTENDED-BOTTOM-UP-CUT-ROD $(p,۱۰)$ آرایه‌های زیر را بازخواهد گرداند:

| $i$ | ۰ | ۱ | ۲ | ۳ | ۴ | ۵ | ۶ | ۷ | ۸ | ۹ | ۱۰ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $r[i]$ | ۰ | ۱ | ۵ | ۸ | ۱۰ | ۱۳ | ۱۷ | ۱۸ | ۲۲ | ۲۵ | ۳۰ |
| $s[i]$ | ۰ | ۱ | ۲ | ۳ | ۲ | ۲ | ۶ | ۱ | ۲ | ۳ | ۱۰ |

یک فراخوانی PRINT-CUT-ROD-SOLUTION $(p,۱۰)$ فقط عدد ۱۰ را چاپ خواهد کرد، ولی یک فراخوانی با $n = ۷$، برش‌های ۱ و ۶ را چاپ می‌کند، متناظر با اولین تجزیه‌ی بهینه برای $r_۷$ که قبلاً دیدیم.

## تمرین‌ها

۱۵-۱-۱ نشان دهید که تساوی (۱۵-۴) از تساوی (۱۵-۳) و شرط اولیه‌ی $T(۰) = ۱$ نتیجه می‌شود.

۱۵-۱-۲ با استفاده از مثال نقض، نشان دهید که استراتژی «حریصانه»ی زیر همیشه یک روش بهینه برای برش میله‌ها ارائه نمی‌کند. *چگالی* یک میله با طول $i$ را به صورت $p_i / i$ تعریف کنید، یعنی ارزش آن بر هر اینچ. استراتژی حریصانه برای یک میله با طول $n$، اولین قطعه را با طول $i$ می‌برد، که در آن $۱ \le i \le n$ بالاترین چگالی را دارد. سپس همین استراتژی حریصانه را برای قطعه‌ی باقی مانده با طول $n - i$ به کار می‌برد.

۱۵-۱-۳ نسخه‌ای اصلاح شده از مسئله‌ی برش میله را در نظر بگیرید که در آن، علاوه بر قیمت $p_i$ برای هر میله، هر برش هزینه‌ی $c$ را به ما تحمیل می‌کند. اکنون درآمد حاصل از یک جواب عبارت‌است از مجموع ارزش قطعه‌ها منهای هزینه‌ی انجام برش‌ها. یک الگوریتم برنامه‌ریزی پویا برای حل این مسئله‌ی اصلاح شده ارائه کنید.

۱۵-۱-۴ رویه‌ی MEMOIZED-CUT-ROD را طوری اصلاح کنید که علاوه بر مقدار جواب بهینه، خود جواب را هم بازگرداند.

۱۵-۱-۵ اعداد فیبوناچی طبق تساوی (۳-۲۲) تعریف می‌شوند. یک الگوریتم برنامه‌ریزی پویا با زمان $O(n)$ برای محاسبه‌ی عدد فیبوناچی $n$ام ارائه کرده و گراف زیرمسئله را بکشید. چند رأس و یال در گراف وجود دارد؟

## ۱۵-۲ ضرب زنجیره‌ی ماتریس‌ها

مثال بعدی ما از برنامه‌ریزی پویا، الگوریتمی است که مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها را حل می‌کند. یک دنباله (زنجیره) $\langle A_1, A_2, ..., A_n \rangle$ از $n$ ماتریس داریم که می‌خواهیم آن‌ها را در یکدیگر ضرب کنیم، یعنی می‌خواهیم

$$A_1 A_2 ... A_n \qquad (۱۵-۵)$$

را محاسبه کنیم. می‌توانیم عبارت (۱۵-۱۰) را با استفاده از الگوریتم استاندارد ضرب ماتریس‌ها به عنوان یک زیرروال محاسبه کنیم، البته بعد از این که ابهام در ترتیب ضرب ماتریس‌ها برطرف شد. یک ضرب ماتریسی *پرانتزی شده‌ی کامل* است اگر یا یک ماتریس تنها باشد، یا ضرب دو ضرب ماتریسی پرانتزی شده‌ی کامل، که با پرانتزهای باز و بسته احاطه شده است. ضرب ماتریس‌ها شرکت‌پذیر است، و بنابراین تمام پرانتزگذاری‌ها به یک نتیجه‌ی یکسان ختم می‌شوند. به عنوان مثال، اگر دنباله‌ی ماتریس‌ها $\langle A_1, A_2, A_3, A_4 \rangle$ باشد، ضرب $A_1 A_2 A_3 A_4$ را می‌توان به پنج شکل زیر پرانتزگذاری کرد:

$$(A_1 (A_2 (A_3 A_4))),$$
$$(A_1 ((A_2 A_3) A_4)),$$
$$((A_1 A_2) (A_3 A_4)),$$
$$((A_1 (A_2 A_3)) A_4),$$
$$(((A_1 A_2) A_3) A_4).$$

روش پرانتز گذاری دنباله‌ی ماتریس‌ها می‌تواند به شدت بر روی هزینه‌ی ضرب تأثیر بگذارد. ابتدا هزینه‌ی ضرب دو ماتریس را در نظر بگیرید. الگوریتم استاندارد در زیر به صورت شبه‌کد داده شده است، که گسترش یافته‌ی رویه‌ی SQUARE-MATRIX-MULTIPLY از بخش ۴-۲ است. خصیصه‌های *rows* و *columns*، تعداد ردیف‌ها و ستون‌ها در یک ماتریس هستند.

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns array
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

فقط در صورتی می‌توانیم دو ماتریس $A$ و $B$ را در هم ضرب کنیم که در هم قابل ضرب باشند: تعداد ستون‌های $A$ باید برابر با تعداد ردیف‌های $B$ 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 محاسبه‌ی $C$ برابر است با تعداد ضرب‌های اسکالر در خط ۷، که برابر است با $pqr$. در ادامه‌ی این بخش، هزینه‌ها را نسبت به تعداد ضرب‌های اسکالر انجام شده توصیف می‌کنیم.

برای تعیین هزینه‌های متفاوت حاصل از پرانتز گذاری‌های مختلف یک ضرب ماتریسی، مسئله‌ی زنجیره‌ی ماتریسی $\langle A_1 A_2 A_3 \rangle$ را در نظر بگیرید. فرض کنید که ابعاد ماتریس‌ها به ترتیب عبارت است از ۱۰۰×۱۰، ۵×۱۰۰، و ۵۰×۵. اگر طبق پرانتز گذاری $((A_1 A_2)A_3)$ ماتریس‌ها را ضرب کنیم، ابتدا برای ضرب $A_1 A_2$، تعداد ۵۰۰۰ = ۵×۱۰۰×۱۰ ضرب اسکالر انجام می‌دهیم، و سپس ۲۵۰۰ = ۵۰×۵×۱۰ ضرب اسکالر دیگر برای ضرب نتیجه‌ی آن در ماتریس $A_3$، که مجموع آن‌ها برابر است با ۷۵۰۰. در عوض اگر طبق پرانتز گذاری $(A_1(A_2 A_3))$ ضرب را انجام دهیم، ۲۵,۰۰۰ = ۵۰×۵×۱۰۰ ضرب اسکالر برای محاسبه‌ی $A_2 A_3$، که ابعاد آن ۵۰×۱۰۰ است انجام می‌دهیم، به علاوه‌ی ۵۰,۰۰۰ = ۵۰×۱۰۰×۱۰ ضرب اسکالر برای محاسبه‌ی ضرب $A_1$ در این ماتریس، که مجموع آن برابر است با ۷۵,۰۰۰ ضرب اسکالر. بنابراین محاسبه‌ی ضرب طبق پرانتز گذاری اول، ۱۰ برابر سریع‌تر است.

*مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها* را می‌توان به صورت زیر تعریف کرد: با داشتن زنجیره‌ی $n$ ماتریسی $\langle A_1, A_2, ..., A_n \rangle$، که در آن $i = 1,2,...,n$، و ماتریس $A_i$ دارای ابعاد $p_{i-1} p_i$ است، ضرب $A_1 A_2 ... A_n$ را طوری پرانتز گذاری کنید که تعداد ضرب‌های اسکالر کمینه شود.

توجه کنید که در مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها، در واقع ما ماتریس‌ها را در هم ضرب نمی‌کنیم. هدف ما فقط این است که یک ترتیب برای ضرب ماتریس‌ها تعیین کنیم که کم‌ترین هزینه را داشته باشد. معمولاً زمان صرفه‌جویی شده هنگام ضرب ماتریس‌ها (مثلاً انجام ۷۵۰۰ ضرب به جای ۷۵۰۰۰) بیشتر از زمان سرمایه‌گذاری شده برای تعیین این ترتیب بهینه است.

## شمارش تعداد پرانتز گذاری‌های ممکن

قبل از حل مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها به کمک برنامه‌ریزی پویا، اجازه دهید خود را قانع کنیم که چک کردن تمام پرانتزگذاری‌های ممکن نمی‌تواند الگوریتم مناسبی باشد. تعداد پرانتزگذاری‌های

مختلف ممکن برای یک دنباله از $n$ ماتریس را با $P(n)$ نشان می‌دهیم. وقتی $n = ۱$، فقط یک ماتریس داریم و بنابراین فقط یک روش برای پرانتز گذاری کامل ضرب ماتریس‌ها وجود دارد. وقتی $n \geq ۲$، یک ضرب ماتریسی به طور کامل پرانتز گذاری شده، عبارت است از ضرب دو ضرب ماتریسی به طور کامل پرانتز گذاری شده، و این تقسیم میان این دو زیردنباله از ضرب ماتریس‌ها بین ماتریس $k$ام و $(k+۱)$ام اتفاق می‌افتد، که $k$ می‌تواند هر یک از اعداد $۱, ۲, \ldots, n-۱$ باشد. بنابراین رابطه‌ی بازگشتی زیر را خواهیم داشت:

$$P(n) = \begin{cases} ۱ & \text{اگر } n = ۱ \\ \sum_{k=۱}^{n-۱} P(k)P(n-k) & \text{اگر } n \geq ۲ \end{cases} \qquad (۶-۱۵)$$

مسئله‌ی ۱۲-۴ از شما خواست که نشان دهید جواب یک رابطه‌ی بازگشتی مشابه، دنباله‌ی *اعداد کاتالان* است، که به صورت $\Omega(۴^n / n^{۳/۲})$ رشد می‌کند. یک تمرین ساده‌تر (تمرین ۱۵-۲-۳ را ببینید) از شما می‌خواهد نشان دهید که جواب رابطه‌ی بازگشتی (۱۵-۶) برابر $\Omega(۲^n)$ است. بنابراین تعداد جواب‌ها نسبت به $n$ نمایی است، و متد بی‌خردانه‌ی جستجوی تمام راه‌ها یک استراتژی ضعیف برای پرانتز گذاری بهینه‌ی زنجیره‌ی ماتریس‌ها است.

## به کار گیری برنامه‌ریزی پویا

از متد برنامه‌ریزی پویا استفاده خواهیم کرد تا تعیین کنیم چگونه باید یک زنجیره از ماتریس‌ها را به شکل بهینه پرانتزگذاری کرد. در انجام این کار، از دنباله‌ی چهار مرحله‌ای که در ابتدای این فصل معرفی کردیم، استفاده خواهیم کرد:

۱. تعیین ساختار یک جواب بهینه.
۲. تعریف بازگشتی مقدار یک جواب بهینه.
۳. محاسبه‌ی مقدار یک جواب بهینه.
٤. ساخت یک جواب بهینه از اطلاعات محاسبه شده.

در این جا به ترتیب این مراحل را انجام خواهیم داد، و با جزئیات توضیح خواهیم داد که هر مرحله را چگونه به کار می‌بریم.

## مرحله‌ی ۱: ساختار یک پرانتز گذاری بهینه

اولین قدم ما در رویکرد برنامه‌ریزی پویا، یافتن زیرساختار بهینه و استفاده از آن برای ساختن جواب بهینه به مسئله از جواب‌های بهینه به زیرمسئله‌ها است. برای مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها می‌توانیم این مرحله را به صورت زیر انجام دهیم. برای راحتی، اجازه دهید نماد $A_{i..k}$ را، که در آن $i \leq j$، برای ماتریسی در نظر بگیریم که حاصل ضرب ماتریس‌های $A_i A_{i+۱} \ldots A_j$ است. مشاهده کنید که اگر مسئله بدیهی نباشد، یعنی $i < j$، آن گاه هر پرانتز گذاری برای ضرب $A_i A_{i+۱} \ldots A_j$ باید دنباله را برای یک عدد $k$ که $i \leq k < j$، بین $A_k$ و $A_{k+۱}$ جدا کنیم. یعنی برای یک مقدار $k$، ابتدا

ماتریس‌های $A_{i..k}$ و $A_{k+1..j}$ را محاسبه، و سپس آن‌ها را در یکدیگر ضرب می‌کنیم تا حاصل ضرب نهایی $A_{i..j}$ را به دست آوریم. بنابراین، هزینه‌ی این پرانتز گذاری برابر است با هزینه‌ی محاسبه‌ی $A_{i..k}$، به علاوه‌ی هزینه‌ی محاسبه‌ی $A_{k+1..j}$، به علاوه‌ی هزینه‌ی ضرب این دو ماتریس در یکدیگر.

زیرساختار بهینه‌ی این مسئله به صورت زیر است. فرض کنید که یک پرانتزگذاری بهینه‌ی $A_i A_{i+1} \ldots A_j$، ضرب را بین ماتریس‌های $A_k$ و $A_{k+1}$ جدا می‌کند. در این صورت پرانتزگذاری زیرزنجیر «پیشین» در این پرانتزگذاری بهینه‌ی $A_i A_{i+1} \ldots A_k$ باید خود یک پرانتزگذاری بهینه باشد. چرا؟ اگر یک روش کم هزینه‌تر برای پرانتزگذاری $A_i A_{i+1} \ldots A_k$ وجود داشته باشد، جایگذاری این پرانتزگذاری در پرانتزگذاری بهینه‌ی $A_i A_{i+1} \ldots A_j$، یک پرانتزگذاری دیگر را نتیجه خواهد داد که هزینه‌ی آن از هزینه‌ی بهینه کم‌تر است: تناقض. استدلال مشابهی برای پرانتزگذاری زیرزنجیره‌ی $A_{k+1} A_{k+2} \ldots A_j$ در پرانتزگذاری بهینه‌ی $A_i A_{i+1} \ldots A_j$ برقرار است: این پرانتزگذاری باید بهینه باشد.

اکنون از زیرساختار بهینه‌ی خود استفاده می‌کنیم تا نشان دهیم که می‌توانیم از جواب بهینه‌ی زیرمسئله‌ها، یک جواب بهینه برای مسئله‌ی اصلی تولید کنیم. قبلاً دیدیم که هر جوابی برای یک نمونه‌ی غیر بدیهی از مسئله‌ی ضرب ماتریس‌ها، نیاز به قسمت کردن زنجیره‌ی ماتریس‌ها دارد، و هر جواب بهینه در خود حاوی جواب‌های بهینه به زیرمسئله‌ها است. بنابراین، برای یافتن یک جواب بهینه برای مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها می‌توانیم مسئله را به دو زیرمسئله تقسیم کنیم (پرانتزگذاری بهینه‌ی $A_i A_{i+1} \ldots A_k$ و $A_{k+1} A_{k+2} \ldots A_j$)، یک جواب بهینه برای این زیرمسئله‌ها بیابیم، و سپس این جواب‌ها را با یکدیگر ترکیب کنیم. باید مطمئن شویم که وقتی به دنبال مکان مناسب برای قسمت کردن زنجیره‌ی ماتریس‌ها می‌گردیم، تمام مکان‌های ممکن را امتحان می‌کنیم، تا بتوانیم اطمینان حاصل کنیم که مکان بهینه در میان آن‌ها است.

## مرحله‌ی ۲: یک جواب بازگشتی

سپس، هزینه‌ی یک جواب بهینه را به صورت بازگشتی نسبت به جواب‌های بهینه‌ی زیرمسئله‌ها تعیین می‌کنیم. برای مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها، مسئله‌ی تعیین هزینه‌ی کمینه‌ی پرانتزگذاری $A_i A_{i+1} \ldots A_j$ را برای $1 \le i \le j \le n$ به عنوان زیرمسئله‌ی خود انتخاب می‌کنیم. فرض کنید $m[i,j]$ کمینه‌ی تعداد ضرب‌های اسکالر مورد نیاز برای محاسبه‌ی ماتریس $A_{i..j}$ باشد؛ برای مسئله‌ی کلی، هزینه‌ی ارزان‌ترین روش برای محاسبه‌ی $A_{1..n}$، $m[1,n]$ خواهد بود.

$m[i,j]$ را می‌توانیم به صورت بازگشتی به شکل زیر تعریف کنیم. اگر $i = j$، مسئله بدیهی است؛ زنجیره فقط شامل یک ماتریس $A_{i..i} = A_i$ می‌شود، و بنابراین برای محاسبه‌ی حاصل ضرب به هیچ ضرب اسکالری نیاز نداریم. بنابراین، $m[i,j] = 0$ برای $i = 1,2,\ldots,n$. برای محاسبه‌ی $m[i,j]$ وقتی $i < j$، از ساختار یک جواب بهینه از مرحله‌ی ۱ بهره می‌گیریم. اجازه دهید فرض کنیم پرانتز گذاری بهینه، دنباله‌ی $A_i A_{i+1} \ldots A_j$ را بین $A_k$ و $A_{k+1}$ قسمت می‌کند، که در آن $i \le k < j$. در این صورت $m[i,j]$ برابر خواهد بود با هزینه‌ی کمینه‌ی محاسبه‌ی حاصل ضرب‌های جزئی $A_{i..k}$ و $A_{k+1..j}$، به علاوه‌ی هزینه‌ی ضرب این دو ماتریس در یکدیگر. با توجه به این که ابعاد هر ماتریس $A_i$ برابر

$p_{i-1} \times p_i$ است، می‌بینیم که محاسبه‌ی ضرب ماتریسی $A_{i..k}\, A_{k+1..j}$ به $p_{i-1} p_k p_j$ ضرب اسکالر نیاز دارد. بنابراین به دست می‌آوریم

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$$

در این رابطه‌ی بازگشتی فرض می‌شود که ما مقدار $k$ را می‌دانیم، که در واقع این فرض درست نیست. با این حال فقط $j-i$ مقدار ممکن برای $k$ وجود دارد، یعنی $k = i, i+1, ..., j-1$. از آن جایی که پرانتز گذاری بهینه باید از یکی از این مقادیر برای $k$ استفاده کند، فقط باید تمام آن‌ها را بررسی کنیم تا بتوانیم بهترین را بیابیم. بنابراین تعریف بازگشتی ما از کم‌ترین هزینه‌ی پرانتز گذاری دنباله‌ی $A_i A_{i+1} ... A_j$ تبدیل می‌شود به

$$m[i,j] = \begin{cases} 0 & \text{اگر } i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{اگر } i < j \end{cases} \qquad (۷-۱۵)$$

مقادیر $m[i,j]$ هزینه‌ی جواب بهینه به زیرمسئله‌ها را می‌دهند، ولی تمام اطلاعات مورد نیاز برای ساختن جواب بهینه را ارائه نمی‌کنند. برای این که بتوانیم این کار را بکنیم، $s[i,j]$ را به صورت مقدار $k$ که دنباله‌ی $A_i A_{i+1} ... A_j$ برای جواب بهینه از آن جا قسمت می‌شود، تعریف می‌کنیم. در واقع، $s[i,j]$ برابر یک مقدار $k$ خواهد بود به طوری که $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$.

## مرحله‌ی ۳: محاسبه‌ی هزینه‌های بهینه

در این جا، به سادگی می‌توانیم یک الگوریتم بازگشتی بر مبنای رابطه‌ی بازگشتی (۷-۱۵) برای محاسبه‌ی هزینه‌ی کمینه‌ی $m[1,n]$ برای ضرب $A_1 A_2 ... A_n$ بنویسیم. با این حال، همان طور که برای مسئله‌ی برش میله دیدیم، و همان طور که در بخش ۳-۱۵ خواهیم دید، این الگوریتم بازگشتی به زمان نمایی نیاز دارد، که از متد بی خردانه‌ی بررسی تمام پرانتز گذاری‌های ممکن بهتر نیست.

مشاهده‌ی مهمی که در این جا می‌توانیم بکنیم این است که تعداد زیرمسئله‌های موجود نسبتا کم است: یک مسئله برای هر انتخاب $i$ و $j$ که $1 \le i \le j \le n$، و یا در مجموع $\binom{n}{2} + n = \theta(n^2)$. یک الگوریتم بازگشتی ممکن است در شاخه‌های درخت بازگشتی خود چندین بار به هر زیرمسئله برخورد کند. این خصوصیت زیرمسئله‌های مشترک، دومین نشانه‌ی قابل استفاده بودن برنامه‌ریزی پویا است (که اولین نشانه، داشتن زیرساختار بهینه است).

به جای محاسبه‌ی جواب رابطه‌ی (۷-۱۵) به صورت بازگشتی، هزینه‌ی بهینه را با استفاده از یک روش جدولی از پایین به بالا محاسبه می‌کنیم. (رویکرد بالا به پایین متناظر با استفاده از به خاطرسپاری را در بخش ۳-۱۵ ارائه خواهیم کرد.)

متد جدولی پایین به بالا را در رویه‌ی MATRIX-CHAIN-ORDER پیاده‌سازی خواهیم کرد، که در زیر آن را می‌بینیم. این رویه فرض می‌کند که ابعاد ماتریس $A_i$، $p_{i-1} \times p_i$ است، برای $i = 1, 2, ..., n$. ورودی، یک دنباله‌ی $p = \langle p_0, p_1, ..., p_n \rangle$ است، که $p.length = n+1$. شبه‌کد از یک جدول کمکی

$m[1..n,1..n]$ برای ذخیره‌ی هزینه‌های $m[i,j]$ استفاده می‌کند، و همچنین یک جدول $s[1..n,1..n]$ که مشخص می‌کند کدام اندیس $k$ در محاسبه‌ی $m[i,j]$ ما را به هزنیه‌ی بهینه می‌رساند. از جدول $s$ برای ساختن یک جواب بهینه استفاده خواهیم کرد.

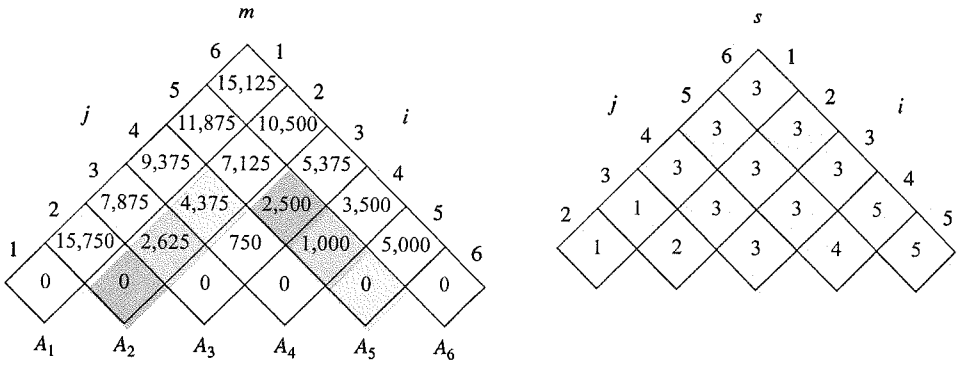
برای پیاده‌سازی رویکرد پایین به بالا، باید تعیین کنیم که از کدام یک از ورودی‌های جدول در محاسبه‌ی $m[i,j]$ استفاده می‌شود. تساوی (۱۵-۷) نشان می‌دهد که هزینه‌ی $m[i,j]$ مربوط به محاسبه‌ی ضرب زنجیره‌ی $j-i+1$ ماتریس فقط به هزینه‌ی ضرب زنجیره‌های ماتریس کم‌تر از $j-i+1$ بستگی دارد. به عبارت دیگر، برای $k=i,i+1,...,j-1$، ماتریس $A_{i..k}$ ضرب $k-i+1<j-i+1$ ماتریس، و $A_{k+1..j}$ ضرب $j-k<i-j+1$ ماتریس است. بنابراین الگوریتم باید جدول $m$ را به ترتیبی پر کند که متناسب باشد با حل مسئله‌ی پرانتز گذاری بر روی زنجیره‌ای از ماتریس‌ها با ترتیب طول صعودی. برای زیرمسئله‌ی پرانتزگذاری بهینه‌ی زنجیره‌ی $A_i A_{i+1} ... A_j$، اندازه‌ی زیرمسئله را برابر با طول زنجیره، یعنی $j-i+1$ در نظر می‌گیریم.

```
MATRIX-CHAIN-ORDER(p)
1   n = p.length – 1
2   let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
2   for i = 1 to n
3       m[i, i] = 0
4   for l = 2 to n                  // l is the chain length.
5       for i = 1 to n - l + 1
6           j = i + l - 1
7               m[i, j] = ∞
8               for k = i to j - 1
9                   q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
10                  if q < m[i, j]
11                      m[i, j] = q
12                      s[i, j] = k
13  return m and s
```

این الگوریتم ابتدا در خطوط ۳-۴، $m[i,i]=0$ را برای $i=1,2,...,n$ مقداردهی می‌کند (هزینه‌ی کمینه برای زنجیره‌های به طول ۱). سپس در اولین تکرار حلقه‌ی for خطوط ۵-۱۳، از رابطه‌ی بازگشتی (۱۵-۷) برای محاسبه‌ی $m[i,i-1]$ برای $i=1,2,...,n-1$ استفاده می‌کند (هزینه‌ی کمینه برای زنجیره‌های به طول $l=2$). در دومین تکرار حلقه، مقدار $m[i,i+2]$ برای $i=1,2,...,n-2$ محاسبه می‌شود (هزینه‌ی کمینه برای زنجیره‌های به طول $l=3$)، و به همین ترتیب تا آخر. در هر مرحله، هزینه‌ی $m[i,j]$ محاسبه شده در خطوط ۱۰-۱۳ فقط به خانه‌های $m[i,k]$ و $m[k+1,j]$ جدول بستگی دارد، که قبلاً محاسبه شده‌اند.

شکل ۱۵-۵ این رویه را بر روی زنجیره‌ای از $n=6$ ماتریس نشان می‌دهد. از آن جایی که $m[i,j]$ را فقط برای $i \le j$ تعریف کرده‌ایم، فقط قسمت‌هایی از جدول که بالای قطر اصلی هستند

مورد استفاده قرار می‌گیرند. در شکل، جدول به طوری دوران داده شده است که قطر اصلی آن به صورت افقی قرار گیرد. زنجیره‌ی ماتریس‌ها در پایین لیست شده است. با استفاده از این آرایش، هزینه‌ی کمینه برای ضرب زنجیره‌ی ماتریس‌های $A_i A_{i+1} \ldots A_j$ را می‌توان در تقاطع خطوطی که از امتداد $A_i$ به سمت شمال شرقی و امتداد $A_j$ به سمت شمال غربی به دست می‌آید، مشاهده کرد. هر ردیف افقی در جدول حاوی ورودی‌هایی برای ضرب زنجیره‌هایی با طول یکسان است. MATRIX-CHAIN-ORDER ردیف‌ها را از پایین به بالا، و از چپ به راست محاسبه می‌کند. هر ورودی $m[i,j]$ با استفاده از ضرب $p_{i-1} p_k p_j$ برای $k = i, i+1, \ldots, j-1$، و تمام ورودی‌های واقع در جنوب غرب و جنوب شرق آن خانه محاسبه می‌شود.




**شکل ۱۵-۵** جدول‌های $m$ و $s$ که توسط MATRIX-CHAIN-ORDER، برای $n = ۶$ و ماتریس‌هایی با ابعاد زیر محاسبه می‌شوند:

| ماتریس | ابعاد |
|---|---|
| $A_۱$ | ۳۰×۳۵ |
| $A_۲$ | ۳۵×۱۵ |
| $A_۳$ | ۱۵×۵ |
| $A_۴$ | ۵×۱۰ |
| $A_۵$ | ۱۰×۲۰ |
| $A_۶$ | ۲۰×۲۵ |

جدول‌ها طوری دوران یافته‌اند که قطر اصلی به صورت افقی قرار گیرد. در جدول $m$، فقط از قطر اصلی و مثلث بالایی، و در جدول $s$ فقط از مثلث بالایی استفاده می‌شود. کمینه‌ی تعداد ضرب‌های اسکالر مورد نیاز برای محاسبه‌ی ضرب این ۶ ماتریس برابر است با $m[۱,۶] = ۱۵,۱۲۵$. در میان خانه‌های تیره‌تر جدول، از جفت‌هایی که سایه‌ی آن‌ها هم رنگ است به همراه هم در خط ۹ برای محاسبه‌ی

$$m[۲,۵] = \min \begin{cases} m[۲,۲] + m[۳,۵] + p_۱ p_۲ p_۵ = ۰ + ۲۵۰۰ + ۳۵ \times ۱۵ \times ۲۰ = ۱۳۰۰ \\ m[۲,۳] + m[۴,۵] + p_۱ p_۳ p_۵ = ۲۶۲۵ + ۱۰۰۰ + ۳۵ \times ۵ \times ۲۰ = ۷۱۲۵ \\ m[۲,۴] + m[۵,۵] + p_۱ p_۴ p_۵ = ۴۳۷۵ + ۰ + ۳۵ \times ۱ \times ۲۰ = ۱۱۳۷۵ \end{cases}$$

استفاده شده است.

یک مشاهده‌ی ساده بر روی حلقه‌ی داخلی در ساختار MATRIX-CHAIN-ORDER زمان اجرای $O(n^3)$ را برای الگوریتم نشان می‌دهد. عمق حلقه‌های تودرتو سه است، و هر اندیس حلقه ($l$، $i$، و $k$) حداکثر $n-1$ مقدار به خود می‌گیرد. تمرین ۱۵-۲-۵ از شما می‌خواهد نشان دهید که زمان اجرای این الگوریتم در واقع $\Omega(n^3)$ است. الگوریتم به فضایی از مرتبه‌ی $O(n^2)$ برای ذخیره‌ی جدول‌های $m$ و $s$ نیاز دارد. بنابراین، MATRIX-CHAIN-ORDER بسیار کاراتر از متد بررسی تمام پرانتز گذاری‌های ممکن با زمان نمایی است.

## مرحله‌ی ۴: ساختن یک جواب بهینه

با این که MATRIX-CHAIN-ORDER تعداد بهینه‌ی ضرب‌های اسکالر مورد نیاز برای محاسبه‌ی ضرب زنجیره‌ی ماتریس‌ها را محاسبه می‌کند، مستقیماً نشان نمی‌دهد که چطور باید این ماتریس‌ها را در هم ضرب کنیم. ولی ساختن یک جواب بهینه با استفاده از اطلاعات ذخیره شده در جدول $s[1..n,1..n]$ کار دشواری نیست. هر ورودی $s[i,j]$ مقدار $k$ را ذخیره می‌کند به طوری که پرانتز گذاری بهینه‌ی $A_iA_{i+1}...A_j$، زنجیره‌ی ضرب را بین $A_k$ و $A_{k+1}$ قسمت می‌کند. بنابراین می‌دانیم که برای ضرب بهینه‌ی $A_{1..n}$، باید ضرب $A_{1..s[1,n]}A_{s[1,n]..n}$ را انجام دهیم. این ضرب‌های ماتریسی را می‌توانیم به صورت بازگشتی محاسبه کنیم، چرا که $s[1,s[1,n]]$ آخرین ضرب ماتریسی را در محاسبه‌ی $A_{1..s[1,n]}$، و $s[s[1,n]+1,n]$ آخرین ضرب ماتریسی را در محاسبه‌ی $A_{s[1,n]+1..n}$ مشخص می‌کند. رویه‌ی بازگشتی زیر با دریافت جدول $s$ که توسط MATRIX-CHAIN-ORDER محاسبه شده است، و اندیس‌های $i$ و $j$، یک پرانتز گذاری بهینه برای ضرب ماتریس‌های $\langle A_i, A_{i+1},..., A_j \rangle$ در صفحه نمایش چاپ می‌کند. فراخوانی اولیه‌ی MATRIX-CHAIN-ORDER$(s,1,n)$، پرانتز گذاری بهینه برای $\langle A_1, A_2,..., A_n \rangle$ را چاپ می‌کند.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      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$(s,1,6)$ پرانتز گذاری

$((A_1 (A_2 A_3)) ((A_4 A_5)A_6))$

را چاپ خواهد کرد.

### تمرین‌ها

۱۵-۲-۱ یک پرانتز گذاری بهینه برای ضرب زنجیره‌ی ماتریس‌هایی بیابید که دنباله‌ی ابعاد آن‌ها به صورت $\langle 5,10,3,12,5,50,6 \rangle$ است.

۱۵-۲-۲ یک الگوریتم بازگشتی MATRIX-CHAIN-MULTIPLY $(A,s,i,j)$ ارائه کنید که با دریافت دنباله‌ی ماتریس‌های $\langle A_1,A_2,...,A_n \rangle$، جدول $s$ محاسبه شده توسط MATRIX-CHAIN-ORDER، و اندیس‌های $i$ و $j$، خود ضرب زنجیره‌ی ماتریسی را انجام می‌دهد. (فراخوانی اولیه MATRIX-CHAIN-MULTIPLY $(A,s,1,n)$ خواهد بود.)

۱۵-۲-۳ با استفاده از متد جانشین‌سازی، نشان دهید که جواب رابطه‌ی بازگشتی (۱۵-۶)، $\Omega(n^2)$ است.

۱۵-۲-۴ گراف زیرمسئله‌ی مربوط به ضرب ماتریس‌ها برای یک زنجیره‌ی ورودی با طول $n$ را توصیف کنید. این گراف چند رأس دارد؟ چند یال دارد، و چه یال‌هایی؟

۱۵-۲-۵ فرض کنید $R(i,j)$ تعداد دفعاتی باشد که به ورودی $m[i,j]$ در حین محاسبه‌ی بقیه‌ی ورودی‌های جدول، هنگام فراخوانی MATRIX-CHAIN-ORDER دسترسی انجام می‌گیرد. نشان دهید که تعداد کل دسترسی‌ها به کل جدول برابر است با

$$\sum_{i=1}^{n}\sum_{j=i}^{n} R(i,j) = \frac{n^3-n}{3}$$

(*راهنمایی:* ممکن است تساوی (الف-۳) را مفید بیابید.)

۱۵-۲-۶ نشان دهید که یک پرانتز گذاری کامل بر روی $n$ عنصر دقیقاً $n-1$ جفت پرانتز دارد.

## ۱۵-۳ عناصر برنامه‌ریزی پویا

با این که تا این جا دو مثال از برنامه‌ریزی پویا را دیدیم، ممکن است شما هنوز از خود بپرسید که این متد در چه مواقعی کاربرد دارد. از دید مهندسی چه موقع باید به دنبال یک جواب برنامه‌ریزی پویا برای یک مسئله بگردیم؟ در این بخش، دو عنصر اصلی را خواهیم دید که یک مسئله‌ی بهینه‌سازی باید داشته باشد تا بتوان از برنامه‌ریزی پویا برای حل آن استفاده کرد: زیرساختار بهینه و زیرمسئله‌های مشترک. همچنین در مورد به خاطر سپاری بیشتر بحث خواهیم کرد، و این که این روش چطور می‌تواند در بهره بردن از خصوصیت زیرمسئله‌های همپوشا در روش بازگشتی بالا به پایین به ما کمک کند.

### زیرساختار بهینه

اولین قدم برای حل یک مسئله‌ی بهینه‌سازی به وسیله‌ی برنامه‌ریزی پویا، تعیین ساختار یک جواب بهینه است. به یاد بیاورید که یک مسئله دارای *زیرساختار بهینه* است اگر یک جواب بهینه‌ی مسئله، درون خود حاوی جواب‌های بهینه برای زیرمسئله‌ها باشد. هر وقت یک مسئله دارای زیرساختار بهینه باشد، نشانه‌ی خوبی است که ممکن است برنامه‌ریزی پویا برای آن کاربرد داشته باشد. (همچنین بدین

معنی است که ممکن است استراتژی حریصانه کاربرد داشته باشد. فصل ۱۶ را ببینید.) در برنامه‌ریزی پویا، جواب بهینه به مسئله را از جواب‌های بهینه به زیرمسئله‌ها می‌سازیم. در نتیجه باید مطمئن شویم که محدوده‌ی زیرمسئله‌هایی که در نظر می‌گیریم شامل آن‌هایی است که در جواب بهینه‌ی کل مسئله از آن‌ها استفاده کرده‌ایم.

در هر دو مسئله‌ای که تا کنون در این فصل بررسی کردیم زیرساختار بهینه وجود داشت. در بخش ۱۵-۱ مشاهده کردیم که روش بهینه برای بریدن یک میله به طول $n$ (البته اگر اصلاً برشی نیاز باشد)، شامل برش بهینه‌ی دو قطعه‌ی حاصل از برش اول خواهد بود. در بخش ۱۵-۲ دیدیم که یک پرانتزگذاری بهینه برای $A_i A_{i+1} \ldots A_j$ که ضرب را میان $A_k$ و $A_{k+1}$ قسمت می‌کند، در خود شامل یک جواب بهینه برای زیرمسئله‌های پرانتزگذاری $A_i A_{i+1} \ldots A_k$ و $A_{k+1} A_{k+2} \ldots A_j$ است.
در زیر، الگوهای مشترکی برای یافتن زیرساختار بهینه آمده است:

۱. نشان می‌دهید که جواب یک مسئله شامل انجام یک انتخاب است، مانند انتخاب یک برش اولیه در یک میله و یا انتخاب اندیسی که در آن یک زنجیره‌ی ماتریس باید شکسته شود. این انتخاب یک یا چند زیرمسئله تولید می‌کند که باید حل شوند.
۲. فرض می‌کنید که برای یک مسئله‌ی خاص، انتخاب‌هایی که به یک جواب بهینه ختم می‌شوند را دارید. هنوز نباید نگران چگونگی انجام انتخاب باشید، فقط فرض می‌کنید که این انتخاب‌ها به شما داده شده است.
۳. با داشتن این انتخاب، تعیین می‌کنید که کدام زیرمسئله‌ها بعد از این به وجود خواهند آمد، و چگونه می‌توان به بهترین شکل دامنه‌ی حاصل از زیرمسئله‌ها را توصیف کرد.
٤. با استفاده از تکنیک «بریدن و چسباندن» (cut-and-paste)، نشان می‌دهید که جواب زیرمسئله‌هایی که برای جواب بهینه‌ی مسئله بدان‌ها نیاز است، خود باید بهینه باشند. این کار را با فرض این که جواب یکی از زیرمسئله‌ها بهینه نیست، و سپس به تناقض رسیدن این کار را انجام می‌دهید. به خصوص، با «بریدن» جواب غیر بهینه‌ی زیرمسئله و «چسباندن» جواب بهینه، نشان می‌دهید که می‌توان یک جواب بهتر از جواب بهینه به دست آورد، که با این فرض که جواب بهینه است تناقض دارد. اگر بیش از یک زیرمسئله وجود داشت، آن‌ها معمولاً آنقدر به یکدیگر شبیه هستند که بحث بریدن و چسباندن برای یکی از آن‌ها را می‌توان به راحتی برای بقیه به کار برد.

برای توصیف فضای زیرمسئله‌ها، یک قانون خوب این است که فضا را تا حد ممکن ساده نگه دارید، و سپس در مواقع لزوم آن را گسترش دهید. به عنوان مثال فضای زیرمسئله‌هایی که برای مسئله‌ی برش میله در نظر گرفتیم شامل مسائل برش یک میله به طول $i$ برای هر اندازه‌ی $i$ بود. این فضای زیرمسئله به خوبی کار کرد، و نیازی به انتخاب فضای بزرگ‌تری برای زیرمسئله‌ها نبود.

بالعکس، فرض کنید که می‌خواستیم فضای زیرمسئله‌های خود را برای مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها به ماتریس‌های $A_1 A_2 \ldots A_j$ محدود کنیم. مانند قبل، یک پرانتز گذاری بهینه باید این ضرب

را میان $A_k$ و $A_{k+1}$ برای یک $1 \le k \le j$ قسمت کند. مگر این که می‌توانستیم تضمین کنیم که $k$ همیشه برابر با $j-1$ است، متوجه می‌شویم که زیرمسئله‌هایی به شکل $A_1 A_2 \ldots A_k$ و $A_{k+1} A_{k+2} \ldots A_j$ داریم، و زیرمسئله‌ی دوم به شکل $A_1 A_2 \ldots A_j$ نیست. برای این مسئله لازم بود که به زیرمسئله‌های خود اجازه دهیم که در «هر دو انتها» تغییر کنند، یعنی، در زیرمسئله‌ی $A_i A_{i+1} \ldots A_j$ به هر دوی $i$ و $j$ اجازه دهیم که تغییر کنند.

زیرساختار بهینه در مسائل مختلف از دو جهت با هم تفاوت دارند:

۱. از چند زیرمسئله در جواب بهینه‌ی مسئله‌ی اصلی استفاده می‌شود، و
۲. در تعیین این که از کدام زیرمسئله(ها) باید در جواب بهینه استفاده کنیم، چند انتخاب مختلف داریم.

در مسئله‌ی برش میله، یک جواب بهینه برای برش یک میله با اندازه‌ی $n$ فقط از یک زیرمسئله (با اندازه‌ی $n-i$) استفاده می‌کند، ولی باید برای تعیین این که کدام $i$ به جواب بهینه ختم می‌شود، باید $n$ انتخاب مختلف را برای آن در نظر بگیریم. ضرب زنجیره‌ی ماتریس‌ها برای تکه زنجیر $A_i A_{i+1} \ldots A_j$ مثالی است با دو زیرمسئله و $j-i$ انتخاب. برای یک ماتریس $A_k$ که زنجیره را به دو قسمت تقسیم می‌کند، دو زیرمسئله داریم – پرانتزگذاری $A_i A_{i+1} \ldots A_k$ و پرانتزگذاری $A_{k+1} A_{k+2} \ldots A_j$ – و باید هر *دوی* آن‌ها را به صورت بهینه حل کنیم. وقتی جواب بهینه به مسئله‌ها را یافتیم، از میان $j-i$ کاندیدا، یک اندیس برای $k$ انتخاب می‌کنیم.

به صورت غیر رسمی، زمان اجرای یک الگوریتم برنامه‌ریزی پویا به ضرب دو فاکتور بستگی دارد: تعداد کل زیرمسئله‌ها و تعداد انتخاب‌هایی که برای هر زیرمسئله آن‌ها را بررسی کنیم. در برش میله، در کل $\theta(n)$ زیرمسئله داشتیم، و حداکثر $n$ انتخاب برای هر کدام، که زمان اجرای $O(n^2)$ را نتیجه می‌دهد. برای ضرب زنجیره‌ی ماتریس‌ها، در کل $\theta(n^2)$ زیرمسئله وجود داشت، و در هر کدام حداکثر $n-1$ انتخاب داشتیم، که حاصل، زمان اجرای $O(n^3)$ بود (در واقع $\theta(n^3)$ طبق تمرین ۱۵-۲-۵).

معمولاً گراف زیرمسئله روشی دیگر برای همین تحلیل به دست می‌دهد. هر رأس متناظر است با یک زیرمسئله، و انتخاب‌های موجود برای یک زیرمسئله متناظرند با یال‌های مجاور رأس مربوطه. به خاطر بیاورید که در برش میله، گراف زیرمسئله $n$ رأس داشت، و حداکثر $n$ یال برای هر رأس، که زمان اجرای $O(n^2)$ را نتیجه می‌دهد. برای ضرب زنجیره‌ی ماتریس‌ها، اگر گراف زیرمسئله‌ها را می‌کشیدیم، $\theta(n^2)$ رأس داشت، و درجه‌ی هر رأس حداکثر $n-1$ بود، که مجموع $\theta(n^3)$ رأس و یال را به دست می‌دهد.

برنامه‌ریزی پویا معمولاً از زیرساختار بهینه به صورت پایین به بالا بهره می‌گیرد. یعنی ابتدا جواب‌های بهینه‌ی زیرمسئله‌ها را می‌یابیم، و سپس با استفاده از جواب آن‌ها یک جواب بهینه برای کل مسئله پیدا می‌کنیم. یافتن یک جواب بهینه برای مسئله مستلزم انتخاب زیرمسئله‌ای است که می‌خواهیم از آن در جواب بهینه استفاده کنیم. معمولاً هزینه‌ی جواب مسئله برابر است با هزینه‌ی

زیرمسئله‌ها به علاوه‌ی هزینه‌ای که مستقیماً از انتخاب حاصل می‌شود. مثلاً در برش میله، ابتدا زیرمسئله‌های برش بهینه‌ی میله‌هایی با طول $i$ برای $i = ۰,۱,...,n-۱$ را حل کردیم، و سپس با استفاده از تساوی (۱۵-۲)، تعیین کردیم که کدام یک از این زیرمسئله‌ها به یک جواب بهینه برای میله‌ی با طول $n$ ختم می‌شوند. درآمد مربوط به همان انتخاب، عبارت $p_i$ در تساوی (۱۵-۲) است. در ضرب زنجیره‌ی ماتریس‌ها، ابتدا پرانتز گذاری بهینه‌ی تکه زنجیر $A_i A_{i+۱}...A_j$ را تعیین کردیم، و سپس ماتریس $A_k$ را انتخاب کردیم که باید در آن محل زنجیر را قسمت کنیم. در این جا هزینه‌ی مربوط به انتخاب برابر است با $p_{i-۱} p_k p_j$.

در فصل ۱۶ «الگوریتم‌های حریصانه» را بررسی خواهیم کرد، که شباهت‌های بسیاری به برنامه‌ریزی پویا دارند. به خصوص، مسئله‌هایی که الگوریتم‌های حریصانه برای آن‌ها قابل کاربرد است، زیرساختار بهینه دارند. یک تفاوت اصلی میان الگوریتم‌های حریصانه و برنامه‌ریزی پویا این است که در الگوریتم‌ها حریصانه، از زیرساختار بهینه به صورت از بالا به پایین استفاده می‌کنیم. به جای این که اول جواب بهینه‌ی زیرمسئله‌ها را بیابیم و سپس از میان آن‌ها یکی را انتخاب کنیم، ابتدا یک انتخاب انجام می‌دهیم – انتخابی که در همان زمان به نظر بهترین می‌آید – و سپس زیرمسئله‌ی مربوطه را حل می‌کنیم، بدون این که به خود زحمت حل تمام زیرمسئله‌های کوچک‌تر مربوط را بدهیم. جالب این جاست که در بسیاری موارد، این استراتژی به خوبی عمل می‌کند!

## مهارت‌ها

باید مواظب باشیم که وقتی زیرساختار بهینه کاربرد ندارد، آن را به کار نبریم. دو مسئله‌ی زیر را در نظر بگیرید که در آن‌ها به ما گراف جهت دار $G = (V, E)$ و رأس‌های $u, v \in V$ داده شده است.

- **کوتاه‌ترین مسیر بدون وزن**[۱]: یک مسیر از $u$ به $v$ شامل کم‌ترین تعداد یال بیابید. چنین مسیری باید ساده باشد، چرا که حذف یک دور از مسیر، یک مسیر با تعداد کم‌تری یال می‌سازد.
- **بلندترین مسیر ساده‌ی بدون وزن**: یک مسیر ساده از $u$ به $v$ شامل بیشترین تعداد یال بیابید. باید ساده بودن مسیر را اجباری کنیم، چرا که در غیر این صورت می‌توانیم یک دور را به هر تعداد دلخواه دور بزنیم، و یک مسیر با بی‌نهایت یال بسازیم.

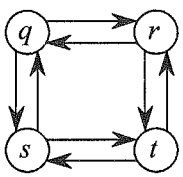
مسئله‌ی کوتاه‌ترین مسیر بدون وزن از زیرساختار بهینه پیروی می‌کند، مانند زیر. فرض کنید که $u \neq v$، و مسئله غیر بدیهی است. در این صورت هر مسیر $p$ از $u$ به $v$ باید شامل یک رأس میانی، مانند $w$ باشد. (توجه کنید که $w$ ممکن است $u$ یا $v$ باشد.) بنابراین می‌توانیم مسیر $u \overset{p}{\rightsquigarrow} v$ را به مسیرهای $u \overset{p_۱}{\rightsquigarrow} w \overset{p_۲}{\rightsquigarrow} v$ تجزیه کنیم. بدیهی است که تعداد یال‌ها در $p$ برابر است با مجموع تعداد

[۱] از اصطلاح «بدون وزن» برای تشخیص این مسئله از مسئله‌ی یافتن کوتاه‌ترین مسیرها با یال‌های وزن‌دار استفاده می‌کنیم، که آن را در فصل ۲۴ و ۲۵ خواهیم دید. می‌توانیم از تکنیک جستجوی سطح اول فصل ۲۲ برای مسئله‌ی بدون وزن استفاده کنیم.

یال‌ها در $p_1$ به علاوه‌ی تعداد یال‌ها در $p_2$. ادعا می‌کنیم که اگر $p$ یک مسیر بهینه (کوتاه‌ترین) از $u$ به $v$ باشد، در این صورت $p_1$ باید کوتاه‌ترین مسیر از $u$ به $w$ باشد. چرا؟ از روش «برش و چسباندن» استفاده می‌کنیم: اگر یک مسیر دیگر، مثلاً $p_1'$ از $u$ به $w$ با تعداد کم‌تری یال از $p_1$ وجود داشت، آن‌گاه می‌توانستیم $p_1$ را بریده و $p_1'$ را به جای آن بچسبانیم تا مسیر $u \overset{p_1'}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ با تعداد کم‌تری یال از $p$ ساخته شود، که با بهینه بودن $p$ تناقض دارد. متشابها، $p_2$ هم باید کوتاه‌ترین مسیر از $w$ به $v$ باشد. بنابراین می‌توانیم کوتاه‌ترین مسیر از $u$ به $v$ را در نظر گرفتن تمام رأس‌های میانی $w$، یافتن کوتاه‌ترین مسیرها از $u$ به $w$ و از $w$ به $v$، و انتخاب یک رأس میانی $w$ که به کوتاه‌ترین مسیر منجر می‌شود، بیابیم. در بخش ۲۵-۲، نسخه‌ی دیگری از این مشاهدات در مورد زیرساختار بهینه را برای یافتن کوتاه‌ترین مسیر میان هر دو جفت رأس در یک گراف جهت دار بدون وزن به کار می‌بریم.

بسیار وسوسه کننده است که فرض کنیم مسئله‌ی یافتن یک طولانی‌ترین مسیر بدون وزن از زیرساختار بهینه پیروی می‌کند. آیا اگر طولانی‌ترین مسیر ساده‌ی $u \overset{p}{\rightsquigarrow} v$ را به زیرمسیرهای $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ تقسیم کنیم، آیا نباید $p_1$ طولانی‌ترین مسیر ساده از $u$ به $w$ باشد، و آیا نباید $p_2$ طولانی‌ترین مسیر ساده از $w$ به $v$ باشد؟ جواب خیر است. شکل ۱۵-۶ یک مثال را نشان می‌دهد. مسیر $p \to r \to t$ را در نظر بگیرید، که یک طولانی‌ترین مسیر ساده از $q$ به $t$ است. آیا $q \to r$ یک طولانی‌ترین مسیر ساده از $q$ به $r$ است؟ خیر، چرا که مسیر $q \to s \to t \to r$ یک مسیر ساده و طولانی‌تر است. آیا $r \to t$ یک طولانی‌ترین مسیر ساده از $r$ به $t$ است؟ دوباره خیر، چرا که مسیر $r \to q \to s \to t$ یک مسیر ساده و طولانی‌تر است.

این مثال نشان می‌دهد که برای طولانی‌ترین مسیرهای ساده، نه تنها زیرساختار بهینه وجود ندارد، بلکه نمی‌توانیم لزوماً از جواب زیرمسئله‌ها، یک جواب «مجاز» برای کل مسئله بسازیم. اگر طولانی‌ترین مسیرهای $r \to q \to s \to t$ و $q \to s \to t \to r$ را با یکدیگر ترکیب کنیم، مسیر $r \to q \to s \to t \to r$ را خواهیم داشت، که ساده نیست. می‌بینیم که مسئله‌ی یافتن طولانی‌ترین مسیر ساده‌ی بدون وزن هیچ گونه زیرساختار بهینه‌ای ندارد. تا کنون هیچ الگوریتم




**شکل ۱۵-۶** یک گراف جهت دار که نشان می‌دهد که مسئله‌ی یافتن یک طولانی‌ترین مسیر ساده در یک گراف جهت دار بدون وزن دارای زیرساختار بهینه نیست. مسیر $q \to r \to t$ یک طولانی‌ترین مسیر ساده از $q$ به $r$ است، ولی زیرمسیر $q \to r$ طولانی‌ترین مسیر ساده از $q$ به $r$ نیست، و زیرمسیر $r \to t$ طولانی‌ترین مسیر ساده از $r$ به $t$ نیست.

برنامه‌ریزی پویای کارایی برای این مسئله یافت نشده است. در واقع این مسئله NP–کامل است، که – همان طور که در فصل ۳۴ خواهیم دید – بدین معنی است که به احتمال زیاد نمی‌توان آن را در زمان چند جمله‌ای حل کرد.

چه چیزی در مورد زیرساختار طولانی‌ترین مسیرها وجود دارد که این قدر با زیرساختار کوتاه‌ترین مسیرها متفاوت است؟ با این که در یک جواب برای هر دو مسئله‌ی طولانی‌ترین و کوتاه‌ترین مسیر، از دو زیرمسئله استفاده می‌شود، زیرمسئله‌های یافتن طولانی‌ترین مسیر، از یکدیگر *مستقل* نیستند، در حالی که برای کوتاه‌ترین مسیرها این گونه است. منظور از مستقل بودن زیرمسئله‌ها چیست؟ یعنی جواب یک زیرمسئله بر روی جواب زیرمسئله‌ای دیگر از یک مسئله‌ی خاص تأثیری نمی‌گذارد. برای مثال شکل ۱۵–۶، مسئله‌ی یافتن یک طولانی‌ترین مسیر ساده از $q$ به $t$ را داریم، با دو زیرمسئله: یافتن طولانی‌ترین مسیر ساده از $q$ به $r$ و از $r$ به $t$. برای زیرمسئله‌ی اول، مسیر $q \to s \to t \to r$ را انتخاب می‌کنیم، که در آن از رأس‌های $s$ و $t$ استفاده شده است. دیگر نمی‌توانیم از این رأس‌ها در زیرمسئله‌ی دوم استفاده کنیم، چرا که در این صورت ترکیب جواب‌های مسئله‌ها، یک مسیر ساده نخواهد بود. اگر نتوانیم از رأس $t$ در زیرمسئله‌ی دوم استفاده کنیم، نمی‌توانیم آن را به طور کامل حل کنیم، زیرا $t$ باید روی مسیری باشد که پیدا می‌کنیم، و $t$ رأسی نیست که ما در آن جواب زیرمسئله‌ها را به هم «متصل» می‌کنیم (این رأس $r$ است). استفاده از $s$ و $t$ در جواب یک زیرمسئله، باعث می‌شود که نتوانیم از آن‌ها در جواب زیرمسئله‌ی دیگر استفاده کنیم. با این حال، باید حداقل از یکی از آن‌ها برای جواب زیرمسئله‌ی دیگر استفاده کنیم، و برای حل کردن به صورت بهینه، باید از هر دوی آن‌ها استفاده کنیم. بنابراین می‌گوییم که این زیرمسئله‌ها مستقل نیستند. به صورت دیگر، می‌توان گفت که استفاده از منابع در حل یکی از زیرمسئله‌ها (که این منابع رأس‌ها هستند) باعث می‌شود که نتوانیم از آن‌ها در حل زیرمسئله‌ی دیگر استفاده کنیم.

پس چرا زیرمسئله‌ها برای یافتن کوتاه‌ترین مسیر مستقل هستند؟ جواب این است که به طور طبیعی، در این مسئله زیرمسئله‌ها از منابع مشترک استفاده نمی‌کنند. ادعا می‌کنیم که اگر یک رأس $w$ بر روی یک کوتاه‌ترین $p$ مسیر از $u$ به $v$ است، آن گاه می‌توانیم هر کوتاه‌ترین مسیر $u \overset{p_1}{\rightsquigarrow} w$ و هر کوتاه‌ترین مسیر $w \overset{p_2}{\rightsquigarrow} v$ را به یکدیگر متصل کرده و یک کوتاه‌ترین مسیر از $u$ به $v$ بسازیم. مطمئن هستیم که غیر از $w$، هیچ رأسی بر روی هر دو مسیر $p_1$ و $p_2$ نیست. چرا؟ فرض کنید که یک رأس $x \neq w$ در هر دو مسیر $p_1$ و $p_2$ وجود دارد، به طوری که می‌توانیم $p_1$ را به صورت $u \overset{p_{ux}}{\rightsquigarrow} x \rightsquigarrow w$ و $p_2$ را به صورت $w \rightsquigarrow x \overset{p_{xu}}{\rightsquigarrow} v$ تجزیه کنیم. طبق زیرساختار بهینه‌ی این مسئله، تعداد یال‌های مسیر $p$ برابر است با مجموع تعداد یال‌های $p_1$ و $p_2$؛ فرض می‌کنیم $p$ دارای $e$ یال باشد. اکنون اجازه دهید یک مسیر $u \overset{p_{ux}}{\rightsquigarrow} x \overset{p_{xv}}{\rightsquigarrow} v$ از $u$ به $v$ بسازیم. چون مسیرهای از $x$ به $w$ و از $w$ به $x$ را حذف کرده‌ایم، که هر کدام حداقل یک یال دارند، این مسیر حداکثر $e-2$ یال دارد، که با فرض کوتاه‌ترین

مسیر بودن $p$ تناقض دارد. بنابراین، اطمینان داریم که زیرمسئله‌ها در مسئله‌ی یافتن کوتاه‌ترین مسیر مستقل هستند.

هر دو مسئله‌ی بررسی شده در بخش‌های ۱۵-۱ و ۱۵-۲ دارای زیرمسئله‌های مستقل هستند. در ضرب زنجیره‌ی ماتریس‌ها، زیرمسئله‌ها ضرب زنجیره‌های $A_i A_{i+1} \ldots A_k$ و $A_{k+1} A_{k+2} \ldots A_j$ هستند. این زیرزنجیرها گسسته هستند، و هیچ ماتریسی نمی‌تواند در هر دوی آن‌ها باشد. در مسئله‌ی برش میله، برای تعیین بهترین راه برای برش میله‌ای به طول $n$، بهترین روش‌ها برای برش میله‌هایی به طول $i$ برای $i = 0,1,\ldots,n-1$ را می‌یابیم. چون یک جواب بهینه برای مسئله با ورودی $n$ حاوی جواب فقط یکی از این زیرمسئله‌ها است (بعد از این که اولین قطعه را بریدیم)، مستقل بودن زیرمسئله‌ها مهم نخواهد بود.

## زیرمسئله‌های مشترک

دومین جزئی که یک مسئله‌ی بهینه‌سازی باید داشته باشد تا بتوان از برنامه‌ریزی پویا برای حل آن استفاده کرد، این است که فضای زیرمسئله‌ها باید «کوچک» باشد، بدین صورت که یک الگوریتم بازگشتی برای مسئله به جای این که همیشه زیرمسئله‌های با جدید برخورد کند، زیرمسئله‌های مشابهی را مکرراً حل کند. معمولاً تعداد کل زیرمسئله‌های یکتا نسبت به اندازه‌ی ورودی از مرتبه‌ی چند جمله‌ای است. وقتی یک الگوریتم بازگشتی مکرراً به یک زیرمسئله برخورد می‌کند، می‌گوییم مسئله‌ی بهینه‌سازی *زیرمسئله‌های مشترک*[1] دارد. در مقابل، مسئله‌ای که رویکرد تقسیم و حل برای آن کاربرد دارد، معمولاً در هر مرحله‌ی بازگشت زیرمسئله‌های کاملاً جدید تولید می‌کند. الگوریتم‌های برنامه‌ریزی پویا معمولاً از خصوصیت زیرمسئله‌های مشترک بهره می‌گیرند، بدین صورت که هر زیرمسئله را یک بار حل، و جواب آن را در یک جدول ذخیره می‌کنند تا بتوانند در هنگام نیاز دوباره به آن رجوع کنند، که زمان جستجوی آن ثابت است.

در بخش ۱۵-۱ مختصرا توضیح دادیم که چگونه یک الگوریتم بازگشتی برای برش میله، تعداد زیادی (از مرتبه‌ی نمایی) فراخوانی برای یافتن جواب زیرمسئله‌های کوچک‌تر می‌کند. جواب برنامه‌ریزی پویای ما یک الگوریتم بازگشتی با زمان نمایی را به یک الگوریتم با زمان درجه‌ی دو تبدیل می‌کند.

برای توضیح خصوصیت زیرمسئله‌های مشترک با جزئیات بیشتر، اجازه دهید دوباره مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها را در نظر بگیریم. با مراجعه‌ی دوباره به شکل ۱۵-۵، مشاهده می‌کنیم که MATRIX-CHAIN-ORDER برای حل زیرمسئله‌ها در ردیف‌های بالا، مکرراً به جواب زیرمسئله‌ها در ردیف‌های پایین‌تر مراجعه می‌کند. به عنوان مثال، ورودی $m[3,4]$ چهار بار مورد دسترسی قرار

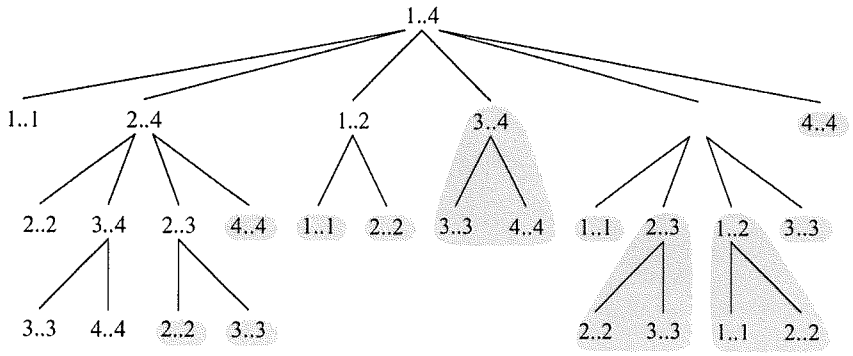
[1] ممکن است عجیب به نظر برسد که برنامه‌ریزی پویا هم به مستقل بودن زیرمسئله‌ها وابسته است و هم به مشترک بودن آن‌ها. با این که این پیش‌نیازها ممکن است متناقض به نظر برسند، با این حال نشان‌دهنده‌ی دو مفهوم مختلف هستند، نه دو نقطه روی یک محور. دو زیرمسئله از یک مسئله مستقل هستند اگر از منابع مشترک استفاده نکنند. دو زیرمسئله مشترک هستند در واقع یک زیرمسئله باشند که به عنوان زیرمسئله‌های دو مسئله‌ی مختلف ظاهر می‌شوند.

می‌گیرد: برای محاسبه‌ی $m[۲,۴]$، $m[۱,۴]$، $m[۳,۵]$، و $m[۳,۴]$. اگر هر بار بخواهیم $m[۳,۴]$ را مجددا محاسبه کنیم، به جای این که از جدول جواب آن را ببینیم، افزایش زمان اجرا وحشتناک خواهد بود. برای دیدن این مسئله، الگوریتم بازگشتی (نابهینه‌ی) زیر را در نظر بگیرید، که $m[i,j]$ را تعیین می‌کند، یعنی کمینه‌ی تعداد ضرب‌های اسکالر مورد نیاز برای محاسبه‌ی ضرب ماتریسی $A_{i..j} = A_i A_{i+۱} ... A_j$. این رویه مستقیماً بر پایه‌ی رابطه‌ی بازگشتی (۱۵–۷) نوشته شده است.

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j - 1
5      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          m[i, j] = q
8  return m[i, j]
```

شکل ۱۵–۷ درخت بازگشتی ساخته شده توسط فراخوانی RECURSIVE-MATRIX-CHAIN $(p,۱,۴)$ را نشان می‌دهد. هر گره با مقادیر پارامترهای $i$ و $j$ علامت گذاری شده است. توجه کنید که بعضی از جفت‌ها چندین بار تکرار شده‌اند.

در واقع می‌توانیم نشان دهیم که زمان محاسبه‌ی $m[۱,n]$ توسط این رویه‌ی بازگشتی نسبت به $n$ حداقل نمایی است. فرض کنید که $T(n)$ نشان دهنده‌ی زمان صرف شده توسط RECURSIVE-MATRIX-CHAIN برای محاسبه‌ی یک پرانتزگذاری بهینه برای زنجیره‌ای از $n$ ماتریس باشد. اگر فرض کنیم که اجرای خطوط ۱–۲ و ۶–۷ حداقل به زمان ثابت نیاز دارد، مانند ضرب خط ۵، مشاهده‌ی رویه رابطه‌ی بازگشتی زیر را به ما خواهد داد:




**شکل ۱۵–۷** درخت بازگشتی برای محاسبه‌ی RECURSICE-MATRIX-CHAIN $(p,۱,۴)$. هر گره حاوی پارامترهای $i$ و $j$ است. محاسبه‌های انجام شده در زیردرخت‌های سایه‌دار در MEMORIZE-MATRIX-CHAIN با یک جستجو در جدول جایگزین شده‌اند.

$$T(1) \geq 1$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{برای } n > 1$$

با توجه به این که برای $i = 1, 2, \ldots, n$، هر عبارت $T(i)$ یک بار به شکل $T(k)$ و یک بار به شکل $T(n-k)$ ظاهر می‌شود، و با جمع آوری ۱ها در سری (که تعداد آن‌ها $n-1$ است)، به علاوه‌ی آخرین ۱، می‌توانیم رابطه‌ی بازگشتی را به صورت زیر بازنویسی کنیم:

$$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n \qquad (8\text{-}15)$$

با استفاده از متد جانشینی اثبات خواهیم کرد که $T(n) = \Omega(2^n)$. به طور خاص، نشان خواهیم داد که برای تمام $n$هایی که $n \geq 1$، $T(n) \geq 2^{n-1}$. پایه‌ی نامساوی ساده است، چرا که $T(1) \geq 1 = 2^0$. به صورت استقرایی، برای $n \geq 2$ داریم:

$$\begin{aligned}
T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n \\
&= 2\sum_{i=0}^{n-2} 2^i + n \\
&= 2(2^{n-1} - 1) + n \qquad \text{(طبق تساوی (الف-۵))} \\
&= (2^n - 2) + n \\
&\geq 2^{n-1}
\end{aligned}$$

که اثبات را کامل می‌کند. بنابراین، کل کار انجام شده توسط فراخوانی RECURSIVE-MATRIX-CHAIN$(p, 1, n)$، نسبت به $n$ حداقل از مرتبه‌ی نمایی است.

این الگوریتم بازگشتی از بالا به پایین (با به خاطر سپاری) را با الگوریتم برنامه‌ریزی پویای از پایین به بالا مقایسه کنید. دومی کاراتر است چرا که از خصوصیت زیرمسئله‌های مشترک بهره می‌گیرد. فقط $\theta(n^2)$ زیرمسئله‌ی متفاوت وجود دارد، و الگوریتم برنامه‌ریزی پویا هر یک از آن‌ها را فقط یک بار حل می‌کند. از طرف دیگر، الگوریتم بازگشتی هر بار که در درخت بازگشتی به یک زیرمسئله بر می‌خورد، دوباره آن را حل می‌کند. هر جایی که یک درخت بازگشتی برای راه حل بازگشتی یک مسئله حاوی زیرمسئله‌های تکراری است، و تعداد کل زیرمسئله‌های متفاوت کم است، برنامه‌ریزی پویا می‌تواند کارایی را، در بعضی مواقع به شدت، افزایش دهد.

## بازسازی یک جواب بهینه

به عنوان یک مسئله‌ی کاربردی، معمولاً انتخاب انجام شده در هر زیرمسئله را در یک جدول ذخیره می‌کنیم تا مجبور نباشیم که از روی هزینه‌های ذخیره شده، این اطلاعات را بازسازی کنیم.

برای ضرب زنجیره‌ی ماتریس‌ها، جدول $s[i, j]$، هنگام بازسازی یک جواب بهینه به مقدار قابل

توجهی صرفه‌جویی در زمان به همراه دارد. فرض کنید که جدول $s[i,j]$ را پر نمی‌کردیم، و فقط جدول $m[i,j]$ که حاوی هزینه‌ی بهینه‌ی زیرمسئله‌ها است را پر می‌کردیم. برای پرانتز گذاری $A_i A_{i+1} \ldots A_j$ تعداد $j-i$ انتخاب برای تعیین زیرمسئله‌ی مورد استفاده در جواب بهینه وجود دارد، و $j-i$ یک ثابت نیست. بنابراین، برای تعیین زیرمسئله‌ی انتخاب شده در جواب یک مسئله‌ی خاص، به زمان $\theta(j-i) = \omega(1)$ نیاز داریم. با ذخیره‌ی اندیس ماتریسی که در مکان آن ضرب را به دو قسمت تقسیم می‌کنیم در $s[i,j]$، می‌توانیم هر انتخاب را در زمان $O(1)$ بازسازی کنیم.

## به خاطر سپاری

همان‌طور که در برای مسئله‌ی برش میله دیدیم، نسخه‌ی دیگری از برنامه‌ریزی پویا وجود دارد که معمولاً کارایی آن به اندازه‌ی کارایی برنامه‌ریزی پویای معمولی است، ولی از استراتژی بالا به پایین استفاده می‌کند. ایده این است که در الگوریتم بازگشتی ناکارآمد، از *به خاطر سپاری* (memoization) استفاده کنیم. مانند روش پایین به بالا، یک جدول با جواب زیرمسئله‌ها نگه می‌داریم، ولی ساختار کنترلی برای پر کردن جدول بیشتر شبیه الگوریتم‌های بازگشتی است.

یک الگوریتم بازگشتی با به خاطر سپاری، برای جواب هر زیرمسئله یک خانه در جدول نگه می‌دارد. در ابتدا هر خانه‌ی جدول حاوی یک مقدار خاص است که نشان می‌دهد این خانه هنوز پر نشده است. وقتی برای اولین بار در زمان اجرای الگوریتم بازگشتی به یک زیرمسئله‌ی خاص برمی‌خوریم، جواب آن محاسبه شده و در جدول ذخیره می‌شود. دفعات بعدی که با این زیرمسئله برخورد می‌کنیم، به سادگی جواب آن را از درون جدول استخراج کرده و آن را بازمی‌گردانیم.[1]

در این جا، یک نسخه از رویه‌ی RECURSIVE-MATRIX-CHAIN با به خاطر سپاری ارائه شده است. توجه کنید که از چه جهاتی مشابه متد بالا به پایین مسئله‌ی برش میله با به خاطر سپاری است.

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length - 1
2  for i = 1 to n
3      for j = i to n
4          m[i, j] = ∞
5  return LOOKUP-CHAIN(p, 1, n)

LOOKUP-CHAIN(p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
```

[1] این رویکرد به صورت پیش فرض در نظر می‌گیرد که تمام پارامترهای ممکن زیرمسئله‌ها شناخته شده هستند و رابطه‌ی بین مکان‌های جدول و زیرمسئله‌ها تعیین شده است. یک رویکرد دیگر این است که به خاطر سپاری را با درهم‌سازی پارامترهای زیرمسئله‌ها به صورت کلید انجام دهیم.

```
5  else for k = i to j - 1
6          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              m[i, j] = q
9  return m[i, j]
```

MEMOIZED-MATRIX-CHAIN، مانند MATRIX-CHAIN-ORDER یک جدول $m[1..n,1..n]$ از مقادیر محاسبه شده‌ی $m[i,j]$، کمینه‌ی تعداد ضرب‌های اسکالر مورد نیاز برای محاسبه‌ی ماتریس $A_{i..j}$، نگه می‌دارد. در ابتدا هر خانه‌ی جدول حاوی مقدار $\infty$ است که نشان می‌دهد این خانه هنوز پر نشده است. وقتی فراخوانی LOOKUP-CHAIN $(p,i,j)$ انجام می‌شود، اگر در خط ۱، $m[i,j] < \infty$ برقرار باشد، به سادگی مقدار از قبل محاسبه شده‌ی $m[i,j]$ در خط ۲ بازگردانده می‌شود. در غیر این صورت، هزینه توسط RECURSIVE-MATRIX-CHAIN محاسبه، در $m[i,j]$ ذخیره، و سپس بازگردانده می‌شود. بنابراین، LOOKUP-CHAIN $(p,i,j)$ همیشه مقدار $m[i,j]$ را بازمی‌گرداند، ولی فقط در صورتی آن را محاسبه می‌کند که این اولین باری باشد LOOKUP-CHAIN با پارامترهای $i$ و $j$ فراخوانی شده است.

شکل ۱۵-۷ نشان می‌دهد که MEMOIZED-MATRIX-CHAIN در مقایسه با RECURSIVE-MATRIX-CHAIN چگونه در زمان صرفه جویی می‌کند. زیردرخت‌های سایه دار نشان دهنده‌ی مقادیری هستند که به جای محاسبه شدن، در جدول جستجو می‌شوند.

مانند الگوریتم برنامه‌ریزی پویا و پایین به بالای MATRIX-CHAIN-ORDER، رویه‌ی MEMOIZED-MATRIX-CHAIN در زمان $O(n^3)$ اجرا می‌شود. خط ۵ رویه‌ی MEMOIZED-MATRIX-CHAIN به تعداد $\theta(n^2)$ بار اجرا می‌شود. می‌توانیم فراخوانی‌های LOOKUP-CHAIN را به دو دسته تقسیم کنیم:

۱. فراخوانی‌هایی که در آن‌ها $m[i,j] = \infty$، به طوری که خطوط ۳-۹ اجرا می‌شوند، و
۲. فراخوانی‌هایی که در آن‌ها $m[i,j] < \infty$، به طوری که LOOKUP-CHAIN به سادگی در خط ۲ بازگشت می‌کند.

$\theta(n^2)$ فراخوانی از نوع اول وجود دارد، یکی برای هر یک از خانه‌های جدول. تمام فراخوانی‌های نوع دوم به صورت بازگشتی توسط فراخوانی‌های نوع اول انجام می‌شوند. هر گاه که یک فراخوانی خاص از LOOKUP-CHAIN، فراخوانی‌های بازگشتی انجام می‌دهد، تعداد این فراخوانی‌ها $O(n)$ است. بنابراین در کل $O(n^3)$ فراخوانی از نوع دوم وجود دارد. هر فراخوانی از نوع دوم در زمان $O(1)$ انجام می‌شود، و هر فراخوانی از نوع اول به زمان $O(n)$ نیاز دارد، به علاوه‌ی زمان صرف شده در فراخوانی‌های بازگشتی آن. بنابراین، زمان کلی برابر است با $O(n^3)$. بنابراین با استفاده از به خاطر سپاری، یک الگوریتم با زمان $\Omega(2^n)$ تبدیل به یک الگوریتم با زمان $O(n^3)$ شده است.

در مجموع، مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها را می‌توان یا با استفاده از یک الگوریتم به خاطرسپاری از بالا به پایین، و یا با استفاده از یک الگوریتم برنامه‌ریزی پویای از بالا به پایین، در زمان $O(n^3)$ حل کرد. هر دو متد از خصوصیت زیرمسئله‌های مشترک بهره می‌گیرند. در مجموع فقط $\theta(n^2)$ زیرمسئله‌ی مجزا وجود دارد، و هر دو متد، جواب هر یک از این زیرمسئله‌ها را فقط یک بار محاسبه می‌کنند. بدون به خاطر سپاری، الگوریتم بازگشتی معمولی در زمان نمایی اجرا می‌شود، چرا که جواب زیرمسئله‌های تکراری را مکرراً محاسبه می‌کند.

به طور کلی، درعمل اگر تمام زیرمسئله‌ها باید حداقل یک بار محاسبه شوند، کارایی یک الگوریتم برنامه‌ریزی پویای از پایین به بالا با یک ضریب ثابت از کارایی یک الگوریتم به خاطرسپاری از بالا به پایین بهتر است، چرا که سرباری برای فراخوانی بازگشتی وجود ندارد، و همچنین سربار نگه‌داری جدول کم‌تر است. به علاوه، مسئله‌هایی هستند که برای آن‌ها می‌توان روش معمولی درسترسی به جدول را طوری تغییر داد که زمان یا فضای مورد نیاز باز هم کاهش پیدا کند. در عوض، اگر به جواب بعضی از زیرمسئله‌ها در فضای مسئله اصلاً نیازی نباشد، به خاطر سپاری این مزیت را دارد که فقط زیرمسئله‌هایی را حل می‌کند که قطعا به جواب آن‌ها نیاز است.

## تمرین‌ها

۱۵-۳-۱ کدام یک از روش‌های زیر برای تعیین تعداد بهینه‌ی ضرب‌های مورد نیاز در مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها کاراتر است: بررسی تمام روش‌های پرانتز گذاری ضرب و محاسبه‌ی تعداد ضرب‌ها برای هر یک، یا اجرای RECURSIVE-MATRIX-CHAIN؟ جواب خود را توجیه کنید.

۱۵-۳-۲ درخت بازگشتی برای رویه‌ی MERGE-SORT از بخش ۲-۳-۱ را بر روی یک آرایه با ۱۶ عنصر بکشید. توضیح دهید که چرا برای سرعت بخشیدن به یک الگوریتم خوب تقسیم و حل مانند MERGE-SORT ، به خاطر سپاری کارایی ندارد؟

۱۵-۳-۳ نسخه‌ای از مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها را در نظر بگیرید که در آن هدف این است که طوری دنباله‌ی ماتریس‌ها را پرانتز گذاری کنیم که تعداد ضرب‌های مورد نیاز، به جای کمینه شدن، بیشینه شود. آیا این مسئله دارای زیرساختار بهینه است؟

۱۵-۳-۴ همان طور که قبلاً گفته شد، در برنامه‌ریزی پویا اول زیرمسئله‌ها را حل می‌کنیم و سپس انتخاب می‌کنیم که از کدام یک از آن‌ها برای جواب بهینه‌ی مسئله استفاده کنیم. پروفسور کاپیولت (Capulet) ادعا می‌کند که همیشه نیاز نیست که برای یافتن جواب بهینه، تمام زیرمسئله‌ها را حل کنیم. او پیشنهاد می‌کند که می‌توان با انتخاب ماتریس $A_k$ برای قسمت کردن زنجیره‌ی $A_i A_{i+1} \cdots A_j$ (با انتخاب $k$ به طوری که عبارت $p_{i-1} p_k p_j$ کمینه شود) *قبل* از حل زیرمسئله‌ها، یک جواب بهینه برای مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها پیدا

کرد. یک نمونه از مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها بیابید که در آن این رویکرد حریصانه به یک جواب غیر بهینه ختم می‌شود.

۱۵-۳-۵ فرض کنید که در مسئله‌ی برش میله در بخش ۱۵-۱، محدودیت $l_i$ را بر روی تعداد قطعه‌های ساخته شده با طول $i$ داشتیم، برای $i = 1,2,...,n$. نشان دهید که زیرساختار بهینه‌ی توصیف شده در بخش ۱۵-۱ دیگر برقرار نیست.

۱۵-۳-۶ فرض کنید که می‌خواهید مقداری پول را از یک واحد به واحد دیگر تبدیل کنید. متوجه می‌شوید که به جای تبدیل مستقیم از یک واحد پول به واحد دیگر، ممکن است بهتر باشد که چندین تبدیل در میان واحدهای دیگر انجام دهید، و در نهایت به واحد مورد نظر خود برسید. فرض کنید که می‌توانید بین $n$ واحد مختلف، با شماره‌های $1,2,...,n$، پول خود را تبدیل کنید، و هدف، تبدیل پول از واحد ۱ به واحد $n$ است. برای هر جفت واحد $i$ و $j$ یک نرخ تبدیل $r_{ij}$ به شما داده شده است، که بدین معنی است که اگر $d$ واحد از پول $i$ داشته باشید، می‌توانید آن را به $d\,r_{ij}$ واحد از پول $j$ تبدیل کنید. ممکن است دنباله‌ای از تبدیل‌ها مقداری کارمزد به همراه داشته باشند، که به تعداد معامله‌هایی که انجام می‌دهید بستگی دارد. فرض کنید $c_k$ نشان دهنده‌ی کارمزد پرداختی هنگام انجام $k$ تبدیل باشد. نشان دهید که اگر $c_k = 0$ برای تمام $k = 1,2,...,n$، آن گاه مسئله‌ی یافتن بهترین دنباله‌ی تبدیل پول از واحد ۱ به واحد $n$ دارای زیرساختار بهینه است. سپس نشان دهید که اگر کارمزدهای $c_k$ مقادیر دلخواه باشند، آن گاه مسئله‌ی یافتن بهترین دنباله‌ی تبدیل پول از واحد ۱ به واحد $n$ لزوماً دارای زیرساختار بهینه نیست.

## ۱۵-۴ طولانی‌ترین زیردنباله‌ی مشترک

در کاربردهای بیولوژیکی، معمولاً نیاز داریم که DNA دو (یا چند) جاندار مختلف را با هم مقایسه کنیم. یک رشته‌ی DNA حاوی رشته‌ای از ملکول‌ها به نام *پایه* (base) است، که پایه‌های موجود عبارتند از adenine، guanine، cytosine و thymine. با نمایش هر یک از این پایه‌ها به صورت حرف آغازین آن‌ها، می‌توان یک رشته‌ی DNA را به صورت رشته‌ای بر روی مجموعه‌ی متناهی $\{A,C,G,T\}$ نشان داد. (برای تعریف رشته به پیوست پ مراجعه کنید.) به عنوان مثال، ممکن است DNA دو جانور مختلف به صورت $S_1$=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA و $S_2$=GTCGTTCGGAAT GCCGTTGCTCTGTAAA باشد. یکی از اهداف مقایسه‌ی دو رشته‌ی مختلف DNA این است که ببینیم با تعریف خاصی از شباهت، این دو چقدر به یکدیگر «شبیه» هستند. شباهت می‌تواند به اشکال مختلفی تعریف شود (و می‌شود). مثلاً، می‌توانیم بگوییم که رشته‌های DNA در صورتی مشابه هستند که یکی زیررشته‌ی دیگری باشد. (در فصل ۳۲ الگوریتم‌هایی برای حل این

مسئله ارائه می‌شود.) در مثال ما، نه $S_1$ زیررشته‌ی $S_2$ است و نه $S_2$ زیررشته‌ی $S_1$. در عوض، می‌توانیم بگوییم که دو رشته مشابه هستند اگر تعداد تغییرات مورد نیاز برای تبدیل یکی از رشته‌ها به دیگری کوچک باشد. (مسئله‌ی ۱۵-۳ نگاهی بر این قضیه دارد.) روشی دیگر برای تعیین مشابهت رشته‌های $S_1$ و $S_2$ این است که رشته‌ای مانند $S_3$ بیابیم که پایه‌های $S_3$ هم در $S_1$ موجود باشد و هم در $S_2$؛ این پایه‌ها باید به ترتیب مشابه ظاهر شوند، ولی نه لزوماً پشت سر هم. هر چه طول $S_3$ که می‌یابیم بیشتر باشد، $S_1$ و $S_2$ به یکدیگر شبیه‌تر هستند. در مثال بالا، طولانی‌ترین $S_3$ ممکن GTCGTCGGAAGCCGGCCGAA است.

آخرین تعریف شباهت که در بالا گفته شد به مسئله‌ی طولانی‌ترین زیررشته‌ی مشترک معروف است. یک زیررشته از یک رشته‌ی داده شده، فقط همان دنباله است که صفر یا تعدادی از عناصر آن بیرون انداخته شده‌اند. به طور رسمی، با داشتن یک دنباله‌ی $X = \langle x_1, x_2, ..., x_m \rangle$، دنباله‌ی دیگر $Z = \langle z_1, z_2, ..., z_k \rangle$ یک *زیردنباله* از $X$ است اگر یک دنباله‌ی صعودی اکید $\langle i_1, i_2, ..., i_k \rangle$ از اندیس‌های $X$ موجود باشد به طوری که برای تمام $j = 1, 2, ..., k$، داشته باشیم $x_{i_j} = z_i$. به عنوان مثال، $Z = \langle B, C, D, B \rangle$ یک زیردنباله از $X = \langle A, B, C, B, D, A, B \rangle$ با اندیس‌های مربوطه‌ی $\langle 2, 3, 5, 7 \rangle$ است.

با داشتن دو دنباله‌ی $X$ و $Y$، می‌گوییم دنباله‌ی $Z$ یک *زیردنباله‌ی مشترک* $X$ و $Y$ است اگر $Z$ هم زیردنباله‌ای از $X$ و هم زیردنباله‌ای از $Y$ باشد. به عنوان مثال، اگر $X = \langle A, B, C, B, D, A, B \rangle$ و $Y = \langle B, D, C, A, B, A \rangle$، دنباله‌ی $\langle B, C, A \rangle$ یک زیردنباله‌ی مشترک از $X$ و $Y$ است. با این حال، این دنباله *بلندترین* زیردنباله‌ی مشترک $X$ و $Y$ نیست، چرا که طول این دنباله ۳ است و دنباله‌ی $\langle B, C, B, A \rangle$ که یک زیردنباله‌ی مشترک $X$ و $Y$ است، دارای طول ۴ است. دنباله‌ی $\langle B, C, B, A \rangle$ یک بلندترین زیردنباله‌ی مشترک $X$ و $Y$ است، همین طور دنباله‌ی $\langle B, D, A, B \rangle$، چرا که هیچ زیردنباله‌ی مشترکی با طول ۵ یا بیشتر برای $X$ و $Y$ وجود ندارد.

در مسئله‌ی بلندترین زیردنباله‌ی مشترک، ما دو دنباله‌ی $X = \langle x_1, x_2, ..., x_m \rangle$ و $Y = \langle y_1, y_2, ..., y_n \rangle$ داریم، و می‌خواهیم طولانی‌ترین زیردنباله‌ی مشترک $X$ و $Y$ را بیابیم. در این بخش نشان می‌دهیم که می‌توان مسئله‌ی بلندترین زیردنباله‌ی مشترک را به صورت بهینه با استفاده از برنامه‌ریزی پویا حل کرد.

## مرحله‌ی اول: تعریف بلندترین زیردنباله‌ی مشترک

یک رویکرد بی‌خردانه برای حل مسئله‌ی بلندترین زیردنباله‌ی مشترک، این است که تمام زیردنباله‌های $X$ را بسازیم و چک کنیم که آیا هر کدام از آن‌ها زیردنباله‌ی $Y$ هم هستند یا خیر، و بلندترین زیردنباله‌ی مشترک یافت شده را ذخیره کنیم. هر زیردنباله‌ی $X$ متناظر است با زیرمجموعه‌ای از اندیس‌های $\{1, 2, ..., m\}$ از $X$. تعداد $2^m$ زیردنباله برای $X$ وجود دارد، بنابراین این روش به زمان نمایی نیاز دارد، که برای دنباله‌های بلند غیر عملی است.

با این حال همان طور که قضیه‌ی زیر نشان می‌دهد، این مسئله دارای زیرساختار بهینه است. همان

طور که خواهیم دید، کلاس زیرمسئله‌ها متناظر است با جفت‌هایی از «پیشوندها»یی از دو دنباله‌ی ورودی. به طور دقیق‌تر با داشتن یک دنباله‌ی $X = \langle x_1, x_2, ..., x_m \rangle$، $i$امین پیشوند $X$ را برای $i = 0, 1, ..., m$ به صورت $X_i = \langle x_1, x_2, ..., x_i \rangle$ تعریف می‌کنیم. به عنوان مثال اگر $X = \langle A, B, C, B, D, A, B \rangle$، آن گاه $X_4 = \langle A, B, C, B \rangle$، و $X_0$ دنباله‌ی تهی است.

**قضیه‌ی ۱۵-۱ (زیرساختار بهینه‌ی مسئله‌ی بلندترین زیردنباله‌ی مشترک)**

فرض کنید $X = \langle x_1, x_2, ..., x_m \rangle$ و $Y = \langle y_1, y_2, ..., y_n \rangle$ دو دنباله باشند، و $Z = \langle z_1, z_2, ..., z_k \rangle$ یک بلندترین زیردنباله‌ی مشترک از $X$ و $Y$ باشد.

۱. اگر $x_m = y_n$، آن گاه $z_k = x_m = y_n$ و $Z_{k-1}$ یک زیردنباله‌ی مشترک $X_{m-1}$ و $Y_{n-1}$ است.

۲. اگر $x_m \neq y_n$، آن گاه $z_k \neq x_m$ نتیجه می‌دهد که $Z$ یک زیردنباله‌ی مشترک از $X_{m-1}$ و $Y$ است.

۳. اگر $x_m \neq y_n$، آن گاه $z_k \neq y_n$ نتیجه می‌دهد که $Z$ یک زیردنباله‌ی مشترک از $X$ و $Y_{n-1}$ است.

*اثبات* (۱) اگر $z_k \neq x_m$، آن گاه می‌توانیم $x_m = y_n$ را به $Z$ اضافه کنیم و یک زیردنباله‌ی مشترک از $X$ و $Y$ با طول $k+1$ به دست آوریم، که با این فرض که $X$ *بلندترین* زیردنباله‌ی مشترک $X$ و $Y$ است، تناقض دارد. بنابراین باید داشته باشیم $z_k = x_m = y_n$. اکنون، پیشوند $Z_{k-1}$ یک زیردنباله‌ی مشترک $X_{m-1}$ و $Y_{n-1}$ با طول $k-1$ است. می‌خواهیم نشان دهیم که این یک بلندترین زیردنباله‌ی مشترک است. با برهان خلف، فرض کنید که یک زیردنباله‌ی مشترک $W$ از $X_{m-1}$ و $Y_{n-1}$ با طول بلندتر از $k-1$ وجود دارد. در این صورت، اضافه کردن $x_m = y_n$ به $W$ یک زیردنباله‌ی مشترک از $X$ و $Y$ می‌سازد که طول آن از $k$ بیشتر است، و در این جا به تناقض می‌رسیم.

(۲) اگر $z_k \neq x_m$، در این صورت $Z$ یک زیردنباله‌ی مشترک از $X_{m-1}$ و $Y$ است. اگر زیردنباله‌ی مشترک $W$ از $X_{m-1}$ و $Y$ با طول بیشتر از $k$ وجود داشته باشد، آن گاه $W$ یک زیردنباله‌ی مشترک از $X_m$ و $Y$ هم هست، که با این فرض که $Z$ یک بلندترین زیردنباله‌ی مشترک $X$ و $Y$ است تناقض دارد.

(۳) اثبات این قسمت مشابه اثبات قسمت (۲) است. ■

تعریف قضیه‌ی ۱۵-۱ نشان می‌دهد که یک بلندترین زیردنباله‌ی مشترک دو دنباله، در خود حاوی یک بلندترین زیردنباله‌ی مشترک از پیشوندهای دو دنباله است. بنابراین، مسئله‌ی بلندترین زیردنباله‌ی مشترک دارای خصوصیت زیرساختار بهینه است. همچنین، یک راه حل بازگشتی برای این مسئله دارای خصوصیت زیرمسئله‌های مشترک است، که به زودی آن را هم خواهیم دید.

### مرحله‌ی ۲: یک راه حل بازگشتی

قضیه‌ی ۱۵-۱ ایجاب می‌کند که برای یافتن یک بلندترین زیردنباله‌ی مشترک از $X = \langle x_1, x_2, ..., x_m \rangle$

و $Y = \langle y_1, y_2, ..., y_n \rangle$، یک یا دو زیرمسئله باید بررسی شوند. اگر $x_m = y_n$، باید یک بلندترین زیردنباله‌ی مشترک برای $X_{m-1}$ و $Y_{n-1}$ بیابیم. اضافه کردن $x_m = y_n$ به این بلندترین زیردنباله‌ی مشترک، یک بلندترین زیردنباله‌ی مشترک از $X$ و $Y$ را نتیجه می‌دهد. اگر $x_m \neq y_n$، آن گاه باید دو زیرمسئله را حل کنیم: یافتن یک بلندترین زیردنباله‌ی مشترک برای $X_{m-1}$ و $Y$ و یافتن یک بلندترین زیردنباله‌ی مشترک برای $X$ و $Y_{n-1}$. هر کدام از این دو بلندترین زیردنباله‌ی مشترک، بلندتر باشد، یک بلندترین زیردنباله‌ی مشترک برای $X$ و $Y$ است. از آن جایی که این حالت‌ها تمام احتمالات ممکن را پوشش می‌دهند، پس باید از جواب بهینه‌ی یکی از زیرمسئله‌ها در بلندترین زیردنباله‌ی مشترک $X$ و $Y$ استفاده شود.

می‌توانیم به سادگی خصوصیت زیرمسئله‌های مشترک را در مسئله‌ی بلندترین زیردنباله‌ی مشترک ببینیم. برای یافتن یک بلندترین زیردنباله‌ی مشترک برای $X$ و $Y$، ممکن است نیاز داشته باشیم که یک بلندترین زیردنباله‌ی مشترک از $X$ و $Y_{n-1}$ و یک بلندترین زیردنباله‌ی مشترک از $X_{m-1}$ و $Y$ بیابیم. ولی هر یک از این زیرمسئله‌ها شامل زیرمسئله‌ی یافتن یک بلندترین زیردنباله‌ی مشترک از $X_{m-1}$ و $Y_{n-1}$ هستند. همچنین، بسیاری از زیرمسئله‌های دیگر دارای زیرمسئله‌های مشترک هستند.

مانند مسئله‌ی ضرب زنجیره‌ی ماتریس‌ها، راه حل بازگشتی برای مسئله‌ی بلندترین زیردنباله‌ی مشترک شامل یافتن یک رابطه‌ی بازگشتی برای مقدار یک جواب بهینه برای مسئله می‌شود. فرض کنید $c[i,j]$ طول یک بلندترین زیردنباله‌ی مشترک از دنباله‌های $X_i$ و $Y_j$ باشد. اگر $i = 0$ یا $j = 0$، طول یکی از دنباله‌ها ۰ است، بنابراین طول بلندترین زیردنباله‌ی مشترک هم ۰ خواهد بود. زیرساختار بهینه‌ی مسئله‌ی بلندترین زیردنباله‌ی مشترک فرمول بازگشتی زیر را به دست می‌دهد:

$$c[i,j] = \begin{cases} 0 & \text{اگر } i = 0 \text{ یا } j = 0 \\ c[i-1, j-1] + 1 & \text{اگر } i,j > 0 \text{ و } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{اگر } i,j > 0 \text{ و } x_i \neq y_j \end{cases} \qquad (۹-۱۵)$$

مشاهده کنید که در این فرمول‌بندی بازگشتی، یک وضعیت در مسئله، زیرمسئله‌هایی که باید بررسی شوند را محدود می‌کند. وقتی $x_i = y_j$، می‌توانیم و باید زیرمسئله‌ی یافتن یک بلندترین زیردنباله‌ی مشترک از $X_{i-1}$ و $Y_{j-1}$ را حل کنیم. در غیر این صورت، باید دو زیرمسئله‌ی یافتن یک بلندترین زیردنباله‌ی مشترک برای $X_i$ و $Y_{j-1}$ و یافتن یک بلندترین زیردنباله‌ی مشترک برای $X_{i-1}$ و $Y_j$ را حل کنیم. برای مسائل برنامه‌ریزی پویایی که قبلاً بررسی کردیم – برنامه‌ریزی خطوط تولید و ضرب زنجیره‌ی ماتریس‌ها – هیچ کدام از زیرمسئله‌ها به خاطر وضعیت مسئله حذف نمی‌شدند. یافتن بلندترین زیردنباله‌ی مشترک تنها مسئله‌ی برنامه‌ریزی پویایی نیست که در آن زیرمسئله‌ها بر مبنای وضعیت مسئله حذف می‌شوند. به عنوان مثال، مسئله‌ی فاصله‌ی ویرایشی (مسئله‌ی ۱۵-۵ را ببینید) هم این خصوصیت را دارد.

## مرحله‌ی ۳: محاسبه‌ی طول یک بلندترین زیردنباله‌ی مشترک

بر مبنای تساوی (۱۵-۹)، می‌توانیم به سادگی یک الگوریتم بازگشتی با زمان نمایی برای محاسبه‌ی طول یک بلندترین زیردنباله‌ی مشترک برای دو دنباله بنویسیم. با این حال از آن جایی که فقط $\theta(mn)$ زیرمسئله‌ی مجزا وجود دارد، می‌توانیم از برنامه‌ریزی پویا برای محاسبه‌ی جواب به صورت از پایین به بالا استفاده کنیم.

رویه‌ی LCS-LENGTH دو دنباله‌ی $X = \langle x_1, x_2, ..., x_m \rangle$ و $Y = \langle y_1, y_2, ..., y_n \rangle$ را به عنوان ورودی می‌گیرد. این رویه، مقادیر $c[i,j]$ را در جدول $c[0..m, 0..n]$ ذخیره می‌کند، که خانه‌های آن به ترتیب ردیفی پر می‌شوند. (یعنی، ابتدا اولین ردیف $c$ از چپ به راست پر می‌شود، سپس ردیف دوم، و الی آخر.) همچنین جدول $b[1..m, 1..n]$ برای ساده شدن بازسازی جواب بهینه نگه‌داری می‌شود. به صورت شهودی، $b[i,j]$ به خانه‌ای از جدول اشاره می‌کند که حاوی جواب بهینه‌ی زیرمسئله‌ای است که هنگام محاسبه‌ی $c[i,j]$ انتخاب شده است. رویه جدول‌های $b$ و $c$ را بازمی‌گرداند؛ $c[m,n]$ حاوی طول یک بلندترین زیردنباله‌ی مشترک برای $X$ و $Y$ است.

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i - 1, j - 1] + 1
12              b[i, j] = "↖"
13          elseif c[i - 1, j] ≥ c[i, j - 1]
14              c[i, j] = c[i - 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j - 1]
17              b[i, j] = "←"
18  return c and b
```

شکل ۱۵-۸ جدول‌های ساخته شده توسط LCS-LENGTH را بر روی دنباله‌های $X = \langle A, B, C, B, D, A, B \rangle$ و $Y = \langle B, D, C, A, B, A \rangle$ نشان می‌دهد. زمان اجرای رویه $O(mn)$ است، چرا که محاسبه‌ی هر خانه‌ی جدول در زمان $O(1)$ انجام می‌شود.

| $i$ | | $j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|
| | | $y_j$ | | $B$ | $D$ | $C$ | $A$ | $B$ | $A$ |
| 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 ۱۵-۸** جدول‌های $c$ و $b$ محاسبه شده توسط LCS-LENGTH بر روی دنباله‌های $X = \langle A,B,C,B,D,A,B \rangle$ و $Y = \langle B,D,C,A,B,A \rangle$. خانه‌ی ردیف $i$ام و ستون $j$ام حاوی مقدار $c[i,j]$ است، به علاوه‌ی پیکان مناسب برای مقدار $b[i,j]$. ورودی ۴ در خانه‌ی $c[۷,۶]$ گوشه‌ی پایین راست جدول – طول یک بلندترین زیردنباله‌ی مشترک $X$ و $Y$ است. برای $i,j > ۰$، ورودی خانه‌ی $c[i,j]$ فقط به درستی $x_i = y_j$ و مقدار ورودی‌های $c[i-۱,j-۱]$، $c[i,j-۱]$، و $c[i-۱,j]$ بستگی دارد، که همه قبل از $c[i,j]$ محاسبه می‌شوند. برای بازسازی عناصر یک بلندترین زیردنباله‌ی مشترک، پیکان‌های $b[i,j]$ را از گوشه‌ی پایین و راست دنبال کنید؛ مسیر، سایه زده شده است. هر "↖" در مسیر متناظر است با یک ورودی (سایه‌ی روشن) که برای آن $x_i = y_j$ عضوی از یک بلندترین زیردنباله‌ی مشترک است

## مرحله‌ی ۴: ساختن یک بلندترین زیردنباله‌ی مشترک

از جدول $b$ که توسط LCS-LENGTH بازگردانده می‌شود، می‌توان برای ساختن سریع بلندترین زیردنباله‌ی مشترک $X = \langle x_۱,x_۲,...,x_m \rangle$ و $Y = \langle y_۱,y_۲,...,y_n \rangle$ استفاده کرد. به سادگی از $b[m,n]$ شروع می‌کنیم و پیکان‌ها را دنبال می‌کنیم. هر گاه به یک "↖" در ورودی خانه‌ی $b[i,j]$ برخوردیم، بدین معنی است که $x_i = y_j$ یک عنصر از بلندترین زیردنباله‌ی مشترکی است که LCS-LENGTH پیدا کرده است. با این روش، عناصر بلندترین زیردنباله‌ی مشترک با ترتیب برعکس به ما داده می‌شوند. رویه‌ی بازگشتی زیر بلندترین زیردنباله‌ی مشترک $X$ و $Y$ را با ترتیب درست چاپ می‌کند. فراخوانی اولیه PRINT-LCS $(b, X, X.length, Y.length)$ است.

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return
3  if b[i, j] == "↖"
```

```
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6   elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8   else PRINT-LCS(b, X, i, j − 1)
```

برای جدول $b$ در شکل ۱۵-۸، این رویه "$BCBA$" را چاپ می‌کند. زمان اجرای رویه $O(m+n)$ است، چرا که در هر مرحله‌ی بازگشت حداقل یکی از دو متغیر $i$ و $j$ کاهش می‌یابد.

### بهبود کد

معمولاً وقتی که یک الگوریتم را پیاده‌سازی کردید، متوجه می‌شوید که می‌توانید زمان یا حافظه‌ی مصرفی آن را بهبود بخشید. بعضی از تغییرات در کد می‌توانند ضرایب ثابت را کاهش دهند، ولی تغییری در کارایی حدی ایجاد نمی‌کنند. ولی با بعضی تغییرات می‌توان به صورت حدی صرفه جویی‌هایی در زمان و حافظه انجام داد.

به عنوان مثال در الگوریتم LCS، می‌توانیم کل جدول $b$ را حذف کنیم. هر ورودی $c[i,j]$ فقط به سه ورودی دیگر در جدول $c$ بستگی دارد: $c[i-1,j-1]$، $c[i-1,j]$، و $c[i,j-1]$. با داشتن مقدار $c[i,j]$، می‌توانیم در زمان $O(1)$ تعیین کنیم که از کدام یک از این سه مقدار برای ساختن $c[i,j]$ استفاده شده است، بدون این که به جدول $b$ احتیاج داشته باشیم. بنابراین، می‌توانیم یک بلندترین زیردنباله‌ی مشترک را در زمان $O(m+n)$ با استفاده از یک رویه مانند PRINT-LCS بسازیم. (تمرین ۱۵-۴-۲ از شما می‌خواهد که این رویه را بنویسید.) با این که می‌توانیم با این روش به اندازه‌ی $\theta(mn)$ در حافظه صرفه جویی کنیم، حافظه‌ی کمکی مورد نیاز برای محاسبه‌ی بلندترین زیردنباله‌ی مشترک به صورت حدی تغییر نمی‌کند، چرا که به هر حال به اندازه‌ی $\theta(mn)$ برای جدول $c$ به حافظه نیاز داریم.

با این حال، می‌توانیم حافظه‌ی مورد نیاز را برای LCS-LENGTH به صورت حدی کاهش دهیم، چرا که این رویه در هر زمان فقط به دو ردیف از جدول $c$ نیاز دارد: ردیف در حال محاسبه شدن، و ردیف قبلی. (در واقع، همان طور که تمرین ۱۵-۴-۴ از شما می‌خواهد نشان دهید، فقط می‌توانیم با مقدار کمی بیش از یک ردیف از $c$ طول یک بلندترین زیردنباله‌ی مشترک را محاسبه کنیم.) این بهبود در صورتی کاربردی است که ما فقط بخواهیم طول یک بلندترین زیردنباله‌ی مشترک را بازسازی کنیم، جدول کوچک‌تر حاوی اطلاعات مورد نیاز برای دنبال کردن مراحل در زمان $O(m+n)$ نیست.

## تمرین‌ها

۱-۴-۱۵ یک بلندترین زیردنباله‌ی مشترک برای $\langle 1,0,0,1,0,1,0,1\rangle$ و $\langle 0,1,0,1,1,0,1,1,0\rangle$ تعیین کنید.

۲-۴-۱۵ شبه‌کدی ارائه کنید که از جدول کامل شده‌ی $c$ و دنباله‌های $X=\langle x_1,x_2,\ldots,x_m\rangle$ و

$\langle y_1, y_2, \ldots, y_n \rangle$ $Y$، در زمان $O(m+n)$ و بدون استفاده از جدول $b$، بلندترین زیردنباله‌ی مشترک دو دنباله را بازسازی می‌کند.

۱۵-۴-۳ با استفاده به خاطر سپاری، یک نسخه از LCS-LENGTH بدهید که در زمان $O(mn)$ اجرا شود.

۱۵-۴-۴ نشان دهید که چگونه می‌توان فقط با استفاده از $2 \times \min(m,n)$ ورودی در جدول $c$ به علاوه‌ی $O(1)$ خانه‌ی اضافی در حافظه، طول یک بلندتری زیردنباله‌ی مشترک را محاسبه کرد. سپس نشان دهید که چگونه می‌توان این کار را با استفاده از $\min(m,n)$ خانه به علاوه‌ی $O(1)$ حافظه‌ی اضافی انجام داد.

۱۵-۴-۵ یک الگوریتم با زمان $O(n^2)$ برای محاسبه‌ی بلندتری زیردنباله‌ی صعودی اکید از یک دنباله شامل $n$ عدد ارائه کنید.

★ ۱۵-۴-۶ یک الگوریتم با زمان $O(n \lg n)$ برای محاسبه‌ی بلندتری زیردنباله‌ی صعودی اکید از یک دنباله شامل $n$ عدد ارائه کنید. (*راهنمایی:* مشاهده کنید که آخرین عنصر یک زیردنباله‌ی کاندیدا با طول $i$ حداقل به بزرگی عنصر آخر یک زیردنباله‌ی کاندیدا با طول $i-1$ است. زیردنباله‌های کاندیدا را با استفاده از متصل کردن آن‌ها در طول دنباله‌ی ورودی نگه دارید.)

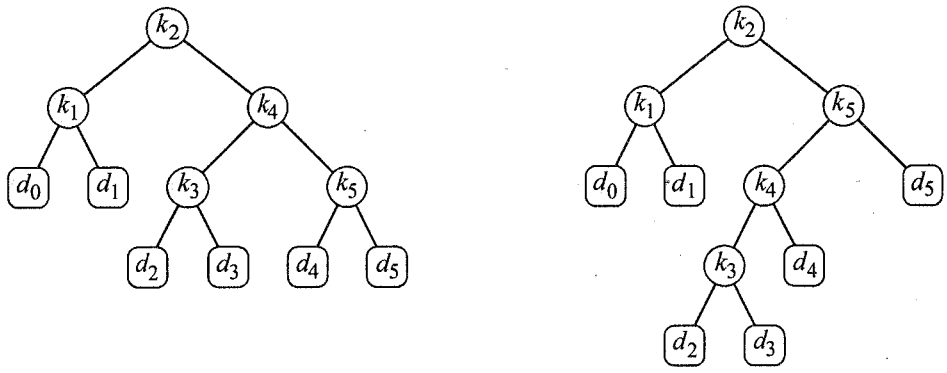
## ۱۵-۵ درخت‌های جستجوی دودویی بهینه

فرض کنید که یک برنامه برای ترجمه‌ی متون از انگلیسی به فرانسه طراحی کرده‌ایم. برای هر بار استفاده از هر کلمه‌ی انگلیسی در متن، نیاز داریم که معادل فرانسوی آن را در دیکشنری بیابیم. یک روش برای انجام این اعمال جستجو، ساختن یک درخت جستجوی دودویی با $n$ کلمه‌ی انگلیسی به عنوان کلید و کلمات فرانسوی معادل به عنوان داده‌های پیرو است. چون ما برای هر کلمه‌ی انگلیسی باید در درخت جستجو کنیم، می‌خواهیم که کل زمان مصرف شده برای جستجو کمینه شود. می‌توانیم با استفاده از یک درخت قرمز–سیاه، و یا هر درخت جستجوی دودویی متوازن دیگر، زمان اجرای $O(\lg n)$ را تضمین کنیم. با این حال، فراوانی کلمات مختلف در متن با یکدیگر متفاوت است، و ممکن است حالتی پیش بیاید که کلمه‌ی پر کاربرد "the" در جایی بسیار دور از ریشه قرار گیرد، و کلمه‌ی کم کاربردی مانند "mycophagist" نزدیک ریشه باشد. چنین آرایشی ترجمه را کند خواهد کرد، چرا که تعداد گره‌های ملاقات شده برای جستجوی یک کلید در یک درخت جستجوی دودویی برابر است با یک به علاوه‌ی عمق گره‌ی حاوی کلید. می‌خواهیم کلمات پر کاربرد نزدیک ریشه باشند.[1] به علاوه، ممکن است کلماتی در متن باشد که هیچ معادل فرانسوی برای آن‌ها وجود نداشته باشد، و

[1] اگر موضوع متن قارچ‌های خوراکی باشد، ممکن است بخواهیم "mycophagist" نزدیک ریشه باشد.

چنین کلمه‌هایی اصلاً نباید در درخت جستجوی دودویی ظاهر شوند. با داشتن فراوانی کلمه‌ها در یک متن، چگونه می‌توان یک درخت جستجوی دودویی را شکل داد به طوری که گره‌های ملاقات شده در تمام جستجوها کمینه شود؟

چیزی که به آن نیاز داریم، *درخت جستجوی دودویی بهینه* (optimal binary search tree) نام دارد. به صورت فرمال، یک دنباله‌ی $K = \langle k_1, k_2, ..., k_n \rangle$ از $n$ کلید مجزا به صورت مرتب شده (به طوری که $k_1 < k_2 < ... < k_n$) به ما داده شده است، و می‌خواهیم یک درخت جستجوی دودویی از این کلیدها بسازیم. برای هر کلید $k_i$، یک احتمال $p_i$ داریم که احتمال جستجو برای $k_i$ را نشان می‌دهد. ممکن است بعضی جستجوها برای مقادیری انجام شود که در $K$ موجود نباشند، و بنابراین $n+1$ «کلید زائد» $d_0, d_1, d_2, ..., d_n$ داریم که نشان‌دهنده‌ی مقادیری هستند که در $K$ وجود ندارند. به طور خاص، کلید زائد $d_0$ نشان‌دهنده‌ی تمام مقادیر کوچک‌تر از $k_1$، کلید زائد $d_n$ نشان‌دهنده‌ی تمام مقادیر بزرگ‌تر از $k_n$، و برای $i = 1, 2, ..., n-1$، کلید زائد $d_i$ نشان‌دهنده‌ی تمام مقادیر بین $k_i$ و $k_{i+1}$ است. برای هر کلید زائد $d_i$، یک احتمال $q_i$ برای جستجوی $d_i$ داریم. شکل ۱۵-۹ دو درخت جستجو دودویی برای مجموعه‌ای از ۵ کلید را نشان می‌دهد. هر کلید $k_i$ یک گره‌ی داخلی، و هر کلید زائد $d_i$ یک برگ است. هر جستجو یا موفق است (یافتن یک کلید $k_i$) و یا ناموفق (یافتن یک کلید زائد $d_i$)، و بنابراین داریم




**شکل ۱۵-۹** دو درخت جستجوی دودویی برای یک مجموعه از $n$ کلید با احتمالات زیر:

| $i$ | ۰ | ۱ | ۲ | ۳ | ۴ | ۵ |
|---|---|---|---|---|---|---|
| $p_i$ | | ۰٫۱۵ | ۰٫۱۰ | ۰٫۰۵ | ۰٫۱۰ | ۰٫۲۰ |
| $p_i$ | ۰٫۰۵ | ۰٫۱۰ | ۰٫۰۵ | ۰٫۰۵ | ۰٫۰۵ | ۰٫۱۰ |

(الف) یک درخت جستجوی دودویی با امیدریاضی هزینه‌ی جستجوی ۲٫۸۰. (ب) یک درخت جستجوی دودویی با امیدریاضی هزینه‌ی جستجوی ۲٫۷۵. این درخت بهینه است.

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 \qquad (۱۵-۱۰)$$

از آن جایی که ما احتمال جستجو برای هر کلید و هر کلید زائد را داریم، می‌توانیم امیدریاضی هزینه‌ی جستجو را در یک درخت جستجوی دودویی داده شده‌ی $T$ به دست آوریم. اجازه دهید فرض کنیم که هزینه‌ی واقعی یک جستجو برابر است با تعداد گره‌های بررسی شده، یعنی عمق گره‌ای که با جستجو در $T$ پیدا شده است، به علاوه‌ی ۱. در این صورت امیدریاضی هزینه‌ی یک جستجو در درخت $T$ برابر است با

$$\begin{aligned} E[\text{هزینه‌ی جستجو در } T] &= \sum_{i=1}^{n} (depth_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n} (depth_T(d_i)+1)\cdot q_i \\ &= 1 + \sum_{i=1}^{n} depth_T(k_i)\cdot p_i + \sum_{i=0}^{n} depth_T(d_i)\cdot q_i \end{aligned} \qquad (۱۵-۱۱)$$

که در آن $depth_T$ نشان‌دهنده‌ی عمق یک گره در درخت $T$ است. تساوی آخر از تساوی (۱۵-۱۱) نتیجه می‌شود. در شکل ۱۵-۹(الف)، می‌توانیم امیدریاضی هزینه‌ی جستجو را برای تک تک گره‌ها محاسبه کنیم:

| گره | عمق | احتمال | سهم |
|---|---|---|---|
| $k_1$ | ۱ | ۰/۱۵ | ۰/۳۰ |
| $k_2$ | ۰ | ۰/۱۵ | ۰/۱۰ |
| $k_3$ | ۲ | ۰/۱۵ | ۰/۱۵ |
| $k_4$ | ۲ | ۰/۱۵ | ۰/۲۰ |
| $k_5$ | ۲ | ۰/۱۵ | ۰/۶۰ |
| $d_0$ | ۲ | ۰/۱۵ | ۰/۱۵ |
| $d_1$ | ۲ | ۰/۱۵ | ۰/۳۰ |
| $d_2$ | ۳ | ۰/۱۵ | ۰/۲۰ |
| $d_3$ | ۳ | ۰/۱۵ | ۰/۲۰ |
| $d_4$ | ۳ | ۰/۱۵ | ۰/۲۰ |
| $d_5$ | ۳ | ۰/۱۵ | ۰/۴۰ |
| مجموع | | | ۲/۸۰ |

برای یک مجموعه‌ی داده شده از احتمالات، هدف ما این است که یک درخت جستجوی دودویی بسازیم که امیدریاضی هزینه‌ی جستجو در آن کمینه باشد. به چنین درختی، یک *درخت جستجوی دودویی بهینه* می‌گوییم. شکل ۱۵-۷(ب) یک درخت جستجوی دودویی بهینه را برای احتمالات داده شده در عنوان شکل نشان می‌دهد؛ امیدریاضی هزینه‌ی آن ۲/۷۵ است. این مثال نشان می‌دهد که یک درخت جستجوی دودویی بهینه لزوماً درختی نیست که ارتفاع کلی آن کمینه باشد. همچنین نمی‌توانیم همیشه با قرار دادن گره‌ی با بیشترین احتمال در ریشه، یک درخت جستجوی دودویی بهینه بسازیم. در این جا، کلید $k_5$ دارای بیشترین احتمال وقوع در میان تمام کلیدها است، با این حال ریشه‌ی درخت جستجوی دودویی بهینه‌ی نشان داده شده $k_2$ است. (کم‌ترین امیدریاضی هزینه برای یک

درخت جستجوی دودویی که ریشه‌ی آن $k_5$ باشد، ۲٫۸۵ است.)

مانند ضرب زنجیره‌ی ماتریس‌ها، چک کردن تمام حالت‌های ممکن به ما الگوریتم کارایی نمی‌دهد. می‌توانیم گره‌های هر درخت جستجوی دودویی با $n$ گره با کلیدهای $k_1,k_2,...,k_n$ را برچسب‌دهی کنیم تا یک درخت جستجوی دودوی بسازیم، و سپس کلیدهای زائد را به صورت برگ به درخت اضافه کنیم. در مسئله‌ی ۱۲-۴ دیدیم که تعداد درخت‌های جستجوی دودویی با $n$ گره برابر است با $\Omega(4^n/n^{3/2})$، و بنابراین تعداد درخت‌هایی که باید بررسی کنیم، نمایی است. همان طور که حدس زده‌اید، این مسئله را با استفاده از برنامه‌ریزی پویا حل خواهیم کرد.

## مرحله‌ی ۱: ساختار یک درخت جستجوی دودویی بهینه

برای توصیف زیرساختار بهینه‌ی درخت‌های جستجوی دودویی بهینه، با یک مشاهده بر روی زیردرخت‌ها شروع می‌کنیم. یک زیردرخت از یک درخت جستجوی دودویی را در نظر بگیرید. این درخت باید حاوی کلیدهایی در دامنه‌ی پیوسته‌ی $k_i,...,k_j$ باشد، برای $1 \le i \le j \le n$. به علاوه، یک زیردرخت که حاوی کلیدهای $k_i,...,k_j$ است همچنین باید برگ‌های آن کلیدهای زائد $d_{i-1},...,d_j$ باشند.

اکنون می‌توانیم زیرساختار اصلی را مشخص کنیم: اگر یک درخت جستجوی دودویی بهینه‌ی $T$ شامل یک زیردرخت $T'$ حاوی کلیدهای $k_i,...,k_j$ باشد، آن گاه زیردرخت $T'$ باید برای زیرمسئله‌ی با کلیدهای $k_i,...,k_j$ و کلیدهای زائد $d_{i-1},...,d_j$ بهینه باشد. برای اثبات می‌توان از بحث برش و چسباندن همیشگی استفاده کرد. اگر یک زیردرخت $T''$ وجود داشته باشد که امیدریاضی هزینه‌ی آن کم‌تر از $T'$ باشد، آن گاه می‌توانیم $T'$ را بریده و $T''$ را به جای آن قرار دهیم، و نتیجه‌ی آن یک درخت جستجوی دودویی با امیدریاضی هزینه‌ی کم‌تر از $T$ است، که با بهینه بودن $T$ تناقض دارد.

باید از زیرساختار بهینه استفاده کنیم تا نشان دهیم که می‌توان با استفاده از جواب‌های بهینه‌ی زیرمسئله‌ها، یک جواب بهینه برای کل مسئله ساخت. با داشتن کلیدهای $k_i,...,k_j$، یکی از این کلیدها، مثلاً $k_r$ ($i \le r \le j$)، باید ریشه‌ی زیردرخت بهینه برای این کلیدها باشد. زیردرخت سمت چپ حاوی کلیدهای $k_i,...,k_{r-1}$ (و کلیدهای زائد $d_{i-1},...,d_{r-1}$)، و درخت سمت راست حاوی کلیدهای $k_{r+1},...,k_j$ (و کلیدهای زائد $d_r,...,d_j$) خواهد بود. اگر تمام ریشه‌های کاندیدای $k_r$ را، که $i \le r \le j$، بررسی کنیم، و تمام درخت‌های جستجوی دودویی بهینه حاوی $k_i,...,k_{r-1}$ و همچنین حاوی $k_{r+1},...,k_j$ را تعیین کنیم، می‌توانیم تضمین کنیم که یک درخت جستجوی دودویی بهینه برای درخت پیدا خواهیم کرد.

یک نکته در مورد زیردرخت‌های «تهی» وجود دارد. فرض کنید که در زیردرختی با کلیدهای $k_i,...,k_j$، کلید $k_i$ را به عنوان ریشه انتخاب می‌کنیم. طبق بحث بالا، زیردرخت سمت چپ $k_i$ حاوی کلیدهای $k_i,...,k_{i-1}$ است. طبیعی است که این دنباله را دنباله‌ی خالی در نظر بگیریم. با این حال، به خاطر داشته باشید که زیردرخت‌ها همچنین حاوی کلیدهای زائد هستند. ما این قرار داد را خواهیم داشت که زیردرخت‌های حاوی کلیدهای $k_i,...,k_{i-1}$، هیچ کلید واقعی ندارند، ولی حاوی یک کلید

زائد $d_{i-1}$ هستند. به طور مشابه اگر $k_j$ را به عنوان ریشه انتخاب کنیم، در این صورت زیردرخت سمت راست حاوی کلیدهای $k_{j+1},...,k_j$ خواهد بود؛ این زیردرخت حاوی هیچ کلید واقعی نیست، ولی کلید زائد $d_j$ در آن واقع شده است.

## مرحله‌ی ۲: یک راه حل بازگشتی

اکنون آماده هستیم که مقدار یک جواب بهینه را به صورت بازگشتی تعیین کنیم. دامنه‌ی زیرمسئله‌ی خود را به صورت یافتن یک درخت جستجوی دودویی بهینه حاوی کلیدهای $k_i,...,k_j$ تعریف می‌کنیم، که در آن $i \geq 1$، $j \leq n$، و $j \geq i-1$. (وقتی $j = i-1$ هیچ کلید واقعی در درخت وجود ندارد؛ تنها کلید موجود در درخت، کلید زائد $d_{i-1}$ است.) تعریف می‌کنیم $e[i,j]$ برابر است با امیدریاضی هزینه‌ی جستجو در درخت جستجوی دودویی بهینه حاوی کلیدهای $k_i,...,k_j$. در نهایت، می‌خواهیم $e[1,n]$ را محاسبه کنیم.

حالت آسان زمانی پیش می‌آید که $j = i-1$. در این صورت فقط کلید زائد $d_{i-1}$ را داریم، که امیدریاضی هزینه‌ی جستجو $e[i, i-1] = q_{i-1}$ است.

وقتی $j \geq i$، باید یک ریشه‌ی $k_r$ از میان کلیدهای $k_i,...,k_j$ انتخاب کنیم و سپس یک درخت جستجوی دودویی بهینه با کلیدهای $k_i,...,k_{r-1}$ به عنوان زیردرخت سمت چپ، و یک درخت جستجوی دودویی بهینه با کلیدهای $k_{r+1},...,k_j$ به عنوان زیردرخت سمت راست بسازیم. وقتی یک درخت، تبدیل می‌شود به زیردرخت یک گره، امیدریاضی هزینه‌ی جستجوی آن چه تغییری می‌کند؟ عمق هر گره در زیردرخت یکی افزایش می‌یابد. طبق تساوی (۱۵-۱۱)، امیدریاضی هزینه‌ی جستجوی این زیردرخت به اندازه‌ی مجموع تمام احتمالات در زیردرخت افزایش می‌یابد. اجازه دهید برای یک زیردرخت با کلیدهای $k_i,...,k_j$، مجموع احتمالات را به صورت زیر نشان دهیم:

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} p_l \qquad (۱۲-۱۵)$$

بنابراین، اگر $k_r$ ریشه‌ی درخت جستجوی دودویی بهینه حاوی کلیدهای $k_i,...,k_j$ 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 این که

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j)$$

$e[i,j]$ را به صورت زیر بازنویسی می‌کنیم

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j) \qquad (۱۳-۱۵)$$

تساوی بازگشتی (۱۵-۱۳) فرض می‌کند که ما می‌دانیم از کدام کلید $k_r$ به عنوان ریشه استفاده کنیم. کلیدی را به عنوان ریشه انتخاب می‌کنیم که کم‌ترین امیدریاضی هزینه‌ی جستجو را به ما بدهد، که فرمول بازگشتی نهایی زیر را به ما می‌دهد:

$$e[i,j]=\begin{cases} q_{i-1} & \text{اگر } j=i-1 \\ \min\limits_{i\le r\le j}\{e[i,r-1]+e[r\_1,j]+w(i,j)\} & \text{اگر } i\le j \end{cases} \qquad (۱۴-۱۵)$$

مقادیر $e[i,j]$ امیدریاضی هزینه‌ی جستجو را در درخت‌های جستجوی دودویی بهینه به ما می‌دهند. برای این که بتوانیم ساختار درخت‌های جستجوی دودویی بهینه را نگه داریم، برای $1\le i\le j\le n$، مقدار $root[i,j]$ را به صورت اندیس $r$ تعریف می‌کنیم که $k_r$ ریشه‌ی یک درخت جستجوی دودویی بهینه حاوی کلیدهای $k_i,...,k_j$ است. با این که بعداً خواهیم دید که چگونه مقادیر $root[i,j]$ را محاسبه کنیم، ساختن درخت جستجوی دودویی بهینه از روی این مقادیر را به عنوان تمرین به شما واگذار می‌کنیم.

## مرحله‌ی ۳: محاسبه‌ی امیدریاضی هزینه‌ی جستجوی یک درخت جستجوی دودویی بهینه

در این جا، ممکن است متوجه بعضی شباهت‌ها میان توصیف درخت جستجوی دودویی بهینه و ضرب زنجیره‌ی ماتریس‌ها شده باشید. برای هر دو مسئله، زیرمسئله‌ها حاوی دامنه‌های پیوسته‌ای از اندیس‌ها هستند. یک پیاده‌سازی بازگشتی مستقیم از روی تساوی (۱۵-۱۳) ناکارامد خواهد بود، همان طور که پیاده‌سازی بازگشتی مستقیم برای ضرب زنجیره‌ی ماتریس‌ها ناکارامد بود. 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، چرا که برای داشتن یک زیردرخت فقط شامل کلید زائد $d_n$، باید مقدار $e[n+1,n]$ را محاسبه و ذخیره کنیم. اندیس دوم باید از ۰ شروع شود چرا که برای این که یک زیردرخت فقط حاوی کلید زائد $d_0$ داشته باشیم، باید $e[1,0]$ را محاسبه و ذخیره کنیم. فقط از خانه‌هایی از جدول $e[i,j]$ که در آن‌ها $j\ge i-1$ است استفاده می‌کنیم. همچنین از یک جدول $root[i,j]$ برای ذخیره‌ی اندیس ریشه‌ی زیردرخت شامل کلیدهای $k_i,...,k_j$ استفاده می‌کنیم. این جدول فقط از خانه‌هایی استفاده می‌کند که در آن‌ها $1\le i\le j\le n$.

برای کارایی بالاتر، به یک جدول دیگر نیز نیاز داریم. به جای این که مقدار $w(i,j)$ را برای محاسبه‌ی $e[i,j]$ هر بار از اول محاسبه کنیم — که به $\theta(j-i)$ زمان اضافی نیاز دارد — این مقادیر را در جدول $w[1..n+1,0..n]$ ذخیره می‌کنیم. برای حالت پایه، برای $1\le i\le n$ مقدار $w[i,i-1]=q_{i-1}$ را محاسبه می‌کنیم. برای $j\ge i$ این مقادیر را به صورت زیر محاسبه می‌کنیم:

$$w[i,j]=w[i,j-1]+p_j+q_j \qquad (۱۵-۱۵)$$

بنابراین، هر کدام از $\theta(n^2)$ مقدار $w[i,j]$ را در زمان $\theta(1)$ محاسبه می‌کنیم.

شبه‌کد زیر به عنوان ورودی، احتمالات $p_1,...,p_n$ و $q_0,...,q_n$ و اندازه‌ی $n$ را می‌گیرد، و جدول‌های $e$ و $root$ را بازمی‌گرداند.

```
OPTIMAL-BST(p, q, n)
1  let e[1..n+1, 0..n], w[1..n+1, 0..n],
       And root[1..n, 1..n] be new tables
2  for i = 1 to n + 1
```

```
3       e[i, i - 1] = q_{i-1}
4       w[i, i - 1] = q_{i-1}
5   for l = 1 to n
6       for i = 1 to n - l + 1
7           j = i + l - 1
8           e[i, j] = ∞
9           w[i, j] = w[i, j - 1] + p_j + q_j
10          for r = i to j
11              t = e[i, r - 1] + e[r + 1, j] + w[i, j]
12              if t < e[i, j]
13                  e[i, j] = t
14                  root[i, j] = r
15  return e and root
```

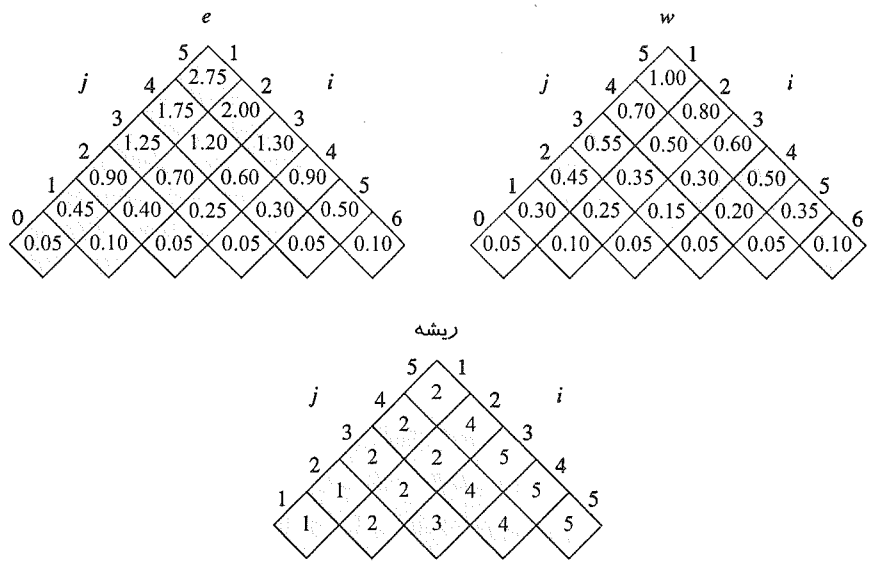
طبق توضیحات بالا و شباهت با رویه‌ی MATRIX-CHAIN-ORDER از بخش ۱۵-۲، کارکرد این رویه باید نسبتا ساده باشد. حلقه‌ی for خطوط ۲-۴ مقادیر $e[i,i-1]$ و $w[i,i-1]$ را مقداردهی اولیه می‌کند. سپس حلقه‌ی for خطوط ۵-۱۴ با استفاده از روابط بازگشتی (۱۵-۱۴) و (۱۵-۱۵)، برای $1 \le i \le j \le n$ مقادیر $e[i,j]$ و $w[i,j]$ را محاسبه می‌کند. در تکرار اول وقتی $l = 1$، حلقه $e[i,i]$ 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. داخلی‌ترین حلقه‌ی for در خطوط ۱۰-۱۴، هر یک از اندیس‌های کاندیدای $r$ را برای تعیین کلید $k_r$ به عنوان ریشه‌ی درخت جستجوی دودویی بهینه با کلیدهای $k_i,...,k_j$ امتحان می‌کند. این حلقه‌ی for هر بار که یک کلید بهتر برای ریشه یافت، مقدار جدید اندیس $r$ را در $root[i,j]$ ذخیره می‌کند.

شکل ۱۵-۱۰ جداول $e[i,j]$، $w[i,j]$، و $root[i,j]$ را که رویه‌ی OPTIMAL-BST برای کلیدهای شکل ۱۵-۹ محاسبه کرده، نشان می‌دهد. مانند مثال ضرب زنجیره‌ی ماتریس‌ها در شکل ۱۵-۵، جداول طوری دوران یافته‌اند که قطرها به صورت افقی قرار گیرند. OPTIMAL-BST ردیف‌ها را از پایین به بالا، و در هر ردیف از چپ به راست محاسبه می‌کند.

رویه‌ی OPTIMAL-BSTS در زمان $\theta(n^3)$ اجرا می‌شود، درست مانند MATRIX-CHAIN-ORDER. به سادگی می‌توان این زمان اجرا را چک کرد، چرا که حلقه‌های for در این رویه به صورت تودرتو و با عمق سه هستند، و هر اندیس حلقه حداکثر $n$ مقدار به خود می‌گیرد. مقادیر مرزی اندیس حلقه‌ها در OPTIMAL-BST دقیقاً مانند MATRIX-CHAIN-ORDER نیست، ولی تفاوت آن‌ها در تمام جهت‌ها حداکثر ۱ است. بنابراین مانند MATRIX-CHAIN-ORDER، رویه‌ی OPTIMAL-BST در زمان $\Omega(n^3)$ اجرا می‌شود.

## تمرین‌ها

۱-۵-۱۵ برای رویه‌ی CONSTRUCT-OPTIMAL-BST(*root*) شبه‌کد بنویسید، که این رویه با دریافت جدول *root*، ساختار یک درخت جستجوی دودویی بهینه را چاپ می‌کند. به عنوان مثال در شکل ۱۵-۱۰، رویه‌ی شما باید ساختار زیر را چاپ کند:




**شکل ۱۵-۱۰** جداول $e[i,j]$، $w[i,j]$، و $root[i,j]$ که رویه‌ی OPTIMAL-BST بر روی کلیدهای نشان داده شده در شکل ۱۵-۹ محاسبه کرده است. جداول طوری دوران یافته‌اند که قطرها به صورت افقی قرار گیرند.

$k_2$ 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 متناظر است با درخت جستجو دودویی بهینه‌ی نشان داده شده در شکل ۱۵-۹(ب).

۱۵-۵-۲ هزینه و ساختار یک درخت جستجوی دودویی بهینه با مجموعه‌ای از $n = ۷$ کلید با احتمالات زیر را تعیین کنید:

| $i$ | ۰ | ۱ | ۲ | ۳ | ۴ | ۵ | ۶ | ۷ |
|---|---|---|---|---|---|---|---|---|
| $p_i$ | | ۰٫۰۴ | ۰٫۰۶ | ۰٫۰۸ | ۰٫۰۲ | ۰٫۱۰ | ۰٫۱۲ | ۰٫۱۴ |
| $q_i$ | ۰٫۰۶ | ۰٫۰۶ | ۰٫۰۶ | ۰٫۰۶ | ۰٫۰۵ | ۰٫۰۵ | ۰٫۰۵ | ۰٫۰۵ |

۱۵-۵-۳ فرض کنید که به جای نگه داری جدول $w[i,j]$، در خط ۸ رویه‌ی OPTIMAL-BST مقدار

$w(i,j)$ را مستقیماً از تساوی (۱۵-۱۲) محاسبه می‌کنیم، و از این مقادیر در خط ۱۰ استفاده می‌کنیم. این تغییر چه تأثیری بر روی زمان اجرای حدی OPTIMAL-BST می‌گذارد؟

★ ۱۵-۵-۴ Knuth نشان داده است که همیشه ریشه‌هایی از زیردرختان بهینه وجود دارند به طوری که $root[i,j-1] \le root[i,j] \le root[i+1,j]$ برای تمام $1 \le i < j \le n$. با استفاده از این قضیه OPTIMAL-BST را طوری اصلاح کنید که زمان اجرای آن به $\theta(n^2)$ کاهش یابد.

## مسائل

### ۱۵-۱ *طولانی‌ترین مسیر ساده در یک گراف جهت‌دار بدون دور*

فرض کنید به ما یک گراف جهت‌دار بدون دور $G=(V,E)$ داده شده است، با یال‌هایی با وزن حقیقی و دو رأس خاص $s$ و $t$. یک رویکرد برنامه‌نویسی پویا برای یافتن طولانی‌ترین مسیر ساده‌ی وزن‌دار از $s$ به $t$ ارائه کنید. گراف زیرمسئله به چه شکلی است؟ کارایی برنامه‌ی شما چقدر است؟

### ۱۵-۲ *طولانی‌ترین زیردنباله‌ی پالیندرام*

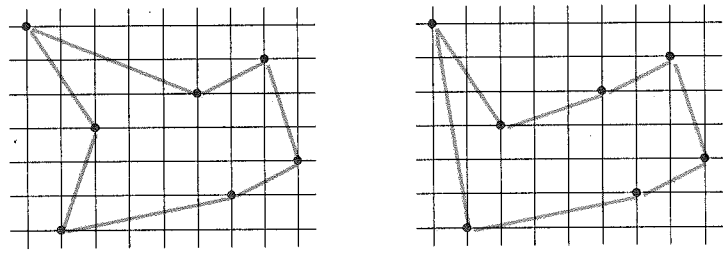
یک *پالیندرام* (palindrome) عبارت است از یک رشته‌ی ناتهی از یک الفبا که از هر دو طرف به یک صورت خوانده می‌شود. مثال‌هایی از پالیندرام‌ها عبارتند از تمام رشته‌های به طول ۱، civic، racecar، و aibohphobia.

یک الگوریتم بهینه برای یافتن طولانی‌ترین پالیندرام ارائه کنید که زیردنباله‌ای از یک رشته‌ی داده شده باشد. مثلاً، برای ورودی character، الگوریتم شما باید carac را بازگرداند. زمان اجرای الگوریتم شما چقدر است؟

### ۱۵-۳ *مسئله‌ی فروشنده‌ی دوره‌گرد اقلیدسی دو جهته*

مسئله‌ی فروشنده‌ی دوره‌گرد اقلیدسی، مسئله‌ی یافتن کوچک‌ترین دور بسته است که مجموعه‌ای از $n$ نقطه در صفحه را به هم متصل می‌کند. شکل ۱۵-۱۱(الف) جواب یک نمونه‌ی ۷ نقطه‌ای را نشان می‌دهد. مسئله‌ی کلی NP-کامل است، و باور بر این است که برای حل آن به بیش از زمان چند جمله‌ای نیاز است (فصل ۳۴ را ببینید).

J. L. Bentley پیشنهاد داده است که برای سادگی، مسئله‌ی خود را به *دورهای دو جهته* (bitonic tours) محدود کنیم، یعنی دورهایی که از چپ‌ترین نقطه آغاز می‌شوند، اکیدا به سمت راست حرکت می‌کنند تا به راست‌ترین نقطه برسند، و سپس با حرکت اکید به سمت چپ به نقطه‌ی اول بازمی‌گردند.



(الف) (ب)

**شکل ۱۵-۱۱** هفت نقطه که در صفحه‌ی مشبک نشان داده شده‌اند. (الف) کوتاه‌ترین مسیر بسته، با طول تقریباً ۲۴٫۸۹. این دور دو جهته نیست. (ب) کوتاه‌ترین دور دو جهته برای همان مجموعه از نقاط. طول این دور تقریباً ۲۵٫۵۸ است.

شکل ۱۵-۱۱(ب) کوتاه‌ترین دور دو جهته را برای همان مجموعه از ۷ نقطه نشان می‌دهد. در این حالت، طراحی یک الگوریتم با زمان چند جمله‌ای امکان‌پذیر است.

یک الگوریتم با زمان $O(n^2)$ برای تعیین یک دور دو جهته‌ی بهینه ارائه کنید. می‌توانید فرض کنید که هیچ دو نقطه‌ای دارای مختصات $x$ یکسانی نیستند. (*راهنمایی:* از چپ به راست صفحه را پویش کنید، و حالت‌های بهینه‌ی ممکن را برای دو قسمت دور ذخیره کنید.)

## ۱۵-۲ چاپ تمیز

مسئله‌ی چاپ کردن یک پارگراف در یک چاپگر با فونت تک‌فاصله (یعنی تمام کاراکترها عرض برابر دارند) را به صورت تمیز در نظر بگیرید. متن ورودی دنباله‌ای از $n$ کلمه با طول‌های $l_1, l_2, \ldots, l_n$، با واحد کاراکتر است. می‌خواهیم این پاراگراف را به صورت تمیز در تعدادی خط که هر یک به اندازه‌ی حداکثر $M$ کاراکتر جا دارند، چاپ کنیم. تعریف ما از «تمیزی» به صورت زیر است. اگر یک خط حاوی کلمه‌های $i$ تا $j$ باشد، که در آن $i \le j$، و دقیقاً یک فاصله بین کلمه‌ها داشته باشیم، در این صورت تعداد کاراکترهای فاصله‌ی اضافی در پایان خط برابر است با $M - j + i - \sum_{k=i}^{j} l_k$، که باید نامنفی باشد تا تمام کلمه‌ها در خط جا شوند. می‌خواهیم مجموع مربع فاصله‌های اضافی در آخر خطوط، در تمام خط‌ها به غیر از خط آخر کمینه شود. یک الگوریتم برنامه‌ریزی پویا بدهید که پاراگرافی از $n$ کلمه را به صورت تمیز در یک چاپگر چاپ می‌کند. زمان اجرا و حافظه‌ی مورد نیاز الگوریتم خود را تحلیل کنید.

## ۱۵-۳ فاصله‌ی ویرایشی

برای تبدیل یک رشته‌ی ورودی $x[1..m]$ به یک رشته‌ی نهایی $y[1..n]$، می‌توانیم اعمال تبدیلی مختلفی انجام دهیم. هدف ما این است که، با دریافت $x$ و $y$، یک سری از تبدیلات انجام دهیم تا $x$ به $y$ تبدیل شود. از یک آرایه‌ی $z$ — که فرض می‌شود به اندازه‌ی کافی برای نگه

داری تمام کاراکترهای مورد نیاز بزرگ است — برای ذخیره‌ی نتایج میانی استفاده می‌کنیم. در ابتدا $z$ تهی است، و در پایان باید برای $j = ۱,۲,...,n$ داشته باشیم $z[j] = y[j]$. اندیس‌های کنونی $i$ را برای $x$ و $j$ را برای $z$ نگه می‌داریم، و اعمال مختلف حق دارند $z$ و این اندیس‌ها را تغییر دهند. در ابتدا، $i = j = ۱$. باید در حین تبدیل، تمام کاراکترهای $x$ را بررسی کنیم، که بدین معنی است که در پایان این دنباله از عملیات تبدیل، باید داشته باشیم $i = m + ۱$. اعمال تبدیل به صورت زیر هستند:

- کپی (copy) یک کاراکتر از $x$ به $z$ با قرار دادن $z[j] = x[i]$ و سپس افزایش هر دو اندیس $i$ و $j$. این عملیات $x[i]$ را بررسی می‌کند.
- جایگزینی (replace) یک کاراکتر از $x$ با کاراکتری دیگر مانند $c$، با قرار دادن $z[j] = c$، و سپس افزایش هر دو اندیس $i$ و $j$. این عملیات $x[i]$ را بررسی می‌کند.
- حذف (delete) یک کاراکتر از $x$ با افزایش $i$ و بدون تغییر باقی گذاشتن $j$. این عملیات $x[i]$ را بررسی می‌کند.
- درج (insert) یک کاراکتر $c$ در $z$ با قرار دادن $z[j] = c$ و سپس افزایش $j$، و بدون تغییر باقی گذاشتن $i$. این عملیات هیچ کاراکتری را از $x$ بررسی نمی‌کند.
- جابه‌جایی (twiddle) دو کاراکتر بعدی با کپی کردن آن‌ها از $x$ به $z$، ولی با ترتیب برعکس؛ این کار را با قرار دادن $z[j] = x[i+۱]$ و $z[j+۱] = x[i]$، و سپس افزایش $i = i + ۲$ و $j = j + ۲$ انجام می‌دهیم. این عملیات $x[i]$ و $x[i+۱]$ را بررسی می‌کند.
- نابود کردن (kill) باقی مانده‌ی $x$ با قرار دادن $i = m + ۱$. این عملیات تمام کاراکترهای درون $x$ را که هنوز بررسی نشده‌اند، بررسی می‌کند. اگر این عملیات انجام شود، باید آخرین عملیات انجام شده باشد.

به عنوان یک مثال، یک روش برای تبدیل رشته‌ی ورودی "algorithm" به رشته‌ی نهایی "altruistic"، انجام دنباله‌ی اعمال زیر است، که در آن کاراکترهایی که زیر آن‌ها خط کشیده شده است، $x[i]$ و $z[j]$ بعد از انجام عملیات هستند.

| عملیات | $x$ | $z$ |
|---|---|---|
| *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 داشته باشید که دنباله‌های تبدیل مختلف دیگری نیز وجود دارند که این تبدیل را انجام می‌دهند.

هر یک از اعمال انجام شده هزینه‌ی خاصی دارند. هزینه‌ی یک عملیات به کاربرد خاص آن بستگی دارد، ولی فرض می‌کنیم که هزینه‌ی هر یک از اعمال یک ثابت است که ما می‌دانیم. همچنین فرض می‌کنیم که هزینه‌ی هر یک از عملیات کپی و جایگزینی کم‌تر از هزینه‌ی ترکیب دو عملیات حذف و درج است؛ در غیر این صورت هیچ وقت از اعمال کپی و جایگزینی استفاده نخواهد شد. هزینه‌ی دنباله‌ای از عملیات برابر است با مجموع هزینه‌های اعمال درون دنباله. برای دنباله‌ی بالا، هزینه‌ی تبدیل "algorithm" به "altruistic" برابر است با

$$(۳cost(copy)) + cost(replace) + cost(delete) + ۴cost(insert)) + cost(twiddle) + cost(kill)$$

۱. با داشتن دو دنباله‌ی $x[۱..m]$ و $y[۱..n]$ و مجموعه‌ی هزینه‌های اعمال تبدیل، *فاصله‌ی ویرایشی* از $x$ به $y$ برابر است با هزینه‌ی کم هزینه‌ترین دنباله‌ی عملیات ممکن که $x$ را به $y$ تبدیل می‌کند. یک الگوریتم برنامه‌ریزی پویا ارائه کنید که فاصله‌ی ویرایشی از $x$ به $y$ را یافته و یک دنباله‌ی عملیات بهینه برای آن چاپ می‌کند. زمان اجرا و فضای مورد نیاز الگوریتم خود را تحلیل کنید.

مسئله‌ی فاصله‌ی ویرایشی، حالت کلی‌تر مسئله‌ی تنظیم (aligning) دو دنباله‌ی DNA است. متدهای مختلفی برای تعیین میزان شباهت دو دنباله‌ی DNA با استفاده از میزان کردن آن‌ها وجود دارد. یکی از این متدهای میزان کردن دو دنباله‌ی $x$ و $y$ این است که در مکان‌هایی از دو دنباله (شامل دو انتهای دنباله) یک فاصله‌ی اضافی درج کنیم به طوری که دنباله‌های حاصل $x'$ و $y'$ دارای طول یکسانی باشند، ولی در مکان یکسانی کاراکتر فاصله نداشته باشند (یعنی برای هیچ مکانی مانند $j$، $x'[j]$ و $y'[j]$ فاصله نباشد.) سپس به هر مکان یک «امتیاز» نسبت می‌دهیم. مکان $j$ به صورت زیر امتیاز خود را دریافت می‌کند:

- ۱+ اگر $x'[j] = y'[j]$ و هیچ یک از این دو فاصله نیست.
- ۱- اگر $x'[j] \neq y'[j]$ و هیچ یک از این دو فاصله نیست.
- ۲- اگر یکی از $x'[j]$ و $y'[j]$ فاصله باشد.

امتیاز کلی تنظیم برابر است با مجموع امتیازات تمام مکان‌ها. به عنوان مثال، برای دو دنباله‌ی $x = GATCGGCAT$ و $y = CAATGTGAATC$، یک تنظیم به صورت زیر است:

```
G ATCG GCAT
CAAT GTGAATC
-*++*+*+-++*
```

یک علامت + زیر یک مکان نشان‌دهنده‌ی امتیاز ۱+، - نشان‌دهنده‌ی امتیاز ۱-، و * نشان‌دهنده‌ی امتیاز ۲- برای آن مکان است، یعنی امتیاز کلی این تنظیم برابر است با $۶ \times ۱ - ۲ \times ۱ - ۴ \times ۲ = -۴$.

II. توضیح دهید که چطور می‌توان با استفاده از زیرمجموعه‌ای از اعمال تبدیل کپی، جایگزینی، حذف، درج، جابه‌جایی و نابود کردن، مسئله‌ی یافتن یک تنظیم بهینه را به یک مسئله‌ی فاصله‌ی ویرایشی تبدیل کرد.

### ۱۵-۶ *برنامه‌ریزی برای میهمانی کمپانی*

پروفسور استورات (Stewart) مشاور مدیر عامل یک کمپانی است که در حال ترتیب دادن یک میهمانی است. کمپانی یک ساختار سلسله مراتبی دارد؛ یعنی، رابطه‌های سرپرستی یک درخت را تشکیل می‌دهند که ریشه‌ی آن مدیر عامل است. پرسنل دفتر هر یک از کارمندان را به نسبت میزان قابلیت معاشرت رتبه‌بندی کرده‌اند، که یک عدد حقیقی است. برای این که میهمانی برای تمام شرکت کننده‌ها لذت بخش باشد، مدیرعامل نمی‌خواهد که یک کارمند و سرپرست آن هر دو در میهمانی باشند. درختی که ساختار کمپانی را توصیف می‌کند با استفاده از نمایش فرزند چپ، برادر راست، که در بخش ۱۰-۴ توضیح داده شد، به پروفسور استورات داده شده است. هر گره‌ی درخت، علاوه بر اشاره‌گرها، نام کارمند و رتبه‌ی میزان معاشرت آن کارمند را نگه می‌دارد.

یک الگوریتم طراحی کنید که لیستی از میهمانان تولید می‌کند که در آن مجموع رتبه‌ی معاشرت میهمانان در آن کمینه است. زمان اجرای الگوریتم خود را تحلیل کنید.

### ۱۵-۷ *الگوریتم Viterbi*

می‌توان از برنامه‌ریزی پویا بر روی یک گراف جهت دار $G=(V,E)$ برای تشخیص صدا استفاده کرد. هر یال $(u,v)\in E$ با یک صدای $\sigma(u,v)$ از یک مجموعه‌ی متناهی $\Sigma$ از صداها برچسب گذاری شده است. این گراف برچسب گذاری شده یک مدل فرمال از صحبت کردن یک انسان در یک زبان محدود شده است. هر مسیر در گراف که از یک رأس مشخص شده‌ی $v_0\in V$ شروع می‌شود متناظر است با یک دنباله‌ی ممکن از صداها که توسط مدل درست می‌شود. برچسب یک مسیر جهت دار به صورت اتصال برچسب یال‌های درون مسیر تعریف می‌شود.

I. یک الگوریتم کارا ارائه کنید که با دریافت یک گراف $G$ با یال‌های برچسب‌دار و رأس مشخص شده‌ی $v_0$ و یک دنباله‌ی $s=\langle\sigma_1,\sigma_2,\ldots,\sigma_k\rangle$ از کاراکترها از $\Sigma$، در صورت وجود یک مسیر در گراف $G$ بازمی‌گرداند که با $v_0$ شروع شده و برچسب آن $s$ است. در صورتی که این مسیر وجود نداشت، گراف باید NO-SUCH-PATH را بازگرداند. زمان اجرای الگوریتم خود را تحلیل کنید. (*راهنمایی:* ممکن است مطالب فصل ۲۲ را مفید بیابید.)

اکنون فرض کنید که به تمام یال‌های $(u,v)\in E$ یک احتمال نامنفی $p(u,v)$ هم داده شده است که احتمال گذار از روی یال از رأس $u$، و ساختن صدای مربوطه است. مجموع احتمالات یال‌های خارج شده از هر رأس برابر ۱ است. احتمال یک مسیر برابر است با حاصل ضرب

احتمالات یال‌های روی آن مسیر. می‌توان احتمال یک مسیر که از $v_0$ شروع می‌شود را به صورت این احتمال توصیف کرد که یک «حرکت تصادفی» (random walk) با شروع از $v_0$ آن مسیر را طی کند، که در آن انتخاب یال بعد در یک رأس $u$ به وسیله‌ی احتمالات تعیین شده بر روی یال‌های خروجی $u$ انجام می‌شود.

II. جواب خود به قسمت I را طوری گسترش دهید که اگر یک مسیر بازگردانده می‌شود، *محتمل‌ترین مسیری* باشد که از $v_0$ شروع شده و دارای برچسب $s$ است. زمان اجرای الگوریتم خود را تحلیل کنید.

### ۱۵-۶ *فشرده‌سازی تصویر با ایجاد درز*

یک تصویر رنگی متشکل از یک آرایه‌ی $A[1..m,1..n]$ با ابعاد $m \times n$ از پیکسل‌ها به ما داده شده است، که در آن هر رنگ هر پیکسل به صورت یک سه‌تایی قرمز، سبز، و آبی (RGB) تعریف می‌شود. فرض کنید که می‌خواهیم تصویر را کمی فشرده کنیم. برای این کار، یک پیکسل از هر یک از $m$ سطر حذف می‌کنیم، تا کل تصویر یک پیکسل نازک‌تر شود. ولی برای جلوگیری از مختل شدن ظاهر تصویر، باید پیکسل‌هایی که از دو سطر مجاور حذف می‌شوند، در یک ستون، یا در دو ستون مجاور باشند؛ پیکسل‌های حذف شده یک «درز» را از سطر بالا تا سطر پایین تشکیل می‌دهند، که در آن پیکسل‌های متوالی، به صورت ستونی یا اریب با هم مجاورند.

I. نشان دهید که تعداد درزهای ممکن به این شکل، حداقل از مرتبه‌ی نمایی بر حسب $m$ رشد می‌کند، با فرض این که $n > 1$.

II. اکنون فرض کنید که برای هر پیکسل $A[i,j]$، یک مقیاس شکستگی حقیقی $d[i,j]$ هم محاسبه می‌کنیم، که نشان می‌دهد حذف پیکسل $A[i,j]$ چقدر باعث شکستگی تصویر می‌شود. به صورت شهودی، هر چه میزان شکستگی یک پیکسل کم‌تر باشد، شباهت آن با همسایه‌هایش بیشتر خواهد بود. همچنین فرض کنید که مقیاس شکستگی درز را به صورت مجموع شکستگی‌های پیکسل‌های آن تعریف می‌کنیم. الگوریتمی بدهید که یک درز با پایین‌ترین مقیاس شکستگی را به دست بدهد. الگوریتم شما چقدر کارامد است؟

### ۱۵-۹ *شکستن یک رشته*

یک زبان خاص پردازش رشته به برنامه‌نویس اجازه می‌دهد که یک رشته را به دو قطعه بشکنند. چون در این عملیات، باید رشته کپی شود، $n$ واحد زمانی طول می‌کشد تا رشته‌ای با $n$ کاراکتر به دو قطعه شکسته شود. فرض کنید که یک برنامه‌نویس می‌خواهد یک رشته را به قطعات زیادی تقسیم کند. ترتیب شکستن‌ها می‌تواند بر روی کل زمان صرف شده تأثیر بگذارد. برای مثال، فرض کنید که برنامه‌نویس می‌خواهد یک رشته‌ی ۲۰ کاراکتری را بعد از کاراکترهای ۲، ۸، و ۱۰ بشکند (که در آن کاراکترها به ترتیب صعودی از چپ به راست، با

آغاز از ۱، شماره‌گذاری شده‌اند). اگر برنامه را طوری بنویسد که شکستن‌ها از چپ به راست رخ دهند، آن گاه شکستگی اول ۲۰ واحد زمان هزینه خواهد داشت، شکستگی دوم ۱۸ واحد زمان (که در آن رشته‌ی متشکل از کاراکترهای ۳ تا ۲۰، از کاراکتر ۸ شکسته خواهد شد)، و شکستگی سوم ۱۲ واحد زمان، که مجموع ۵۰ واحد زمان را خواهد داد. ولی اگر شکستگی‌ها را از راست به چپ برنامه‌ریزی کند، آن گاه اولین شکستگی ۲۰ واحد زمان هزینه خواهد داشت، دومین شکستگی ۱۰ واحد زمان، و سومین شکستگی ۸ واحد زمان، که به مجموع ۳۸ واحد زمان منجر می‌شود. باز با یک ترتیب دیگر، می‌توان اول از کاراکتر ۸ رشته را شکست (با هزینه‌ی ۲۰)، سپس قطعه‌ی سمت چپ را از کاراکتر ۲ (با هزینه‌ی ۸)، و نهایتاً قطعه‌ی سمت راست را از کاراکتر ۱۰ (با هزینه‌ی ۱۲)، با هزینه‌ی کلی ۴۰.

الگوریتمی طراحی کنید که با دریافت شماره‌ی کاراکترهایی که بعد از آن‌ها باید رشته شکسته شود، یک دنباله با کم‌ترین هزینه برای شکستن رشته تعیین کند. به صورت رسمی‌تر، با دریافت یک رشته‌ی $S$ با $n$ کاراکتر و یک آرایه‌ی $L[1..m]$ حاوی نقاط شکستگی، کم‌ترین هزینه‌ی ممکن برای دنباله‌ای از شکستگی‌ها را، به همراه خود دنباله محاسبه کنید.

### ۱۵-۱۰ *برنامه‌ریزی برای یک استراتژی سرمایه‌گذاری*

اطلاعات کامپیوتر شما به شما کمک کرده است که یک شغل جذاب در کمپانی کامپیوتر Acme به دست آورید، با یک پرداخت اولیه‌ی ۱۰،۰۰۰$. می‌خواهید با این پول سرمایه‌گذاری کنید، به طوری که در پایان ۱۰ سال بالاترین سود ممکن را به دست آورید. تصمیم می‌گیرید که از کمپانی سرمایه‌گذاری ملقمه برای برنامه‌ریزی سرمایه‌گذاری خود استفاده کنید. کمپانی ملقمه از شما می‌خواهد که قوانین زیر را در نظر بگیرید. این کمپانی $n$ سرمایه‌گذاری مختلف ارائه می‌کند، با شماره‌های ۱ تا $n$. در هر سال $j$، سرمایه‌گذاری $i$ نرخ سود $r_{ij}$ را برای شما خواهد داشت. به عبارت دیگر، اگر در سال $j$، $d$ دلار در سرمایه‌گذاری $i$ خرج کنید، در پایان سال $j$، $d\,r_{ij}$ دلار دریافت خواهید کرد. نرخ‌های بازگشتی تضمین شده هستند، یعنی، نرخ‌های بازگشتی برای تمام ۱۰ سال آینده از قبل به شما داده شده است. شما فقط سالی یک بار در مورد سرمایه‌گذاری تصمیم‌گیری می‌کنید. اگر تصمیم بگیرید که پول خود را در دو سال متوالی در یک مجموعه‌ی سرمایه‌گذاری باقی بگذارید، باید یک هزینه‌ی $f_1$ بپردازید، در حالی که اگر بخواهید مجموعه‌ی سرمایه‌گذاری‌های خود را عوض کنید باید $f_2$ دلار هزینه کنید.

مسئله، همان طور که گفته شد، به شما اجازه می‌دهد که در هر سال پول خود را در مجموعه‌های مختلفی سرمایه‌گذاری کنید. اثبات کنید که یک استراتژی سرمایه‌گذاری بهینه وجود دارد که در هر سال، تمام پول را فقط در یک سرمایه‌گذاری خرج می‌کند. (به خاطر بیاورید که یک استراتژی سرمایه‌گذاری بهینه مقدار پول را پس از ۱۰ سال بیشینه می‌کند، و کاری به اهداف دیگر، مانند کمینه کردن ریسک، ندارد.)

II. اثبات کنید که مسئله‌ی برنامه‌ریزی برای استراتژی بهینه‌ی سرمایه‌گذاری از زیرساختار بهینه پیروی می‌کند.

III. الگوریتمی ارائه کنید که استراتژی بهینه‌ی سرمایه‌گذاری را برای شما طراحی کند. زمان اجرای برنامه‌ی شما چقدر است؟

IV. فرض کنید که کمپانی ملقمه یک محدودیت جدید را به اجرا می‌گذارد، که طبق آن در هر لحظه، نمی‌توانید بیش از ۱۵,۰۰۰$ در هیچ یک از طرح‌های سرمایه‌گذاری داشته باشید. نشان دهید که مسئله‌ی بیشینه کردن درآمد در پایان ۱۰ سال دیگر از زیرساختار بهینه پیروی نمی‌کند.

### ۱۵-۱۱ *برنامه‌ریزی برای انبار*

کمپانی Rinky Dinks ماشین‌هایی می‌سازد که بر روی یخ سر می‌خورند. تقاضا برای چنین محصولاتی در هر ماه متفاوت است، و بنابراین کمپانی می‌خواهد با داشتن این تقاضای متغیر، ولی قابل پیش‌بینی، برنامه‌ای برای تولید خود طراحی کند. کمپانی می‌خواهد برنامه را برای $n$ ماه آینده طراحی کند. برای هر ماه $i$، کمپانی تقاضای $d_i$ (تعداد ماشین‌هایی که در ماه $i$ به فروش خواهد رفت) را می‌داند. فرض کنید $D = \sum_{i=1}^{n} d_i$ کل تقاضا برای $n$ ماه آینده باشد. کمپانی یک خدمه‌ی تمام وقت دارد که تا $m$ ماشین در هر ماه تولید می‌کنند. اگر کمپانی در یک ماه خاص به بیش از $n$ ماشین نیاز داشته باشد، می‌تواند کارگرهای جدید نیمه وقت استخدام کند، با هزینه‌ی $c$ دلار برای هر ماشین. به علاوه اگر در انتهای یک ماه، کمپانی ماشین‌های به فروش نرفته داشته باشد، باید هزینه‌ای برای انبار پرداخت کند. هزینه‌ی نگهداری $j$ ماشین به صورت یک تابع $h(j)$ برای $j = 1,2,...,D$ داده شده است، که در آن $h(j) \geq 0$ برای $1 \leq j \leq D$، و $h(j) \leq h(j+1)$ برای $1 \leq j \leq D-1$.

الگوریتمی ارائه کنید که یک برنامه برای کمپانی طراحی می‌کند که علاوه بر رسیدگی به تمام تقاضاها، هزینه‌های کمپانی را هم کمینه می‌کند. زمان اجرا باید بر حسب $n$ و $D$ از مرتبه‌ی چندجمله‌ای باشد.

### ۱۵-۱۲ *خرید بازیکن‌های بدون قرارداد بیسبال*

فرض کنید شما مدیر یک تیم در لیگ بیسبال هستید. در فصل تعطیلی مسابقات، باید چند بازیکن بدون قرارداد برای تیم خود استخدام کنید. مالک تیم به میزان $X$$ بودجه برای خرید بازیکن به شما داده است. شما فقط اجازه دارید که کم‌تر از $X$$ برای خرید تمام بازیکن‌ها خرج کنید، و اگر بیش از آن خرج کنید، مالک تیم شما را اخراج خواهد کرد.

شما $N$ پست مختلف را در نظر گرفته‌اید، و برای هر پست، $P$ بازیکن بدون قرارداد در دسترس هستند. از آن جایی که نمی‌خواهید لیست خود را با بازیکن‌های زیاد در هر پست

شلوغ کنید، برای هر پست حداکثر یک بازیکن استخدام خواهید کرد. (اگر برای یک پست خاص هیچ بازیکنی استخدام نکنید، آن گاه از همان بازیکنی استفاده خواهید کرد که از قبل برای آن پست داشتید.)

برای تعیین این که هر بازیکن چقدر ارزش خواهد داشت، تصمیم می‌گیرید که از یک آمار sabermetric[1] با نام "VORP" استفاده کنید. یک بازیکن با VORP بالاتر، با ارزش‌تر از بازیکنی با VORP پایین‌تر است. استخدام یک بازیکن با VORP بالاتر لزوماً گران‌تر از یک بازیکن با VORP پایین‌تر نیست، چرا که فاکتورهای دیگری غیر از ارزش بازیکن، هزینه‌ی استخدام او را تعیین می‌کنند.

برای هر بازیکن موجود، سه فقره اطلاعات در دسترس دارید:

- پست بازیکن،
- مقدار پول مورد نیاز برای استخدام بازیکن، و
- VORP بازیکن.

الگوریتمی طراحی کنید که مجموع VORP تمام بازیکنانی که استخدام می‌کنید را بیشینه می‌کند، بدون این که بیش از $X خرج کند. می‌توانید فرض کنید که هزینه‌ی استخدام هر بازیکن مضربی از ۱۰۰,۰۰۰$ است. الگوریتم شما باید مجموع VORP بازیکنان، کل پول هزینه شده، و لیست بازیکنان استخدام شده را به خروجی بدهد. زمان اجرا و حافظه‌ی مصرفی برنامه‌ی خود را تحلیل کنید.

[1] Sabermetric عبارت است از کاربرد تحلیل آماری برای داده‌های بیسبال، که روش‌های مختلفی برای مقایسه‌های ارزش‌های مربوطه‌ی بازیکنان بیسبال ارائه می‌کند.


# الگوریتم‌های حریصانه

## ۱۶-۰ مقدمه

الگوریتم‌های مسئله‌های بهینه‌سازی معمولاً مراحل مختلفی را طی می‌کنند، و در هر مرحله مجموعه‌ای از انتخاب‌ها را پیش رو دارند. برای بسیاری از مسائل بهینه‌سازی، استفاده از برنامه‌ریزی پویا برای تعیین بهترین انتخاب کار اضافی است؛ می‌توان با الگوریتم‌های ساده‌تر و کاراتر هم همان انتخاب را انجام داد. یک *الگوریتم حریصانه* همیشه گزینه‌ای را انتخاب می‌کند که در لحظه به نظر بهترین می‌آید. یعنی، یک انتخاب بهینه‌ی محلی (نسبی) انجام می‌دهد به این امید که این انتخاب به یک جواب بهینه‌ی مطلق ختم شود. در این فصل مسائل بهینه‌سازی مطرح می‌شوند که می‌توان آن‌ها را با استفاده از الگوریتم‌های حریصانه حل کرد. قبل از خواندن این فصل، باید در مورد برنامه‌ریزی پویا در فصل ۱۵ مطالعاتی کرده باشید، مخصوصاً بخش ۱۵-۳.

الگوریتم‌های حریصانه همیشه به جواب‌های بهینه ختم نمی‌شوند، ولی برای بسیاری از مسائل می‌شوند. ابتدا در بخش ۱۶-۱ یک مسئله‌ی ساده ولی نابدیهی را بررسی می‌کنیم، مسئله‌ی انتخاب فعالیت، که در آن یک الگوریتم حریصانه جواب بهینه را محاسبه می‌کند. ولی ابتدا یک راه حل برنامه‌ریزی پویا برای آن در نظر می‌گیریم، و سپس نشان می‌دهیم که یک راه حل حریصانه همیشه به یک جواب بهینه ختم می‌شود. در بخش ۱۶-۲ عناصر اصلی رویکرد حریصانه بررسی می‌شوند، و در آن یک رویکرد سرراست برای اثبات درستی الگوریتم‌های حریصانه به ما می‌دهد. در بخش ۱۶-۳ کاربرد مهمی از تکنیک حریصانه معرفی می‌شود: طراحی یک کد متراکم سازی داده (Huffman). در بخش ۱۶-۴ بعضی از تئوری‌های پشت ساختارهای ترکیبی به نام «ماتروید‌ها» (matroids) بررسی می‌شوند که در آن‌ها همیشه یک الگوریتم حریصانه به جواب بهینه می‌رسد. در نهایت در بخش ۱۶-۵، کاربرد ماتروید‌ها در مسئله‌ی برنامه‌ریزی کارهایی با زمان واحد و مهلت انجام و جریمه معرفی

خواهد شد.

متد حریصانه، متدی است قدرتمند که برای محدوده‌ی وسیعی از مسائل کاربرد دارد. در فصل‌های آینده الگوریتم‌های بسیاری معرفی می‌شوند که می‌توان به آن‌ها به عنوان کاربردهایی از الگوریتم‌های حریصانه نگاه کرد، مانند الگوریتم‌های درخت پوشای کمینه (فصل ۲۳)، الگوریتم Dijkstra برای یافتن کوتاه‌ترین مسیرها از یک رأس (فصل ۲۴)، و مکاشفه‌ی تبدیل مجموعه‌ی حریصانه‌ی Chv'atal (فصل ۳۵). الگوریتم‌های درخت پوشای کمینه مثالی معروف از متدهای حریصانه هستند. با این که می‌توانید این فصل و فصل ۲۳ را مستقل از هم مطالعه کنید، ممکن است خواندن آن‌ها را به همراه یکدیگر مفید بیابید.

## ۱-۱۶ یک مسئله‌ی انتخاب فعالیت

اولین مثال ما، مسئله‌ی برنامه‌ریزی برای چندین فعالیت است که نیاز دارند از یک منبع مشترک به صورت اختصاصی استفاده کنند، و هدف انتخاب بزرگ‌ترین مجموعه‌ی فعالیت‌های سازگار است. فرض کنید یک مجموعه‌ی $S = \{a_1, a_2, ..., a_n\}$ از $n$ *فعالیت* ارائه شده داریم، که همه باید از یک منبع، مانند یک سالن سخنرانی استفاده کنند، که در هر زمان فقط یکی از فعالیت‌ها می‌تواند از منبع استفاده کند. هر فعالیت $a_i$ دارای یک زمان شروع $s_i$ و یک زمان پایان $f_i$ است، که در آن $0 \le s_i < f_i < \infty$. فعالیت $a_i$، در صورت انتخاب، در بازه‌ی نیمه باز $[s_i, f_i)$ اجرا می‌شود. فعالیت‌های $a_i$ و $a_j$ *سازگار* هستند اگر بازه‌های $[s_i, f_i)$ و $[s_j, f_j)$ با یکدیگر تلاقی نداشته باشند (به عبارت دیگر $a_i$، و $a_j$ سازگار هستند اگر $s_i \ge f_j$ یا $s_j \ge f_i$). *مسئله‌ی انتخاب فعالیت* این است که یک مجموعه از فعالیت‌های دوبه‌دو سازگار با اندازه‌ی بیشینه انتخاب کنیم. فرض می‌کنیم که فعالیت‌ها به ترتیب صعودی زمان پایان مرتب شده‌اند:

$$f_1 \le f_2 \le f_3 \le \ldots \le f_{n-1} \le f_n \qquad (۱-۱۶)$$

(بعداً فایده‌ی این فرض را خواهیم دید.) به عنوان مثال، مجموعه‌ی $S$ زیر از فعالیت‌ها را در نظر بگیرید:

| $i$ | ۱ | ۲ | ۳ | ۴ | ۵ | ۶ | ۷ | ۸ | ۹ | ۱۰ | ۱۱ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | ۱ | ۳ | ۰ | ۵ | ۳ | ۵ | ۶ | ۸ | ۸ | ۲ | ۱۲ |
| $f_i$ | ۴ | ۵ | ۶ | ۷ | ۸ | ۹ | ۱۰ | ۱۱ | ۱۲ | ۱۳ | ۱۴ |

در این مثال، زیرمجموعه‌ی $\{a_3, a_9, a_{11}\}$ حاوی فعالیت‌های دوبه‌دو سازگار است. با این حال، این یک زیرمجموعه‌ی بیشینه نیست، چرا که زیرمجموعه‌ی $\{a_1, a_4, a_8, a_{11}\}$ از آن بزرگ‌تر است. در واقع، $\{a_1, a_4, a_8, a_{11}\}$ یکی از بزرگ‌ترین زیرمجموعه‌ها از فعالیت‌های دوبه‌دو سازگار است: یکی دیگر از بزرگ‌ترین زیرمجموعه‌ها $\{a_2, a_4, a_9, a_{11}\}$ است.

این مسئله را طی مراحل مختلفی حل خواهیم کرد. با فکر کردن به یک جواب برنامه‌ریزی پویا آغاز می‌کنیم، که در آن، هنگام تصمیم برای انتخاب زیرمسئله‌های مورد استفاده در جواب بهینه،

گزینه‌های مختلفی را در نظر می‌گیریم. بعداً خواهیم دید که فقط به در نظر گرفتن یک گزینه نیاز داریم - گزینه‌ی حریصانه - و وقتی انتخاب حریصانه را انجام دادیم، فقط یکی از زیرمسئله‌ها باقی می‌ماند. بر مبنای این مشاهدات، یک الگوریتم بازگشتی حریصانه توسعه می‌دهیم که مسئله‌ی انتخاب فعالیت را حل می‌کند. فرآیند توسعه‌ی راه حل بازگشتی را با تبدیل الگوریتم بازگشتی به یک الگوریتم تکراری کامل می‌کنیم. با این که مراحلی که در این بخش طی می‌کنیم کمی پیچیده‌تر از حالت معمول برای توسعه‌ی یک الگوریتم حریصانه است، ولی این مراحل رابطه‌ی میان الگوریتم‌های حریصانه و برنامه‌ریزی پویا را مشخص می‌کنند.

## زیرساختار بهینه‌ی مسئله‌ی انتخاب فعالیت

به سادگی می‌توانیم ببینیم که مسئله‌ی انتخاب فعالیت از زیرساختار بهینه پیروی می‌کند. اجازه دهید فرض کنیم $S_{ij}$ نشان دهنده‌ی مجموعه‌ی فعالیت‌هایی است که بعد از پایان فعالیت $a_i$، آغاز می‌شوند، و قبل از آغاز فعالیت $a_j$ پایان می‌یابند. فرض کنید که می‌خواهیم یک مجموعه‌ی بیشینه از فعالیت‌های دوبه‌دو سازگار در $S_{ij}$ بیابیم، و همچنین فرض کنید که چنین مجموعه‌ی بیشینه‌ای $A_{ij}$ است، شامل یک فعالیت $a_k$. با شامل کردن $a_k$ در یک جواب بهینه، دو زیرمسئله برای ما باقی می‌ماند: یافتن فعالیت‌های دوبه‌دو سازگار در مجموعه‌ی $S_{ik}$ (فعالیت‌هایی که بعد از پایان فعالیت $a_i$ آغاز می شوند و قبل از آغاز فعالیت $a_k$ پایان می‌یابند)، و یافتن فعالیت‌های دوبه‌دو سازگار در مجموعه‌ی $S_{kj}$ (فعالیت‌هایی که بعد از پایان فعالیت $a_k$ آغاز می شوند و قبل از آغاز فعالیت $a_j$ پایان می‌یابند). فرض کنید $A_{ik} = A_{ij} \cap S_{ik}$ و $A_{kj} = A_{ij} \cap S_{kj}$، به طوری که $A_{ik}$ حاوی فعالیت‌هایی در $A_{ij}$ است که قبل از آغاز $a_k$، پایان می‌یابند، و $A_{kj}$ حاوی فعالیت‌هایی در $A_{ij}$ است که بعد از پایان $a_k$ آغاز می‌شوند. بنابراین داریم $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$، و همچنین مجموعه‌ی بیشینه‌ی $A_{ij}$ از فعالیت‌های دوبه‌دو سازگار در $S_{ij}$ از $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ فعالیت تشکیل شده است.

بحث برش و چسباندن معمول نشان می‌دهد که جواب بهینه‌ی $A_{ij}$ همچنین باید حاوی جواب‌های بهینه به دو زیرمسئله‌ی $S_{ik}$ و $S_{kj}$ باشد. اگر می‌توانستیمک مجموعه‌ی $A'_{kj}$ از فعالیت‌های دوبه‌دو سازگار در $S_{kj}$ بیابیم که در آن $|A'_{kj}| > |A_{kj}|$، آن گاه به جای $A_{kj}$ می‌توانستیم از $A'_{kj}$ در جواب زیرمسئله‌ی $S_{ij}$ استفاده کنیم. در این صورت یک مجموعه از $|A_{ij}| = |A_{kj}| + |A_{ik}| + 1 < |A'_{kj}| + |A_{ik}| + 1$ فعالیت دوبه‌دو سازگار داشتیم، که با فرض این که $A_{ij}$ یک جواب بهینه است تناقض دارد. یک بحث مشابه را می‌توان برای فعالیت‌های $S_{ik}$ به کار برد.

این روش تعریف زیرساختار بهینه پیشنهاد می‌کند که احتمالاً می‌توانیم مسئله‌ی انتخاب فعالیت را به کمک برنامه‌ریزی پویا حل کنیم. اگر اندازه‌ی یک جواب بهینه برای مجموعه‌ی $S_{ij}$ را با $c[i,j]$ نشان دهیم، آن گاه رابطه‌ی بازگشتی زیر را داریم:

$$c[i,j] = c[i,k] + c[k,j] + 1$$

مسلماً اگر نمی‌دانستیم که یک جواب بهینه برای مجموعه‌ی $S_{ij}$ حاوی یک فعالیت $a_k$ است، مجبور

بودیم که تمام فعالیت‌های $S_{ij}$ را امتحان کنیم تا ببینیم از کدام یک باید استفاده کنیم، پس

$$c[i,j]=\begin{cases}0 & S_{ij}=\varnothing \text{ اگر}\\ \max\limits_{i<k<j}\{c[i,k]+c[k,j]+1\} & S_{ij}\neq\varnothing \text{ اگر}\end{cases} \qquad (۳-۱۶)$$

سپس می‌توانیم یک الگوریتم بازگشتی طراحی کرده و در آن از به خاطر سپاری استفاده کنیم، یا می‌توانیم از پایین به بالا کار کرده و ورودی‌های جدول را به ترتیب پر کنیم. ولی در این جا یک خصوصیت مهم دیگر مسئله‌ی انتخاب فعالیت را فراموش کرده‌ایم که می‌تواند بسیار مفید باشد.

## انجام انتخاب حریصانه

آیا می‌توانیم یک فعالیت را برای اضافه کردن به راه حل بهینه انتخاب کنیم بدون این که مجبور باشیم ابتدا تمام زیرمسئله‌ها را حل کنیم؟ در این صورت دیگر نیازی نیست که تمام انتخاب‌های دخیل در رابطه‌ی بازگشتی (۱۶-۲) را در نظر بگیریم. در واقع برای مسئله‌ی انتخاب فعالیت، نیاز داریم که فقط یک انتخاب را در نظر بگیریم: انتخاب حریصانه.

منظور از انتخاب حریصانه برای مسئله‌ی انتخاب فعالیت چیست؟ شهود پیشنهاد می‌کند که می‌توانیم فعالیتی را انتخاب کنیم که بیشترین منبع ممکن را برای فعالیت‌های دیگر باقی می‌گذارد. اکنون از فعالیت‌هایی که در نهایت انتخاب خواهیم کرد، یکی از آن‌ها باید قبل از بقیه تمام شود. بنابراین، شهود می‌گوید که فعالیتی را از $S$ انتخاب کنیم که زودترین زمان پایان را دارد، چرا که در این صورت منابع را برای بیشترین فعالیت‌های ممکن بعد از آن باقی می‌گذارد. (اگر بیش از یک فعالیت در $S$، زودترین زمان پایان را داشت، آن گاه می‌توانیم به دلخواه یکی را انتخاب کنیم.) به عبارت دیگر، از آن جایی که فعالیت‌ها به ترتیب صعودی زمان پایان مرتب شده‌اند، انتخاب حریصانه $a_1$ است. انتخاب اولین فعالیتی که پایان می‌یابد تنها راه انتخاب حریصانه برای این مسئله نیست؛ تمرین ۱۶-۱-۳ از شما می‌خواهد که گزینه‌های دیگر را هم بررسی کنید.

اگر انتخاب حریصانه را انجام دهیم، فقط یک زیرمسئله‌ی دیگر برای حل کردن باقی می‌ماند: یافتن فعالیت‌هایی که بعد از پایان $a_1$ آغاز می‌شوند. چرا نیازی نیست به دنبال فعالیت‌هایی بگردیم که قبل از آغاز $a_1$ پایان می‌یابند؟ داریم $s_1 < f_1$، و $f_1$ اولین زمان پایان در تمام فعالیت‌ها است، و بنابراین هیچ فعالیتی نمی‌تواند زمان پایانی کم‌تر یا مساوی $s_1$ داشته باشد. پس تمام فعالیت‌هایی که با $a_1$ سازگار هستند باید بعد از پایان $a_1$ آغاز شوند.

به علاوه، قبلاً تعین کردیم که مسئله‌ی انتخاب فعالیت از زیرساختار بهینه پیروی می‌کند. فرض کنید $S_k=\{a_i\in S: s_i\geq f_k\}$ مجموعه‌ی فعالیت‌هایی باشد که بعد از پایان فعالیت $a_k$ پایان می‌یابند. اگر به صورت حریصانه فعالیت $a_1$ را انتخاب کنیم، آن گاه $S_1$ به عنوان تنها مسئله‌ای که باید حل شود باقی می‌ماند.[1] زیرساختار بهینه به ما می‌گوید که اگر $a_1$ در جواب بهینه باشد، آن گاه یک جواب بهینه

[1] بعضی مواقع منظور ما از $S_k$ یک زیرمسئله است، نه مجموعه‌ای از فعالیت‌ها. همیشه از محتوای متن واضح خواهد بود که منظور ما از $S_k$ مجموعه‌ای از فعالیت‌ها است و یا زیرمسئله‌ای که ورودی آن مجموعه‌ی مربوطه است.

به مسئله‌ی اصلی حاوی فعالیت $a_1$ و تمام فعالیت‌هایی خواهد بود که در یک جواب بهینه به زیرمسئله‌ی $S_1$ هستند.

یک سؤال بزرگ باقی می‌ماند: آیا این شهود صحیح است؟ آیا انتخاب حریصانه – که در آن همیشه فعالیتی را انتخاب می‌کنیم که زودتر از بقیه پایان می‌یابد – همیشه بخشی از جواب بهینه است؟ قضیه‌ی زیر نشان می‌دهد که در واقع این گونه است.

**قضیه‌ی ۱-۱۶** یک زیرمسئله‌ی ناتهی $S_k$ را در نظر بگیرید، و فرض کنید $a_m$ یک فعالیت در $S_k$ باشد که کم‌ترین زمان پایان را دارد. در این صورت یک زیرمجموعه‌ی بیشینه از فعالیت‌های دوبه‌دو سازگار در $S_k$ وجود دارد که شامل $a_m$ است.

*اثبات* فرض کنید $A_k$ یک مجموعه‌ی بیشینه از فعالیت‌های دوبه‌دو سازگار در $S_k$ باشد، و فرض کنید $a_j$ فعالیتی در $A_k$ با زودترین زمان پایان باشد. اگر $a_j = a_m$، کار ما تمام است، چرا که نشان داده‌ایم که $a_m$ در یک مجموعه‌ی بیشینه از فعالیت‌های دوبه‌دو سازگار از $S_k$ است. اگر $a_j \neq a_m$، فرض کنید مجموعه‌ی $A'_k = A_k - \{a_j\} \cup \{a_m\}$ همان مجموعه‌ی $A_k$ باشد که در آن $a_m$ را جایگزین $a_j$ کرده‌ایم. فعالیت‌های درون $A'_k$ سازگار هستند چرا که فعالیت‌های درون $A_k$ سازگار هستند، $a_j$ اولین فعالیتی در $A_k$ است که پایان می‌یابد، و $f_m \leq f_j$. چون $|A'_k| = |A_k|$، نتیجه می‌گیریم که $A'_k$ یک زیرمجموعه‌ی بیشینه از فعالیت‌های دوبه‌دو سازگار در $S_k$ است، حاوی $a_m$. ■

بنابراین می‌بینیم که با این که ممکن است بتوانیم مسئله‌ی انتخاب فعالیت را با برنامه‌ریزی پویا حل کنیم، نیازی به این کار نیست. (به علاوه، هنوز بررسی نکرده‌ایم که آیا مسئله‌ی انتخاب فعالیت زیرمسئله‌های تکراری دارد یا نه.) در عوض، می‌توانیم مکرراً فعالیتی را انتخاب کنیم که زودتر از بقیه پایان می‌یابد، فقط فعالیت‌هایی را نه داریم که با این فعالیت سازگار هستند، و این کار را تکرار کنیم تا دیگر هیچ فعالیتی باقی نماند. به علاوه چون همیشه فعالیتی را انتخاب می‌کنیم که کم‌ترین زمان پایان را دارد، زمان پایان فعالیت‌هایی که انتخاب می‌کنیم باید به ترتیب صعودی زمان پایان باشند.

الگوریتمی که مسئله‌ی انتخاب فعالیت را حل می‌کند، لازم نیست که مانند یک الگوریتم برنامه‌ریزی پویای بر مبنای جدول، از پایین به بالا کار کند. در عوض می‌تواند به صورت بالا به پایین کار کند، و در هر لحظه فعالیتی را برای جواب بهینه انتخاب کند، و سپس زیرمسئله‌ی باقی مانده را حل کند، که عبارت است از انتخاب از میان فعالیت‌هایی که با انتخاب‌های قبلاً انجام شده سازگار هستند. معمولاً الگوریتم‌های حریصانه همین طراحی بالا به پایین را دارند: انجام یک انتخاب حریصانه و سپس حل زیرمسئله، به جای تکنیک پایین به بالا و حل زیرمسئله‌ها قبل از انتخاب یک گزینه.

## یک الگوریتم حریصانه‌ی بازگشتی

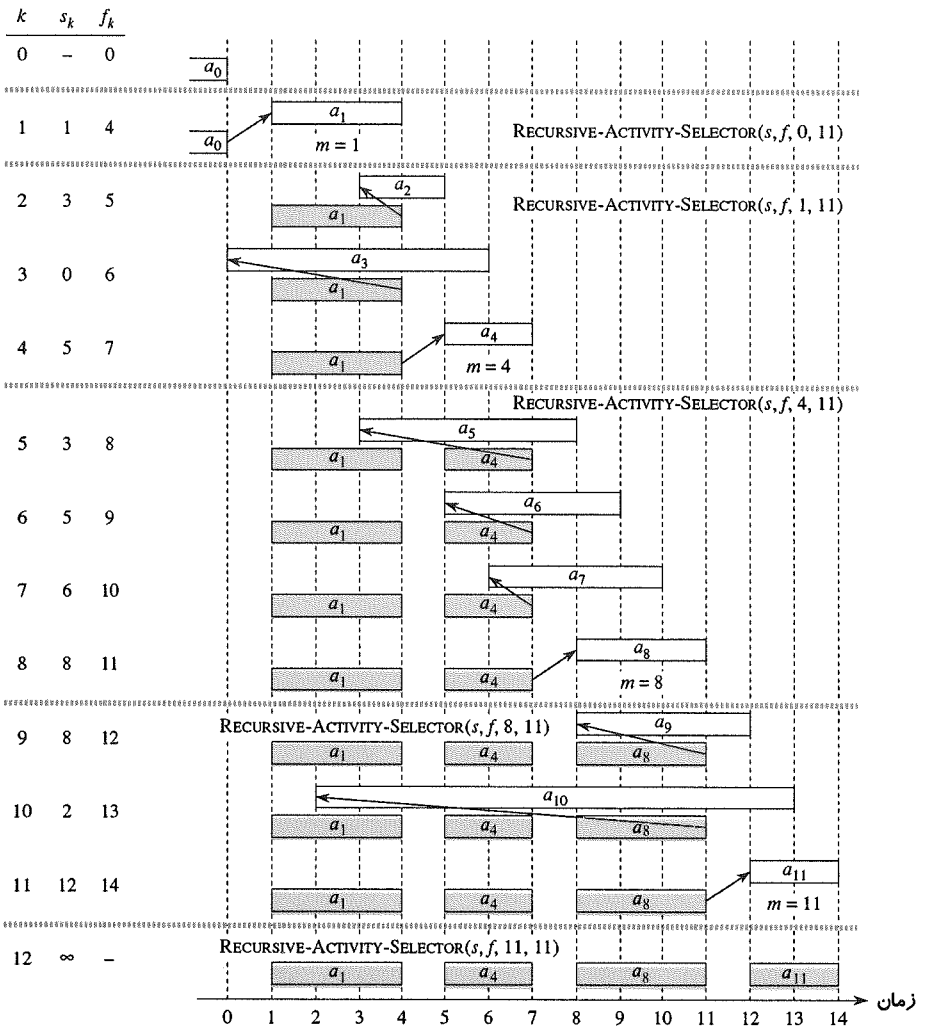
اکنون که دیدیم چگونه می‌توان به جای رویکرد برنامه‌ریزی پویا از یک الگوریتم حریصانه‌ی بالا به پایین استفاده کرد، می‌توانیم یک رویه‌ی بازگشتی سرراست برای حل مسئله‌ی انتخاب فعالیت بنویسیم. رویه‌ی RECURSIVE-ACTIVITY-SELECTOR زمان آغاز و پایان فعالیت‌ها را، که به صورت آرایه‌های $s$ و $f$[1] نشان داده می‌شوند، دریافت می‌کند، به همراه اندیس $k$ که زیرمسئله‌ی $S_k$ را نشان می‌دهد که باید حل شود، و $n$، اندازه‌ی مسئله‌ی اصلی. این رویه یک مجموعه‌ی با اندازه‌ی بیشینه از فعالیت‌های دوبه‌دو سازگار در $S_k$ را بازمی‌گرداند. فرض می‌کنیم که $n$ فعالیت ورودی از قبل به صورت صعودی بر حسب زمان پایان مرتب شده‌اند، طبق تساوی (۱۶-۱). اگر چنین نبود، می‌توانیم آن‌ها را در زمان $O(n \lg n)$ مرتب کنیم، و که در آن حالت‌های تساوی به دلخواه حل می‌شوند. برای آغاز، فعالیت مجازی $a_0$ با زمان پایان $f_0 = 0$ را به مجموعه اضافه می‌کنیم، تا زیرمسئله‌ی $S_0$ مجموعه‌ی کل فعالیت‌های $S$ باشد. فراخوانی اولیه، که تمام مسئله را حل می‌کند، عبارت است از RECUVSIVE-ACTIVITY-SELECTOR $(s, f, 0, n)$.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, i, j)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]      // Find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

شکل ۱۶-۱ عملیات الگوریتم را نشان می‌دهد. در یک فراخوانی RECURSIVE-ACTIVITY-SELECTOR $(s, f, k, n)$، حلقه‌ی while خطوط ۲-۳ به دنبال اولین فعالیت در $S_k$ می‌گردد. این حلقه $a_{k+1}, a_{k+2}, \ldots, a_n$ را بررسی می‌کند تا اولین فعالیت $a_m$ را که با $a_k$ سازگار است بیابد؛ برای چنین فعالیتی داریم $s_m \geq f_k$. اگر حلقه در اثر یافتن چنین فعالیتی پایان یابد، رویه در خط ۵ اجتماع $\{a_m\}$ و زیرمجموعه‌ی بیشینه‌ی $S_m$ را (که توسط فراخوانی بازگشتی RECURSIVE-ACTIVITY-SELECTOR $(s, f, m, n)$ به دست می‌آید) بازمی‌گرداند. در عوض، حلقه ممکن است در صورت برقراری رابطه‌ی $m > n$ پایان یابد، که در این صورت ما تمام فعالیت‌های درون $S_k$ را بررسی کرده‌ایم و هیچ کدام از آن‌ها با $a_i$ سازگار نبوده‌اند. در این صورت، $S_k = \emptyset$، و بنابراین رویه در خط ۶، $\emptyset$ را بازمی‌گرداند.

با فرض این که فعالیت‌ها قبلاً به ترتیب صعودی زمان پایان مرتب شده‌اند، زمان اجرای RECURSIVE-ACTIVITY-SELECTOR $(s, f, 0, n+1)$ برابر است با $\theta(n)$، که می‌توان به صورت زیر آن را نشان داد. در کل فراخوانی‌های بازگشتی هر فعالیت دقیقاً یک بار در حلقه‌ی while در خط ۲ بررسی می‌شود. به خصوص، فعالیت $a_k$ در آخرین فراخوانی که $k < i$ بررسی می‌شود.

[1] چون شبه‌کد $s$ و $f$ را به عنوان آرایه دریافت می‌کند، اندیس‌گذاری آن‌ها با کروشه انجام می‌شود، نه با زیرنویس.




**شکل ۱۶-۱** عملکرد RECURSIVE-ACTIVITY-SELECTOR بر روی ۱۱ فعالیتی که در بالا معرفی شده‌اند. فعالیت‌هایی که در هر فراخوانی بازگشتی در نظر گرفته می‌شوند، بین خطوط افقی مشخص شده‌اند. فعالیت ساختگی $a_0$ در زمان ۰ پایان می‌یابد، و در فراخوانی اولیه، یعنی RECURSIVE-ACTIVITY-SELECTOR $(s, f, ۰, ۱۲)$، فعالیت $a_1$ انتخاب می‌شود. در هر فراخوانی بازگشتی، فعالیت‌های سایه‌دار آن‌هایی هستند که قبلاً انتخاب شده‌اند، و فعالیت سفید، فعالیتی است که در همان مرحله در نظر گرفته می‌شود. اگر زمان شروع یک فعالیت قبل از اتمام آخرین فعالیت اضافه شده باشد (فلش بین آن‌ها به سمت چپ اشاره کند)، آن فعالیت رد می‌شود. در غیر این صورت (در صورتی که فلش دقیقاً به سمت بالا، و یا به سمت راست اشاره کند)، آن فعالیت انتخاب می‌شود. آخرین فراخوانی بازگشتی، یعنی RECURSIVE-ACTIVITY-SELECTOR $(s, f, ۱۱, ۱۲)$، $\emptyset$ را بازمی‌گرداند. مجموعه‌ی حاصل از فعالیت‌های انتخاب شده عبارت است از $\{a_1, a_4, a_8, a_{11}\}$.

## یک الگوریتم حریصانه‌ی تکراری

به سادگی می‌توانیم الگوریتم بازگشتی خود را به یک الگوریتم تکراری تبدیل کنیم. رویه‌ی RECURSIVE-ACTIVITY-SELECTOR تقریباً «بازگشتی دنباله‌ای» (tail recursive) است (مسئله‌ی ۷-۴ را ببینید): این رویه با یک فراخوانی بازگشتی به خود و به دنبال آن یک عملیات اجتماع پایان می‌یابد.

معمولاً تبدیل یک رویه‌ی بازگشتی دنباله‌ای به یک رویه‌ی تکراری کار سرراستی است؛ در واقع، بعضی از کامپایلرها برای بعضی زبان‌های برنامه‌نویسی خاص این کار را به صورت خودکار انجام می‌دهند. همان طور که گفته شد، RECURSIVE-ACTIVITY-SELECTOR برای زیرمسئله‌های $S_k$ کار می‌کند، یعنی زیرمسئله‌هایی که شامل آخرین فعالیت‌هایی هستند که تمام می‌شوند.

رویه‌ی GREEDY-ACTIVITY-SELECTOR یک نسخه‌ی تکراری از رویه‌ی RECURSIVE-ACTIVITY-SELECTOR است، که فرض می‌کند که فعالیت‌ها به ترتیب صعودی زمان پایان مرتب شده‌اند. این رویه فعالیت‌های انتخاب شده را در یک مجموعه‌ی $A$ ذخیره کرده، و پس از پایان این مجموعه را بازمی‌گرداند.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  i = 1
4  for m = 2 to n
5      if s[m] ≥ f[i]
6          A = A ∪ {a_m}
7          i = m
8  return A
```

این رویه به صورت زیر کار می‌کند. متغیر $i$ آخرین فعالیتی را نگه می‌دارد که به $A$ اضافه شده است، که متناظر است با فعالیت $a_k$ در نسخه‌ی بازگشتی. از آن جایی که فعالیت‌ها به ترتیب صعودی زمان پایان در نظر گرفته شده‌اند، $f_k$ همیشه بیشینه‌ی زمان‌های پایان در تمام فعالیت‌های $A$ است. یعنی،

$$f_k = \max\{f_i : a_i \in A\} \qquad (۳-۱۶)$$

در خطوط ۲-۳ فعالیت $a_1$ انتخاب می‌شود، $A$ طوری مقداردهی اولیه می‌شود که فقط شامل این فعالیت باشد، و $k$ برابر اندیس این فعالیت قرار می‌گیرد. حلقه‌ی for در خطوط ۴-۷ زود هنگام‌ترین فعالیت را در $S_k$ پیدا می‌کند. حلقه تمام فعالیت‌های $a_m$ را به ترتیب در نظر می‌گیرد و اولین فعالیتی را که با تمام فعالیت‌های قبل سازگار است، به مجموعه‌ی $A$ اضافه می‌کند؛ چنین فعالیتی، اولین فعالیتی در $S_k$ است که پایان می‌یابد. برای این که ببینیم فعالیت $a_m$ با تمام فعالیت‌هایی که هم اکنون در $A$ هستند سازگار است یا خیر، طبق تساوی (۱۶-۳) کافی است که چک کنیم (خط ۵) که زمان شروع آن، $s_m$، زودتر از $f_k$ (زمان پایان آخرین فعالیتی که به $A$ اضافه شده است) نباشد. اگر فعالیت $a_m$ سازگار باشد، آن گاه در خطوط ۶-۷ فعالیت $a_m$ به $A$ اضافه، و $k$ با $m$ مقداردهی می‌شود.

مجموعه‌ی $A$ که توسط GREEDY-ACTIVITY-SELECTOR $(s, f)$ بازگردانده می‌شود، دقیقاً مجموعه‌ای است که RECURSIVE-ACTIVITY-SELECTOR $(s, f, 0, n)$ بازمی‌گرداند.

GREEDY-ACTIVITY-SELECTOR مانند نسخه‌ی بازگشتی، برنامه‌ریزی مجموعه‌ای از $n$ فعالیت را در زمان $\theta(n)$ انجام می‌دهد، با فرض این که فعالیت‌ها قبلاً به ترتیب صعودی زمان پایان مرتب شده باشند.

## تمرین‌ها

۱-۱-۱۶ یک الگوریتم برنامه‌ریزی پویا برای مسئله‌ی انتخاب فعالیت بر مبنای رابطه‌ی بازگشتی (۱۶-۲) ارائه کنید. الگوریتم شما باید اندازه‌ی $c[i, j]$ را همان طور که در بالا تعیین شد، محاسبه، و زیرمجموعه‌ی بیشینه از فعالیت‌های دوبه‌دو سازگار را تولید کند. فرض کنید که ورودی‌ها در تساوی (۱۶-۱) مرتب شده‌اند. زمان اجرای راه حل خود را با زمان اجرای GREEDY-ACTIVITY-SELECTOR مقایسه کنید.

۲-۱-۱۶ فرض کنید که به جای این که همیشه اولین فعالیتی را انتخاب کنیم پایان می‌یابد، فعالیتی را انتخاب می‌کنیم زمان شروع آن از همه بیشتر است، و با تمام فعالیت‌هایی که قبلاً انتخاب شده‌اند، سازگار است. توضیح دهید که چرا این رویکرد، حریصانه است، و اثبات کنید که به یک جواب بهینه ختم می‌شود.

۳-۱-۱۶ تمام رویکردهای حریصانه نمی‌توانند برای مسئله‌ی انتخاب فعالیت یک مجموعه‌ی بیشینه از فعالیت‌های دوبه‌دو سازگار بیابند. مثالی بیاورید که نشان دهد انتخاب کوتاه‌ترین فعالیتی که با تمام فعالیت‌های قبلی سازگار است، به جواب بهینه ختم نمی‌شود. همین کار را برای روش انتخاب فعالیتی که با کم‌ترین فعالیت‌ها ناسازگاری دارد، و روش انتخاب فعالیت با کم‌ترین زمان شروع انجام دهید.

۴-۱-۱۶ فرض کنید که یک مجموعه از فعالیت‌ها داریم که باید برای تعداد زیادی سالن سخنرانی برای آن‌ها برنامه‌ریزی کنیم، که در آن هر فعالیتی می‌تواند در هر سالن سخنرانی برگذار شود. می‌خواهیم با کم‌ترین تعداد سالن‌های سخنرانی ممکن، برای تمام فعالیت‌ها برنامه‌ریزی کنیم. یک الگوریتم حریصانه‌ی کارا ارائه کنید که تعیین می‌کند کدام فعالیت باید از کدام سالن سخنرانی استفاده کند.

(این مسئله با نام *مسئله‌ی رنگ‌آمیزی گراف بازه‌ای* (interval-graph coloring problem) نیز معروف است. می‌توانیم یک گراف بازه‌ای بسازیم که رأس‌های آن فعالیت‌ها هستند، و یال‌ها فعالیت‌های ناسازگار را به یکدیگر متصل می‌کنند. کم‌ترین تعداد رنگ‌های مورد نیاز برای رنگ‌آمیزی تمام رأس‌ها به طوری که هیچ دو رأس کنار مجاوری هم رنگ نباشند، متناظر است با یافتن کم‌ترین تعداد سالن سخنرانی مورد نیاز برای برگزاری تمام فعالیت‌های داده شده.)

۵-۱-۱۶ یک تغییر در مسئله‌ی انتخاب فعالیت را در نظر بگیرید، که در آن هر فعالیت $a_i$، علاوه بر زمان شروع و زمان پایان، یک ارزش $v_i$ هم دارد. هدف دیگر بیشینه کردن تعداد فعالیت‌ها نیست، بلکه بیشینه کردن کل ارزش فعالیت‌های انجام شده است. یعنی، می‌خواهیم یک مجموعه‌ی $A$ از فعالیت‌های سازگار انتخاب کنیم به طوری که $\sum_{a_k \in A} v_k$ بیشینه شود. یک الگوریتم با زمان چندجمله‌ای برای این مسئله ارائه کنید.

## ۲-۱۶ عناصر استراتژی حریصانه

یک الگوریتم حریصانه با استفاده از دنباله‌ای از انتخاب‌ها، یک جواب بهینه برای یک مسئله می‌یابد. برای هر نقطه‌ی تصمیم‌گیری در الگوریتم، گزینه‌ای که در همان لحظه بهترین به نظر می‌آید انتخاب می‌شود. این استراتژی مکاشفه‌ای همیشه یک جواب بهینه تولید نمی‌کند، ولی همان طور که در مسئله‌ی انتخاب فعالیت‌ها دیدیم، بعضی مواقع این کار را می‌کند. در این بخش بعضی از خصوصیات کلی متدهای حریصانه مورد بررسی قرار می‌گیرند.

فرآیندی که در بخش ۱-۱۶ برای توسعه‌ی یک الگوریتم حریصانه دنبال کردیم، کمی بیشتر از حالت کلی بود. ما این مراحل را طی کردیم:

۱. تعیین زیرساختار بهینه‌ی مسئله.
۲. توسعه‌ی یک راه حل بازگشتی. (برای مسئله‌ی انتخاب فعالیت، رابطه‌ی بازگشتی (۱۶-۲) را ارائه کردیم، ولی از توسعه‌ی یک الگوریتم بازگشتی بر مبنای این رابطه صرف نظر کردیم.)
۳. نشان دادن این که اگر انتخاب حریصانه را انجام دهیم، فقط یک زیرمسئله باقی می‌ماند.
۴. اثبات این که انتخاب حریصانه همیشه گزینه‌ی مناسبی است. (مراحل ۳ و ۴ را می‌توان به ترتیب دلخواه انجام داد.)
۵. توسعه‌ی یک الگوریتم بازگشتی که استراتژی حریصانه را پیاده‌سازی می‌کند.
۶. تبدیل الگوریتم بازگشتی به یک الگوریتم تکراری.

طی این مراحل، جزئیات زیربنای مربوط به برنامه‌ریزی پویا را در یک الگوریتم حریصانه دیدیم. به عنوان مثال در مسئله‌ی انتخاب فعالیت، ابتدا زیرمسئله‌های $S_{ij}$ را تعریف کردیم، که در آن هر دوی $i$ و $j$ تغییر می‌کردند. سپس فهمیدیم که اگر همیشه انتخاب حریصانه را انجام دهیم، می‌توانیم زیرمسئله‌های خود را به آن‌هایی محدود کنیم که به شکل $S_k$ هستند.

از طرف دیگر، می‌توانستیم زیرساختار بهینه را با یک انتخاب حریصانه در ذهن خود شکل دهیم، به طوری که فقط یک زیرمسئله برای حل باقی بماند. در مسئله‌ی انتخاب فعالیت، می‌توانستیم با حذف اندیس دوم آغاز کرده و زیرمسئله‌ها را به شکل $S_k$ تعریف کنیم. سپس، می‌توانستیم اثبات کنیم که ترکیب یک انتخاب حریصانه (اولین فعالیت $a_m$ که در بین فعالیت‌های $S_k$ پایان می‌یابد) با یک

جواب بهینه‌ی مجموعه‌ی باقی مانده‌ی $S_m$ از فعالیت‌های سازگار، یک جواب بهینه برای $S_k$ به ما می‌دهد. به طور کلی‌تر، مراحل طراحی یک الگوریتم حریصانه به صورت زیر است:

۱. تبدیل مسئله‌ی بهینه‌سازی به مسئله‌ای که در آن یک انتخاب انجام می‌دهیم و بعد از آن یک مسئله برای حل کردن باقی می‌ماند.
۲. اثبات این که همیشه یک جواب بهینه به مسئله‌ی کلی وجود دارد که انتخاب حریصانه را انجام می‌دهد، و بنابراین انتخاب حریصانه همیشه می‌تواند به جواب بهینه ختم شود.
۳. توصیف زیرساختار بهینه با نشان دادن این که با انجام انتخاب حریصانه، چیزی که باقی می‌ماند یک زیرمسئله با این خصوصیت است که اگر یک جواب بهینه به آن را با انتخاب حریصانه‌ای که انجام دادیم، ترکیب کنیم، به یک جواب بهینه به مسئله‌ی اصلی می‌رسیم.

در بخش‌های بعدی این فصل از این روند ساده‌تر استفاده می‌کنیم. با این وجود، پشت هر الگوریتم حریصانه، تقریباً همیشه یک راه حل برنامه‌ریزی پویای سخت‌تر وجود دارد.

چطور می‌توان تعیین کرد که آیا می‌توان با یک الگوریتم حریصانه، یک مسئله‌ی بهینه‌سازی خاص را حل کرد یا نه؟ در حالت کلی راه حلی برای این کار وجود ندارد، ولی دو عنصر اصلی، خصوصیت انتخاب حریصانه و زیرساختار بهینه هستند. اگر بتوانیم نشان دهیم که مسئله این خصوصیات را دارد، در این صورت با احتمال زیاد می‌توانیم یک الگوریتم حریصانه برای آن توسعه دهیم.

## خصوصیت انتخاب حریصانه

اولین عنصر اصلی *خصوصیت انتخاب حریصانه* است: با انتخاب‌های بهینه‌ی محلی (حریصانه) می‌توان یک جواب بهینه‌ی کلی بهینه تولید کرد. به عبارت دیگر، وقتی می‌خواهیم تصمیم بگیریم که کدام گزینه را انتخاب کنیم، انتخابی را انجام می‌دهیم که در مسئله‌ی فعلی از همه بهتر به نظر بیاید، بدون این که نتیجه‌ی این کار را برای زیرمسئله‌ها در نظر بگیریم.

در این جا تفاوت میان الگوریتم‌های حریصانه با برنامه‌ریزی پویا مشخص می‌شود. در برنامه‌ریزی پویا در هر مرحله یک انتخاب انجام می‌دهیم، ولی این انتخاب به جواب زیرمسئله‌ها بستگی دارد. در نتیجه، معمولاً مسائل برنامه‌ریزی پویا را به روش پایین به بالا حل می‌کنیم، که از مسائل کوچک‌تر شروع شده و با مسائل بزرگ‌تر پایان می‌یابد. (همچنین، می‌توانیم آن‌ها را به صورت بالا به پایین و با استفاده از به خاطر سپاری حل کنیم. البته با این که کد به صورت بالا به پایین کار می‌کند، باز هم باید زیرمسئله‌های کوچک‌تر قبل از زیرمسئله‌های بزرگ‌تر حل شوند.) در الگوریتم‌های حریصانه، انتخابی را انجام می‌دهیم که در همان لحظه بهترین به نظر می‌رسد و زیرمسئله‌ای را که بعد از انجام انتخاب به وجود می‌آید، حل می‌کنیم. انتخابی که در الگوریتم‌های حریصانه انجام می‌شود ممکن است به انتخاب‌های قبل از این بستگی داشته باشد، ولی نمی‌تواند به انتخاب‌های آینده و یا جواب زیرمسئله‌ها وابسته باشد. بنابراین برخلاف برنامه‌ریزی پویا که زیرمسئله‌ها را به روش از پایین به بالا حل می‌کند، معمولاً یک استراتژی حریصانه به روش بالا به پایین عمل می‌کند، و یکی بعد از دیگری انتخاب‌های

حریصانه انجام داده و هر نمونه‌ی مسئله را به نمونه‌ای کوچک‌تر تبدیل می‌کند.

مسلماً باید اثبات کنیم که یک انتخاب حریصانه در هر مرحله به یک جواب بهینه برای کل مسئله ختم می‌شود. معمولاً مانند قضیه‌ی ۱۶-۱، در اثبات، یک جواب بهینه‌ی کلی برای یک زیرمسئله بررسی می‌شود. سپس نشان داده می‌شود که می‌توان جواب را طوری اصلاح کرد که از انتخاب حریصانه استفاده کند، که به یک زیرمسئله‌ی مشابه ولی کوچک‌تر ختم می‌شود.

معمولاً خصوصیت انتخاب حریصانه در انجام انتخاب در زیرمسئله‌ها مقداری کارایی را بالا می‌برد. مثلاً در مسئله‌ی انتخاب فعالیت، با فرض این که فعالیت‌ها قبلاً به ترتیب صعودی زمان پایان مرتب شده‌اند، نیاز داشتیم که هر فعالیت را فقط یک بار بررسی کنیم. معمولاً این گونه است که با پیش پردازش ورودی و یا با استفاده از یک ساختمان داده‌ی مناسب (معمولاً یک صف اولویت)، می‌توانیم به سرعت انتخاب‌های حریصانه را انجام دهیم، که باعث کارایی الگوریتم می‌شود.

## زیرساختار بهینه

یک مسئله دارای *زیرساختار بهینه* است اگر یک جواب بهینه به مسئله در خود شامل جواب‌های بهینه به زیرمسئله‌ها باشد. این خصوصیت یک عنصر اصلی برای تعیین قابلیت کاربرد برنامه‌ریزی پویا و الگوریتم‌های حریصانه برای یک مسئله است. به عنوان یک مثال از زیرساختار بهینه، به یاد بیاورید که چگونه در بخش ۱۶-۱ نشان دادیم که اگر یک جواب بهینه به زیرمسئله‌ی $S_{ij}$ شامل فعالیت $a_k$ باشد، آن گاه باید حاوی جواب‌های بهینه‌ای به زیرمسئله‌های $S_{ik}$ و $S_{kj}$ هم باشد. با داشتن این زیرساختار بهینه، بحث کردیم که اگر می‌دانستیم از کدام فعالیت به عنوان $a_k$ استفاده کنیم، می‌توانستیم با انتخاب $a_k$ به همراه جواب‌های بهینه‌ی مسائل $S_{ik}$ و $S_{kj}$، یک جواب بهینه برای مسئله‌ی $S_{ij}$ بسازیم. بر مبنای این مشاهدات از زیرساختار بهینه توانستیم رابطه‌ی بازگشتی (۱۶-۲) را به دست آوریم، که مقدار یک جواب بهینه را تعیین می‌کند.

معمولاً هنگام کار با زیرساختار بهینه برای استفاده در الگوریتم‌های حریصانه از رویکرد ساده‌تری استفاده می‌کنیم. همان طور که در بالا گفته شد، ما این فرض تجملاتی را داریم که با یک انتخاب حریصانه در مسئله‌ی اصلی به یک زیرمسئله رسیده‌ایم. ولی تمام چیزی که به آن نیاز داریم این است که اثبات کنیم که ترکیب یک جواب بهینه به زیرمسئله با انتخاب حریصانه به یک جواب بهینه به مسئله‌ی کلی ختم می‌شود. این روش به طور ضمنی از استقرا بر روی زیرمسئله‌ها استفاده می‌کند تا اثبات کند که حاصل انتخاب حریصانه در هر مرحله، یک جواب بهینه است.

## الگوریتم‌های حریصانه در مقابل برنامه‌ریزی پویا

از آن جایی که در هر دو استراتژی حریصانه و برنامه‌ریزی پویا از زیرساختار بهینه استفاده می‌شود، بعضی مواقع ممکن است برای مسئله‌ای که یک الگوریتم حریصانه کفایت می‌کند، از برنامه‌ریزی پویا استفاده کنیم، و یا زمانی که به برنامه‌ریزی پویا نیاز داریم، از یک الگوریتم حریصانه استفاده کنیم. برای تعیین تفاوت‌های ریز میان این دو تکنیک، اجازه دهید دو نسخه از یک مسئله‌ی بهینه‌سازی معروف را

بررسی کنیم.

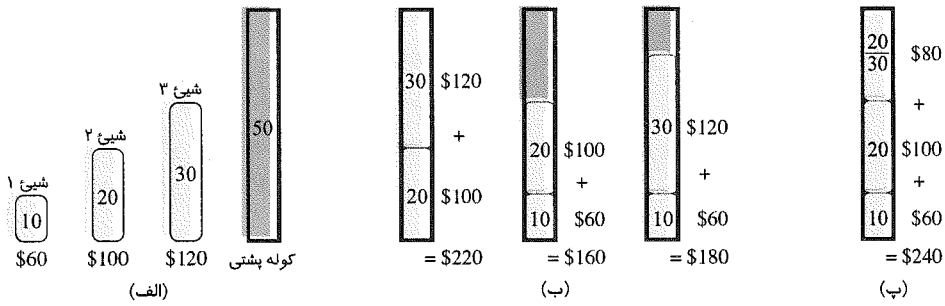
*مسئله‌ی کوله پشتی ۰-۱* (0-1 knapsack problem) به صورت زیر است. یک دزد که در حال دزدی از یک مغازه است، $n$ فقره جنس پیدا می‌کند؛ قیمت جنس $i$ام $v_i$ دلار و وزن آن $w_i$ پوند است، که در آن $v_i$ و $w_i$ اعداد صحیح هستند. او می‌خواهد با ارزش‌ترین بار ممکن را بردارد، ولی کوله پشتی فقط به اندازه‌ی $W$ پوند فضا دارد، که $W$ یک عدد صحیح است. او باید کدام اجناس را بردارد؟ (از این رو به این مسئله، کوله پشتی ۰-۱ گفته می‌شود که در آن هر جنس یا برداشته می‌شود و یا خیر؛ دزد نمی‌تواند قسمتی از یکی از اجناس را بردارد، و یا یکی از اجناس را دو بار بردارد.)

در *مسئله‌ی کوله پشتی کسری* (fractional knapsack problem)، صورت مسئله یکی است، ولی دزد می‌تواند کسری از اجناس را بردارد، و مجبور نیست که برای هر جنس یک انتخاب دودویی (۰-۱) انجام دهد. می‌توانید در مسئله‌ی کوله پشتی ۰-۱، اجناس را به صورت شمش‌های طلا فرض کنید، و در مسئله‌ی کوله پشتی کسری به صورت گرد طلا.

هر دو مسئله‌ی کوله پشتی دارای زیرساختار بهینه هستند. برای مسئله‌ی ۰-۱، با ارزش‌ترین باری را در نظر بگیرید که حداکثر $W$ پوند وزن‌دارد. اگر جنس $j$ام را از این بار حذف کنیم، بار باقی‌مانده باید با ارزش‌ترین باری باشد که حداکثر $W - w_j$ وزن دارد و دزد می‌تواند از میان $n-1$ جنس باقی مانده، غیر ازجنس $j$ام، هر کدام را بردارد. برای مسئله‌ی کسری، فرض کنید که به مقدار $w$ پوند از جنس $j$ام را از بار حذف کنیم، بنابراین بار باقی مانده باید با ارزش‌ترین باری باشد که حداکثر $W - w$ وزن دارد و دزد می‌تواند از $n-1$ جنس باقی مانده به علاوه‌ی $w_j - w$ پوند از جنس $j$ام بردارد.

با این که دو مسئله مشابه هستند، ولی مسئله‌ی کوله پشتی کسری با یک استراتژی حریصانه قابل حل است، در حالی که برای مسئله‌ی ۰-۱ این طور نیست. برای حل مسئله‌ی کسری، ابتدا مقدار ارزش بر وزن $v_i/w_i$ را برای هر جنس محاسبه می‌کنیم. با دنبال کردن استراتژی حریصانه، دزد با برداشتن بیشترین مقدار ممکن از جنس با بالاترین ارزش بر وزن شروع می‌کند. اگر باز هم جای خالی برای اجناس دیگر داشت، در حد ممکن از جنس بعدی (که هم اکنون بیشترین ارزش بر وزن را دارد) برمی‌دارد، و همین طور ادامه می‌دهد تا دیگر نتواند جنس دیگری بردارد. بنابراین، با مرتب کردن اجناس به ترتیب مقدار ارزش بر وزن، الگوریتم حریصانه در زمان $O(n \lg n)$ اجرا می‌شود. اثبات این که مسئله‌ی کوله پشتی کسری دارای خصوصیت حریصانه است به عنوان تمرین ۱۶-۲-۱ به خواننده واگذار شده است.

برای این که ببینیم این استراتژی حریصانه برای مسئله‌ی کوله پشتی ۰-۱ کار نمی‌کند، نمونه‌ی مسئله‌ی نشان داده شده در شکل ۱۶-۲ (الف) را در نظر بگیرید. سه جنس وجود دارد و کوله پشتی می‌تواند ۵۰ پوند را در خود جای دهد. جنس ۱، ۱۰ پوند وزن و ۶۰ دلار ارزش دارد. جنس ۲، ۲۰ پوند وزن و ۱۰۰ دلار ارزش، و جنس ۳، ۳۰ پوند وزن و ۱۲۰ دلار ارزش دارد. بنابراین، مقدار ارزش بر وزن جنس ۱، ۶ دلار بر پوند است، که از مقدار ارزش بر وزن جنس ۲ (۵ دلار بر پوند) و یا جنس ۳ (۴ دلار بر پوند) بیشتر است. بنابراین، در استراتژی حریصانه ابتدا جنس ۱ برداشته می‌شود. با این حال




**شکل ۱۶-۲** یک مثال که نشان می‌دهد استراتژی حریصانه برای مسئله‌ی کوله پشتی ۰-۱ کار نمی‌کند. (الف) دزد باید زیرمجموعه‌ای از سه جنس نشان داده شده را بردارد، و وزن این زیرمجموعه نباید از ۵۰ پوند بیشتر شود. (ب) زیرمجموعه‌ی بهینه حاوی اجناس ۲ و ۳ است. هر جوابی که شامل جنس ۱ باشد، نابهینه است، با این که جنس ۱ دارای بالاترین ارزش بر وزن است. (پ) برای مسئله‌ی کوله پشتی کسری، برداشتن اجناس به ترتیب بیشترین مقدار ارزش بر وزن به جواب بهینه ختم می‌شود.

همان طور که در تحلیل شکل ۱۶-۲ (ب) دیده می‌شود، در جواب بهینه اجناس ۲ و ۳ برداشته، و جنس ۱ باقی گذاشته می‌شود. دو جواب ممکن که در آن‌ها از جنس ۱ استفاده می‌شود، هر دو نابهینه هستند.

با این حال، در همین نمونه برای مسئله‌ی کسری، استراتژی حریصانه که ابتدا جنس ۱ را برمی‌دارد به یک جواب بهینه ختم می‌شود، همان طور که در شکل ۱۶-۲ (پ) نشان داده شده است. برداشتن جنس ۱ در مسئله‌ی ۰-۱ کار نمی‌کند، چرا که دزد نمی‌تواند تمام ظرفیت کوله پشتی خود را پر کند، و فضای خالی باعث می‌شود که مقدار ارزش بر وزن مؤثر بار آن پایین بیاید. در مسئله‌ی ۰-۱، وقتی یک جنس را برای قرار دادن در کوله پشتی در نظر می‌گیریم، باید قبل از انتخاب زیرمسئله‌ای را که در آن جنس برداشته شده است، با زیرمسئله‌ای که در آن جنس برداشته نشده است، مقایسه کنیم. چنین راه حلی به تعداد زیادی زیرمسئله‌ی مشترک ختم می‌شود — یک نشانه‌ی برنامه‌ریزی پویا، و در واقع، همان طور که تمرین ۱۶-۲-۲ از شما می‌خواهد نشان دهید، می‌توان از برنامه‌ریزی پویا برای حل مسئله‌ی ۰-۱ استفاده کرد.

## تمرین‌ها

**۱۶-۲-۱** اثبات کنید که مسئله‌ی کوله پشتی کسری دارای خصوصیت انتخاب حریصانه است.

**۱۶-۲-۲** یک جواب برنامه‌ریزی پویا برای مسئله‌ی کوله پشتی ۰-۱ بدهید که در زمان $O(nW)$ اجرا می‌شود، که در آن $n$ تعداد اجناس موجود و $W$ بیشینه‌ی وزنی است که دزد می‌تواند با خود ببرد.

**۱۶-۲-۳** فرض کنید که در یک مسئله‌ی کوله پشتی ۰-۱، ترتیب اجناس وقتی به ترتیب افزایشی وزن مرتب می‌شوند، برابر است با ترتیب آن‌ها وقتی به ترتیب نزولی ارزش مرتب می‌شوند. یک الگوریتم کارا ارائه کنید که یک جواب بهینه برای این نسخه از مسئله‌ی کوله

پشتی بیابید، و بحث کنید که جواب شما صحیح است.

۴-۲-۱۶ پروفسور Gecko همیشه آرزو داشته است که عرض داکوتای شمالی را با اسکیت طی کند. او تصمیم می‌گیرد که از طریق بزرگراه U.S. 2 از ایالت عبور کند، که از گرند فورکز، در مرز شرقی با مینسوتا آغاز شده، و تا ویلیستون، نزدیک مرز غربی با مونتانا پایان می‌یابد. پروفسور می‌تواند دو لیتر آب با خود حمل کند، و با این دو لیتر حداکثر می‌تواند $m$ مایل را با اسکیت طی کند. (چون تمام داکوتای شمالی تقریباً هم‌سطح است، پروفسور نیازی به نگرانی در مورد مصرف بیشتر آب در سربالایی نسبت به سرپایینی یا مسیر صاف ندارد.) پروفسور از گرند فورکز با دو لیتر آب آغاز می‌کند. نقشه‌ی او موقعیت تمام مکان‌های بزرگراه U.S. 2 را که او می‌تواند آب خود را پر کند، به همراه فاصله‌ی آن‌ها نشان می‌دهد.

هدف پروفسور این است که تعداد توقف‌های خود در مسیر عبور از ایالت برای آب را حداقل کند. یک متد کارامد ارائه کنید که پروفسور بتواند به کمک آن تصمیم بگیرد که در کدام ایستگاه‌های آب باید توقف کند. اثبات کنید که استراتژی شما به یک جواب بهینه ختم می‌شود، و زمان اجرای آن را تحلیل کنید.

۵-۲-۱۶ یک الگوریتم کارا ارائه کنید که با دریافت یک مجموعه‌ی $\{x_1, x_2, \ldots, x_n\}$ از نقاط بر روی خط حقیقی، کوچک‌ترین مجموعه از بازه‌های بسته‌ی واحد را تعیین می‌کند که شامل تمام نقاط داده شده در مجموعه می‌شود. اثبات کنید که الگوریتم شما صحیح است.

★ ۶-۲-۱۶ نشان دهید که چگونه می‌توان مسئله‌ی کوله پشتی کسری را در زمان $O(n)$ حل کرد. فرض کنید که یک جواب برای مسئله‌ی ۹-۲ دارید.

۷-۲-۱۶ فرض کنید که دو مجموعه‌ی $A$ و $B$ به شما داده شده است، که هر کدام حاوی $n$ عدد صحیح هستند. شما می‌توانید این دو مجموعه را به هر شکلی که می‌خواهید دوباره مرتب کنید. بعد از مرتب‌سازی دوباره، فرض کنید $a_i$ عنصر $i$ام در مجموعه‌ی $A$، و $b_i$ عنصر $i$ام در مجموعه‌ی $B$ باشد. شما می‌توانید پس از این کار، مقدار $\prod_{i=1}^{n} a_i^{b_i}$ ارزش دریافت کنید. یک الگوریتم بدهید که ارزش دریافتی شما را بیشینه می‌کند. اثبات کنید که الگوریتم شما ارزش دریافتی را بیشینه می‌کند، و زمان اجرای آن را مشخص کنید.

## ۳-۱۶ کدهای هافمن

کد هافمن یک تکنیک بسیار پر کاربرد و بهینه برای فشرده کردن داده است؛ به طور معمول، صرفه‌جویی ۲۰ درصدی تا ۹۰ درصدی در حافظه، بسته به نوع داده‌های مورد استفاده. ما در این جا داده‌ها را به صورت دنباله‌ای از کاراکترها در نظر می‌گیریم. الگوریتم حریصانه‌ی هافمن از جدول

فراوانی کاراکترها استفاده می‌کند تا یک روش بهینه برای نمایش هر یک از کاراکترها به صورت یک رشته‌ی دودویی به دست آورد.

فرض کنید که یک فایل داده حاوی ۱۰۰،۰۰۰ کاراکتر داریم و می‌خواهیم آن را به صورت فشرده ذخیره کنیم. مشاهده می‌کنیم که فراوانی کاراکترهای درون فایل به صورت شکل ۱۶-۳ است. یعنی فقط ۶ کاراکتر وجود دارد، و ۴۵،۰۰۰ نمونه از کاراکتر 'a' در فایل وجود دارد.

راه‌های بسیاری برای نشان دادن چنین فایلی از اطلاعات وجود دارد. ما مسئله‌ی طراحی یک *کد کاراکتر دودویی* (یا به اختصار *کد*) را در نظر می‌گیریم که در آن هر کاراکتر با یک رشته‌ی دودویی یکتا نمایش داده می‌شود. اگر از یک *کد با طول ثابت* استفاده کنیم، ۳ بیت برای نمایش ۶ کاراکتر نیاز داریم: $a = ۰۰۰$، $b = ۰۰۱$،... ، $f = ۱۰۱$. این متد به ۳۰۰،۰۰۰ بیت برای کد کردن کل فایل نیاز دارد. آیا می‌توان بهتر از این هم عمل کرد؟

یک *کد با طول متغیر* می‌تواند به مقدار قابل ملاحظه‌ای بهتر از یک کد با طول ثابت عمل کند، بدین صورت که به کاراکترهای پرکاربرد کدهای کوتاه، و به کاراکترهای کم کاربرد کدهای بلند می‌دهد. شکل ۱۶-۳ چنین کدی را نشان می‌دهد؛ در این جا، رشته‌ی یک بیتی ۰ نشان‌دهنده‌ی کاراکتر 'a' است، و رشته‌ی ۴ بیتی ۱۱۰۰ نشان دهنده‌ی کاراکتر 'f'. این کد به

$$(۴۵\times۱+۱۳\times۳+۱۲\times۳+۱۶\times۳+۹\times۴+۵\times۴)\times۱,۰۰۰=۲۲۴,۰۰۰$$

بیت برای نمایش فایل نیاز دارد، که حدود ۲۵٪ در حافظه صرفه‌جویی می‌شود. در واقع همان طور که بعداً خواهیم دید، این یک کد بهینه برای این فایل است.

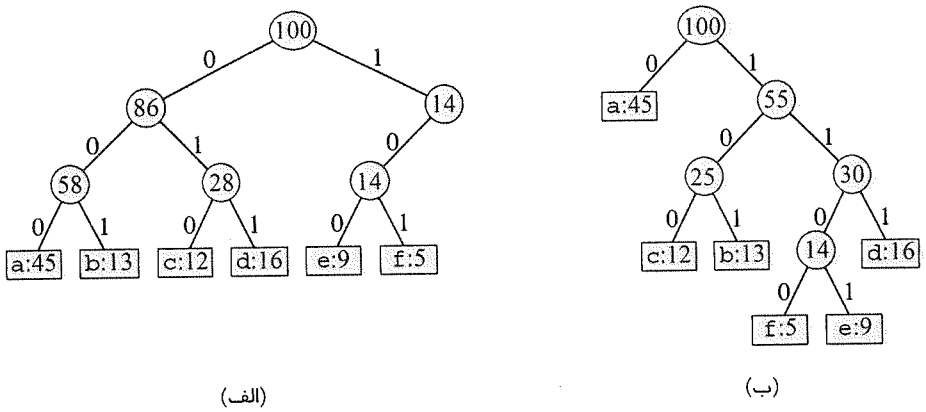
## کدهای پیشوندی

ما در این جا فقط کدهایی را در نظر می‌گیریم که هیچ کد کلمه‌ای پیشوند کد کلمه‌ی دیگر نیست. به چنین کدهایی، *کدهای پیشوندی*[1] (prefix codex) گفته می‌شود. می‌توان نشان داد (با این که ما در این جا نشان نمی‌دهیم) که فشرده‌سازی داده‌ی بهینه که با کد کلمه قابل دستیابی است همیشه می‌تواند با کدهای پیشوندی هم قابل دستیابی باشد، و با قرار دادن محدودیت استفاده از کدهای پیشوندی، کلیت مسئله به هیچ وجه از دست نمی‌رود.

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| فرکانس (تقسیم بر هزار) | ۴۵ | ۱۳ | ۱۲ | ۱۶ | ۹ | ۵ |
| کد کلمه با طول ثابت | ۰۰۰ | ۰۰۱ | ۰۱۰ | ۰۱۱ | ۱۰۰ | ۱۰۱ |
| کد کلمه با طول متغیر | ۰ | ۱۰۱ | ۱۰۰ | ۱۱۱ | ۱۱۰۱ | ۱۱۰۰ |

**شکل ۱۶-۳** یک مسئله‌ی کد کردن کاراکترها. یک فایل داده با ۱۰۰،۰۰۰ کاراکتر که فقط حاوی کاراکترهای a-f است، که فراوانی آن‌ها داده شده است. اگر به هر کاراکتر یک کد ۳ بیتی داده شود، فایل را می‌توان با ۳۰۰،۰۰۰ بیت کد کرد. با استفاده از کد با طول متغیر نشان داده شده، با فایل ۲۲۴،۰۰۰ بیت کد می‌شود.

[1] احتمالاً «کدهای بدون پیشوند (prefix-free codes)» نام بهتری است، ولی اصطلاح «کدهای پیشوندی» در ادبیات الگوریتم‌ها استاندارد است.




**شکل ۱۶-۴** درخت‌های مربوط به روش‌های کد گذاری شکل ۱۶-۳. هر برگ با یک کاراکتر و فراوانی آن برچسب شده است. برچسب هر گره‌ی داخلی، مجموع فراوانی‌های برگ‌های زیردرخت آن گره است. (الف) درخت مربوط به کد با طول ثابت $a = ۰۰۰$، ...، $f = ۱۰۱$. (ب) درخت مربوط به کد پیشوندی بهینه‌ی $a = ۰$، $b = ۱۰۱$،...، $f = ۱۱۰۰$.

کد کردن همیشه برای هر کد کاراکتر دودویی بسیار ساده است؛ فقط کافی است که کد کلمه‌های نشان‌دهنده‌ی هر کاراکتر را به یکدیگر متصل کنیم. به عنوان مثال، با کدهای پیشوندی با طول متغیر نشان داده شده در شکل ۱۶-۳، فایل ۳ کاراکتری abc را به صورت ۰.۱۰۱.۱۰۰ = ۰۱۰۱۱۰۰ کد می‌کنیم، که در آن از «.» برای نشان دادن اتصال استفاده کرده‌ایم.

کدهای پیشوندی از این رو مناسب هستند که رمزگشایی را آسان می‌کنند. از آن جایی که هیچ کد کلمه‌ای پیشوند کد کلمه‌ی دیگری نیست، کد کلمه‌ای که در شروع یک فایل کد شده قرار دارد نامبهم است. برای رمز گشایی، می‌توانیم به سادگی کد کلمه‌ی اول را شناسایی کنیم، آن را به کاراکتر اصلی ترجمه کرده، و فرآیند رمزگشایی را بر روی بقیه‌ی فایل ادامه دهیم. در مثال بالا، رشته‌ی ۰۰۱۰۱۱۱۰۱ به صورت یکتا به شکل ۰.۰.۱۰۱.۱۱۰۱ تجزیه می‌شود، که رمزگشایی شده‌ی آن به صورت aabe است.

فرآیند رمزگشایی به یک نمایش مناسب برای کد پیشوندی نیاز دارد تا بتوان کد کلمه‌ی اول را به راحتی تشخیص داد. یک درخت دودویی که برگ‌های آن کاراکترهای متن هستند، چنین نمایشی را فراهم می‌کند. کد کلمه‌ی دودویی یک کاراکتر را به صورت مسیر طی شده از ریشه به آن کاراکتر ترجمه می‌کنیم، که در آن ۰ یعنی «برو به فرزند چپ» و ۱ یعنی «برو به فرزند راست». شکل ۱۶-۴ درخت کدهای مثال بالا را نشان می‌دهد. توجه کنید که این درخت‌ها، درخت‌های جستجوی دودویی نیستند، چرا که نیازی نیست که برگ‌ها به صورت مرتب قرار گیرند و گره‌های داخلی حاوی کلیدهای کاراکتری نیستند.

یک کد بهینه برای یک فایل همیشه با یک درخت دودویی پر نشان داده می‌شود، که در آن هر گره‌ی غیر برگ، دو فرزند دارد (تمرین ۱۶-۳-۲ را ببینید). کد با طول ثابت در مثال بالا یک کد بهینه نیست چرا که درخت آن که در شکل ۱۶-۴ (الف) نشان داده شده است یک درخت دودویی پر

نیست: کد کلمه‌هایی وجود دارند که با ۱۰ شروع می‌شوند، ولی هیچ کدام با ۱۱ شروع نمی‌شوند. از آن جایی که می‌خواهیم توجه خود را به درخت‌های دودویی پر متمرکز کنیم، می‌توانیم بگوییم که اگر $C$ الفبایی باشد که کاراکترها متعلق به آن هستند و فراوانی تمام کاراکترها مثبت باشد، آن گاه درخت یک کد پیشوندی بهینه دقیقاً $|C|$ برگ دارد، هر یک مربوط به یکی از حروف الفبا، و دقیقاً $|C|+1$ گره‌ی داخلی دارد (تمرین ب-۵-۳ را ببینید).

با داشتن یک درخت $T$ مربوط به یک کد پیشوندی، می‌توان به راحتی تعداد بیت‌های مورد نیاز برای کد کردن یک فایل را محاسبه کرد. برای هر کاراکتر $c$ در الفبای $C$، فرض کنید $c.freq$ نشان دهنده‌ی فراوانی $c$ در فایل باشد، و $d_T(c)$ نشان دهنده‌ی عمق برگ $c$ در درخت. توجه کنید که $d_T(c)$ طول کد کلمه‌ی کاراکتر $c$ نیز هست. بنابراین، تعداد بیت‌های مورد نیاز برای کد کردن فایل برابر است با

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c) \qquad (۴-۱۶)$$

که آن را به صورت *هزینه‌ی* درخت $T$ تعریف می‌کنیم.

## ساختن یک کد هافمن

هافمن یک الگوریتم حریصانه طراحی کرده است که یک کد پیشوندی بهینه با نام *کد هافمن* (Huffman code) تولید می‌کند. با در نظر داشتن مشاهدات انجام شده در بخش ۱۶-۲، اثبات درستی آن بر مبنای خصوصیت انتخاب حریصانه و زیرساختار بهینه است. به جای این که ابتدا نشان دهیم که این خصوصیات برای این مسئله وجود دارند و سپس یک شبه‌کد طراحی کنیم، ابتدا شبه‌کد را ارائه خواهیم کرد. با این کار راحت‌تر می‌توان نشان داد که الگوریتم چگونه انتخاب حریصانه را انجام می‌دهد.

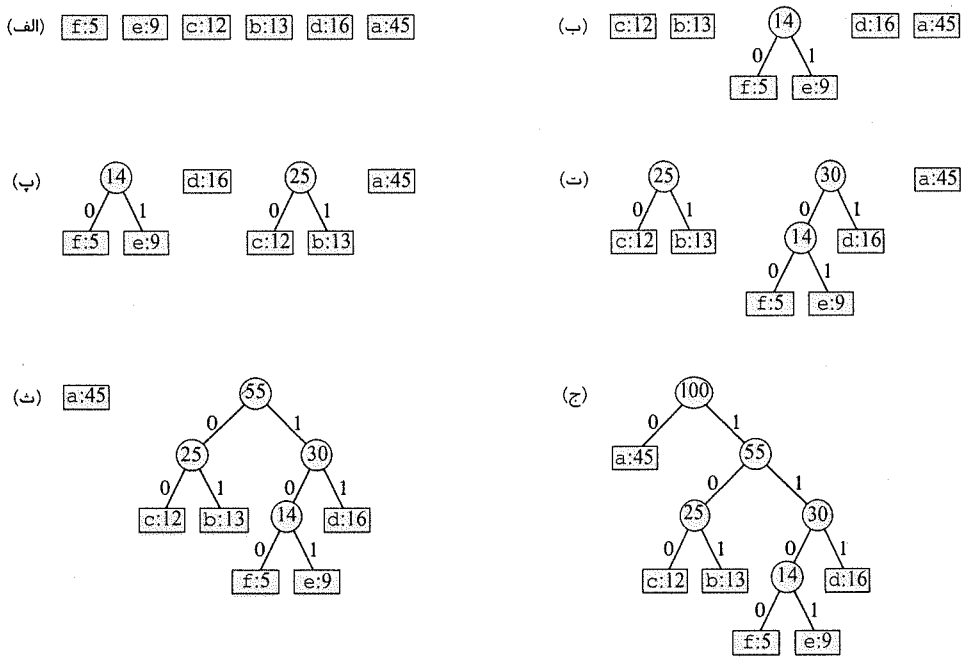
در شبه‌کد زیر، فرض می‌کنیم که $C$ مجموعه‌ای از $n$ کاراکتر است و هر کاراکتر $c \in C$ یک شیئ با یک فراوانی معین $c.freq$ است. الگوریتم، درخت $T$ را که متناظر با کد بهینه است به صورت از پایین به بالا می‌سازد. ساختن درخت با مجموعه‌ای از $|C|$ برگ آغاز می‌شود و سپس دنباله‌ای از $|C|-1$ عملیات «ادغام» انجام می‌شود تا درخت نهایی تولید شود. از یک صف اولویت کمینه $Q$، که کلیدهای آن $freq$ است، برای شناسایی دو شیئ با کم‌ترین فراوانی و ادغام آن‌ها استفاده می‌شود. نتیجه‌ی ادغام دو شیئ یک شیئ است که فراوانی آن مجموع فراوانی دو شیئی است که با یکدیگر ادغام شده‌اند.

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n - 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
```

```
8       INSERT(Q, z)
9   return EXTRACT-MIN(Q)        //return the root of the tree.
```

برای مثال بالا، الگوریتم هافمن مانند شکل ۱۶-۵ عمل می‌کند. از آن جایی که ۶ حرف در الفبا وجود دارد، اندازه‌ی اولیه‌ی صف $n = 6$ است، و برای ساختن کل درخت به ۵ مرحله ادغام نیاز داریم. درخت نهایی، کد پیشوندی نهایی را نشان می‌دهد. کد کلمه‌ی یک حرف، دنباله‌ی برچسب یال‌هایی است که در مسیر از ریشه به آن حرف باید طی شوند.

در خط ۲ صف اولویت کمینه $Q$ با کاراکترهای درون $C$ مقداردهی اولیه می‌شود. حلقه‌ی for در خطوط ۳-۸ متناوبا دو گره‌ی $x$ و $y$ با کم‌ترین فراوانی را از صف استخراج کرده و آن‌ها را با گره‌ی جدید $z$ که ادغام شده‌ی آن دو گره است، جایگزین می‌کند. فراوانی $z$ در خط ۷ به صورت مجموع فراوانی‌های $x$ و $y$ محاسبه می‌شود. فرزند سمت چپ $z$، $x$ و فرزند سمت راست آن $y$ است.




**شکل ۱۶-۵** مراحل الگوریتم هافمن برای فراوانی‌های داده شده در شکل ۱۶-۳. در هر قسمت محتویات صف با ترتیب صعودی فراوانی نشان داده شده است. در هر مرحله، دو درختی که کم‌ترین فراوانی را دارند با یکدیگر ادغام می‌شوند. برگ‌ها به صورت مستطیل‌هایی حاوی حرف مربوطه و فراوانی آن نشان داده شده‌اند، و گره‌های داخلی به صورت دایره‌هایی حاوی مجموع فراوانی فرزندان آن گره. یک یال که یک گره‌ی داخلی را به فرزندش متصل می‌کند، اگر مربوط به فرزند سمت چپ باشد با ۰، و اگر مربوط به فرزند سمت راست باشد، با ۱ مشخص شده است. کد کلمه‌ی یک حرف برابر است با دنباله‌ی برچسب یال‌های مسیری که ریشه را به برگ مربوط به آن حرف متصل می‌کند. (الف) مجموعه‌ی اولیه‌ی $n = 6$ گره، یکی برای هر حرف. (ب)-(ث) مراحل میانی. (ج) درخت نهایی.

(این ترتیب دلخواه است؛ جابه‌جا کردن فرزند سمت چپ و راست هر گره منجر به ایجاد یک کد جدید با همان هزینه می‌شود.) بعد از $n-1$ ادغام، تنها گره‌ی باقی مانده در صف – ریشه‌ی درخت کد – در خط ۹ بازگردانده می‌شود.

با این که اگر متغیرهای $x$ و $y$ را به کل حذف می‌کردیم – و در خطوط ۵ و ۶ کد مستقیماً $z.left$ و $z.right$ را مقداردهی می‌کردیم، و خط ۷ را به صور $z.freq = z.left.freq + z.right.freq$ تغییر می‌دادیم – خروجی الگوریتم تغییری نمی‌کرد، از اسامی $x$ و $y$ در اثبات درستی استفاده خواهیم کرد. بنابراین ترجیح می‌دهیم که آن‌ها را به همین صورت رها کنیم.

در تحلیل زمان اجرای الگوریتم هافمن فرض می‌شود که $Q$ به صورت یک هرم کمینه‌ی دودویی پیاده‌سازی شده است (فصل ۶ را ببینید). برای یک مجموعه‌ی $C$ از $n$ کاراکتر، مقداردهی اولیه‌ی $Q$ درخط ۲ می‌تواند با استفاده از رویه‌ی BUILD-MIN-HEAP که در بخش ۶-۳ ارائه شد، در زمان $O(n)$ انجام شود. حلقه‌ی for در خطوط ۳-۸ دقیقاً $n-1$ بار انجام می‌شود، و از آن جایی که هر یک از اعمال هرم‌ها به زمان $O(\lg n)$ نیاز دارد، اجرای حلقه به $O(n \lg n)$ زمان نیاز دارد. بنابراین، کل زمان اجرای HUFFMAN بر روی یک مجموعه از $n$ کاراکتر برابر است با $O(n \lg n)$. می‌توان با استفاده از درخت‌های van Emde Boas (فصل ۲۰ را ببینید) می‌توانیم این زمان را به $O(n \lg \lg n)$ کاهش دهیم.

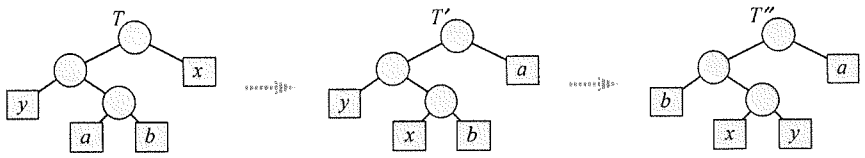
## درستی الگوریتم هافمن

برای اثبات این که الگوریتم حریصانه‌ی HUFFMAN صحیح است، نشان می‌دهیم که مسئله‌ی یافتن یک کد پیشوندی بهینه دارای خصوصیات انتخاب حریصانه و زیرساختار بهینه است. لم زیر نشان می‌دهد که خصوصیت انتخاب حریصانه برای این مسئله صحیح است.

**لم ۱۶-۲** فرض کنید $C$ یک الفبا باشد که در آن هر کاراکتر $c \in C$ دارای فراوانی $c.freq$ است. همچنین فرض کنید $x$ و $y$ دو کاراکتر درون $C$ باشند که دارای کم‌ترین فراوانی هستند. در این صورت یک کد پیشوندی برای $C$ وجود دارد که در آن کد کلمه‌های $x$ و $y$ دارای طول یکسانی هستند، و فقط در بیت آخر با یکدیگر متفاوتند.

*اثبات* ایده‌ی اثبات این است که ابتدا درخت $T$ را به صورت درخت یک کد پیشوندی بهینه‌ی دلخواه بسازیم و سپس طوری آن را اصلاح کنیم که درخت جدید هم نشان‌دهنده‌ی یک کد پیشوندی بهینه باشد، و در آن کاراکترهای $x$ و $y$ برادر یکدیگر باشند با عمق بیشینه. اگر بتوانیم چنین کاری بکنیم، در این صورت کد کلمه‌ی آن‌ها دارای طول یکسانی خواهد بود و فقط در بیت آخر با یکدیگر فرق خواهند داشت.

فرض کنید $a$ و $b$ دو کاراکتر باشند برگ‌های آن‌ها در درخت $T$ با هم برادر باشند با عمق بیشینه. بدون از دست دادن کلیت، فرض می‌کنیم که $a.freq \le b.freq$ و $x.freq \le y.freq$. از آن جایی که هر




**شکل ۱۶-۶** نمایش مرحله‌ی کلیدی در اثبات لم ۱۶-۲. در درخت بهینه‌ی $T$ برگ‌های $a$ و $b$ دو تا از عمیق‌ترین برگ‌ها هستند که برادرند. برگ‌های $x$ و $y$ دو برگ هستند که الگوریتم هافمن اول از همه آن‌ها را با هم ادغام می‌کند؛ آن‌ها در $T$ در مکان‌های نامعلومی قرار دارند. برگ‌های $a$ و $x$ جابه‌جا می‌شوند تا درخت $T'$ به دست آید. سپس، برگ‌های $b$ و $y$ جابه‌جا می‌شوند تا به درخت $T''$ برسیم. از آن جایی که هر جابه‌جایی هزینه را افزایش نمی‌دهد، درخت حاصل $T''$ هم یک درخت بهینه است.

دوی $x.freq$ و $y.freq$ برگ‌های با کم‌ترین فراوانی هستند، به ترتیب، و $a.freq$ و $b.freq$ دو فراوانی دلخواه هستند، باز هم به ترتیب، بنابراین داریم $x.freq \le a.freq$ و $y.freq \le b.freq$.

همان طور که در شکل ۱۶-۶ نشان داده شده است، می‌توانیم در درخت $T$، مکان $a$ را با $x$ عوض کنیم تا درخت $T'$ را بسازیم، و سپس در $T'$ مکان $b$ را با $y$ عوض کنیم تا درخت $T''$ را بسازیم، که در آن $x$ و $y$ برادرهایی با عمق بیشینه هستند. (توجه کنید که اگر $x = b$ ولی $y \neq a$، آن گاه در درخت $T$، $x$ و $y$ برادرهای با عمق بیشینه نخواهند بود. چو فرض کردیم $x \neq b$، چنین حالتی رخ نخواهد داد.) طبق تساوی (۱۶-۴)، تفاوت هزینه‌ی مربوط به درخت $T$ و $T'$ عبارت است از

$$\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) + a.freq \cdot d_{T'}(a) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) + a.freq \cdot d_T(x) \\
&= (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\ge 0
\end{aligned}$$

چرا که هر دوی $a.freq - x.freq$ و $d_T(a) - d_T(x)$ نامنفی هستند. به طور دقیق‌تر، $a.freq - x.freq$ نامنفی است زیرا $x$ یک برگ با فراوانی کمینه است، و $d_T(a) - d_T(x)$ نامنفی است زیرا $a$ یک برگ با عمق بیشینه در $T$ است. به طور مشابه، جابه‌جایی $y$ و $b$ هزینه را افزایش نمی‌دهد، و بنابراین $B(T') - B(T'')$ نامنفی است. از این رو، $B(T) \le B(T'')$، و از آن جایی که $T$ بهینه است، $B(T'') \le B(T)$، که نتیجه می‌دهد $B(T'') = B(T)$. بنابراین $T''$ یک درخت بهینه است که در آن $x$ و $y$ به صورت برگ‌های برادر با بیشترین عمق ظاهر می‌شوند، و لم بدین صورت اثبات می‌شود. ■

لم ۱۶-۲ بر این دلالت دارد که فرآیند ساختن یک درخت بهینه، بدون از دست دادن کلیت، می‌تواند با انتخاب حریصانه‌ی ادغام دو کاراکتری که کم‌ترین فراوانی را دارند آغاز شود. چرا این

انتخاب حریصانه است؟ ما می‌توانیم هزینه‌ی یک عنصر تولید شده از ادغام را به صورت مجموع دو عنصری ببینیم که با یکدیگر ادغام شده‌اند و این عنصر را ساخته‌اند. تمرین ۱۶-۳-۳ نشان می‌دهد که هزینه‌ی کل درخت ساخته شده برابر است با مجموع هزینه‌های عناصر ادغامی. در میان تمام ادغام‌های ممکن، HUFFMAN آن را انتخاب می‌کند که کم‌ترین هزینه را ایجاد کند.

لم بعد نشان می‌دهد که مسئله‌ی ساختن کدهای پیشوندی بهینه دارای خصوصیت زیرساختار بهینه است.

**لم ۱۶-۳** فرض کنید $C$ یک الفبا باشد که در آن فراوانی هر کاراکتر $c.freq$ با $c \in C$ مشخص شده است، و $x$ و $y$ دو کاراکتر در $C$ باشند که دارای کم‌ترین فراوانی هستند. فرض کنید $C'$ الفبای $C$ باشد که در آن دو کاراکتر $x$ و $y$ حذف شده‌اند و کاراکتر (جدید) $z$ به آن اضافه شده است، یعنی $C' = C - \{x, y\} \cup \{z\}$؛ $f$ را برای $C'$ مانند $C$ تعریف می‌کنیم، فقط با این تفاوت که $z.freq = x.freq + y.freq$. فرض کنید $T'$ یک درخت دلخواه باشد که نشان‌دهنده‌ی یک کد پیشوندی بهینه برای الفبای $C'$ است. آن گاه درخت $T$، که از قرار دادن یک گره‌ی داخلی که $x$ و $y$ فرزندان آن هستند، به جای $z$ در درخت $T'$ به وجود می‌آید، نشان‌دهنده‌ی یک کد پیشوندی بهینه برای الفبای $C$ است.

*اثبات* ابتدا نشان می‌دهیم که هزینه‌ی $B(T)$ مربوط به درخت $T$ را می‌توان بر حسب هزینه‌ی $B(T')$ مربوط به درخت $T'$، با در نظر گرفتن هزینه‌ی عناصر به صورت تساوی (۱۶-۵) توصیف کرد. برای هر $c \in C - \{x, y\}$ داریم $d_T(c) = d_{T'}(c)$، و بنابراین $c.freq \cdot d_T(c) = c.freq \cdot d'(c)$. از آن جایی که $d_T(x) = d_T(y) = d'(z) + 1$، داریم

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d'(z) + 1) \\ &= z.freq \cdot d'(z) + (x.freq + y.freq) \end{aligned}$$

که می‌توانیم از آن نتیجه بگیریم

$$B(T) = B(T') + x.freq + y.freq$$

یا به طور مشابه

$$B(T') = B(T) - x.freq - y.freq$$

لم را با استفاده از برهان خلف اثبات می‌کنیم. فرض کنید که $T$ نشان‌دهنده‌ی یک کد پیشوندی بهینه برای $C$ نباشد. در این صورت یک درخت $T''$ وجود دارد به طوری که $B(T'') < B(T)$. بدون از دست دادن کلیت مسئله (طبق لم ۱۶-۲)، فرض می‌کنیم $x$ و $y$ در $T''$ برادر هستند. فرض کنید $T'''$ درخت $T''$ باشد که در آن پدر $x$ و $y$ با یک گره‌ی $z$ با فراوانی $z.freq = x.freq + y.freq$ جایگزین شده است. در این صورت

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') \end{aligned}$$

که با فرض این که $T'$ نشان‌دهنده‌ی یک کد پیشوندی بهینه برای $C'$ است تناقض دارد. بنابراین، $T$ باید نشان‌دهنده‌ی یک کد پیشوندی بهینه برای الفبای $C$ باشد.

**قضیه‌ی ۱۶-۴** رویه‌ی HUFFMAN یک کد پیشوندی بهینه تولید می‌کند.

*اثبات* مستقیماً از لم‌های ۱۶-۲ و ۱۶-۳ نتیجه می‌شود. ■

## تمرین‌ها

۱۶-۳-۱ توضیح دهید چرا در اثبات لم ۱۶-۲، اگر $x.freq = b.freq$، آن گاه باید داشته باشیم $a.freq = b.freq = x.freq = y.freq$.

۱۶-۳-۲ اثبات کنید که یک درخت دودویی که پر نیست نمی‌تواند نشان‌دهنده‌ی یک کد پیشوندی بهینه باشد.

۱۶-۳-۳ کد هافمن بهینه‌ی مربوط به فراوانی‌های زیر، که ۸ عدد اول فیبوناچی هستند، چیست؟

$a:۱ \quad b:۱ \quad c:۲ \quad d:۳ \quad e:۵ \quad f:۸ \quad g:۱۳ \quad h:۲۱$

آیا می‌توانید حالت کلی جواب خود را برای وقتی که فراوانی‌ها $n$ عدد اول فیبوناچی هستند، به دست آورید؟

۱۶-۳-۴ اثبات کنید که هزینه‌ی کلی یک درخت برای یک کد می‌تواند به صورت مجموع هزینه‌ی فرزندان تمام گره‌های داخلی هم محاسبه شود.

۱۶-۳-۵ اثبات کنید که اگر کارکترهای الفبا را به ترتیب صعودی فراوانی‌ها مرتب کنیم، آن گاه یک کد بهینه برای آن الفبا وجود دارد که طول کد کلمه‌های آن به صورت نزولی است.

۱۶-۳-۶ فرض کنید که یک کد پیشوندی بهینه بر روی یک مجموعه‌ی $C\{۰,۱,...,n-۱\}$ از کاراکترها داریم و می‌خواهیم این کد را با کم‌ترین تعداد بیت ممکن مخابره کنیم. نشان دهید که چگونه می‌توان هر کد پیشوندی بهینه‌ای بر روی $C$ را فقط با استفاده از $۲n-۱+n\lceil \lg n \rceil$ بیت نمایش داد. (*راهنمایی:* از $۲n-۱$ بیت برای نشان دادن ساختار درخت استفاده کنید، که این ساختار با پیمایش بر روی درخت به دست می‌آید.)

۱۶-۳-۷ الگوریتم هافمن را طوری اصلاح کنید که کد کلمه‌های منبای سه (مثلاً کد کلمه‌هایی که از سمبل‌های ۰، ۱ و ۲ استفاده می‌کنند) تولید کند، و اثبات کنید که کدهای مبنای ۳ تولید

شده بهینه هستند.

۱۶-۳-۸ فرض کنید که یک فایل حاوی دنباله‌ای از کاراکترهای ۸ بیتی داریم به طوری که عمومیت تمام ۲۵۶ کاراکتر درون فایل تقریباً مشابه است: فراوانی بیشینه کم‌تر از دو برابر فراوانی کمینه است. اثبات کنید که کد هافمن برای این فایل بهینه‌تر از کد ۸ بیتی معمولی با طول ثابت نیست.

۱۶-۳-۹ اثبات کنید که نمی‌توانیم انتظار داشته باشیم که هیچ روش فشرده‌سازی یک فایل شامل کاراکترهای ۸ بیتی تصادفی را حتی به اندازه‌ی یک بیت فشرده کند. (یعنی امیدریاضی میزان فشرده‌سازی یک روش بر روی یک فایل با کاراکترهای تصادفی ۰ است.) *(راهنمایی:* تعداد فایل‌های ممکن را با تعداد فایل‌های کد شده‌ی ممکن مقایسه کنید.)

## ۱۶-۴ مبانی تئوری برای متدهای حریصانه

یک تئوری زیبا در مورد الگوریتم‌های حریصانه وجود دارد، که در این بخش در مورد آن بحث خواهیم کرد. این تئوری موقعیت‌های بسیاری را توصیف می‌کند که در آن‌ها الگوریتم‌های حریصانه به جواب‌های بهینه ختم می‌شوند، و شامل ساختارهایی ترکیبیاتی به نام «ماتروییدها» است. با این که این تئوری تمام حالت‌هایی را که برای آن‌ها می‌توان از یک متد حریصانه استفاده کرد پوشش نمی‌دهد (مسئله‌ی انتخاب فعالیت بخش ۱۶-۱ و مسئله‌ی کد هافمن بخش ۱۶-۳ نمونه‌هایی هستند که این تئوری پوشش نمی‌دهد)، ولی بسیاری از حالت‌های عملی را پوشش می‌دهد. به علاوه، این تئوری در حال توسعه‌ی شدید و به کارگیری برای بسیاری موارد دیگر است.

### ماتروییدها

یک *ماترویید*(matroid) یک جفت مرتب $M = (S, \ell)$ است که شرایط زیر را ارضا می‌کند.

۱. $S$ یک مجموعه‌ی متناهی ناتهی است.

۲. $\ell$ یک خانواده‌ی ناتهی از زیرمجموعه‌های $S$ است، با نام زیرمجموعه‌های *مستقل* $S$، به طوری که اگر $B \subset \ell$ و $A \subset B$، آن گاه $A \subset \ell$. می‌گوییم $\ell$ *ارثی*(hereditary) است اگر این خصوصیت را ارضا کند. توجه داشته باشید که مجموعه‌ی تهی $\varnothing$ لزوماً عضوی از $\ell$ است.

۳. اگر $A \in \ell$، $B \in \ell$، و $|A| < |B|$، آن گاه یک عنصر $x \in B - A$ وجود دارد به طوری که $A \cup \{x\} \in \ell$. می‌گوییم $M$ *خصوصیت تعویض*(exchanging property) را ارضا می‌کند.

کلمه‌ی «ماترویید» منتسب است به Hassler Whitney . او در حال یادگیری ماترویدهای ماتریسی بود، که در آن عناصر $S$ ردیف‌های یک ماتریس هستند و یک مجموعه از ردیف‌ها در صورتی مستقل هستند که مستقل خطی باشند. همان طور که تمرین ۱۶-۴-۲ از شما می‌خواهد نشان دهید، این

ساختار تعریف کننده‌ی یک ماتروید است.

به عنوان مثالی دیگر از ماتروییدها، *ماتروید گرافی* (graphic matroid) $M_G = (S_G, \ell_G)$ را در نظر بگیرید که به صورت یک گراف بدون جهت $G = (V, E)$ به صورت زیر تعریف می‌شود.

- مجموعه‌ی $S_G$ به صورت $E$، مجموعه‌ی یال‌های $G$ تعریف می‌شود.
- اگر $A$ یک زیرمجموعه از $E$ باشد، آن گاه $A \in \ell_G$ اگر و فقط اگر $A$ بدون دور باشد. یعنی، مجموعه‌ی یال‌های $A$ مستقل است اگر و فقط اگر زیرگراف $G_A = (V, A)$ یک جنگل را تشکیل دهد.

ماتروید گرافی $M_G$ به طور نزدیک با مسئله‌ی درخت پوشای کمینه ارتباط دارد، که در فصل ۲۳ به تفصیل در مورد آن بحث می‌شود.

**قضیه‌ی ۱۶-۵** اگر $G = (V, E)$ یک گراف بدون جهت باشد، آن گاه $M_G = (S_G, \ell_G)$ یک ماتروید است.

*اثبات* به وضوح، $S_G = E$ یک مجموعه‌ی متناهی است. به علاوه $\ell_G$ ارثی است، چرا که زیرمجموعه‌ای از یک جنگل یک جنگل است. به عبارت دیگر، حذف چند یال از یک مجموعه‌ی بدون دور از یال‌ها نمی‌تواند یک دور به وجود آورد.

بنابراین فقط این باقی می‌ماند که نشان دهیم $M_G$ خصوصیت تعویض را ارضا می‌کند. فرض کنید که $G_A = (V, A)$ و $G_B = (V, B)$ جنگل‌هایی از $G$ باشند و $|A| > |B|$. یعنی $A$ و $B$ مجموعه‌هایی از یال‌های بدون دور هستند، و تعداد یال‌های $B$ بیشتر از تعداد یال‌های $A$ است.

ادعا می‌کنیم که یک جنگل $F = (V_F, E_F)$ دقیقاً حاوی $|V_F| - |E_F|$ درخت است. برای این که ببینیم چرا، فرض کنید که $F$ حاوی $t$ درخت باشد، که در آن درخت $i$ام، $v_i$ رأس و $e_i$ یال دارد. آن گاه داریم

$$\begin{aligned} |E_F| &= \sum_{i=1}^{t} e_i \\ &= \sum_{i=1}^{t} (v_i - 1) \qquad \text{(طبق قضیه‌ی ب-۲)} \\ &= \sum_{i=1}^{t} v_i - t \\ &= |V_F| - t \end{aligned}$$

از آن جایی که تعداد درخت‌های $G_B$ کم‌تر از تعداد درخت‌های $G_A$ است، جنگل $G_B$ باید حاوی یک درخت $T$ باشد به طوری که رأس‌های آن در جنگل $G_A$ در دو درخت متفاوت هستند. به طور دقیق‌تر، از آن جایی که $T$ همبند است، باید حاوی یک یال $(u, v)$ باشد به طوری که رأس‌های $u$ و $v$

در جنگل $G_A$ در دو درخت متفاوت باشند. چون یال $(u,v)$ در جنگل $G_A$ دو درخت متفاوت را به هم متصل می‌کند، می‌توان این یال را به جنگل $G_A$ اضافه کرد بدون این که در این جنگل دوری به وجود بیاید. بنابراین $M_G$ خصوصیت تعویض را ارضا می‌کند، و اثبات ماتروید بودن $M_G$ در این جا کامل می‌شود.

با داشتن یک ماتروید $M=(S,\ell)$، می‌گوییم یک عنصر $x \notin A$ یک *بسط* (extension) $A$ است اگر بتوان $x$ را با حفظ استقلال به $A$ اضافه کرد؛ یعنی $x$ یک بسط $A$ است اگر $A \cup \{x\} \in \ell$. به عنوان یک مثال، ماتروید گرافی $M_G$ را در نظر بگیرید. اگر $A$ یک مجموعه‌ی مستقل از یال‌ها باشد، آن گاه $e$ یک بسط $A$ است اگر و فقط اگر $e$ در $A$ نباشد و اضافه کردن $e$ به $A$ یک دور به وجود نیاورد.

اگر $A$ یک زیرمجموعه‌ی مستقل در ماتروید $M$ باشد، می‌گوییم $A$ *ماکسیمال* (maximal) است اگر هیچ بسطی نداشته باشد. یعنی $A$ ماکسیمال است اگر در هیچ زیرمجموعه‌ی مستقل دیگر $M$ وجود نداشته باشد. خصوصیت زیر معمولاً مفید است.

**قضیه‌ی ۱۶.۶** تمام زیرمجموعه‌های مستقل ماکسیمال در یک ماتروید دارای اندازه‌ی یکسانی هستند.

*اثبات* طبق برهان خلف فرض کنید $A$ یک زیرمجموعه‌ی مستقل ماکسیمال در $M$ باشد، و یک زیرمجموعه‌ی مستقل ماکسیمال دیگر $B$ نیز وجود داشته باشد. بنابراین خصوصیت تعویض ایجاب می‌کند که $A$ قابل بسط به یک مجموعه‌ی مستقل دیگر $A \cup \{x\}$ باشد، برای یک $x \in B - A$، که با فرض این که $A$ ماکسیمال است تناقض دارد.

به عنوان نمونه‌ای از این قضیه، یک ماتروید گرافی $M_G$ را برای یک گراف بدون جهت همبند $G$ در نظر بگیرید. تمام زیرمجموعه‌های مستقل ماکسیمال $M_G$ باید یک درخت با دقیقاً $|V|-1$ یال باشند که تمام رأس‌های $G$ را به هم متصل می‌کنند. به چنین درختی، یک *درخت پوشای* (spanning tree) $G$ گفته می‌شود.

می‌گوییم یک ماتروید $M=(S,\ell)$ *وزن‌دار* (weighted) است اگر یک تابع وزن مربوطه‌ی $w$ وجود داشته باشد که به هر عنصر $x \in S$ یک وزن اکیدا مثبت (بزرگ‌تر از ۰) $w(x)$ نسبت می‌دهد. وزن زیرمجموعه‌های $S$ به صورت زیر است

$$w(A) = \sum_{x \in A} w(x)$$

برای هر $A \subseteq S$. مثلاً اگر $w(e)$ نشان‌دهنده‌ی طول یک یال در یک ماتروید گرافی $M_G$ باشد، آن گاه $W(A)$ مجموع طول‌های یال‌های درون $A$ است.

## الگوریتم‌های حریصانه بر روی ماتروییدهای وزن‌دار

بسیاری از مسائلی را که به کمک رویکرد حریصانه می‌توان برای آن‌ها یک جواب بهینه یافت، می‌توان به صورت یافتن یک زیرمجموعه‌ی مستقل با وزن بیشینه در یک ماتروید وزن‌دار فرموله کرد. یعنی به ما یک ماتروید وزن‌دار $M(S,\ell)$ داده شده است، و می‌خواهیم یک مجموعه‌ی مستقل $A \in \ell$ بیابیم به طوری که $w(A)$ بیشینه باشد. به چنین زیرمجموعه‌ای که مستقل است و بیشترین وزن ممکن را دارد، یک ماتروید**بهینه** می‌گوییم. از آن جایی که وزن $w(x)$ مربوط به هر عنصر $x \in S$ مثبت است، یک زیرمجموعه‌ی بهینه همیشه یک زیرمجموعه‌ی مستقل بیشینه است – همیشه مفید است که $A$ را تا حد ممکن بزرگ کنیم.

به عنوان مثال در مسئله‌ی درخت پوشای کمینه، یک گراف بدون جهت همبند $G=(V,E)$ و یک تابع طول $w$ داریم که $w(e)$ طول (مثبت) یال $e$ است. (در این جا از اصطلاح «طول» برای وزن اصلی یال‌های گراف استفاده می‌کنیم، و اصطلاح «وزن» را برای اشاره به وزن‌های مربوط به ماتروید نگه می‌داریم.) ما باید یک زیرمجموعه از یال‌های گراف بیابیم که تمام رأس‌ها را به هم متصل می‌کند و دارای طول کلی کمینه است. برای دیدن این مسئله به صورت مسئله‌ی یافتن یک زیرمجموعه‌ی بهینه از یک ماتروید، ماتروید وزن‌دار $M_G$ را با تابع وزن $w'$ در نظر بگیرید، که در آن $w'(e)=w_0-w(e)$ و $w_0$ بزرگ‌تر از طول تمام یال‌ها است. در این ماتروید وزن‌دار، تمام وزن‌ها مثبت هستند و یک زیرمجموعه‌ی بهینه، یک درخت پوشا با طول کلی کمینه در گراف اصلی است. به طور دقیق‌تر، هر زیرمجموعه‌ی مستقل ماکسیمال $A$ متناظر است با یک درخت پوشا با $|V|-1$ یال، و بنابراین

$$\begin{aligned} w'(A) &= \sum_{e\in A} w'(e) \\ &= \sum_{e\in A} (w_0 - w(e)) \\ &= (|V|-1)w_0 - \sum_{e\in A} w(e) \\ &= (|V|-1)w_0 - w(A) \end{aligned}$$

برای هر زیرمجموعه‌ی مستقل ماکسیمال $A$، یک زیرمجموعه‌ی مستقل که کمیت $w'(A)$ را بیشینه می‌کند باید $w(A)$ را کمینه کند. بنابراین، هر الگوریتمی که بتواند یک زیرمجموعه‌ی بهینه‌ی $A$ در یک ماتروید دلخواه بیابد می‌تواند مسئله‌ی درخت پوشای کمینه را حل کند.

در فصل ۲۳ الگوریتم‌هایی برای مسئله‌ی درخت پوشای کمینه ارائه خواهد شد، ولی در این جا یک الگوریتم حریصانه ارائه می‌کنیم که برای هر ماتروید وزن‌داری کار می‌کند. این الگوریتم یک ماتروید وزن‌دار $M=(S,\ell)$ را به همراه تابع وزن مربوطه‌ی $w$ به عنوان ورودی دریافت می‌کند، و یک زیرمجموعه‌ی بهینه‌ی $A$ را بازمی‌گرداند. در شبه‌کد زیر، عناصر $M$ را با $S[M]$ و $\ell[M]$ و تابع وزن را با $w$ نشان می‌دهیم. این الگوریتم از آن جا حریصانه است که هر عنصر $x \in S$ را به ترتیب

نزولی وزن در نظر می‌گیرد و در صورتی که $A \cup \{x\}$ مستقل باشد، $x$ را به $A$ اضافه می‌کند.

```
GREEDY(M, w)
1  A = Ø
2  sort M.S into monotonically decreasing order by weight w
3  foreach x ∈ M.S, taken in monotonically decreasing order by weight w(x)
4      if A ∪ {x} ∈ M.ℓ
5          A = A ∪ {x}
6  return A
```

خط ۴ چک می‌کند که آیا اضافه کردن هر عنصر $x$ به $A$ مستقل بودن $A$ را به هم می‌زند یا خیر. اگر $A$ مستقل باقی بماند، آن گاه خط ۵، $x$ را به $A$ اضافه می‌کند. در غیر این صورت از $x$ صرف نظر می‌شود. از آن جایی که طبق تعریف ماتروییدها مجموعه‌ی تهی مستقل است، و از آن جایی که $x$ فقط در صورتی به $A$ اضافه می‌شود که $A \cup \{x\}$ مستقل باشد، طبق استقرا زیرمجموعه‌ی $A$ همیشه مستقل است. بنابراین، GREEDY همیشه یک زیرمجموعه‌ی مستقل $A$ را بازمی‌گرداند. در ادامه خواهیم دید که $A$ یک زیرمجموعه با بیشینه وزن ممکن است، و بنابراین یک زیرمجموعه‌ی بهینه.

تحلیل زمان اجرای GREEDY ساده است. فرض کنید $n$ نشان‌دهنده‌ی $|S|$ باشد. مرحله‌ی مرتب‌سازی GREEDY به زمان $O(n \lg n)$ نیاز دارد. خط ۴ دقیقاً $n$ بار اجرا می‌شود، یک بار برای هر یک از عناصر $S$. هر بار اجرای خط ۴ باید چک کند که آیا مجموعه‌ی $A \cup \{x\}$ مستقل است یا خیر. اگر چنین چک کردنی $O(f(n))$ زمان بگیرد، کل زمان اجرای الگوریتم $O(n \lg n + nf(n))$ خواهد بود.

اکنون اثبات می‌کنیم که GREEDY یک زیرمجموعه‌ی مستقل بازمی‌گرداند.

**لم ۱۶.۷ (بیروی ماتروییدها از خصوصیت انتخاب حریصانه)**

فرض کنید که $M = (S, \ell)$ یک ماتروید وزن‌دار با تابع وزن $w$ باشد، و همچنین $S$ به ترتیب نزولی وزن مرتب شده باشد. فرض کنید $x$ اولین عنصر $S$ باشد به طوری که $\{x\}$ مستقل است، البته در صورت وجود. اگر $x$ وجود داشته باشد، در این صورت یک زیرمجموعه‌ی بهینه‌ی $A$ از $S$ وجود دارد که شامل $x$ می‌شود.

*اثبات* اگر چنین $x$ی وجود نداشته باشد، در این صورت تنها زیرمجموعه‌ی مستقل موجود، مجموعه‌ی تهی است و کار ما تمام می‌شود. در غیر این صورت، فرض کنید $B$ یک زیرمجموعه‌ی مستقل ناتهی باشد. فرض کنید $x \notin B$؛ در غیر این صورت $A = B$ و اثبات تمام است.

هیچ عنصری از $B$ وزنی بیشتر از $w(x)$ ندارد. برای دیدن این مطلب، مشاهده کنید که $y \in B$ نتیجه می‌دهد که $\{y\}$ مستقل است، چرا که $B \in \ell$، و $\ell$ ارثی است. بنابراین انتخاب ما از $x$ تضمین می‌کند که برای هر $y \in B$، $w(x) \geq w(y)$.

مجموعه‌ی $A$ را به صورت زیر می‌سازیم. با $A = \{x\}$ شروع می‌کنیم. طبق انتخاب $x$، $A$ مستقل

است. با استفاده از خصوصیت تعویض، مکرراً یک عنصر جدید در $B$ می‌یابیم که می‌توان آن را بـه $A$ اضافه کرد و استقلال $A$ حفظ شود، تا وقتی که $|A| = |B|$. در این لحظه، $A$ و $B$ یکسان هستند، غیر از این که $A$ حاوی $x$ است، و $B$ به جای آن یک عنصر دیگر $y$ را دارد. یعنی $A = B - \{y\} \cup \{x\}$ برای یک $y \in B$، و بنابراین

$$\begin{aligned} w(A) &= w(B) - w(y) + w(x) \\ &\geq w(B) \end{aligned}$$

از آن جایی که $B$ بهینه است، $A$ هم باید بهینه باشد، و از آن جایی که $x \in A$، اثبات لم تمام شده است. ■

در ادامه نشان می‌دهیم که اگر یک عنصر در ابتدا یک گزینه نباشد، نمی‌تواند بعداً یک گزینه باشد.

**لم ۱۶-۸** فرض کنید $M = (S, \ell)$ یک ماتروید باشد. اگر $x$ یک عنصر در $S$ باشد که یک بسط از یک زیرمجموعه‌ی مستقل $A$ از $S$ است، در این صورت $x$ یک بسط از $\emptyset$ هم هست.

*اثبات* از آن جایی که $x$ یک بسط از $A$ است، $A \cup \{x\}$ مستقل است. از آن جایی که $\ell$ ارثی است، باید مستقل باشد. بنابراین $x$ یک بسط از $\emptyset$ است. ■

**نتیجه‌ی ۱۶-۹** فرض کنید $M = (S, \ell)$ یک ماتروید باشد. اگر $x$ یک عنصر از $S$ باشد به طوری که $x$ یک بسط $\emptyset$ نباشد، آن گاه $x$ بسط هیچ زیرمجموعه‌ی مستقل $A$ از $S$ نیست.

*اثبات* این نتیجه به سادگی عکس نقیض لم ۱۶-۸ است. ■

نتیجه‌ی ۱۶-۹ می‌گوید که هر عنصری که در همان ابتدا قابل استفاده نیست، هیچ وقت قابل استفاده نخواهد بود. بنابراین، GREEDY نمی‌تواند با رد کردن یک عنصر $S$ که بسطی از $\emptyset$ نیست اشتباهی انجام دهد، چرا که هیچ وقت نمی‌توان از آن عنصر استفاده کرد.

**لم ۱۶-۱۰ (بیرون ماتروید‌ها از خصوصیت زیرساختار بهینه)** فرض کنید $x$ اولین عنصری از $S$ باشد که توسط GREEDY برای یک ماتروید وزن‌دار $M = (S, \ell)$ انتخاب می‌شود. مسئله‌ی باقی مانده‌ی یافتن یک زیرمجموعه‌ی مستقل با وزن بیشینه شامل $x$ به مسئله‌ی یافتن یک زیرمجموعه‌ی مستقل با وزن بیشینه از ماتروید وزن‌دار $M' = (S', \ell')$ کاهش می‌یابد، که در آن

$$S' = \{y \in S \mid \{x, y\} \in \ell\}$$
$$\ell' = \{B \subseteq S - \{x\} \mid B \cup \{x\} \in \ell\}$$

و تابع وزن $w'$ همان تابع وزن $w$ است، که به $S'$ محدود شده است. (به $M'$، *اختصار* (contradiction) $M$ توسط عنصر $x$ می‌گوییم.)

*اثبات* اگر $A$ یک زیرمجموعه‌ی مستقل $M$ با وزن بیشینه و شامل $x$ باشد، در این صورت $A' = A - \{x\}$ یک زیرمجموعه‌ی مستقل $M'$ است. بالعکس، هر زیرمجموعه‌ی مستقل $A'$ از $M'$ می‌تواند به یک زیرمجموعه‌ی مستقل $A = A' \cup \{x\}$ از $M$ تبدیل شود. از آن جایی که در هر دو حالت درایم $w(A) = w(A') + w(x)$، یک جواب با وزن بیشینه در $M$ شامل $x$ به یک جواب با وزن بیشینه در $M$ منجر می‌شود، و بالعکس. ■

**قضیه‌ی ۱۶-۱۱ درستی الگوریتم حریصانه بر روی ماتروید‌ها**

اگر $M = (S, \ell)$ یک ماتروید وزن‌دار با تابع وزن $w$ باشد، آن گاه $\text{GREEDY}(M, w)$ یک زیرمجموعه‌ی بهینه باز می‌گرداند.

*اثبات* طبق نتیجه‌ی ۱۶-۹ هر عنصری را که در ابتدا به دلیل بسط $\varnothing$ نبودن رد شود، می‌توان به کل فراموش کرد، چرا که هیچ وقت نمی‌توان آن‌ها را مورد استفاده قرار داد. وقتی اولین عنصر $x$ انتخاب شد، لم ۱۶-۷ ایجاب می‌کند که GREEDY با اضافه کردن $x$ به $A$ اشتباه نمی‌کند، چرا که یک زیرمجموعه‌ی بهینه شامل $x$ وجود دارد. در نهایت لم ۱۶-۱۰ نتیجه می‌دهد که مسئله‌ی باقی مانده، مسئله‌ی یافتن یک زیرمجموعه‌ی بهینه در ماتروید $M'$ است، که اختصار ماتروید $M$ توسط $x$ است. بعد از این که رویه‌ی GREEDY، $A$ را برابر $\{x\}$ قرار می‌دهد، تمام مراحل بعدی آن را می‌توان به صورت اعمالی بر روی ماتروید $M' = (S', \ell')$ تلقی کرد، چون $B$ در $M'$ مستقل است اگر و فقط اگر $B \cup \{x\}$ در $M$ مستقل باشد، برای تمام مجموعه‌های $B \in \ell'$. بنابراین، اعمال بعدی GREEDY یک زیرمجموعه‌ی مستقل با وزن بیشینه برای $M'$ خواهد یافت، و نتیجه‌ی عملیات کلی GREEDY، یافتن یک زیرمجموعه‌ی مستقل با وزن بیشینه در $M$ است. ■

## تمرین‌ها

۱-۴-۱۶ نشان دهید که $(S, \ell_k)$ یک ماتروید است، که در آن $S$ یک مجموعه‌ی متناهی است، و $\ell_k$ مجموعه‌ی تمام زیرمجموعه‌های $S$ با حداکثر اندازه‌ی $k$، که در آن $k \le |S|$.

★ ۲-۴-۱۶ با داشتن یک ماتریس $T$ با اندازه‌ی $m \times n$ بر روی یک فیلد خاص (مانند اعداد طبیعی)، نشان دهید که $(S, \ell)$ یک ماتروید است، که در آن $S$ مجموعه‌ی ستون‌های $T$ است، و $A \in \ell$ اگر و فقط اگر ستون‌های $A$ مستقل خطی باشند.

★ ۳-۴-۱۶ نشان دهید اگر $(S, \ell)$ یک ماتروید باشد، آن گاه $(S, \ell')$، که در آن

$\ell' = \{A' \mid A \in \ell$ باشد $A' \subseteq S - A\}$ حاوی یک مجموعه‌ی ماکزیمال

هم یک ماتروید است. یعنی، مجموعه‌های مستقل ماکسیمال $(S,\ell')$ دقیقاً مکمل مجموعه‌های مستقل ماکسیمال $(S,\ell)$ هستند.

★ ۴-۴-۱۶ فرض کنید $S$ یک مجموعه‌ی متناهی و $S_1,S_2,...,S_k$ تقسیم‌بندی $S$ به زیرمجموعه‌های گسسته‌ی ناتهی باشد. ساختار $(S,\ell)$ را با شرط $\ell=\{A:|A\cap S_i|\le 1 \ \ for\ i=1,2,...,k\}$ تعریف کنید. نشان دهید که $(S,\ell)$ یک ماتروید است. یعنی مجموعه‌ی تمام مجموعه‌های $A$ که حاوی حداکثر یک عنصر در هر بلوک تقسیم‌بندی دارند، نشان‌دهنده‌ی مجموعه‌های مستقل یک ماتروید است.

۵-۴-۱۶ نشان دهید که چطور می‌توان تابع وزن یک مسئله‌ی ماتروید وزن‌دار را، که در آن جواب بهینه‌ی مورد نظر یک زیرمجموعه‌ی ماکسیمال مستقل با *وزن کمینه* است، طوری تغییر داد که مسئله به یک مسئله‌ی ماتروید وزن‌دار استاندارد تبدیل شود. با دقت بحث کنید که تبدیل شما صحیح است.

## ★ ۵-۱۶ یک مسئله‌ی انتخاب فعالیت

یک مسئله‌ی جذاب که می‌توان با استفاده از ماتروئیدها آن را حل کرد، مسئله‌ی برنامه‌ریزی بهینه‌ی فعالیت‌های با زمان واحد بر روی یک پردازنده است، که در آن هر فعالیت یک مهلت پایان دارد، به همراه یک جریمه که باید در صورت گذشتن مهلت پرداخت شود. این مسئله به نظر پیچیده می‌آید، ولی با استفاده از یک الگوریتم حریصانه به شکل تعجب‌آوری به سادگی حل می‌شود.

یک *فعالیت زمان واحد* یک کار است، مانند یک برنامه برای اجرا بر روی کامپیوتر، که دقیقاً به یک واحد زمان برای اجرا نیاز دارد. با داشتن یک مجموعه‌ی متناهی $S$ از فعالیت‌های زمان واحد، یک *برنامه* (schedule) برای $S$ یک جایگشت از $S$ است که ترتیب اجرای فعالیت‌ها را مشخص می‌کند. اولین فعالیت برنامه در زمان ۰ آغاز و در زمان ۱ تمام می‌شود، فعالیت دوم در زمان ۱ آغاز و در زمان ۲ تمام می‌شود، و الی آخر.

مسئله‌ی *برنامه‌ریزی برای فعالیت‌های زمان واحد با مهلت پایان و جریمه برای یک پردازنده* دارای ورودی‌های زیر است:

- یک مجموعه‌ی $S=\{a_1,a_2,...,a_n\}$ از فعالیت‌های زمان واحد؛
- یک مجموعه از $n$ *مهلت پایان* (deadline) $d_1,d_2,...,d_n$، که اعداد صحیح هستند و هر $d_i$ شرط $1\le d_i\le n$ را ارضا می‌کند، و فعالیت $a_i$ باید در زمان $d_i$ پایان یابد؛ و
- یک مجموعه از $n$ وزن نامنفی یا *جریمه* (penalty) $w_1,w_2,...,w_n$، به طوری که باید جریمه‌ی $w_i$ را پرداخت کنیم اگر فعالیت $a_i$ در زمان $d_i$ انجام نشود، و اگر هر فعالیت در زمان خود انجام شود، هیچ جریمه‌ای پرداخت نمی‌کنیم.

ما باید یک برنامه برای $S$ بیابیم به طوری که میزان کل جریمه‌های پرداخت شده به خاطر رعایت نکردن مهلت‌ها کمینه شود.

یک برنامه‌ی خاص را در نظر بگیرید. می‌گوییم یک فعالیت در این برنامه *دیر* (late) است اگر بعد از مهلت اجرای خود انجام شود. در غیر این صورت، این فعالیت *به‌هنگام* (early) است. همیشه یک برنامه را می‌توان به صورت *به‌هنگام-اول* (early-first form) در آورد، که در آن فعالیت‌های به‌هنگام قبل از فعالیت‌های دیر انجام می‌شوند. برای اطمینان، دقت کنید که اگر یک فعالیت به‌هنگام $a_i$ بعد از یک فعالیت دیر $a_j$ انجام شود، در این صورت می‌توانیم جای این دو فعالیت را عوض کنیم، و $a_i$ همچنان به‌هنگام انجام شود و $a_j$ همچنان دیر.

به طور مشابه ادعا می‌کنیم که همیشه می‌توان یک برنامه را به *صورت متعارف* (canonical form) درآورد، که در آن فعالیت‌های به‌هنگام قبل از فعالیت‌های و به ترتیب صعودی مهلت اجرا انجام می‌شوند. برای این کار، ابتدا برنامه را به صورت به‌هنگام-اول در می‌آوریم. سپس تا زمانی که دو فعالیت $a_i$ و $a_j$ با زمان پایان $k$ و $k+1$ در برنامه وجود دارند به طوری که $d_j < d_i$، مکان $a_i$ و $a_j$ را عوض می‌کنیم. از آن جایی که $a_j$ قبل از جابه‌جایی به‌هنگام است، $d_j \leq k+1$. بنابراین $d_i > k+1$، و $a_i$ بعد از جابه‌جایی همچنان به‌هنگام است. فعالیت $a_j$ هم در برنامه به جلو انداخته شده است، و بنابراین بعد از جابه‌جایی باز هم به‌هنگام است. بنابراین جستجو برای یک برنامه‌ی بهینه به مسئله‌ی یافتن یک مجموعه‌ی $A$ از فعالیت‌هایی که در برنامه‌ی بهینه به‌هنگام هستند کاهش می‌یابد. وقتی $A$ تعیین شد، می‌توانیم برنامه‌ی واقعی را با لیست کردن فعالیت‌های $A$ به ترتیب صعودی مهلت اجرا و سپس لیست کردن بقیه‌ی فعالیت‌ها ($S - A$) به ترتیب بسازیم، که یک ترتیب متعارف از برنامه‌ی بهینه است.

می‌گوییم یک مجموعه‌ی $A$ از فعالیت‌ها *مستقل* است اگر یک برنامه برای این فعالیت‌ها موجود باشد به طوری که هیچ فعالیتی دیر نشود. به وضوح، مجموعه‌ی فعالیت‌های به‌هنگام در یک برنامه یک مجموعه‌ی مستقل است. فرض کنید $\ell$ نشان‌دهنده‌ی مجموعه‌ی تمام فعالیت‌های مستقل باشد. مسئله‌ی تعیین مستقل بودن مجموعه‌ی فعالیت‌های $A$ را در نظر بگیرید. برای $t = 0,1,2,\ldots,n$، فرض کنید $N_t(A)$ نشان‌دهنده‌ی تعداد فعالیت‌هایی در $A$ باشد که مهلت اجرای آن‌ها $t$ یا زودتر است. توجه داشته باشید که برای هر مجموعه‌ی $A$، $N_0(A)$.

**لم ۱۶-۱۲**

برای هر مجموعه‌ای از فعالیت‌ها مانند $A$، عبارات زیر معادل یکدیگر هستند.

۱. مجموعه‌ی $A$ مستقل است.

۲. برای $t = 0,1,2,\ldots,n$، داریم $N_t(A) \leq t$.

۳. اگر فعالیت‌های $A$ به ترتیب صعودی مهلت اجرا مرتب شوند، آن گاه هیچ فعالیتی دیر نمی‌شود.

*اثبات* برای این که نشان دهیم (۱)، (۲) را نتیجه می‌دهد، عکس نقیض آن را اثبات می‌کنیم: اگر برای یک $t$ داشته باشیم $N_t(A) > t$، آن گاه هیچ راهی برای برنامه‌ریزی فعالیت‌های $A$ به طوری که هیچ فعالیتی دیر نشود، وجود ندارد، چرا که بیش از $t$ فعالیت وجود دارند که باید قبل از زمان $t$ انجام شوند. بنابراین، (۱) نتیجه می‌دهد (۲). اگر (۲) برقرار باشد، در این صورت (۳) هم باید برقرار باشد: هیچ راهی برای «گیر افتادن» وقتی فعالیت‌ها را به ترتیب صعودی مهلت اجرا مرتب می‌کنیم، وجود ندارد، چرا که (۲) نتیجه می‌دهد که $i$امین مهلت اجرای بزرگ حداکثر $i$ است. نهایتاً، (۳) به شکل بدیهی نتیجه می‌دهد (۱). ■

با استفاده از خصوصیت ۲ از لم ۱۶-۱۲ می‌توانیم به سادگی تعیین کنیم که یک مجموعه مستقل است یا خیر (تمرین ۱۶-۵-۲ را ببینید).

مسئله‌ی کمینه کردن مجموع جریمه‌های فعالیت‌های دیر معادل است با مسئله‌ی بیشینه کردن مجموع جریمه‌های فعالیت‌های به‌هنگام. بنابراین قضیه‌ی زیر تضمین می‌کند که می‌توانیم از یک الگوریتم حریصانه برای یافتن یک مجموعه‌ی مستقل $A$ از فعالیت‌ها با بیشینه‌ی کل جریمه‌ها استفاده کنیم.

**قضیه‌ی ۱۶-۱۳** اگر $A$ مجموعه‌ای از فعالیت‌های زمان واحد با مهلت اجرا باشد، و $\ell$ مجموعه‌ای از تمام مجموعه‌های مستقل، در این صورت سیستم مربوطه‌ی $(S, \ell)$ یک ماتروید است.

*اثبات* هر زیرمجموعه‌ای از یک مجموعه‌ی مستقل از فعالیت‌ها مطمئنا مستقل است. برای اثبات خصوصیت تعویض، فرض کنید که $B$ و $A$ مجموعه‌هایی مستقل از فعالیت‌ها هستند و $|B| > |A|$. فرض کنید $k$ بزرگ‌ترین $t$ باشد به طوری که $N_t(B) \le N_t(A)$. (چنین مقداری از $t$ وجود دارد، چرا که $N_0(A) = N_0(B) = 0$.) از آن جایی که $N_n(B) = |B|$ و $N_n(A) = |A|$، ولی $|B| > |A|$، باید داشته باشیم $k < n$ و $N_j(B) > N_j(A)$ برای تمام $j$های در محدوده‌ی $k+1 \le j \le n$. بنابراین تعداد فعالیت‌های با مهلت اجرای $k+1$ در $B$ بیشتر از $A$ است. فرض کنید $a_i$ یک فعالیت در $B - A$ با مهلت اجرای $k+1$ باشد، و $A' = A \cup \{a_j\}$.

اکنون با استفاده از لم ۱۶-۱۲ نشان می‌دهیم که $A'$ باید مستقل باشد. برای $0 \le t \le k$ داریم $N_t(A') = N_t(A) \le t$، چرا که $A$ مستقل است. برای $k < t = n$ داریم $N_t(A') \le N_t(B) \le t$، چرا که $B$ مستقل است. بنابراین $A'$ باید مستقل باشد، که اثبات ماتروید بودن $(S, \ell)$ را کامل می‌کند. ■

طبق قضیه‌ی ۱۶-۱۱ می‌توانیم از یک الگوریتم حریصانه برای یافتن یک مجموعه‌ی مستقل $A$ از فعالیت‌ها با وزن بیشینه بیابیم. سپس می‌توانیم با داشتن فعالیت‌های $A$ به عنوان فعالیت‌های به‌هنگام، یک برنامه‌ی بهینه بسازیم. این متد یک الگوریتم کارا برای برنامه‌ریزی برای فعالیت‌های زمان واحد با مهلت اجرا و جریمه برای یک پردازنده است. زمان اجرای GREEDY برابر است با $O(n^2)$، چرا که هر یک از $O(n)$ بار چک کردن استقلال انجام شده توسط الگوریتم به زمان $O(n)$ نیاز دارد (تمرین ۱۶-۵-۲ را ببینید). یک پیاده‌سازی سریع‌تر در مسئله‌ی ۱۶-۴ داده شده است. شکل ۱۶-۷ یک مثال از

مسئله‌ی برنامه‌ریزی برای فعالیت‌های زمان واحد با مهلت اجرا و جریمه برای یک پردازنده ارائه می‌کند. در این مثال، الگوریتم حریصانه فعالیت‌های $a_1$، $a_2$، $a_3$، و $a_4$ را انتخاب، و سپس $a_5$ و $a_6$ را رد می‌کند (به ترتیب چون $N_4(\{a_1,a_2,a_3,a_4,a_5\}) = 5$ و $N_4(\{a_1,a_2,a_3,a_4,a_6\}) = 5$)، و نهایتاً $a_7$ را می‌پذیرد. برنامه‌ی بهینه‌ی نهایی

$$\langle a_2,a_4,a_1,a_3,a_7,a_5,a_6\rangle$$

است که کل جریمه‌ی آن برابر است با $w_5 + w_6 = 50$.

## تمرین‌ها

۱-۵-۱۶ نمونه‌ی مسئله‌ی برنامه‌ریزی داده شده در شکل ۱۶-۷ را با جایگزینی هر جریمه‌ی $w_i$ با $80 - w_i$ حل کنید.

۲-۵-۱۶ نشان دهید که چطور می‌توان از خصوصیت ۲ لم ۱۶-۱۲ برای تعیین استقلال یک مجموعه‌ی $A$ از فعالیت‌ها در زمان $O(|A|)$ استفاده کرد.

# مسائل

۱-۱۶ ***خرد کردن پول***

مسئله‌ی خرد کردن $n$ سنت با استفاده از کم‌ترین تعداد سکه‌ی ممکن را در نظر بگیرید. فرض کنید که مقدار هر سکه یک عدد صحیح است.

I. یک الگوریتم حریصانه ارائه کنید که با سکه‌های ۲۵ سنتی، ۱۰ سنتی، ۵ سنتی و ۱ سنتی پول را خرد می‌کند. اثبات کنید که الگوریتم شما به یک جواب بهینه ختم می‌شود.

II. فرض کنید سکه‌های موجود توان‌هایی از $c$ موجود هستند، یعنی واحدها عبارتند از $c^0, c^1, \ldots, c^k$ برای یک عدد صحیح $c > 1$ و $k \geq 1$. نشان دهید که یک الگوریتم حریصانه همیشه به یک جواب بهینه ختم می‌شود.

III. مجموعه‌ای از واحدهای پولی برای سکه‌ها بدهید که در آن الگوریتم حریصانه به جواب بهینه نمی‌رسد. مجموعه‌ی شما باید شامل سکه‌ی ۱ سنتی باشد تا برای هر مقدار $n$ یک جواب داشته باشیم.

| | وظیفه | | | | | | |
|---|---|---|---|---|---|---|---|
| $a_i$ | 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** یک مثال از مسئله‌ی برنامه‌ریزی برای فعالیت‌های زمان واحد با مهلت اجرا و جریمه برای یک پردازنده.

IV. یک الگوریتم با زمان $O(nk)$ بدهید که برای هر مجموعه از $k$ سکه‌های مختلف پول را خرد می‌کند، با فرض این که یکی از سکه‌ها یک سنتی است.

### ۱۶-۲ برنامه‌ریزی برای کمینه کردن متوسط زمان پایان

فرض کنید یک مجموعه‌ی $S = \{a_1, a_2, ..., a_n\}$ از فعالیت‌ها داریم، که در آن فعالیت $a_i$ پس از شروع به $p_i$ واحد زمان پردازش برای پایان یافتن نیاز دارد. یک کامپیوتر برای انجام این فعالیت‌ها داریم، و این کامپیوتر در هر لحظه فقط می‌تواند یکی از این فعالیت‌ها را اجرا کند. فرض کنید $c_i$ *زمان پایان فعالیت* $a_i$ باشد، یعنی زمانی که اجرای فعالیت $a_i$ پایان می‌یابد. هدف کمینه کردن متوسط زمان پایان فعالیت‌ها است، یعنی کمینه کردن عبارت $(1/n)\sum_{i=1}^{n} c_i$. به عنوان مثال فرض کنید که دو فعالیت $a_1$ و $a_2$ داریم، و $p_1 = 3$ و $p_2 = 5$. همچنین فرض کنید برنامه طوری است که ابتدا $a_2$ و سپس $a_1$ انجام می‌شود. در این صورت $c_2 = 5$، $c_1 = 8$، و متوسط زمان پایان برابر است با $(8+5)/2 = 6/5$. ولی اگر ابتدا فعالیت $a_1$ انجام شود، آن گاه $c_1 = 3$، $c_2 = 8$، و زمان متوسط پایان برابر است با $(3+8)/2 = 5/5$.

I. یک الگوریتم ارائه کنید که طوری برای فعالیت‌ها برنامه‌ریزی می‌کند که متوسط زمان پایان آن‌ها کمینه شود. هر فعالیت باید بدون وقفه اجرا شود، یعنی فعالیت $a_i$ پس از شروع باید بدون وقفه به اندازه‌ی $p_i$ واحد زمان اجرا شود تا پایان یابد. اثبات کنید که الگوریتم شما متوسط زمان پایان را کمینه می‌کند، و زمان اجرای الگوریتم خود را مشخص کنید.

اکنون فرض کنید که تمام فعالیت‌ها در لحظه‌ی اول در دسترس نیستند. یعنی هر فعالیت یک *زمان آزادسازی* (release time) $r_i$ دارد که بعد از آن زمان می‌تواند اجرا شود. همچنین فرض کنید که اجازه‌ی انجام *وقفه* (preemption) در فعالیت‌ها را داریم، و می‌توانیم یک فعالیت را متوقف کرده و بعداً دوباره اجرای آن را ادامه دهیم. مثلاً ممکن است یک فعالیت $a_i$ با زمان اجرای $p_i = 6$ در زمان ۱ شروع شده و در زمان ۴ متوقف شود. سپس در زمان ۱۰ دوباره شروع شده و باز در زمان ۱۱ متوقف شود، و نهایتاً در زمان ۱۳ شروع شده و در زمان ۱۵ پایان یابد. فعالیت $a_i$ در کل برای ۶ واحد زمان اجرا شده است، ولی زمان اجرای آن به سه بخش تقسیم شده است. می‌گوییم که زمان پایان $a_i$، ۱۵ بوده است. یک الگوریتم برای این سناریوی جدید ارائه کنید که متوسط زمان پایان را کمینه کند. اثبات کنید که الگوریتم شما این کار را به درستی انجام می‌دهد، و زمان اجرای الگوریتم خود را تعیین کنید.

### ۱۶-۳ زیرگراف‌های بدون دور

I. *ماتریس تلاقی* (incidence matrix) یک گراف بدون جهت $G = (V, E)$، یک ماتریس با $M$ اندازه‌ی $|V| \times |E|$ است به طوری که $M_{ve} = 1$ اگر یال $e$ با رأس $v$ برخورد داشته باشد، و در غیر این صورت $M_{ve} = 0$. بحث کنید که مجموعه‌ای از ستون‌های $M$ بر روی اعداد صحیح

به پیمانه‌ی ۲ مستقل خطی است اگر و فقط اگر مجموعه‌ی یال‌های مربوطه بدون دور باشد.

II. فرض کنید که یک وزن نامنفی $w(e)$ به هر یال در یک گراف بدون جهت $G=(V,E)$ نسبت داده‌ایم. یک الگوریتم کارا ارائه کنید که یک زیرمجموعه‌ی بدون دور در $E$ با وزن بیشینه می‌یابد.

III. فرض کنید $G=(V,E)$ یک گراف جهت دار دلخواه باشد، و $(E,\ell)$ بدین صورت تعریف شده باشد که $A\in\ell$ اگر و فقط اگر $A$ شامل هیچ دور جهت داری نباشد. یک مثال از یک گراف $G$ بدهید به طوری که سیستم مربوطه‌ی $(E,\ell)$ یک ماتروید نباشد. مشخص کنید که در این جا کدام یک از شرایط ماتروید بودن برقرار نیست.

IV. *ماتریس تلاقی* یک گراف جهت‌دار بدون طوقه‌ی $G=(V,E)$ یک ماتریس $M$ با اندازه‌ی $|V|\times|E|$ است به طوری که $M_{ve}=-1$ اگر یال $e$ از رأس $v$ خارج می‌شود، $M_{ve}=1$ اگر یال $e$ وارد رأس $v$ می‌شود، و در غیر این صورت $M_{ve}=0$. بحث کنید که اگر یک مجموعه از ستون‌های $M$ مستقل خطی باشد، در این صورت مجموعه‌ی یال‌های مربوطه دارای دور جهت دار نیست.

V. تمرین ۱۶-۴-۲ می‌گوید که مجموعه‌ی تمام مجموعه‌های مستقل خطی از ستون‌های هر ماتریس $M$ یک ماتروید است. به دقت توضیح دهید که چرا نتیجه‌ی قسمت‌های III و IV با این مسئله تناقض ندارد. چگونه است که نمی‌توان یک تناظر کامل میان بدون دور بودن مجموعه‌ی یال‌ها و مستقل خطی بودن مجموعه‌ی ستون‌های مربوطه در ماتریس تلاقی برقرار کرد؟

### ۱۶-۴ راه‌های دیگر برنامه‌ریزی

الگوریتم زیر را برای مسئله‌ی بخش ۱۶-۵ (برنامه‌ریزی برای فعالیت‌های زمان واحد با مهلت اجرا و جریمه) در نظر بگیرید. فرض کنید که در ابتدا تمام $n$ واحد زمان خالی باشد، که در آن واحد زمان $i$ام همان یک واحد زمان است که در زمان $i$ پایان می‌یابد. فعالیت‌ها را به ترتیب نزولی جریمه‌ها در نظر می‌گیریم. وقتی فعالیت $a_j$ را در نظر می‌گیریم، اگر یک واحد زمانی در مهلت اجرای $a_j$ و یا قبل از آن وجود داشت که هنوز خالی بود، $a_j$ را به آخرین واحد زمانی ممکن نسبت می‌دهیم و آن را پر می‌کنیم. اگر چنین زمانی وجود نداشت، $a_j$ را به آخرین زمان خالی ممکن نسبت می‌دهیم.

I. بحث کنید که این الگوریتم همیشه به جواب بهینه ختم می‌شود.

II. از جنگل مجموعه‌های منفصل ارائه شده در بخش ۲۱-۳ برای پیاده‌سازی این الگوریتم به صورت کارا استفاده کنید. فرض کنید که مجموعه‌ی فعالیت‌های ورودی قبلاً به ترتیب نزولی جریمه مرتب شده‌اند. زمان اجرای پیاده‌سازی خود را تحلیل کنید.

### ۱۶-۵ کَش کردن برون خطی

کامپیوترهای مدرن از یک کَش (cache) برای ذخیره‌ی مقدار کمی از داده‌ها در یک حافظه‌ی

سریع استفاده می‌کنند. با این که یک برنامه ممکن است به حجم زیادی از داده‌ها دسترسی پیدا کند، با ذخیره‌ی زیرمجموعه‌ی کوچکی از حافظه‌ی اصلی در *کَش* – یک حافظه‌ی کوچک، ولی سریع – زمان کلی دسترسی می‌تواند به شدت کاهش یابد. وقتی یک برنامه‌ی کامپیوتری اجرا می‌شود، دنباله‌ای از $n$ درخواست حافظه به شکل $\langle r_1, r_2, \ldots, r_n \rangle$ خواهد داشت، که در آن هر درخواست حافظه برای یک عنصر داده‌ای خاص است. برای مثال، برنامه‌ای که به چهار عنصر حافظه‌ی مجزای $\{a,b,c,d\}$ دسترسی پیدا می‌کند، ممکن است یک دنباله‌ی درخواست به شکل $\langle d,b,d,b,d,a,c,d,b,a,c,b \rangle$ داشته باشد. فرض کنید $k$ اندازه‌ی کَش باشد. وقتی کَش حاوی $k$ عنصر است، و برنامه برای $k+1$امین عنصر درخواست می‌کند، سیستم باید برای این درخواست و درخواست‌های بعدی تصمیم بگیرد که کدام $k$ عنصر را در کَش نگه دارد. به طور دقیق‌تر، برای هر درخواست $r_i$، الگوریتم مدیریت کَش چک می‌کند که آیا عنصر $r_i$ در کَش وجود دارد یا نه. اگر وجود داشته باشد، یک *برخورد کَش*(cache hit) خواهیم داشت، و در غیر این صورت یک *خطای کَش*(cache miss) خواهیم داشت. هنگام رخ دادن یک خطای کَش، سیستم $r_i$ را از حافظه‌ی اصلی بازیابی می‌کند، و برنامه‌ی مدیریت کَش باید تصمیم بگیرد که $r_i$ را در کَش نگه دارد یا نه. اگر تصمیم بگیرد که $r_i$ را نگه دارد، و $k$ عنصر در کَش وجود داشته باشند، آن گاه باید یک عنصر حذف شود تا جا برای $r_i$ باز شود. برنامه‌ی مدیریت کَش داده‌ها را به هدف کمینه کردن خطاهای کَش در کل دنباله‌ی درخواست‌ها، از کَش حذف می‌کند.

معمولاً، کَش کردن یک مسئله‌ی برخط (online) است. یعنی بدون دانستن تقاضاهای بعدی باید در مورد نگه داشتن یا حذف داده‌ها تصمیم بگیریم. ولی در این جا، نسخه‌ی آفلاین (off-line) این مسئله را در نظر می‌گیریم، که در آن از قبل کل دنباله‌ی $n$ درخواست و اندازه‌ی کَش ($k$) به ما داده شده است، و می‌خواهیم تعداد کل خطاهای کَش را کمینه کنیم.

این مسئله‌ی آفلاین را می‌توان به کمک یک استراتژی حریصانه به نام *دورترین در آینده* (furthest-in-future) حل کنیم، که عنصری در کَش را برای حذف انتخاب می‌کند که درخواست بعدی برای آن عنصر در دنباله دورتر از تمام عناصر دیگر است.

I. یک شبه‌کد برای یک الگوریتم مدیریت کَش بنویسید که از استراتژی دوترین در آینده استفاده می‌کند. ورودی باید یک دنباله‌ی $\langle r_1, r_2, \ldots, r_n \rangle$ از درخواست‌ها باشد، به همراه $k$ که نشان‌دهنده‌ی اندازه‌ی کَش است، و خروجی باید دنباله‌ای از تصمیم‌ها در مورد این باشد که کدام یک از داده‌ها (در صورت وجود) باید هنگام هر درخواست حذف شوند. زمان اجرای الگوریتم شما چقدر است؟

II. نشان دهید که استراتژی دورترین در آینده از زیرساختار بهینه پیروی می‌کند.

III. اثبات کنید که استراتژی دورترین در آینده کم‌ترین تعداد خطاهای ممکن را در کَش تولید می‌کند.


# تحلیل سرشکن

## ۱۷-۰ مقدمه

در یک *تحلیل سرشکن* (amortized analysis)، زمان مورد نیاز برای انجام دنباله‌ای از اعمال بر روی یک ساختمان داده با میانگین گرفتن بر روی تمام اعمال انجام شده محاسبه می‌شود. به کمک تحلیل سرشکن می‌توان نشان داد که اگر از دنباله‌ای از اعمال میانگین بگیریم، هزینه‌ی متوسط کم خواهد بود، حتی اگر هزینه‌ی یکی از عملیات در این دنباله زیاد باشد. تحلیل سرشکن از این لحاظ با تحلیل حالت متوسط متفاوت است که در تحلیل سرشکن از احتمالات استفاده نمی‌شود؛ تحلیل سرشکن *کارایی متوسط هر یک از عملیات را در بدترین حالت* تضمین می‌کند.

سه بخش اول این فصل، سه تا از رایج‌ترین تکنیک‌های مورد استفاده در تحلیل سرشکن را پوشش می‌دهند. بخش ۱۷-۱ با تحلیل متراکم آغاز می‌شود، که در آن یک کران بالای $T(n)$ برای هزینه‌ی دنباله‌ای از $n$ عمل پیدا می‌کنیم. در این صورت میانگین هزینه برای هر یک از اعمال برابر است با $T(n)/n$. هزینه‌ی میانگین را به عنوان هزینه‌ی سرشکن هر یک از اعمال در نظر می‌گیریم، و بنابراین هزینه‌ی سرشکن تمام اعمال با یکدیگر برابر است.

در بخش ۱۷-۲ متد حسابداری معرفی می‌شود، که در آن هزینه‌ی سرشکن هر یک از اعمال را به صورت جداگانه بررسی می‌کنیم. وقتی بیش از یک نوع از عملیات موجود باشد، هر نوع ممکن است هزینه‌ی سرشکن متفاوتی داشته باشد. در متد حسابداری به بعضی از اعمال در دنباله ابتدا هزینه‌ای اضافی تعلق می‌گیرد، و این هزینه‌ی اضافی به عنوان «اعتبار پیش پرداختی» بر روی اشیای خاص در ساختمان داده ذخیره می‌شود. بعداً در ادامه‌ی دنباله از این اعتبار برای پرداخت برای اعمالی که هزینه‌ی تعلق گرفته به آن‌ها کم‌تر از هزینه‌ی واقعی آن‌ها بوده استفاده می‌شود.

در بخش ۱۷-۳ متد پتانسیل بحث خواهد شد، که از این نظر شبیه متد حسابداری است که هزینه‌ی سرشکن هر عمل را جداگانه محاسبه می‌کند و ممکن است ابتدا به اعمال هزینه‌ی اضافی نسبت دهد و در مقابل بعداً هزینه‌ی کم‌تری از هزینه‌ی واقعی برای آن‌ها در نظر گیرد. متد پتانسیل اعتبارها را به عنوان «انرژی پتانسیل» کل ساختمان داده در نظر می‌گیرد، به جای این که اعتبار هر یک از اشیای درون ساختمان داده را جداگانه ذخیره کند.

برای بررسی این سه متد از دو مثال استفاده خواهیم کرد. یکی از این مثال‌ها، یک پشته است با عملیات اضافه‌ی MULTIPOP، که چندین شیئ را به صورت یکباره از پشته بازیابی می‌کند. مثال دیگر یک شمارنده‌ی دودویی است که به کمک یک عمل INCREMENT از ۰ به سمت بالا می‌شمارد.

هنگام خواندن این فصل، به خاطر داشته باشید که هزینه‌های اضافی نسبت داده شده هنگام یک تحلیل سرشکن فقط برای اهداف تحلیلی است. احتیاجی نیست — و نباید — که این هزینه‌ها در کد مشاهده شوند. به عنوان مثال، اگر هنگام استفاده از متد حسابداری یک اعتبار به یک شیئ نسبت $x$ داده می‌شود، نیازی نیست که یک مقدار مناسب به یک خصوصیت $x.credit$ در کد تعلق گیرد.

دید بدست آمده از انجام یک تحلیل سرشکن بر روی یک ساختمان داده‌ی خاص می‌تواند در طراحی بهینه به ما کمک کند. مثلاً در بخش ۱۷-۴، از متد پتانسیل برای تحلیل یک جدول پویا با اندازه‌ی متغیر استفاده می‌شود.

## ۱۷-۱ تحلیل متراکم

در *تحلیل متراکم* (aggregate analysis) نشان می‌دهیم که برای هر $n$، دنباله‌ای از $n$ عمل در به‌ترین حالت به زمان $T(n)$ نیاز دارد. بنابراین در بدترین حالت، هزینه‌ی متوسط، یا *هزینه‌ی سرشکن* (amortized cost) برای هر یک از اعمال $T(n)/n$ است. توجه کنید که این هزینه‌ی سرشکن برای تمام اعمال به کار می‌رود، حتی وقتی که انواع مختلفی از اعمال در یک دنباله وجود دارند. دو روش دیگری که در این فصل بررسی خواهیم کرد، یعنی متد حسابداری و متد پتانسیل، هزینه‌های سرشکن متفاوتی را به انواع مختلف عملیات نسبت می‌دهند.

### اعمال پشته

برای اولین مثال از تحلیل متراکم، پشته‌هایی را تحلیل می‌کنیم که با یک عملیات جدید تکمیل شده‌اند. در بخش ۱۰-۱ دو عملیات اولیه‌ی پشته معرفی شدند، که هر کدام در زمان $O(1)$ انجام می‌شوند.

PUSH$(S, x)$ شیئ $x$ را در پشته‌ی $S$ می‌نشاند.

POP$(S)$ بالای پشته‌ی $S$ را بازیابی کرده و شیئ بازیابی شده را بازمی‌گرداند. فراخوانی POP بر روی یک پشته‌ی خالی، یک خطا تولید می‌کند.

از آن جایی که هر یک از این اعمال در زمان $O(1)$ انجام می‌شوند، اجازه دهید فرض کنیم که هزینه‌ی هر کدام ۱ است. بنابراین هزینه‌ی کلی دنباله‌ای از $n$ عملیات PUSH و POP، $n$ خواهد بود، و

همچنین زمان اجرای $n$ عمل $\theta(n)$ است.

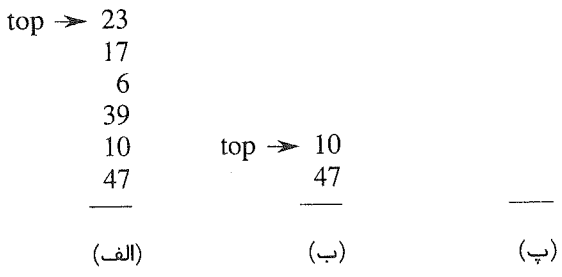
اکنون عملیات پشته‌ی MUTLIPOP$(S,k)$ را اضافه می‌کنیم، که $k$ عنصر بالایی پشته‌ی $S$ را حذف می‌کند، یا در صورتی که کم‌تر از $k$ عنصر در پشته موجود باشد تمام پشته را بازیابی می‌کند. مسلماً فرض می‌کنیم که $k$ مثبت است؛ در غیر این صورت عملیات MULTIPOP تغییری در پشته ایجاد نمی‌کند. در شبه‌کد زیر، عملیات STACK-EMPTY در صورتی TRUE را بازمی‌گرداند که هیچ عنصری در پشته نباشد، و در غیر این صورت FALSE را بازمی‌گرداند.

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2      POP(S)
3      k = k - 1
```

شکل ۱۷-۱ مثالی از MULTIPOP را نشان می‌دهد.

زمان اجرای MULTIPOP$(S,k)$ بر روی پشته‌ای با $s$ عنصر چیست؟ زمان اجرای واقعی نسبت به تعداد POPهای انجام شده خطی است، و بنابراین کافی است که MULTIPOP را با هزینه‌ی ۱ برای هر PUSH و POP تحلیل کنیم. تعداد تکرارهای حلقه‌ی while برابر است با $\min(s,k)$. برای هر تکرار حلقه، در خط ۲ یک بار POP فراخوانی می‌شود. بنابراین هزینه‌ی کلی MULTIPOP برابر است با $\min(s,k)$، و زمان اجرای واقعی یک تابع خطی از این هزینه است.

اجازه دهید دنباله‌ای از $n$ عملیات PUSH، POP، و MULTIPOP را بر روی یک پشته‌ی خالی تحلیل کنیم. هزینه‌ی یک عملیات MULTIPOP در دنباله در بدترین حالت $O(n)$ است، چرا که اندازه‌ی پشته حداکثر $n$ است. بنابراین بدترین حالت زمان یک عملیات پشته $O(n)$ است، و از این رو دنباله‌ای از $n$ عمل $O(n^2)$ هزینه خواهد داشت، چرا که ممکن است $O(n)$ عملیات MULTIPOP داشته باشیم که هر یک هزینه‌ی $O(n)$ دارد. با این که این تحلیل درست است، نتیجه‌ی $O(n^2)$ که با در نظر گرفتن بدترین حالت هر یک از اعمال به صورت جداگانه به دست آمده است، نزدیک نیست.




شکل ۱۷-۱ عملیات MULTIPOP بر روی پشته‌ی $S$، که در ابتدا مانند (الف) است. چهار عنصر بالایی توسط MULTIPOP$(S,۴)$ بازیابی می‌شوند، که نتیجه‌ی آن را در (ب) مشاهده می‌کنید. عملیات بعدی MULTIPOP$(S,۷)$ است، که پشته را خالی می‌کند – که در بخش (پ) نشان داده شده است – چرا که کم‌تر از ۷ عنصر در پشته باقی مانده‌اند.

با استفاده از یک تحلیل متراکم، می‌توانیم یک کران بالای بهتر به دست آوریم که کل دنباله را یکجا در نظرمی‌گیرد. در واقع، با این که یک عمل MULTIPOP می‌تواند پرهزینه باشد، هر دنباله‌ای از $n$ عملیات PUSH، POP، و MULTIPOP بر روی یک پشته‌ی خالی حداکثر می‌تواند $O(n)$ هزینه داشته باشد. چرا؟ هر شیئ هر بار که نشانده می‌شود، حداکثر می‌تواند یک بار هم بازیابی شود. بنابراین، تعداد دفعاتی که می‌توان POP را بر روی یک پشته‌ی ناتهی فراخوانی کرد، شامل فراخوانی‌های درون MULTIPOP، حداکثر برابر است با تعداد اعمال PUSH، که آن هم حداکثر برابر است با $n$. برای هر مقدار $n$، هر دنباله‌ای از $n$ عملیات PUSH، POP، و MULTIPOP در کل به زمان $O(n)$ نیاز خواهد داشت. هزینه‌ی متوسط یک عمل برابر است با $O(n)/n = O(1)$. در تحلیل متراکم، هزینه‌ی هر یک از عملیات را برابر با هزینه‌ی متوسط در نظر می‌گیریم. بنابراین، در این مثال هر یک از اعمال بر روی پشته هزینه‌ی سرشکنی برابر با $O(1)$ خواهند داشت.

دوباره تأکید می‌کنیم که با این که هزینه‌ی متوسط، و بنابراین زمان اجرای یک عمل پشته برابر با $O(1)$ است، از هیچ گونه منطق احتمالاتی استفاده نشده است. ما در واقع یک کران *بدترین حالت* $O(n)$ برای زمان اجرای دنباله‌ای از $n$ عمل به دست آوردیم. تقسیم این هزینه‌ی کلی بر $n$ هزینه‌ی متوسط برای هر یک از اعمال، یا هزینه‌ی سرشکن را به ما می‌دهد.

## افزایش یک شمارنده‌ی دودویی

به عنوان یک مثال دیگر از تحلیل متراکم، مسئله‌ی پیاده‌سازی یک شمارنده‌ی دودویی را در نظر بگیرید که از ۰ به سمت بالا می‌شمارد. از یک آرایه‌ی $A[0..k-1]$ از بیت‌ها به عنوان شمارنده استفاده می‌کنیم، که در آن $A.length = k$. کم‌ارزش‌ترین بیت یک عدد دودویی $x$ که در شمارنده ذخیره شده است، در $A[0]$ و پرارزش‌ترین بیت آن در $A[k-1]$ خواهد بود، به طوری که $x = \sum_{i=0}^{k-1} A[i].2^i$. در ابتدا $x = 0$، و بنابراین $A[i] = 0$ برای $i = 0,1,...,k-1$. برای اضافه کردن ۱ (به پیمانه‌ی $2^k$) به مقدار شمارنده از رویه‌ی زیر استفاده می‌کنیم.

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

شکل ۱۷-۲ یک شمارنده‌ی دودویی را نشان می‌دهد که مقدار اولیه‌ی آن ۰ است، و سپس ۱۶ بار افزایش داده می‌شود و مقدار آن به ۱۶ می‌رسد. در آغاز هر بار تکرار حلقه‌ی while در خطوط ۲-۴، می‌خواهیم یک ۱ به مکان $i$ام اضافه کنیم. اگر $A[i] = 1$، اضافه کردن ۱ مقدار بیت $i$ را ۰ می‌کند و یک رقم نقلی ۱ تولید می‌کند، که باید در تکرار بعدی به مکان $i+1$ اضافه شود. در غیر این صورت حلقه پایان می‌یابد، و سپس اگر $i < k$، می‌دانیم که $A[i] = 0$، و بنابراین اضافه کردن ۱ به مکان $i$، و تبدیل ۰ به ۱، اجرای رویه را در خط ۶ پایان می‌دهد. هزینه‌ی هر یک از اعمال INCREMENT نسبت

به تعداد بیت‌های تغییر کرده خطی است.

مانند مثال پشته، یک تحلیل بی‌دقت به یک کران صحیح منجر می‌شود که نزدیک نیست. یک اجرای INCREMENT در بدترین حالت، که در آن تمام آرایه‌ی شامل ۱ است، به زمان $\theta(k)$ نیاز دارد. بنابراین، دنباله‌ای از $n$ عمل INCREMENT بر روی یک شمارنده با مقدار اولیه‌ی ۰ در بدترین حالت در زمان $O(nk)$ اجرا خواهد شد.

می‌توانیم تحلیل خود را دقیق‌تر کنیم و نشان دهیم که در بدترین حالت هزینه‌ی دنباله‌ای از $n$ عمل INCREMENT برابر با $O(n)$ خواهد بود، بدین صورت که مشاهده می‌کنیم که در هر بار فراخوانی INCREMENT، تمام بیت‌ها تغییر نمی‌کنند. همان طور که شکل ۱۷-۲ نشان می‌دهد، $A[0]$ با هر بار فراخوانی INCREMENT تغییر می‌کند. بیت پرارزش بعدی، $A[1]$، به صورت یکی در میان تغییر می‌کند: دنباله‌ای از $n$ عمل INCREMENT بر روی یک شمارنده‌ی دودویی با مقدار اولیه‌ی صفر بیت $A[1]$ را $\lceil n/2 \rceil$ بار تغییر می‌دهد. به صورت مشابه بیت $A[2]$ با هر چهار بار افزایش یک بار تغییر می‌کند، یا به عبارتی $\lceil n/4 \rceil$ بار برای دنباله‌ای از $n$ عمل INCREMENT. به صورت کلی برای $i = 0,1,\ldots,\lceil \lg n \rceil$، در دنباله‌ای از $n$ عمل INCREMENT بر روی یک شمارنده‌ی دودویی با مقدار اولیه‌ی صفر، بیت $A[i]$ به تعداد $\lceil n/2^i \rceil$ بار تغییر می‌کند. برای $i \geq k$، بیت $A[i]$ اصلاً وجود ندارد، و بنابراین نمی‌تواند تغییر کند. پس تعداد کل تغییرات بر روی بیت‌ها در یک دنباله برابر است با

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i}$$
$$= 2n$$

| مقدار شمارنده | A[7] | A[6] | A[5] | A[4] | A[3] | A[2] | A[1] | A[0] | هزینه‌ی کل |
|---|---|---|---|---|---|---|---|---|---|
| 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 شمارنده‌ی دودویی ۸ بیتی که با دنباله‌ای از ۱۶ عمل INCREMENT، مقدار آن از ۰ به ۱۶ می‌رسد. بیت‌هایی که برای رسیدن به مقدار بعدی تغییر می‌کنند در سمت راست با سایه مشخص شده‌اند. توجه داشته باشید که کل هزینه هیچ وقت بیش‌تر از دو برابر تعداد اعمال INCREMENT نیست.

طبق تساوی (الف-۶). بنابراین بدترین حالت زمان اجرای دنباله‌ای از $n$ عمل INCREMENT بر روی یک شمارنده با مقدار اولیه‌ی صفر برابر است با $O(n)$. بنابراین هزینه‌ی متوسط هر یک از اعمال، و همچنین هزینه‌ی سرشکن برای هر یک از اعمال برابر است با $O(n)/n = O(1)$.

### تمرین‌ها

۱۷-۱-۱ اگر مجموعه‌ی اعمال پشته شامل یک عمل MULTIPUSH هم باشد، که $k$ عنصر را در پشته می‌نشاند، آیا کران $O(1)$ بر روی هزینه‌ی سرشکن اعمال پشته همچنان برقرار خواهد بود؟

۱۷-۱-۲ نشان دهید که اگر در مثال شمارنده‌ی $k$ بیتی، یک عمل DECREMENT (کاهش) هم داشته باشیم، دنباله‌ای از $n$ عمل می‌تواند به اندازه‌ی $\theta(nk)$ هزینه داشته باشد.

۱۷-۱-۳ دنباله‌ای از $n$ عمل بر روی یک ساختمان داده انجام می‌شود. اگر $i$ توانی از ۲ باشد، عمل $i$ام هزینه‌ای برابر با $i$ خواهد داشت، و در غیر این صورت هزینه‌ی آن ۱ خواهد بود. با استفاده از تحلیل متراکم هزینه‌ی سرشکن هر عمل را به دست آورید.

## ۱۷-۲ متد حسابداری

در *متد حسابداری* (accounting method) از تحلیل سرشکن، به اعمال مختلف هزینه‌های مختلف نسبت می‌دهیم، که بعضی از این هزینه‌ها ممکن است بیش‌تر یا کم‌تر از هزینه‌ی واقعی باشند. به هزینه‌ای که به یک عملیات نسبت می‌دهیم هزینه‌ی سرشکن گفته می‌شود. وقتی هزینه‌ی سرشکن یک عملیات از هزینه‌ی واقعی آن بیش‌تر می‌شود، تفاوت این دو هزینه به یک شیئ خاص در ساختمان داده به نام *اعتبار* (credit) نسبت داده می‌شود. از اعتبار بعداً برای پرداخت برای اعمالی استفاده می‌شود که هزینه‌ی سرشکن آن‌ها کم‌تر از هزینه‌ی واقعی است. بنابراین می‌توان به هزینه‌ی سرشکن به صورت هزینه‌ای نگاه کرد که بین هزینه‌ی واقعی و اعتبار تقسیم می‌شود، که این اعتبار یا مثبت است و یا منفی. اعمال مختلف می‌توانند هزینه‌های سرشکن مختلف داشته باشند. این متد با تحلیل متراکم متفاوت است، که در آن تمام اعمال دارای هزینه‌ی سرشکن یکسانی بودند.

هزینه‌ی سرشکن اعمال باید به دقت انتخاب شود. اگر بخواهیم به کمک تحلیل سرشکن نشان دهیم که در بدترین حالت هزینه‌ی متوسط برای هر یک از اعمال کوچک است، کل هزینه‌ی سرشکن یک دنباله باید کران بالایی برای کل هزینه‌ی واقعی آن باشد. به علاوه، مانند تحلیل متراکم، این رابطه باید برای تمام دنباله‌ها برقرار باشد. اگر هزینه‌ی واقعی عملیات $i$ام را با $c_i$ و هزینه‌ی سرشکن آن را با $\widehat{c_i}$ نشان دهیم، باید داشته باشیم

$$\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i \qquad (۱-۱۷)$$

برای تمام دنباله‌های با $n$ عملیات. کل اعتبار ذخیره شده در ساختمان داده برابر است با تفاوت میان کل هزینه‌ی سرشکن و کل هزینه‌ی واقعی، یا $\sum_{i=1}^{n} \widehat{c_i} - \sum_{i=1}^{n} c_i$. طبق نامساوی (۱۷-۱) کل اعتبار نسبت داده شده به ساختمان داده باید همیشه نامنفی باشد. اگر هرگاه به اعتبار اجازه داده شود که منفی شود (نتیجه‌ی هزینه‌های سرشکن پایین برای اعمال ابتدایی و قول جبران آن در اعمال بعدی)، در این صورت کل هزینه‌های سرشکن در آن لحظه از کل هزینه‌های واقعی کم‌تر خواهد شد؛ برای دنباله‌ای از عملیات تا آن‌جا کل هزینه‌های سرشکن کران بالایی برای کل هزینه‌های واقعی نخواهد بود. بنابراین باید مواظب باشیم که کل اعتبار درون یک ساختان داده هیچ گاه منفی نشود.

## اعمال پشته

برای روشن شدن متد حسابداری از تحلیل سرشکن، اجازه دهید به مثال پشته بازگردیم. به یاد بیاورید که هزینه‌های واقعی اعمال به صورت

```
PUSH        1
POP         1
MULTIPOP    min(k, s)
```

بود که در آن $k$ آرگومان فرستاده شده به MULTIPOP است، و $s$ اندازه‌ی پشته وقتی که رویه فراخوانی می‌شود. اجازه دهید هزینه‌های سرشکن زیر را به این اعمال نسبت دهیم:

```
PUSH        2
POP         0
MULTIPOP    0
```

توجه کنید که هزینه‌ی سرشکن MULTIPOP ثابت (۰) است، در حالی که هزینه‌ی واقعی متغیر است. در این جا، هر سه هزینه‌ی سرشکن ثابت است، هر چند به طور کلی هزینه‌ی سرشکن اعمال تحت شرایط مختلف ممکن است متفاوت باشد، حتی به صورت حدی.

اکنون نشان خواهیم داد که می‌توانیم با استفاده از این هزینه‌های سرشکن، هزینه‌های واقعی تمام اعمال پشته را پرداخت کنیم. فرض کنید که از یک اسکناس یک دلاری برای نشان دادن هر واحد هزینه استفاده می‌کنیم. با یک پشته‌ی خالی شروع می‌کنیم. قیاس بخش ۱۰-۱ میان یک ساختمان داده‌ی پشته و پشته‌ای از بشقاب‌ها در یک رستوران را به خاطر آورید. وقتی یک بشقاب را بر روی یک پشته قرار می‌دهیم، از یک اسکناس ۱ دلاری برای پرداخت هزینه‌ی واقعی نشاندن استفاده می‌کنیم، و ۱ دلار اعتبار (از ۲ دلار هزینه‌ی نسبت داده شده) برای ما باقی می‌ماند، که آن را بر روی بشقاب قرار می‌دهیم. در هر لحظه‌ای از زمان، بر روی هر بشقابی از پشته ۱ دلار اعتبار قرار دارد.

یک دلار ذخیره شده بر روی بشقاب، هزینه‌ی بازیابی آن از روی پشته است. وقتی یک عملیات POP را اجرا می‌کنیم، هیچ هزینه‌ای به این عملیات نسبت نمی‌دهیم و در عوض هزینه‌ی واقعی آن را با استفاده از اعتبار ذخیره شده در پشته پرداخت می‌کنیم. برای بازیابی یک بشقاب، یک دلار اعتبار بر روی آن را برمی‌داریم و از آن برای پرداخت هزینه‌ی واقعی عملیات استفاده می‌کنیم. بنابراین، با مقداری هزینه‌ی بیشتر برای عملیات PUSH، احتیاجی به هزینه کردن برای POP نداریم.

به علاوه نیازی به هزینه کردن برای MULTIPOP هم نیست. برای بازیابی اولین بشقاب، اسکناس ۱ دلاری روی آن را برمی‌داریم و از آن برای پرداخت هزینه‌ی POP استفاده می‌کنیم. برای بازیابی دومین بشقاب، باز هم یک اعتبار ۱ دلاری بر روی آن بشقاب داریم که می‌توانیم از آن برای پرداخت هزینه‌ی POP استفاده کنیم، و به همین ترتیب تا آخر. بنابراین، همیشه به مقدار کافی برای پرداخت برای عملیات MULTIPOP اعتبار داریم. به عبارت دیگر از آن جایی که هر بشقاب بر روی خود یک اعتبار ۱ دلاری دارد، و تعداد بشقاب‌های پشته همیشه مقداری نامنفی است، مطمئن خواهیم بود که مقدار اعتبار پشته همیشه مقداری نامنفی است. بنابراین برای هر دنباله‌ای از اعمال PUSH، POP، و MULTIPOP، کل هزینه‌ی سرشکن کران بالایی برای کل هزینه‌ی واقعی خواهد بود. از آن جایی که کل هزینه‌ی سرشکن $O(n)$ است، کل هزینه‌ی واقعی هم همین طور خواهد بود.

### افزایش یک شمارنده‌ی دودویی

به عنوان مثالی دیگر برای روشن شدن متد حسابداری، عملیات INCREMENT را بر روی یک شمارنده‌ی دودویی با مقدار اولیه‌ی صفر تحلیل می‌کنیم. همان طور که قبلاً مشاهده کردیم، زمان اجرای این عملیات متناسب است با تعداد بیت‌هایی که تغییر می‌کنند، که در این مثال از آن به عنوان هزینه استفاده می‌کنیم. دوباره اجازه دهید که از یک اسکناس ۱ دلاری برای نشان دادن واحد هزینه (در این مثال، تغییر یک بیت) استفاده استفاده کنیم.

برای تحلیل سرشکن، اجازه دهید که از هزینه‌ی سرشکن ۲ برای مقداردهی یک بیت با ۱ استفاده کنیم. وقتی یک بیت مقداردهی می‌شود، با ۱ دلار (از ۲ دلار هزینه شده) هزینه‌ی واقعی مقداردهی بیت را پرداخت می‌کنیم، و ۱ دلار دیگر را بر روی بیت قرار می‌دهیم تا بعداً برای بازگرداندن آن به ۰ استفاده کنیم. در هر لحظه‌ای از زمان، بر روی هر بیت ۱ از شمارنده یک اسکناس یک دلاری قرار دارد، و بنابراین احتیاجی به هزینه کردن برای بازگرداندن بیت به ۰ نداریم؛ برای این کار از اسکناس ۱ دلاری بر روی بیت استفاده می‌کنیم.

اکنون می‌توانیم هزینه‌ی سرشکن INCREMENT را تعیین کنیم. هزینه‌ی بازنشانی (reset) بیت‌ها در حلقه‌ی while به وسیله‌ی اسکناس‌های یک دلاری قرار داده شده بر روی آن‌ها پرداخت می‌شود. در خط ۶ رویه‌ی INCREMENT، حداکثر یک بیت با ۱ مقداردهی می‌شود، و بنابراین هزینه‌ی سرشکن عملیات INCREMENT حداکثر ۲ دلار است. تعداد ۱ها در شمارنده هیچ گاه منفی نیست، و بنابراین مقدار اعتبار همیشه نامنفی است. بنابراین برای $n$ عملیات INCREMENT، کل هزینه‌ی سرشکن $O(n)$ است، که کرانی است برای کل هزینه‌ی واقعی.

## تمرین‌ها

۱۷-۲-۱ بر روی یک پشته که اندازه‌ی آن هیچ گاه از $k$ فراتر نمی‌رود، دنباله‌ای از اعمال پشته را انجام می‌دهیم. بعد از هر $k$ عمل، یک کپی از کل پشته برای اهداف پشتیبانی انجام می‌گیرد. با نسبت دادن هزینه‌های سرشکن مناسب به اعمال مختلف پشته، نشان دهید که هزینه‌ی $n$ عملیات پشته، به همراه هزینه‌ی کپی $O(n)$ است.

۱۷-۲-۲ تمرین ۱۷-۱-۳ را با استفاده از متد حسابداری انجام دهید.

۱۷-۲-۳ فرض کنید که علاوه بر افزایش یک شمارنده، می‌خواهیم که بتوانیم آن را به صفر بازنشانی کنیم (تمام بیت‌های آن را ۰ کنیم). با در نظر گرفتن $\theta(1)$ به عنوان زمان بررسی یا تغییر یک بیت، نشان دهید که چطور می‌توان شمارنده را به صورت آرایه‌ای از بیت‌ها پیاده‌سازی کرد که هر دنباله‌ای از $n$ عملیات INCREMENT و RESET بر روی یک شمارنده با مقدار اولیه‌ی صفر، به زمان $O(n)$ نیاز داشته باشد. (*راهنمایی*: یک اشاره‌گر به با ارزش‌ترین بیت ۱ نگه دارید.)

## ۱۷-۳ متد پتانسیل

در ***متد پتانسیل*** (potential method) از تحلیل سرشکن، به جای نشان دادن کار از پیش پرداخت شده با اعتبار ذخیره شده در اشیای خاص در ساختمان داده، این کار به صورت «انرژی پتانسیل»، و یا فقط «پتانسیل» نشان داده می‌شود، که می‌توان بعداً این انرژی را آزاد کرد و از آن برای پرداخت برای اعمالی که بعداً انجام می‌شوند استفاده کرد. این انرژی پتانسیل به جای این که در اشیای خاصی در ساختمان داده ذخیره شود، به کل ساختمان داده نسبت داده می‌شود.

متد پتانسیل به صورت زیر عمل می‌کند. با یک ساختمان داده‌ی اولیه‌ی $D_0$ و $n$ عمل برای انجام بر روی این ساختمان داده شروع می‌کنیم. برای هر $i = 1,2,\ldots,n$، فرض می‌کنیم $c_i$ هزینه‌ی واقعی عملیات $i$ام باشد، و $D_i$ ساختمان داده‌ی حاصل از انجام عملیات $i$ام بر روی ساختمان داده‌ی $D_{i-1}$. یک *تابع پتانسیل* $\Phi$ هر ساختمان داده‌ی $D_i$ را به یک عدد حقیقی $\Phi(D_i)$ می‌نگارد، که *پتانسیل* نسبت داده شده به ساختمان داده‌ی $D_i$ است. *هزینه‌ی سرشکن* $\hat{c}_i$ مربوط به $i$امین عملیات نسبت به تابع پتانسیل $\Phi$ به صورت زیر تعریف می‌شود:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \tag{۲-۱۷}$$

بنابراین هزینه‌ی سرشکن هر عملیات برابر است با هزینه‌ی واقعی آن به علاوه‌ی افزایش پتانسیل ایجاد شده توسط آن عملیات. طبق تساوی (۱۷-۲) کل هزینه‌ی سرشکن $n$ عملیات برابر است با

$$\begin{aligned}\sum_{i=1}^{n}\hat{c}_i &= \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned} \tag{۳-۱۷}$$

تساوی دوم از تساوی (الف-۹) نتیجه می‌شود، چرا که $\Phi(D_i)$ از سری تلسکوپی پیروی می‌کند.

اگر بتوانیم یک تابع پتانسیل $\Phi$ تعریف کنیم به طوری که $\Phi(D_n) \geq \Phi(D_0)$، در این صورت کل هزینه‌ی سرشکن $\sum_{i=1}^{n}\hat{c}_i$ یک کران بالا برای کل هزینه‌ی واقعی $\sum_{i=1}^{n} c_i$ خواهد بود. در عمل

همیشه نمی‌دانیم که چند عمل ممکن است انجام شود. بنابراین اگر برای تمام $i$ها داشته باشیم $\Phi(D_i) \geq \Phi(D_0)$، در این صورت مانند متد حسابداری می‌توانیم تضمین کنیم که هیچ گاه بدهکار نمی‌شویم. معمولاً راحت است که $\Phi(D_0)$ را به صورت ۰ تعریف کنیم و سپس نشان دهیم که برای تمام $i$ها $\Phi(D_i) \geq 0$. (تمرین ۱۷-۳-۱ را برای یک راه سادهٔ برای اداره‌ی موقعیت‌هایی که در آن $\Phi(D_0) \neq 0$ ببینید.)

به صورت شهودی، اگر اختلاف پتانسیل $\Phi(D_i) - \Phi(D_{i-1})$ مربوط به $i$امین عملیات مثبت باشد، در این صورت هزینه‌ی سرشکن $\widehat{c_i}$ نشان دهنده‌ی یک هزینه‌ی اضافی برای عملیات $i$ام است، و پتانسیل ساختمان داده افزایش می‌یابد. اگر اختلاف پتانسیل منفی باشد، هزینه‌ی سرشکن نشان دهنده‌ی هزینه‌ای کم‌تر از هزینه‌ی عملیات $i$ام است، و هزینه‌ی واقعی آن عملیات توسط کاهش در انرژی پتانسیل پرداخته می‌شود.

هزینه‌های سرشکن تعریف شده توسط تساوی‌های (۱۷-۲) و (۱۷-۳) به انتخاب تابع پتانسیل $\Phi$ بستگی دارد. توابع پتانسیل مختلف ممکن است به هزینه‌های سرشکن مختلف منجر شوند، و همچنان کران بالایی برای هزینه‌ی واقعی باشند. معمولاً در انتخاب تابع پتانسیل باید چند چیز را سبک سنگین کرد؛ بهترین تابع پتانسیل برای استفاده به کران‌های زمانی مورد نظر بستگی دارد.

## عملیات پشته

برای روشن شدن متد پتانسیل، دوباره به مثال اعمال پشته‌ی PUSH، POP، و MULTIPOP بازمی‌گردیم. تابع پتانسیل $\Phi$ را بر روی یک پشته به صورت تعداد اشیاء درون پشته تعریف می‌کنیم. برای پشته‌ی خالی $D_0$ که با آن شروع می‌کنیم، داریم $\Phi(D_0) = 0$. از آن جایی که تعداد اشیاء درون پشته هیچ‌گاه منفی نیست، پتانسیل پشته‌ی $D_i$ که بعد از انجام عملیات $i$ام حاصل می‌شود هیچ‌گاه منفی نیست، و بنابراین

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

بنابراین کل هزینه‌ی سرشکن $n$ عملیات نسبت به $\Phi$ نشان‌دهنده‌ی یک کران بالا برای هزینه‌ی واقعی است.

اکنون اجازه دهید هزینه‌ی سرشکن اعمال مختلف پشته را محاسبه کنیم. اگر $i$امین عملیات بر روی پشته‌ای که حاوی $s$ عنصر است، عملیات PUSH باشد، در این صورت اختلاف پتانسیل برابر است با

$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$$

طبق تساوی (۱۷-۲)، هزینه‌ی سرشکن این عملیات 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 که $i$امین عملیات بر روی پشته MULTIPOP باشد، و همچنین $k' = \min(k, s)$ شیئ از روی پشته بازیابی شوند. هزینه‌ی واقعی این عملیات $k'$ است، و اختلاف پتانسیل برابر است با

$$\Phi(D_i)-\Phi(D_{i-1})=-k'$$

بنابراین، هزینه‌ی سرشکن عملیات MULTIPOP برابر است با

$$\widehat{c_i}=c_i+\Phi(D_i)-\Phi(D_{i-1})$$

متشابها هزینه‌ی سرشکن یک عملیات POP معمولی هم ۰ است.

هزینه‌ی سرشکن هر یک از سه عملیات $O(1)$ است، و بنابراین کل هزینه‌ی سرشکن دنباله‌ای از $n$ عملیات $O(n)$ خواهد بود. از آن جایی که قبلاً بحث کردیم که $\Phi(D_i)\geq\Phi(D_0)$، کل هزینه‌ی سرشکن $n$ عملیات کران بالایی برای کل هزینه‌ی واقعی است. بنابراین هزینه‌ی بدترین حالت $n$ عملیات $O(n)$ است.

## افزایش یک شمارنده‌ی دودویی

به عنوان مثالی دیگر از متد پتانسیل، دوباره به مثال افزایش یک شمارنده‌ی دودویی نگاهی می‌اندازیم. این بار پتانسیل یک شمارنده بعد از $i$امین عملیات INCREMENT را به صورت $b_i$ تعریف می‌کنیم، که تعداد ۱ها در شمارنده بعد از $i$امین عملیات است.

اجازه دهید هزینه‌ی سرشکن یک عملیات INCREMENT را محاسبه کنیم. فرض کنید که $i$امین عملیات INCREMENT، $t_i$ بیت را بازنشانی می‌کند. بنابراین هزینه‌ی واقعی عملیات حداکثر $t_i+1$ خواهد بود، چرا که علاوه بر بازنشانی $t_i$ بیت، حداکثر یک بیت با ۱ مقداردهی می‌شود. اگر $b_i=0$، عملیات $i$ام تمام $k$ بیت را بازنشانی می‌کند، و بنابراین $b_{i-1}=t_i=k$. اگر $b_i>0$، آن گاه $b_i=b_{i-1}-t_i+1$. در هر دو صورت $b_i\leq b_{i-1}-t_i+1$، و اختلاف پتانسیل برابر است با

$$\begin{aligned}\Phi(D_i)-\Phi(D_{i-1}) &\leq (b_{i-1}-t_i+1)-b_{i-1}\\ &=1-t_i\end{aligned}$$

بنابراین هزینه‌ی سرشکن برابر است با

$$\begin{aligned}\widehat{c_i} &= c_i+\Phi(D_i)-\Phi(D_{i-1})\\ &\leq (t_i+1)+(1-t_i)\\ &=2\end{aligned}$$

اگر شمارنده از صفر شروع شود، آن گاه $\Phi(D_0)=0$. از آن جایی که برای تمام $i$ها $\Phi(D_i)\geq 0$، کل هزینه‌ی سرشکن دنباله‌ای از $n$ عملیات INCREMENT کران بالایی برای کل هزینه‌ی واقعی است، و بنابراین بدترین حالت زمان اجرای $n$ عملیات INCREMENT برابر است با $O(n)$.

متد پتانسیل روش ساده‌ای برای تحلیل شمارنده به ما می‌دهد، حتی زمانی که شمارنده از صفر شروع نمی‌شود. در ابتدا $b_0$ ۱ در شمارنده وجود دارد، و بعد از $n$ عملیات INCREMENT، $b_n$ ۱ وجود دارد، که $b_0\leq 0$ و $b_n\leq k$. (به خاطر بیاورید که $k$ تعداد بیت‌ها در شمارنده است.) می‌توانیم تساوی (۱۷-۳) را به صورت 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 $1 \le i \le n$ داریم $\widehat{c_i} \le 2$. از آن جایی که $\Phi(D_0) = b_0$ و $\Phi(D_n) = b_n$، کل هزینه‌ی واقعی $n$ عملیات افزایش برابر است با

$$\begin{aligned}\sum_{i=1}^{n} c_i &\le \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0\end{aligned}$$

به خصوص توجه داشته باشید که از آن جایی که $b_0 \le k$، تا وقتی که $k = O(n)$، کل هزینه‌ی واقعی $O(n)$ است. به عبارت دیگر اگر حداقل $n = \Omega(n)$ عملیات INCREMENT انجام دهیم، کل هزینه‌ی واقعی $O(n)$ است، مستقل از این که مقدار اولیه‌ی شمارنده چقدر باشد.

## تمرین‌ها

۱۷-۳-۱ فرض کنید یک تابع پتانسیل $\Phi$ داریم به طوری که برای تمام $i$ها $\Phi(D_i) \ge \Phi(D_0)$، ولی $\Phi(D_0) \ne 0$. نشان دهید که یک تابع پتانسیل $\Phi'$ وجود دارد به طوری که $\Phi'(D_0) = 0$، برای تمام $i \ge 1$، $\Phi'(D_i) \ge 0$، و هزینه‌های سرشکن با استفاده از $\Phi'$ برابر است با هزینه‌های سرشکن با استفاده از $\Phi$.

۱۷-۳-۲ تمرین ۱۷-۱-۳ را با استفاده از متد پتانسیل انجام دهید.

۱۷-۳-۳ یک ساختمان داده‌ی هرم دودویی کمینه‌ی معمولی با $n$ عنصر را در نظر بگیرید که از اعمال INSERT و EXTRACT-MIN در بدترین حالت زمان $O(\lg n)$ پشتیبانی می‌کند. یک تابع پتانسیل $\Phi$ ارائه کنید به طوری که هزینه‌ی سرشکن INSERT، $O(\lg n)$ و هزینه‌ی سرشکن EXTRACT-MIN، $O(1)$ باشد، و سپس نشان دهید که روش شما کار می‌کند.

۱۷-۳-۴ کل هزینه‌ی اجرای $n$ عملیات پشته‌ی PUSH، POP، و MULTIPOP چقدر است، با فرض این که پشته با $s_0$ عنصر آغاز می‌شود و با $s_n$ عنصر پایان می‌یابد؟

۱۷-۳-۵ فرض کنید یک شمارنده به جای این که از صفر شروع شود، از یک عدد شروع می‌شود که در نمایش دودویی خود $b$ بیت ۱ دارد. نشان دهید که اگر $n = \Omega(b)$، هزینه‌ی اجرای $n$ عملیات INCREMENT برابر است با $O(n)$. (فرض نکنید که $b$ یک ثابت است.)

۱۷-۳-۶ نشان دهید که چطور می‌توان با استفاده از دو پشته‌ی معمولی، یک صف را پیاده‌سازی کرد (تمرین ۱۰-۱-۶ را ببینید) به طوری که هزینه‌ی سرشکن هر عملیات ENQUEUE و هر عملیات DEQUEUE، $O(1)$ باشد.

۱۷-۳-۷ یک ساختمان داده برای یک مجموعه‌ی چندگانه‌ی پویای $S$ از اعداد صحیح (که در آن مقادیر تکراری مجاز هستند) طراحی کنید که از اعمال زیر پشتیبانی کند:

- INSERT$(S, x)$ عدد $x$ را در $S$ درج می‌کند.

- DELETE-LARGER-HALF $\lceil |S|/2 \rceil$ عنصر بزرگ را از $S$ حذف می‌کند.

توضیح دهید که چطور می‌توان این ساختمان داده را طوری پیاده‌سازی کرد که هر دنباله‌ای از $m$ عملیات INSERT و DELETE-LAFGER-HALF در زمان $O(m)$ انجام شود. همچنین پیاده‌سازی شما باید حاوی روشی برای دادن عناصر $S$ به خروجی در $O(|S|)$ باشد.

## ۱۷-۴ جداول پویا

در بعضی کاربردها از پیش نمی‌دانیم قرار است چند عنصر در جدول ذخیره شوند. ممکن است مقداری حافظه به یک جدول اختصاص دهیم، ولی بعداً دریابیم که این مقدار کافی نیست. آن گاه باید حافظه‌ی بیش‌تری به جدول اختصاص یابد، و تمام اشیاء ذخیره شده در جدول اصلی به جدول جدید بزرگ‌تر کپی شوند. به طور مشابه اگر اشیاء زیادی از جدول حذف شوند، ممکن است مفید باشد که جدول کوچک‌تری برای آن‌ها اختصاص دهیم. در این بخش مسئله‌ی گسترش و کاهش یک جدول را به صورت پویا بررسی می‌کنیم. با استفاده از تحلیل سرشکن، نشان خواهیم داد که هزینه‌ی سرشکن هر درج و حذف فقط $O(1)$ است، هرچند که هزینه‌ی واقعی اعمال وقتی که یک عمل گسترش یا کاهش اتفاق می‌افتد، زیاد است. به علاوه خواهیم دید چطور می‌توان تضمین کرد که فضای بی‌استفاده‌ی درون یک جدول پویا هیچ گاه از ضریب ثابتی از کل فضا فراتر نرود.

فرض می‌کنیم که جدول پویا از اعمال TABLE-INSERT و TABLE-DELETE پشتیبانی می‌کند. TABLE-INSERT یک عنصر را درون یک مکان (slot) جدول درج می‌کند. به طور مشابه، TABLE-DELETE را می‌توان به صورت حذف یک عنصر از یک مکان جدول تلقی کرد، که یک مکان را در جدول آزاد می‌کند. جزئیات متد ساختمان داده‌ی استفاده شده برای سازمان دهی جدول مهم نیست؛ می‌توانیم از یک پشته (بخش ۱۰-۱)، یک هرم (فصل ۶)، و یا یک جدول درهم (فصل ۱۱) استفاده کنیم. همچنین می‌توانیم از یک آرایه و یا مجموعه‌ای از آرایه‌ها برای پیاده‌سازی حافظه‌ی اشیاء استفاده کنیم، همان طور که در بخش ۱۰-۳ این کار را کردیم.

خواهیم دید که استفاده از مفهوم استفاده شده در تحلیل درهم سازی (فصل ۱۱) مفید است. فاکتور بار $\alpha(T)$ یک جدول ناتهی $T$ را به صورت تعداد عناصر ذخیره شده در جدول تقسیم بر اندازه‌ی جدول (تعداد مکان‌های آن) تعریف می‌کنیم. به یک جدول خالی (جدولی که هیچ عنصری در آن ذخیره نشده است) اندازه‌ی ۰ را نسبت می‌دهیم، و فاکتور بار آن را برابر با ۱ تعریف می‌کنیم. اگر فاکتور بار یک جدول پویا از پایین با یک عدد ثابت کران دار شده باشد، فضای بی‌استفاده‌ی جدول هیچ گاه از ضریب ثابتی از کل فضای جدول فراتر نمی‌رود.

با تحلیل یک جدول پویا شروع می‌کنیم که در آن فقط عملیات درج انجام می‌شود. سپس، حالت کلی‌تری را در نظر می‌گیریم که در آن اجازه‌ی انجام هر دو عملیات درج و حذف را داریم.

### ۱۷-۴-۱ گسترش جدول

اجازه دهید فرض کنیم که حافظه‌ی جدول به صورت آرایه‌ای از مکان‌ها اختصاص داده شده است. یک جدول وقتی پر می‌شود که از تمام مکان‌های آن استفاده شود، یا به صورت معادل، وقتی که فاکتور بار آن ۱ باشد.[1] در بعضی از محیط‌های نرم‌افزاری، اگر تلاشی برای درج در یک جدول پر صورت گیرد راهی به جز لغو آن به همراه یک پیغام خطا وجود ندارد. با این حال ما فرض خواهیم کرد که محیط نرم‌افزاری ما، مانند بسیاری از محیط‌های نرم‌افزاری مدرن، سیستمی برای مدیریت حافظه فراهم می‌کند که در هنگام تقاضا می‌تواند بلوک‌های حافظه را تخصیص دهد و یا آزاد کند. بنابراین وقتی که یک عنصر در یک جدول پر درج می‌شود، می‌توانیم با تخصیص دادن یک جدول جدید با مکان‌های بیشتری از مکان‌های جدول فعلی، جدول خود را *گسترش* دهیم. از آن جایی که جدول همیشه باید در حافظه‌ای پیوسته قرار داشته باشد، باید یک آرایه‌ی جدید برای جدول بزرگ‌تر تخصیص دهیم و سپس عناصر را از جدول قدیمی به جدول جدید کپی کنیم.

یک کار مکاشفه‌ای (heuristic) معمول این است که یک جدول با اندازه‌ی دو برابر جدول قدیمی تخصیص دهیم. اگر فقط درج در جدول انجام شود، فاکتور بار جدول همیشه حداقل ۱/۲ است، و بنابراین مقدار حافظه‌ی بی‌استفاده هیچ وقت از نصف کل حافظه بیشتر نمی‌شود.

در شبه‌کد زیر، فرض می‌کنیم که $T$ یک شیئ نشان دهنده‌ی جدول است. فیلد $T.table$ حاوی یک اشاره‌گر است که به خانه‌ای از حافظه که نشان دهنده‌ی جدول است اشاره می‌کند. فیلد $T.num$ حاوی تعداد عناصر درون جدول، و فیلد $T.size$ تعداد کل مکان‌ها در جدول است. در ابتدا جدول خالی است: $T.num = T.size = 0$.

```
TABLE-INSERT (T, x)
1  if T.size = 0
2      allocate T.table with 1 slot
3      T.size = 1
4  if T.num = T.size
5      allocate new-table with 2 · T.size slots
6      insert all items in T.table into new-table
7      free T.table
8      T.table = new-table
9      T.size = 2 · T.size
10 insert x into T.table
11 T.num = T.num + 1
```

توجه کنید که در این جا دو رویه‌ی «درج» داریم: خود رویه‌ی TABLE-INSERT و *درج اولیه* درون یک جدول در خطوط ۶ و ۱۰. می‌توانیم زمان اجرای TABLE-INSERT را نسبت به تعداد

[1] در بعضی موقعیت‌ها، مانند جداول درهم‌سازی آدرس‌باز، ممکن است بخواهیم وقتی که فاکتور بار یک جدول به یک ثابت کوچک‌تر از ۱ می‌رسد، جدول را پر در نظر بگیریم. (تمرین ۱۷-۴-۱ را ببینید.)

درج‌های اولیه‌ی انجام شده تحلیل کنیم، بدین صورت که به هر یک از درج‌های اولیه‌ی انجام شده هزینه‌ی ۱ را نسبت می‌دهیم. فرض می‌کنیم که هزینه‌ی واقعی TABLE-INSERT نسبت به زمان درج عناصر تکی، خطی است، به طوری که سربار تخصیص یک جدول خالی در خط ۲ ثابت است، و سربار تخصیص و آزاد کردن حافظه در خطوط ۵ و ۷ در هزینه‌ی جابه‌جایی عناصر در خط ۶ محاط شده است. به رخداد اجرای عبارت then در خطوط ۵-۹ را یک گسترش می‌نامیم.

اجازه دهید دنباله‌ای از $n$ عملیات TABLE-INSERT را بر روی یک جدول خالی تحلیل کنیم. هزینه‌ی $c_i$ مربوط به $i$امین عملیات چقدر است؟ اگر در جدول فعلی مکانی خالی وجود داشته باشد (یا اگر عملیات اول در حال انجام باشد)، در این صورت $c_i = ۱$، چرا که فقط باید یک درج اولیه در خط ۱۰ انجام دهیم. با این حال اگر جدول فعلی پر باشد و یک گسترش رخ دهد، آن گاه $c_i = i$: هزینه برابر است با ۱ برای درج اولیه در خط ۱۰ به علاوه‌ی $i-۱$ برای عناصری که باید در خط ۶ از جدول قدیمی به جدول جدید کپی شوند. اگر $n$ عملیات انجام شود، بدترین حالت زمان اجرای یک عملیات $O(n)$ است، که به یک کران بالای $O(n^۲)$ بر روی کل زمان اجرا برای $n$ عملیات حاصل می‌شود.

این کران نزدیک نیست، چرا که هزینه‌ی گسترش جدول در $n$ عملیات TABLE-INSERT به تعداد زیادی تحمیل نمی‌شود. به خصوص، عملیات $i$ام فقط زمانی به یک گسترش ختم می‌شود که $i-۱$ توانی از ۲ باشد. در واقع هزینه‌ی سرشکن یک عملیات $O(۱)$ است، که این را می‌توانیم با استفاده از تحلیل متراکم نشان دهیم. هزینه‌ی $i$امین عملیات برابر است با

$$c_i = \begin{cases} i & \text{اگر } i-۱ \text{ توانی از ۲ باشد} \\ ۱ & \text{در غیر این صورت} \end{cases}$$

بنابراین کل هزینه‌ی $n$ عملیات TABLE-INSEERT برابر است با

$$\begin{aligned} \sum_{i=۱}^{n} c_i &\le n + \sum_{j=۰}^{\lfloor \lg n \rfloor} ۲^j \\ &< n + ۲n \\ &= ۳n \end{aligned}$$

زیرا حداکثر $n$ عملیات با هزینه‌ی ۱ وجود دارند و هزینه‌ی بقیه‌ی اعمال یک سری هندسی را تشکیل می‌دهند. از آن جایی که کران هزینه‌ی $n$ عملیات TABLE-INSERT، $۳n$ است، هزینه‌ی سرشکن یک عملیات حداکثر ۳ است.

با استفاده از متد حسابداری می‌توانیم احساس کنیم که چرا هزینه‌ی سرشکن عملیات TABLE-INSERT باید ۳ باشد. به طور شهودی، هر عنصر هزینه‌ی ۳ درج اولیه را پرداخت می‌کند: درج خود عنصر در جدول، جابه‌جایی عنصر وقتی که جدول گسترش می‌یابد، و جابه‌جایی عنصری دیگر که قبلاً با گسترش جدول جابه‌جا شده است. به عنوان مثال، فرض کنید که اندازه‌ی جدول درست پس از یک گسترش $m$ باشد. در این صورت تعداد عناصر درون جدول $m/۲$ است، و جدول حاوی هیچ اعتباری نیست. برای هر درج ۳ دلار هزینه می‌کنیم. درج اولیه‌ای که همان زمان انجام

می‌شود یک دلار هزینه خواهد داشت. یک دلار دیگر به عنوان اعتبار بر روی عنصر قرار خواهد گرفت. یک دلار دیگر بر روی یکی از $m/۲$ عنصری که در جدول وجود دارند قرار می‌گیرد. پر کردن جدول به $m-m/۲$ درج دیگر نیاز دارد، و بنابراین زمانی که جدول حاوی $m$ عنصر است و پر شده، هر عنصر یک دلار برای درج خود در زمان گسترش خواهد داشت.

همچنین می‌توان از متد پتانسیل برای تحلیل دنباله‌ای از $n$ عملیات TABLE-INSERT استفاده کرد، که در بخش ۱۷-۴-۲ از آن برای طراحی عملیات TABLE-DELETE استفاده می‌کنیم که هزینه‌ی سرشکن آن $O(۱)$ خواهد بود. با یک تابع پتانسیل $\Phi$ شروع می‌کنیم که دقیقاً بعد از گسترش جدول ۰ است، ولی زمانی که جدول پر می‌شود مقدار آن برابر با اندازه‌ی جدول خواهد بود، به طوری که می‌توان از پتانسیل برای پرداخت برای گسترش جدول استفاده کرد. تابع

$$\Phi(T) = ۲\cdot T.num - T.size \qquad (۵-۱۷)$$

یک تابع ممکن است. دقیقاً بعد از یک گسترش داریم $T.num = T.size/۲$، و بنابراین $\Phi(T) = ۰$، که همان طور است که ما می‌خواهیم. دقیقاً قبل از یک گسترش داریم $T.num = T.size$، و بنابراین $\Phi(T) = T.num$، که همان طور است که ما می‌خواهیم. مقدار اولیه‌ی پتانسیل ۰ است، و از آن جایی که جدول همیشه حداقل نیمه پر است، $T.num \geq T.size/۲$، که نتیجه می‌دهد که $\Phi(T)$ همیشه نامنفی است. بنابراین مجموع هزینه‌های سرشکن $n$ عملیات TABLE-INSERT کران بالایی برای مجموع هزینه‌های واقعی است.

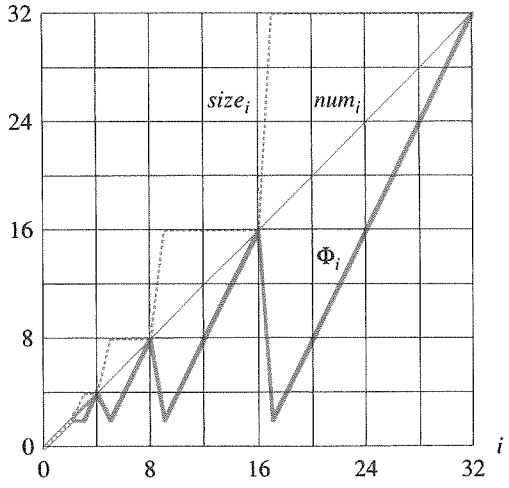
برای تحلیل هزینه‌ی سرشکن $i$امین عملیات TABLE-INSERT، فرض کنید $num_i$ نشان دهنده‌ی تعداد عناصر ذخیره شده در جدول بعد از عملیات $i$ام، $size_i$ نشان‌دهنده‌ی اندازه‌ی کل جدول بعد از عملیات $i$ام، و $\Phi_i$ نشان دهنده‌ی پتانسیل بعد از عملیات $i$ام باشند. در ابتدا داریم $num_۰ = ۰$، $size_۰ = ۰$، و $\Phi_۰ = ۰$.

اگر $i$امین عملیات TABLE-INSERT به یک گسترش منجر نشود، آن گاه داریم $size_i = size_{i-۱}$ و هزینه‌ی سرشکن عملیات برابر است با

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-۱}\\ &= ۱ + (۲num_i - size_i) - (۲num_{i-۱} - size_{i-۱})\\ &= ۱ + (۲num_i - size_i) - (۲(num_{i-۱}) - size_i)\\ &= ۳\end{aligned}$$

اگر $i$امین عملیات به یک گسترش منجر شود، آن گاه داریم $size_i = ۲size_{i-۱}$ و $size_{i-۱} = num_{i-۱}$، که نتیجه می‌دهد $size_i = ۲(num_{i-۱})$. بنابراین هزینه‌ی سرشکن عملیات برابر است با

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-۱}\\ &= num_i + (۲num_i - size_i) - (۲num_{i-۱} - size_{i-۱})\\ &= num_i + (۲num_i - ۲(num_i - ۱)) - (۲(num_i - ۱) - num_i - ۱)\\ &= num_i + ۲ - (num_i - ۱)\\ &= ۳\end{aligned}$$




**شکل ۱۷-۳** تأثیر دنباله‌ای از $n$ عملیات TABLE-INSERT بر روی $num_i$، تعداد عناصر درون جدول، $size_i$، تعداد مکان‌های خالی درون جدول، و $\Phi_i$، تابع پتانسیل، که مقدار هر کدام بعد از $i$امین عملیات اندازه‌گیری شده است. خط نازک نشان‌دهنده‌ی $num_i$، خط نقطه چین نشان‌دهنده‌ی $size_i$ و خط کلفت نشان‌دهنده‌ی $\Phi_i$ است. توجه کنید که دقیقاً قبل از یک گسترش، پتانسیل برابر است با تعداد عناصر درون جدول، و بنابراین می‌تواند هزینه‌ی جابه‌جایی تمام عناصر جدول را پرداخت کند. بعد از آن پتانسیل به ۰ افت می‌کند، ولی بعد از این که عنصری که باعث گسترش شد درج می‌شود، سریعا ۲ تا افزایش می‌یابد.

شکل ۱۷-۳ نمودار مقادیر $num_i$، $size_i$، و $\Phi_i$ را نسبت به $i$ نشان می‌دهد. توجه کنید که پتانسیل چگونه خود را برای پرداخت برای گسترش جدول آماده می‌کند.

### ۱۷-۴-۲ گسترش و کاهش جدول

برای پیاده‌سازی یک عملیات TABLEL-DELETE، بسیار ساده است که عنصر مورد نظر را از جدول حذف کنیم. با این حال، معمولاً خوب است که وقتی فاکتور جدول بار بسیار کوچک می‌شود، جدول را *کاهش* (contract) دهیم تا فضای بی‌استفاده بیش از اندازه نباشد. کاهش جدول مشابه گسترش جدول است: وقتی تعداد عناصر درون جدول بسیار کم می‌شود، یک جدول جدید کوچک‌تر در حافظه تخصیص می‌دهیم و سپس عناصر را از جدول قدیمی به جدول جدید منتقل می‌کنیم. سپس می‌توان با بازگرداندن جدول قدیمی به سیستم مدیریت حافظه، فضای آن را آزاد کرد. به صورت ایده‌آل، می‌خواهیم دو خصوصیت زیر را داشته باشیم:

- فاکتور بار جدول پویا از پایین با یک ثابت مثبت محدود شده باشد، و
- هزینه‌ی سرشکن یک عملیات جدول از بالا با یک ثابت محدود شده باشد.

فرض می‌کنیم که هزینه را می‌توان نسبت به درج‌ها و حذف‌های اولیه سنجید.

یک استراتژی طبیعی برای گسترش و کاهش جدول این است که هر گاه یک عنصر در یک جدول پر درج شد، اندازه‌ی جدول را دو برابر کنیم، و هر گاه یک حذف باعث شد که جدول کم‌تر از نیمه پر شود، اندازه‌ی آن را نصف کنیم. این استراتژی تضمین می‌کند که فاکتور بار جدول هیچ گاه از ۱/۲ کم‌تر نمی‌شود، ولی متأسفانه باعث می‌شود که هزینه‌ی سرشکن یک عملیات بزرگ شود. سناریوی زیر را در نظر بگیرید. $n$ عملیات بر روی یک جدول $T$ انجام می‌دهیم، که $n$ توانی از ۲ است. $n/2$ عملیات اول درج هستند، که طبق تحلیل قبلی هزینه‌ی کلی آن‌ها $\Theta(n)$ خواهد بود. در انتهای این دنباله از درج‌ها، $T.num = T.size = n/2$. برای $n/2$ عملیات بعدی، دنباله‌ی زیر را اجرا می‌کنیم:

درج، حذف، حذف، درج، درج، حذف، حذف، درج، درج، ...

اولین درج باعث می‌شود که جدول به اندازه‌ی $n$ گسترش یابد. سپس دو حذف بعد باعث می‌شوند که اندازه‌ی جدول به $n/2$ کاهش یابد. دو درج بعد یک گسترش دیگر را باعث می‌شوند، و همین طور تا آخر. هزینه‌ی هر گسترش و کاهش $\theta(n)$ است، و $\theta(n)$ تا از این عملیات وجود دارد. بنابراین، هزینه‌ی کلی $n$ عملیات $\theta(n^2)$ است، و هزینه‌ی سرشکن یک عملیات $\theta(n)$ خواهد بود.

مشکل این استراتژی مشخص است: بعد از یک گسترش، به اندازه‌ی کافی حذف انجام نمی‌دهیم که بتوانیم هزینه‌ی یک کاهش را پرداخت کنیم. متشابها، بعد از یک کاهش، به اندازه‌ی کافی درج انجام نمی‌دهیم که بتوانیم هزینه‌ی یک گسترش را پرداخت کنیم.

می‌توانیم این استراتژی را با کاهش کران پایین فاکتور بار جدول بهبود بخشیم. به خصوص، با درج یک عنصر در یک جدول پر، همچنان اندازه‌ی آن را دو برابر می‌کنیم، ولی وقتی اندازه‌ی جدول را نصف می‌کنیم که حذف یک عنصر باعث شود جدول کم‌تر از ۱/۴ پر شود، نه ۱/۲ پر. بنابراین فاکتور بار جدول از پایین با ۱/۴ محدود شده است.

به‌طور شهودی، فاکتور بار ۱/۲ را ایده‌آل در نظر می‌گیریم، و در این صورت پتانسیل جدول ۰ خواهد بود. وقتی فاکتور بار از ۱/۲ دور می‌شود، پتانسیل زیاد می‌شود تا زمانی که جدول را گسترش یا کاهش می‌دهیم، جدول به اندازه‌ی کافی پتانسیل در خود دارد که هزینه‌ی کپی تمام عناصر در جدول تازه ساخته شده را بدهد. بنابراین به یک تابع پتانسیل نیاز داریم که زمانی که فاکتور بار به ۱ یا ۱/۴ رسیده است، به اندازه‌ی $T.num$ رشد کرده باشد. پس از گسترش یا کاهش جدول، فاکتور بار به ۱/۲، و پتانسیل جدول به ۰ بازمی‌گردد.

از ارائه‌ی کد TABLE-DELETE صرف نظر می‌کنیم، چرا که مشابه TABLE-INSERT است. با این حال، برای اهداف تحلیلی مناسب است که فرض کنیم وقتی تعداد عناصر جدول به ۰ می‌رسد، حافظه‌ی جدول آزاد می‌شود. یعنی، اگر $T.num = 0$ آن گاه $T.size = 0$.

اکنون از متد پتانسیل استفاده می‌کنیم تا هزینه‌ی دنباله‌ای از $n$ عملیات TABLE-INSERT و TABLE-DELETE را تحلیل کنیم. با تعریف یک تابع پتانسیل $\Phi$ شروع می‌کنیم که دقیقاً بعد از یک گسترش یا کاهش ۰ است و با افزایش فاکتور بار به ۱ یا کاهش آن به ۱/۴ رشد می‌کند. اجازه دهید فاکتور بار یک جدول ناتهی $T$ را با $\alpha(T) = T.num/T.size$ تعریف کنیم. چون برای یک جدول تهی داریم $T.num = T.size = 0$ و $\alpha(T) = 1$، همیشه خواهیم داشت $T.num = \alpha(T) \cdot T.size$، چه جدول

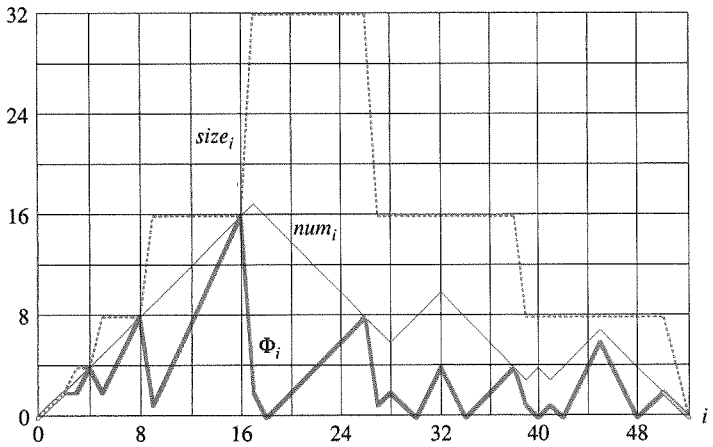
تهی باشد و چه نباشد. از تابع زیر به عنوان تابع پتانسیل استفاده می‌کنیم:

$$\Phi(T) = \begin{cases} 2 \cdot T.num - T.size & \text{اگر } \alpha(T) \geq 1/2 \\ T.size/2 - T.num & \text{اگر } \alpha(T) < 1/2 \end{cases} \qquad (۶-۱۷)$$

توجه کنید که پتانسیل یک جدول تهی ۰ است، و پتانسیل هیچ گاه منفی نمی‌شود. بنابراین کل هزینه‌ی سرشکن دنباله‌ای از عملیات نسبت به $\Phi$ کران بالایی برای هزینه‌ی واقعی دنباله است.

قبل از ادامه با یک تحلیل دقیق، لحظه‌ای تأمل می‌کنیم تا بعضی از خصوصیات تابع پتانسیل را مشاهده کنیم، که در شکل ۱۷-۴ نشان داده شده‌اند. توجه کنید که وقتی فاکتور بار ۱/۲ است، پتانسیل ۰ است. وقتی فاکتور بار ۱ است، داریم $T.size = T.num$، که نتیجه می‌دهد $\Phi(T) = T.num$، و بنابراین در صورت درج یک عنصر، پتانسیل می‌تواند هزینه‌ی گسترش را پرداخت کند. وقتی فاکتور بار ۱/۴ است، داریم $T.size = 4 \cdot T.num$، که نتیجه می‌دهد $\Phi(T) = T.num$، و بنابراین در صورت حذف یک عنصر پتانسیل می‌تواند هزینه‌ی آن را پرداخت کند.

برای تحلیل دنباله‌ای از $n$ عملیات TABLE-INSERT و TABLE-DELETE، فرض می‌کنیم $c_i$ نشان‌دهنده‌ی هزینه‌ی واقعی عملیات $i$ام، $\widehat{c_i}$ نشان‌دهنده‌ی هزینه‌ی سرشکن بر حسب $\Phi$، $num_i$ نشان‌دهنده‌ی تعداد عناصر ذخیره شده در جدول بعد از عملیات $i$ام، $size_i$ نشان‌دهنده‌ی اندازه‌ی کل جدول بعد از عملیات $i$ام، $\alpha_i$ نشان‌دهنده‌ی فاکتور بار جدول بعد از عملیات $i$ام، و $\Phi_i$ نشان‌دهنده‌ی پتانسیل بعد از عملیات $i$ام باشند. در ابتدا، $num_0 = 0$، $size_0 = 0$، $\alpha_0 = 1$، و $\Phi_0 = 0$.




**شکل ۱۷-۴** تأثیر دنباله‌ای از $n$ عملیات TABLE-INSERT و TABLE-DELETE بر روی $num_i$، تعداد عناصر درون جدول، $size_i$، تعداد مکان‌های جدول، و $\Phi$، مقدار پتانسیل، که هر کدام بعد از $i$امین عملیات اندازه‌گیری شده‌اند. توجه کنید که دقیقاً قبل از یک گسترش، پتانسیل به اندازه‌ی تعداد عناصر درون جدول رشد کرده است، و بنابراین می‌تواند هزینه‌ی جابه‌جایی تمام عناصر درون جدول را پرداخت کند. به طور مشابه، دقیقاً قبل از یک کاهش پتانسیل به اندازه‌ی تعداد عناصر درون جدول رشد کرده است.

با حالتی شروع می‌کنیم که عملیات $i$ام TABLE-INSERT است. تحلیل مشابه حالت گسترش جدول در بخش ۱۷-۴-۱ است وقتی $\alpha_{i-1} \geq 1/2$. چه جدول گسترش یابد و چه نیابد، $\widehat{c_i}$، هزینه‌ی سرشکن عملیات، حداکثر ۳ است. اگر $\alpha_{i-1} < 1/2$، جدول نمی‌تواند در نتیجه‌ی عملیات گسترش یابد، چرا که گسترش فقط زمانی رخ می‌دهد که $\alpha_{i-1} = 1$. همچنین اگر $\alpha_i < 1/2$، هزینه‌ی سرشکن $i$امین عملیات 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 $\alpha_{i-1} < 1/2$ ولی $\alpha_i \geq 1/2$، آن گاه

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (2(num_{i-1} + 1) - size_i) - (size_{i-1}/2 - num_i) \\ &= 3num_{i-1} - 3/2 size_{i-1} + 3 \\ &= 3\alpha_{i-1} size_{i-1} - 3/2 size_{i-1} + 3 \\ &< 3/2 size_{i-1} - 3/2 size_{i-1} + 3 \\ &= 3\end{aligned}$$

بنابراین هزینه‌ی سرشکن یک عملیات TABLE-INSERT حداکثر ۳ است.

اکنون به سراغ حالتی می‌رویم که در آن $i$امین عملیات TABLE-DELETE است. در این حالت $num_i = num_{i-1} - 1$. اگر $\alpha_{i-1} < 1/2$، آن گاه باید در نظر بگیریم که آیا یک کاهش رخ می‌دهد یا خیر. اگر کاهش رخ ندهد، آن گاه $size_i = size_{i-1}$، و هزینه‌ی سرشکن عملیات 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 عملیات $i$ام باعث یک کاهش شود، در این صورت هزینه‌ی واقعی عملیات $c_i = num_i + 1$ است، چرا که باید یک عنصر را حذف و $num_i$ عنصر را جابه‌جا کنیم. داریم $size_i/2 = size_{i-1}/4 = num_{i-1} + 1$، و هزینه‌ی سرشکن عملیات برابر است با

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= (num_i + 1) + (size_i/2 - num_i)(size_{i-1}/2 - num_{i-1}) \\ &= (num_i + 1) + ((num_i + 1) - num_i) - ((2num_i + 2) - (num_i + 1)) \\ &= 1\end{aligned}$$

وقتی عملیات $i$ام TABLE-DELETE است و $\alpha_{i-1} \geq 1/2$، هزینه‌ی سرشکن هم با یک ثابت محدود

شده است. تحلیل به عنوان تمرین ۱۷-۴-۲ به خواننده واگذار شده است.

در مجموع، از آن جایی که هزینه‌ی سرشکن هر عملیات از بالا با یک ثابت محدود شده است، هزینه‌ی واقعی دنباله‌ای از $n$ عملیات بر روی یک جدول پویا $O(n)$ است.

## تمرین‌ها

۱-۴-۱۷ فرض کنید که می‌خواهیم یک جدول درهم آدرس‌باز پویا پیاده‌سازی کنیم. چرا باید وقتی فاکتور بار جدول به عددی مانند $\alpha$ می‌رسد که مقدار آن منحصرا کوچک‌تر از ۱ است، جدول را پر در نظر بگیریم؟ مختصرا شرح دهید که چگونه می‌توانیم در یک جدول درهم آدرس‌باز پویا، درج انجام دهیم به طوری که امیدریاضی هزینه‌ی سرشکن برای هر درج $O(1)$ باشد. چرا امیدریاضی هزینه‌ی واقعی برای هر درج لزوماً $O(1)$ نیست؟

۲-۴-۱۷ نشان دهید که اگر $\alpha_{i-1} \geq 1/2$ و $i$ امین عملیات بر روی یک جدول پویا TABLE-DELETE باشد، آن گاه هزینه‌ی سرشکن عملیات بر حسب تابع پتانسیل (۱۷-۶) از بالا یک کران ثابت دارد.

۳-۴-۱۷ فرض کنید که به جای این که وقتی فاکتور بار جدول از ۱/۴ کم‌تر می‌شود اندازه‌ی جدول را نصف کنیم، وقتی اندازه‌ی فاکتور بار به زیر ۱/۳ می‌رسد، اندازه‌ی جدول را در ۲/۳ ضرب کنیم. با استفاده از تابع پتانسیل

$$\Phi(T) = |2 \cdot T.num - T.size|$$

نشان دهید که هزینه‌ی سرشکن یک TABLE-DELETE که از این استراتژی استفاده می‌کند از بالا یک کران ثابت دارد.

# مسائل

۱-۱۷ ***شمارنده‌ی دودویی با بیت‌های معکوس***

در فصل ۳۰ یک الگوریتم مهم به نام تبدیل سریع فوریه (Fast Fourier Transform)، یا FFT بررسی می‌شود. در مرحله‌ی اول الگوریتم FFT یک *جایگشت بیتی معکوس* (bit-reversal permutation) بر روی یک آرایه‌ی ورودی $A[0..n-1]$ که طول آن به ازای یک عدد صحیح نامنفی $k$، $n = 2^k$ است، انجام می‌شود. این جایگشت، عناصری را که نمایش دودویی اندیس آن‌ها معکوس یکدیگر است را با هم جابه‌جا می‌کند.

می‌توانیم هر اندیس $a$ را با یک دنباله‌ی $k$ بیتی $\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle$ نمایش دهیم، که $a = \sum_{i=0}^{k-1} a_i 2^i$. تعریف می‌کنیم

$$rev_k(\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle) = \langle a_0, a_1, \ldots, a_{k-1} \rangle$$

بنابراین،

$$rev_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i$$

مثلاً اگر $n = 16$ (یا به طور معادل، $k = 4$)، آن گاه $rev_k(3) = 12$، چرا که نمایش ۴ بیتی ۳، ۰۰۱۱ است، که معکوس آن ۱۱۰۰ است، نمایش ۴ بیتی عدد ۱۲.

I. یک تابع $rev_k$ بدهید که در زمان $\Theta(k)$ اجرا می‌شود، و یک الگوریتم برای انجام جایگشت بیتی مکعوس بر روی یک آرایه با طول $n = 2^k$ در زمان $O(nk)$ بنویسید.

می‌توانیم از یک الگوریتم بر مبنای تحلیل سرشکن برای بهبود زمان اجرای جایگشت بیتی معکوس استفاده کنیم. یک «شمارنده‌ی بیتی معکوس» ایجاد می‌کنیم، و همچنین یک رویه‌ی BIT-REVERSED-INCREMENT، که وقتی مقدار یک شمارنده‌ی بیتی معکوس $a$ به آن داده شد، $rev_k(rev_k(a)+1)$ را تولید می‌کند. مثلاً اگر $k = 4$، و شمارنده‌ی بیتی معکوس از ۰ شروع شود، آن گاه فراخوانی‌های پی در پی BIT-REVERSED-INCREMENT دنباله‌ی زیر را تولید می‌کند:

۰۰۰۰,۱۰۰۰,۰۱۰۰,۱۱۰۰,۰۰۱۰,۱۰۱۰,... = ۰,۸,۴,۱۲,۲,۱۰,...

II. فرض کنید که کلمه‌ها در کامپیوتر شما مقادیر $k$ بیتی هستند، و کامپیوتر شما می‌تواند در واحد زمان اعمالی را بر روی مقادیر دودویی انجام دهد، مانند جابه‌جایی به چپ یا راست به مقدار دلخواه، «و»ی بیتی، «یا»ی بیتی، و غیره. یک پیاده‌سازی برای رویه‌ی BIT-REVERSED-INCREMENT ارائه کنید که اجازه می‌دهد که جایگشت بیتی معکوس بر روی آرایه‌ای با $n$ عنصر در زمان $O(n)$ اجرا شود.

III. فرض کنید که جابه‌جایی کلمات را می‌توانید فقط به اندازه‌ی یک بیت در هر واحد زمان انجام دهید. آیا همچنان می‌توان جایگشت بیتی معکوس را در زمان $O(n)$ انجام داد؟

## ۲-۱۷ جستجوی دودویی پویا

جستجوی دودویی بر روی یک آرایه‌ی مرتب‌شده در زمان لگاریتمی انجام می‌شود، ولی زمان درج یک عنصر جدید نسبت به اندازه‌ی آرایه خطی است. با نگه داشتن آرایه‌های مرتب‌شده‌ی متعدد، می‌توان زمان درج را بهبود بخشید.

به خصوص، فرض کنید که می‌خواهیم بر روی مجموعه‌ای از $n$ عنصر از اعمال SEARCH و INSERT پشتیبانی کنیم. فرض کنید $k = \lceil \lg n(n+1) \rceil$، و نمایش دودویی $n$ به صورت $\langle n_{k-1}, n_{k-2}, ..., n_0 \rangle$ باشد. $k$ آرایه‌ی مرتب‌شده‌ی $A_0, A_1, ..., A_{k-1}$ خواهیم داشت، که در آن برای $i = 0, 1, ..., k-1$، طول آرایه‌ی $A_i$، $2^i$ خواهد بود. هر آرایه یا پر است و یا خالی، بسته به این که به ترتیب $n_i = 1$ یا $n_i = 0$. بنابراین تعداد کل عناصری که در $k$ آرایه نگه داشته می‌شود عبارت است از $n = \sum_{i=0}^{k-1} n_i 2^i$. با این که هر آرایه مرتب‌شده است، ولی هیچ رابطه‌ای میان عناصر در آرایه‌های مختلف وجود ندارد.

I. توضیح دهید که چگونه می‌توان عملیات SEARCH را بر روی این ساختار انجام داد. بدترین حالت زمان اجرای آن را تحلیل کنید.

II. توضیح دهید که چگونه می‌توان یک عنصر جدید در این ساختار درج کرد. بدترین حالت زمان اجرا و زمان اجرای سرشکن آن را تحلیل کنید.

III. توضیح دهید که چگونه می‌توان DELETE را برای این ساختار پیاده‌سازی کرد.

### ۱۷-۳ *درختان متوازن وزنی سرشکن (Amortized weight-balanced trees)*

یک درخت جستجوی دودویی معمولی را در نظر بگیرید که با اضافه کردن فیلد $size[x]$ به گره‌ی $x$ گسترش داده شده است، که این فیلد نشان دهنده‌ی تعداد عناصر ذخیره شده در زیردرخت با ریشه‌ی $x$ است. فرض کنید $\alpha$ یک ثابت در بازه‌ی $1/2 < \alpha < 1$ باشد. می‌گوییم گره‌ی $x$، $\alpha$-متوازن است اگر داشته باشیم $x.left.size \le \alpha \cdot x.size$ و $x.right.size \le \alpha \cdot x.size$. کل درخت $\alpha$-متوازن است اگر تمام عناصر درخت $\alpha$-متوازن باشند. رویکرد سرشکن زیر توسط G. Varghese برای نگه‌داری درختان متوازن وزنی پیشنهاد شده است.

I. یک درخت $1/2$-متوازن، در متوازن‌ترین حالت ممکن قرار دارد. برای یک گره‌ی $x$ در یک درخت جستجوی دودویی دلخواه، نشان دهید که چگونه می‌توان طوری زیردرخت $x$ را بازسازی کرد که $x$، $1/2$-متوازن شود. الگوریتم شما باید در زمان $\theta(x.size)$ اجرا شود، و می‌تواند از $O(x.size)$ حافظه‌ی کمکی استفاده کند.

II. نشان دهید که انجام یک جستجو در یک درخت $\alpha$-متوازن با $n$ گره در بدترین حالت به زمان $O(\lg n)$ نیاز دارد.

برای ادامه‌ی این مسئله، فرض کنید که $\alpha$ اکیدا بزرگ‌تر از $1/2$ است. فرض کنید که INSERT و DELETE به شکل معمول برای یک درخت جستجوی دودویی پیاده‌سازی شده‌اند، غیر از این که بعد از هر یک از این اعمال، اگر در درخت گره‌ای وجود داشته باشد که $\alpha$-متوازن نباشد، آن گاه ساختار بالاترین گره‌ای که چنین خصوصیتی دارد «بازسازی» می‌شود تا $\alpha$-متوازن شود.

روش بازسازی را با استفاده از متد پتانسیل تحلیل خواهیم کرد. برای یک گره‌ی $x$ در درخت جستجوی دودویی، تعریف می‌کنیم

$$\Delta(x) = |x.left.size - x.right.size|$$

و پتانسیل $T$ را به صورت

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \ge 2} \Delta(x)$$

تعریف می‌کنیم، که در آن $c$ یک ثابت به اندازه‌ی کافی بزرگ است که به $\alpha$ بستگی دارد.

III. بحث کنید که هر درخت جستجوی دودویی دلخواهی دارای پتانسیل نامنفی، و پتانسیل یک درخت ۱/۲-متوازن ۰ است.

IV. فرض کنید که $m$ واحد پتانسیل می‌تواند هزینه‌ی بازسازی یک زیردرخت با $m$ گره را پرداخت کند. $c$ بر حسب $\alpha$ چقدر باید بزرگ باشد که بازسازی یک زیردرخت که $\alpha$-متوازن نیست در زمان $O(1)$ انجام شود؟

V. نشان دهید که هزینه‌ی سرشکن درج یک گره در یک درخت $\alpha$-متوازن با $n$ گره و حذف یک گره از آن $O(\lg n)$ است.

## ۱۷-۴ هزینه‌ی ساختاردهی دوباره‌ی درختان قرمز-سیاه

چهار عملیات اصلی بر روی درختان قرمز-سیاه وجود دارد که بر روی درخت *اصلاحات ساختاری* انجام می‌دهند: درج گره، حذف گره، دوران‌ها، و تغییر رنگ. قبلاً دیدیم که RB-INSERT و RB-DELETE فقط $O(1)$ دوران، درج گره، و حذف گره برای حفظ خصوصیات درختان قرمز-سیاه انجام می‌دهند، ولی تعداد تغییر رنگ‌های انجام شده در آن‌ها بسیار بیشتر است.

I. یک درخت قرمز-سیاه با $n$ گره توصیف کنید که فراخوانی RB-INSERT برای درج گره‌ی $(n+1)$ام به $\Omega(\lg n)$ تغییر رنگ نیاز داشته باشد. سپس یک درخت قرمز-سیاه دیگر با $n$ گره توصیف کنید که در آن فراخوانی RB-DELETE بر روی یک گره‌ی خاص به $\Omega(\lg n)$ تغییر رنگ نیاز داشته باشد.

با این که بدترین حالت تعداد تغییر رنگ‌ها برای هر عملیات می‌تواند لگاریتمی باشد، اثبات خواهیم کرد که هر دنباله‌ای از $m$ عملیات RB-INSERT و RB-DELETE بر روی یک درخت قرمز-سیاه تهی در بدترین حالت به $O(m)$ اصلاح ساختاری منجر می‌شود. توجه کنید که هر تغییر رنگ را به صورت یک اصلاح ساختاری در نظر می‌گیریم.

II. بعضی از حالت‌های اداره شده توسط حلقه‌ی اصلی کد در هر دوی RB-INSERT-FIXUP و RB-DELETE-FIXUP *پایان دهنده* هستند: وقتی این حالت‌ها اتفاق بیفتند، باعث می‌شوند حلقه پس از انجام تعداد ثابتی اعمال اضافی پایان یابد. برای هر یک از حالت‌های RB-INSERT-FIXUP و RB-DELETE-FIXUP، تعیین کنید که آیا آن حالت پایان دهنده است و یا خیر. (*راهنمایی*: به شکل‌های ۱۳-۵، ۱۳-۶ و ۱۳-۷ نگاه کنید.)

ابتدا اصلاحات ساختاری را بررسی می‌کنیم که در آن‌ها فقط درج انجام می‌شود. فرض کنید که $T$ یک درخت قرمز-سیاه باشد، و $\Phi(T)$ را به صورت تعداد گره‌های درون $T$ تعریف کنید. فرض کنید که با ۱ واحد پتانسیل می‌توان هزینه‌ی هر اصلاح ساختاری را که توسط هر یک از سه حالت RB-INSERT-FIXUP انجام می‌شود، پرداخت کرد.

III. فرض کنید $T'$ حاصل اعمال حالت ۱ RB-INSERT-FIXUP بر روی $T$ باشد. بحث کنید که $\Phi(T') = \Phi(T) - 1$.

IV. درج گره با استفاده از RB-INSERT در یک درخت قرمز سیاه را می‌توان به سه بخش تقسیم کرد. اصلاحات ساختاری و تغییرات پتانسیل در نتیجه‌ی خطوط ۱-۱۶ RB-INSERT، در نتیجه‌ی حالت‌های غیر پایان دهنده‌ی RB-INSERT-FIXUP، و در نتیجه‌ی حالت‌های پایان دهنده‌ی RB-INSERT-FIXUP را لیست کنید.

V. با استفاده از قسمت IV، بحث کنید که تعداد اصلاحات ساختاری انجام شده توسط هر فراخوانی RB-INSERT به صورت سرشکن $O(1)$ است.

اکنون می‌خواهیم اثبات کنیم که در حالتی که هم درج داریم و هم حذف، تعداد اصلاحات ساختاری $O(m)$ است. اجازه دهید برای هر گره‌ی $x$ تعریف کنیم

$$w(x)=\begin{cases}0 & \text{اگر } x \text{ قرمز باشد}\\ 1 & \text{اگر } x \text{ سیاه باشد و هیچ فرزند قرمزی نداشته باشد}\\ 0 & \text{اگر } x \text{ سیاه باشد و یک فرزند قرمز داشته باشد}\\ 2 & \text{اگر } x \text{ سیاه باشد و دو فرزند قرمز داشته باشد}\end{cases}$$

اکنون پتانسیل درخت قرمز-سیاه $T$ را دوباره به صورت زیر تعریف می‌کنیم:

$$\Phi(T)=\sum_{x\in T} w(x)$$

و همچنین فرض می‌کنیم که $T'$ درخت حاصل از اعمال هر یک از حالت‌های غیر پایان دهنده از RB-INSERT-FIXUP یا RB-DELETE-FIXUP بر روی $T$ باشد.

VI. نشان دهید که برای تمام حالت‌های غیر پایان‌دهنده از RB-INSERT-FIXUP، $\Phi(T')\le\Phi(T)-1$. بحث کنید که تعداد اصلاحات ساختاری انجام شده توسط هر فراخوانی RB-INSERT-FIXUP به صورت سرشکن $O(1)$ است.

VII. نشان دهید که برای تمام حالت‌های غیر پایان دهنده از RB-DELETE-FIXUP، $\Phi(T')\le\Phi(T)-1$. بحث کنید که تعداد اصلاحات ساختاری انجام شده توسط هر فراخوانی RB-DELETE-FIXUP به صورت سرشکن $O(1)$ است.

VIII. اثبات این قضیه را کامل کنید که در بدترین حالت، هر دنباله‌ای از $m$ عملیات RB-INSERT-FIXUP و RB-DELETE-FIXUP، $O(m)$ اصلاح ساختاری انجام می‌دهد.

### ۱۷-۵ تحلیل رقابتی لیست‌های خود-متغیر به همراه جابه‌جایی به جلو

یک لیست خود-متغیر (self-organizing list)، یک لیست پیوندی از $n$ عنصر است، که در آن هر عنصر یک کلید یکتا دارد. وقتی برای یک عنصر در لیست جستجو می‌کنیم، به ما یک کلید داده شده است، و می‌خواهیم عنصری با آن کلید بیابیم.

یک لیست خود- متغیر دو خصوصیت مهم دارد:

۱. برای یافتن یک عنصر در لیست، با داشتن کلید آن، باید از ابتدا لیست را جستجو کرده تا به عنصر مورد نظر برسیم. اگر عنصر مورد نظر، عنصر $k$ام از ابتدا باشد، آن گاه هزینه‌ی یافتن کلید $k$ است.

۴. می‌توانیم پس از هر عملیات، عناصر لیست را بازآرایی کنیم، بر حسب یک قانون داده شده با یک هزینه‌ی داده شده. می‌توانیم هر روش مکاشفه‌ای دلخواه را برای بازآرایی لیست انتخاب کنیم.

فرض کنید که با یک لیست داده شده با $n$ عنصر آغاز می‌کنیم، و به ما یک دنباله‌ی دسترسی $\sigma = \langle \sigma_1, \sigma_2, ..., \sigma_m \rangle$ از کلیدها برای جستجو کردن، به ترتیب، داده شده است. هزینه‌ی کل دنباله برابر است با مجموع هزینه‌ی دسترسی‌های درون لیست.

از میان روش‌های مختلف بازآرایی لیست پس از هر عملیات، این مسئله بر روی جابه‌جایی عناصر مجاور در لیست تمرکز می‌کند، با هزینه‌ی واحد برای هر جابه‌جایی. به کمک یک تابع پتانسیل، نشان خواهید داد که یک روش مکاشفه‌ای خاص برای بازآرایی لیست (جابه‌جایی به جلو) هزینه‌ی خواهد داشت که از ۴ برابر هزینه‌ی هر روش مکاشفه‌ای دیگر برای تغییر ترتیب لیست، بیشتر نخواهد بود، حتی اگر روش دیگر، دنباله‌ی دسترسی را از قبل بداند! این روش تحلیل را یک *تحلیل رقابتی* (competitive analysis) می‌نامیم.

برای یک روش مکاشفه‌ای H و یک ترتیب اولیه‌ی داده شده برای لیست، هزینه‌ی دسترسی دنباله‌ی $\sigma$ را با $C_H(\sigma)$ نشان می‌دهیم. فرض کنید $m$ نشان‌دهنده‌ی تعداد دسترسی‌ها در $\sigma$ باشد.

I. بحث کنید که اگر مکاشفه‌ی H دنباله‌ی دسترسی را از قبل نداند، آن گاه هزینه‌ی بدترین حالت برای H بر روی یک دنباله‌ی دسترسی $\sigma$ برابر است با $C_H(\sigma) = \Omega(mn)$.

با مکاشفه‌ی *جابه‌جایی به جلو* (move-to-front)، درست پس از جستجو برای یک عنصر $x$، آن عنصر را به مکان اول در لیست (ابتدای لیست) جابه‌جا می‌کنیم.

فرض کنید $rank_L(x)$ نشان دهنده‌ی رتبه‌ی عنصر $x$ در لیست $L$ باشد، یعنی مکان $x$ در $L$. برای مثال، اگر $x$ چهارمین عنصر در $L$ باشد، آن گاه $rank_L(x) = ۴$. فرض کنید $c_i$ نشان دهنده‌ی هزینه‌ی درسترسی $\sigma_i$ با استفاده از روش مکاشفه‌ای جابه‌جایی به جلو باشد، که شامل هزینه‌ی یافتن عنصر در لیست و هزینه‌ی جابه‌جایی آن به ابتدای لیست با استفاده از یک سری جابه‌جایی با عناصر مجاور در لیست است.

II. نشان دهید که اگر $\sigma_i$، به عنصر $x$ در لیست $L$ با استفاده از مکاشفه‌ی جابه‌جایی به جلو دسترسی داشته باشد، آن گاه $c_i = ۲ \cdot rank_L(x) - ۱$.

اکنون جابه‌جایی به جلو را با هر روش مکاشفه‌ای دیگر H مقایسه می‌کنیم که یک دنباله‌ی دسترسی را طبق دو خصوصیت داده شده‌ی بالا پردازش می‌کند. مکاشفه‌ی H می‌تواند عناصر لیست را به هر روشی که می‌خواهد جابه‌جا کند، و حتی ممکن است کل دنباله‌ی دسترسی را از قبل بداند.

فرض کنید $L_i$ نشان‌دهنده‌ی لیست پس از دسترسی $\sigma_i$ با استفاده از جابه‌جایی به جلو باشد، و $L_i^*$ نشان‌دهنده‌ی لیست پس از دسترسی $\sigma_i$ با استفاده از مکاشفه‌ی H. هزینه‌ی دسترسی $\sigma_i$ را با $c_i$ برای جابه‌جایی به جلو و با $c_i^*$ برای مکاشفه‌ی H نشان می‌دهیم. فرض کنید که

مکاشفه‌ی H حین دسترسی $\sigma_i$، $t_i^*$ جابه‌جایی انجام می‌دهد.

III. در بخش II، نشان دادید که $c_i = ۲\cdot rank_{L_{i-۱}}(x) - ۱$. اکنون نشان دهید که $c_i^* = rank_{L_{i-۱}^*}(x) + t_i^*$.

یک *وارونگی* (inversion) در لیست $L$ را به صورت جفتی از عناصر $y$ و $z$ تعریف می‌کنیم به طوری که $y$ در لیست $L_i$ قبل از $z$ باشد، و در لیست $L_i^*$ بعد از $z$. فرض کنید که لیست $L_i$، پس از پردازش لیست دسترسی $\langle \sigma_۱, \sigma_۲, ..., \sigma_i \rangle$، $q_i$ وارونگی داشته باشد. در این صورت یک تابع پتانسیل $\Phi$ تعریف می‌کنیم که $L_i$ را به عدد حقیقی $\Phi(L_i) = ۲q_i$ نگاشت می‌کند. برای مثال، اگر $L_i$ حاوی عناصر $\langle e,c,a,d,b \rangle$ باشد، و $L_i^*$ حاوی عناصر $\langle c,a,b,d,e \rangle$، آن گاه $L_i$، ۵ وارونگی دارد، که عبارتند از $((e,c),(e,a),(e,d),(e,b),(d,b))$، و بنابراین $\Phi(L_i) = ۱۰$. مشاهده کنید که $\Phi(L_i) \geq ۰$ برای هر $i$، و این که اگر جابه‌جایی به جلو و مکاشفه‌ی H هر دو با یک لیست $L_۰$ آغاز کنند، آن گاه $\Phi(L_۰) = ۰$.

IV. بحث کنید که یک جابه‌جایی پتانسیل را یا ۲ واحد افزایش و یا ۲ واحد کاهش می‌دهد.

فرض کنید که دسترسی $\sigma_i$ عنصر $x$ را می‌یابد. برای درک این که پتانسیل چگونه توسط $\sigma_i$ تغییر می‌کند، اجازه دهید عناصر غیر $x$ را به چهار دسته تقسیم کنیم، بسته به این که درست قبل از $i$امین دسترسی در کجای لیست قرار دارند:

- مجموعه‌ی $A$ حاوی عناصری که در هر دوی $L_{i-۱}$ و $L_{i-۱}^*$ قبل از $x$ قرار دارند.
- مجموعه‌ی $B$ حاوی عناصری که در $L_{i-۱}$ قبل از $x$ و در $L_{i-۱}^*$ بعد از $x$ قرار دارند.
- مجموعه‌ی $c$ حاوی عناصری که در $L_{i-۱}$ بعد از $x$ و در $L_{i-۱}^*$ قبل از $x$ قرار دارند.
- مجموعه‌ی $D$ حاوی عناصری که در هر دوی $L_{i-۱}$ و $L_{i-۱}^*$ بعد از $x$ قرار دارند.

V. بحث کنید که $rank_{L_{i-۱}}(x) = |A| + |B| + ۱$ و $rank_{L_{i-۱}^*}(x) = |A| + |C| + ۱$.

VI. نشان دهید که دسترسی $\sigma_i$ یک تغییر

$$\Phi(L_i) - \Phi(L_{i-۱}) \leq ۲(|A| - |B| + t_i^*)$$

در پتانسیل ایجاد می‌کند، که در آن، مانند قبل، مکاشفه‌ی H حین دسترسی $\sigma_i$، $t_i^*$ جابه‌جایی انجام می‌دهد.

هزینه‌ی سرشکن $\widehat{c_i}$ مربوط به دسترسی $\sigma_i$ را به صورت $\widehat{c_i} = c_i + \Phi(L_i) - \Phi(L_{i-۱})$ تعریف می‌کنیم.

VII. نشان دهید که هزینه‌ی سرشکن $\widehat{c_i}$ مربوط به دسترسی $\sigma_i$ دارای کران بالای $۴c_i^*$ است.

VIII. نتیجه بگیرید که هزینه‌ی $C_{MTF}(\sigma)$ مربوط به یک دسترسی $\sigma$ با جابه‌جایی به جلو حداکثر ۴ برابر هزینه‌ی $C_H(\sigma)$ مربوط به دسترسی $\sigma$ با هر مکاشفه‌ی دیگر H است، با فرض این که در هر دو مکاشفه، لیست اولیه یکسان است.

نص
ناشر کتاب‌های فنی، مهندسی، دانشگاهی، کامپیوتر و مدیریت

# فهرست بخشی از کتاب‌های «مؤسسه علمی فرهنگی نص»

**توجه:** کتابهایی که با علامت * مشخص شده جزء چاپهای جدید مؤسسه می‌باشد.

## کتاب‌های مهندسی

| ردیف | عنوان | مولف/ مترجم | قیمت به تومان / تعداد صفحات |
|---|---|---|---|
| ۱ | مبانی فیزیک(۱) مکانیک/ ویراست ۷/ چاپ سوم/ تمام رنگی/ با CD | هالیدی- رزنیک/ دیانی | ۱۲۰۰۰/ ۴۰۸ ص |
| ۲ | مبانی فیزیک الکتریسیته/ ویراست ۷/ چاپ دوم/ تمام رنگی/ با CD | هالیدی- رزنیک/ دیانی | ۹۵۰۰/ ۴۱۶ ص |
| ۳ | مبانی فیزیک سیالات، موج، حرارت/ ویراست ۷/ تمام رنگی/ باCD | هالیدی- رزنیک/ دیانی | ۴۵۰۰/ ۲۴۸ ص |
| ۴ | مبانی الکترومغناطیس همراه با میدان‌ها و امواج/ ویراست ۵/ جلد سخت/ دو رنگ/ با CD | ماتیو ان .او سدیکو/ دیانی | ۱۹۰۰۰/ ۹۱۲ ص |
| ۵ | مبانی الکترومغناطیس/ جلد نرم/ دو رنگ/ با CD/ جلد نرم/ویراست ۵ | ماتیو ان .او سدیکو/ دیانی | ۱۱۰۰۰/ ۵۶۸ ص |
| ۶ | شیمی/ جلد ۱/ تمام رنگی/ قطع رحلی | سومدال/ محمود دیانی | ۹۵۰۰/ ۴۰۰ ص |
| ۷ | شیمی/ جلد ۲/ تمام رنگی/ قطع رحلی | سومدال/ محمود دیانی | ۹۵۰۰/ ۴۰۸ ص |
| ۸ | الکترونیک (مدار، طراحی ،کاربرد)/ ویراست ۷/ تمام رنگی/ با CD/ قطع رحلی | فلوید/ دیانی | ۲۸۰۰۰/ ۸۳۲ ص |
| ۹ | طراحی اجزاء ماشین/ جلد ۱/ چاپ ۶/ دورنگ/ با CD/ قطع رحلی | شیگلی/ زارعپور | ۱۸۰۰۰/ ۸۸۸ ص |
| ۱۰ | طراحی اجزاء ماشین/ جلد ۲/ چاپ ۴/ دورنگ/ با CD/ قطع رحلی | شیگلی/ زارعپور | ۹۵۰۰/ ۳۹۲ ص |
| ۱۱ | تحلیل مهندسی مدار/ ویراست ۷/ تمام رنگی/ با CD/ قطع رحلی/چاپ پنجم | هیت/ دیانی | ۴۵۰۰۰/ ۷۲۰ ص |
| ۱۲ | تحلیل مهندسی مدار/ ویراست ۶/ چاپ پانزدهم/ دورنگ/ با CD | هیت/ دیانی | ۲۵۰۰۰/ ۶۴۰ ص |
| ۱۳ | آموزش نرم افزار CFX با DVD | مهرجوئی- نیک‌فر | ۶۰۰۰/ ۲۹۶ ص |
| ۱۴ | مدارهای میکروالکترونیک/ جلد ۱/ چاپ ۴/ دورنگ/ با CD | سدره –اسمیت/ دیانی | ۱۵۰۰۰/ ۷۵۲ ص |
| ۱۵ | مدارهای میکروالکترونیک/ جلد ۲/ چاپ ۳/ دورنگ/ با CD | سدره –اسمیت/ دیانی | ۱۵۰۰۰/ ۷۲۸ ص |
| ۱۶ | مدارهای میکروالکترونیک/ویراست ۶/جلد ۱/ دو رنگ/ با DVD | کنت اسمیت/ سدره/ دیانی | ۱۴۰۰۰/ ۵۷۶ ص |
| ۱۷ | مدارهای میکروالکترونیک/ویراست ۶/ جلد ۲/ دو رنگ/ با DVD | کنت اسمیت/ سدره/ دیانی | ۱۶۰۰۰/ ۵۶۵ص |
| ۱۸ | الکترونیک ۳/ ویراست ۲/ دو رنگ/ چاپ ششم | دکتر نشاطی | ۱۸۰۰۰/ ۴۴۸ ص |
| ۱۹ | بررسی و طراحی مدارهای الکترونیکی الکترونیک ۱ | دکتر نشاطی | ۸۵۰۰/ ۴۶۴ ص |
| ۲۰ | تحلیل و طراحی مدارهای مخابراتی/ دورنگ/ چاپ دوم/ ویراست دوم | دکتر نشاطی | ۱۸۰۰۰/ ۴۸۸ ص |
| ۲۱ | مدارهای مخابراتی(تحلیل غیرخطی، طراحی و شبیه‌سازی)/ دورنگ | دکتر عبدی‌پور | ۱۶۰۰۰/ ۳۵۲ص |
| ۲۲ | مهندسی کنترل/ اوگاتا/ ویراست پنجم/ چاپ هفتم/ جلد سخت | اوگاتا – دیانی | ۳۵۰۰۰/ ۹۴۴ ص |
| ۲۳ | ماشین‌های الکتریکی/ ویراست ۵/ چاپ چهارم/ جلد سخت | چاپمن/ دیانی | ۲۵۰۰۰/ ۶۷۲ص |
| ۲۴ | ماشین‌های الکتریکی/ ویراست ۶/ چاپ دوم | فیتزجرالد/ دیانی | ۲۰۰۰۰/ ۶۴۸ ص |

| | | | |
|---|---|---|---|
| ۲۵ | بررسی سیستم‌های قدرت/ چاپ ششم | استیونسون/ دیانی | ۱۵۰۰۰/ ۴۳۲ ص |
| ۲۶ | سیستم‌های مخابراتی/ ویراست ۴ / چاپ یازدهم/ جلد سخت | کارلسون/ دیانی | ۱۴۰۰۰/ ۸۰۰ ص |
| ۲۷ | اپتیک/ قطع رحلی/ جلد سخت | هشت/ دیانی | ۱۸۰۰۰/ ۶۴۰ ص |
| ۲۸ | سیگنالها و سیستمها/ ویراست ۲/ چاپ ۲۴/ جلد سخت | اپنهایم/ دیانی | ۳۲۰۰۰/ ۸۸۰ ص |
| ۲۹ | تحلیل و طراحی مدارهای مجتمع دیجیتال (الکترونیک دیمیتال) | هامس- جکسون/ دیانی | ۴۸۰۰/ ۵۵۲ ص |
| ۳۰ | طراحی دیجیتال مدار منطقی/ ویرایش چهارم/ دورنگ/ با CD/ چاپ سوم | مانو- چیلتی/ دیانی | ۹۵۰۰/ ۵۷۶ ص |
| ۳۱ | تحلیل و طراحی مدار منطقی دیجیتال/ چاپ هفتم/ جلد سخت | تلسون/ دیانی | ۲۵۰۰۰/ ۷۸۴ ص |
| ۳۲ | مدارهای مجتمع دیجیتال (الکترونیک دیمیتال)/ چاپ دوم/ جلد سخت | رابی- چاندراکازان/ شیری | ۲۸۰۰۰/ ۷۱۲ ص |
| ۳۳ | مدارهای مجتمع آنالوگ/ ویراست ۳/ چاپ هفتم | گری/ امسانی | ۲۵۰۰۰/ ۷۵۲ ص |
| ۳۴ | طراحی مدارهای مجتمع CMOS آنالوگ/ چاپ هفتم | رضوی/ شیری- معارفی | ۲۵۰۰۰/ ۶۵۶ ص |
| ۳۵ | میکروالکترونیک RF/ویراست اول/ چاپ سوم | رضوی/ شیری | ۱۲۰۰۰/ ۳۳۶ ص |
| ۳۶ | عملکرد و کاربردهای PLC در اتوماسیون صنعتی/ چاپ سوم | وارناک/ صفوی- شجاعی | ۹۵۰۰/ ۴۹۶ ص |
| ۳۷ | VHDL مقدماتی از شبیه‌سازی تا سنتز/ چاپ دوم/ با CD | یالامانچی/ نکویی- زعمری | ۴۵۰۰/ ۳۳۶ ص |
| ۳۸ | تکنیک پالس/ ویراست ۲/ چاپ ۱۷ | معتمدی- نشاطی | ۱۵۰۰۰/ ۴۴۰ ص |
| ۳۹ | میکروکنترلرهای ۸۰۵۱/ ویراست دوم/ با CD/ چاپ دوم | مزیدی/ محمدی | ۱۲۰۰۰/ ۶۶۴ ص |
| ۴۰ | میکروپروسسورها (۸۰-Z، ۸۰۸۰، ۸۰۸۵)/ چاپ چهارم | افتبک/ دیانی- ارشدینژاد | ۱۲۰۰۰/ ۷۳۶ ص |
| ۴۱ | میکروکنترلرهای PIC (برنامه‌نویسی اسمبلی و C)/ با CD | مزیدی/ سمندری- مختاری | ۱۴۰۰۰/ ۷۵۲ ص |
| ۴۲ | میکروکنترلرهای AVR و کاربردهای آن/ با CD/ چاپ دهم | ره افروز | ۱۶۰۰۰/ ۴۶۴ ص |
| ۴۳ | میکروکنترلرهای AVR/ با CD و برد اختیاری/ چاپ ۲۵ ویراست دوم | کاهه | ۱۵۰۰۰/ ۳۸۴ ص |
| ۴۴ | میکروکنترلر AVR (با پروژه‌های ۱۰۰٪ عملی)/ با CD/ چاپ هشتم | جابر الوندی | ۱۳۰۰۰/ ۳۵۲ ص |
| ۴۵ | راهنمای AVR الوندی */چاپ اول | صمد الهی | ۸۰۰۰/ ۲۳۲ ص |
| ۴۶ | میکروکنترلرهای dsPIC | مظاهریان- پرتوی‌فر- دهاقین | ۹۸۰۰/ ۴۸۸ ص |
| ۴۷ | میکروکنترلر AVR برنامه‌نویسی اسمبلی و C/ با DVD/ چاپ سوم | مزیدی/ نعیمی | ۲۰۰۰۰/ ۶۸۸ ص |
| ۴۸ | مرجع کامل میکروکنترلرهای AVR/ ویراست۲/ با CD/ چاپ سیزدهم | پرتویفر- مظاهریان | ۲۲۰۰۰/ ۶۴۰ ص |
| ۴۹ | مرجع کامل میکروکنترلرهای ARM سری AT91/ با CD/ چاپ سوم/ جلد نرم | طلایی- پورخواجه | ۱۶۰۰۰/ ۵۲۸ ص |
| ۵۰ | مرجع کامل طراحی با FPGA/ با DVD | شکارپزاده | ۹۵۰۰/ ۳۸۴ ص |
| ۵۱ | مرجع کامل SIMATIC Manager/ چاپ چهارم/ با CD | صداقت- صادقی | ۵۵۰۰/ ۴۰۸ ص |
| ۵۲ | مرجع کامل شبکه‌بندی PLC/ با CD/ چاپ سوم | زیمنس/ صداقت | ۹۰۰۰/ ۳۹۲ ص |
| ۵۳ | مرجع کامل Pspice Schematic 9.2/ ویراست دوم/ چاپ چهارم/ با CD | مدیرنیا | ۹۸۰۰/ ۵۲۸ ص |
| ۵۴ | مرجع کامل Pspice با استفاده از OrCAD/ با CD/ چاپ دوم | رشید/ بنواری- ریاضی | ۹۵۰۰/ ۴۸۰ ص |
| ۵۵ | آموزش EWB/ چاپ چهارم/ با CD | حمید یوسفزاده | ۴۵۰۰/ ۲۱۶ ص |
| ۵۶ | رهیافت حل مسئله در سیگنالها و سیستمها/ چاپ چهارم | دیانی | ۱۲۰۰۰/ ۶۰۸ ص |
| ۵۷ | رهیافت حل مسئله در الکترونیک (۱)/ چاپ نهم | دیانی | ۱۴۰۰۰/ ۳۶۸ ص |
| ۵۸ | رهیافت حل مسئله در الکترونیک (۲)/ چاپ هشتم | دیانی | ۱۵۰۰۰/ ۴۹۶ ص |
| ۵۹ | رهیافت حل مسئله در مدار (۱)/ چاپ دهم | دیانی | ۸۵۰۰/ ۴۰۸ ص |
| ۶۰ | رهیافت حل مسئله در مدار (۲)/ چاپ ششم | دیانی | ۷۵۰۰/ ۴۰۶ ص |
| ۶۱ | رهیافت حل مسئله در سیستم‌های کنترل/ چاپ پنجم | دیانی | ۱۸۰۰۰/ ۵۶۰ ص |
| ۶۲ | رهیافت حل مسئله در الکترومغناطیس/ چاپ نهم | دیانی | ۱۶۰۰۰/ ۴۳۲ ص |
| ۶۳ | رهیافت حل مسئله در مخابرات (۱)/ ویراست دوم | دکتر مریخ بیات | ۴۵۰۰/ ۳۰۴ ص |
| ۶۴ | رهیافت حل مسئله در ریاضیات مهندسی/ چاپ دوم | افشار/ مریخبیات | ۷۰۰۰/ ۴۰۰ ص |

| | | | |
|---|---|---|---|
| ۶۵ | حل مسائل تحلیل مهندسی مدار/ جلد اول | اصفهانی | ۲۲۰۰/ ۲۳۲ ص |
| ۶۶ | حل مسائل تحلیل مهندسی مدار/ جلد دوم | اصفهانی | ۲۸۰۰/ ۲۱۲ ص |
| ۶۷ | ANSYS 10/ ویراست سوم/ چاپ پنجم/ با CD | شعبانعلی | ۹۵۰۰/ ۴۵۶ ص |
| ۶۸ | تبدیل فوریه و کاربردهای آن در مهندسی پزشکی | ستارهدان- بهنام | ۲۰۰۰/ ۱۸۴ ص |
| ۶۹ | کنترل فرآیندهای شیمی | استفانوپلوس/ دکترناصر | ۳۸۰۰/ ۶۱۶ ص |
| ۷۰ | بهسازی محیط در شرایط اضطراری | پیتر هاروی/ دکتر ندافی | ۶۰۰۰/ ۳۶۸ ص |
| ۷۱ | MATLAB برای مهندسی کنترل/ چاپ سوم | اوگاتا/ دیانی | ۱۱۰۰۰/ ۴۸۰ ص |
| ۷۲ | آلودگی هوا/ چاپ سوم | کنت وارک و.../ دکتر ندافی و... | ۲۵۰۰۰/ ۶۵۶ ص |
| ۷۳ | بررسی و طراحی مدارهای الکترونیکی (الکترونیک۲)/جلد اول/ ویراست دوم | دکتر نشاطی | ۷۰۰۰/ ۴۴۰ ص |
| ۷۴ | الکترونیک ۲ (جلد دوم)/ دو رنگ | دکتر نشاطی | ۶۵۰۰/ ۳۷۶ ص |
| ۷۵ | مبانی میکروالکترونیک/ دورنگ/ چاپ پنجم | رضوی/ زارع- دیانی | ۳۵۰۰۰/ ۱۰۳۲ص |
| ۷۶ | سیستم وری لاگ/ ویراست دوم | ساترلند/ شکاریزاده | ۵۲۰۰/ ۳۵۲ ص |
| ۷۷ | مولتی‌سیم (آزمایشگاه پیشرفته مجازی الکترونیک) | حمید یوسف‌زاده | ۸۰۰۰/ ۴۰۸ ص |
| ۷۸ | تلویزیون دیجیتال/ چاپ دوم | دکتر فریدون بهنیا | ۴۸۰۰/ ۲۷۲ ص |
| ۷۹ | اصول نمایشگرهای کریستال مایع | دکتر فریدون بهنیا | ۶۰۰۰/ ۳۲۰ ص |
| ۸۰ | تلویزیون سه بُعدی/ دو رنگ * | دکتر فریدون بهنیا | ۷۰۰۰/ ۱۳۶ ص |
| ۸۱ | مقدمه‌ای بر الگوریتم/ (۲ جلدی)/ جلد اول/ ویراست سوم/ ۲ رنگ/ چاپ سوم | کورمن/ دهقان | ۲۰۰۰۰/ ۴۹۶ ص |
| ۸۲ | مقدمه‌ای بر الگوریتم/ (۲ جلدی)/ جلد دوم/ ویراست سوم/ ۲ رنگ/ چاپ دوم | کورمن/ دهقان | ۲۵۰۰۰/ ۷۶۰ ص |
| ۸۳ | سیستم‌های مخابراتی (مقدمه‌ای بر سیگنال و نویز در مخابرات الکتریکی) ۲۰۱۰/ ویراست ۵/ چاپ چهارم | کارلسون/ محمود دیانی | ۲۸۰۰۰/ ۹۲۸ ص |
| ۸۴ | مبانی متروولوژی (ابعاد، جرم، فشار، نیرو، دما، الکتریک و عدم‌قطعیت اندازه‌گیری) | جی.ام.اس. سیلوا/ آذرپرا | ۶۰۰۰/ ۲۲۴ ص |
| ۸۵ | اندازه‌گیری و کالیبراسیون دما/ ویراست دوم | جی.وی.نیکولاس - وایت/ آذرپرا | ۱۲۰۰۰/ ۴۶۴ ص |
| ۸۶ | متروولوژی الکتریکی (کالیبراسیون و عدم‌قطعیت اندازه‌گیری) | آذرپرا | ۶۵۰۰/ ۳۸۴ ص |
| ۸۷ | مدلسازی و کنترل صنعتی/ با CD | مریخ بیات | ۸۵۰۰/ ۴۷۲ ص |
| ۸۸ | اصول انتشار امواج رادیویی | دکتر نرگس نوری | ۵۰۰۰/ ۲۳۲ ص |
| ۸۹ | مهندسی مایکروویو/ دو رنگ/ ویراست چهارم * | پوزار/ نوری | ۲۵۰۰۰/ ۸۱۶ص |
| ۹۰ | آنتنهای هوشمند | جداری/ داداشزاده | ۱۰۰۰۰/ ۴۴۰ ص |
| ۹۱ | مکانیک سیالات/دو رنگ/ ۲ جلدی/ رحلی/ با CD/ جلد اول * | سنمل- جان سیمبالا/ تیلک‌فو - مهرجوئی | ۱۸۰۰۰/ ۴۱۲ ص |
| ۹۲ | تحلیل مدارهای الکترونیکی * | مدبرنیا | ۱۲۰۰۰/ ۷۸۴ ص |
| ۹۳ | پردازش سیگنال گسسته در زمان/ ویراست سوم/ جلد اول/ چاپ دوم | اپنهایم - دیانی | ۲۲۰۰۰/ ۵۹۲ ص |
| ۹۴ | پردازش سیگنال گسسته در زمان/ ویراست سوم/ جلد دوم | اپنهایم - دیانی | ۱۲۰۰۰/ ۶۴۰ ص |
| ۹۵ | حسابان توماس/حساب دیفرانسیل وانتگرال/ ویراست ۱۲/ سه جلدی * جلد اول (دو قسمت) | ویر-هاس-توماس/ دیانی | (۱) قسمت اول-۱۵۰۰۰<br>(۱) قسمت دوم-۱۰۰۰۰<br>(۲)۱۱۰۰۰<br>(۳)۴۰۰۰ |
| ۹۶ | مرجع سیستم‌های حفاظت در برابر صاعقه */ با DVD | سید امیر محمدی | ۷۵۰۰/ ۳۵۲ ص |
| ۹۷ | طراحی و اجرای سیستم‌های ویدئوی حفاظتی (CCTV) * | سید امیر محمدی | ۱۶۰۰۰/ ۳۶۸ ص |
| ۹۸ | مقدمه‌ای بر رشد بلور از مذاب | فری‌پور | ۳۸۰۰/ ۱۷۶ ص |
| ۹۹ | الگوریتمهای بهینه‌سازی الهام گرفته از طبیعت | بیات | ۶۰۰۰/ ۲۱۶ ص |
| ۱۰۰ | طراحی کامپیوترهای X86 و پنتیوم برنامه‌نویسی اسمبلی و مدارهای واسط/ با CD | مزیدی/ امیدوار- کوثری | ۱۶۰۰۰/ ۷۶۰ ص |
| ۱۰۱ | احتمال، متغیر، و فرآیندهای تصادفی (جلد اول)/ دو رنگ | پاپولیس/ دیانی | ۱۶۰۰۰/ ۴۴۸ ص |
| ۱۰۲ | احتمال، متغیر، و فرآیندهای تصادفی (جلد دوم)/ دو رنگ | پاپولیس/ دیانی | ۲۰۰۰۰/ ۵۵۲ ص |

# همکاران نص در شهرستان‌ها

## در شهرستانها کتابهای ما را از این فروشگاهها بخواهید.

| | | |
|---|---|---|
| | خ چهارباغ – ابتدای سید علی خان | ۰۳۱-۳۲۲۱۳۷۵۱ |
| | خ آمادگاه مقابل هتل عباسی | ۰۳۱-۳۲۲۲۶۱۲ |
| | خ چهارباغ – خ سید علی خان | ۰۳۱-۳۲۲۲۳۷۲۵ |
| رشد اهواز | خ حافظ – بین سیروس و نادری – نمایشگاه رشد اهواز | ۰۶۱-۳۲۲۱۷۰۰۱-۳ |
| شهر کتاب اهواز | خ نادری – بین حافظ و فردوسی – ساختمان فرهاد | ۰۶۱-۳۲۲۱۲۳۱۴ |
| فروشگاه شرق | خ حافظ – حد فاصل خ نادری و سیروس | ۰۶۱-۳۲۲۱۰۲۵۴ |
| | م شریعتی – جنب بانک رفاه مرکزی | ۰۴۵-۳۲۲۴۹۰۷۲ |
| | | ۰۴۵-۳۳۲۴۳۱۳۸ |
| دانش پژوه ارومیه | خ امام – روبروی آموزش و پرورش | ۰۴۴-۳۲۲۲۷۵۷۸ |
| فروشگاه پژوهش- بابل | خ شریعتی- روبروی دانشگاه صنعتی | ۰۱۱-۳۲۳۵۳۰۸ |
| کتابسرای نگار - بهبهان | خ عدالت – پاساژ مودتی | ۰۶۱-۳۲۲۲۰۷۱۵ |
| ولایت - بروجرد | خ شهدا – پاساژ آینه – طبقه پایین | ۰۶۶-۴۲۶۲۹۵۵۰ |
| فروشگاه حاتمی- بجنورد | خ ۱۷ شهریور جنوبی- ایستگاه دانشگاه | ۰۵۸-۳۲۲۳۸۴۴۵ |
| | خ امام خمینی – بازار بزرگ تربیت – طبقه پایین – پلاک ۷۰۱ | ۰۴۱-۳۵۵۳۶۱۹۶ |
| | خ امام – روبروی مصلی | ۰۴۱-۳۵۵۶۵۴۰۵ |
| | خ امام – فلکه دانشگاه اول خ دانشگاه – پلاک ۲۳۰/۱ | ۰۴۱-۳۳۳۴۱۶۶۹ |
| کتابسرای معراج - دزفول | خ شریعتی – بین فردوسی و خیام | ۰۶۱-۴۲۲۶۱۸۲۱ |
| کتابفروشی مژده رشت | خ امام – جنب سینما انقلاب | ۰۱۳-۳۲۲۲۳۶۳۷ |
| نشر و قلم خرم آباد | خ امام – بالاتر از میدان شهدا | ۰۶۶-۳۲۲۱۱۸۵۹ |
| کتابسرای شریعتی خرم آباد | خ شریعتی – میدان بسیج – بالاتر از بانک مسکن | ۰۶۶-۳۲۲۲۸۷۸۴ |
| | م انقلاب – خ سعدی وسط | ۰۲۴-۳۳۲۲۴۴۸۱ |
| | سعدی وسط – نرسیده به چهارراه سعدی پایین | ۰۲۴-۳۳۲۳۰۵۵۲ |
| کتابسرای معین شاهرود | خ ۲۲ بهمن | ۰۲۳-۳۲۳۹۷۰۲۴ |
| فروشگاه صدرا- شاهرود | بلوار دانشگاه – دانشگاه صنعتی | ۰۲۳-۳۲۲۲۱۴۱۰ |
| کتابسرای پویش شاهین شهر | بلوار امام خمینی – نبش حافظ شمالی – پاساژ معینی | ۰۳۱۲-۵۲۷۴۱۲۹ |
| | خ زند ابتدای خ ملاصدرا روبروی خ اردیبهشت | ۰۷۱-۳۶۲۶۹۹۴۴ |
| | شهرداری – دهنادی | ۰۷۱-۳۳۲۴۰۵۷۱ |
| رسالت شهرکرد | خ مولوی – حسینیه امامیه | ۰۳۸-۳۲۲۵۰۹۹۶ |
| دانشجو شوشتر | جنب دانشگاه آزاد | ۰۶۱-۳۶۲۳۶۸۵۸ |
| | شعبه ۱: چهارراه طالقانی نبش کوچه نسیم طبقه زیرین فروشگاه هامون | ۰۲۶-۳۲۲۶۲۹۱۱ |
| | شعبه ۲: رجائی‌شهر فلکه دوم چهارم غربی داخل پاساژ فاطمی | ۰۲۶-۳۴۴۳۷۵۳۳ |
| | خ دکتر شریعتی – چهارراه کاظمی بعد از بانک صادرات | ۰۳۴-۳۲۲۲۴۶۰۸ |
| دانشمند کرمانشاه | چهارراه اجاق پشت پاساژ سعید | ۰۸۳۱-۷۲۸۲۰۴۴ |
| انتشارات دلیل ما- قم | خ صفاییه - مقابل کوچه ۳۸ | ۰۲۵-۳۷۷۳۷۰۱۱ |
| انتشارات امیرکبیر ملایر | چهارراه خیام – پاساژ قصرنور | ۰۸۱-۳۲۲۱۱۱۴۵ |
| | خ سعدی - پاساژ مهتاب شماره ۲۵ | ۰۵۱-۳۲۲۵۱۹۲۳ |
| | شعبه ۱: میدان تقی‌آباد – ابتدای احمدآباد | ۰۵۱-۳۸۴۱۸۰۷۰ |
| | شعبه ۲: میدان آزادی – روبروی پارک ملت | ۰۵۱-۳۸۸۲۹۵۸۹ |
| | خ سعدی – بازارچه کتاب | ۰۵۱-۳۲۲۵۷۳۲۷ |
| انتشارات دانشجو همدان | ابتدای خ مهدیه انتشارات دانشجو | ۰۸۱-۳۸۲۷۲۶۳۳ |
| جهان دانش همدان | میدان امام - ابتدای خ شریعتی | ۰۸۱-۳۲۵۲۴۳۴۴ |
| | میدان آزادی - ابتدای خ فرخی | ۰۳۵-۳۶۲۷۱۵۳۹ |

## ویژگی‌های کتابهای نص :

| | |
|---|---|
| متن روان و رسا | قیمت مناسب |
| کیفیت در چاپ | ساختار زیبا |

# روش های خرید از موسسه نص

**خرید تلفنی**

با تماس تلفنی کتابهای درخواستی برایتان ارسال می‌شود.

**خرید مستقیم**

با مراجعه حضوری به دفتر انتشارات کتابهای موسسه را با تخفیف خریداری کنید.

**خرید پستی**

مبلغ کتابهای درخواستی موسسه را به حساب جاری سپهر [redacted] نزد بانک صادرات بنام انتشارات نص واریز و رسید آنرا همراه‌با آدرس دقیق پستی به آدرس موسسه پست یا فکس نمایید.

**خرید از فروشگاه**

با مراجعه به فروشگاه نص علاوه بر خرید کتابهای انتشارات می‌توانید در جریان جدیدترین کتابهای منتشره در بازار کتاب قرار گیرید.

**اطلاع‌رسانی**

برای علاقه‌مندان با ارسال درخواست (کتبی یا فکس) بروشور و لیست کتابها ارسال می‌شود.

**خرید اینترنتی** www.nass.ir

با مراجعه به وبگاه نص ضمن اطلاع از آخرین کتابهای نشر یافته با عضویت در سایت، از امکان خرید اینترنتی با کلیه کارتهای شتاب بهره‌مند شوید.


**دفتر انتشارات :**

**تهران - م انقلاب - خیابان منیری جاوید - بن بست مبین - شماره ۶**

**تلفن : ۶۶۴۱۲۳۸۵-۶۶۹۵۳۸۸۳- ۶۶۴۶۵۶۷۴ فکس : ۶۶۹۵۷۶۹۰**

**فروشگاه :**

**تهران- ضلع جنوب شرقی میدان انقلاب - شماره ۲۵**

**تلفن فروشگاه : ۶۶۴۰۵۳۷۲ ص- پ ۸۶۳-۱۳۱۴۵**