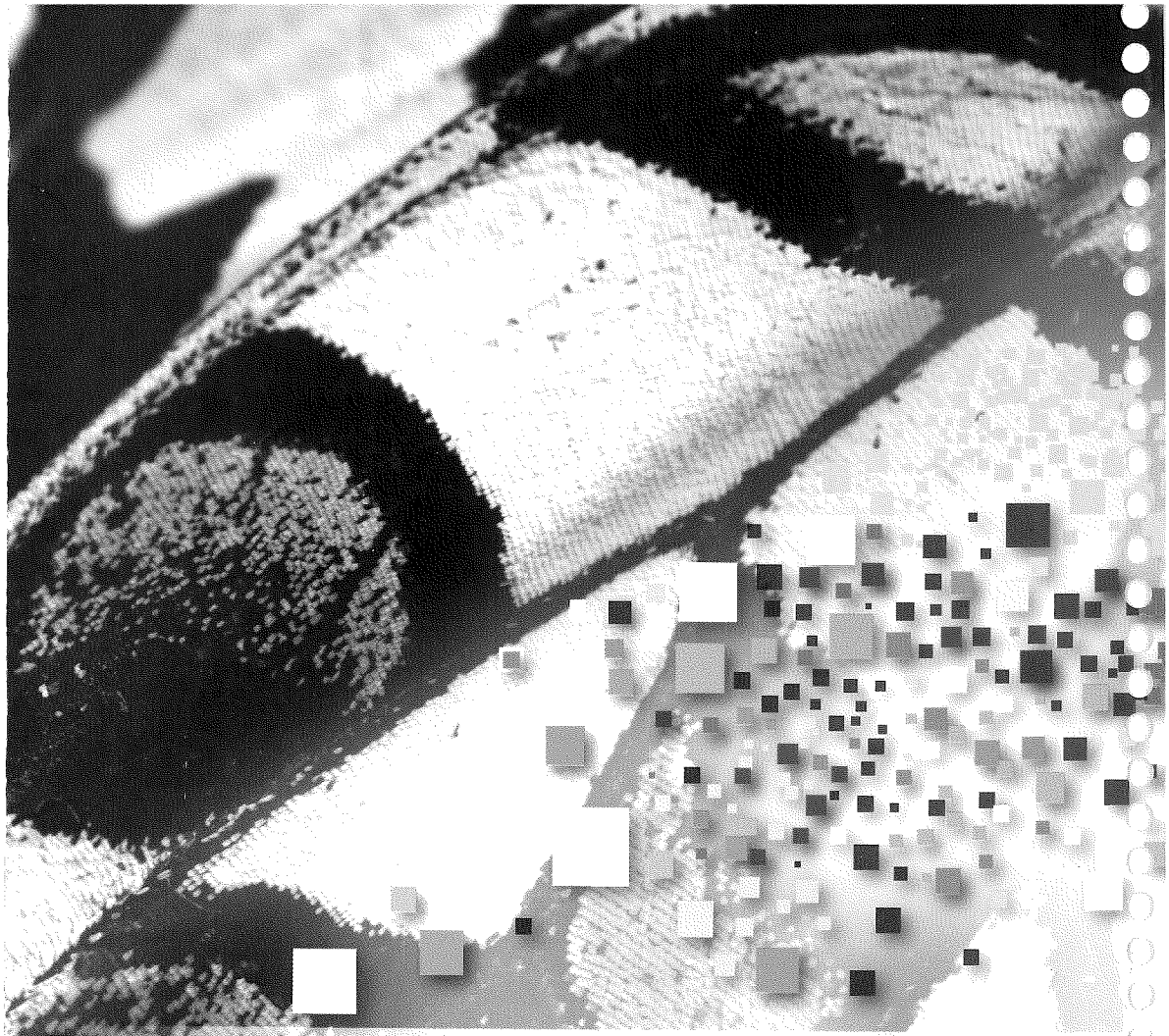
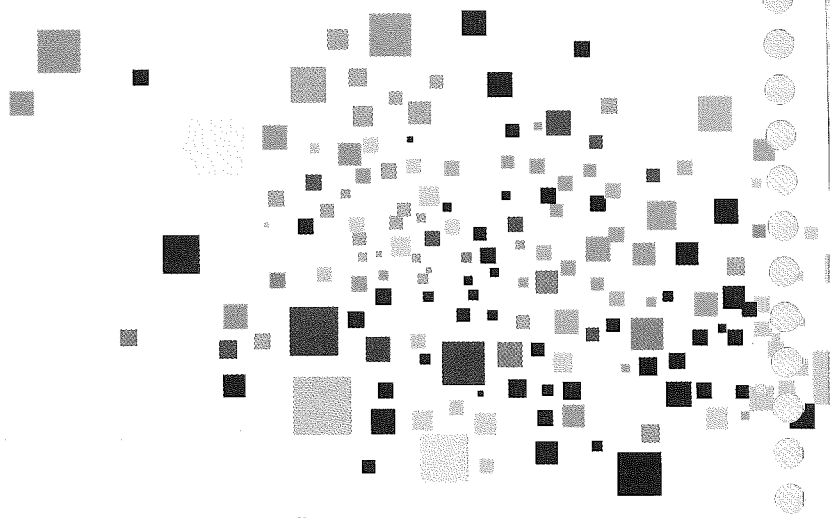


داده‌ساختارها
و مبانی الگوریتم‌ها

محمد قدسی

تقدیر شده ی کتاب فصل

نشارات فاطمی



داده‌ساختارها و مبانی الگوریتم‌ها

محمد قدسی
استاد دانشکده‌ی مهندسی کامپیوتر دانشگاه صنعتی شریف

انتشارات فاطمی

داده ساختارها و مبانی الگوریتم‌ها

مؤلف: محمد قدسی
ویراستار: محمدامین صادقی
ناشر: انتشارات فاطمی
چاپ ششم، ۱۳۹۵
شمارگان: ۱۰۰۰ نسخه
قیمت: ۳۵۰۰۰ تومان
شابک ۷_۵۴۹_۳۱۸_۹۶۴_۹۷۸
ISBN 978-964-318-549-7

آماده‌سازی پیش از چاپ: واحد تولید انتشارات فاطمی
- مدیر تولید: فرید مصلحی
- طراح جلد: علیرضا طاهرنجمی
- نمونه‌خوان: شقایق میرصیافی
- نظارت بر چاپ: علی محمدپور

چاپ و صحافی: خاشع

نشانی دفتر: میدان فاطمی، خیابان جویبار، خیابان میرهادی،
شمارهٔ ۱۴، کدپستی ۱۴۱۵۸۸۴۷۴۱، تلفن: ۸۸۹۴۵۵۴۵ (۲۰ خط)
نمابر: ۸۸۹۴۴۰۵۱ www.fatemi.ir • info@fatemi.ir
نشانی فروشگاه: تهران، خیابان انقلاب، خیابان دانشگاه، تقاطع شهدای ژاندارمری
تلفن: ۶۶۹۷۳۴۷۸ نمابر: ۶۶۹۷۳۷۱۰

قدسی، محمد، ۱۳۳۱ - .
داده ساختارها و مبانی الگوریتم‌ها/مؤلف محمد قدسی. ــ تهران: فاطمی، ۱۳۸۸.
پانزده، ۵۰۷ ص.: مصور (رنگی). جدول، نمودار.
ISBN 978-964-318-549-7
فیپا.
کتابنامه: ص. ۴۹۹.
نمایه
چاپ ششم: ۱۳۹۵.
۱. ساختار داده‌ها ـ ـ مسائل، تمرین‌ها و غیره (عالی). ۲. الگوریتم‌های کامپیوتری. ۳. الگوریتم‌های کامپیوتری ـ ـ مسائل. تمرین‌ها و غیره (عالی). الف. صادقی، محمدامین. ویراستار. ب. عنوان.
۴ق۲س/LB۷۶/۹ ۰۰۵/۷۳۰۷۶
۱۳۸۸
کتابخانهٔ ملی ایران ۱۷۰۴۱۴۵

فهرست

به‌نام آن‌که جان‌را فکرت آموخت

پیش‌گفتار مؤلف

در مورد داده‌ساختارها و طراحی الگوریتم‌ها کتاب‌های زیادی به‌زبان فارسی نوشته یا ترجمه شده است. اما اغلب این کتاب‌ها یا بیش‌تر به بیان مفاهیم داده‌ساختارها می‌پردازند یا تأکید خود را به طراحی الگوریتم‌ها معطوف می‌کنند. یکی از هدف‌های این کتاب، تلفیق این دو موضوع با هم در قالب یک کتاب پایه است. در این کتاب ضمن آن‌که می‌خواهیم شما را با اکثر مطالب داده‌ساختارهای کامپیوتر، در سطح پایه و پیش‌رفته آشنا کنیم، در همه‌ی مراحل نگاهی الگوریتمی به موضوعات مورد بحث داریم.

کار تهیه‌ی محتوای این کتاب را از سال ۱۳۷۴ و با تهیه‌ی جزوه‌هایی از مطالبی که در آن‌زمان تدریس می‌کردم آغاز نمودم. این مطالب را به‌تدریج با تدریس درس‌هایی در دانشکده‌ی مهندسی کامپیوتر دانشگاه صنعتی شریف، چون «روش‌های حل مسئله»، «ساختمان داده‌ها»، «ساختمان داده‌ها و الگوریتم‌ها»، «طراحی و تحلیل الگوریتم‌ها»، «مبانی علم کامپیوتر ۱ و ۲» تکمیل، و از آن‌ها دو جزوه‌ی درسی تهیه کردم.

حدود ۱۰ سال پیش تصمیم گرفتم این جزوه‌ها را که بی‌غلط هم نبودند، به دو کتاب تبدیل کنم، اما هرگز فکر نمی‌کردم که تهیه‌ی اولین کتاب از این مجموعه بیش از ۱۰ سال به‌طول انجامد. طی دو سال اخیر ساعت‌های بسیار زیادی بر روی این کتاب کار کرده‌ام و به مرور، این کتاب به‌عنوان یک محصول مهم از زندگی علمی‌ام درآمد و تکمیل آن به‌صورت یک کتاب درسی کامل و منسجم، شامل تمرین‌ها و پروژه‌های مناسب یکی از هدف‌هایم شد.

در تهیه‌ی مطالب این کتاب از بخش‌هایی از کتاب‌های [۱۱]، [۲]، [۱۳]، [۳] و [۴] (و ویرایش سال ۲۰۰۱ آن [۵] که به کتاب CLRS مشهور است) و چند کتاب دیگر مانند [۱]، [۹]، [۱۲] و [۱۴] که به‌ترتیب زمانی از سی سال پیش، به‌عنوان مراجع درس‌های خود به‌کار برده‌ام استفاده کرده‌ام. در این میان، از کتاب CLRS بیش‌تر استفاده شده است. مثلاً بخش‌هایی از فصل سوم (روش‌های تحلیل الگوریتم‌ها)، فصل پنجم (درهم‌سازی)، فصل ششم (مرتب‌سازی و مرتبه‌ی آماری) و بخش‌هایی از فصل هفتم (داده‌ساختارهای پیشرفته) برگرفته از مطالب این کتاب است.

نقش المپیاد کامپیوتر در تکمیل محتوای این کتاب انکارناپذیر است. ۱۸ سال خدمت در المپیاد کامپیوتر ایران و سروکار داشتن با دانش‌آموزان و دانش‌جویان خوش‌فکر و تیزهوشی که درگیر این المپیاد بودند، به من نکات بسیاری آموخته است. برخی از ایده‌های

نو در این کتاب و تعدادی از تمرین‌ها (اکثر تمرین‌های فصل ۲) و پروژه‌ها، حاصل این تعامل است. مثلاً، بخش‌هایی از فصل ۲، مبتنی بر کتاب [۱۸] است.

در این کتاب، برخی از تمرین‌ها که مشکل‌ترند با علامت ستاره (⋆) و آن‌هایی که بسیار مشکل هستند با علامت دو ستاره (⋆⋆) مشخص شده‌اند.

من سال‌هاست که این کتاب را تقریباً به‌طور کامل، در درسی به‌همین نام تدریس می‌کنم. این اولین درسی است که دانش‌جویان رشته‌ی مهندسی کامپیوتر، پس از گذراندن دروس «مبانی کامپیوتر» و «ساختمان‌های گسسته» می‌گیرند و به‌طور جدی با این مفاهیم آشنا می‌شوند. این کتاب برای همه‌ی دانش‌جویان رشته‌های مهندسی و علوم کامپیوتر و همچنین، دانش‌آموزانی که خود را برای ورود به دوره‌های المپیاد کامپیوتر آماده می‌کنند، مناسب خواهد بود.

به‌زودی اسلایدهایی را که برای آن تهیه کرده‌ام در وبگاهی که به‌منظور پشتیبانی از کتاب توسط انتشارات فاطمی طراحی و راه‌اندازی خواهد شد در اختیار علاقه‌مندان قرار خواهم داد. از خوانندگان محترم تقاضا می‌کنم اشکال‌های احتمالی کتاب را از طریق همین وبگاه با من در میان بگذارند.

سپاس‌گزاری

در تهیه‌ی اولین نسخه‌ی جزوه‌ی درسی‌ام افراد بسیاری کمک کردند. برخی از آنان هم‌اکنون مدارج عالی را به‌اتمام رسانده و استاد دانشگاه یا پژوهشگر برجسته‌ای هستند و برای من افتخاری است که زمانی استاد آن‌ها بوده‌ام. این افراد، به‌ترتیب حروف الفبا عبارت‌اند از: مسعود اسدپور، سیدعلی اکرمی‌فر، اختای ایلغمی، جلال بنایی بروجنی، روزبه پورنادر، طلا تفضلی، آرش رجاییان، آرش رستگار، حبیب رستمی، ساسان دشتی‌نژاد، آزاده شاکری، افسانه فضلی، هشام فیلی، حجت قادری، محمدرضا قهرمانی، مسلم کاظمی، شهاب کمالی، سولماز کلاهی، ناصر عزتی، علی‌رضا ملک‌زاده، محمد مهدیان، مهران مهر، محمودرضا صانعی‌پور، محمدرضا صلواتی‌پور، ابوالفضل هادی اسفنگره، شیوا نجاتی و احسان نوربخش.

برخی دیگر نقش بیش‌تری داشتند: وهاب میرکنی، در رسم اولیه‌ی تعدادی از شکل‌ها و تهیه‌ی مطالب اولیه‌ی بخش ۳–۶؛ و سارا احمدیان، نیما پوردامغانی و هدا اکبری هم در تهیه و ترجمه‌ی برخی از تمرین‌ها و بخش‌هایی از کتاب مرا یاری دادند.

پروژه‌های فصل‌ها منتخبی از تمرین‌های برنامه‌نویسی است که در زمان تدریس این درس‌ها به دانش‌جویان واگذار کرده‌ام؛ کیان میرجلالی در زمانی که دست‌یار درس من بود

تعدادی از آن‌ها را تهیه کرد.

آتنا احمدی نیز در برگردان خودکار واژه‌نامه‌های پیوستِ کتاب مرا یاری داد.

این کتاب بیش از ۷ بار ویرایش شد تا به‌شکل نهایی در آمد. محمد امین صادقی، ویراستار علمی کتاب، درستی الگوریتم‌ها و رویه‌ها را بررسی کرد و پیشنهادهای سودمندی داد. پیش از آن نیز، وحید لیاقت و مرجان قزوینی‌نژاد فصل‌هایی از کتاب را بازخوانی کرده بودند.

آقای فرید مصلحی، مدیر فنی تولید انتشارات فاطمی، نسخه‌های نهایی کتاب را چند بار بازبینی کرد و با تیزبینی خود نکته‌های مفید زیادی را متذکر شد. شکل نهایی کتاب مدیون دقت ایشان است.

از همه‌ی این عزیزان صمیمانه متشکرم.

کل این کتاب را خود تایپ و تماماً با استفاده از نرم‌افزار فارسی‌تک حروف‌چینی کرده‌ام، که کاری سنگین و همراه با صرف وقت زیاد بود. در این رابطه، بهداد اسفهبد در حل برخی مشکلات مرا یاری کرد. نرم‌افزار فارسی‌تک زیر نظر این‌جانب و به‌وسیله‌ی گروه پروژه‌ی فارسی‌تک تهیه شد و از سال ۱۳۷۵ به‌صورت رایگان در اختیار عموم قرار گرفته است. از اعضای این گروه نیز تشکر می‌نمایم.

شکل‌ها را با نرم‌افزار xfig در محیط cygwin خود رسم کرده‌ام. در انتها، آقای مصطفی نوری‌بایگی رنگ‌های شکل‌ها را متناسب با نظر ناشر اصلاح کرد.

بخشی از وقت خود را در زمانی که پژوهشگر مقیم در پژوهشکده‌ی علوم کامپیوترِ پژوهشگاه دانش‌های بنیادی بودم و با کسب اجازه، صرف اتمام این کتاب کردم. از این پژوهشکده هم تشکر می‌کنم.

در انتها، از همسر مهربان و دختران عزیزم که با بردباری خود مرا در تهیه‌ی این کتاب حمایت کردند سپاس‌گزاری می‌کنم.

محمد قدسی،
دانشکده‌ی مهندسی کامپیوتر،
دانشگاه صنعتی شریف
sharif.edu/~ghodsi
شهریور ۱۳۸۸

فصل

۱

معرفی

هدف اصلی این کتاب ارائه‌ی مبانی نظری مورد نیاز برای کسب مهارت لازم در «حل مسئله[۱]» به‌کمک کامپیوتر است. یک کارشناس «علم و مهندسی کامپیوتر[۲]» در طول تحصیل خود باید به توانایی حل مسئله‌های مختلف به‌کمک یک یا چند زبان برنامه‌نویسی مجهز شود. در این فصل با ذکر مثالی، مراحل مختلف حل یک مسئله را تشریح می‌کنیم و با مفاهیم اولیه‌ی هر مرحله آشنا می‌شویم. برخی از این مفاهیم را ابتدا به‌صورت شهودی مطرح می‌کنیم، بدون آن‌که وارد جزئیات شویم. انتظار می‌رود خواننده‌ای که با این عناوین آشنا نیست، ابتدا تصویر مناسبی از این مفاهیم دریافت کند و از ارتباط بین آن‌ها مطلع شود. این برداشت اولیه کمک می‌کند تا با خواندن جزئیات این مفاهیم در فصل‌های بعد، عمق دانش کسب شده‌ی او بیش‌تر شود.[۳]

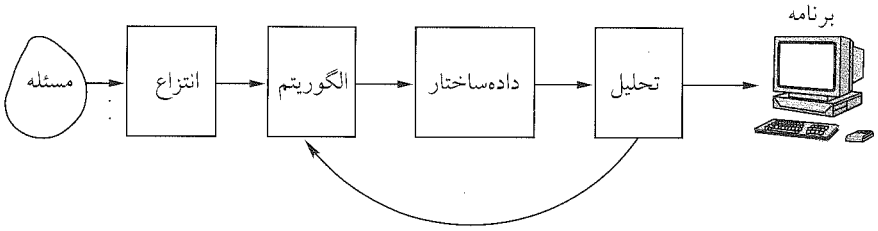
شکل ۱-۱ نمایی کلی از مراحل مختلف حل یک مسئله را نشان می‌دهد. یک مسئله‌ی واقعی حاوی جنبه‌ها و ویژگی‌های مختلف و اغلب متعددی است که به برخی از آن‌ها در طراحی راه‌حل و پیاده‌سازی چندان نیازی نیست. گام اول در حل هر مسئله، شناسایی کامل و تفکیک این ویژگی‌هاست، به‌طوری‌که پس از پالایش، از آن بتوان یک مدل ساده‌ی قابل

---

[۱] problem solving

[۲] computer science and engineering

[۳] به این روش آموزش سطح‌ـاول (breadth-first) گفته می‌شود که برنامه‌های پیش‌نهادی انجمن‌های معتبر اِی‌سی‌اِم و آی‌تریپل‌ای برای رشته‌ی کامپیوتر آن‌را توصیه می‌کنند.



شکل ۱-۱ نمایی کلی از مراحل مختلف حل یک مسئله.

حل ساخت. به گونه‌ی پالایش‌شده‌ی مسئله، مدل «انتزاعی[۴]» می‌گوییم. این عمل انتزاع، یا پالایش ابعاد و ویژگی‌های مسئله، یکی از مفاهیم مهم در علم کامپیوتر است که در بسیاری از مراحل حل مسئله‌ها به‌کار می‌رود. برای یک مسئله می‌توان سطوح مختلفی از انتزاع را تصور کرد که یک سطح آن تبدیل مسئله‌ی واقعی به مدلی ریاضی است که ممکن است چندان شباهتی به مسئله‌ی اصلی نداشته باشد. یافتن یک مدل انتزاعی خوب از مسئله‌ی مورد نظر، که در مورد آن قبلاً مطالعه شده باشد، بخش مهمی از هنر حل مسئله است. فهم مناسب از استقرا، ترکیبیات و ریاضی گسسته به استخراج مدل انتزاعیِ قابل حل کمک شایان می‌کند.

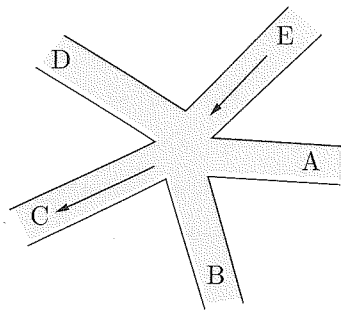
پس از تبدیل مسئله به یک مدل انتزاعی قابل حل با کامپیوتر، الگوریتم حل مسئله‌ی ساده‌شده طراحی می‌شود. در ابتدا، همه‌ی جزئیات این الگوریتم مشخص نیست. اما در همین مرحله، الگوریتم طراحی شده مشخص می‌کند که داده‌های مسئله باید به‌وسیله‌ی چه ساختارهایی عرضه شود و بر روی هر قلم داده‌ی آن چه اَعمالی انجام می‌گیرد. مجموعه‌ی این اطلاعات کمک می‌کند تا برای ذخیره‌ی داده، «داده‌ساختار[۵]»های مناسب انتخاب یا طراحی شوند. الگوریتم طراحی شده را با توجه به داده‌ساختارهای انتخابی می‌توان در همین مرحله تحلیل کرد و میزان زمان اجرا و حافظه‌ی مصرفی الگوریتم را برای اندازه‌های مختلف ورودی حدس زد. اگر این الگوریتم پس از تحلیل راضی‌کننده نبود، لازم است مراحل طراحی الگوریتم و انتخاب داده‌ساختارهای مورد نیاز تکرار شود. توجه کنید که این مراحل معمولاً مستقل از یک زبان برنامه‌نویسی خاص است. پس از نهایی شدن الگوریتم، می‌توان با استفاده از یک زبان برنامه‌نویسی مناسب، الگوریتم مطلوب را پیاده‌سازی کرد و در عمل مورد آزمون قرار داد.

---

[۴] abstract
[۵] data structure

۱-۱ یک مثال: برنامه‌ریزی چراغ‌های راهنما

فرض کنید می‌خواهیم برنامه‌ریزی چراغ‌های راهنمای محل تقاطعی در نزدیکی دانشگاه را که در شکل ۱-۲ نشان داده شده است به‌عهده بگیریم. تعدادی از خیابان‌های منتهی به این تقاطع یک‌طرفه هستند که با پیکان نمایش داده شده‌اند. باید مشخص کنیم که چراغ راهنمای این تقاطع چندزمانه است و در هر بازه‌ی زمانی آن، در چه مسیرهایی مجاز به حرکت و در کدام مسیرها مجبور به توقف پشت چراغ قرمز هستند. این‌کار را می‌خواهیم طوری انجام دهیم که تعداد زمان‌های مختلف چراغ راهنمایی کمینه باشد، و نیز در هر زمان، تا جایی که ممکن است حرکت‌ها در مسیرهای مجاز به عبور باشند.



شکل ۱-۲ یک تقاطع از خیابان‌ها.

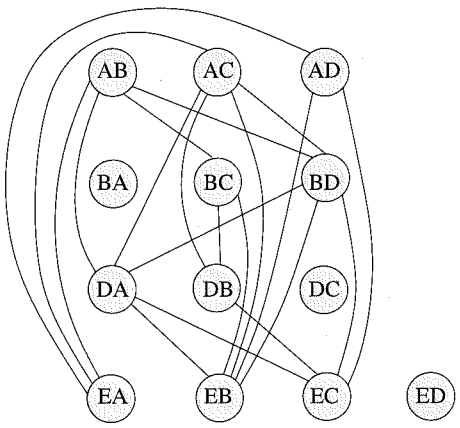
در این مسئله بار ترافیکی خیابان‌ها مهم نیستند یا مساوی در نظر گرفته می‌شوند. در واقع اطلاعات ترافیکی مانند میزان ترافیک، عرض و تعداد مسیرها در هر خیابان نمونه‌هایی از ویژگی‌های مسئله هستند که تأثیری در راه‌حل ندارند و در مرحله‌ی انتزاع نادیده گرفته می‌شوند. شما هم می‌توانید اطلاعات مشابه دیگری را نام ببرید.

توجه کنید که هدف ما، حل این مسئله برای هر ورودی دل‌خواه است که می‌تواند شامل تعداد نامشخص و شاید زیادی خیابان باشد. البته برای یک مسئله‌ی خاص مانند شکل داده شده، می‌توان از روش‌های دیگر مسئله را حل کرد.

در مدل انتزاعی این مسئله آن‌چه مهم است «گردش[۶]»های مختلف این تقاطع و تلاقی این گردش‌هاست. بدیهی است که امکان حرکت گردش‌های متلاقی در یک بازه‌ی زمانی وجود ندارد. اگر گردش از خیابان X به خیابان Y را XY بنامیم، خود گردش‌ها و تلاقی بین

---

[۶] turn



شکل ۱–۳ گراف تقاطع مثال گفته‌شده در شکل ۱–۲.

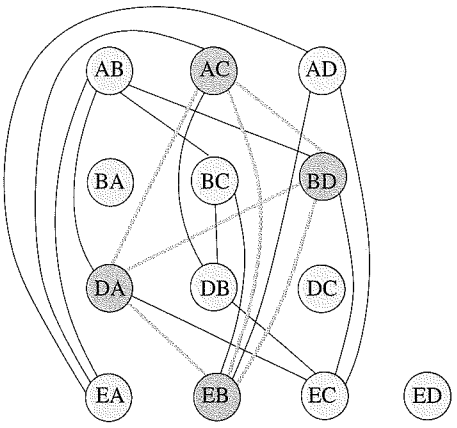
آن‌ها را می‌توان با یک گراف نشان داد که آن‌را «گراف تقاطع[۷]» می‌نامیم. در این گراف، هر گردش یک رأس است و بین دو گردش متلاقی یک یال قرار داده‌ایم. شکل ۱–۳ گراف تقاطع مثال شکل ۱–۲ را نشان می‌دهد. این همان مدل انتزاعی مسئله است که هیچ شباهتی به مسئله‌ی اصلی ندارد. این مدل از این نظر مناسب است که مباحث زیادی برای نظریه و الگوریتم‌های گراف موجود است.

حال باید خود چراغ راهنمایی و بازه‌های زمانی آن‌را نیز مدل کنیم. چراغ راهنمایی یک تقاطع اصطلاحاً چندزمانه است و تعداد زمان‌های آن با توجه به جهت‌های حرکت (گردش به راست، چپ، و غیره) مشخص می‌شود. برای سادگی کار فرض می‌کنیم چراغ مورد نظر k زمان دارد و با رنگ‌های ۱ تا k مشخص شده است (مقدار k را الگوریتم تعیین می‌کند). همچنین در ابتدای هر خیابان محل تقاطع، تابلویی نصب شده است که نشان می‌دهد هر گردش فقط در چه رنگ‌هایی از چراغ مجاز به حرکت است. البته چنان‌چه می‌دانید این کار در عمل به‌صورت دیگری انجام می‌شود.

در مدل گراف، مسئله انتساب یک یا چند رنگ به هر رأس گراف تقاطع است به‌طوری که دو رأسی که دو انتهای یک یال هستند رنگ یک‌سان نداشته باشند. تعداد این رنگ‌ها همان k است و ما می‌خواهیم کم‌ترین مقدار آن‌را به‌دست بیاوریم.

این یک مسئله‌ی شناخته‌شده در نظریه‌ی گراف‌هاست که «رنگ‌آمیزی گراف[۸]» نامیده می‌شود. در این مسئله، یک گراف G با مجموعه‌ی رأس‌های V و یال‌های E داده شده است و هدف، پیداکردن کمینه‌ی تعداد رنگ‌هاست که رأس‌های گراف را بتوان با آن رنگ

[۷] conflict graph
[۸] graph coloring



شکل ۱-۴ یک خوشه‌ی چهار رأسی در گراف تقاطع.

کرد به‌طوری‌که دو رأس همسایه (مجاور) یک‌رنگ نباشند. در مورد این مسئله‌ی انتزاعی اطلاعات زیادی وجود دارد، مثلاً می‌دانیم که این مسئله «ان‌پی‌ـسخت[۹]» است.

روشن است که اگر گراف G یک زیرگراف G' با r رأس و به شکل خوشه[۱۰] داشته باشد، برای رنگ‌آمیزی G' و در نتیجه G به دست‌کم r رنگ نیاز است. شکل ۱-۴ یک خوشه‌ی ۴ رأسی در این گراف را نشان می‌دهد. بنابراین اگر بتوانیم در این گراف بزرگ‌ترین زیرگراف خوشه‌ای‌شکل با اندازه‌ی r پیدا کنیم، می‌توان کل گراف را با دست‌کم r رنگ، رنگ‌آمیزی کرد. مشکل این‌جاست که مسئله‌ی پیدا کردن بزرگ‌ترین زیرگراف خوشه‌ای‌شکل در یک گراف نیز یک مسئله‌ی ان‌پی‌ـسخت است.

از این آگاهی که مسئله‌ی مورد نظر ان‌پی‌ـسخت است، در می‌یابیم که نباید در حالت کلی به‌دنبال راه‌حل سریع برای حل مسئله باشیم، چرا که یک راه‌حل بهینه برای مسئله‌های بزرگ شامل گردش‌های زیاد نمی‌تواند در زمان معقولی جواب بهینه را تولید کند. ممکن است رضایت دهیم که یک الگوریتم سریع، تعداد رنگ‌های «نزدیک به بهینه[۱۱]» پیدا کند.

روش‌های مختلفی برای حل سریع ولی نزدیک به بهینه برای یک مسئله‌ی ان‌پی‌ـسخت

---

[۹] NP-Hard: مجموعه‌ی ان‌پی‌ـسخت شامل چند هزار مسئله‌ی مختلف با کاربردهای فراوان است که تاکنون برای آن‌ها راه‌حل «سریع» و قابل انجام در زمان معقول پیدا نشده است و به‌احتمال زیاد در آینده نیز یافت نخواهد شد؛ این‌که راه‌حل سریعی برای آن‌ها وجود ندارد هم اثبات نشده است. البته ثابت شده است که اگر فقط برای یکی از این مسئله‌ها راه‌حل سریعی پیدا شود، این راه‌حل موجب حل سریع بقیه‌ی مسئله‌ها خواهد شد. البته احتمال پیدا شدن چنین الگوریتمی ضعیف است. منظور از راه‌حل سریع آن است که زمان اجرای آن با اندازه‌ی ورودی مسئله به‌صورت چندجمله‌ای رابطه داشته باشد.

[۱۰] clique: یک گراف کامل که هر رأس آن به بقیه وصل باشد یک خوشه نامیده می‌شود.

[۱۱] near optimal

وجود دارد:

- راه‌حل تقریبی قابل اثبات[۱۲]: که در آن یک الگوریتم سریع برای حل مسئله ارائه می‌شود ولی اثبات می‌شود که اندازه‌ی خروجی (مثلاً تعداد رنگ‌ها در این مسئله) ضریبی از اندازه‌ی خروجی بهینه‌ی مسئله است.
- الگوریتم‌های «مکاشفه‌ای[۱۳]»: با این‌که الگوریتم‌هایی سریع هستند و به‌صورت تقریبی جواب را به‌دست می‌آورند، اما در مورد ضریب تقریب یا میزان خوبی الگوریتم اثباتی وجود ندارد. بسیاری از این الگوریتم‌ها به‌صورت تجربی آزمایش می‌شوند. برخی از این الگوریتم‌ها از روش «حریصانه[۱۴]» برای حل استفاده می‌کنند.

۱-۱-۱ یک راه‌حل حریصانه برای مسئله

یک راه‌حل حریصانه معمولاً ساده است و در هر مرحله از اجرا، تنها بر اساس اطلاعات محلی بخشی از جواب را پیدا می‌کند، به امید این‌که در انتها به جواب بهینه نزدیک شود. البته برخی الگوریتم‌های حریصانه راه‌حل بهینه‌ی مسئله‌های مهمی را به‌دست می‌آورند.[۱۵]

برای مسئله‌ی زمان‌بندی چراغ راهنما یک راه‌حل حریصانه ارائه می‌کنیم، اما در مورد میزان نزدیکی جواب به مقدار بهینه ادعایی نداریم. در این الگوریتم رأس‌های گراف به‌ترتیب دل‌خواهی شماره‌گذاری می‌شوند. در ابتدا رنگ شماره‌ی ۱ را برمی‌داریم و رأس شماره‌ی ۱ را با آن رنگ می‌کنیم. سپس بقیه‌ی رأس‌ها را به‌ترتیب شماره‌شان نگاه می‌کنیم و آن‌هایی را که امکان دارد (یعنی آن‌هایی که رأس مجاور همرنگ نداشته باشند) با رنگ ۱ رنگ می‌کنیم. در مرحله‌ی بعد، رنگ شماره‌ی ۲ را برمی‌داریم و اولین رأسی را که رنگ نشده است و سپس هر رأس دیگری را که بتوانیم با آن رنگ می‌کنیم. این کار را ادامه می‌دهیم تا همه‌ی رأس‌ها رنگ شوند.

کلیات این الگوریتم در رَویّه‌ی[۱۶] CROSS-SOLUTION نگارش شده است. ورودی آن گراف تقاطع G است که رأس‌ها و یال‌های آن به‌ترتیبی دل‌خواه شماره‌گذاری شده‌اند. در این الگوریتم از یک آرایه به‌نام $VerticesWithColor[]$ استفاده می‌شود که درایه‌ی i ام آن

[۱۲] provable approximation algorithm
[۱۳] heuristic
[۱۴] greedy
[۱۵] برخی از این الگوریتم‌ها به دلیل اهمیت‌شان به‌نام طراح‌شان ثبت شده‌اند، مانند الگوریتم‌های هافمن، دایکسترا، پریم، کروسکال و ...
[۱۶] routine

حاوی مجموعه‌ای از رأس‌هاست که با رنگ i رنگ می‌شوند و برای به‌دست آوردن این مجموعه رَویّه‌ی GREEDYSOLUTION فراخوانده می‌شود.

```
CROSS-SOLUTION (G)
    ▷ Input: G گراف
    ▷ Output: مجموعه‌ای از رأس‌ها برای هر رنگ؛ رأس‌هایی که آن رنگ را دارند
    ▷ متغیرهای محلی: ColorNo و آرایه‌ی VerticesWithColor
1   MAKEGRAPH(G)
2   ColorNo← 0
3   while رأس رنگ نشده در G وجود دارد
4      do ColorNo← ColorNo+1
           ▷ همه‌ی رأس‌هایی را که بتوان با ColorNo رنگ کرد به‌دست آور
5          VerticesWithColor[ColorNo]← GREEDYSOLUTION (G)
6   return VerticesWithColor


GREEDYSOLUTION (G)
1   MAKEEMPTYSET (newcolor) ▷ مجموعه‌ی تهی newcolor را ایجاد کن
2   for هر رأس v در G
3     do if رنگ نشده است v
4           then if مجاور هیچ رأسی در newcolor نباشد v
5                   then به v برچسب «رنگ‌شده» بزن
6                        v را به مجموعه‌ی newcolor اضافه کن
7   return newcolor
```

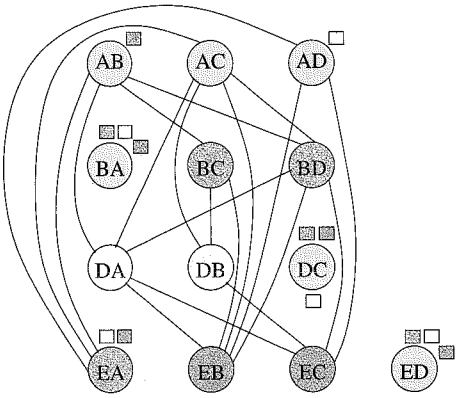
در رویه‌ی GREEDYSOLUTION فرض می‌شود که ممکن است تعدادی از رأس‌های گراف G قبلاً رنگ شده باشند. رأس‌های رنگ‌شده دارای برچسب «رنگ‌شده» (colored) هستند. این ویژگی‌ها به رأس‌های گراف منتسب می‌شوند و قابل خواندن هستند. این الگوریتم رأس‌ها را به‌ترتیب شماره‌شان بررسی می‌کند و اگر رأس v قبلاً رنگ نشده بود آن‌را «رنگ‌شده» ثبت می‌کند و به مجموعه‌ی $newcolor$ اضافه می‌کند. به‌سادگی می‌توان در این الگوریتم کاری کرد که یک رأس بیش از یک رنگ داشته باشد. این مشخص می‌کند که هر گردش در چه رنگ‌هایی از چراغ راهنما می‌تواند حرکت کند.

دقت کنید که رویه‌های فوق به‌صورت «شبه‌کد[۱۷]» نوشته شده‌اند که یک زبان

[۱۷] pseudo-code

برنامه‌نویسی نیست و ما می‌توانیم در آن از هر دستوری استفاده کنیم. بدیهی است که هنگام پیاده‌سازی واقعی، باید محدودیت‌های زبان برنامه‌نویسی را رعایت کنیم.

اگر رأس‌های گراف شکل ۱-۳ را از بالا به پایین و از چپ به راست شماره‌گذاری کنیم، حاصل الگوریتم فوق بر روی این گراف در شکل ۱-۵ نشان داده شده است که در آن رأس‌هایی که با چند رنگ بتوان رنگ کرد نیز مشخص شده‌اند. مثلاً گردش‌های BA، DC و ED در هر زمانِ چراغ راهنما مجاز به حرکت‌اند و مزاحم گردش دیگری نیستند.

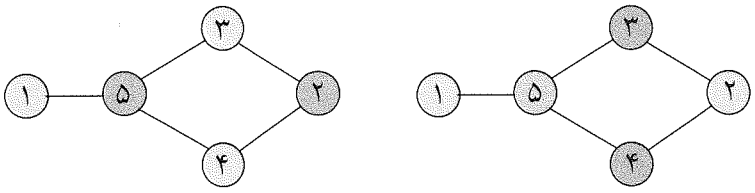


شکل ۱-۵ نتیجه‌ی انجام الگوریتم حریصانه.

این الگوریتم برای این مسئله ۴ رنگ به‌دست می‌آورد که با توجه به وجود بزرگ‌ترین خوشه‌ی ۴ رأسی در این گراف، این تعداد رنگ بهینه است. البته این جواب کاملاً وابسته به ترتیب بررسی رأس‌های گراف است و به‌دست آوردن جواب بهینه در این مثال خاص یک استثنا است. مثلاً اگر همین الگوریتم بر روی گراف شکل ۱-۶ و با ترتیبی از رأس‌ها که مشخص شده است اجرا شود، سه رنگ به‌دست می‌آورد (شکل ۱-۶ (الف))، در حالی که آن‌را با دو رنگ می‌توان رنگ کرد (شکل ۱-۶ (ب)).

۱-۱-۲ داده‌های مسئله

برای پیاده‌سازی الگوریتم حریصانه باید داده‌های مختلف آن‌را به‌صورتی ذخیره کنیم تا انجام اَعمال مختلف بر روی آن‌ها در زمان کوتاهی انجام شود. داده‌های اصلی این الگوریتم «گراف» و «مجموعه» است. قبل از مشخص کردن نحوه‌ی پیاده‌سازی آن‌ها، اَعمال مختلفی



(ب) (الف)

شکل ۱-۶ (الف) الگوریتم حریصانه این گراف را با ۳ رنگ، رنگ‌آمیزی می‌کند. (ب) رنگ‌آمیزی بهینه‌ی این گراف.

را که این الگوریتم بر روی این داده‌ها انجام می‌دهد فهرست می‌کنیم.

الگوریتم، اَعمال زیر را بر روی گراف G انجام می‌دهد:

- MAKEEMPTY (G): ایجاد یک گراف تهی،
- INSERTVERTEX(G, v) و INSERTEDGE(G, u, v): درج یک رأس یا یک یال در G،
- NUMVERTICES(G)، NUMEDGES(G): تعداد رأس‌ها و یال‌های G را برمی‌گرداند،
- LISTOFVERTICES(G): ترتیبی از رأس‌ها را تولید می‌کند،
- ADJACENTVERTICES(G, v): رأس‌های مجاور یک رأس v را تولید می‌کند،
- COLORVERTEX(G, v): رأس v را رنگ می‌کند،
- ISCOLORED(G, v): آیا رأس v رنگ شده است؟ و
- ISADJACENT(G, u, v): آیا رأس u مجاور رأس v است؟

و نیز بر روی مجموعه‌ی Set اَعمال زیر را انجام می‌دهد:

- MAKEEMPTY(S): ایجاد مجموعه‌ی تهی S،
- INSERTSET(S, e): درج عنصر e در S، و
- LISTOFELEMENTS(S): در اختیار گذاردن لیست تک‌تک عناصر موجود در S.

علاوه بر این دو ساختار، متغیرهای محلی دیگری مانند آرایه‌ای از مجموعه‌ها و متغیرهای ساده‌ی دیگر مورد نیاز است.

در این مرحله، از داده‌ساختارهای مناسب برای پیاده‌سازی داده‌های مسئله، یعنی «گراف» و «مجموعه» استفاده می‌کنیم. داده‌ساختاری مناسب و کاراست که مجموعه‌ی اَعمال گفته‌شده را در زمان کوتاهی انجام دهد. داده‌ساختارهای شناخته‌شده‌ای برای هر کدام از این‌دو وجود دارند که آگاهی از آن‌ها و نیز نحوه‌ی تعریف و ساخت داده‌ساختارهای جدید، از مطالب مهمی است که بخش مهمی از این کتاب را تشکیل می‌دهد.

با انتخاب هر داده‌ساختار، زمان اجرای (یا هزینه‌ی انجام) هر یک از اَعمال گفته‌شده را برحسب اندازه‌ی داده‌ی ورودی تحلیل می‌کنیم. سپس، با روش‌هایی که در این کتاب خواهیم گفت زمان اجرای کل برنامه را در هر مرحله تخمین زده و در نهایت الگوریتم و داده‌ساختار کارایی را به‌عنوان راه‌حل انتخاب می‌کنیم. در انتها نیز، با استفاده از یک زبان برنامه‌نویسی مناسب، الگوریتم را پیاده‌سازی می‌کنیم.

۱-۲ گونه‌های مختلف داده

در یک برنامه، داده‌ها از طریق متغیرهای مختلفی در اختیار کاربر قرار می‌گیرند و این متغیرها خود به‌گونه‌های مختلفی پیاده‌سازی می‌شوند که به هر نوع آن یک «داده‌گونه[۱۸]» می‌گوییم. در زبان‌های «ساخت‌یافته[۱۹]» مانند پاسکال[۲۰] و سی[۲۱]، داده‌گونه‌ها به دسته‌های مختلف تقسیم می‌شوند: «داده‌گونه‌های ساده[۲۲]» که هر زبان برنامه‌نویسی آن‌را در ساده‌ترین شکل در اختیار کاربر قرار می‌دهد (مانند «عدد صحیح»، «عدد حقیقی»، «اشاره‌گر»، شاید «رشته» و نظایر آن). «داده‌گونه‌های مرکب[۲۳]» ترکیبی از داده‌گونه‌های ساده یا داده‌گونه‌های مرکبِ ساده‌تر است. مثلاً، «رکورد[۲۴]» از مؤلفه[۲۵]های مختلفی تشکیل شده است که هر یک از داده‌گونه‌ی دیگری است، یا «آرایه[۲۶]» از چندین درایه[۲۷] از یک داده‌گونه‌ی مشخص

---

[۱۸] data type
[۱۹] structured
[۲۰] Pascal
[۲۱] C
[۲۲] simple data types
[۲۳] compound data types
[۲۴] record
[۲۵] field
[۲۶] array
[۲۷] array entry

تشکیل شده است و به هر درایه‌ی آن می‌توان به‌طور مستقیم دست‌رسی داشت.

۱-۲-۱ داده‌گونه‌ی انتزاعی

در طراحی یک برنامه با زبان‌های ساخت‌یافته، تعریف داده به‌صورت انتزاعی انجام می‌شود که به آن «داده‌گونه‌ی انتزاعی[۲۸]» می‌گوییم. این نوع سازمان‌دهی داده از دو اصل پیروی می‌کند:

- انتزاع[۲۹]: پالایش داده و استخراج ویژگی‌های با اهمیت و تفکیک آن‌ها از بخش‌های کم اهمیت و جزیی، و
- بسته‌بندی[۳۰]: مخفی‌کردن جزییات پیاده‌سازی از دید کاربر؛ جزییات داده‌ساختارها و الگوریتم‌های استفاده‌شده در پیاده‌سازیِ اَعمال مختلف بر روی داده‌گونه‌ها، در اختیار کاربر قرار نمی‌گیرد؛ او تنها از عناوین داده‌گونه‌ها برای دست‌رسی به داده‌ی مورد نظر و نیز از عناوین رَویّه‌ها و پارامترهای هریک برای فراخوانی مطلع است.

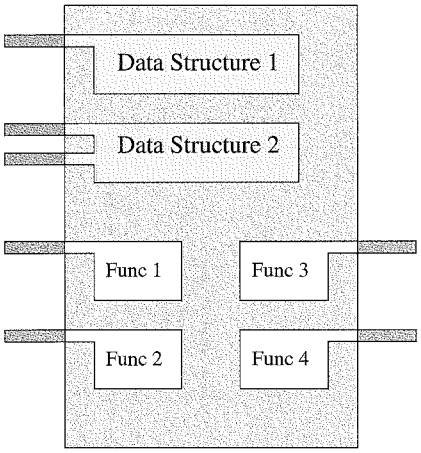
یک داده‌گونه‌ی انتزاعی متشکل از دو قسمت است: داده‌ساختارهای مورد استفاده و نیز اَعمال[۳۱] مختلفی که بر روی آن انجام می‌شوند. شکل ۱-۷ یک داده‌گونه‌ی انتزاعی را نشان می‌دهد که از دو داده‌ساختار و چهار عمل بر روی این داده‌ساختارها تشکیل شده است؛ این داده‌گونه مانند یک آی‌سی است که فقط از طریق ۶ درگاه نشان داده شده می‌توان به آن دست‌رسی داشت.

۱-۲-۲ داده‌ها در زبان‌های شیءگرا

در زبان‌های «شیءگرا[۳۲]» برخلاف زبان‌های ساخت‌یافته، مجموعه‌ای از داده‌های مرتبط به‌هم و نیز اَعمال مربوط به آن‌ها در یک قالب و به اسم «شیء[۳۳]» تعریف می‌شوند. به بیانی دیگر، هر شیء مجموعه‌ای از داده‌ها و اَعمال روی آن‌هاست.

---

[۲۸] Abstract Data Type (ADT)
[۲۹] abstraction
[۳۰] encapsulation
[۳۱] operations
[۳۲] object oriented
[۳۳] object



شکل ۱-۷ داده‌گونه‌ی انتزاعی.

یک داده‌ساختار برای سهولت در انجام اَعمال روی مجموعه‌ای از داده طراحی می‌شود، پس به کمک شیءگرایی می‌توان مستقل از نحوه‌ی پیاده‌سازی داده‌ساختار، مجموعه‌ی رفتارها را به‌صورت یک «میانا[۳۴]» تعریف کرد و با روش پیاده‌سازی، که وابسته به نحوه‌ی نگه‌داری داده‌هاست، اَعمال را تعریف و از آن‌ها در پیاده‌سازی الگوریتم‌ها استفاده کرد.

به‌عنوان مثال، در زبان «جاوا[۳۵]» که یک زبان برنامه‌نویسی شیءگرا است میانایی به نام Set وجود دارد که تمام اَعمال مورد نظر روی «مجموعه» در آن تعریف شده است ضمن آن‌که الزامی در روش پیاده‌سازی این اَعمال ندارد. از طرفی، در این زبان «رده[۳۶]»های متعددی از شیءها وجود دارند که هرکدام این میانا را به روش خاصی پیاده‌سازی کرده است؛ از جمله رده‌ی LinkedList که یک مجموعه را با استفاده از داده‌ساختار «لیست پیوندی[۳۷]» پیاده‌سازی می‌کند و رده‌ی ArrayList که همان میانا را با استفاده از آرایه پشتیبانی می‌کند یا رده‌های HashMap و HashSet که با استفاده از «جدول درهم‌سازی[۳۸]» عملیات مربوط به «مجموعه» در آن پیاده‌سازی شده است.

این قابلیت زبان‌های شیءگرا از این نظر قابل توجه است که می‌توان صرف‌نظر از روش پیاده‌سازی داده‌ساختار و فقط به‌کمک میانای تعریف شده برای آن، الگوریتم مورد نظر را

---

[۳۴] interface یا واسط
[۳۵] java
[۳۶] class
[۳۷] linked list
[۳۸] hash table

طراحی کرد و سپس با توجه به معیارهای کارایی، روش پیاده‌سازی دل‌خواه را انتخاب نمود. مثلاً چنان‌چه حافظه‌ی مصرفی و پیمایش ترتیبی معیار انتخاب باشد بهتر است از LinkedList استفاده شود و اگر سرعت دست‌رسی و جست‌وجو مد نظر باشد انتخاب HashSet یا HashMap مناسب‌تر است.

اگر بخواهیم برای مسئله‌ی برنامه‌ریزی چراغ راهنما داده‌ساختارهایی در یک زبان شیءگرا مانند جاوا تعریف کنیم، می‌توان به‌صورت زیر عمل کرد: هر رأس، یال و گراف را یک شیء در نظر می‌گیریم که دارای مشخصات (داده‌ها) و رفتار (اَعمال) خاص خود است. یک رأس دارای داده‌ساختار ساده‌ای برای نگه‌داری شماره، رنگ و وضعیت آن (رنگ‌شده، رنگ‌نشده) و نیز رفتارهایی برای انجام عملیات ایجاد، تعیین و گرفتن هر کدام از مشخصات آن و نیز مقایسه‌ی دو رأس است.

هر یال هم دارای داده‌ساختارهایی برای نگه‌داری رأس‌های دو سر یال و نیز دستورهایی برای ایجاد یال، تعیین و گرفتن هر کدام از رأس‌های دو سر یال و مقایسه‌ی دو یال است. برای هر گراف نیز داده‌ساختارهای بزرگ‌تری برای نگه‌داری مجموعه‌ی رأس‌ها و یال‌های گراف نیاز است که باید عملیات مربوط به درج و حذف رأس و یال، گرفتن مجموعه‌ی تمام رأس‌ها و یال‌ها و نیز رأس‌های مجاور یک رأس خاص قابل انجام باشند. این عملیات مستقل از روش نگه‌داری مجموعه‌ی رأس‌ها و یال‌ها در گراف وجود دارند و برحسب روش نگه‌داری داده‌ساختارها، پیاده‌سازی می‌شوند.

در پیوست ۱، یک روش پیاده‌سازی از این شیءها و نیز پیاده‌سازی الگوریتم مورد نظر ارائه شده است.

۱-۳ زبان برنامه‌نویسی استفاده‌شده در این کتاب

این کتاب مبتنی بر زبان برنامه‌نویسی خاصی نیست. برای بیان برنامه و رَویّه‌ها از همان شبه‌کدی (زبان سطح بالایی) که در کتاب معروف [۵] به‌کار رفته استفاده می‌شود. این کتاب در اکثر دانشگاه‌های دنیا به‌عنوان کتاب اول درس «داده‌ساختارها و الگوریتم‌ها» استفاده می‌شود و از اعتبار زیادی برخوردار است. به این زبان CLRS می‌گوییم که مخفف حرف اول نویسندگان این کتاب است.

تمرین‌های فصل ۱

سعی کنید مسئله‌های زیر را مدل کنید. ارائه‌ی حل کامل ضروری نیست و شاید از عهده‌ی شما خارج باشد. ایده‌هایی را مطرح کنید که مسئله به چه مدلی تبدیل می‌شود.

۱.۱ می‌خواهیم برنامه‌ی زمان‌بندی یک دوره بازی‌ها با شرکت n تیم با شماره‌های ۱ تا n را به‌دست آوریم. برنامه‌ی بازی‌ها به‌صورت جدولی است که سطر iام آن برنامه‌ی کامل هفته‌ی iام را نشان می‌دهد که در آن مشخص می‌شود هر تیم با چه تیمی بازی خواهد کرد. می‌خواهیم جدول کامل بازی‌ها را طوری به‌دست آوریم تا هر تیم با بقیه‌ی تیم‌ها بازی کند، در هر هفته یک تیم حداکثر یک بازی انجام دهد و تعداد هفته‌های بازی‌ها کمینه شود.

الف) این مسئله را برای هر n دل‌خواه حل کنید. فرض کنید که در ابتدا هیچ بازی انجام نشده است.

ب) فرض کنید تعدادی از بازی‌ها از قبل انجام شده و نباید در برنامه‌ی جدید تکرار شوند. همین مسئله را با این فرض حل کنید.

** **۲.۱** گروهی از دانش‌جویان «زرنگ» رشته‌ی کامپیوتر می‌خواهند ضمن یادگیری کلیه‌ی مطالب درس، شرکت خود در کلاس‌های دانشکده را به حداقل برسانند با این شرط که در هریک از کلاس‌هایی که با هم دارند فقط یک نفر از آن‌ها شرکت کند. فرض بر این است که دیگر اعضای این گروه با استفاده از یادداشت‌های فرد شرکت کننده و بحث با او می‌توانند درس را به‌خوبی فرا بگیرند و نیازی به شرکت در کلاس ندارند.

فرض کنید همه‌ی این دانش‌جویان در تعدادی درس که در هفته n کلاس c_1 تا c_n دارد ثبت‌نام کرده‌اند. کلاس c_i از زمان a_i آغاز و در زمان b_i تمام می‌شود (زمان از ساعت صفر روز شنبه تا ۲۴ روز پنج‌شنبه به‌صورت یک عدد صحیح سراسری بیان می‌شود). مدت زمان لازم برای رفتن از کلاس c_i به کلاس c_j برابر r_{ij} است. بدیهی است که یک نفر نمی‌تواند در دو کلاس که زمان برگزاری آن‌ها مشترک است شرکت کند و نیز یک فرد مأمور می‌شود که به کلاس خاصی برود مشروط بر آن‌که بتواند از ابتدا تا پایان کلاس در آن شرکت کند.

ورودی این مسئله n، زمان‌های a_i و b_i برای هر کلاس درس c_i، زمان r_{ij} برای هر زوج کلاس c_i و c_j است. همه‌ی اعداد صحیح هستند. خروجی، کم‌ترین تعداد دانش‌جویان مورد نیاز در این گروه است و این‌که هر یک از اعضای گروه در کدام کلاس حتماً باید شرکت کند.[۳۹]

[۳۹] سؤال المپیاد دانش‌جویی کامپیوتر در تابستان ۸۲.

پروژه‌های برنامه‌نویسی فصل ۱

رنگ‌آمیزی گراف

همان‌طور که بیان شد، حل رده‌ی وسیعی از مسئله‌ها منجر به حل مسئله‌ی رنگ‌آمیزی گراف می‌شود که هدف آن انتساب یک رنگ از مجموعه‌ی رنگ‌ها به هر رأس گراف است، به‌طوری‌که هیچ دو رأس مجاوری هم‌رنگ نباشند. این یک مسئله‌ی ان‌پی‌ـسخت است، بنابراین یافتن جواب بهینه، که شامل تعداد کمینه‌ی رنگ‌ها است، برای گراف‌های بزرگ امکان‌پذیر نیست، و در نتیجه دست‌یابی به یک رنگ‌آمیزی نزدیک‌به‌بهینه ولی سریع، ارزش‌مند است.

در این فصل، یک روش تقریبی حریصانه برای حل این مسئله بیان شد که در این‌جا آن‌را approx1 می‌نامیم. یک راه‌حل تقریبی دیگر، انتخاب یک رأس با بیش‌ترین تعداد همسایه و رنگ‌آمیزی آن با یک رنگ مجاز و ادامه‌ی این کار به‌صورت بازگشتی بر روی بقیه‌ی گراف است. ما این روش تقریبی دوم را approx2 نام نهاده‌ایم.

به‌عنوان تمرین برنامه‌نویسی، از شما خواسته می‌شود که الگوریتم بهینه (یا exact) و دو الگوریتم تقریبی فوق را برای مسئله‌ی رنگ‌آمیزی گراف پیاده‌سازی کنید و نتایج آن‌ها را از نظر زمان اجرا و تعداد رنگ‌های مصرفی با هم مقایسه کنید. مقایسه‌ی زمان‌های اجرا را به‌صورت یک منحنی برحسب اندازه‌های مختلف ورودی نشان دهید.

ورودی: از ورودی استاندارد بخوانید. در سطر اول ورودی، به‌ترتیب n (تعداد رأس‌های گراف) و e (تعداد یال‌های گراف) با یک فاصله از هم آمده‌اند. در هر یک از e سطر بعد، دو عدد با فاصله از هم هستند که دو سر یک یال از گراف می‌باشند. رأس‌ها از ۱ تا n شماره‌گذاری شده‌اند. می‌دانیم که گراف داده شده هم‌بند است.

خروجی: در خروجی استاندارد بنویسید. در سطر اول، تعداد رنگ‌های لازم و در سطر دوم، n عدد با یک فاصله از هم بنویسید که عدد iام رنگ رأس iام است. رنگ‌ها را از ۱ شماره‌گذاری کنید.

راهنمایی: راه‌حل‌های تقریبی را می‌توان با زمان اجرایی متناسب با n^2 نوشت. اما در مورد پیاده‌سازی الگوریتم بهینه، رنگ‌آمیزی را به صورت بازگشتی انجام دهید. در هر سطح از اجرای تابع، یک رأس را انتخاب و با رنگ‌های مختلف رنگ کنید و تابع سطح بعدی را فراخوانی کنید. سعی کنید هرجا که می‌توانید از ادامه‌ی رنگ‌آمیزی صرف‌نظر کنید تا زمان اجرای برنامه کوتاه‌تر شود. مثلاً، ممکن است رنگ‌آمیزی فعلی نامعتبر شده باشد، یا این‌که تعداد رنگ‌هایی که تاکنون برای رنگ‌آمیزی استفاده شده بیش‌تر از تعداد رنگ‌های بهترین جوابی باشد که تا کنون پیدا شده است. یک ایده‌ی مهم دیگر برای بهبود زمان اجرا، این است که سعی کنید در هر مرحله رأسی را انتخاب کنید که با شماره‌ی رنگ کم‌تری بتوان آن را رنگ کرد (طوری که رنگ‌آمیزی معتبر بماند). این کار فضای جستجو را به طرز

چشم‌گیری کاهش می‌دهد. این نکته را هم به‌خاطر داشته باشید که هر گراف G، دارای رنگ‌آمیزی معتبری با $\Delta(G)+1$ رنگ است.[۴۰]

نمونه‌ی ورودی و خروجی

input	outputs		
	approx1	approx2	exact
8 12 1 2 1 3 1 4 2 3 2 4 3 4 4 5 5 6 6 7 6 8 4 8 5 7	4 4 3 2 1 2 1 3 2	4 1 2 3 4 1 2 3 1	4 4 3 2 1 2 1 3 2
8 7 1 5 2 5 3 5 3 6 4 6 4 7 4 8	3 2 2 3 1 1 2 2 2	2 1 1 1 1 2 2 2 2	2 2 2 2 2 1 1 1 1
4 3 1 3 3 4 4 2	2 1 2 2 1	3 1 1 2 3	2 1 2 2 1

[۴۰] $\Delta(G)$ بیشترین درجه در گراف G است.

فصل

۲

مبانی استقرا و شمارش

در این فصل به مفاهیم استقرا و مبانی روش‌های شمارش می‌پردازیم که در طراحی و تحلیل داده‌ساختارها و الگوریتم‌ها نقش کلیدی دارند.

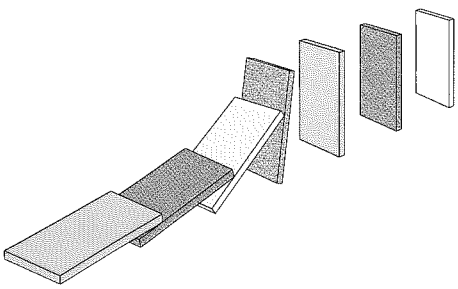
۲-۱ استقرای ریاضی

استقرای ریاضی یکی از مفاهیم مهم در منطق ریاضی است که از آن در طراحی راه‌حل بازگشتی مسئله‌ها، تحلیل و شمارش حالات و نیز در اثبات درستی الگوریتم‌ها و بسیاری موارد دیگر استفاده می‌شود. مفاهیم اولیه‌ی استقرا ساده است، اما استفاده از آن در اثبات برخی از مسئله‌ها پیچیدگی‌هایی دارد که ممکن است به ابتکار، ذکاوت و نیز تجربه نیازمند باشد.

برای آشنایی با مفهوم استقرا یک ردیف از دامینوها[1] را در نظر بگیرید که به‌صورت عمودی پشت سر هم ایستاده‌اند (شکل ۲-۱). فاصله‌ی بین دامینوها به‌قدری است که اگر یکی از آن‌ها بیفتد، دامینوی بعدی هم خواهد افتاد. این مثال، به‌نحوی که خواهیم گفت، استقرای ریاضی را توصیف می‌کند.

---

[1] dominos



شکل ۲-۱ دامینوها، تشابهی با استقرای ریاضی.

فرض کنید دنباله‌ای از گزاره‌های نامتناهی زیر داده شده‌اند:

$$P(۱), P(۲), \ldots, P(n), P(n+۱) \ldots,$$

و می‌خواهیم ثابت کنیم که $P(i)$ ها همه درست هستند. این گزاره‌ها همان دامینوهای مثال فوق هستند و به‌زمین افتادن هر دامینو به‌معنی درست بودن گزاره‌ی متناظر با آن است. اگر بتوانیم ثابت کنیم که $P(۱)$ درست است و نیز برای هر $k > ۱$، از درستی $P(k-۱)$ بتوان درستی $P(k)$ را نتیجه گرفت، ادعا اثبات شده است. انگار که نشان دهیم دامینوی اول حتماً می‌افتد و فاصله‌ی بین دامینوها به‌قدری کم است که افتادن یکی موجب افتادن دامینوی بعدی خواهد شد.

به‌عبارت دیگر، برای اثبات درستی گزاره‌های فوق، کافی است دو شرط زیر را ثابت کنیم:

۱. $P(۱)$ درست است، و

۲. به‌ازای هر $k > ۱$، از درستی $P(k-۱)$ درستی $P(k)$ را نتیجه می‌گیریم (یا $(P(k-۱) \Rightarrow P(k))$.

پس $P(۱) \Rightarrow P(۲) \Rightarrow P(۳) \cdots \Rightarrow P(i)$.

وقتی از استقرا استفاده می‌کنیم، معمولاً بررسی شرط (۱) ساده است و اثبات شرط (۲) نیز از اثبات مستقیم مسئله ساده‌تر است.

در این‌جا «استقرای ضعیف[۲]» و «استقرای قوی[۳]» را مورد بررسی قرار می‌دهیم و در هر مورد تعدادی مسئله حل می‌کنیم.

---

[۲] weak induction
[۳] strong induction

۲-۱-۱ استقرای ضعیف

قضیه‌ی ۲-۱ (اصل استقرای ریاضی ضعیف) اگر m یک عدد صحیح و $P(n)$ گزاره‌ی n ام باشد، به‌طوری که:

۱. $P(m)$ برقرار باشد، و

۲. به‌ازای هر $k > m$، $P(k-1) \Rightarrow P(k)$،

در این صورت، به‌ازای هر $n \geq m$، $P(n)$ برقرار است.

اثبات: فرض کنید که چنین نباشد، یعنی حکم برای همه‌ی مقادیر طبیعی $n \geq m$ برقرار نباشد. در این‌صورت حتماً کوچک‌ترین عددی مثل p وجود دارد که حکم برای $n = p$ برقرار نیست ولی برای تمام مقادیر $n < p$ برقرار است. واضح است که $p > m$ است، بنابراین $p - ۱$ یک عدد صحیح بزرگ‌تر یا مساوی m است. از این نتیجه می‌شود که برای عدد صحیح $p - ۱$ حکم برقرار است، ولی برای عدد صحیح بعد از آن برقرار نیست و این با فرض (۲) بالا متناقض است. □

$P(m)$ را «پایه‌ی استقرا[۴]»، $P(k-1)$ را «فرض استقرا[۵]»، $P(n)$ را «حکم استقرا» و $P(k-1) \Rightarrow P(k)$ را «گام استقرا[۶]» می‌گوییم. برای اثبات یک مسئله با استقرا، ابتدا درستی پایه‌ی استقرا را ثابت می‌کنیم. سپس، با فرض درستی فرض استقرا، گام استقرا را ثابت می‌کنیم، که در نتیجه حکم استقرا اثبات می‌شود. بدون این دو مرحله، اثبات کامل نیست.

۲-۱-۲ استقرای قوی

با فرض درستی پایه‌ی استقرای $P(m)$، گاهی برای اثبات $P(n)$ به‌ازای هر مقدار صحیح $n > m$، لازم است که حکم استقرا را به‌ازای هر عدد صحیح k که $m \leq k < n$، درست فرض کنیم. در چنین مواردی از اصل استقرای قوی کمک می‌گیریم:

---

[۴] induction basis
[۵] induction hypothesis
[۶] induction step

قضیه‌ی ۲-۲ (اصل استقرای ریاضی قوی) اگر m یک عدد طبیعی و $P(n)$ گزاره‌ی nام باشد به‌طوری که:

۱. $P(m)$ برقرار باشد، و

۲. به‌ازای هر $k \geq m$، بتوان از برقراری $P(m), P(m+1), \ldots, P(k)$، برقراری $P(k+1)$ را نتیجه گرفت،

در این صورت، $P(n)$ به‌ازای هر $n \geq m$ برقرار است.

درستی استقرای قوی را می‌توان از اصل استقرای ضعیف نتیجه گرفت. استقرای قوی تکنیکی است که استفاده از آن برای حل برخی از مسئله‌ها لازم است.

۲-۱-۳ مثال‌هایی از استقرا

مثال ۲-۱ هرم اعداد زیر موجود است:

$$\begin{aligned} ۱ &= ۱ \\ ۳+۵ &= ۸ \\ ۷+۹+۱۱ &= ۲۷ \\ ۱۳+۱۵+۱۷+۱۹ &= ۶۴ \\ ۲۱+۲۳+۲۵+۲۷+۲۹ &= ۱۲۵ \\ &\vdots \end{aligned}$$

مجموع اعداد سطر iام یا $T(i)$ را به‌دست آورید.

حدس: مجموع اعداد سطر iام برابر $i^۳$ است، یعنی $T(i) = i^۳$.

پایه‌ی استقرا: $T(۱) = ۱$

فرض استقرا: $T(i) = i^۳$

حکم استقرا: $T(i+۱) = (i+۱)^۳$. با فرض

$$T(i) = a_۱ + a_۲ + a_۳ + a_۴ + \cdots + a_i$$

$$T(i+1) = b_1 + b_2 + b_3 + b_4 + \cdots + b_i + b_{i+1}$$

که $b_1 = a_i + 2$، و نیز برای $1 \le j \le i$ داریم $a_{j+1} = a_j + 2$ و $b_{j+1} = b_j + 2$. بنابراین،

$$\begin{aligned} b_1 - a_1 &= 2i \\ b_2 - a_2 &= 2i \\ &\vdots \\ b_i - a_i &= 2i. \end{aligned}$$

پس $T(i+1) = T(i) + (2i) \times i + b_{i+1}$.

اما می‌دانیم که b_{i+1} برابر است با $k = 1 + 2 + 3 + 4 + \cdots + (i+1)$ امین عدد فرد، یعنی $k = (i+1)(i+2)/2$. پس، $b_{i+1} = 2\frac{(i+1)(i+2)}{2} - 1 = i^2 + 3i + 1$. پس،

$$\begin{aligned} T(i+1) &= T(i) + 2i^2 + i^2 + 3i + 1 \\ &= i^3 + 3i^2 + 3i + 1 \\ &= (i+1)^3. \end{aligned}$$

مثال ۲-۲ (**نامساوی برنولی**) به‌ازای هر عدد صحیح $n \ge 0$ و هر عدد حقیقی $x \ge -1$ ثابت کنید:

$$(1+x)^n \ge 1 + nx \qquad (۱-۲)$$

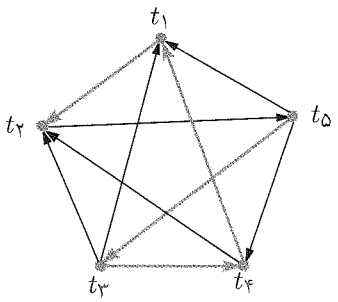
اثبات: پایه‌ی استقرا $(1+x)^0 \ge 1 + 0$ یا $1 \ge 1$ برقرار است. طبق فرض استقرا $(1+x)^n \ge 1 + nx$. حال با توجه به $1 + x \ge 0$، داریم

$$\begin{aligned} (1+x)^{n+1} &= (1+x)(1+x)^n \ge (1+x)(1+nx) \\ &\ge 1 + (n+1)x + nx^2 \ge 1 + (n+1)x. \end{aligned}$$

پس حکم استقرا ثابت شده است. □

دقت کنید که اگر شرط $1 + x \ge 0$ را نداشتیم، نمی‌توانستیم طرفین معادله را در $1 + x$ ضرب کنیم و علامت نامساوی تغییر نکند. باید به این نکته‌ها دقت داشت چون ممکن است موجب خطا در اثبات شود. در بخش ۲-۱-۴ به خطاهای استقرا خواهیم پرداخت.

مثال ۲-۳ فرض کنید n تیم در یک «تورنمنت[۷]» شرکت می‌کنند که در آن هر تیم با هر یک از $n-1$ تیم دیگر دقیقاً یک‌بار بازی می‌کند. اگر هیچ دو تیمی مساوی نشوند، ثابت کنید که دنباله‌ی $\langle t_1, t_2, \ldots, t_n \rangle$ از تیم‌ها وجود دارد به‌طوری که تیم t_1 از تیم t_2، تیم t_2 از تیم t_3، ...، و تیم t_{n-1} از تیم t_n برده باشد[۸] (یا $t_1 \to t_2 \to \ldots \to t_n$). به عنوان مثال، شکل ۲-۲ یک تورنمنت ۵ تیمی را نشان می‌دهد که در آن $\langle t_5, t_3, t_4, t_2, t_1 \rangle$ یک دنباله‌ی جواب است.



شکل ۲-۲ یک تورنمنت با ۵ رأس.

فرض کنید برای هر تورنمنت با n تیم، چنین دنباله‌ای وجود دارد. ثابت می‌کنیم که برای هر تورنمنت $n+1$ تیمی نیز، دنباله‌ی مورد نظر یافت می‌شود. یکی از $n+1$ تیم، مثلاً t_{n+1} را کنار بگذارید. با در نظر گرفتن بازی‌های بین n تیم باقی‌مانده، یک تورنمنت n تیمی داریم و طبق فرض استقرا، دنباله‌ی $\langle t_1, t_2, \ldots, t_n \rangle$ وجود دارد. یعنی،

$$t_1 \to t_2 \to \ldots \to t_{n-1} \to t_n.$$

توجه کنید که شماره‌گذاری تیم‌ها در اختیار ماست (شکل ۲-۳(الف)).

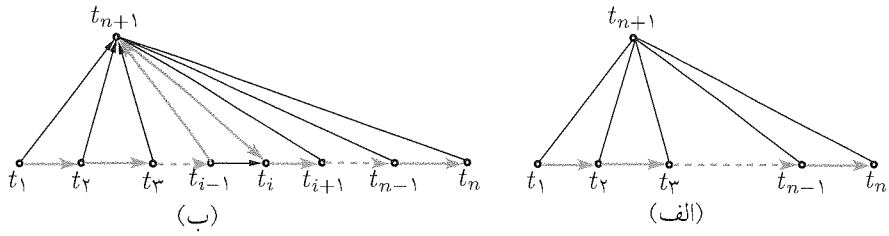
اثبات: پایه‌ی استقرا: در یک تورنمنت با $n = 1$ تیم، دنباله‌ی $\langle t_1 \rangle$، جواب است.

اگر t_{n+1} به t_n باخته باشد، دنباله‌ی $\langle t_1, t_2, \ldots, t_n, t_{n+1} \rangle$ جواب مسئله است. وگرنه، فرض کنید t_i اولین تیمی باشد که به t_{n+1} باخته است. چنین تیمی وجود دارد زیرا t_{n+1} به t_n باخته است. پس t_{i-1}، تیم t_{n+1} را و t_{n+1} هم تیم t_i را برده است. بنابراین دنباله‌ی $\langle t_1, t_2, \ldots, t_{i-1}, t_{n+1}, t_i, \ldots, t_n \rangle$ جواب مسئله است. شکل ۲-۳(ب) این نکته را نشان می‌دهد. □

---

[۷] tournament

[۸] به زبان نظریه‌ی گراف، یعنی در تورنمنت با n رأس، یک مسیر همیلتونی وجود دارد.



شکل ۲-۳ اثبات وجود دنباله‌ی مورد نظر در دوره‌ی بازی‌های $n+1$ تیمی.

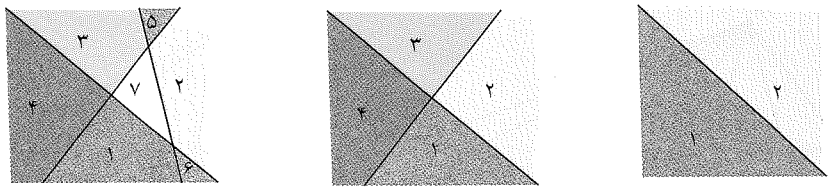
مثال ۲-۴ یک ردیف از سربازها با هر سوتِ فرمانده، بعضی به چپ و بعضی به راست می‌چرخند. پس از این مرحله، در هر سوت هر دو سربازی که روبه‌روی هم ایستاده‌اند عقب‌گرد می‌کنند. ثابت کنید بعد از مدتی سربازها از حرکت می‌ایستند.

اثبات: از استقرا بر روی تعداد سربازها استفاده می‌کنیم. از سمت راست، نخستین سرباز را در نظر بگیرید. اگر این سرباز هیچ‌گاه برنگردد، می‌توانیم او را کنار بگذاریم و در نتیجه بقیه‌ی سربازها طبق فرض استقرا، در نهایت متوقف می‌شوند. ولی اگر این سرباز در یک مرحله عقب‌گرد کند، باید قبل از این مرحله به سمت چپ بوده باشد و پس از عقب‌گرد به سمت راست برگردد. بنابراین، این سرباز هیچ‌گاه حرکت نخواهد کرد. پس می‌توانیم در این مرحله او را کنار بگذاریم و در نتیجه بقیه‌ی سربازها بنا بر فرض استقرا متوقف خواهند شد. □

مثال ۲-۵ n خط دو به دو متقاطع در صفحه‌ای نامتناهی داده شده‌اند. فرض کنید که هیچ سه خطی از این خط‌ها هم‌رس نیستند. تعداد ناحیه‌های ایجادشده بین این n خط را دقیقاً محاسبه کنید.

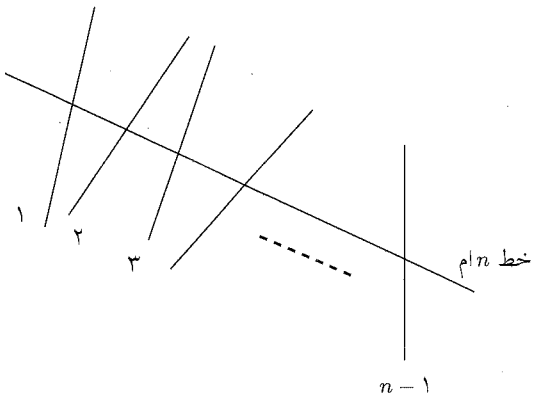
حل: فرض کنید که این تعداد برابر $S(n)$ است. خط‌ها را به‌ترتیب رسم می‌کنیم. اگر پس از رسم خط $n-1$ام بتوانیم تعداد ناحیه‌هایی که به $S(n-1)$ اضافه می‌شود (یعنی $S(n)-S(n-1)$) را به‌دست بیاوریم، مسئله با استقرا حل می‌شود. برای پایه‌ی استقرا، بدیهی است که $S(1)=2$ (شکل ۲-۴ را ببینید).

خط nام هرکدام از بقیه‌ی خط‌ها را در یک نقطه قطع می‌کند. بنابراین بر روی خط nام $n-1$ نقطه‌ی تلاقی با خطوط قبلی خواهیم داشت (شکل ۲-۵).



$S(1) = ۲$ (الف) $S(۲) = ۴$ (ب) $S(۳) = ۷$ (پ)

شکل ۲-۴ شمارش ناحیه‌های بین خط‌ها در یک صفحه



شکل ۲-۵ تلاقی خط nام با خط‌های قبلی.

هیچ‌یک از این نقاط تلاقی تکراری نیست، پس خط nام به n پاره‌خط تقسیم می‌شود. هر یک از این پاره‌خط‌ها از درون یکی از ناحیه‌های قبلی می‌گذرد و لذا آن ناحیه را به دو ناحیه‌ی جدید تقسیم می‌کند. پس n ناحیه به ناحیه‌هایی که با $n-۱$ خط ایجاد شده بود اضافه می‌شود.

$$S(n) = \begin{cases} ۱ & n = ۲ \\ S(n-۱) + n, & n > ۲ \end{cases}$$

به چنین رابطه‌ای «رابطه‌ی بازگشتی[۹]» می‌گوییم که در بخش ۳-۵ در مورد روش‌های حل‌شان مفصل بحث خواهیم کرد. در این‌جا، برای حل این رابطه از جای‌گذاری خودش

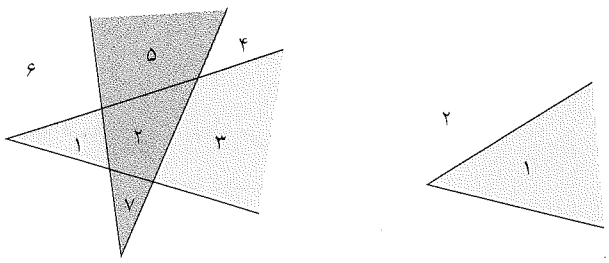
[۹] recurrence relation

در مراحل مختلف استفاده می‌کنیم، که نتیجه می‌دهد:

$$\begin{aligned} S(n) &= S(n-۲)+n-۱+n \\ &= S(n-۳)+n-۲+n-۱+n \\ &\vdots \\ &= S(۱)+۲+۳+\ldots+n \\ &= ۲+۲+۳+\ldots+n \\ &= \frac{n(n+۱)}{۲}+۱ \end{aligned}$$

مثال ۲-۶ بیشینه‌ی تعداد ناحیه‌های ایجاد شده با رسم n زاویه (با زاویه‌های دل‌خواه) در یک صفحه را به‌دست آورید.

حل: اگر $D(n)$ جواب باشد،
پایه‌ی استقرا: $D(۱)=۲$ بدیهی است.
فرض استقرا: بیشینه‌ی تعداد ناحیه‌های ایجادشده با $n-۱$ زاویه برابر است با $D(n-۱)$.
زاویه‌ی nام را چنان رسم می‌کنیم که هر ضلع آن هر دو ضلع تمامی زاویه‌های قبلی را قطع کند. برای این‌کار رأس این زاویه را در یک ناحیه‌ی خارجی انتخاب می‌کنیم؛ یعنی ناحیه‌ای که داخل هیچ زاویه‌ای قرار نداشته باشد. به عنوان مثال، ناحیه‌ی ۲ در شکل ۲-۶(الف) و ناحیه‌های ۴ و ۷ در شکل ۲-۶(ب) خارجی هستند.



$D(۱)=۲$ (الف)　　$D(۲)=۷$ (ب)

شکل ۲-۶ شمارش ناحیه‌های بین زاویه‌ها در یک صفحه

حکم استقرا: برای محاسبه‌ی $D(n)$ کافی است مقدار $d = D(n) - D(n-۱)$ را پیدا کنیم. توجه کنید که هنگام رسم ضلع اول زاویه‌ی nام تعداد نیم‌خط‌های موجود در صفحه برابر است با $۲(n-۱)$. پس حداکثر می‌توان $۲(n-۱)$ نقطه‌ی تلاقی با خط جدید ایجاد کرد و تعداد ناحیه‌های اضافه شده $۱ + ۲(n-۱)$ خواهد بود. اکنون تعداد نیم‌خط‌های موجود در صفحه برابر با $۲n - ۱$ است. بنابراین با رسم ضلع دوم زاویه، به تعداد $۲n$ ناحیه‌ی دیگر اضافه می‌شود. اما همان‌طور که در شکل دیده می‌شود به‌دلیل این‌که امتداد دو ضلع وجود ندارد، دو ناحیه از مجموع کل ناحیه‌ها کسر می‌شود. در نتیجه داریم $d = ۴n - ۳$.

حال با باز کردن رابطه‌ی بازگشتی، فرمول دل‌خواه را به‌دست می‌آوریم:

$$\begin{aligned}
D(n) &= D(n-۱) + ۴n - ۳ \\
&= D(n-۲) + ۴[n + (n-۱)] - ۲ \times ۳ \\
&= D(n-۳) + ۴[n + (n-۱) + (n-۲)] - ۳ \times ۳ \\
&\vdots \\
&= D(n-i) + ۴[n + (n-۱) + (n-۲) + \ldots + (n-i+۱)] - i \times ۳ \\
&= D(۱) + ۴[n + (n-۱) + (n-۲) + \ldots + ۳ + ۲] - (n-۱) \times ۳ \\
&= ۲ + ۴(n+۲)(n-۱)/۲ - ۳(n-۱) \\
&= ۲n^۲ - n + ۱.
\end{aligned}$$

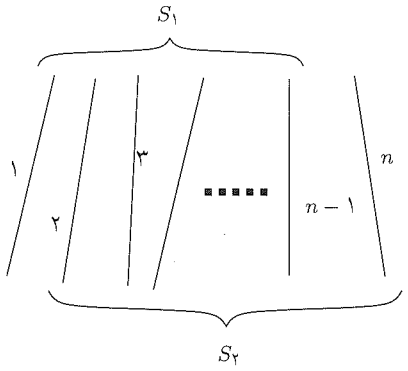
۲-۱-۴ خطاهای معمول در اثبات با استقرا

با ذکر مثال، به چند خطا که ممکن است در فرایند استفاده از استقرا رخ دهد توجه کنید.

۱) n خط در صفحه همه از یک نقطه می‌گذرند.

پایه‌ی استقرا: برای $n = ۲$ ادعا درست است.
فرض استقرا: برای $n - ۱$ خط ادعا درست است.
حکم استقرا: برای n خط ادعا درست است.

مطابق شکل ۲-۷، طبق فرض $n - ۱$ خط از دسته‌ی $S_۱$ از یک نقطه می‌گذرند. $n - ۱$ خط از دسته‌ی $S_۲$ هم به همین دلیل از یک نقطه می‌گذرند. این دو نقطه باید یکی باشند چون این دو دسته خط شامل خط‌های مشترک‌اند. بنابراین همه‌ی n خط از یک نقطه می‌گذرند.



شکل ۲-۷ اثبات غلط با استقرا.

اِشکال اثبات در پایه‌ی استقرا است؛ پایه باید $n = ۳$ باشد، نه $n = ۲$ (چون گام استقرا هنگامی درست است که $n \geq ۳$ باشد) و برای $n = ۳$، پایه‌ی استقرا درست نیست!

۲) رابطه‌ی زیر را ثابت کنید.

$$n = \sqrt{۱ + (n-۱)\sqrt{۱ + n\sqrt{۱ + \cdots}}}$$

پایه‌ی استقرا: برای $n = ۱$ مسئله صحیح است.
طرفین فرمول فوق را به توان دو می‌رسانیم،

$$n^۲ - ۱ = (n-۱)\sqrt{۱ + (n)\sqrt{۱ + \cdots}} \Rightarrow n + ۱ = \sqrt{۱ + n\sqrt{۱ + \cdots}}$$

که همان فرمول فوق برای $n + ۱$ است و مسئله اثبات شده است.

اِشکال اثبات در این‌جاست که در گام استقرا، تقسیم بر $n - ۱$ کرده‌ایم درحالی که پایه‌ی استقرا $n = ۱$ است؛ یعنی گام استقرا هنگامی درست است که $n > ۱$ باشد.

۳) یک ماتریس $n \times n$ از اعداد صحیح نامنفی دارای این ویژگی است که به‌ازای هر درایه‌ی صفر، مجموع درایه‌های سطر و ستونی که شامل آن درایه هستند، دست‌کم n است. نشان دهید که مجموع همه‌ی درایه‌های ماتریس مزبور دست‌کم $\frac{n^۲}{۲}$ است.

اثبات نادرست: به‌ازای $n = ۱$ حکم برقرار است. فرض کنید که حکم به‌ازای $n = k - ۱$ برقرار باشد. ماتریس $k \times k$ را در نظر بگیرید. روشن است که اگر درایه‌ی

صفری موجود نباشد، حکم درست است. هرگاه $a_{ij} = ۰$ باشد، آن‌گاه طبق فرض، مجموع سطر i و ستون j دست‌کم k و مجموع درایه‌های ماتریس $(k-۱) \times (k-۱)$ که از حذف سطر i و ستون j به‌دست می‌آید، طبق فرض استقرا، دست‌کم $\frac{(k-۱)^۲}{۲}$ است. پس نتیجه می‌شود که مجموع درایه‌های ماتریس $k \times k$ اولیه دست‌کم برابر

$$\frac{(k-۱)^۲}{۲} + k = \frac{k^۲ - ۲k + ۱}{۲} + k = \frac{k^۲ + ۱}{۲} > \frac{k^۲}{۲}$$

است و حکم با استقرا ثابت می‌شود.

اِشکال این اثبات چیست؟ اشکال این است که در ماتریس $(k-۱) \times (k-۱)$ که از حذف سطر i و ستون j به‌وجود آمده است، لزومی ندارد به‌ازای هر درایه‌ی صفر، مجموع درایه‌های سطر و ستونی که شامل آن درایه هستند، دست‌کم $k-۱$ باشد.

تمرین‌های بخش ۲-۱

۱.۱-۲ درستی اتحاد زیر را ثابت کنید:

$$\frac{۱}{۱+x} + \frac{۲}{۱+x^۲} + \frac{۴}{۱+x^۴} + \cdots + \frac{۲^n}{۱+x^{۲^n}} = \frac{۱}{x-۱} + \frac{۲^{n+۱}}{۱-x^{۲^{n+۱}}}$$

۲.۱-۲ به‌ازای هر عدد طبیعی n، و با فرض آن‌که برد ۲ امتیاز و باخت صفر امتیاز دارد، ثابت کنید که یک تورنمنت بازی‌ها با $۲n-۱$ تیم وجود دارد که در آن هیچ دو تیمی با هم مساوی نکرده و امتیاز همه‌ی تیم‌ها مساوی باشند.

۳.۱-۲ تعدادی دایره روی صفحه رسم کرده‌ایم. این دایره‌ها صفحه را به تعدادی ناحیه تقسیم کرده‌اند. با استقرا ثابت کنید این ناحیه‌ها را می‌توان با دو رنگ آبی و قرمز طوری رنگ‌آمیزی کرد که هیچ دو ناحیه‌ی مجاوری هم‌رنگ نباشند.

۴.۱-۲ اگر در مسئله‌ی قبل برای هر یک از دایره‌ها یک وتر نیز رسم کنیم، ثابت کنید که هم‌چنان می‌توان ناحیه‌های حاصل را با سه رنگ چنان رنگ کرد که هیچ دو ناحیه‌ی مجاوری هم‌رنگ نباشند.

⋆ **۵.۱-۲** ثابت کنید در یک جمع n نفره، اگر هر کس دست‌کم نیمی از افراد آن جمع را بشناسد، آن‌گاه این n نفر را می‌توان دور یک میز دایره‌ای‌شکل طوری نشاند که هر شخص دو نفر سمت راست و سمت چپ خود را بشناسد.

۶.۱-۲ می‌دانیم که از n سکه‌ی کاملاً مشابه یکی از آن‌ها از بقیه سبک‌تر است. با استقرا نشان دهید که حداکثر با $\lceil \log_۳(n) \rceil$ بار استفاده از یک ترازوی دوکفه‌ای می‌توان سکه‌ی تقلبی را پیدا کرد.

* **۷.۱-۲** سعید در بانکی سه حساب دارد. مقدار پولی که او در هر یک از این حساب‌ها گذاشته است یک عدد صحیح است. او تنها مجاز است که مقداری از پول‌هایش را از یک حساب به حساب دیگر جابه‌جا کند، به‌شرطی‌که وقتی از حساب اول پولی به حساب بعدی منتقل می‌کند، مجموع پول حساب دوم، دو برابر قبل شود.

الف) ثابت کنید که سعید همواره می‌تواند تمام پول‌هایش را به دو تا از حساب‌ها منتقل کند.

ب) آیا سعید همیشه می‌تواند تمام پول‌هایش را به یک حساب منتقل کند؟

۸.۱-۲ یک فدراسیون تنیس، تنیس‌بازان عضو خود را رتبه‌بندی کرده است: به بهترین بازی‌کن رتبه‌ی ۱، به نفر بعدی رتبه‌ی ۲ و ... فرض کنید تنیس‌بازی که با رقیب خود، بیش از ۲ رتبه اختلاف داشته باشد، همیشه پیروز می‌شود. دوره‌ای از مسابقه با شرکت 2^n تنیس‌باز، به شیوه‌ی المپیک برگزار می‌شود. یعنی، بنا بر قرعه‌کشی، هر دو نفر تعیین‌شده با هم بازی می‌کنند، سپس برندگان دور اول هم مطابق همین الگوریتم در گروه‌های دو نفری به رقابت می‌پردازند، به‌طوری‌که تعداد بازی‌کنان هر دور، نصف تعداد بازی‌کنان دور قبل شود. با این ترتیب، بعد از n دور، قهرمان مسابقه معلوم می‌شود. قهرمان مسابقه حداکثر چه رتبه‌ای دارد؟

۹.۱-۲ n ظرف حاوی مواد شیمیایی داده شده است ($n \leq 2$). می‌دانیم که درون دست‌کم $\frac{n}{2}$ از این ظرف‌ها یک ماده‌ی شیمیایی خاص ریخته شده است که ما می‌خواهیم آن را مشخص کنیم. تنها آزمون مجاز استفاده از دستگاهی است که مواد داخل دو ظرف را بررسی و تعیین می‌کند که آیا این دو ماده یکسان هستند یا خیر. ثابت کنید که همواره با $n - 1$ آزمون می‌توانیم ماده‌ای را که در اکثریت ظرف‌ها هست پیدا کنیم. سعی کنید این‌کار را با کم‌تر از $n - 1$ آزمون نیز انجام دهید.

۱۰.۱-۲ $n \geq 4$ نفر را در نظر بگیرید که هرکدام در ابتدا یک خبر جدید را می‌داند (خبرها دوبه‌دو متمایزند). در هر مرحله، دو تن از این افراد به هم تلفن می‌زنند و تمام اخباری را که دارند با هم مبادله می‌کنند. ثابت کنید این افراد می‌توانند با $2n - 4$ بار تلفن‌زدن، همگی از همه‌ی اخبار مطلع شوند.[۱۰]

۱۱.۱-۲ یک رشته‌ی موزون را به این صورت تعریف می‌کنیم:

- x یک رشته‌ی موزون است.
- اگر A و B دو رشته‌ی موزون باشند، (AB) هم یک رشته‌ی موزون است. برای مثال، با تعریف فوق $((xx)x)$ و $(((x(xx))(x(xx)))(xx))$ دو رشته‌ی موزون هستند.

الف) در یک رشته‌ی موزون به هر x عددی به نام عمق آن نسبت می‌دهیم. عمق x تعداد جفت پرانتزهای باز و بسته‌ی متناظر هم است که در دو طرف x قرار دارند. به‌عنوان مثال، عمق هر

[۱۰] مسئله‌ی ۱، مرحله‌ی اول پنجمین المپیاد کامپیوتر ایران

x در رشته‌های موزون فوق به‌صورت زیر است:

$$((x^{۲}x^{۲})x^{۱})$$
$$(((x^{۳}(x^{۴}x^{۴}))(x^{۳}x^{۳}))(x^{۲}(x^{۳}x^{۳})))$$

عمق یک رشته‌ی موزون را بیش‌ترین عمق x های آن می‌گیریم. پس عمق دو رشته‌ی فوق به‌ترتیب ۲ و ۴ است. کم‌ترین عمق هر رشته‌ی موزون با n تا x را برحسب n به‌دست آورید.

ب) یک رشته‌ی موزون خلاصه‌شده، رشته‌ی موزونی است که $'$()ها از آن حذف شده‌اند. مثلاً (xxx) و $(x(xx)(x(xx)(x(xx))))$ رشته‌های موزون خلاصه‌شده‌ی دو رشته‌ی فوق هستند. نشان دهید که از رشته‌ی موزون خلاصه‌شده می‌توان رشته‌ی موزون اولیه را به‌دست آورد.[۱۱]

⋆ **۱۲.۱-۲** اعداد طبیعی را به‌صورت زیر در یک مثلث می‌نویسیم:

				۲۶	...	
				۱۷ ۲۷	...	
			۱۰	۱۸ ۲۸	...	
			۵	۱۱ ۱۹ ۲۹	...	
			۲	۶ ۱۲ ۲۰ ۳۰	...	
		۱	۳	۷ ۱۳ ۲۱ ۳۱	...	
			۴	۸ ۱۴ ۲۲ ۳۲	...	
				۹ ۱۵ ۲۳ ۳۳	...	
				۱۶ ۲۴ ۳۴	...	
				۲۵ ۳۵	...	
				۳۶	...	

الف) رابطه‌ای برای محاسبه‌ی اولین عدد هر سطر پیدا کنید و آن را اثبات کنید.

ب) ثابت کنید که حاصل‌ضرب هر دو عدد مجاور در هر سطر در همان سطر می‌افتد. برای مثال حاصل‌ضرب عددهای ۳ و ۷، عدد ۲۱ است که در همان سطر واقع شده است.[۱۲]

⋆ **۱۳.۱-۲** دو دسته سنگ‌ریزه را که در یکی از آن‌ها m و در دیگری n سنگ‌ریزه قرار دارد، در نظر بگیرید. دو بازی‌کن بازی زیر را با این سنگ‌ریزه‌ها انجام می‌دهند:

هر بازی‌کن در نوبت خود، از یکی از دسته‌ها (یک دسته‌ی دل‌خواه که دست‌کم دو سنگ‌ریزه داشته باشد) دو سنگ‌ریزه برداشته و یکی از آنها را به دسته‌ی دیگر اضافه می‌کند. دو بازی‌کن یکی در میان این حرکت را انجام می‌دهند تا جایی که دیگر حرکتی امکان نداشته باشد. در این هنگام کسی که آخرین حرکت را انجام داده است، برنده‌ی بازی محسوب می‌شود. شرط لازم و کافی برای m و n را به‌دست آورید تا نفر دوم بتواند طوری بازی کند که برنده‌ی بازی شود.[۱۳]

---

[۱۱] مسئله‌ی ۶، مرحله‌ی دوم ششمین المپیاد کامپیوتر ایران
[۱۲] مسئله‌ی ۳، مرحله‌ی اول پنجمین المپیاد کامپیوتر ایران
[۱۳] مسئله‌ی ۲، مرحله‌ی دوم هفتمین المپیاد کامپیوتر ایران

⋆ **۱۴.۱-۲** عمل $\oplus$ را تعریف می‌کنیم: فرض کنید نمایش عددهای x و y در مبنای ۲ به‌صورت:

$$x = x_n x_{n-1} \cdots x_1 x_0 \quad \text{و} \quad y = y_n y_{n-1} \cdots y_1 y_0$$

باشد. (در صورت لزوم در سمت چپ نمایش دودویی عدد کوچک‌تر به تعداد مورد نظر صفر اضافه می‌کنیم.) برای هر i ($0 \le i \le n$)، در صورتی که دقیقاً یکی از دو رقم x_i و y_i برابر با ۱ و دیگری برابر با صفر باشد، a_i را برابر ۱ و گرنه برابر صفر تعریف می‌کنیم. عددی که نمایش آن در مبنای ۲ به‌صورت $a_n a_{n-1} \cdots a_1 a_0$ است، با $x \oplus y$ نشان می‌دهیم. حال الگوریتم زیر را در نظر بگیرید:

```
1   a_0 ← 1; k ← 1
2   a_k ← a_{k-1}
3   a_k ← a_k + 1
4   F ← 1
5   for i ← 0 to k − 1
6     do for j ← 0 to k − 1
7           do if a_k = a_i ⊕ a_j
8                 then F ← 0
9   if F = 1
10    then k ← k + 1
11    else goto 3
12  if k ≤ 1388
13    then goto 2
```

مقدار a_{1388} در انتهای این الگوریتم چند است؟ برای ادعای خود دلیل بیاورید.[۱۴]

۱۵.۱-۲ در یک شبکه‌ی ابرمکعب[۱۵]، 2^k پردازنده با شماره‌های ۱ تا 2^k وجود دارند. هریک از این پردازنده‌ها با یک کُد یک‌تا، که یک دنباله‌ی k تایی از رقم‌های ۰ و ۱ است، مشخص می‌شود. دو پردازنده به صورت مستقیم به هم متصل هستند اگر و تنها اگر کد مربوط به آن‌ها دقیقاً در یک رقم با هم تفاوت داشته باشند. برای مثال اگر $k = 4$ باشد، پردازنده‌ای که دارای کد ۰۱۰۰ است مستقیماً به پردازنده‌هایی با کدهای ۱۱۰۰، ۰۰۰۰، ۰۱۱۰ و ۰۱۰۱ متصل است.

در ابتدای کار، هر یک از پردازنده‌ها دارای یک پیام است. پیامی که در ابتدا در پردازنده‌ی شماره‌ی i ($1 \le i \le 2^k$) است، باید در نهایت به کامپیوتر p_i ($1 \le p_i \le 2^k$) برسد. فرض کنید که در این p_iها عدد تکراری وجود ندارد، یعنی در نهایت هرکدام از پردازنده‌ها باید یک پیام دریافت کنند.

در هر مرحله، هر پردازنده می‌تواند پیامی را که دارد به یکی از پردازنده‌هایی که مستقیماً به آن متصل است بدهد؛ به شرطی که هر پردازنده پس از پایان آن مرحله بیش از یک پیام نداشته باشد. (یعنی اگر در یک مرحله، پردازنده‌ی a پیام خود را به پردازنده‌ی b بدهد، پردازنده‌ی b هم باید پیامی که قبل از این مرحله داشته است، در همین مرحله به یک پردازنده‌ی دیگر بدهد. هم‌چنین هیچ پردازنده‌ی

[۱۴] مسئله‌ی ۳، مرحله‌ی دوم هفتمین المپیاد کامپیوتر ایران

[۱۵] hypercube

دیگری غیر از a نمی‌تواند پیام خود را در همین مرحله به b بدهد.)
ثابت کنید در حداکثر $۲^k - ۱$ مرحله، پردازنده‌ها می‌توانند همه‌ی پیام‌ها را با توجه به شرط فوق به مقصدشان برسانند.[۱۶]

* **۲-۱.۱۶** n چراغ با شماره‌های ۱ تا n بر روی یک خط مستقیم قرار دارند که تعدادی از آن‌ها خاموش و بقیه روشن هستند. A و B این بازی را با هم انجام می‌دهند. از ابتدا و در تمام مراحل بازی، چشم B بسته است و وضعیت لامپ‌ها را نمی‌داند. در هر مرحله از بازی، B مجموعه‌ای از اعداد ۱ تا n را انتخاب می‌کند و به A می‌گوید. A لامپ‌هایی که شماره‌ی آن‌ها در آن مجموعه است، تغییر می‌دهد؛ یعنی اگر لامپ خاموش بود، آن را روشن و اگر روشن بود آن را خاموش می‌کند. مثلاً اگر ۳ لامپ داشته باشیم و لامپ‌های ۱ و ۳ خاموش باشند و لامپ ۲ روشن، و B مجموعه‌ی $\{۱,۲\}$ را انتخاب کند، در مرحله‌ی بعد لامپ ۱ روشن و لامپ‌های ۲ و ۳ خاموش خواهند شد. در هر مرحله‌ای که تمام لامپ‌ها خاموش شوند، بازی به نفع B تمام می‌شود. مثلاً اگر $n = ۲$ و B به ترتیب مجموعه‌های $\{۱,۲\}$، $\{۱\}$ و $\{۱,۲\}$، را انتخاب کند، به هر ترتیب B برنده‌ی بازی خواهد شد.
ثابت کنید برای هر n، B می‌تواند طوری بازی کند که ببرد. یعنی دنباله‌ای از زیرمجموعه‌های $\{۱,۲,\ldots,n\}$ را انتخاب کند که برای هر وضعیت اولیه‌ی دل‌خواه از چراغ‌ها، در حین انجام عمل به جایی برسیم که همه‌ی چراغ‌ها خاموش باشند.[۱۷]

۲-۲ مبانی روش‌های شمارش

روش‌های شمارش خود یک علم است که به‌طور مفصل در ریاضی کاربردی، ترکیبیات و نظریه‌ی احتمالات مطالعه می‌شود. این روش‌ها یکی از پایه‌های مهم در طراحی و تحلیل الگوریتم‌هاست، چرا که بررسی و شمارش حالت‌های مختلف یک مسئله و این‌که آیا راه‌حل مورد نظر همه‌ی این حالت‌ها را در بر می‌گیرد، نقش اساسی در طراحی روش حل آن مسئله دارد. در تحلیل یک الگوریتم نیز در بسیاری از موارد باید تعداد تکرار بخش‌های اصلی آن الگوریتم را بشماریم.

در این بخش مبانی روش‌های شمارش را به‌اختصار و با ذکر چند مثال بیان می‌کنیم. در ابتدا نمونه‌هایی از چند مسئله را ببینید:

۱. چه تعداد گراف با n رأس وجود دارد؟

[۱۶] مسئله‌ی ۷، مرحله‌ی دوم هفتمین المپیاد کامپیوتر ایران
[۱۷] مسئله‌ی ۲، مرحله‌ی دوم دهمین المپیاد کامپیوتر ایران

۲. چه تعداد «تطابق[۱۸]» بین n پسر و n دختر وجود دارد؟

۳. هر الگوریتم مرتب‌ساز دست‌کم چه تعداد مقایسه نیاز دارد؟

۴. چه تعداد توزین برای تشخیص یک سکه‌ی تقلبی از بین ۱۳ سکه ضروری است؟

۵. نام یک فایل در یک کامپیوتر، رشته‌ای است به‌طول ۱ تا ۱۰ از نویسه‌های حرفی و ارقام که اولین آن باید یک حرف باشد. حرف‌های بزرگ و کوچک برای این کامپیوتر مشابه تلقی می‌شود. چه‌تعداد فایل درست در این کامپیوتر می‌توان داشت؟

۶. در ضرب دو ماتریس $n \times n$ از اعداد صحیح، دقیقاً چند عمل ضرب و چند عمل جمع انجام می‌شود؟

۷. یک فروشنده کالای خود را در مراکز ۳۰ استان کشور می‌فروشد. او تصمیم می‌گیرد که از یک مرکز استان به مرکز استان بعدی برود و پس از رفتن به همه‌ی استان‌ها به جای اول خود برگردد. فرض کنید که می‌توان از هر مرکز استان مستقیماً به هر مرکز دیگر سفر کرد. او به‌چند طریق می‌تواند این کار را انجام دهد؟

در شمارش دو اصل مهم وجود دارد: یکی اصل جمع و دیگری اصل ضرب.

اصل جمع

اگر E مجموعه‌ی حالات یک رخ‌داد به‌نام e باشد و برای مجموعه‌های متمایز E_1، E_2 تا E_k داشته باشیم:

$$E = E_1 \cup E_2 \cup \cdots E_k$$

یعنی e را بتوان به روش‌های e_1، e_2، ...، یا e_k انجام داد و E_i مجموعه‌ی حالات e_i باشد، در آن‌صورت تعداد حالات رخ‌داد e، یا $|E|$ برابر است با

$$|E| = |E_1| + \cdots + |E_k|. \qquad (۲-۲)$$

[۱۸] matching

اصل ضرب

اگر $E_۱$، $E_۲$، تا E_k به‌ترتیب مجموعه‌ی حالات انجام رخ‌دادهای مستقل از هم $e_۱$، $e_۲$ تا e_k باشند، ضرب کارتزین

$$E = E_۱ \times E_۲ \times \cdots \times E_k = \{(a_۱, a_۲, \ldots, a_k) | a_i \in E_i\}$$

مجموعه‌ی حالات رخ‌داد e را نشان می‌دهد که از k رخ‌دادِ پشت سرهم $e_۱$، $e_۲$ تا e_k تشکیل می‌شود. در آن صورت،

$$|E| = |E_۱| \times |E_۲| \times \cdots \times |E_k|. \qquad (۳-۲)$$

حل تعدادی از مسئله‌های نمونه

سه مسئله‌ی آخر این بخش را با استفاده از اصول گفته‌شده حل می‌کنیم.

۵. اسم یک فایل بین ۱ تا ۱۰ حرف دارد. حرف اول ۲۶ حالت و بقیه‌ی حرف‌ها هر کدام ۳۶ حالت دارند. بنابراین با توجه به اصل جمع و ضرب، تعداد کل حالات برابر است با

$$T = \sum_{k=۱}^{۱۰} ۲۶ \times ۳۶^{k-۱} = ۲۶(۳۶^{۱۰} - ۱)/۳۵ \approx ۲/۷ \times ۱۰^{۱۵}.$$

۶. اگر ابعاد ماتریس‌های A و B هر کدام $n \times n$ و $C = A \times B$ ماتریس حاصل باشد، می‌دانیم که

$$c_{ij} = \sum_{k=۱}^{n} a_{ik} \times b_{kj},$$

که در آن c_{ij} درایه‌ی سطر i و ستون j از ماتریس C و نیز a_{ik} و b_{kj} درایه‌های متناظر ماتریس‌های A و B هستند. c_{ij} از ضرب داخلی سطر iام A و ستون jام B به‌دست می‌آید. بنابراین برای هر درایه‌ی C دقیقاً n عدد ضرب و $n-۱$ عدد جمع انجام می‌شود. چون تعداد این درایه‌ها $n^۲$ است، در مجموع $n^۳$ عمل ضرب و $n^۲(n-۱)$ عمل جمع انجام می‌شود.

۷. در ابتدا فروشنده می‌تواند هر یک از ۳۰ استان را به‌عنوان اولین استان انتخاب کند. بنابراین برای اولین استان ۳۰ انتخاب دارد. با این فرض، تعداد استان‌های دوم او ۲۹ عدد، تعداد استان‌های سوم ۲۸ تا، و همین‌طور تا این‌که به یک حالت برای استان آخر برسیم. بنابراین تعداد حالات مطابق فرمول زیر خواهد شد:

$$۳۰ \times ۲۹ \times ۲۸ \times \cdots \times ۲ \times ۱ = ۳۰!$$

۲-۲-۱ ترتیب و ترکیب

ترتیب[۱۹] و ترکیب[۲۰] پایه‌های اولیه‌ی «ریاضیات ترکیبیاتی[۲۱]» است که به شمارش تعداد حالات یک یا چند شیء می‌پردازد.

ترتیب

ترتیب k از n که با P_k^n نشان داده می‌شود، تعداد حالات قرار گرفتن k شیء از مجموعه‌ی دوبه‌دو متمایز n عضوی در یک ردیف است. یعنی P_k^n تعداد جای‌گشت‌های k تایی از n شیء متمایز است. در یک دنباله‌ی k تایی، عنصر اول n حالت می‌تواند داشته باشد، عنصر دوم $n-۱$ حالت، ...، و عنصر k ام $n-k+۱$ حالت. پس طبق اصل ضرب، داریم

$$\begin{aligned} P_k^n &= n(n-۱)(n-۲)\cdots(n-k+۱) \\ &\times \frac{(n-k)(n-k-۱)\times ۲ \times ۱}{(n-k)(n-k-۱)\times ۲ \times ۱} = \frac{n!}{(n-k)!} \end{aligned} \qquad (۴-۲)$$

ترتیب n از n همان $n!$ است.

ترکیب

تعداد زیرمجموعه‌های k عضوی از یک مجموعه‌ی n عضوی با عناصر دوبه‌دو متمایز را ترکیب k از n می‌گوییم و با نماد $\binom{n}{k}$ یا C_k^n نشان می‌دهیم.

---

[۱۹] permutation
[۲۰] combination
[۲۱] combinatorial mathematics

مقدار $\binom{n}{k}$ را می‌توان به‌صورت زیر به‌دست آورد. اگر ترتیب در انتخاب عناصر مهم باشد، جواب برابر با P_k^n است. ولی چون در ترکیب ترتیب عناصر مهم نیست، هر یک از این حالت‌ها $k!$ بار شمرده شده است. بنابراین

$$\binom{n}{k} = \frac{P_k^n}{k!} = \frac{n!}{k!(n-k)!}.$$

لم ۲-۱ به‌ازای هر $1 \leq k < n$ ثابت کنید

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \qquad (۵-۲)$$

اثبات: فرض کنید $A = \{1, 2, \ldots, n\}$. طبق تعریف، $\binom{n}{k}$ برابر با تعداد زیرمجموعه‌های k عضوی A است. این زیرمجموعه‌ها را به دو بخش تقسیم می‌کنیم:

الف) زیرمجموعه‌هایی که شامل n هستند. تعداد این زیرمجموعه‌ها برابر است با تعداد زیرمجموعه‌های $k-1$ عضوی مجموعه‌ی $A' = A - \{n\}$. طبق تعریف تعداد این زیرمجموعه‌ها برابر است با $\binom{n-1}{k-1}$.

ب) زیرمجموعه‌هایی که شامل n نیستند. تعداد این زیرمجموعه‌ها برابر است با تعداد زیرمجموعه‌های k عضوی مجموعه‌ی $A' = A - \{n\}$. طبق تعریف تعداد این زیرمجموعه‌ها برابر است با $\binom{n-1}{k}$.

طبق اصل جمع، به این صورت لم ثابت می‌شود. □

مثال ۲-۷ n کتاب ریاضی و m کتاب فیزیک را به چند طریق می‌توان در یک قفسه چید به‌طوری که:

الف) کتاب‌های فیزیک کنار هم باشند. (یعنی، بین هیچ دو کتاب فیزیکی، کتاب ریاضی نباشد.)

ب) هیچ دو کتاب فیزیکی کنار هم نباشند و کتاب اول و آخر ریاضی باشد.

حل:

الف) همه‌ی کتاب‌های فیزیک را یک کتاب به‌نام P در نظر می‌گیریم. n کتاب ریاضی و کتاب P را به $(n+1)!$ حالت، می‌توانیم بچینیم. کتاب P را هم به $m!$ حالت می‌توان چید. پس، مطابق اصل ضرب، جواب مسئله برابر $(n+1)! \times m!$ است.

ب) ابتدا کتاب‌های ریاضی را در قفسه می‌چینیم، سپس کتاب‌های فیزیک را بین آن‌ها قرار می‌دهیم. به $n!$ حالت می‌توانیم کتاب‌های ریاضی را بچینیم. هر کتاب فیزیک باید بین دو کتاب ریاضی باشد، بنابراین، برای چیدن آن‌ها $n-1$ جای خالی داریم. پس به P_m^{n-1} طریق می‌توانیم کتاب‌های فیزیک را بچینیم. پس طبق اصل ضرب جواب نهایی برابر با $n! \times P_m^{n-1}$ است.

۲-۲-۲ ترتیب دوری و حلقوی

مثال ۲-۸ به چند طریق می‌توان k نفر را دور یک میز دایره‌ای نشاند؟

حل: روش اول: جای یک فرد مشخص (مثلاً A)، را ثابت می‌کنیم. چون می‌توان میز را دوران داد، پس جای A یک حالت دارد. $k-1$ نفر بقیه را می‌خواهیم در $k-1$ جای باقی‌مانده بنشانیم. این کار را به $(k-1)!$ راه می‌توان انجام داد.

روش دوم: اگر شرط دوران را در نظر نگیریم، به $k!$ حالت می‌توان این k نفر را دور میز نشاند. با وجود دوران، هر k حالت معادل یک حالت است. پس تعداد حالت‌های مسئله برابر است با $\frac{k!}{k} = (k-1)!$.

مثال ۲-۹ به چند طریق می‌توان k نفر را دور یک میز دایره‌ای با $n \geq k$ صندلی نشاند؟

حل: روش اول: $k-1$ نفر را A_1، A_2 تا A_n و صندلی‌ها را B_1، B_2 تا B_n می‌نامیم. می‌توان فرض کرد که A_1 روی صندلی B_1 نشسته است (و گرنه میز را می‌چرخانیم). برای نشستن A_2، $n-1$ صندلی خالی، برای نشستن A_3، $n-2$ صندلی خالی، ... و برای نشستن A_{k-1}، $n-k+1$ صندلی خالی داریم. پس طبق اصل ضرب جواب برابر است با $(n-1) \times (n-2) \times \cdots \times (n-k+1) = P_{k-1}^{n-1} = \frac{P_k^n}{n}$.

روش دوم: اگر دوران را در نظر نگیریم به P_k^n طریق می‌توان این k نفر را دور میز نشاند. با وجود دوران، هر حالت را n بار شمرده‌ایم. بنابراین جواب برابر است با $\frac{P_k^n}{n}$.

مثال ۲-۱۰ به چند طریق می‌توان n دانش‌جوی S_1 تا S_n و m استاد P_1 تا P_m را دور یک میز دایره‌ای نشاند، به‌طوری که،

الف) هیچ محدودیت دیگری وجود نداشته باشد.

ب) استاد P_1 و دانش‌جوی S_1 کنار هم ننشینند.

پ) هیچ دو استاد کنار هم ننشینند.

ت) استادان کنار یکدیگر نشسته باشند.

حل:

الف) $m+n$ نفر داریم؛ جواب برابر است با $(m+n-1)!$.

ب) روش اول: ابتدا صندلی دانش‌جوی S_1 را ثابت فرض می‌کنیم. به‌علت دوران، برای نشستن S_1 فقط یک حالت وجود دارد. چون یک صندلی را S_1 اشغال کرده است و P_1 نمی‌تواند در هیچ‌یک از دو صندلی کناری او بنشیند، بنابراین به $m+n-3$ طریق می‌توان P_1 را نشاند. حال $m+n-2$ صندلی و $m+n-2$ نفر داریم که طبق اصل ضرب، این افراد را می‌توان به $(m+n-2)!$ طریق دور میز نشاند. پس جواب برابر است با $(m+n-3)(m+n-2)!$.

روش دوم: در این روش جواب را از تفریق تعداد حالات نامطلوب از تعداد کل حالات به‌دست می‌آوریم. این روش حالت خاصی از «اصل شمول و عدم شمول[۲۲]» است که در بخش ۲-۲-۵ گفته خواهد شد. اگر شرط گفته‌شده وجود نداشت به $(m+n-1)!$ طریق می‌توانستیم همه را بچینیم. حال تعداد حالات نامطلوب را می‌شماریم.

اگر S_1 و P_1 را یک فرد به‌نام A در نظر بگیریم، $n+m-1$ نفر داریم. این افراد را به $(n+m-2)!$ حالت می‌توانیم دور میز بنشانیم. ولی خود A دو حالت دارد (P_1 سمت راست S_1 باشد یا S_1 سمت چپ P_1). بنابراین در $2(m+n-2)!$ حالت، S_1 و P_1 مجاور یک‌دیگرند. پس تعداد حالات مطلوب برابر است با:

$$(m+n-1)! - 2(m+n-2)! = (m+n-3)(m+n-2)!$$

پ) اگر $m > n$ باشد، جواب صفر است. فرض کنید $m \le n$. اول دانش‌جویان را به $(n-1)!$ طریق دور میز می‌نشانیم. حال بین آن‌ها n جا وجود دارد که باید استادها

[۲۲] inclusion-exclusion

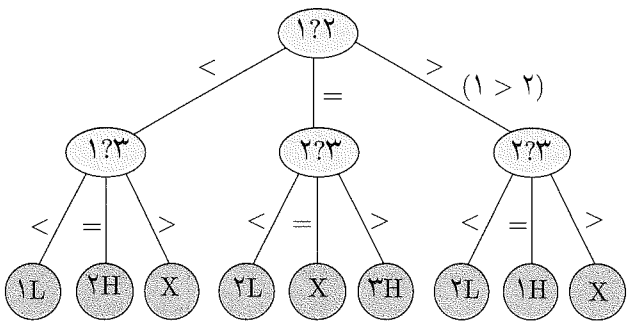
در m جا از آن‌ها بنشینند. پس استادها هم به P_m^n طریق می‌توانند بنشینند. پس طبق اصل ضرب، جواب مسئله برابر است با $P_m^n(n-1)!$.

ت) تمام استادها را یک نفر به‌نام P، در نظر می‌گیریم. n دانش‌جو و M، مثل $n+1$ نفر هستند. آن‌ها را به $n!$ حالت می‌توانیم دور میز بنشانیم. ولی خود استادها می‌توانند به $m!$ حالت کنار هم بنشینند. بنابراین جواب برابر است با $n!m!$. دقت کنید که اگر قرار بود به‌جای استادان، دانش‌جویان کنار هم بنشینند، همین جواب به‌دست می‌آمد.

مثال ۲-۱۱ کران پایین تعداد توزین‌های لازم با یک ترازوی دو کفه‌ای و بدون وزنه را به‌دست آورید تا بتوان تنها سکه‌ی تقلبی از میان ۱۳ سکه‌ی داده‌شده را پیدا کرد. توجه کنید که از قبل نمی‌دانیم که سکه‌ی تقلبی سبک‌تر یا سنگین‌تر از بقیه است. آیا می‌توان با این تعداد توزین، مسئله را هم حل کرد؟ با هر بار توزین فقط می‌توانیم وزن دو دسته از سکه‌هایی را که در دو کفه قرار داده‌ایم با هم مقایسه کنیم.

حل: ابتدا مسئله را برای ۳ سکه حل می‌کنیم. اگر بدانیم که یکی از ۳ سکه‌ی داده‌شده تقلبی است، و ندانیم که سکه‌ی تقلبی سبک‌تر است یا سنگین‌تر، با ۲ بار توزین می‌توان مسئله را حل کرد. یک راه‌حل در درخت شکل ۲-۸ نمایش داده شده است. در این شکل، مثلاً ۲H یعنی سکه‌ی دوم تقلبی و سنگین‌تر، ۱L یعنی سکه‌ی اول تقلبی و سبک‌تر از دو سکه‌ی دیگر و X یعنی این حالت امکان ندارد.

برای مسئله‌ای با ۱۳ سکه، نشان می‌دهیم که نمی‌توان با کم‌تر از ۴ بار توزین، سکه‌ی تقلبی را پیدا کرد. سپس راه‌حلی با این تعداد توزین ارائه می‌کنیم.



شکل ۲-۸ درخت توزین برای تشخیص سکه‌ی تقلبی بین ۳ سکه. در گام اول سکه‌های ۱ و ۲ را با یک توزین با هم مقایسه می‌کنیم، که با ۱?۲ نشان داده شده است. بسته به نتیجه‌ی این مقایسه، یکی از سه توزین نشان داده شده را انجام می‌دهیم.

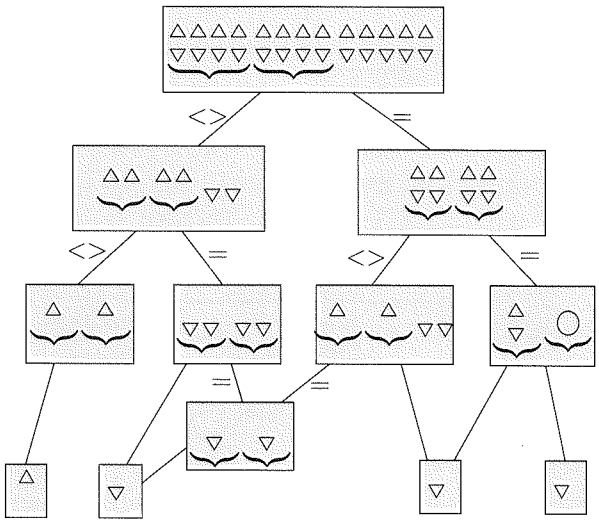
چون هر کدام از ۱۳ سکه می‌تواند تقلبی و سبک‌تر یا سنگین‌تر باشد، وضعیت سکه‌ها یکی از این $۲۶ = ۱۳ \times ۲$ حالت است. مطابق شکل ۲-۸، با k بار توزین می‌توان حداکثر ۳^k حالت را مشخص کرد. یعنی ۲ توزین حداکثر ۹ حالت را تمیز می‌دهد. هر توزین یکی از ۳ حالت بزرگ‌تر، کوچک‌تر و مساوی را نمایان می‌سازد. قبل از توزین اول می‌توانیم ۲۶ حالت ممکن را به ۳ دسته تقسیم کنیم: هر دسته برای نتیجه‌ی توزین اول (بزرگ‌تر، کوچک‌تر یا مساوی). ممکن است بر حسب روش توزین، تعداد اعضای این ۳ دسته مثلاً برابر ۱۴، ۶ و ۶ یا ۱۰، ۸ و ۸ باشد. بعد از توزین اول در می‌یابیم که وضعیت سکه‌ها در کدام‌یک از این ۳ دسته قرار دارد، سپس با یک توزین دیگر وضعیت را به ۳ دسته‌ی کوچک‌تر تقسیم می‌کنیم. برای این‌که بتوان مجموعاً با ۳ بار توزین، سکه‌ی تقلبی را پیدا کرد، لازم است در مرحله‌ی اول دسته‌ها حداکثر ۹ حالت داشته باشند. یعنی تنها تقسیم‌بندی قابل قبول ۹، ۹ و ۸ است. این تقسیم بندی نیز مقدور نیست زیرا در مرحله‌ی اول باید k سکه در یک کفه و k سکه در کفه‌ی دیگر قرار دهیم، که در نتیجه تعداد حالت‌های نهایی در هر دسته برابر $۲k$، $۲k$، و $۲ \times (۲k - ۱۳)$ می‌شود یعنی تعداد اعضای این ۳ دسته نمی‌تواند فرد باشد. بنابراین دسته‌بندی ۹، ۹ و ۸ مقدور نیست، یعنی دست‌کم به ۴ بار توزین احتیاج داریم.

حال نشان می‌دهیم که با ۴ بار توزین می‌توانیم این مسئله را حل کنیم. برای نشان دادن وضعیت سکه‌ها از یک نمادگذاری استفاده می‌کنیم. برای مثال $\overset{\triangle\triangle}{\triangledown\triangledown}$ یعنی از ۲ سکه یا سکه‌ی اول تقلبی است (سنگین‌تر یا سبک‌تر) یا سکه‌ی دوم (سنگین‌تر یا سبک‌تر). و $\triangledown\triangledown^{\triangle\triangle}$ هم یعنی می‌دانیم که سکه‌ی تقلبی یکی از ۴ سکه‌ی مورد نظر و سنگین‌تر یا سبک‌تر است: یا سکه‌ی اول و سنگین‌تر، یا سکه‌ی دوم و سنگین‌تر، یا سکه‌ی سوم و سبک‌تر، یا سکه‌ی چهارم و سبک‌تر.

به این ترتیب حالت اولیه‌ی مسئله‌ی ۱۳ سکه به‌صورت $\overset{\triangle\triangle\triangle\triangle\triangle\triangle\triangle\triangle\triangle\triangle\triangle\triangle\triangle}{\triangledown\triangledown\triangledown\triangledown\triangledown\triangledown\triangledown\triangledown\triangledown\triangledown\triangledown\triangledown\triangledown}$ است و جواب نهایی وقتی است که به حالت $^{\triangle}$ یا $_{\triangledown}$ با شماره‌ی سکه می‌رسیم.

حالت‌های $\overset{\triangle}{\triangledown}$، $^{\triangle\triangle}$، $_{\triangledown}{}^{\triangle}$ و $^{\triangle\triangle\triangle}$ را می‌توان با یک بار توزین حل کرد. حالت $\triangledown\triangledown$ مشابه $^{\triangle\triangle}$ است، در این‌جا از نوشتن حالت‌های مشابه صرف‌نظر می‌کنیم. برای $^{\triangle\triangle\triangle}$ سکه‌ی اول را با سکه‌ی دوم مقایسه می‌کنیم، اگر سکه‌ی اول بزرگ‌تر، کوچک‌تر یا مساوی بود، به ترتیب سکه‌ی تقلبی سکه‌ی اول، دوم یا سوم خواهد بود. در حالت $\overset{\triangle}{\triangledown}$ و $_{\triangledown}{}^{\triangle}$ نیز سکه‌ی اول را با یک سکه‌ی سالم دیگر مقایسه می‌کنیم.

حالت‌های $^{\triangle\triangle\triangle\triangle}$ و $\triangledown\triangledown^{\triangle\triangle}$ با ۲ بار توزین و حالت $\triangledown\triangledown\triangledown\triangledown_{\triangledown}{}^{\triangle\triangle\triangle\triangle}$ با ۳ بار توزین به روش مشابه حل می‌شوند. حالت $\overset{\triangle\triangle\triangle\triangle\triangle}{\triangledown\triangledown\triangledown\triangledown\triangledown}$ با ۳ بار توزین به این صورت حل می‌شود: ۲ سکه را با ۲ سکه‌ی دیگر مقایسه می‌کنیم. در صورت نتیجه‌ی بزرگ‌تر، کوچک‌تر یا مساوی



شکل ۲-۹ روش بهینه‌ی یافتن سکه‌ی تقلبی و تشخیص سبک‌تر یا سنگین‌تر بودن آن سکه از میان ۱۳ سکه که تنها یکی از آن‌ها تقلبی است.

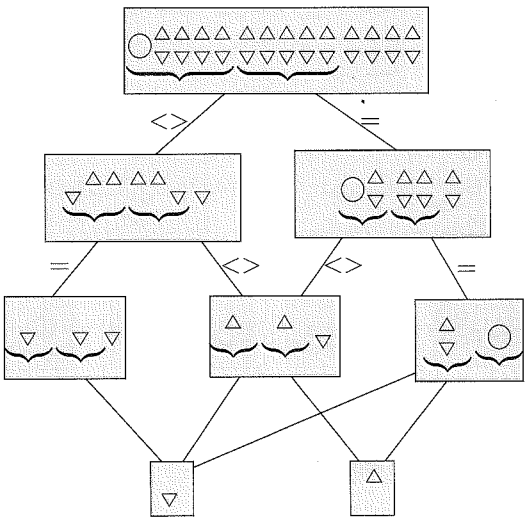
به ترتیب به حالت‌های $\overset{\triangle\triangle}{\triangledown\triangledown}$، $\overset{\triangle\triangle}{\triangledown\triangledown}$ و $\overset{\triangle}{\triangledown}$ می‌رسیم که هر کدام با یک یا دو بار توزین قابل حل است.

در نهایت برای ۱۳ سکه، ۴ سکه را با ۴ سکه‌ی دیگر مقایسه می‌کنیم. در حالت بزرگ‌تر، کوچک‌تر یا مساوی به‌ترتیب به حالت‌های $\overset{\triangle\triangle\triangle\triangle}{\triangledown\triangledown\triangledown\triangledown}$، $\overset{\triangle\triangle\triangle\triangle}{\triangledown\triangledown\triangledown\triangledown}$ یا $\overset{\triangle\triangle\triangle\triangle}{\triangledown\triangledown\triangledown\triangledown}$ می‌رسیم که همگی با ۳ بار توزین قابل حل هستند. درخت این توزین‌ها در شکل ۲-۹ نمایش داده شده است.

جالب است بدانید اگر در اول کار یک سکه‌ی سالم اضافی در اختیار داشتیم، در مرحله‌ی اول دسته‌بندی ۹، ۹ و ۸ مقدور بود و مسئله با ۳ بار توزین قابل حل بود. شکل ۲-۱۰ درخت توزین این مسئله را نشان می‌دهد.

۲-۲-۳ تناظر یک‌به‌یک

تناظر یک ابزار مهم در شمارش حالات است. با این روش یک مسئله‌ی شمارشی را با مسئله‌ی دیگری که راه‌حل آن‌را می‌دانیم متناظر می‌کنیم.



شکل ۲-۱۰ حل مسئله‌ی یافتن سکه‌ی تقلبی از میان ۱۳ سکه با ۲ بار توزین اگر یک سکه‌ی سالم اضافی (◯) داشته باشیم.

برای شروع به تعریف‌های زیر توجه کنید:

تعریف ۲-۱ تابع $f : A \to B$ را یک‌به‌یک[۲۳] می‌نامیم، اگر و تنها اگر به‌ازای هر دو عضو $\{x_1, x_2\} \subseteq A$، داشته باشیم: $f(x_1) \neq f(x_2)$.

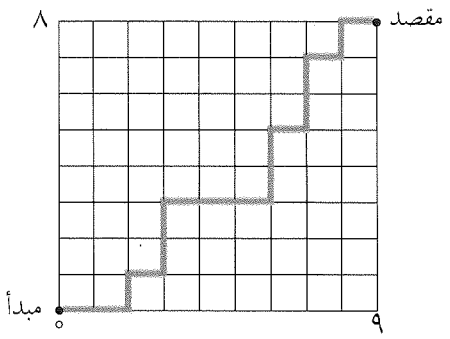
تعریف ۲-۲ تابع $f : A \to B$ را پوشا[۲۴] می‌نامیم، اگر و تنها اگر به‌ازای هر $y \in B$، یک $x \in A$ باشد که $f(x) = y$.

در نتیجه اگر A و B مجموعه‌های متناهی باشند و تابع $f : A \to B$،

- یک‌به‌یک باشد، داریم $|A| \leq |B|$.
- پوشا باشد، داریم $|A| \geq |B|$.
- یک‌به‌یک و پوشا باشد، داریم $|A| = |B|$.

---

[۲۳] one-to-one
[۲۴] on-to



شکل ۲-۱۱ تعداد حالات رسیدن از مبدأ به مقصد.

مثال‌هایی از تناظر یک‌به‌یک

مثال ۲-۱۲ به چند طریق می‌توان در شکل ۲-۱۱ (با عرض ۹ و ارتفاع ۸) از مبدأ به مقصد رفت؟

حل: هر مسیر را می‌توانیم با رشته‌ای ۱۷ حرفی با ۹ نویسه‌ی R و ۸ نویسه‌ی U متناظر کنیم که در آن R و U به‌ترتیب نمادهای حرکت به راست و بالا است. مثلاً مسیر نشان داده‌شده در شکل، متناظر با RRURUURRRUURUURUR است. هم‌چنین هر رشته با شرایط فوق، متناظر با یک مسیر است. بنابراین تعداد این مسیرها برابر تعداد چنین رشته‌هایی و برابر $\binom{۱۷}{۸}$ است.

مثال ۲-۱۳ اگر S یک مجموعه‌ی n عضوی باشد، تعداد اعضای $P(S)$ چند تاست؟

حل: یک تناظر یک‌به‌یک بین زیرمجموعه‌های یک مجموعه‌ی n عضوی و دنباله‌های به‌طول n در مبنای ۲ برقرار می‌کنیم؛ ۱ برای عضوی که در زیرمجموعه هست و ۰ برای عضوی که در زیرمجموعه نیست. این تناظر یک‌به‌یک و پوشا است، بنابراین $|P(S)| = |B|$. می‌دانیم $|B| = ۲^n$.

مثال ۲-۱۴ فرض کنید $X = \{۱, ۲, \cdots, n\}$. تعداد زیرمجموعه‌های k عضوی X که شامل هیچ دو عضو متوالی نباشند چند تاست؟

حل: یک تناظر یک‌به‌یک بین زیرمجموعه‌های k تایی X که شامل هیچ دو عضو متوالی نباشد (مسئله‌ی C) و تعداد زیرمجموعه‌های k عضوی مجموعه‌ی $Y = \{۱, ۲, \cdots, n-k+۱\}$ (مسئله‌ی D) برقرار می‌کنیم.

اگر $A = \{a_1, a_2, \cdots, a_k\}$ یک جواب باشد، می‌توان فرض کرد $a_1 < a_2 < \cdots < a_k$، ولی چون هیچ دو عنصری در A نباید متوالی باشند، بنابراین $a_1 < a_2 - 1$، $a_2 < a_3 - 1$، ... و $a_{k-1} < a_k - 1$. در نتیجه

$$1 \le a_1 < a_2 - 1 < a_3 - 2 < \cdots < a_k - k + 1 \le n - k + 1.$$

حال اگر فرض کنیم

$$b_1 = a_1,\ b_2 = a_2 - 1,\ \ldots,\ b_k = a_k - k + 1,$$

هر جواب $\{a_1, a_2, \cdots, a_k\}$ مسئله‌ی C با یک جواب $\{b_1, b_2, \ldots, b_k\}$ مسئله‌ی D متناظر است. هم‌چنین هر حالت از مسئله‌ی D با یک حالت از مسئله‌ی C متناظر است. بنابراین یک تناظر یک‌به‌یک بین این دو مسئله برقرار است. می‌دانیم که جواب مسئله‌ی D برابر با $\binom{n-k+1}{k}$ است، بنابراین جواب این مسئله هم همین رابطه است.

۲-۲-۴ مسئله‌های توپ و ظرف

مسئله‌ی «توپ و ظرف[۲۵]» به این صورت است که با فرض داشتن u عدد ظرف (چه مشابه، چه متمایز) و b عدد توپ (چه مشابه، چه متمایز)، به چند طریق می‌توان این توپ‌ها را در این ظرف‌ها قرار داد؟ به مثال‌های زیر توجه کنید:

مثال ۲-۱۵ در مسئله‌ی کلاسیکِ برج هانوی، n سکه با اندازه‌های متفاوت در ۳ میله به‌ترتیب اندازه‌ی نزولی سکه‌ها (از پایین به بالا) قرار دارند. اگر ترتیب سکه‌ها در میله‌ها مهم نباشد، این مسئله چند حالت دارد؟

مسئله‌ی برج هانوی در بخش ۳-۴-۲ به‌طور دقیق‌تر بیان می‌شود. تعداد حالات خواسته‌شده برابر تعداد حالات مسئله‌ی توپ و ظرف است که در آن n توپ متمایز قرار است در ۳ ظرف متمایز قرار گیرند.

مثال ۲-۱۶ معادله‌ی $x + y + z + w = 73$ چند جواب صحیح نامنفی دارد؟

این مسئله همان مسئله‌ی توپ و ظرف است با ۷۳ توپ مشابه که در ۴ ظرف متمایز با نام‌های x، y، z و w قرار می‌گیرند.

[۲۵] balls and urns

مثال ۲-۱۷ به چند حالت می‌توان $۴n$ نفر مشخص را به $۲n$ تیم برای یک بازی دوره‌ای تقسیم کرد؟

این هم مسئله‌ی توپ و ظرف است که در آن $۴n$ توپ متمایز در $۲n$ ظرف مشابه قرار می‌گیرند، چون افراد متمایزند ولی تیم‌ها از هم متمایز نیستند.

مثال ۲-۱۸ افراز یک عدد صحیح و مثبت n، هر مجموعه از اعداد صحیح و مثبت است که مجموع عناصرش n شود. مثلاً $\{۵,۱,۱\}$ و $\{۳,۴\}$ دو افراز برای $n = ۷$ هستند. تعداد افرازهای n چند تاست؟

این همان مسئله‌ی توپ و ظرف است که در آن n توپ مشابه در n ظرف مشابه قرار می‌گیرند.

راه‌حل‌های ۴ گونه از مسئله‌ی توپ و ظرف

۱. **تعداد حالات قرار دادن b توپ متمایز در u ظرف متمایز:** هر توپ می‌تواند در یکی از u ظرف متمایز از هم قرار گیرد. چون توپ‌ها متمایزند، پس تعداد کل حالت‌های این مسئله u^b است.

۲. **تعداد حالات قرار دادن b توپ مشابه در u ظرف متمایز:** فرض کنید که در ظرف u_i به تعداد b_i توپ قرار گرفته باشد $(۰ \le b_i \le b)$ به‌طوری که $\sum_{i=۱}^{u} b_i = b$. این حالت از مسئله را به‌صورت زیر می‌توان نمایش داد:

$$\underbrace{\bullet\bullet\cdots\bullet\bullet}_{b_۱}|\underbrace{\bullet\bullet\cdots\bullet\bullet}_{b_۲}|\ldots|\underbrace{\bullet\bullet\cdots\bullet\bullet}_{b_u}.$$

بنابراین هر حالت از مسئله یک رشته از b عدد $\bullet$ و $u-۱$ نویسه‌ی $|$ است و این تناظر یک‌به‌یک و پوشاست. تعداد کل حالات برابر انتخاب $u-۱$ مکان برای نویسه‌ی $|$ است از طول $u+b-۱$ نویسه‌ای رشته، که برابر است با $\binom{b+u-۱}{u-۱} = \binom{b+u-۱}{b}$.

۳. **تعداد حالات قرار دادن b توپ متمایز در u ظرف مشابه:** ممکن است به نظر برسد که جواب $\frac{u^b}{u!}$ است. این جواب غلط است؛ چون حالتی را می‌توان در نظر گرفت که ۲ ظرف خالی باشند. در فرمول u^b این حالت و حالتی که فقط جای این دو ظرف خالی جابه‌جا شده باشد، یکی در نظر گرفته می‌شود. در حالی که با تقسیم بر $u!$، این دو حالت متمایز از هم در نظر گرفته می‌شوند.

برای حل این حالت از مسئله، این مسئله‌ها را در نظر می‌گیریم:

- $T(b,u)$: تعداد حالات قرار دادن b توپ متمایز در u ظرف متمایز اگر هیچ ظرفی خالی نباشد.
- $S(b,u)$: تعداد حالات قرار دادن b توپ متمایز در u ظرف مشابه اگر هیچ ظرفی خالی نباشد.
- $B(b,u)$: تعداد حالات قرار دادن b توپ متمایز در u ظرف مشابه اگر ظرف خالی ممکن باشد. $B(b,u)$ همان جواب مسئله‌ی مورد نظر ماست.

چون ظرف‌ها خالی نیستند، رابطه‌ی $T(b,u)$ و $S(b,u)$ به‌صورت زیر است:

$$S(b,u) = \frac{T(b,u)}{u!}$$

برای $B(b,u)$، فرض می‌کنیم که j ظرف خالی نیستند. با استفاده از اصل جمع،

$$\begin{aligned} B(b,u) &= \sum_{j=0}^{u} S(b,j) \\ &= \sum_{j=0}^{u} \frac{T(b,j)}{j!} \\ &= \sum_{j=0}^{u} \sum_{k=0}^{j} \frac{(-1)^k}{j!} \binom{j}{k} (j-k)^b \end{aligned}$$

فرمول $T(b,j) = \sum_{k=0}^{j} (-1)^k \binom{j}{k} (j-k)^b$ با اصل شمول و عدم شمول و در بخش ۲-۲-۵ اثبات می‌شود.

۴. **تعداد حالات قرار دادن b توپ مشابه در u ظرف مشابه:** این مسئله جوابی با فرمول بسته ندارد!

۲-۲-۵ شمول و عدم شمول

اصل جمع، تعداد عناصر یک مجموعه را که اجتماع چند زیرمجموعه‌ی متمایز است، محاسبه می‌کند. قاعده‌ی شمول و عدم شمول همین تعداد را برای مجموعه‌هایی که اجتماع چند زیرمجموعه‌ی معلوم هستند به‌دست می‌آورد، در صورتی‌که این زیرمجموعه‌ها عناصر مشترک داشته باشند.

رابطه‌هایی بر روی مجموعه‌ها

چنان‌چه می‌دانیم، رابطه‌های زیر بین مجموعه‌ها برقرار است:

$$|A \cup B| = |A| + |B| - |A \cap B| \qquad (۶-۲)$$

$$|\bar{A} \cap \bar{B}| = |U| - |A \cup B| = |U| - |A| - |B| + |A \cap B| \qquad (۷-۲)$$

مثال ۲-۱۹ چه‌تعداد عدد صحیح کم‌تر یا مساوی ۱۰۰ داریم که نسبت به ۱۰۰ اول است؟

حل: می‌دانیم که $۱۰۰ = ۲^۲ ۵^۲$. بنابراین $U = \{۱, ۲, ..., ۱۰۰\}$ و داریم

$$A = \{n \in U | \text{ مضرب ۲ است } n\}$$

$$B = \{n \in U | \text{ مضرب ۵ است } n\}$$

بنابراین جواب برابر است با $|\bar{A} \cap \bar{B}| = ۱۰۰ - ۵۰ - ۲۰ + ۱۰ = ۴۰$. به‌طور کلی، قضیه‌ی زیر برقرار است:

قضیه‌ی ۲-۳ اگر $A_۱$، $A_۲$، ...، و A_n مجموعه‌هایی از مجموعه‌ی جهانی U باشند، و نیز داشته باشیم:

$$\begin{aligned} N &= |U| \\ N_۰ &= |\cap_{i=۱}^{n} \bar{A}_i| \\ N_i &= |A_i| \\ N_{ij} &= |A_i \cap A_j| \\ N_{ijk} &= |A_i \cap A_j \cap A_k| \\ N_{۱۲۳...n} &= |\cap_{i=۱}^{n} A_i| \end{aligned}$$

در آن‌صورت، رابطه‌ی زیر برقرار است:

$$N_۰ = N - \sum_i N_i + \sum_{i,j} N_{i,j} - \sum_{i,j,k} N_{i,j,k} + \cdots + (-۱)^n N_{۱۲۳...n} \qquad (۸-۲)$$

در رابطه‌ی بالا $\sum_i$ جمع برروی همه‌ی iها از ۱ تا n، و $\sum_{i,j}$ جمع بر روی همه‌ی $\binom{n}{۲}$ ترکیب دو اندیس i و j است. علامت آخرین عبارت مثبت است اگر n زوج باشد و گرنه منفی است.

در بسیاری از موارد، همه‌ی مقادیر N_i با هم برابرند، هم‌چنین همه‌ی مقادیر N_{ij} با هم برابرند، و همین‌طور حالت‌های مشابه. در این‌صورت راه ساده‌تری برای نوشتن فرمول ۲-۸ وجود دارد.

نتیجه: (رد و شمول متقارن[۲۶]) اگر برای هر k از ۱ تا n، تعداد عناصر موجود در اشتراک هر k زیرمجموعه از A_1 تا A_n مساوی و برابر n_k و نیز $n_0 = |U|$ باشد، داریم

$$N_0 = \sum_{k=0}^{n} (-1)^k \binom{n}{k} n_k \qquad (۹-۲)$$

مثال ۲-۲۰ تعداد اعداد صحیح m رقمی در مبنای ۱۰ را پیدا کنید که در آن ۰ تا ۹ دست‌کم یک‌بار آمده باشد. فرض کنید رقم ۰ می‌تواند در سمت چپ هم بیاید.

حل: فرض کنید A_i مجموعه‌ی اعداد m رقمی است که در آن‌ها رقم i نیامده باشد. روشن است که $N_i = 9^m$ و جواب این مسئله برابر $N_0 = |\cap_{i=0}^{9} \bar{A}_i|$ است. هم‌چنین اگر $S = \{i, j, k\}$ و A_S مجموعه‌ی اعداد m رقمی باشد که رقم‌های موجود در S در آن‌ها نیامده باشد، در آن‌صورت $N_S = 7^m$. به‌طور کلی اگر $|S| = k$، $N_S = (10-k)^m$. پس طبق نتیجه‌ی فوق، تعداد برابر است با

$$\sum_{k=0}^{10} (-1)^k \binom{10}{k} (10-k)^m$$

از این جواب $T(b,u)$ (b توپ متمایز در u ظرف متمایز به‌شرط آن‌که هیچ ظرفی خالی نباشد) به‌دست می‌آید

$$T(b,u) = \sum_{k=0}^{u} (-1)^k \binom{u}{k} (u-k)^m. \qquad (۱۰-۲)$$

۲-۲-۶ اصل لانه‌کبوتری

«اصل لانه‌کبوتری[۲۷]» بر این شهود روشن استوار است که،

> اگر در n لانه‌ی کبوتر بیش از n عدد کبوتر باشد، حتماً یکی از لانه‌ها دست‌کم دو کبوتر دارد.

[۲۶] Symmetric Inclusion-Exclusion
[۲۷] Pigeonhole Principle

مثال ۲-۲۱ در یک بازی، نفر اول ۱۰ عدد صحیح بین ۱ تا ۴۰ را انتخاب می‌کند و نفر دوم برای برنده شدن باید از این ۱۰ عدد دو مجموعه‌ی سه‌تایی متمایز انتخاب کند که مجموع عناصر آن دو برابر باشند. ثابت کنید که نفر دوم همواره می‌تواند برنده شود.

حل: تعداد این زیرمجموعه‌ها برابر است با $\binom{10}{3} = 120$. مجموع عناصر هر زیرمجموعه می‌تواند بین $6 = 3+2+1$ و $117 = 40+39+38$ باشد، یعنی تعداد حالت‌ها ۱۱۲ عدد است. این یعنی ۱۲۰ کبوتر در ۱۱۲ لانه! پس برای انتخاب نفر دوم دو عدد وجود دارد.

مثال ۲-۲۲

الف) نشان دهید که اگر ۱۰ نقطه داخل یک مربع 3×3 باشند، ۲ نقطه از آن‌ها پیدا می‌شوند که فاصله‌شان از هم حداکثر $\sqrt{2}$ باشد.

ب) نشان دهید که اگر n^2+1 نقطه در داخل مربع $n \times n$ داشته باشیم، دست‌کم ۲ نقطه از آن‌ها با فاصله‌ی حداکثر $\sqrt{2}$ از هم قرار دارند.

حل:

الف) مربع اصلی را به ۹ مربع 1×1 تقسیم می‌کنیم. ۱۰ کبوتر داریم و ۹ لانه. پس حتماً دست‌کم ۲ نقطه در داخل یک مربع قرار می‌گیرند.

ب) مربع اصلی را به n^2 مربع 1×1 تقسیم می‌کنیم. مانند استدلال قبلی، n^2+1 کبوتر داریم و n^2 لانه، پس حکم برقرار است.

مثال ۲-۲۳ ۱۰ بازی‌کن در یک تورنمنت شرکت می‌کنند و هر بازی‌کن دقیقاً با هر تیم دیگر یک بار بازی می‌کند. برنده‌ی هر بازی $+1$ امتیاز، بازنده -1 امتیاز و مساوی، صفر امتیاز دارد. می‌دانیم که بیش از ۷۰٪ بازی‌ها به‌تساوی کشیده شده‌اند. نشان دهید که دست‌کم دو بازی‌کن امتیاز مساوی کسب کرده‌اند.

حل: تعداد بازی‌ها برابر $\binom{10}{2} = 45$ است. دست‌کم $32 = \lceil 45 \times \frac{7}{10} \rceil$ از بازی‌ها مساوی شده‌اند، پس حداکثر ۱۳ بازی برنده دارد. با برهان خلف، فرض می‌کنیم که امتیازهای ۱۰ بازی‌کن نامساوی هستند (مثبت یا منفی). پس دست‌کم ۹ بازی‌کن امتیاز غیر صفر خواهند داشت. این یعنی دست‌کم ۵ بازی‌کن امتیاز مثبتِ غیر صفر یا ۵ بازی‌کن امتیاز منفیِ غیر صفر کسب کرده‌اند.

در حالت اول، امتیازهای این ۵ نفر دست‌کم برابر $۱۵ = ۵ + ۴ + ۳ + ۲ + ۱$ است. از این نتیجه می‌گیریم که دست‌کم ۱۵ بازی به‌تساوی کشیده نشده است، و این با فرض تناقض دارد! حالت دوم هم مانند حالت اول است.

تمرین‌های بخش ۲-۲

⋆ **۱.۲-۲** گروهی متشکل از n دانش‌مند بر روی پروژه‌ای محرمانه کار می‌کنند. اسناد این پروژه‌ها در یک گاوصندوق نگه‌داری می‌شود. می‌خواهیم فقط هنگامی که اکثریت گروه حضور دارند، باز کردن درِ این گاوصندوق امکان‌پذیر باشد. به این منظور روی درِ گاوصندوق تعدادی قفل می‌گذاریم و به هر دانش‌مند کلید بعضی از این قفل‌ها را می‌دهیم. حداقل تعداد قفل‌ها و نیز کلیدهایی را تعیین کنید که هر یک از دانش‌مندان باید داشته باشد.

⋆ **۲.۲-۲** دنباله‌ای از اعداد ۱ تا $۲n + ۱$ داده شده است. الگوریتم زیر را بر روی این اعداد اجرا می‌کنیم: ابتدا سه عنصر اول، دوم و سوم؛ سپس سه عنصر سوم، چهارم و پنجم؛ بعد سه عنصر پنجم، ششم و هفتم، ...، و در انتها عناصر $۲n - ۱$، $۲n$ و $۲n + ۱$ را مرتب می‌کنیم. برای چه تعداد از دنباله‌های ۱ تا ۹ دنباله‌ای که با این روش به‌دست می‌آید مرتب است؟

۳.۲-۲ تعداد جواب‌های صحیح و نامنفی دستگاه معادله‌ی زیر چقدر است؟

$$x_۱ + x_۲ + \cdots + x_۷ = ۳۷$$

$$x_۱ + x_۲ + x_۳ = ۶$$

۴.۲-۲ ۵ نفر به‌نام‌های $a_۱$ تا $a_۵$ در ۸ جلسه شرکت کرده‌اند. می‌دانیم که در هر جلسه دقیقاً یک نفر غایب بوده است. $a_۱$ در ۵ جلسه و $a_۲$ در ۸ جلسه شرکت داشته است و سه نفر دیگر هر یک در بیش از ۵ جلسه شرکت کرده‌اند. $a_۳$ تا $a_۵$ هر یک در چند جلسه شرکت داشته‌اند؟

۵.۲-۲ در عبارت $x_۱/x_۲/\cdots/x_n$ برای نشان دادن ترتیب عمل‌گرها، پرانتزهایی گذاشته شده است و نتیجه به‌صورت زیر در آمده است:

$$\frac{x_{i_۱} x_{i_۲} \cdots x_{i_k}}{x_{j_۱} x_{j_۲} \cdots x_{j_{n-k}}}$$

(هر یک از حرف‌های $x_۱, x_۲, \ldots, x_n$ در صورت یا در مخرج کسر قرار دارد.) اگر همه‌ی روش‌های ممکن برای پرانتزگذاری را در نظر بگیریم، به چند کسر از این‌گونه می‌رسیم؟

۶.۲-۲ n کارت داریم که پشت هر یک از آن‌ها عدد «۱+» یا «۱−» نوشته شده است. در هر حرکت می‌توانیم سه کارت از بین کارت‌ها انتخاب کنیم و بپرسیم: حاصل‌ضرب عددهای روی این سه کارت چقدر است؟ (از خود عددها اطلاعی نداریم.)

الف) دست‌کم چند بار باید این پرسش‌ها را تکرار کنیم تا بتوانیم حاصل‌ضرب عددهای روی همه‌ی کارت‌ها را پیدا کنیم؟ (راهنمایی: سه حالت $n = ۳k$، $n = ۳k + ۱$ یا $n = ۳k + ۲$ را

جداگانه بررسی کنید.)

ب) اگر کارت‌ها روی محیط یک دایره قرار گرفته باشند و در هر حرکت مجاز باشیم سه کارت متوالی انتخاب کنیم و حاصل‌ضرب آن‌ها را بپرسیم؛ برای فهمیدن حاصل‌ضرب همه‌ی عددهای روی کارت‌ها، به دست‌کم چند پرسش نیاز داریم؟

۷.۲-۲ ۲۰۰۰ بلوک ساختمانی با فاصله بر روی زمین قرار دارند. می‌خواهیم با قرار دادن همه‌ی این بلوک‌ها روی هم برجی بسازیم. برای این‌کار تعداد نامحدودی جرثقیل داریم که می‌توانند به‌صورت هم‌زمان کار کنند. هر جرثقیل می‌تواند یک برج متشکل از یک یا چند بلوک را بر روی یک برج دیگر قرار دهد و یک برج جدید بسازد. اگر تعداد بلوک‌های برجی که توسط جرثقیل برداشته می‌شود کم‌تر یا مساوی ۱۰۰ باشد، این‌کار یک ساعت و در غیر این‌صورت دو ساعت طول می‌کشد. برای ساختن این برجِ ۲۰۰۰ بلوکی به حداقل چند ساعت وقت نیاز است؟

۸.۲-۲ ماتریس M با درایه‌های ۰ و ۱ و ابعاد $۲^n \times ۲^n$ موجود است. S رشته‌ی متناظر با ماتریس M را به‌صورت زیر محاسبه می‌کنیم: اگر تمام درایه‌های M صفر باشند؛ $S = ۰$ و اگر همه‌ی آن‌ها یک باشند، $S = ۱$ است، و گرنه ماتریس را به چهار ماتریس مساوی $M_۱$، $M_۲$، $M_۳$ و $M_۴$ تقسیم می‌کنیم. رشته‌ی S_i $(i = ۱, ۲, ۳, ۴)$ متناظر با ماتریس M_i را به‌دست می‌آوریم. سپس $S = ۲S_۱S_۲S_۳S_۴$. برای مثال رشته‌ی متناظر با ماتریس شکل، برابر ۲۱۰۰۱ است.

$M_۱$	$M_۲$
$M_۳$	$M_۴$

$\Rightarrow$

۱	۰
۰	۱

چند رشته‌ی متفاوت متناظر با ماتریس‌های $۲^n \times ۲^n$ وجود دارد؟

۹.۲-۲ n نفر به نام‌های $a_۱$، $a_۲$، $\cdots$ تا a_n هرکدام خبری در اختیار دارند. خبر a_i را b_i می‌نامیم. توجه کنید که b_iها در حالت کلی با هم متفاوتند.

دو نفر می‌توانند با هم تماس بگیرند. اگر a_i با a_j تماس بگیرد و a_i قبل از تماس، مجموعه‌ی خبرهای β_i و a_j مجموعه‌ی خبرهای β_j را داشته باشد، پس از تماس، هر دو از خبرهای $\beta_i \cup \beta_j$ مطلع خواهند شد. به‌عبارتی خبرهای خود را به یک‌دیگر منتقل می‌کنند.

یک مرحله از خبرپراکنی، تماس هم‌زمان و دوبه‌دوی این n نفر با هم است. توجه کنید که در یک مرحله از خبرپراکنی یک فرد نمی‌تواند با بیش از یک نفر دیگر تماس بگیرد. هدف این است که بیابیم در چند مرحله و چگونه می‌توان خبرهای اولیه، b_iها، را در اختیار همه‌ی a_iها قرار داد.

الف) اگر $n = ۷$ باشد، کمینه‌ی تعداد مراحل خبرپراکنی را به‌دست آورید و نیز نشان دهید که در هر مرحله چه افرادی باید با هم تماس بگیرند.

ب) کم‌ترین تعداد مرحله را برای $n = ۲^k$ به‌دست آورید و ثابت کنید.

⋆ **۱۰.۲-۲** ماتریس A به اندازه‌ی $n \times n$ از اعداد حقیقی را در نظر بگیرید. در این ماتریس اختلاف بزرگ‌ترین و کوچک‌ترین عدد در هر سطر d است. اعداد هر ستون را به‌صورت نزولی مرتب

می‌کنیم. نشان دهید که در ماتریس حاصل هم اختلاف بزرگ‌ترین و کوچک‌ترین عدد در هر سطر حداکثر d است.

⋆ **۲-۱۱.۲** دانش‌جویی a روز فرصت دارد که برای آزمونی آماده شود. او می‌داند که برای این آزمون به بیش از b ساعت مطالعه نیاز ندارد، البته مایل است که روزانه دست‌کم ۱ ساعت مطالعه کند. اگر تعداد ساعات مطالعه‌ی روزانه‌اش عدد طبیعی باشد، به‌ازای چه مقادیری از c، حتماً چند روز متوالی وجود دارد که جمع ساعات مطالعه‌اش دقیقاً c شود.

⋆ **۲-۱۲.۲** در یک مسابقه‌ی دوره‌ای، هر دو تیم یک‌بار با هم بازی می‌کنند. برنده ۲ و بازنده ۰ امتیاز و در صورت تساوی، هر تیم ۱ امتیاز می‌گیرد. ۲۸ تیم در این مسابقه شرکت کرده‌اند. می‌دانیم که بیش از ۷۵٪ بازی‌ها به تساوی کشیده شده‌اند. ثابت کنید ۲ تیم امتیازشان مساوی شده است.

۲-۱۳.۲ ترتیبی از چهار عدد صحیح مثبت را در نظر بگیرید. با انجام این عمل، یک ترتیب دیگر به‌دست می‌آوریم: ابتدا تفاضل عددهای اول و دوم، سپس تفاضل عددهای دوم و سوم، بعد تفاضل عددهای سوم و چهارم، و در نهایت تفاضل عددهای اول و چهارم را می‌نویسیم. ($|a-b|$ تفاضل دو عدد a و b است.) برای مثال از ترتیب $\langle ۵, ۴, ۸, ۲۷\rangle$ ترتیب $\langle ۱, ۴, ۱۹, ۲۲\rangle$ ساخته می‌شود.

الف) ثابت کنید که اگر این عمل را چند بار انجام دهیم، بالاخره بعد از مدتی به ترتیب $\langle ۰, ۰, ۰, ۰\rangle$ می‌رسیم. برای مثال برای ترتیب $\langle ۵, ۴, ۸, ۲۷\rangle$ پس از ۶ مرحله به $\langle ۰, ۰, ۰, ۰\rangle$ می‌رسیم.

$$\begin{aligned}
&\langle ۵, ۴, ۸, ۲۷\rangle \\
\to\ &\langle ۱, ۴, ۱۹, ۲۲\rangle \\
\to\ &\langle ۳, ۱۵, ۳, ۲۱\rangle \\
\to\ &\langle ۱۲, ۱۲, ۱۸, ۱۸\rangle \\
\to\ &\langle ۰, ۶, ۰, ۶\rangle \\
\to\ &\langle ۶, ۶, ۶, ۶\rangle \\
\to\ &\langle ۰, ۰, ۰, ۰\rangle
\end{aligned}$$

ب) ثابت کنید اگر به جای ۴ عدد ۳ عدد داشته باشیم، گزاره‌ی (الف) ممکن است درست نباشد.[۲۸]

۲-۱۴.۲ جدولی را در نظر بگیرید که از سمت چپ، راست و پایین نامتناهی و فقط از طرف بالا محدود است. بالاترین سطر جدول، سطر شماره‌ی ۱ است و سطرها به طرف پایین شماره‌گذاری می‌شوند. در سطر اول، در یک خانه عدد ۱ و در بقیه‌ی خانه‌ها عدد صفر نوشته شده است. در سطرهای بعدی یک خانه مقدار ۱ دارد اگر و فقط اگر دقیقاً یکی از خانه‌های چپ و راست خانه‌های

[۲۸] مسئله‌ی ۴، مرحله‌ی اول پنجمین المپیاد کامپیوتر ایران

بالای آن ۱ باشد، در غیر این‌صورت مقدارش صفر خواهد بود. در مثال زیر چند سطر اول این جدول نوشته شده است.

```
... ۰ ۰ ۰ ۰ ۰ ۰ ۱ ۰ ۰ ۰ ۰ ۰ ۰ ...
... ۰ ۰ ۰ ۰ ۰ ۱ ۰ ۱ ۰ ۰ ۰ ۰ ۰ ...
... ۰ ۰ ۰ ۰ ۱ ۰ ۰ ۰ ۱ ۰ ۰ ۰ ۰ ...
... ۰ ۰ ۰ ۱ ۰ ۱ ۰ ۱ ۰ ۱ ۰ ۰ ۰ ...
... ۰ ۰ ۱ ۰ ۰ ۰ ۰ ۰ ۰ ۰ ۱ ۰ ۰ ...
... ۰ ۱ ۰ ۱ ۰ ۰ ۰ ۰ ۰ ۱ ۰ ۱ ۰ ...
        ⋮       ⋮       ⋮
```

در سطر ۱۳۸۸ این جدول چند عدد ۱ وجود دارد؟ روش محاسبه‌ی خود را دقیقاً بیان و ثابت کنید.[۲۹]

[۲۹] مسئله‌ی ۴، مرحله‌ی دوم دهمین المپیاد کامپیوتر ایران

فصل

۳

روش‌های تحلیل الگوریتم‌ها

تحلیل الگوریتم‌ها با هدف‌های زیر انجام می‌شود:

- بررسی و پیش‌بینی زمان اجرا و میزان حافظه‌ی مصرفی یک الگوریتم قبل از پیاده‌سازی و
- مقایسه‌ی الگوریتم‌های مختلف برای حل یک مسئله از نظر میزان کارایی.

زمان اجرای یک برنامه و مقدار حافظه‌ی مصرفی آن برای ورودی‌های مختلف، میزان خوبی برای سنجش کارایی الگوریتم استفاده شده در آن برنامه است.

در این فصل با روش‌های تحلیل الگوریتم‌ها آشنا می‌شویم. به‌این منظور، در مورد موضوعاتی چون تابع‌های رشد، روش‌های تحلیل الگوریتم‌های ترتیبی و بازگشتی، روش‌های حل رابطه‌های بازگشتی ساده، همگن و ناهمگن و نیز روش تحلیل سرشکنی به بحث می‌پردازیم. هر موضوع را با ذکر مثال‌هایی از الگوریتم‌های واقعی بررسی خواهیم کرد. مثلاً با جزئیات الگوریتم‌های ساده‌ی مرتب‌سازی مانند مرتب‌سازی درجی، حبابی، ادغامی و چند مسئله‌ی ساده‌ی دیگر مانند برج‌های هانوی آشنا می‌شویم، و در مورد اثبات درستی و تحلیل آن‌ها به بحث خواهیم پرداخت.

پیوست ۲ حاوی تعدادی نماد و تابع‌های مهم است که در این فصل و دیگر بخش‌های کتاب مورد استفاده قرار می‌گیرند.

۳-۱ زمان اجرای برنامه‌ها

عوامل زیر در زمان اجرای یک برنامه بر روی یک کامپیوتر مؤثرند:

۱. سرعت پردازنده‌ی کامپیوتر،

۲. نوع کامپایلر یا زبان برنامه‌نویسی استفاده‌شده،

۳. اندازه‌ی داده‌ی ورودی مسئله،

۴. ترکیب یا ساختار داده‌های ورودی،

۵. پیچیدگی الگوریتم استفاده‌شده در برنامه، و

۶. عوامل دیگر که وابسته به ورودی نیستند و تأثیر خطی در زمان اجرای برنامه دارند.

از این موارد، سرعت پردازنده، نوع کامپایلر استفاده‌شده و عوامل مشابه به‌صورت خطی و با ضریب ثابت در زمان اجرای برنامه‌ها مؤثرند؛ یعنی مثلاً اگر پردازنده‌ی کامپیوتر را دو برابر سریع کنیم و دیگر عوامل تغییر نکنند، انتظار داریم که در بهترین حالت، زمان اجرای برنامه بر روی کامپیوتر جدید نصف شود. انتخاب نوع کامپایلر هم تأثیری مشابه سرعت پردازنده دارد.

از عوامل ذکر شده‌ی بالا، پیچیدگی ذاتی الگوریتمی که در برنامه استفاده شده است، یک ویژگی بسیار مهم است. بعداً به‌تفصیل و با ذکر مثال‌هایی در مورد پیچیدگی توضیح خواهیم داد، اما عجالتاً باید گفت که پیچیدگی مفهومی انتزاعی و مستقل از زبان برنامه‌نویسی و سخت‌افزار مورد استفاده است و معمولاً آن‌را به‌صورت تابعی از مشخصه‌های مهم داده‌های ورودی مسئله تعریف می‌کنیم.

هر ورودی مسئله دو مشخصه‌ی مهم دارد: یکی اندازه یا تعداد داده‌ها و دیگری ترکیب یا ساختار آن‌ها، که دومی رابطه‌ی بین جزئیات داده‌ها را بیان می‌کند.

روشن است که هر قدر ورودی بزرگ‌تر باشد، اجرای برنامه هم با آن ورودی زمان بیش‌تری به‌طول خواهد کشید. تابع پیچیدگی الگوریتم استفاده شده، رابطه‌ی این دو مقدار را معین می‌کند.

این فقط اندازه‌ی ورودی نیست که در زمان اجرا مؤثر است، بلکه ساختار یا ترکیب اجزای داده‌ی ورودی هم می‌تواند شرایطی را ایجاد کند که برنامه مثلاً خیلی کند، یا برای همان تعداد داده با ترکیب متفاوت، سریع جواب دهد. مثالی از ترکیب داده‌ی ورودی، مثلاً مرتب‌بودن یا نبودن، برعکس یا تقریباً مرتب‌بودن مجموعه‌ی داده در یک الگوریتم مرتب‌سازی است.

از مفهوم ترکیب داده‌ی ورودی نتیجه می‌گیریم که بهتر است در صورت امکان، تابع زمان اجرای یک برنامه (یا پیچیدگی الگوریتم مورد استفاده) را هم برای بهترین و هم برای بدترین حالت ورودی (که برای یک اندازه‌ی ورودی ثابت، به‌ترتیب کم‌ترین و بیش‌ترین زمان اجرا را نشان می‌دهد) به‌دست آوریم. همچنین با فرض داشتن یک تابع توزیع از ترکیب ورودی، می‌توانیم تابع زمان اجرای قابل انتظار (یا پیچیدگی در حالت میانگین) را هم به‌دست آوریم. اگر تابع پیچیدگی را در حالت‌های مختلفی که گفته شد به‌دست آوریم، دانش ما از الگوریتم مورد استفاده بیش‌تر می‌شود.

زمان اجرای یک برنامه را می‌توان به‌صورت تابعی از مشخصه‌های مهم داده‌ی ورودی آن بیان کرد. مثلاً اگر اندازه‌ی ورودی مسئله، یا n، یک مشخصه‌ی مهم ورودی باشد، $T(n)$ یک مثال از این تابع است. در مواردی، مسئله ممکن است چند نوع داده‌ی ورودی داشته باشد، مثلاً اگر ورودی یک گراف باشد، علاوه بر تعداد رأس‌ها (مثلاً n)، تعداد یال‌های آن گراف (مثلاً m) هم از مشخصه‌های داده‌ی ورودی است. در این صورت، زمان اجرای الگوریتم را می‌توان با تابعی مانند $T(n,m)$ بیان کرد. ممکن است مسئله چندین مشخصه داشته باشد، اما ناچاریم با انجام انتزاع فقط آن‌هایی را که در تحلیل مهم هستند در نظر بگیریم.

در این کتاب به هر مقدار و پارامتری که مستقل از اندازه یا ترکیب داده‌ی ورودی باشد، «ثابت» می‌گوییم. بر این اساس، چنان‌چه خواهیم دید، در تحلیل زمان اجرای برنامه یا محاسبه‌ی پیچیدگی الگوریتم استفاده شده در آن، می‌توان از عوامل ثابت چشم‌پوشی کرد. آن‌چه واقعاً اهمیت دارد، تابع زمان اجرای برنامه برحسب اندازه‌ی ورودی‌های مختلف و به‌خصوص اندازه‌ی ورودی بزرگ است.

مراحل مختلف تحلیل الگوریتم‌ها را با ذکر یک مثال نشان می‌دهیم.

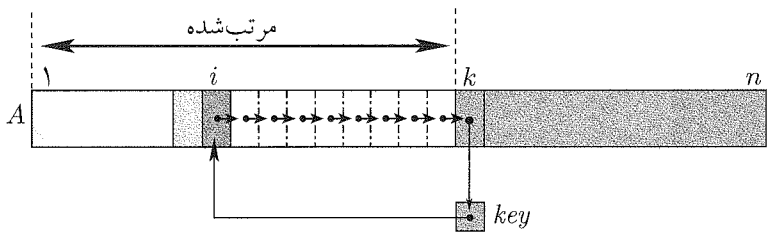
۳-۱-۱ مثال: مرتب‌سازی درجی

مرتب‌سازی درجی یک الگوریتم ساده‌ی مرتب‌سازی است. این الگوریتم در ابتدای مرحله‌ی kام (برای $2 \le k \le n$) فرض می‌کند که عناصر موجود در درایه‌های ۱ تا $k-1$ آرایه نسبت به‌هم مرتب هستند. در انتهای این مرحله، $A[k]$ هم در جای مناسبی بین این عناصر درج می‌شود، تا همه‌ی این k عنصر مرتب شوند. رویه‌ی INSERTION-SORT این‌کار را انجام می‌دهد.

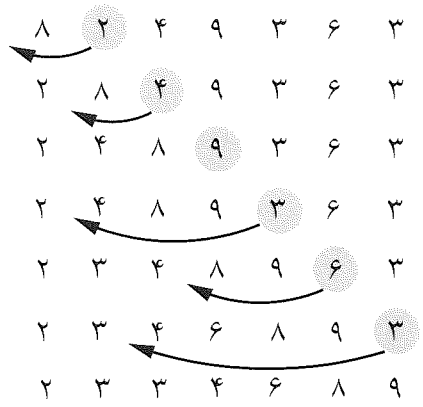
```
INSERTION-SORT (A, n)
    ▷ A: آرایه‌ای که باید مرتب شود
    ▷ n: (length[A] یعنی) طول آرایه‌ی A
1   for k ← 2 to n
2     do key ← A[k]
3        i ← k
4        while i > 1 and A[i − 1] > key
5           do A[i] ← A[i − 1]
6              i ← i − 1
7        A[i] ← key
```

نحوه‌ی درج به‌این‌صورت است که $A[k]$ درابتدا در متغیری به‌نام key قرار می‌گیرد. سپس key به‌ترتیب با تک‌تک عنصرهای قبلی (یا $A[i]$ برای $i < k$) مقایسه می‌شود و اگر $A[i]$ بزرگ‌تر بود، در درایه‌ی بعدی نوشته می‌شود تا این‌که به‌عنصری برسیم که کوچک‌تر یا مساوی key باشد. در واقع، همه‌ی عناصری که اکیداً از key بزرگ‌ترند، یک مکان شیفت داده می‌شوند تا جا برای درج key در درایه‌ی مورد نظر باز شود، به‌طوری که در انتهای این مرحله درایه‌های ۱ تا k مرتب شوند (این همان «رابطه‌ی مستقل از حلقه» برای این الگوریتم است که از آن برای اثبات درستی الگوریتم‌های غیربازگشتی استفاده می‌شود). توجه کنید که در این الگوریتم ترتیب نسبی عناصر با کلیدهای یک‌سان در انتها تغییر نمی‌کند. به چنین الگوریتم‌هایی «پایدار» می‌گوییم و این از ویژگی‌های مهم یک الگوریتم مرتب‌سازی است.

شکل ۳-۱ شمایی از مرحله‌ی kام مرتب‌سازی درجی و شکل ۳-۲ مثالی از این الگوریتم را نمایش می‌دهد.



شکل ۳-۱ مرحله‌ی kام مرتب‌سازی درجی.



شکل ۳-۲ مثالی از مراحل اجرای مرتب‌سازی درجی.

برای محاسبه‌ی زمان اجرای این برنامه، فرض می‌کنیم که هر بار اجرای سطر i ام به‌اندازه‌ی c_i واحد زمان می‌گیرد. یا به‌زبانی دیگر هزینه‌ی اجرای آن سطر c_i است. حال تعداد تکرار هر سطر را نسبت به اندازه‌ی ورودی، یا n، می‌شماریم. این اعداد در جدول ۳-۱ آمده‌اند. تعداد تکرار سطر ۴ وابسته به ترکیب داده‌ی ورودی است. این تعداد را در مرحله‌ی k ام، t_k می‌نامیم. توجه کنید که در آن‌صورت سطر ۵ و ۶ هر کدام $t_k - ۱$ بار تکرار می‌شوند.

جمع حاصل‌ضرب تعداد دفعات اجرای هر سطر در زمان اجرای آن، برای همه‌ی سطرها، زمان اجرای کل را به‌دست می‌دهد:

$$T(n) = c_۱ n + (c_۲ + c_۳ + c_۷)(n - ۱) + c_۴ \sum_{k=۲}^{n} t_k + (c_۵ + c_۶) \sum_{k=۲}^{n} (t_k - ۱)$$

مقادیر ثابت‌های $c_۱$ تا $c_۷$ به نوع سخت‌افزار، زبان برنامه‌نویسی مورد استفاده و عوامل مشابه دیگری بستگی دارد که به‌صورت تجربی قابل محاسبه‌اند. می‌توانیم به‌جای جمع چند ثابت و عباراتی از آن‌ها، ثابت جدیدی قرار دهیم.

$$T(n) = An + B \sum_{k=۲}^{n} t_k + C \qquad (۱-۳)$$

جدول ۳-۱ هزینه‌ی انجام هر سطر در الگوریتم INSERTION-SORT و تعداد تکرار هر سطر.

سطر	هزینه	تعداد
۱	c_1	n
۲	c_2	$n-1$
۳	c_3	$n-1$
۴	c_4	$\sum_{k=2}^{n} t_k$
۵	c_5	$\sum_{k=2}^{n} (t_k-1)$
۶	c_6	$\sum_{k=2}^{n} (t_k-1)$
۷	c_7	$n-1$

بهترین و بدترین زمان اجرای مرتب‌سازی درجی

مقدار واقعی زمان اجرای این رویه به مقدار t_kها بستگی دارد. «بهترین حالت[1]» هنگامی اتفاق می‌افتد که آرایه از قبل مرتب باشد. در این حالت همیشه $t_k = 1$ و در نتیجه،

$$T(n) = An + B(n-1) + C,$$

یک تابع خطی برحسب n است.

البته رفتار یک الگوریتم در بهترین حالت چندان مهم نیست. آن‌چه اهمیت دارد رفتار الگوریتم در «بدترین حالت[2]» و نیز «حالت میانگین[3]» است.

بدترین زمان اجرای این الگوریتم هنگامی اتفاق می‌افتد که آرایه برعکس مرتب شده باشد. در این صورت، مقدار t_k برای همه‌ی مقادیر k برابر k است (مقایسه‌ی آخر با صفر هم حساب شده است). برای این حالت داریم،

$$\begin{aligned} T(n) &= An + B\sum_{k=2}^{n} t_k + C \\ &= An + B\left(\frac{(n-1)(n+2)}{2} - 1\right) + C \end{aligned}$$

[1] best-case
[2] worst-case
[3] average-case

$$= an^۲ + bn + c, \qquad (۲-۳)$$

که یک تابع درجه‌ی ۲ برحسب n است. از این‌که بهترین و بدترین رفتار این الگوریتم با هم متفاوتند، در می‌یابیم که رفتار میانگین آن، که در ادامه توضیح داده می‌شود، قابل توجه است.

میانگین زمان اجرا

زمان اجرای یک الگوریتم را در حالت میانگین به‌صورت زیر تعریف می‌کنیم:

> با فرض این‌که ترتیب اولیه‌ی داده‌ی ورودی با احتمال یک‌سان یکی از حالات ممکن است، رفتار الگوریتم (زمان اجرای آن) به‌طور میانگین چقدر است؟ یعنی به‌ازای یک ورودی دل‌خواه تصادفی، امید ریاضی زمان اجرای الگوریتم چقدر است؟

در مورد مرتب‌سازی درجی، هر حالت ورودی با n داده یکی از $n!$ جای‌گشت ممکن است و طبق فرض بالا، احتمال هر حالت برابر $\frac{۱}{n!}$ فرض می‌شود. اگر $t_{k,i}$ مقدار t_k در حالت i باشد، زمان اجرا در حالت میانگین برابر است با

$$\overline{T}(n) = An + C + \frac{B}{n!}\sum_{i=۱}^{n!}\sum_{k=۲}^{n} t_{k,i}.$$

که در آن عملاً تک‌تک حالت‌ها بررسی و میانگین آن‌ها محاسبه می‌شود. اما اگر مقدار میانگین t_k را با $\bar{t}_k$ نشان دهیم، داریم

$$\bar{t}_k = \frac{۱}{n!}\sum_{i=۱}^{n!} t_{k,i}.$$

در آن‌صورت و با توجه به آن‌که جابه‌جایی دو مجموع به‌سادگی قابل اثبات است، داریم

$$\overline{T}(n) = An + C + B\sum_{k=۲}^{n} \bar{t}_k. \qquad (۳-۳)$$

برای محاسبه‌ی $\bar{t}_k$ فرض کنید که در مرحله‌ی k ام، عنصر $A[k]$ پس از تعدادی جابه‌جایی در $A[i]$ قرار می‌گیرد. می‌دانیم که $۱ \le i \le k$ و احتمال هر یک از این حالات برابر $\frac{۱}{k}$ است.

روشن است که در آن صورت $t_k = k - i + ۱$. پس،

$$\begin{aligned} \bar{t}_k &= \sum_{i=۱}^{k} \frac{۱}{k}(k - i + ۱) \\ &= \frac{۱}{k} \times \frac{k(k+۱)}{۲} \\ &= \frac{k+۱}{۲}. \end{aligned} \tag{۴-۳}$$

این نتیجه به‌صورت شهودی قابل پیش‌بینی بود. یعنی در هر مرحله به‌طور میانگین عضو مورد نظر به وسط بخش مرتب‌شده‌ی آرایه منتقل می‌شود.

از فرمول‌های ۳-۳ و ۳-۴ نتیجه می‌گیریم،

$$\begin{aligned} \overline{T}(n) &= An + C + B\sum_{k=۲}^{n} \frac{k+۱}{۲} \\ &= a'n^۲ + b'n + c'. \end{aligned} \tag{۵-۳}$$

پس رفتار الگوریتم در حالت میانگین و نیز در بدترین حالت از نظر آهنگ رشد، متناسب با یک تابع (درجه‌ی ۲) است.

۳-۱-۲ مثال: مرتب‌سازی درجی دودویی

الگوریتم ارائه شده در این بخش را گاهی «مرتب‌سازی درجی مستقیم[۴]» نیز می‌نامیم. گونه‌ی دیگری از این الگوریتم «مرتب‌سازی درجی دودویی[۵]» است که در مرحله‌ی k برای یافتن محل نهایی درج عنصر $A[k]$، از جست‌وجوی دودویی[۶] استفاده می‌کند. این‌کار حداکثر به‌تعداد $\lceil \lg k \rceil$ مقایسه بین عناصر آرایه نیاز دارد. رویه‌ی این جست‌وجو در BINARY-SEARCH آمده است.

این الگوریتم بر اساس روش «تقسیم‌وحل[۷]» کار می‌کند که در هر مرحله مسئله را به دو بخش با اندازه‌ی تقریباً مساوی تقسیم می‌کند و کار جست‌وجو را فقط در یکی از این بخش‌ها دنبال می‌کند. درستی این الگوریتم به‌سادگی با استقرا اثبات می‌شود، البته باید

[۴] straight insertion sort
[۵] binary insertion sort
[۶] binary search
[۷] divide and conquer

پایه‌ی استقرا (یعنی $l = r$) به‌دقت بررسی شود. بعداً، به روش استقرا هم این الگوریتم را تحلیل خواهیم کرد.

```
BINARY-SEARCH (A, l, r, key)
    ▷ آرایه‌ی A مرتب است
    ▷ اندیس i را پیدا کن به‌طوری‌که l ≤ i ≤ r و key حتماً قبل از A[i] در آرایه درج شود
1   if l ≥ r
2      then i = l
3   m ← ⌊(l+r)/2⌋
4   if key ≤ A[m]
5      then i ← BINARY-SEARCH(A, l, m − 1)
6      else i ← BINARY-SEARCH(A, m + 1, r)
7   return i
```

در صورتی که بتوان عمل واقعی درج را هم در زمانی متناسب با لگاریتم تعداد عناصر انجام داد، زمان اجرای کل این الگوریتم مرتب‌سازی متناسب با $n \lg n$ خواهد شد که ــ چنان‌چه خواهیم دید ــ یک الگوریتم بهینه است. اما هزینه‌ی درج واقعی عنصر k ام در داده‌ساختار آرایه، همان‌طور که در مثال قبل دیدیم، متناسب با k است؛ چرا که باید تعدادی (حداکثر k) عنصر را جابه‌جا کنیم. پس با این ساختار نمی‌توان به الگوریتم بهینه رسید و لازم است از داده‌ساختارهای مناسب دیگری استفاده کرد. بعداً خواهیم دید که «درخت‌های دودویی جست‌وجوی متوازن[۸]» برای این‌کار مناسب هستند، هرچند که سربار الگوریتم افزایش خواهد یافت و حافظه‌ی بیش‌تری مورد نیاز خواهد بود.

تمرین‌های بخش ۳-۱

۳-۱.۱ روش هورنر[۹] مطابق فرمول زیر مقدار یک چندجمله‌ای $P(x)$ را با داشتن ضرایب a_0، a_1 تا a_n و مقدار x به‌دست می‌آورد.

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

[۸] balanced binary search trees
[۹] Horner

قطعه‌کد زیر این روش را پیاده‌سازی کرده است.

```
HORNER (P, x)
1  y ← 0
2  i ← n
3  while i ≥ 0
4      do y ← a_i + x · y
5         i ← i − 1
6  return y
```

الف) زمان اجرای این قطعه‌کد چقدر است؟

ب) شبه‌کدی بنویسید که الگوریتم ساده‌ی ارزیابی مقدار چندجمله‌ای را پیاده‌سازی کند که در آن هر جمله به‌طور مجزا از پایه محاسبه می‌شود. زمان اجرای این الگوریتم چیست؟ زمان اجرای آن‌را با پیاده‌سازی روش هورنر مقایسه کنید.

پ) نشان دهید که رابطه‌ی مستقل از حلقه‌ی زیر برای **while** در سطرهای ۳ تا ۵ درست است.

در آغاز هر تکرار حلقه‌ی **while** در سطرهای ۳ تا ۵، رابطه‌ی زیر برقرار است:

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

مجموعی را که هیچ جمله‌ای ندارد، برابر با ۰ در نظر بگیرید. همچنین نشان دهید که در پایان، $y = \sum_{k=0}^{n} a_k x^k$.

ت) با استفاده از بخش‌های قبل، نتیجه بگیرید که این قطعه‌کد مقدار چندجمله‌ای با ضرایب a_1، a_2 تا a_n را به‌درستی محاسبه می‌کند.

۳-۱-۲ الگوریتم زیر، عناصر بیشینه و کمینه‌ی آرایه‌ی A را به‌دست می‌آورد:

```
MINMAX (A)
1   min ← 1; max ← 1
2   for i ← 2 to N
3     do Set B=true or B=false with equal probabilities
4        if B
5          then if A[i] < A[min]
6                    then min ← i
7                    else  if A[i] > A[max]
8                              then max ← i
9          else
10                 if A[i] > A[max]
11                   then max ← i
12                   else  if A[i] < A[min]
13                             then min ← i
```

حداقل، حداکثر و میانگین تعداد مقایسه‌های بین عناصر این آرایه را دقیقاً محاسبه کنید.

۳.۱-۳ آرایه‌ی n تایی A داده شده است. می‌خواهیم از آن ماتریس B را بسازیم که در آن $B[i,j] = \sum_{k=i}^{j} A[k]$ (برای $i \le j$). اگر $i > j$ مقدار $B[i,j]$ مهم نیست.

الف) الگوریتم زیر را برای محاسبه‌ی B پیش‌نهاد می‌کنیم:

```
1  for i ← 1 to n
2    do for j = i to n
3         do B[i,j] = Σ_{k=i}^{j} A[k]
```

دقیقاً چه تعداد عمل جمع در این الگوریتم انجام می‌شود؟

ب) الگوریتمی با تعداد بهینه‌ی جمع ارائه دهید. این تعداد دقیقاً چقدر است؟

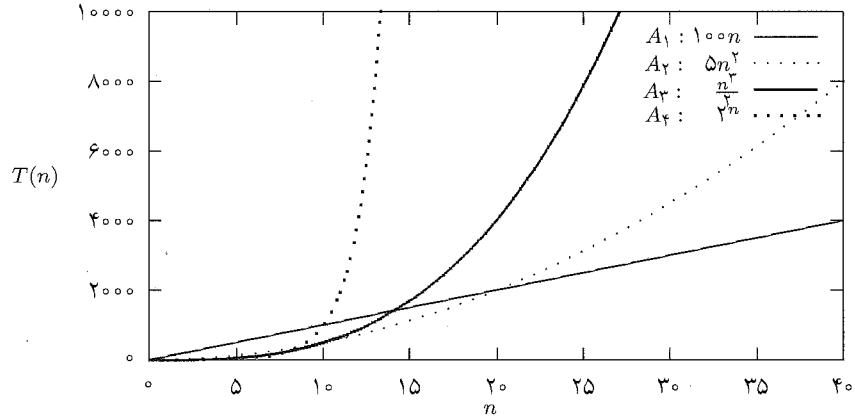
۲-۳ پیچیدگی الگوریتم‌ها

چنان‌چه گفته شد، در تحلیل یک الگوریتم، بهترین زمان اجرای آن اهمیت چندانی ندارد، چرا که این حالت معمولاً برای ورودی با تعداد کم و در شرایط خاص اتفاق می‌افتد، و زمان اجرا هم اغلب آن‌قدر کوتاه است که مهم نیست از چه الگوریتمی استفاده شود.

اما لازم است الگوریتم‌ها را در بدترین حالت و در صورت امکان در حالت میانگین تحلیل و بر این اساس، الگوریتم مناسب را انتخاب کنیم. اگر رفتار یک الگوریتم یا پیچیدگی آن در بهترین و بدترین حالات مختلف باشد، ممکن است پیچیدگی الگوریتم در حالت میانگین بهتر از پیچیدگی آن در بدترین حالت باشد، و این اطلاعی مهم در مورد آن الگوریتم است. در چنین وضعیتی اغلب بدترین حالت الگوریتم نیز به‌ندرت اتفاق می‌افتد و پیچیدگی حالت میانگین الگوریتم، همان رفتار مورد انتظار است. برای مثال، زمان اجرای الگوریتم «مرتب‌سازی سریع[۱۰]» در بدترین حالت متناسب با n^2 و در حالت میانگین متناسب با $n \lg n$ است که اختلاف این دو تابع برای مقادیر بزرگ n چشم‌گیر است.

باید توجه داشت که مقدار زمان اجرای الگوریتم در اندازه‌ی ورودی بزرگ یا خیلی بزرگ بسیار تعیین‌کننده است. در این حالت، چنان‌چه خواهیم دید، مقدار ثابت‌ها در معادله‌ی زمان اجرا، مثلاً A، B و C در الگوریتم مرتب‌سازی درجی (معادله‌ی ۳-۱)، تأثیر قابل‌توجهی ندارد، و با افزایش اندازه‌ی مسئله اثرشان کم‌تر می‌شود، لذا آن‌ها را نادیده

[۱۰] quicksort



شکل ۳-۳ زمان‌های اجرای چهار الگوریتم مختلف A_1 تا A_4 برای حل یک مسئله. محور افقی n اندازه‌ی ورودی مسئله و محور عمودی $T(n)$، زمان اجرای آن الگوریتم برحسب ثانیه است. روشن است که برای n های بزرگ، الگوریتم‌های A_1، A_2، A_3 و A_4 به‌ترتیب سریع‌ترین تا کندترین این الگوریتم‌ها هستند.

می‌گیریم. این مطلب با مفهوم «پیچیدگی الگوریتم[11]» نشان داده می‌شود.

برای درک بهتر مفهوم پیچیدگی، فرض کنید برای حل یک مسئله با n عدد ورودی چهار الگوریتم مختلف با نام‌های A_1 تا A_4 در دست داریم. هم‌چنین فرض کنید که زمان اجرای این الگوریتم‌ها برحسب n، یعنی $T(n)$ را به‌طور دقیق و بر حسب ثانیه به‌دست آورده‌ایم. منحنی‌های زمان‌های اجرای این الگوریتم‌ها را می‌توانید در شکل ۳-۳ و نیز اطلاعات لازم را در دو ستون اول جدول ۳-۲ ببینید. در ستون سوم جدول، پیچیدگی این الگوریتم‌ها با نماد $\mathcal{O}$ آمده است. در این خصوص در ادامه بیش‌تر صحبت خواهیم کرد. روشن است که A_1 از نظر درجه‌ی پیچیدگی سریع‌ترین و A_4 کندترین الگوریتم است، A_2 و A_3 هم به‌همین ترتیب در این میان قرار می‌گیرند.

فرض کنید که حداکثر می‌توانیم ۱۰۰۰ ثانیه برای دریافت پاسخ یک برنامه صبر کنیم. بنابراین مثلاً اگر از الگوریتم A_1 استفاده کنیم، بزرگ‌ترین مسئله‌ای که می‌توانیم با آن حل کنیم برابر ۱۰ است ($100n = 1000$). همین اطلاعات برای الگوریتم‌های دیگر نیز محاسبه و در ستون چهارم جدول نوشته شده است. حال فرض کنید سرعت پردازنده را ۱۰ برابر سریع‌تر کنیم. در این صورت زمان انتظار ما ۱۰,۰۰۰ ثانیه بر روی پردازنده‌ی اولی است. به

[11] complexity of algorithm

جدول ۳-۲ مقایسه‌ی پیچیدگی چهار الگوریتم A_1 تا A_4. ستون دوم زمان اجرای هر الگوریتم برحسب ثانیه و به‌صورت تابعی از اندازه‌ی ورودی مسئله (n) است. در ستون سوم درجه‌ی پیچیدگی برحسب نماد $\mathcal{O}$ بیان شده است.

الگوریتم	$T(n)$	$\mathcal{O}(.)$	حداکثر اندازه‌ی مسئله که در ۱۰۰۰ ثانیه حل می‌شود	حداکثراندازه‌ی مسئله در ۱۰۰۰ ثانیه با پردازنده‌ی ۱۰ برابر سریع‌تر	نسبت اعداد ستون ۵ به ستون ۴	نسبت اگر پردازنده‌ای ۱۰۰۰ برابرسریع‌تر انتخاب کرده باشیم
A_1	$100n$	$\mathcal{O}(n)$	۱۰	۱۰۰	۱۰	۱۰۰۰
A_2	$5n^2$	$\mathcal{O}(n^2)$	۱۴	۴۵	۳/۲	۱۳۱/۹۴
A_3	$n^3/2$	$\mathcal{O}(n^3)$	۱۲	۲۷	۲/۳	۱۲۵/۹۹
A_4	2^n	$\mathcal{O}(2^n)$	۱۰	۱۳	۱/۳	۲

این ترتیب حداکثر اندازه‌ی مسئله‌ای که با این الگوریتم‌ها بر روی پردازنده‌ی سریع‌تر می‌توان اجرا کرد به‌دست می‌آید که این اعداد و نسبت‌شان به اعداد قبلی در جدول آمده است. این اطلاعات را برای حالتی که پردازنده‌ای ۱۰۰۰ برابر سریع‌تر انتخاب کنیم نیز به‌دست آورده‌ایم و در جدول نشان داده‌ایم.

در شکل ۳-۳ می‌بینیم که با این‌که A_4 درجه‌ی پیچیدگی بالایی دارد، برای n های کوچک، زمان A_4 کم‌تر از دیگر الگوریتم‌هاست. در جدول ۳-۲ هم می‌توان دید که افزایش سرعت پردازنده تأثیر چندانی در زمان اجرای الگوریتم‌های با درجه‌ی پیچیدگی زیاد ندارد.

توجه کنید که ضریب ثابت تابع زمان اجرا برای هر الگوریتم به نوعی با تعداد دستورهای برنامه‌ای که از آن الگوریتم استفاده می‌کند متناسب است. یعنی پیاده‌سازی A_1 از همه‌ی الگوریتم‌ها مشکل‌تر و پیاده‌سازی A_4 خیلی ساده است. این نکته با این شهود سازگار است که برای آن‌که یک برنامه سریع شود باید از الگوریتمی که پیاده‌سازی آن مفصل‌تر است استفاده کرد. مثلاً، پیاده‌سازی A_4 خیلی ساده است و با این‌که این الگوریتم برای اندازه‌ی ورودی کوچک از همه سریع‌تر است، اما در عمل، برای n های بزرگ‌تر قابل استفاده نیست و زمان مورد نیاز آن بسیار بسیار بیش‌تر از حد تحمل کاربر است.

اما می‌دانیم الگوریتمی خوب است که مسئله را برای اندازه‌ی ورودی بزرگ در زمان کوتاهی حل کند. بنابراین از عددهای کوچک n صرف‌نظر می‌کنیم و ملاک مقایسه را n های بزرگ می‌گیریم. برای این‌کار کافی است که فقط آهنگ رشد تابع زمان اجرا را در نظر بگیریم. با این توضیح، الگوریتم A_1 از بقیه‌ی الگوریتم‌ها بهتر است، چون یک مقدار ثابت $n_0 > 0$ پیدا می‌شود که زمان اجرای A_1 برای همه‌ی مقادیر $n > n_0$ از زمان اجرای هر یک از الگوریتم‌های دیگر کم‌تر است.

برای بررسی دقیق‌تر این مطلب، تابع‌های رشد را تعریف می‌کنیم.

تمرین‌های بخش ۳-۲

۳-۲.۱ در جدول زیر، در هر سطر یک تابع $f(n)$ نوشته‌شده است که زمان اجرای یک الگوریتم را برحسب میکروثانیه نشان می‌دهد. به‌ازای هر زمان t که در ستون‌ها آمده است، و هر تابع $f(n)$، بزرگ‌ترین اندازه‌ی مسئله‌ای (n) را مشخص کنید که آن الگوریتم، مسئله را در زمان t حل می‌کند.

	۱ ثانیه	۱ دقیقه	۱ ساعت	۱ روز	۱ ماه	۱ سال	۱ قرن
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
$n^۲$							
$n^۳$							
۲^n							
$n!$							

۳-۲.۲ آرایه‌ی $S[۱ \cdots n]$ از اعداد صحیح به‌صورت صعودی مرتب شده است. می‌خواهیم ببینیم آیا i و j ای وجود دارند که $s[i] + s[j] = k$ بشود. برای این کار الگوریتم زیر پیش‌نهاد شده است:

```
1  i ← 1
2  j ← n
3  while i ≤ j
4      do if s[i] + s[j] = k
5             then return i, j
6          if s[i] + s[j] < k
7             then i ← i + 1
8          if s[i] + s[j] > k
9             then j ← j − 1
10 return 'No Solution'
```

درستی این الگوریتم را ثابت کنید و زمان اجرای آن‌ها را برحسب n به‌دست آورید.

۳-۳ تابع‌های رشد

همان‌طور که در بخش قبل بیان شد، برای بررسی درجه‌ی خوبی یک الگوریتم به آهنگ رشد منحنی زمان اجرا یا منحنی میزان حافظه‌ی مصرفی برحسب اندازه‌ی ورودی مسئله توجه می‌کنیم. برای بررسی دقیق‌تر از «تابع‌های رشد[۱۲]» استفاده می‌کنیم که با نمادهای $\mathcal{O}$، Θ، Ω، o و ω نمایش داده می‌شوند. بعضی از گزاره‌هایی که در این رابطه مطرح می‌شوند، به‌قرار زیرند:

- زمان اجرای الگوریتم مرتب‌سازی درجی در بدترین حالت و در حالت میانگین $T(n) = \Theta(n^۲)$ یا از مرتبه‌ی دقیق $n^۲$ است. یعنی آهنگ رشد آن برابر $n^۲$ است.
- الگوریتم «مرتب‌سازی هرمی[۱۳]» از مرتبه‌ی $n \lg n$ یا $\mathcal{O}(n \lg n)$ است.
- هر الگوریتم مرتب‌سازی مقایسه‌ای از مرتبه‌ی $\Omega(n \lg n)$ است.

تعریف چند تابع رشد

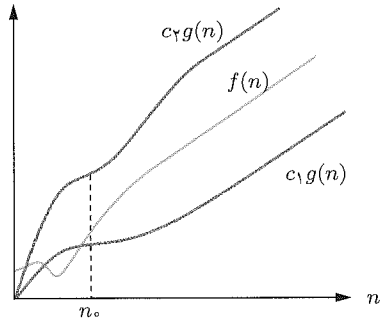
برای تابع داده‌شده‌ی $g(n)$، $\Theta(g(n))$ را به‌صورت مجموعه‌ی تابع‌های زیر تعریف می‌کنیم:

تعریف ۳-۱ (نماد Θ)

$$\Theta(g(n)) = \{f(n) : \exists c_۱, c_۲ > ۰ \text{ and } n_۰, \text{ such that } \forall n \geq n_۰, \ ۰ \leq c_۱ g(n) \leq f(n) \leq c_۲ g(n)\} \qquad (۳-۶)$$

یعنی مقادیر ثابت و مثبت $c_۱$، $c_۲$ و عدد صحیح $n_۰ > ۰$ وجود دارند که برای همه‌ی مقادیر $n > n_۰$ داریم $c_۱ g(n) \leq f(n) \leq c_۲ g(n)$. این رابطه به این معنی است که آهنگ رشد f و g برای مقادیر بزرگ n یک‌سان است و هیچ‌یک از این دو تابع از دیگری جلو نمی‌زنند. شکل ۳-۴ مثالی از این تابع رشد را نمایش می‌دهد. برای نشان دادن دقیق این مفهوم،

[۱۲] growth functions
[۱۳] heapsort



$f(n) = \Theta(g(n))$

شکل ۳-۴ نمایشی از تابع رشد Θ.

می‌نویسیم

$$f(n) \in \Theta\big(g(n)\big) \tag{۳-۷}$$

و می‌گوییم که تابع $g(n)$ «کران بسته‌ی مجانبی[۱۴]» برای $f(n)$ است. برای سادگی معمولاً نمایش زیر به‌کار برده می‌شود:

$$f(n) = \Theta\big(g(n)\big). \tag{۳-۸}$$

مثلاً دیدیم که زمان اجرای مرتب‌سازی درجی مطابق معادله‌ی ۳-۲ است. بنابراین، $T(n) = \Theta(n^2)$، زیرا می‌توان به‌سادگی مقادیر مثبتی برای c_1 و c_2 (مثلاً $a = c_1$ و $c_2 = a + b + c$) یافت که $c_1 n^2 \le an^2 + bn + c \le c_2 n^2$.

مثال ۳-۱ ثابت کنید که $100n^2 + 5n - 4 = \Theta(n^2)$.

اثبات: روشن است که اگر $c_1 = 1$ و $c_2 = 200$، برای هر مقدار $n \ge 1$ داریم

$$n^2 \le 100n^2 + 5n - 4 \le 200n^2.$$

مثال ۳-۲ نشان دهید که $100n^2 + 5n - 4 \ne \Theta(n^3)$.

[۱۴] asymptotically tight bound

حل: باید نشان داد که نمی‌توان مقادیر مثبتی برای c_1، c_2 و n_0 یافت که رابطه‌ی $c_1 n^3 \leq 100n^2 + 5n - 4 \leq c_2 n^3$ برای همه‌ی مقادیر $n \geq n_0$ برقرار باشد. به‌وضوح به‌ازای هر مقدار $c_1 > 0$ و برای مقادیر بزرگ n داریم $c_1 n^3 \not\leq 100n^2 + 5n - 4$.

لم ۳-۱ یک تابع چندجمله‌ای، از مرتبه‌ی دقیقِ جمله‌ی با بزرگ‌ترین توانش است.
یعنی با فرض $k > 0$ و $a_k > 0$ داریم

$$a_k n^k + a_{k-1} n^{k-1} + \cdots = \Theta(n^k)$$

در مورد مرتب‌سازی درجی با زمان اجرای $T(n)$ داریم

$$T(n) = \begin{cases} \Theta(n^2) \\ \Theta(100n^2) \end{cases} \neq \begin{cases} \Theta(n^3) \\ \Theta(n^2 \lg n) \\ \Theta(n \lg n) \end{cases}$$

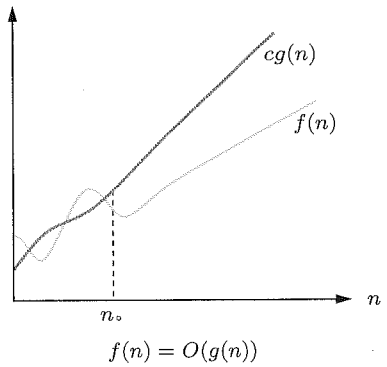
برای اثبات $T(n) \neq \Theta(n \lg n)$ کافی است نشان دهیم که نمی‌توان ثابت‌های c_1، c_2 و n_0ای یافت به‌طوری‌که برای هر $n > n_0$، رابطه‌ی $c_1 n \lg n \leq n^2 \leq c_2 n \lg n$ برقرار باشد.

تعریف ۳-۲ (نماد $\mathcal{O}$)
برای تابع داده‌شده‌ی $g(n)$، $\mathcal{O}(g(n))$ را به صورت مجموعه‌ی تابع‌های زیر تعریف می‌کنیم:

$$\mathcal{O}(g(n)) = \{f(n) : \exists c > 0 \text{ and } n_0, \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\} \qquad (۳-۹)$$

یعنی آهنگ رشد تابع $g(n)$ برای مقادیر بزرگ n، بیش‌تر یا مساوی آهنگ رشد تابع $f(n)$ است. در این صورت، $g(n)$ «کران بالای مجانبی[۱۵]» برای $f(n)$ است. شکل ۳-۵ مثالی از این تابع رشد را نمایش می‌دهد.

[۱۵] asymptotically upper bound



شکل ۳-۵ نمایشی از تابع رشد $\mathcal{O}$.

مثلاً در مورد مرتب‌سازی درجی، داریم

$$T(n) = \Theta(n^۲) = \begin{cases} \mathcal{O}(n^۲) \\ \mathcal{O}(۱۰۰n^۲) \\ \mathcal{O}(n^۳) \\ \mathcal{O}(n^۲ \lg n) \end{cases} \neq \begin{cases} \mathcal{O}(n \lg n) \\ \mathcal{O}(n) \end{cases}$$

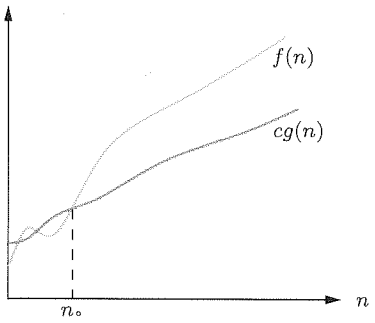
توجه کنید که می‌توان گفت مسئله‌ی مرتب‌سازی درجی مثلاً از $\mathcal{O}(n^{۱۰۰})$ است، ولی این اطلاع مفیدی نیست. به‌همین جهت در حالت‌هایی که محاسبه‌ی Θ مشکل است، یا آهنگ رشد زمان اجرا با ترکیب ورودی تغییر می‌کند، باید تابع داخل پرانتز $\mathcal{O}$ تا حد امکان کوچک باشد.

نکته‌ی ۳-۱ طبق تعریف، $\Theta(g(n)) \subseteq \mathcal{O}(g(n))$.

تعریف ۳-۳ (نماد Ω)

برای تابع داده‌شده‌ی $g(n)$، $\Omega(g(n))$ را به‌صورت مجموعه‌ی تابع‌های زیر تعریف می‌کنیم:

$$\Omega(g(n)) = \{f(n) : \exists c > \circ \text{ and } n_\circ, \text{ such that } \forall n \geq n_\circ, \ \circ \leq cg(n) \leq f(n)\} \qquad (۱۰-۳)$$



$f(n) = \Omega(g(n))$

شکل ۳-۶ نمایشی از تابع رشد Ω.

یعنی آهنگ رشد تابع $g(n)$ برای مقادیر بزرگ n کندتر یا مساوی آهنگ رشد $f(n)$ است. در این صورت، $g(n)$ «کران پایین مجانبی[۱۶]» برای $f(n)$ است. شکل ۳-۶ این تابع رشد را نمایش می‌دهد.

مثلاً برای مرتب‌سازی درجی داریم

$$T(n) = \begin{cases} \Omega(n \lg n) \\ \Omega(n) \\ \Omega(n^2) \end{cases} \neq \Omega(n^2 \lg n)$$

نماد Ω برای نشان‌دادن کران پایین یک تابع دیگر هم به‌کار می‌رود. هم‌چنین تابع Θ در واقع عطف $\mathcal{O}$ و Ω است یعنی،

$$f(n) = \Theta\big(g(n)\big) \Leftrightarrow f(n) = \mathcal{O}\big(g(n)\big) \wedge f(n) = \Omega\big(g(n)\big).$$

به‌عبارت دیگر،

قضیه‌ی ۳-۱ شرط لازم و کافی برای $f(n) = \Theta(g(n))$ آن است که $f(n) = \mathcal{O}(g(n))$ و $f(n) = \Omega(g(n))$.

[۱۶] asymptotically lower bound

دو نماد دیگر زیر نیز در همین رابطه تعریف شده‌اند.

تعریف ۳-۴ (نماد o)

$$o(g(n)) = \{f(n) | \forall c > 0, \exists n_0 > 0, \text{ such that } \forall n > n_0, \ 0 \leq f(n) < cg(n)\} \quad (۱۱-۳)$$

به‌عبارت دیگر، $f(n) = o(g(n))$. یعنی برای مقادیر بزرگ n، آهنگ رشد $f(n)$ از $g(n)$ اکیداً کم‌تر است.

نکته‌ی ۳-۲ اگر $f(n) = o(g(n))$، داریم: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

تعریف ۳-۵ (نماد ω)

$$\omega(g(n)) = \{f(n) | \forall c > 0, \exists n_0 > 0, \text{ such that } 0 \leq cg(n) < f(n)\} \quad (۱۲-۳)$$

به‌عبارت دیگر، $f(n) = \omega(g(n))$. یعنی برای مقادیر بزرگ n، آهنگ رشد $f(n)$ اکیداً از $g(n)$ بیش‌تر است. هم‌چنین رابطه‌ی ω با Ω مثل o با $\mathcal{O}$ است.

نکته‌ی ۳-۳ $f(n) = o(g(n))$ اگر و فقط اگر $g(n) = \omega(f(n))$.

نکته‌ی ۳-۴ اگر $f(n) = \omega(g(n))$، داریم: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

مثلاً $n^2/2 = \omega(n)$ ولی $n^2/2 \neq \omega(n^2)$.

مقایسه‌ی تابع‌های رشد

خاصیت تراگذری

تابع‌های رشد تعریف شده خاصیت تراگذری[۱۷] دارند. یعنی،

۱. از $f(n) = \Theta(g(n))$ و $g(n) = \Theta(h(n))$ نتیجه می‌شود $f(n) = \Theta(h(n))$.

۲. از $f(n) = \mathcal{O}(g(n))$ و $g(n) = \mathcal{O}(h(n))$ نتیجه می‌شود $f(n) = \mathcal{O}(h(n))$.

۳. از $f(n) = \Omega(g(n))$ و $g(n) = \Omega(h(n))$ نتیجه می‌شود $f(n) = \Omega(h(n))$.

۴. از $f(n) = o(g(n))$ و $g(n) = o(h(n))$ نتیجه می‌شود $f(n) = o(h(n))$.

۵. از $f(n) = \omega(g(n))$ و $g(n) = \omega(h(n))$ نتیجه می‌شود $f(n) = \omega(h(n))$.

خاصیت بازتابی[۱۸]

$f(n) = \Theta(f(n))$، $f(n) = \mathcal{O}(f(n))$ و $f(n) = \Omega(f(n))$.

خاصیت تقارن[۱۹]

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

خاصیت تقارن ترانهاده[۲۰]

۱. شرط لازم و کافی برای $f(n) = \mathcal{O}(g(n))$ آن است که $g(n) = \Omega(f(n))$.

۲. شرط لازم و کافی برای $f(n) = o(g(n))$ آن است که $g(n) = \omega(f(n))$.

نتیجه‌ای را که از این تعریف‌ها به‌دست می‌آید در این عبارت‌ها (هرچند از نظر ریاضی نادقیق) خلاصه می‌کنیم. به‌صورت شهودی، اگر F آهنگ رشد f، و G آهنگ رشد g باشد،

---

[۱۷] transitive
[۱۸] reflexive
[۱۹] symmetry
[۲۰] transpose symmetry

$$F = G \iff g = \Theta(f) \text{ یا } f = \Theta(g)$$
$$F \leq G \iff g = \Omega(f) \text{ یا } f = \mathcal{O}(g)$$
$$F < G \iff g = w(f) \text{ یا } f = o(g)$$
$$F \geq G \iff g = \mathcal{O}(f) \text{ یا } f = \Omega(g)$$
$$F > G \iff g = o(f) \text{ یا } f = \omega(g)$$

نکته‌ی ۳-۵ با استفاده از روابط پیوست ۲ نتایج زیر به‌سادگی ثابت می‌شوند:

$$n! = o(n^n), \quad (۱۳-۳)$$
$$n! = \omega(۲^n), \quad (۱۴-۳)$$
$$\lg(n!) = \Theta(n \lg n) \quad (۱۵-۳)$$

تمرین‌های بخش ۳-۳

۳-۳.۱ فرض کنید $f(n)$ و $g(n)$ توابعی به طور مجانبی نامنفی‌اند. با استفاده از تعریف نماد Θ، ثابت کنید $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

۳-۳.۲ نشان دهید که برای هر دو ثابت حقیقی a و b، که $b > ۰$، داریم $(n + a)^b = \Theta(n^b)$.

۳-۳.۳ آیا $۲^{n+۱} = \mathcal{O}(۲^n)$؟ $۲^{۲n} = \mathcal{O}(۲^n)$ چطور؟

۳-۳.۴ ثابت کنید که $o(g(n)) \cap \omega(g(n))$ تهی است.

۳-۳.۵ آیا تابع $\lceil \lg n \rceil!$ دارای کران چندجمله‌ای است؟ $\lceil \lg \lg n \rceil!$ چطور؟

۳-۳.۶ کدام تابع به‌طور مجانبی بزرگ‌تر است، $\lg(\lg^* n)$ یا $\lg^*(\lg n)$؟

۳-۳.۷ بدون استفاده از تقریب استرلینگ ثابت کنید $\lg(n!) = \Theta(n \lg n)$.

۳-۳.۸ ثابت کنید $\sum_{i=۱}^{n} \sqrt{i} = ۱ + \sqrt{۲} + \sqrt{۳} + \cdots + \sqrt{n} = \Theta(n\sqrt{n})$.

۳-۳.۹ با فرض $T_۱(n) = \Omega(f(n))$ و $T_۲(n) = \Omega(g(n))$، کدام‌یک از رابطه‌های زیر درست است؟

الف) $T_۱(n) + T_۲(n) = \Omega(\max\{f(n), g(n)\})$

ب) $T_۱(n) T_۲(n) = \Omega(f(n) g(n))$

۳-۳.۱۰ a_1 تا a_k چه شرایطی داشته باشند تا $T(n) = \Sigma_{i=1}^{k} T(n/a_i) + \Theta(n)$ از مرتبه‌ی $\Theta(n)$ باشد؟

۳-۳.۱۱ روشی برای مرتب‌سازی اعداد ذخیره‌شده در آرایه‌ی A در نظر بگیرید که در آن ابتدا کوچک‌ترین عنصر را پیدا و جای آن را با $A[1]$ عوض می‌کنیم. سپس دومین عنصر کوچک‌تر در A را یافته و جای آن را با $A[2]$ عوض می‌کنیم. این کار را برای $n-1$ عنصر اول A تکرار می‌کنیم. این الگوریتم به «مرتب‌سازی انتخابی[21]» معروف است.

الف) شبه‌کدی برای این الگوریتم بنویسید.

ب) رابطه‌ی مستقل از حلقه در این الگوریتم چیست؟

پ) آیا این الگوریتم پایدار است (یعنی ترتیب نسبی عناصر با کلید یک‌سان تغییر نمی‌کند)؟

ت) چرا به‌جای این‌که الگوریتم را روی همه‌ی n عنصر آرایه اجرا کنیم، آن‌را تنها روی $n-1$ عنصر اول اجرا می‌کنیم؟

ث) با استفاده از نماد Θ، زمان اجرای مرتب‌سازی انتخابی را در بهترین و بدترین حالت بیان کنید.

۳-۳.۱۲ رَویه‌ی زیر قرار است مقدار a^n را محاسبه کند.

```
POWER (a, n)
1  x ← a; z ← 1; m ← n
2  while m > 0
3      do if m is even
4           then m ← ⌊m/2⌋
5                x ← x^2
6           else عبارت ۱
7                عبارت ۲
8  return z
```

الف) به‌جای «عبارت ۱» و «عبارت ۲» به‌ترتیب چه دستورهایی بگذاریم تا الگوریتم درست کار کند؟

ب) رابطه‌ی مستقل از حلقه‌ی این الگوریتم چیست؟

پ) تعداد دقیق تکرار سطر ۵ الگوریتم برحسب n چند تاست؟

ت) زمان اجرای الگوریتم چقدر است؟

۳-۳.۱۳ تابع‌های زیر را برحسب درجه‌ی رشدشان مرتب کنید. یعنی ترتیبی از این ۳۰ تابع g_1 تا g_{30} را طوری به‌دست آورید که $g_1 = \Omega(g_2)$، $g_2 = \Omega(g_3)$، و ... همچنین این تابع‌ها را در رده‌های

[21] selection sort

هم‌ارزی افراز کنید. دو تابع f_1 و f_2 که $f_1 = \Theta(f_2)$ در یک رده قرار می‌گیرند.

$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$	$2^{2^{n+1}}$
$(\frac{3}{2})^n$	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	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$2^{\sqrt{2\lg n}}$	n	2^n	$n\lg n$	$n^{2+\sin n}$	$\lg^*(n^n)$

۱۴.۳-۳ در جدول زیر، برای هر دو تابع A و B با پاسخ «بله» یا «خیر»، تعیین کنید که آیا $A = X(B)$ (X یکی از رابطه‌های جدول است). فرض کنید $k \geq 1$، $\alpha > 0$ و $c > 1$ ثابت هستند.

	A	B	$\mathcal{O}$	o	Ω	ω	Θ
الف	$\lg^k n$	n^α					
ب	n^k	c^n					
پ	$\sqrt{n}$	$n^{\sin n}$					
ت	2^n	$2^{\frac{n}{2}}$					
ث	$n^{\lg c}$	$c^{\lg n}$					
ج	$\lg(n!)$	$\lg(n^n)$					

۱۵.۳-۳ با فرض $k > 3$ و $\epsilon > 0$، برای هر حالت زیر، رابطه‌های A و B را با نمادهای رشد بیان کنید. (مثلاً اگر $A = \Omega(B)$، نماد Ω را برای آن حالت انتخاب کنید.) در هر مورد ادعای خود را ثابت کنید.

	A	B	$\mathcal{O}$	o	Ω	ω	Θ
الف	2^{n+k}	$2^n + 2^k - n^k$					
ب	$(\lg n)^3$	$\lg(n^{\lg n^3})$					
پ	$\lg^*(\lg n)$	$\lg^*(n^{n^{(n!)}})$					
ت	2^{kn}	$2^{n/\epsilon}$					
ث	k^n	$(3k)^{n/3}$					
ج	n^2	$n^{3\sin n}$					
چ	$(kn)^{1-\epsilon}$	$n^{1-\epsilon}$					
ح	$\sum_{i=1}^{kn} \frac{1}{i}$	$\lg n^k$					
خ	$(n\cos n)^2$	n^2					

۱۶.۳-۳ فرض کنید $p(n) = \sum_{i=0}^{d} a_i n^i$ ($a_d > 0$)، یک چندجمله‌ای از درجه‌ی d بر حسب n و k یک عدد ثابت باشد. با استفاده از تعاریف نمادهای مجانبی، ویژگی‌های زیر را ثابت کنید:

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

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

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

ت) اگر $k > d$، آن‌گاه $p(n) = o(n^k)$.

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

۳-۳.۱۷ فرض کنید $f(n)$ و $g(n)$ توابعی هستند که به‌طور مجانبی مثبت‌اند. هر یک از حدس‌های زیر را اثبات یا نقض کنید:

الف) $f(n) = \mathcal{O}(g(n))$ نتیجه می‌دهد $g(n) = \mathcal{O}(f(n))$.

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

پ) اگر برای n های به‌اندازه‌ی کافی بزرگ داشته باشیم $\lg(g(n)) \geq 1$، بنابراین $f(n) \geq 1$ نتیجه می‌دهد $\lg(f(n)) = \mathcal{O}(\lg(g(n)))$.

ت) $f(n) = \mathcal{O}(g(n))$ نتیجه می‌دهد $2^{f(n)} = \mathcal{O}(2^{g(n)})$.

ث) $f(n) = \mathcal{O}((f(n))^2)$.

ج) $f(n) = \mathcal{O}(g(n))$ نتیجه می‌دهد $g(n) = \Omega(f(n))$.

چ) $f(n) = \Theta(f(\frac{n}{2}))$.

ح) $f(n) + o(f(n)) = \Theta(f(n))$.

۳-۴ روش‌های تحلیل الگوریتم‌ها

چنان‌چه نشان داده‌ایم، الگوریتم‌ها به دو دسته‌ی اصلی تقسیم می‌شوند: ترتیبی و بازگشتی. الگوریتم‌هایی را که بازگشتی نیستند ترتیبی[۲۲] می‌گوییم. در این بخش با روش تحلیل هر کدام از این‌ها آشنا می‌شویم. الگوریتم‌های ترتیبی را معمولاً با شمارش تعداد دفعات اجرای دستوری که برحسب اندازه‌ی داده‌ی ورودی، بیش‌ترین بار اجرا می‌شود تحلیل می‌کنیم. اما زمان اجرا و میزان حافظه‌ی مصرفی الگوریتم‌های بازگشتی با رابطه‌های بازگشتی بیان می‌شوند که در این بخش به روش‌های مختلف حل این رابطه‌ها می‌پردازیم. به‌عنوان مثال، با مسئله‌ی برج هانوی و مرتب‌سازی ادغامی و حبابی و چند مسئله‌ی مشابه آشنا می‌شویم.

[۲۲] sequential

۳-۴-۱ تحلیل الگوریتم‌های ترتیبی

برای به‌دست آوردن زمان اجرای یک الگوریتم ترتیبی فرض می‌کنیم که اجرای هر «دستور ساده[۲۳]» به زمان ثابتی نیاز دارد. بنابراین اگر تعداد ثابتی دستور ساده پشت سر هم باشند، در مجموع زمانی ثابت می‌گیرند.

$T(n)$ زمان اجرای یک تکه‌برنامه‌ی $\mathcal{T}$، به صورت‌های زیر محاسبه می‌شود:

- با فرض ثابت بودن k، اگر $\mathcal{T}$ خود شامل k تکه‌برنامه با زمان‌های اجرای $T_1(n) = \mathcal{O}(f_1(n))$، $T_2(n) = \mathcal{O}(f_2(n))$، تا $T_k(n) = \mathcal{O}(f_k(n))$ باشد، در آن‌صورت

$$T(n) = \sum_{i=1}^{k} T_i(n) = \mathcal{O}(\max_{1 \le i \le k} (f_i(n)))$$

- اگر $\mathcal{T}$ ساختار حلقه داشته باشد (مانند **repeat**، **while** و **for** و امثال آن[۲۴]) و $g(n)$ بار تکه‌برنامه‌ای دیگر با زمان اجرای $S(n) = \mathcal{O}(f(n))$ را تکرار کند،

$$T(n) = g(n)S(n) = \mathcal{O}(g(n)f(n))$$

- اگر $\mathcal{T}$ ساختار **if** باشد و بخش‌های **then** و **else** آن به‌ترتیب زمان‌های $T_1(n) = \mathcal{O}(f_1(n))$ و $T_2(n) = \mathcal{O}(f_2(n))$ را داشته باشند،

$$T(n) = \mathcal{O}(\max\{f_1(n), f_2(n)\}).$$

البته معمولاً $T(n) = \max\{T_1(n), T_2(n)\}$، ولی ممکن است یکی از قسمت‌های **then** یا **else** هیچ‌گاه اتفاق نیفتد.

به‌طور شهودی، معمولاً مرتبه‌ی زمان اجرای یک الگوریتم، همان مرتبه‌ی تکه‌ای از برنامه است که بیش‌ترین زمان را می‌گیرد، چرا که همیشه برای تابع رشد، بدترین حالت مد نظر است.

---

[۲۳] simple statement

[۲۴] گاهی حلقه با goto هم ساخته می‌شود که تشخیص این موارد نیاز به دقت دارد.

البته این روش، کران بالای مقدار مورد تحلیل را به‌دست می‌آورد که آن‌را با نماد $\mathcal{O}$ نشان داده‌ایم. گاهی برای به‌دست آوردن پیچیدگی دقیق یا Θ باید از روش‌های پیچیده‌تری مثل «روش سرشکنی[۲۵]» استفاده کرد که در بخش ۳-۷ به‌آن می‌پردازیم.

مرتب‌سازی حبابی

«مرتب‌سازی حبابی[۲۶]» یک الگوریتم ساده برای مرتب‌سازی یک آرایه‌ی n عضوی است. این الگوریتم در مرحله‌ی i $(1 \leq i < n)$ با یک‌بار پیمایش از پایین به بالای بخشی از آرایه با درایه‌های n تا $i+1$، عناصر کوچک آن‌را مانند «حباب» به بخش‌های بالای (اندیس‌های کم‌تر) آرایه هدایت می‌کند. در نتیجه حتماً کوچک‌ترین عنصر این بخش هم به درایه‌ی i منتقل می‌شود. روشن است که با تکرار این کار، در انتها آرایه مرتب می‌شود. این الگوریتم در زیر آمده است.

```
BUBBLE-SORT (A)
    ▷ مرتب‌سازی حبابی
1   n ← length[A]
2   for i ← 1 to n − 1
3     do for j ← n downto i + 1
4           do if A[j] > A[j − 1]
5                 then SWAP (A[j], A[j − 1] )
```

زمان هر مقایسه و تعویض را ثابت در نظر می‌گیریم. در بدترین حالت، تعداد دقیق مقایسه‌ها و حداکثر تعداد تعویض‌ها برابر است با

$$T(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathcal{O}(1) = \Theta(n^2).$$

بدترین حالت هنگامی است که آرایه با عناصر مجزا در ابتدا برعکس مرتب شده باشد. بهترین حالت هنگامی است که آرایه از ابتدا مرتب باشد؛ در این صورت، هیچ

[۲۵] amortized
[۲۶] bubble sort

تعویضی صورت نمی‌گیرد و الگوریتم در زمان $\mathcal{O}(n)$ خاتمه می‌یابد. در حالت کلی تعداد تعویض‌های این الگوریتم متناسب است با تعداد «وارونگی[۲۷]» آن. این مطلب در تمرین ۳.۳ بررسی شده است. به‌این دلیل هم تحلیل میانگین این الگوریتم چندان ساده نیست و به خود شما واگذار می‌شود.

البته با توجه به مطالب بخش قبل، زمان اجرای این الگوریتم را در بدترین حالت به‌صورت نماد $\mathcal{O}$، بدون محاسبه می‌توان به‌دست آورد: زمان مقایسه و تعویض و در نتیجه زمان اجرای دستور **if** در سطر ۴ مقدار ثابتی است. تکرار حلقه‌ی داخلی در بدترین حالت از مرتبه‌ی n است. حلقه‌ی بیرونی نیز در بدترین حالت n بار تکرار می‌شود. بنابراین زمان کل حلقه برابر حاصل‌ضرب آن‌دو یعنی $\mathcal{O}(n^2)$ است.

در مورد اثبات درستی الگوریتم‌های ترتیبی هم چنان‌چه گفته شد، باید در هر مورد رابطه‌ی مستقل از حلقه را به‌دست آورد و با استقرا آن‌را اثبات کرد.

تمرین‌های زیربخش ۳-۴-۱

۳-۴-۱.۱ رویه‌ی زیر بررسی می‌کند که آیا عدد N اول است یا خیر. زمان اجرای آن چقدر است؟

```
IsPrime (N)
 1  i ← 3
 2  if N = 2
 3     then return true
 4  if N mod 2 = 0
 5     then return false
 6  while i^2 ≤ N
 7     do if N mod i = 0
 8           then return false
 9           else  i ← i + 2
10  return true
```

۳-۴-۱.۲ زمان اجرای الگوریتم‌های زیر را در بدترین حالت به‌صورت $\Theta()$ نشان دهید.

```
1  sum ← 1
2  for i ← 1 to n
3    do for j ← 1 to i^2
4         do if j mod i = 0
5              then for k ← 1 to j
6                   do sum ← sum + 1
```

[۲۷] inversion

```
1  i ← n
2  while i ≥ 1
3      do j ← i
4          while j ≤ n
5              do j ← 2j
6          i ← ⌊i/2⌋
```

۳-۴-۳.۱ مقدار *count* را در انتهای رویه‌ی زیر برحسب n به‌دست آورید.

```
MYSTRY (n)
1  count ← 0
2  x ← 2
3  while x < n
4      do x ← x * 2
5          count ← count + 1
6  print count
```

۳-۴-۴.۱ رویه‌ی زیر، عنصر بیشینه‌ی n عدد در آرایه‌ی A را از اندیس i به‌دست می‌آورد. یک کران بالای خوب برای تعداد کل اعمال مقایسه در سطر ۵ را در رَویه‌ی زیر به‌دست آورید.

```
MAX (A, i, n)
1  if n = 1
2     then return A[i]
3     else  m1 ← MAX (A, i, ⌊n/2⌋)
4           m2 ← MAX (A, i + ⌊n/2⌋, ⌊n/2⌋)
5           if m1 < m2
6              then return m2
7              else  return m1
```

۳-۴-۵.۱ در مورد مرتب‌سازی حبابی به سؤالات زیر پاسخ دهید:

الف) فرض کنید A' خروجی BUBBLESORT(A) باشد که در صفحه‌ی ۸۱ همین بخش آمد. برای اثبات این‌که BUBBLESORT درست کار می‌کند، لازم است ثابت کنیم که این شبه‌کد پایان می‌یابد و همچنین در انتها داریم، $(n = length[A])$

$$A'[1] \leq A'[2] \leq \ldots \leq A'[n] \qquad (۳-۱۶)$$

برای نشان دادن این‌که BUBBLESORT واقعاً مرتب می‌کند، اثبات چه حقیقت دیگری لازم است؟

در دو قسمت بعدی، نامساوی ۳-۱۶ ثابت می‌شود.

ب) رابطه‌ی مستقل از حلقه‌ی **for** در سطرهای ۳ تا ۵ را به‌طور دقیق بیان و برقراری آن را ثابت کنید.

پ) با استفاده از شرط پایان حلقه و ثابتی که برقراری آن را در قسمت قبل ثابت کرده‌اید، برای حلقه‌ی **for** در سطرهای ۲ تا ۵ یک رابطه‌ی مستقل از حلقه پیدا کنید که به اثبات نامساوی ۳-۱۶ کمک کند.

ت) زمان اجرای مرتب‌سازی حبابی در بدترین حالت چیست؟ آن را با مرتب‌سازی درجی مقایسه کنید.

۳-۴-۲ تحلیل الگوریتم‌های بازگشتی

معمولاً یک الگوریتم بازگشتی مسئله را به یک یا چند زیرمسئله‌ی کوچک‌تر تقسیم می‌کند، زیرمسئله(ها)ی کوچک‌تر را به‌صورت بازگشتی و با همان الگوریتم حل می‌کند و از نتیجه‌ی آن‌ها جواب مسئله را به‌دست می‌آورد. بنابراین، زمان اجرای الگوریتم، حاصل جمع زمان حل زیرمسئله(ها) به‌اضافه‌ی زمان لازم برای تقسیم مسئله به زیرمسئله(ها) و نیز ترکیب جواب(های) به‌دست آمده است. البته برخی از این زمان‌ها ممکن است ثابت باشد.

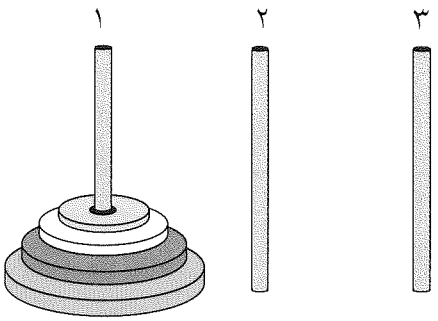
به‌عنوان مثال، با مسئله‌ی برج‌های هانوی و نیز الگوریتم مرتب‌سازی ادغامی آشنا می‌شویم و آن‌ها را به روش‌هایی که بیان خواهد شد تحلیل می‌کنیم.

مسئله‌ی برج‌های هانوی

در این مسئله‌ی قدیمی[۲۸] و آشنا، n سکه‌ی سوراخ‌دار با قطرهای متفاوت به‌ترتیب قطرشان در میله‌ی شماره‌ی ۱ از ۳ میله‌ی موجود قرار دارند (سکه‌ی بزرگ‌تر در پایین سکه‌هاست). شکل ۳-۷ مثالی از حالت اولیه‌ی این مسئله را با ۴ سکه نشان می‌دهد. هدف، انتقال همه‌ی سکه‌ها به میله‌ی شماره‌ی ۳ به‌کمک میله‌ی شماره‌ی ۲ است به‌طوری که،

- هر بار فقط یک سکه جابه‌جا شود،
- در هر حرکت سکه‌ی کوچک‌تر بر روی سکه‌ی بزرگ‌تر قرار گیرد، و

[۲۸] این معما در سال ۱۸۸۳ توسط ادوارد لوکاس، ریاضی‌دان فرانسوی مطرح شد.



شکل ۳-۷ مثالی از مسئله‌ی برج‌های هانوی.

- تعداد حرکت‌ها کمینه باشد.

کمینه‌ی تعداد جابه‌جایی سکه‌ها در این مسئله هنگامی رخ می‌دهد که بزرگ‌ترین سکه مستقیماً به میله‌ی مقصد منتقل شود. برای این کار لازم است که حتماً بقیه‌ی سکه‌ها روی میله‌ی ۲ باشند، و این کار نیز خود به‌صورت بازگشتی و با کمینه‌ی تعداد حرکت‌ها انجام شود. در این کار نیز بزرگ‌ترین سکه نقشی ندارد و می‌تواند نادیده گرفته شود. پس از انتقال سکه‌ی بزرگ‌تر، بقیه‌ی سکه‌ها ($n-1$ عدد) مجدداً به صورت بازگشتی از میله‌ی ۲ به میله‌ی مقصد منتقل می‌شوند. توجه کنید که با این الگوریتم، کلیه‌ی قواعد بازی رعایت شده و نمی‌توان الگوریتمی با تعداد حرکت کم‌تر از این برای این مسئله پیدا کرد.

رویه‌ی TOWER-OF-HONOI این الگوریتم را برای انتقال n سکه از میله‌ی f به میله‌ی t و با کمک میله‌ی h نشان می‌دهد.

```
TOWER-OF-HONOI (n, f, t, h)
    ▷ حرکت n سکه از میله‌ی f به میله‌ی t به کمک میله‌ی h و طبق قواعد برج هانوی
1   if n = 1
2      then Move the coin from leg f to leg t
3      else TOWER-OF-HONOI(n − 1, f, h, t)
4           Move the coin from leg f to leg t
5           TOWER-OF-HONOI(n − 1, h, t, f)
```

با توجه به این‌که الگوریتم بازگشتی است، اثبات درستی آن نیز با استفاده از استقرا انجام می‌شود.

پایه‌ی استقرا: $n = ۱$، روشن است.

فرض استقرا: برای $n - ۱$ سکه الگوریتم درست کار می‌کند، یعنی فراخوانی‌های بازگشتی فوق به‌درستی $n - ۱$ سکه را جابه‌جا می‌کنند.

گام استقرا: با توجه به این‌که وجود بزرگ‌ترین سکه تأثیری در حرکت بقیه‌ی سکه‌ها ندارد، و هیچ حرکت غیر قانونی انجام نمی‌شود، با استفاده از فرض استقرا، مشاهده می‌کنیم که الگوریتم به‌درستی n سکه را هم جابه‌جا می‌کند.

اگر $T(n)$ کمینه‌ی تعداد حرکت‌ها برای n سکه باشد، داریم

$$T(n) = \begin{cases} ۱, & n = ۱ \\ T(n-۱) + ۱ + T(n-۱), & n > ۱ \end{cases} . \qquad (۱۷\text{-}۳)$$

به چنین عبارتی، یک «رابطه‌ی بازگشتی[۲۹]» می‌گوییم. این رابطه را با روش‌هایی که در همین بخش اشاره می‌شود حل می‌کنیم و نشان می‌دهیم که جواب آن $T(n) = ۲^n - ۱$ است.

حدس $T(n) = ۲^n - ۱$ برای جواب دقیق رابطه‌ی ۳-۱۷ خیلی مشکل نیست. اثبات این حدس هم با استقرا بسیار ساده است. در این مورد بیش‌تر صحبت خواهیم کرد.

راه‌حل غیر بازگشتی مسئله‌ی برج‌های هانوی

جالب است بدانید که مسئله‌ی برج هانوی یک راه‌حل غیر بازگشتی ساده نیز دارد. میله‌ها را مطابق شکل ۳-۸ بر روی یک دایره می‌چینیم و فرض می‌کنیم میله‌ی ۱ مبدأ و میله‌ی ۳ مقصد است. الگوریتم به‌صورت زیر است:

۱. اگر n زوج است «جهت حرکت» را ساعت‌گرد و گرنه پادساعت‌گرد فرض کن.

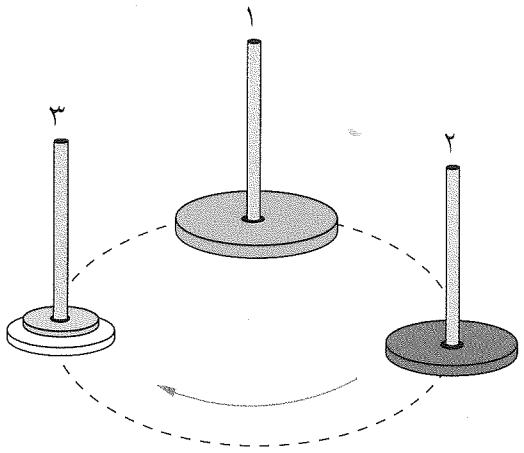
۲. تکرار کن تا همه‌ی سکه‌ها در میله‌ی مقصد قرار گیرند.

الف) کوچک‌ترین سکه را به‌میله‌ی بعدی در جهت حرکت ببر، و

ب) در صورت امکان تنها حرکتی را که شامل کوچک‌ترین سکه نباشد انجام بده.

---

[۲۹] recurrence relation



شکل ۳-۸ راه‌حل غیر بازگشتی مسئله‌ی برج‌های هانوی.

اثبات درستی این الگوریتم به‌سادگی الگوریتم بازگشتی نیست، به‌خصوص آن‌که باید ثابت کنیم که تعداد حرکت‌ها کمینه است. همین مثال نشان می‌دهد که چرا سعی می‌کنیم در صورت امکان الگوریتم‌ها را به‌صورت بازگشتی طراحی کنیم.

برای اثبات درستی، باز از استقرا استفاده می‌کنیم. ثابت می‌کنیم که با مشخص شدن جهت حرکت، این الگوریتم n سکه‌ی موجود در میله‌ی ۱ را با تعداد کمینه‌ی حرکت‌ها به‌درستی به میله‌ی مقصد می‌برد. اگر n فرد باشد، میله‌ی مقصد، میله‌ی بعدی در جهت حرکت (یعنی میله‌ی ۲) و اگر n زوج باشد، میله‌ی مقصد میله‌ی قبلی در جهت حرکت (یعنی میله‌ی ۳) خواهد بود. هم‌چنین تعداد حرکت‌ها $۲^n - ۱$ است.

برای سادگی بیان اثبات، فرض می‌کنیم جهت حرکت ساعت‌گرد است و n سکه قرار است از میله‌ی ۱ به میله‌ی مقصد (میله‌ی ۳ برای n زوج و میله‌ی ۲ برای n فرد) بروند.

پایه‌ی استقرا: $n = ۱$، روشن است و این‌کار با یک حرکت انجام می‌شود.

فرض استقرا: برای $n - ۱$ سکه الگوریتم درست کار می‌کند. یعنی $n - ۱$ سکه را به‌صورت مستقل و با $۲^{n-۱} - ۱$ حرکت از میله‌ی ۱ به میله‌ی بعدی در جهت ساعت‌گرد (برای $n - ۱$ زوج) یا قبلی (برای $n - ۱$ فرد) به‌درستی و مطابق مقررات مسئله منتقل می‌کند.

گام استقرا: فرض کنید n زوج است. این الگوریتم مطابق فرض استقرا پس از $۲^{n-۱} - ۱$ حرکت $n - ۱$ سکه‌ی بالای میله‌ی مبدأ را به میله‌ی بعدی در جهت ساعت‌گرد (یعنی میله‌ی ۲) منتقل می‌کند. در این حرکت‌ها بزرگ‌ترین سکه نقشی ندارد و در انتها این سکه در میله‌ی ۱ باقی می‌ماند. سپس تنها حرکت ممکن را انجام می‌دهد که بزرگ‌ترین سکه را

از میله‌ی ۱ به ۳ می‌برد. سپس با توجه به جهت حرکت که ساعت‌گرد است، کاری که الگوریتم به‌صورت استقرایی انجام می‌دهد، حرکت $n-1$ سکه‌ی موجود در میله‌ی ۲ به میله‌ی بعدی (یعنی ۳) است. و این کار با فرض استقرا پس از $۲^{n-1}-1$ حرکت دیگر به‌درستی انجام می‌شود. مشابه این استدلال برای nهای فرد هم برقرار است. جمع کل حرکت‌ها $۲(۲^{n-1}-1)+1=۲^n-1$ است. اگر طبق فرض استقرا تعداد حرکت‌های زیرمسئله‌ها بهینه باشد، با توجه به این‌که برای حرکت بزرگ‌ترین سکه لازم است همه‌ی $n-1$ سکه به میله‌ی ۲ بروند تا امکان حرکت این سکه به‌طور مستقیم به میله‌ی مقصد فراهم آید، پس تعداد کل حرکت‌های این الگوریتم هم کمینه است.

مثال ۳-۳ نوع دیگری از مسئله‌ی برج هانوی را با ۲^k سکه‌ی سوراخ‌دار با شماره‌های ۱ تا ۲^k در نظر بگیرید. $k+۲$ میله هم با شماره‌های صفر تا $k+۱$ در یک ردیف پشت سرهم قرار دارند. ابتدا، سکه‌ها با یک ترتیب دل‌خواه در میله‌ی صفر روی هم قرار دارند. در هر حرکت، می‌توان بالاترین سکه‌ی موجود در میله‌ی i را برداشت و در بالای میله‌ی $(i+۱)$ ام قرار داد $(۰ \le i \le k)$. به این حرکت T_i می‌گوییم. حالتی نهایی است که در آن تمام سکه‌ها به‌ترتیب از شماره‌ی ۱ تا ۲^k بر روی میله‌ی آخر قرار بگیرند و پایین‌ترین سکه، شماره‌ی ۱ باشد. دقت کنید که در این مسئله بجز در آخرین میله، سکه‌ها می‌توانند با هر ترتیبی روی هم قرار بگیرند. برای مثال، به‌ازای $k=۱$، حرکت‌های لازم برای رسیدن به حالت نهایی، از یک حالت اولیه‌ی مرتب (از پایین ۱ و ۲) از چپ‌به‌راست به صورت $\langle T_۰, T_۰, T_۱, T_۱\rangle$ است. ثابت کنید به‌ازای هر k می‌توان با هر ترتیب اولیه، سکه‌ها را مرتب کرد.[۳۰]

اثبات: برای $k=۰$ حکم بدیهی است. حال با فرض این‌که مسئله برای $k=m$ حل شده است، درستی آن‌را برای $k=m+۱$ ثابت می‌کنیم. در این‌صورت، $m+۳$ میله و $۲^{m+۱}$ سکه داریم. اگر سکه‌های شماره‌ی $۱+۲^m$ تا $۲^{m+۱}$ وجود نداشتند، می‌توانستیم سکه‌های شماره‌ی ۱ تا ۲^m را بدون استفاده از میله‌ی ۱ به میله‌ی $m+۲$ منتقل کنیم. دنباله‌ی این حرکات را $\mathcal{T}$ می‌نامیم. ولی با وجود این سکه‌ها و میله‌ی شماره‌ی ۱، می‌توان مشکل را به این‌صورت حل کرد: هرگاه در $\mathcal{T}$، نوبت حرکت یک سکه‌ی ۱ تا ۲^m باشد و سکه‌هایی با شماره‌ی $۱+۲^m$ تا $۲^{m+۱}$ بالاتر از آن روی میله‌ی صفر قرار داشته باشند، ابتدا آن سکه‌ها را به میله‌ی ۱ منتقل می‌کنیم و بعد حرکت مورد نظر را انجام می‌دهیم. توجه کنید که در حرکات مورد نظر، هر حرکت $T_i \in \mathcal{T}$ به حرکت $T_{i+۱}$ $(i \ge ۲)$ و هر حرکت $T_۱ \in \mathcal{T}$ به حرکت‌های لازم برای انتقال سکه‌های با شماره‌ی $۱+۲^m$ تا $۲^{m+۱}$ به میله‌ی ۱ و سپس حرکت $T_۱$ و $T_۲$ تبدیل می‌شود.

[۳۰] این مسئله‌ی ۶، از مرحله‌ی دوم هفتمین المپیاد کامپیوتر ایران بود.

حال سکه‌های با شماره‌ی ۱ تا ۲^m به‌صورت مرتب و با ترتیب درست روی میله‌ی $m+۲$ام قرار گرفته‌اند. ۲^m سکه با شماره‌های $۱ + ۲^m$ تا $۲^{m+۱}$ را که روی میله‌ی ۱ قرار گرفته‌اند، طبق فرض استقرا به میله‌ی شماره‌ی $m+۳$ منتقل می‌کنیم (زیرا $m+۲$ میله داریم). در نهایت، سکه‌ها از شماره‌ی ۱ تا $۲^{m+۱}$ به‌ترتیب (از پایین به بالا) روی میله‌ی شماره‌ی $m+۳$ قرار خواهند گرفت و حکم ثابت است. □

مرتب‌سازی ادغامی

به‌عنوان مثالی دیگر، الگوریتم «مرتب‌سازی ادغامی[۳۱]» را در نظر می‌گیریم. این الگوریتم آرایه‌ی A را از اندیس p تا r به روش تقسیم‌وحل مرتب می‌کند. فراخوانی اصلی $p = ۱$ و $r = n$ (n تعداد عناصر آرایه) است. رویه‌ی MERGE-SORT این‌کار را انجام می‌دهد .

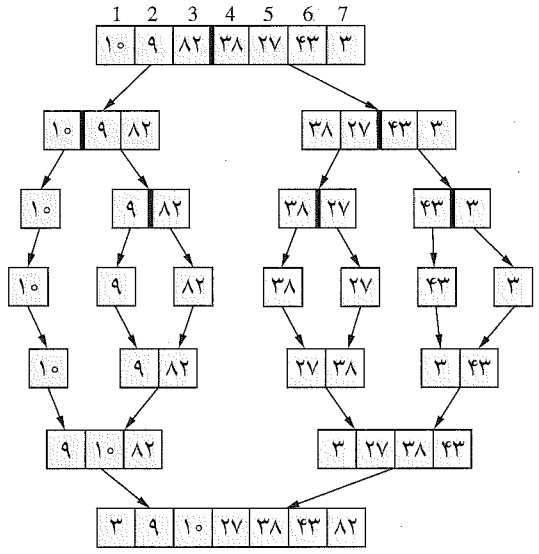
```
MERGE-SORT (A, p, r)
    ▷ A: آرایه‌ای که قرار است بین دو اندیس زیر مرتب شود
    ▷ p: اندیس شروع
    ▷ r: اندیس پایانی
1   if p < r
2      then q ← ⌊(p+r)/2⌋
3           MERGE-SORT(A, p, q)
4           MERGE-SORT(A, q + 1, r)
5           MERGE (A, p, q, r)
```

این الگوریتم، اگر بخش مورد نظر (از p تا r) بیش از یک عنصر داشت (یعنی $p < r$)، آن‌را به دو نیمه با اندازه‌های تقریباً برابر (با اختلاف حداکثر ۱ عنصر) تقسیم می‌کند. برای این‌کار مرز این دو نیمه یا q را در سطر ۲ پیدا می‌کند و هر نیمه را به‌صورت بازگشتی مرتب می‌کند. با ادغام دو نیمه‌ی مرتب، کار را تمام می‌کند.

پس این مرتب‌سازی در واقع مجموعه‌ای از ادغام‌های بخش‌های مختلف یک آرایه است. قبل از آن‌که جزئیات ادغام را بررسی کنیم، به شکل ۳-۹ نگاه کنید که مراحل مختلف اجرای این الگوریتم را بر روی یک آرایه‌ی ۷ عضوی نشان می‌دهد.

---

[۳۱] merge sort



شکل ۳-۹ مراحل اجرای مرتب‌سازی ادغامی بر روی یک آرایه با ۷ عنصر.

اگر مجاز باشیم از حافظه‌ی اضافی (متناسب با اندازه‌ی دو بخش) استفاده کنیم، ادغام دو بخش مرتب و مجاور از یک آرایه کار ساده‌ای است، اما اگر فقط بتوانیم به اندازه‌ی ثابتی حافظه‌ی اضافی مصرف کنیم، ادغام کار پیچیده‌ای ــ هرچند امکان‌پذیر ــ است. به‌حالت دوم «ادغام درجا[۳۲]» می‌گوییم. در این بخش، ادغام را با حافظه‌ی اضافی در نظر می‌گیریم.

مسئله را به این صورت بیان می‌کنیم: می‌خواهیم دو آرایه‌ی مرتب و مستقل A با n عنصر و B با m عنصر را در هم ادغام کنیم و حاصل را در یک آرایه‌ی دیگر به‌نام C با $n+m$ عنصر بنویسیم. برای این‌کار، سه اندیس i و j و k درنظر می‌گیریم که i و j به‌ترتیب اندیس‌های اولین درایه‌های A و B هستند که هنوز در C کپی نشده‌اند. هر بار عنصر جدید در $C[k]$ نوشته می‌شود. در ابتدا، $i = j = ۱$ و $k = ۰$. در هر مرحله، اگر عنصری برای کپی‌کردن در C داشته باشیم، مقدار k را یک واحد اضافه می‌کنیم تا محل کپی‌شدن C به‌دست آید و از بین $A[i]$ و $B[j]$ کوچک‌ترین آن‌دو را در $C[k]$ کپی می‌کنیم. اگر این عنصر از A کپی شده باشد، یک واحد به i و اگر از B آمده باشد، یک واحد به j اضافه می‌کنیم تا در مرحله‌ی بعد هم بتوانیم همین کار را تکرار کنیم. البته حالت‌هایی را که $i > n$ یا $j > m$ شود نیز باید در نظر بگیریم. رویه‌ی MERGE-1 این‌کار را انجام می‌دهد.

[۳۲] in-place merge

```
MERGE-1 (A, B)
 1  n ← length[A];     m ← length[B]
 2  i ← 1;     j ← 1;     k ← 0
 3  while i ≤ n or j ≤ m
 4      do k ← k + 1
 5          if j > m  or ((i ≤ n)  and  (A[i] ≤ B[j]))
 6            then C[k] ← A[i]
 7                 i ← i + 1
 8            else  C[k] ← B[j]
 9                  j ← j + 1
10  length[C] ← n + m
11  return C
```

در این‌جا فرض می‌کنیم که در شرط X **OR** Y ابتدا شرط X بررسی می‌شود و اگر درست بود، دیگر شرط Y بررسی نمی‌شود[۳۳] (چون ممکن است درایه‌های مورد نظر وجود نداشته باشند). همچنین در مورد X **and** Y، که اگر اولی غلط باشد دومی اصلاً بررسی نمی‌شود.

رویه‌ی MERGE-2 یک راه‌حل ساده‌تر برای ادغام دو آرایه‌ی مرتب است.

```
MERGE-2 (A, B)
 1  n ← length[A];     m ← length[B]
 2  A[n + 1] ← ∞;      B[m + 1] ← ∞
 3  i ← 1;     j ← 1
 4  for k ← 1 to n + m
 5    do if A[i] < B[j]
 6          then C[k] ← A[i]
 7               i ← i + 1
 8          else  C[k] ← B[j]
 9               j ← j + 1
10  length[C] ← n + m
11  return C
```

[۳۳] در زبان برنامه‌نویسی به این‌کار «میان‌بر زدن» می‌گوییم.

رویه‌ی MERGE-1 با حداکثر $۲(m+n)-۱$ و حداقل $m+n-۱$ عدد مقایسه بین عناصر A و B، این دو آرایه را به‌درستی در هم ادغام می‌کند و حاصل را در آرایه‌ی C می‌نویسد. این الگوریتم از نظر تعداد مقایسه‌ها بهینه است.

مراحل مختلف رویه‌ی MERGE-2 هم در شکل ۳-۱۰ نشان داده شده است.

برای آن‌که از یکی از رویه‌های فوق در الگوریتم مرتب‌سازی ادغامی استفاده کنیم باید یک آرایه‌ی سراسری C تعریف و به‌طرز مناسبی پارامترهای رویه‌ی MERGE را اصلاح کنیم. در هر صورت، الگوریتم مرتب‌سازی درجا نیست. این نکته هم شایان ذکر است که می‌توان ادغام را طوری نوشت که مرتب‌سازی پایدار شود.

اثبات درستی الگوریتم مرتب‌سازی ادغامی به شکل استقرایی صورت می‌گیرد، یعنی برای پایه‌ی استقرا یا $n=۱$ اثبات بدیهی است و برای $n>۱$ درستی آن به‌سادگی از فرض استقرا اثبات می‌شود.

در بخش ۴-۳ گونه‌ی دیگری از مرتب‌سازی ادغامی را بر روی لیست‌ها بیان می‌کنیم و نشان می‌دهیم که زمان اجرای آن بهینه است.

تحلیل مرتب‌سازی ادغامی

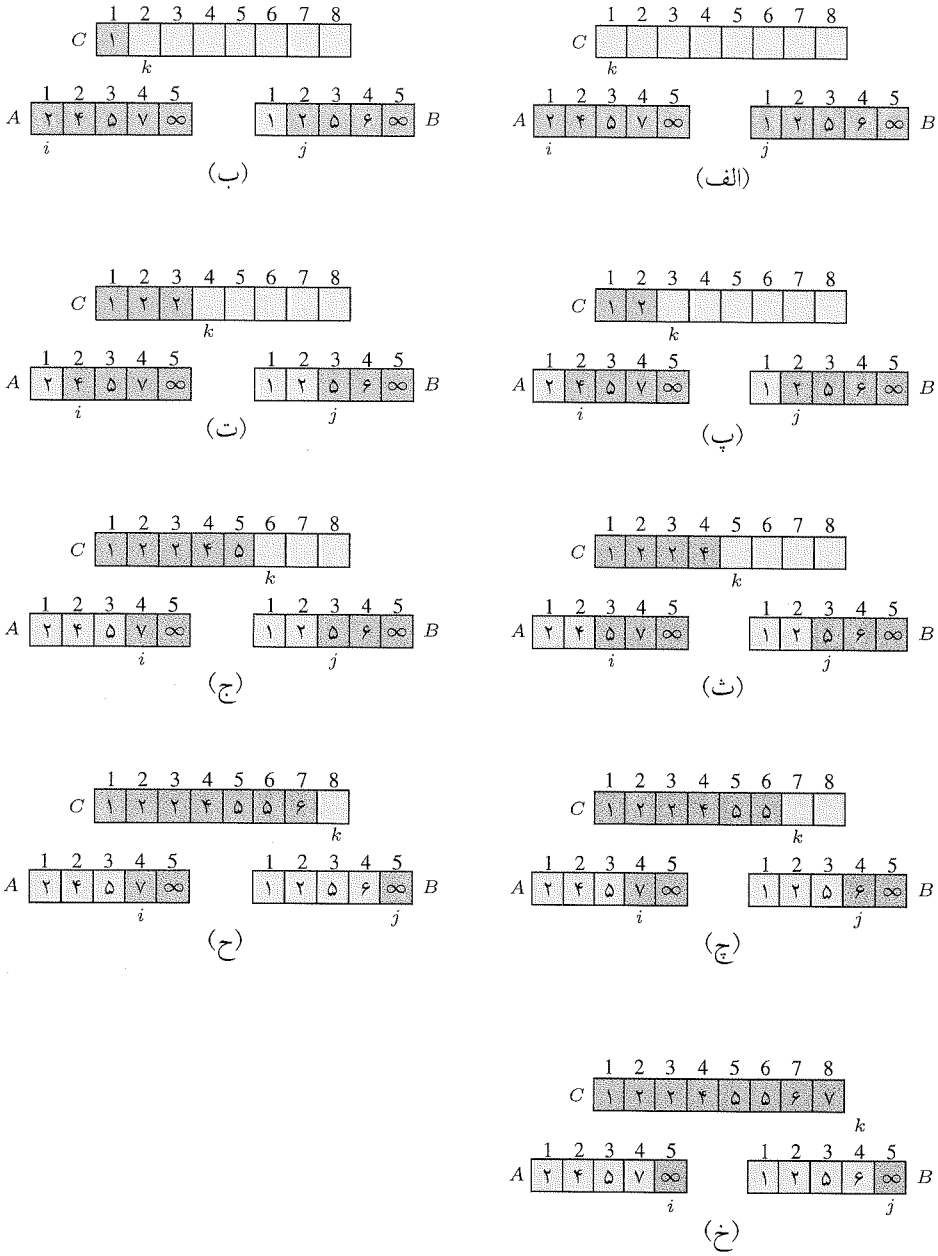
با توجه به این‌که عمل ادغام دو زیرآرایه با هزینه‌ای متناسب با n انجام می‌شود، زمان اجرای مرتب‌سازی ادغامی را به‌صورت زیر تحلیل می‌کنیم:

$$T(n)=\begin{cases} c_۱, & n=۲ \\ T(\lfloor n/۲ \rfloor)+T(\lceil n/۲ \rceil)+c_۲n, & n>۲ \end{cases} \qquad (۱۸-۳)$$

در این‌جا $T(\lfloor n/۲ \rfloor)$ و $T(\lceil n/۲ \rceil)$ هزینه‌های حل بازگشتی زیرمسئله‌ها و $c_۲n$ هزینه‌ی ادغام آن‌هاست.

اگر $n=۲^k$، رابطه به این صورت خواهد شد.

$$T(n)=\begin{cases} c_۱, & n=۲ \\ T(n)=۲T(n/۲)+c_۲n, & n>۲ \end{cases} \qquad (۱۹-۳)$$



شکل ۳-۱۰ مراحل اجرای الگوریتم MERGE-2 برای ادغام دو آرایه‌ی مرتب A و B. حاصل در آرایه‌ی C ذخیره می‌شود.

تمرین‌های زیربخش ۳-۴-۲

۳-۴-۲-۱.۲ اگر در مسئله‌ی برج هانوی امکان انتقال سکه‌ها از میله‌ی ۱ به ۲ و از ۲ به ۱ نباشد، کم‌ترین تعداد حرکت‌ها برای انتقال n سکه از میله‌ی ۱ به ۲ چه‌قدر است؟

۳-۴-۲-۲.۲ مسئله‌ی برج هانوی را در حالتی حل کنید که در ابتدا n_1 سکه‌ی دل‌خواه به‌صورت مرتب در میله‌ی ۱، n_2 سکه‌ی دل‌خواه به‌صورت مرتب در میله‌ی ۲ و بقیه‌ی سکه‌ها به‌صورت مرتب در میله‌ی ۳ باشند و بخواهیم که در انتها، همه‌ی سکه‌ها به‌صورت مرتب در میله‌ی ۳ قرار گیرند. تعداد حرکت‌های بهینه را به‌دست آورید.

۳-۴-۲-۳.۲ فرض کنید سکه‌ها به‌صورت نامرتب در میله‌ها قرار داشته باشند و حتی سکه‌ی بزرگ‌تر بتواند فقط در ابتدا بر روی سکه‌ی کوچک‌تر قرار بگیرد. چگونه همه‌ی سکه‌ها را با رعایت مقررات بازی به یکی از میله‌ها منتقل کنیم به‌طوری که در انتها مرتب باشند؟ روش کار را توضیح دهید.

۳-۴-۲-۴.۲ در گونه‌ای از مسئله‌ی برج هانوی از $2n$ سکه با اندازه‌های ۱ تا $2n$، در ابتدا n سکه با اندازه‌های فرد به‌صورت مرتب در میله‌ی ۱ و n سکه با اندازه‌های زوج به‌صورت مرتب در میله‌ی ۲ قرار دارند. می‌خواهیم همه را به میله‌ی ۳ منتقل کنیم. اگر $T_{ijk}(p,q)$ تعداد کمینه‌ی حرکت‌ها برای انتقال p سکه از میله‌ی i و q سکه از میله‌ی j (که سکه‌ها یک در میان به‌ترتیب قرار گیرند) به میله‌ی k باشد، رابطه‌ی بازگشتی برای $T_{۱۲۳}(n,n)$ را بنویسید.

۳-۴-۲-۵.۲ سه میله با شماره‌های ۱،۲ و ۳، و n سکه‌ی سوراخ‌دار با شماره‌های ۱ تا n داریم. سکه‌های زوج قرمز و سکه‌های فرد آبی هستند. هر سکه فقط می‌تواند روی سکه‌ی بزرگ‌تر و با رنگ متفاوت قرار بگیرد. همچنین بزرگ‌ترین سکه‌ی میله‌ی شماره‌ی ۱ و ۲ آبی و بزرگ‌ترین سکه‌ی میله‌ی شماره‌ی ۳ باید قرمز باشد. با این قواعد، به‌چند طریق می‌توان این سکه‌ها را روی میله‌ها قرار داد؟ ادعای خود را ثابت کنید.[۳۴]

۳-۴-۲-۶.۲ مرتب‌سازی درجی را می‌توان به این صورت به‌عنوان یک رویه‌ی بازگشتی بیان کرد: برای این‌که $A[۱\cdots n]$ را مرتب کنیم، ابتدا به طور بازگشتی $A[۱\cdots n-۱]$ را مرتب و سپس $A[n]$ را در آرایه‌ی مرتب‌شده‌ی $A[۱\cdots n-۱]$ درج می‌کنیم. یک رابطه‌ی بازگشتی برای زمان اجرای این الگوریتم بازگشتی بنویسید.

۳-۴-۲-۷.۲ رویه‌ی MERGE برای ادغام دو آرایه‌ی مرتب A و B به‌ترتیب با اندازه‌های m و n به این شکل ارائه می‌شود. نتیجه باید در آرایه‌ی مرتب C به‌اندازه‌ی $m+n$ ذخیره شود.

```
MERGE (A, B, n, m)
1  i ← 1; j ← 1; k ← 0
2  while i ≤ n or j ≤ m
```

[۳۴] سؤال ۳، مرحله‌ی اول ششمین المپیاد کامپیوتر ایران

```
3     do k ← k + 1
4        if 'conditional statement'
5           then if A[i] ≤ A[j]
6                   then C[k] ← A[i]
7                        i ← i + 1
8                   else C[k] ← B[j]
9                        j ← j + 1
10          else if i > n
11                  then C[k] ← A[i]
12                       i ← i + 1
13                  else C[k] ← B[j]
14                       j ← j + 1
```

الف) برای چه عبارتی به‌جای 'conditional statement'، این الگوریتم درست کار می‌کند؟

ب) بیشینه و کمینه‌ی تعداد مقایسه‌های بین عناصر آرایه‌ها (سطر ۷) در این الگوریتم چند تاست؟ (توجه کنید که در شرط A or B، اگر A درست باشد، B بررسی نمی‌شود و در شرط A and B، اگر A نادرست باشد، B بررسی نمی‌شود.)

۵-۳ روش‌های حل رابطه‌های بازگشتی

رابطه‌های بازگشتی را می‌توان به‌روش‌های زیر حل کرد:

۱. حدس و استقرا: حدس خوب و استفاده از استقرا برای اثبات آن،

۲. تکرار با جای‌گذاری: بازکردن فرمول و به‌دست آوردن جواب به‌طور صریح،

۳. درخت بازگشت[۳۵]،

۴. قضیه‌ی اصلی،

۵. روش‌های حل رابطه‌های بازگشتی همگن، و

۶. روش‌های دیگر مانند تابع مولد.

---

[۳۵] recursion tree

۳-۵-۱ حدس و استقرا

در این روش ابتدا سعی می‌کنیم جواب را حدس بزنیم و سپس با استقرا آن‌را اثبات کنیم. برای حدس جواب ممکن است از قضیه‌ی اصلی یا درخت بازگشت، که در ادامه توضیح داده خواهند شد، استفاده کنیم.

برای اثبات $T(n) = \mathcal{O}(f(n))$، باید مطابق تعریف $\mathcal{O}$، با استقرا ثابت کنیم که برای هر $n > n_0$، $T(n) \leq cf(n)$ و در مراحل استقرا نشان دهیم که یک مقدار ثابت و مثبت برای c و یک عدد صحیح و مثبت n_0 وجود دارد که در این نامعادله صدق می‌کند. برای اثبات $T(n) = \Omega(f(n))$، باید به‌همین ترتیب نشان دهیم که رابطه‌ی $T(n) \geq cf(n)$ برای یک c ثابت و مثبت و nهای بزرگ برقرار است. برای اثبات $T(n) = \Theta(f(n))$، باید هر دو رابطه‌ی $T(n) = \mathcal{O}(f(n))$ و $T(n) = \Omega(f(n))$ را ثابت کنیم.

مثال ۳-۴

$$\begin{aligned} T(n) &= T(\lfloor \frac{n}{2} \rfloor) + n \\ T(2) &= T(1) = \mathcal{O}(1) \end{aligned}$$

حل: به نظر می‌رسد که جواب این رابطه‌ی بازگشتی، در نهایت برابر $n + n/2 + n/4 + n/8 + \cdots$ باشد و کران این مقدار برای $n \to \infty$ برابر $2n$ می‌شود. پس حدس می‌زنیم که $T(n) = \Theta(n)$. برای اثبات $T(n) = \mathcal{O}(n)$ فرض می‌کنیم مقادیر درستی برای $c > 0$ و $n_0 > 0$ داریم که برای $n_0 \leq k < n$ رابطه‌ی $T(k) \leq ck$ برقرار است. این دو مقدار c و n_0 را در مرحله‌ی گام استقرا مشخص می‌کنیم. این فرض در پایه‌ی $n = 1$ برای هر c درست است. در گام استقرا داریم

$$T(n) \leq c\lfloor \frac{n}{2} \rfloor + n \leq (c/2 + 1)n.$$

و این نامعادله برای $c > 2$ برقرار است.

برای اثبات $T(n) = \Omega(n)$، به‌طور مشابه باید نشان دهیم که مقادیری برای $c > 0$ و $n_0 > 0$ وجود دارند که به‌ازای $n \geq n_0$ رابطه‌ی $T(n) \geq cn$ همیشه برقرار است. باز با فرض استقرای $T(k) \geq ck$، می‌توان به‌سادگی دید که برای $c \leq 2$ و هر مقدار مثبت n_0 داریم $T(n) \geq cn$. پس حدس خود را به‌طور کامل ثابت کردیم.

نکته‌ی ۳-۶ در حل رابطه‌های بازگشتی، برای به‌دست آوردن مرتبه یا $\mathcal{O}()$ می‌توان از $\lfloor .. \rfloor$

و [..] صرف‌نظر کرد. البته اگر از این علایم صرف‌نظر نکنیم جواب دقیق‌تری به‌دست خواهیم آورد.

مثال ۳-۵ رابطه‌ی بازگشتی ۳-۱۸ را حل کنید و با آن الگوریتم مرتب‌سازی ادغامی را تحلیل کنید.

حل: با توجه به این‌که هر بار مسئله را به دو زیرمسئله با اندازه‌ی نزدیک به نصف تقسیم می‌کنیم، بعد از حدود $\lg n$ بار، مسئله‌ای به اندازه‌ی ۱ خواهیم داشت. چون زیرمسئله‌های اصلی را با $\mathcal{O}(n)$ با هم ترکیب می‌کنیم، حدس می‌زنیم که کل مسئله از مرتبه‌ی $\mathcal{O}(n \lg n)$ است. البته بعداً می‌بینیم که این حدس با استفاده از قضیه‌ی اصلی ساده‌تر بیان می‌شود. البته ثابت می‌کنیم که $T(n) = \Theta(n \lg n)$. برای این‌کار، با استقرا ثابت می‌کنیم که برای مقادیر بزرگ n، $a > 0$ وجود دارد که $T(n) \leq an \lg n$ و نیز $b > 0$ وجود دارد که $T(n) \geq bn \lg n$.

برای اثبات $T(n) = \mathcal{O}(n \lg n)$، توجه می‌کنیم که،

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_2 n \leq 2T(\lfloor \frac{n}{2} \rfloor) + c_2 n \qquad (۳-۲۰)$$

پایه‌ی استقرا: $n = 2 \Rightarrow T(2) \leq 2a \Rightarrow a \geq \frac{c_1}{2} > 0$

فرض استقرا: برای $n \geq n_0$ و به‌ازای $k < n$ فرض می‌کنیم $T(k) \leq ck \lg k$.

حکم استقرا: با استفاده از فرمول ۳-۲۰ ثابت می‌کنیم که $T(n) \leq an \lg n$. برای این‌کار،

$$\begin{aligned} T(n) &\leq 2a\lfloor \tfrac{n}{2} \rfloor \lg \lfloor \tfrac{n}{2} \rfloor + c_2 n \\ &\leq an \lg \tfrac{n}{2} + c_2 n \\ &= an \lg n - an + c_2 n. \end{aligned}$$

در نتیجه اگر $a \geq c_2$، $T(n) \leq an \lg n$ و حکم ثابت می‌شود.

نکته‌ی مهم در این روش حل آن است که باید در حکم استقرا دقیقاً به همان فرمولی برسیم که حدس زده‌ایم (مثلاً $\leq an \lg n$). در صورتی‌که این امر ثابت نشود، حدس اولیه نادرست بوده و باید آن‌را تصحیح کرد.

برای اثبات $T(n) = \Omega(n \lg n)$، مراحل اثبات را با توجه به رابطه‌ی زیر بیان می‌کنیم:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_2 n \geq 2T(\lceil \frac{n}{2} \rceil) + c_2 n, \qquad (۳-۲۱)$$

و در این‌جا، چنان‌چه گفته شد، مقدار $b > 0$ را به‌دست می‌آوریم که برای مقادیر بزرگ n همیشه رابطه‌ی $T(n) \geq bn \lg n$ برقرار باشد. مراحل این اثبات هم دقیقاً مشابه قبل است و از تکرار آن صرف‌نظر می‌کنیم.

تمرین ۳-۱ ثابت کنید که $T(n) = O(n \lg n)$ حل رابطه‌ی بازگشتی زیر است:

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n, \quad n > 100$$
$$T(1) = T(2) = \ldots = T(100) = 1.$$

مثال ۳-۶ مرتبه‌ی $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1$ را به‌دست آورید.

حل: حدس می‌زنیم جواب از مرتبه‌ی $O(n)$ است. باید ثابت کنیم $T(n) \leq cn$. اگر برای اثبات از استقرا استفاده کنیم، در نهایت خواهیم داشت

$$T(n) \leq c\lfloor \frac{n}{2} \rfloor + c\lceil \frac{n}{2} \rceil + 1 = cn + 1 > cn.$$

بنابراین حدس اشتباه است. آن‌را به این صورت تصحیح می‌کنیم: $T(n) \leq cn + b$. حال داریم

$$T(n) \leq c\lfloor \frac{n}{2} \rfloor + c\lceil \frac{n}{2} \rceil + 2b + 1 = cn + 2b + 1 \leq cn + b + 1,$$

که اگر فرض کنیم $b \leq -1$، اثبات کامل می‌شود.

مثال ۳-۷ $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$.

حل: حدس می‌زنیم که جواب از مرتبه‌ی n باشد. باید نشان دهیم $T(n) \leq cn$. با استفاده از استقرا داریم

$$T(n) \leq 2C\lfloor \frac{n}{2} \rfloor + n \leq cn + n = (c+1)n \geq cn.$$

در مورد این مثال حدس اولیه غلط است، ولی از نتیجه‌ی به‌دست آمده متوجه می‌شویم که مرتبه‌ی الگوریتم بالاتر از n است. اگر حدس جدید $T(n) \leq cn \lg n + b$ را امتحان کنیم جواب به‌دست می‌آید.

تغییر متغیر

یکی از راه‌های حل برای بعضی از رابطه‌های بازگشتی، تغییر متغیر است. به مثال زیر توجه کنید.

مثال ۳-۸ $T(n) = ۲T(\sqrt{n}) + \lg n$.

حل: متغیر جدید m را به‌صورت $m = \lg n$ در نظر می‌گیریم. در نتیجه خواهیم داشت $T(۲^m) = ۲T(۲^{\frac{m}{۲}}) + m$، یا $S(m) = ۲S(\frac{m}{۲}) + m$ که در آن $S(m) = T(۲^m)$. با توجه به مثال‌های قبلی داریم

$$S(m) = \mathcal{O}(m \lg m) \Rightarrow T(n) = \mathcal{O}(\lg n \lg \lg n).$$

۳-۵-۲ تکرار با جای‌گذاری

در این روش، رابطه‌ی بازگشتی را آن قدر بسط می‌دهیم تا به جواب نهایی برسیم. از این روش برای پیدا کردن جواب دقیق رابطه‌های بازگشتی هم استفاده می‌شود.

مثال ۳-۹ رابطه‌ی بازگشتی $T(n) = ۳T(\lfloor \frac{n}{۴} \rfloor) + n$ را با فرض $T(۱) = c$ حل کنید.

حل:

$$\begin{aligned} T(n) &= ۳T(\lfloor \tfrac{n}{۴} \rfloor) + n \\ &= ۳^۲T(\lfloor \tfrac{\lfloor \frac{n}{۴} \rfloor}{۴} \rfloor) + ۳\lfloor \tfrac{n}{۴} \rfloor + n = ۳^۲T(\lfloor \tfrac{n}{۴^۲} \rfloor) + ۳\lfloor \tfrac{n}{۴} \rfloor + n \\ &= ۳^۳T(\lfloor \tfrac{n}{۴^۳} \rfloor) + ۳^۲\lfloor \tfrac{n}{۴^۲} \rfloor + ۳\lfloor \tfrac{n}{۴} \rfloor + n \\ &= \dots \\ &= \dots \\ &\leq ۳^i T(\tfrac{n}{۴^i}) + n \sum_{j=۰}^{i-۱} (\tfrac{۳}{۴})^j. \end{aligned}$$

از برابری $\lfloor \frac{\lfloor \frac{n}{۴} \rfloor}{۴} \rfloor = \lfloor \frac{n}{۴^۲} \rfloor$ در فرمول‌های فوق استفاده می‌کنیم.

برای حل این رابطه‌ی بازگشتی باید آن را باز کنیم تا به $T(۱)$ برسیم. از $\frac{n}{۴^i} = ۱$ نتیجه می‌گیریم که $i = \log_۴ n$. بنابراین،

$$T(n) \leq ۳^{\log_۴ n} T(۱) + n \sum_{j=۰}^{\log_۴ n} (\tfrac{۳}{۴})^j.$$

اما می‌دانیم،

$$\lim_{n \to \infty} \sum_{j=۰}^{\lg n - ۱} (\tfrac{۳}{۴})^j = ۴ \Rightarrow c_۲ < ۴,$$

و داریم، $۳^{\log_۴ n} = n^{\log_۴ ۳}$. لذا $T(n) \leq cn^{\log_۴ ۳} + ۴n \Rightarrow T(n) = \mathcal{O}(n)$.

معمولاً روش بسط فرمول جواب می‌دهد، ولی در برخی مسئله‌ها، از باز کردن فرمول به‌جایی نمی‌رسیم. مثلاً، رابطه‌ی بازگشتی برای تولید اعداد فیبوناچی، یا $F(n) = F(n-۱) + F(n-۲)$ و $F(۱) = F(۲) = ۱$ را نمی‌توان با این روش حل کرد. برای حل این رابطه‌ها، از روش حل روابط بازگشتی همگن که در بخش ۳-۶ می‌آید استفاده می‌کنیم.

۳-۵-۳ درخت بازگشت

یکی از روش‌های خوب برای حل یا حدس رابطه‌های بازگشتی، استفاده از «درخت بازگشت[۳۶]» است. این روش نحوه‌ی جای‌گذاری یک عبارت بازگشتی و نیز مقدار ثابتی را که در هر سطح از آن عبارت به‌دست می‌آید نشان می‌دهد. حاصل جمع مقادیر ثابت تمام سطح‌ها، جواب راه‌حل بازگشتی است. این یا جواب دقیق رابطه‌ی بازگشتی است یا حدس مناسبی که با روش حدس و استقرا می‌توان آن را به‌طور دقیق اثبات کرد.

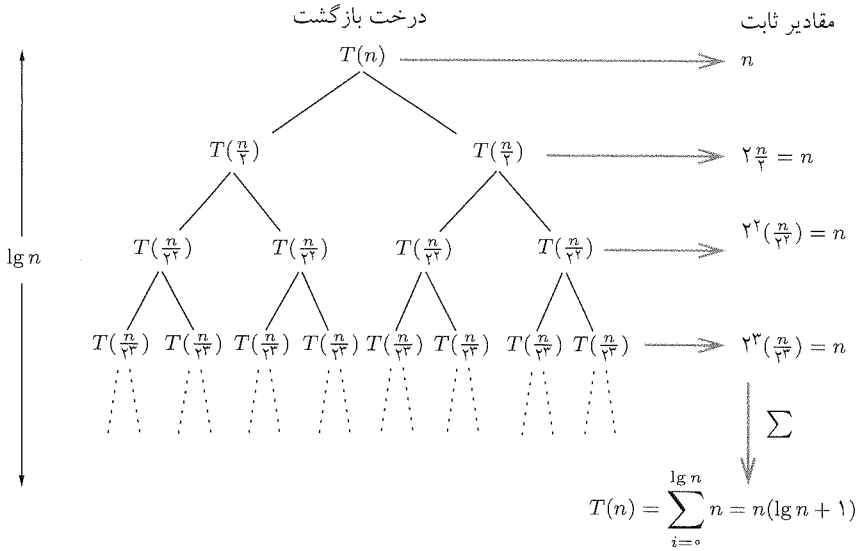
مثال ۳-۱۰ جواب دقیق رابطه‌ی زیر را برای $n = ۲^k$ به‌دست آورید:

$$T(n) = \begin{cases} ۱, & n = ۱ \\ ۲T(\frac{n}{۲}) + n, & n > ۱ \end{cases} \qquad (۲۲-۳)$$

حل: شکل ۳-۱۱ درخت بازگشت این رابطه است. در اولین سطح، محاسبه‌ی $T(n)$ منجر به دو بار محاسبه‌ی $T(\frac{n}{۲})$ و تولید مقدار ثابت n می‌شود. در سطح دوم، از هر کدام از زیردرخت‌های $T(\frac{n}{۲})$ یک مقدار ثابت $\frac{n}{۲}$ به‌دست می‌آید؛ یعنی در مجموع در این سطح هم $۲\frac{n}{۲} = n$ به‌دست می‌آید. اگر این کار را ادامه دهیم، می‌بینیم که در هر سطح مقدار n حاصل می‌شود. این درخت تا عمق $k = \lg n$ باز می‌شود که در آن $T(۱)$ محاسبه می‌شود. بنابراین در مجموع،

$$T(n) = \sum_{i=۰}^{\lg n} n = n(\lg n + ۱)$$

[۳۶] recursion tree



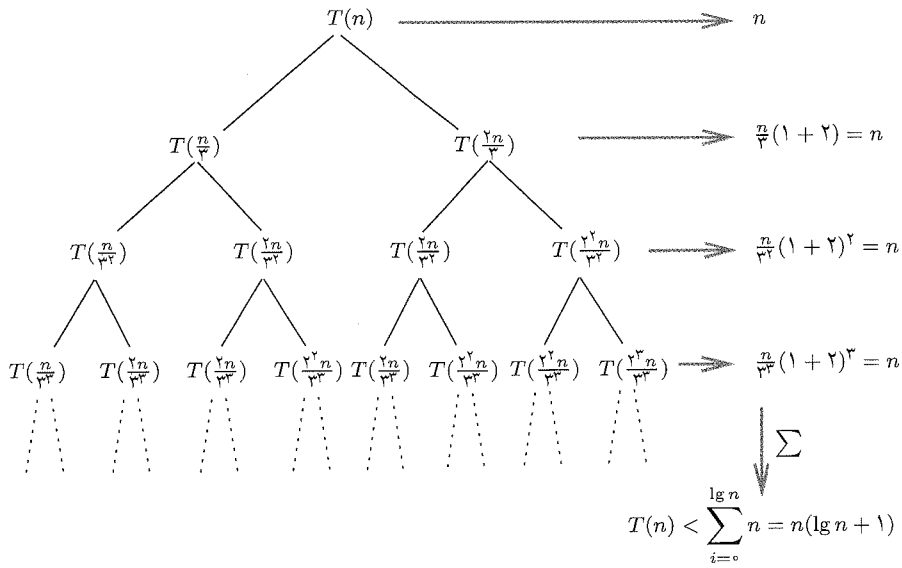
شکل ۳-۱۱ درخت بازگشت برای رابطه‌ی $T(n) = ۲T(\frac{n}{۲}) + n$.

به‌صورت دقیق محاسبه می‌شود. دقت کنید که اگر هم n توانی از ۲ نباشد، این محاسبات درست و $T(n) \leq n(\lg n + ۱) = \mathcal{O}(n \lg n)$ به‌دست خواهد آمد.

مثال ۳-۱۱ رابطه‌ی بازگشتی زیر را حل کنید و جواب را به‌صورت $\mathcal{O}()$ به‌دست آورید.

$$T(n) = \begin{cases} ۱, & n = ۱ \\ T(\frac{n}{۳}) + T(\frac{۲n}{۳}) + n, & n > ۱ \end{cases} \qquad (۲۳-۳)$$

حل: درخت بازگشت این رابطه، چنان‌چه در شکل ۳-۱۲ دیده می‌شود، به‌گونه‌ی غیرمتوازن رشد می‌کند و عمق سمت چپ آن کم‌تر است و در عمق $\log_۳ n$ به برگ $T(۱)$ می‌رسد. اما عمق سمت راست درخت برابر $\log_{\frac{۳}{۲}} n$ است. مقدار ثابتی که از هر سطح $i \leq \log_۳ n$ به‌دست می‌آید برابر n است، ولی در سطح‌های بالاتر، این مقدار کم‌تر می‌شود. بنابراین $T(n) \leq n(\lg n + ۱)$ و در نتیجه $T(n) = \mathcal{O}(\lg n)$. البته می‌توان این را با دقت بیش‌تری از روش حدس و استقرا اثبات کرد.



شکل ۳-۱۲ درخت بازگشت برای رابطه‌ی $T(n) = T(\frac{n}{۳}) + T(\frac{۲n}{۳}) + n$.

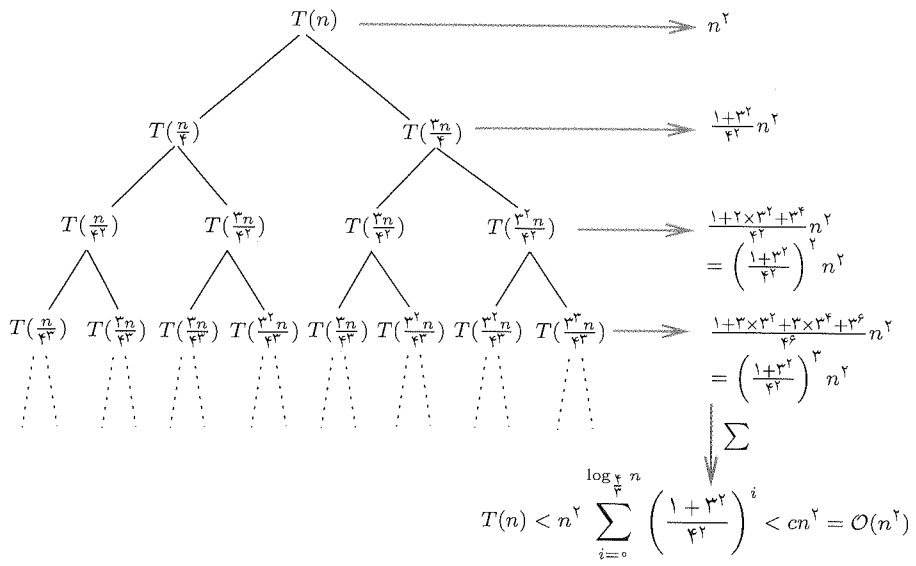
مثال ۳-۱۲ رابطه‌ی بازگشتی زیر را حل کنید.

$$T(n) = \begin{cases} ۱, & n = ۱ \\ T(\frac{n}{۴}) + T(\frac{۳n}{۴}) + n^۲, & n > ۱ \end{cases}$$

حل: درخت بازگشت مطابق شکل ۳-۱۳ است. ارتفاع درخت برابر $\log_{\frac{۴}{۳}} n$ است و چنان‌چه در شکل آمده است، جواب این رابطه‌ی بازگشتی $\mathcal{O}(n^۲)$ خواهد بود.

۳-۵-۴ قضیه‌ی اصلی

قضیه‌ی اصلی، رابطه‌های بازگشتی $T(n) = aT(\frac{n}{b}) + f(n)$ را با فرض $a \geq ۱$، $b > ۱$ و تابع $f(n)$ که به‌صورت مجانبی مثبت است در نظر می‌گیرد. این رابطه زمان اجرای الگوریتمی را برای مسئله‌ای با اندازه‌ی ورودی n توصیف می‌کند که برای حل، آن‌را به a زیرمسئله که هر کدام اندازه‌ای برابر $\frac{n}{b}$ دارند تقسیم می‌کند. زیرمسئله‌ها به‌صورت بازگشتی با همین الگوریتم حل می‌شوند و ترکیب حاصل آن‌ها نیاز به زمانی برابر $f(n)$ دارد. توجه



شکل ۳-۱۳ درخت بازگشت برای رابطه‌ی $T(n) = T(\frac{n}{4}) + T(\frac{3n}{4}) + n^2$

کنید که در الگوریتم‌های واقعی اندازه‌ها اعداد صحیح هستند، پس اندازه‌ی واقعی زیرمسئله‌ها یا $\lceil \frac{n}{b} \rceil$ یا $\lfloor \frac{n}{b} \rfloor$ است. البته قضیه‌ی اصلی برای این مقادیر هم درست است.

قضیه‌ی ۳-۲ (قضیه‌ی اصلی) فرض کنید $a \geq 1$، $b > 1$ مقادیر ثابت و $f(n)$ یک تابع است. رابطه‌ی بازگشتی

$$T(n) = aT(\frac{n}{b}) + f(n) \qquad (۳-۲۴)$$

برروی مقادیر مثبت n تعریف شده است که در آن $\frac{n}{b}$ برابر $\lceil \frac{n}{b} \rceil$ یا $\lfloor \frac{n}{b} \rfloor$ است. در آن‌صورت

الف) اگر برای $\epsilon > 0$، $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$، (یعنی آهنگ رشد تابع $f(n)$ از تابع $n^{\log_b a}$ به‌صورت چند جمله‌ای کم‌تر است) در این صورت $T(n) = \Theta(n^{\log_b a})$،

ب) اگر $f(n) = \Theta(n^{\log_b a})$، در این صورت $T(n) = \Theta(n^{\log_b a} \log_2 n)$، و

پ) اگر $f(n) = \Omega(n^{\log_b a + \epsilon})$، در این صورت $T(n) = \Theta(f(n))$.

در این سه حالت، آهنگ رشد دو تابع $g(n) = n^{\log_b a}$ و $f(n)$ را با هم مقایسه می‌کنیم و جواب رابطه‌ی بازگشتی به‌کمک تابعی که آهنگ رشد بیش‌تری نسبت به دیگری دارد تعیین می‌شود؛ البته چنان‌چه گفته شد، اختلاف آهنگ رشد تابع‌ها باید به‌صورت چندجمله‌ای باشد.

به‌عبارت ساده‌تر، هر چند نادقیق، اگر G آهنگ رشد تابع $g(n) = n^{\log_b a}$ و F آهنگ رشد $f(n)$ باشد، در آن‌صورت،

- اگر $F > G$، $T(n) = \Theta(f(n))$.
- اگر $G > F$، $T(n) = \Theta(g(n))$.
- اگر $F = G$، $T(n) = \Theta(g(n) \lg n)$.

توجه کنید که $<$ و $>$ به‌معنی بزرگ‌تر یا کوچک‌تر بودن آهنگ رشد دو چند جمله‌ای برای مقادیر بزرگ n است و بزرگ یا کوچک بودن آهنگ رشد هم به‌صورت چندجمله‌ای است. مثلاً آهنگ رشد n^2 از $n \log n$ و یا n از $\sqrt{n}$ به‌صورت چندجمله‌ای بیش‌تر است، ولی آهنگ رشد $n \lg n$ از n به‌صورت چندجمله‌ای بیش‌تر نیست. چرا که در حالت آخر نمی‌توان $\epsilon > 0$ ای پیدا کرد که برای مقادیر بزرگ n داشته باشیم: $n \lg n \leq n^{1+\epsilon}$.

این نکته هم قابل ذکر است که سه مورد فوق، همه‌ی حالات را در بر نمی‌گیرند و تابع‌های زیادی پیدا می‌شوند، مانند رابطه‌ی بخش ۳-۵-۵، که هیچ‌یک از این حالت‌ها برای آن‌ها صادق نیست. در این موارد، قضیه‌ی اصلی قابل استفاده نیست و باید از روش‌های دیگری برای حل رابطه‌ی بازگشتی استفاده کرد.

این قضیه با استفاده از روش جای‌گذاری قابل اثبات است، ولی در این‌جا از ارائه‌ی جزئیات اثبات صرف‌نظر می‌کنیم. علاقه‌مندان می‌توانند به کتاب‌های مختلف مانند [۵] مراجعه کنند.

مثال‌هایی از حل رابطه‌های بازگشتی با استفاده از قضیه‌ی اصلی

مثال ۳-۱۳ $T(n) = 9T(\frac{n}{3}) + n$.

حل: داریم، $f(n) = n$ و $g(n) = n^{\log_3 9} = n^2$. بدیهی است که آهنگ رشد n^2 از n به‌صورت چند جمله‌ای بیش‌تر است. لذا، $T(n) = \Theta(n^2)$.

مثال ۳-۱۴ $T(n) = T(\frac{2n}{3}) + 1$.

حل: داریم، $g(n) = n^{\log_{\frac{۳}{۲}} ۱} = n^{۰} = ۱$ و $f(n) = ۱$. پس، $T(n) = \Theta(\lg n)$.

مثال ۳-۱۵ $T(n) = ۳T(\frac{n}{۴}) + n \lg n$.

حل: چون، $f(n) = n \lg n$ و $g(n) = n^{\log_۴ ۳} = n^{۰/۷۹۳}$ و $f(n) = \Omega(n^{۰/۷۹۳+۰/۱})$، داریم $T(n) = \Theta(n \lg n)$.

مثال ۳-۱۶ $T(n) = ۲T(\frac{n}{۲}) + n \lg n$.

حل: برای این مثال، $g(n) = n^{\lg ۲} = n$ و $f(n) = n \lg n$. ولی اختلاف این‌دو به‌صورت چندجمله‌ای نیست یعنی هیچ ϵ ای نمی‌توان پیدا کرد که برای کلیه‌ی n های بزرگ، رابطه‌ی $n \lg n > n^{۱+\epsilon}$ برقرار باشد. بنابراین، این مسئله را باید از روش دیگری، مثلاً استقرا حل کرد. جواب $T(n) = O(n \lg^۲ n)$ است که با استقرا به‌صورت زیر اثبات می‌شود.

پایه‌ی استقرا: $T(۲) \leq ۲c$،

فرض استقرا: $T(k) \leq ck \lg^۲ k$ برای $k < n$.

حکم:

$$\begin{aligned} T(n) &\leq ۲c\frac{n}{۲}\lg^۲\frac{n}{۲} + n\lg n \\ &\leq cn(\lg n - ۱)^۲ + n\lg n \\ &\leq cn\lg^۲ n + n[c + (۱ - ۲c)\lg n] \end{aligned}$$

برای $c = ۱$ و $n > ۲$ داریم $c + (۱ - ۲c)\lg n < ۰$ و حکم ثابت می‌شود.

۳-۵-۵ حل مستقیم یک رابطه‌ی بازگشتی

در این بخش مثالی را بررسی می‌کنیم که حل آن مستقیماً با استفاده از هیچ‌یک از روش‌های گفته‌شده مقدور نیست.

تابع بازگشتی زیر را در نظر بگیرید:

$$f(n) = \begin{cases} ۱, & n = ۰ \\ \frac{۱+n(n-۱)f(n-۱)}{n^۲+۱}, & n > ۰. \end{cases} \qquad (۳-۲۵)$$

پس $f(۱) = \frac{۱}{۲}$، $f(۲) = \frac{۲}{۵}$، و ....

می‌خواهیم رفتار مجانبی $f(n)$ را برای $n \to \infty$ به‌دست آوریم، یعنی تابع ساده‌ی $g(n)$ را پیدا کنیم که $f(n) = \Theta(g(n))$.

حل: ثابت می‌کنیم که $f(n) = \mathcal{O}(\frac{\lg n}{n})$ و $f(n) = \Omega(\frac{\lg n}{n})$.

برای اثبات رابطه‌ی $\mathcal{O}$، ابتدا $f(n) = \frac{۱+n(n-۱)f(n-۱)}{n^۲+۱}$ را به‌صورت زیر می‌نویسیم:

$$nf(n) = \left[\frac{۱}{n} + (n-۱)f(n-۱)\right]\frac{n^۲}{n^۲+۱}$$

و $g_۱(n)$ را به‌این صورت تعریف می‌کنیم:

$$g_۱(n) = nf(n). \qquad (۲۶\text{-}۳)$$

از این نتیجه می‌گیریم

$$\begin{aligned} g_۱(n) &= \frac{n^۲}{n^۲+۱}\left[\frac{۱}{n} + g_۱(n-۱)\right] \\ &< \frac{۱}{n} + g_۱(n-۱). \end{aligned}$$

یعنی، $g_۱(n) < ۱ + \frac{۱}{۲} + \cdots + \frac{۱}{n}$. اما می‌دانیم

$$h_۱(n) = ۱ + \frac{۱}{۲} + \cdots + \frac{۱}{n} = \Theta(\lg n). \qquad (۲۷\text{-}۳)$$

بنابراین، از این و تعریف ۳-۲۶ نتیجه می‌گیریم که $f(n) = \mathcal{O}(\frac{\lg n}{n})$.

برای اثبات $f(n) = \Omega(\frac{\lg n}{n})$، رابطه‌ی ۳-۲۵ را به‌صورت زیر می‌نویسیم:

$$\frac{n^۲}{n+۱}f(n) = \frac{n^۴}{n^۴-۱}\left[\frac{n-۱}{n^۲} + \frac{(n-۱)^۲}{n}f(n-۱)\right]$$

و از تعریف

$$g_۲(n) = \frac{n^۲}{n+۱}f(n) \qquad (۲۸\text{-}۳)$$

به رابطه‌های زیر می‌رسیم:

$$\begin{aligned} g_۲(n) &= \frac{n^۴}{n^۴-۱}\left[\frac{n-۱}{n^۲} + g_۲(n-۱)\right] \\ &> \frac{۱}{n} - \frac{۱}{n^۲} + g_۲(n-۱) \\ &> ۱ + \frac{۱}{۲} + \cdots + \frac{۱}{n} - (۱ + \frac{۱}{۴} + \cdots + \frac{۱}{n^۲}). \end{aligned}$$

اگر رابطه‌ی $h_۲(n) = ۱ + \frac{۱}{۴} + \cdots + \frac{۱}{n^۲}$ را تعریف کنیم و برای یک ثابت c فرض کنیم که $h_۲(n) < c$، در آن‌صورت $g_۲(n) > h_۱(n) - c$. در نتیجه،

$$f(n) > \left[\frac{۱}{n} + \frac{۱}{n^۲}\right](h_۱(n) - c)$$

که نتیجه می‌دهد $f(n) > \frac{h_۱(n)-c}{n}$ یا $f(n) = \Omega(\frac{\lg n}{n})$. بنابراین،

$$f(n) = \Theta(\frac{\lg n}{n}).$$

تمرین‌های بخش ۳-۵

۱.۵-۳ با استفاده از درخت بازگشت، استدلال کنید که جواب رابطه‌ی بازگشتی $T(n) = T(\frac{n}{۳}) + T(\frac{۲n}{۳}) + cn$ که در آن c یک ثابت است، $\Omega(n \lg n)$ می‌شود.

۲.۵-۳ برای رابطه‌ی بازگشتی $T(n) = ۴T(\lceil \frac{n}{۲} \rceil) + cn$ که در آن c یک ثابت است یک درخت بازگشت بکشید و برای پاسخ آن یک کران مجانبی دقیق ارائه نمایید. با استفاده از روش جای‌گذاری، درستی کران به‌دست‌آمده را تحقیق کنید.

۳.۵-۳ با استفاده از درخت بازگشت، یک جواب مجانبی دقیق برای رابطه‌ی بازگشتی $T(n) = T(n-a) + T(a) + cn$ به‌دست آورید. $c > ۰$ و $a \geq ۱$ ثابت هستند.

۴.۵-۳ جواب رابطه‌ی بازگشتی $T(n) = ۹۹T(n/۱۰۰) + \lg(n!)$ چیست؟

۵.۵-۳ فقط از طریق جای‌گذاری ثابت کنید که جواب رابطه‌ی بازگشتی $T(n) = T(۲n/۳) + \lg^۲ n$ برابر $\Theta(\lg^۳ n)$ است.

۶.۵-۳ جواب رابطه‌ی بازگشتی $T(n) \leq T(\frac{n}{۵}) + T(\frac{۷n}{۱۰}) + \Theta(n)$ را از طریق درخت بازگشت به‌صورت $T(n) = \Theta(f(n))$ به‌دست آورید.

۷.۵-۳ رابطه‌ی بازگشتی $T(n) \leq ۲T(n/۲) + \frac{n}{\lg n}$ را حل کنید.

۸.۵-۳ می‌خواهیم رابطه‌ی بازگشتی $T(n) = T(۲n/۵) + T(۳n/۱۰) + n$ را (برای $T(۱) = c$) به‌صورت $T(n) \leq f(n)$ حل کنیم. کوچک‌ترین فرمول $f(n)$ را به‌طور دقیق به‌دست آورید.

۹.۵-۳ در رابطه‌ی بازگشتی $T(n) = T(\alpha n) + T(\beta n) + n$ با مقادیر مثبت برای α و β، ثابت کنید که اگر $\alpha + \beta = ۱$، در آن‌صورت $T(n) = O(n \lg n)$.

۱۰.۵-۳ نشان دهید که در حل رابطه‌ی بازگشتی $T(n) = ۲T(\lfloor \frac{n}{۲} \rfloor) + n$، با استفاده از یک فرض استقرای متفاوت، می‌توان بر دشواری شرایط مرزی $T(۱) = ۱$ غلبه کرد، بدون آن‌که نیاز به تنظیم

شرایط مرزی برای اثبات گام استقرا باشد.

۳-۵.۱۱ فرض کنید که رابطه‌ی بازگشتی $T(n) = ۷T(\frac{n}{۲}) + n^۲$ زمان اجرای الگوریتم A است. الگوریتم A' رقیب الگوریتم A، دارای زمان اجرای $T'(n) = aT'(\frac{n}{۴}) + n^۲$ است. بزرگ‌ترین مقدار عدد صحیح a را بیابید که به‌ازای آن A' به‌طور مجانبی سریع‌تر از A باشد.

۳-۵.۱۲ رابطه‌های بازگشتی زیر را از هر راهی که می‌دانید حل کنید و جواب را بر حسب Θ یا $\mathcal{O}$ به‌دست آورید.

الف) $T(n) = T(n/۵) + T(n/۲) + n$

ب) $T(n) = T(n/۲ + ۵) + n^۲$

پ) $T(n) = T(n/۲) + T(n/۳) + n$

ت) $T(n) = ۴T(n/۲) + n^۲/\lg n$

ث) $T(n) = ۳T(n^{\frac{۱}{۴}}) + \log_۳ n$

ج) $T(n) = T(n-۱) + n^۲$

چ) $T(n) = T(n/۲ + \sqrt{n}) + n$

ح) $T(n) = ۲T(\frac{n}{۲}) + n\lg^۲ n$

خ) $T(n) = T(\frac{n}{۴}) + T(\frac{۳n}{۴}) + n^۲$

د) $T(n) = ۲T(n/۲) + n\lg n$

ذ) $T(n) = T(n/۳) + T(۲n/۳) + n$

ر) $T(n) = ۴T(n/۲) + n^۲/\lg n$

ز) $T(۲^n) = T(۲^{n-۱}) + ۲^n$

ژ) $T(n) = T(\frac{n}{۹}) + T(\frac{n}{۱۶}) + T(\frac{n}{۲۵}) + \sqrt{n}$

س) $T(n) = ۹T(\frac{n}{۱۶}) + (\sqrt[۳]{n})$

۳-۶ رابطه‌های بازگشتی همگن

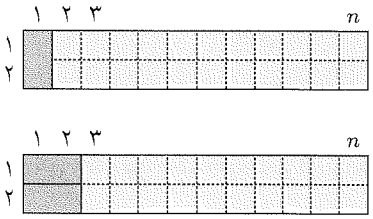
هدف این بخش، معرفی و بررسی «رابطه‌های بازگشتی همگن[۳۷]» و ارائه‌ی روش‌های حل آن‌هاست. این رابطه‌ها گونه‌ی کلی‌تر از رابطه‌های بازگشتی‌اند که ما تاکنون بررسی کرده‌ایم و خواهیم دید که روش‌های حلی که گفته‌ایم برای این رابطه‌ها قابل استفاده نیستند. در این‌جا برای سادگی، از نماد a_n به‌عنوان nامین دنباله‌ی بازگشتیِ a استفاده می‌کنیم؛ در بخش‌های قبل، نماد $T(n)$ به این منظور به‌کار می‌رفت که به زمان اجرای الگوریتم نزدیک‌تر بود.

[۳۷] homogeneous recursive relations

به مثال‌های زیر توجه کنید:

مثال ۳-۱۷ به چند طریق می‌توان صفحه‌ای شطرنجی با اندازه‌ی $n \times ۲$ را با موزاییک‌های ۱×۲ فرش کرد؟

حل: اگر جدول $n \times ۲$ را بتوان به f_n روش مختلف با موزاییک‌های $n \times ۲$ فرش کرد، به‌راحتی می‌توان ثابت کرد که $f_n = f_{n-۱} + f_{n-۲}$. به‌این ترتیب که اگر در ستون اول جدول، یک موزاییک به‌صورت عمودی گذاشته شود، جدولی $(n - ۱) \times ۲$ می‌ماند که باید با همین موزاییک‌ها فرش شود که به $f_{n-۱}$ طریق ممکن است. وگرنه، حتماً دو ستون اول و دوم با دو موزاییک افقی پوشانده شده‌اند، که در آن صورت جدول $(n - ۲) \times ۲$ می‌ماند که باید با همین موزاییک‌ها فرش شود که به $f_{n-۲}$ طریق ممکن است (شکل ۳-۱۴ را ببینید). توجه کنید که تعداد حالات فرش‌کردن متفاوت مورد نظر است، و این دو موزاییک افقی یک حالت محسوب می‌شوند، پس $f_n = f_{n-۱} + f_{n-۲}$. این رابطه، با مقادیر اولیه‌ی $f_۰ = ۰$ و $f_۱ = ۱$ همان دنباله‌ی معروف فیبوناچی است که در پیوست ۲ هم به آن اشاره شده است.



شکل ۳-۱۴ فرش‌کردن صفحه‌ای $n \times ۲$ با موزاییک‌های ۱×۲.

تمرین ۳-۲ به چند طریق می‌توان صفحه‌ای $n \times ۳$ را با موزاییک‌های ۱×۲ فرش کرد؟ (رابطه‌ای بازگشتی بیابید و آن را حل کنید.)

چنان‌چه در بخش ۳-۵ هم اشاره شد، استفاده از روش‌های بیان‌شده برای حل رابطه‌های بازگشتی، مثلاً تکرار با جای‌گذاری، برای حل این رابطه امکان‌پذیر نیست و نمی‌توان به فرمول مشخصی برای جواب مرحله‌ی iام جای‌گذاری رسید تا از آن جواب نهایی را حدس زد و آن‌را با استقرا ثابت کرد (امتحان کنید).

تعریف ۳-۶ اگر c_i ها عددهای حقیقی باشند، به رابطه‌ی بازگشتی زیر «رابطه‌ی بازگشتی همگن از درجه‌ی k» می‌گوییم:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}. \qquad (۲۹-۳)$$

در این رابطه a_n بر حسب k جمله‌ی قبل آن داده شده است و برای تولید کامل دنباله به k جمله‌ی اول آن احتیاج داریم.

تابع $g(n)$ را یک جواب دنباله‌ی بازگشتی ۳-۲۹ می‌نامیم، اگر برای هر n صحیح، دنباله‌ی $a_n = g(n)$ در رابطه‌ی بازگشتی صدق کند.

قضیه‌ی ۳-۳ (اصل برهم‌نهی) اگر $g_i(n)$ برای هر $1 \le i \le r$ یک جواب برای رابطه‌ی همگن

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} \qquad (۳۰-۳)$$

باشد، آن‌گاه هر ترکیب خطی از این r جواب، یعنی

$$A_1 g_1(n) + A_2 g_2(n) + \cdots + A_r g_r(n)$$

که در آن A_i ها اعدادی حقیقی‌اند، نیز یک جواب برای رابطه‌ی ۳-۳۰ است.

اثبات: فرض کنید

$$h(n) = A_1 g_1(n) + A_2 g_2(n) + \cdots + A_r g_r(n)$$

چون برای هر $1 \le i \le r$، $g_i(n)$ یک جواب برای رابطه‌ی ۳-۳۰ است، پس داریم

$$g_i(n) = c_1 g_i(n-1) + c_2 g_i(n-2) + \cdots + c_k g_i(n-k). \qquad (۳۱-۳)$$

اگر این رابطه را به‌ازای هر i در A_i ضرب کنیم، از جمع این r رابطه نتیجه می‌گیریم

$$h(n) = c_1 h(n-1) + c_2 h(n-2) + \cdots + c_k h(n-k). \qquad (۳۲-۳)$$

بنابراین، حکم قضیه ثابت می‌شود. □

حل رابطه‌های بازگشتی همگن

رابطه‌ی بازگشتی همگن را می‌توان در حالت کلی به این صورت حل کرد: اگر $g(n) = x^n$، جواب رابطه‌ی بازگشتی همگن ۳-۲۹ باشد، داریم

$$x^n - c_1 x^{n-1} - c_2 x^{n-2} - \cdots - c_k x^{n-k} = 0,$$

یا به عبارت دیگر، با فرض این‌که x متحد با ۰ نیست،

$$x^k - c_1 x^{k-1} - \cdots - c_k = 0. \qquad (۳۳-۳)$$

یعنی x جواب معادله‌ی درجه‌ی k فرمول ۳-۳۳ است. این معادله را «معادله‌ی سرشت‌نما[۳۸]» یا «معادله‌ی مشخصه»ی رابطه‌ی بازگشتی ۳-۲۹ می‌نامیم. اگر x_i، یک ریشه برای معادله‌ی سرشت‌نما باشد، $a_n = x_i^n$ یک جواب رابطه‌ی بازگشتی است و بنا بر قضیه‌ی ۳-۳، هر ترکیب خطی از x_i^nها هم یک جواب برای رابطه‌ی بازگشتی است. ضمناً این جواب‌ها پایه‌ای برای مجموعه جواب‌های این رابطه هستند. بنابراین می‌توان گفت که تمام جواب‌های رابطه‌ی ۳-۲۹ به‌صورت ترکیبی خطی از x_i^nها است. (چگونگی اثبات را تحقیق کنید.)

به‌عنوان مثال، ترکیب خطی

$$a_n = t_1 x_1^n + t_2 x_2^n + \cdots + t_k x_k^n \qquad (۳۴-۳)$$

را در نظر می‌گیریم. اگر در این رابطه‌ی بازگشتی، k عنصر اول این دنباله داده شده باشند، k معادله‌ی زیر برقرار هستند،

$$\begin{cases} a_0 = t_1 + t_2 + \cdots + t_k \\ a_1 = t_1 x_1 + t_2 x_2 + \cdots + t_k x_k \\ \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\ a_{k-1} = t_1 x_1^{k-1} + t_2 x_2^{k-1} + \cdots + t_k x_k^{k-1} \end{cases}$$

این یک دستگاه k معادله و k مجهول با مجهول‌های t_1 تا t_k است. با توجه به تشکیل دترمینان ضرایب، نتیجه می‌گیریم که اگر x_iها متمایز باشند این دستگاه معادلات یک جواب منحصربه‌فرد دارد، یعنی با دادن k عنصر اول، می‌توان جوابی منحصربه‌فرد برای

[۳۸] characteristic equation

دنباله پیدا کرد. (تحقیق کنید که چرا در صورت متمایز بودن x_iها، این دستگاه معادلات یک جواب منحصربه‌فرد دارد! برای این کار باید دترمینان ماتریس ضرایب را تشکیل دهید.)

مثال ۳-۱۸ اعداد دنباله‌ی فیبوناچی را که در مثال ۳-۱۸ تعریف شد به‌دست آورید.

حل: معادله‌ی سرشت‌نمای این رابطه به‌صورت $x^۲ - x - ۱ = ۰$ است و ریشه‌های آن $x_۱ = \frac{۱+\sqrt{۵}}{۲}$ و $x_۲ = \frac{۱-\sqrt{۵}}{۲}$ هستند. پس،

$$f_n = t_۱ \left(\frac{۱+\sqrt{۵}}{۲}\right)^n + t_۲ \left(\frac{۱-\sqrt{۵}}{۲}\right)^n.$$

با توجه به مقادیر اولیه داریم

$$\begin{cases} t_۱ + t_۲ = f_۰ = ۰ \\ t_۱(\frac{۱+\sqrt{۵}}{۲}) + t_۲(\frac{۱-\sqrt{۵}}{۲}) = f_۱ = ۱ \end{cases}.$$

از این معادله‌ها نتیجه می‌شود

$$t_۱ = \frac{۱}{\sqrt{۵}}, \quad t_۲ = -\frac{۱}{\sqrt{۵}}$$

$$f_n = \frac{۱}{\sqrt{۵}} \left(\left(\frac{۱+\sqrt{۵}}{۲}\right)^n - \left(\frac{۱-\sqrt{۵}}{۲}\right)^n \right). \qquad (۳۵\text{-}۳)$$

توجه کنید که جمله‌ی $(\frac{۱-\sqrt{۵}}{۲})^n$ با بزرگ شدن n بسیار کوچک می‌شود و با توجه به این‌که f_n عددی حسابی است، اگر $\langle x \rangle$ را نزدیک‌ترین عدد صحیح به x تعریف کنیم، یعنی

$$\langle x \rangle = \lfloor x + \frac{۱}{۲} \rfloor,$$

نتیجه می‌گیریم که

$$f_n = \left\langle \frac{۱}{\sqrt{۵}} \left(\frac{۱+\sqrt{۵}}{۲}\right)^n \right\rangle. \qquad (۳۶\text{-}۳)$$

در حل دستگاه به‌دست آمده برای یافتن ضرایب، این شرط را قرار دادیم که x_iها متمایز باشند. حال اگر x_i ریشه‌ی مضاعف درجه‌ی ۲ معادله‌ی سرشت‌نما باشد، $a_n = nx_i^n$ نیز یک جواب رابطه‌ی بازگشتی است (اثبات از راه مشتق‌گیری از معادله‌ی سرشت‌نما

است؛ به این ترتیب که ریشه‌ی مضاعف، ریشه‌ی مشتق معادله‌ی سرشت‌نما است). به‌همین طریق می‌توان استدلال کرد که اگر x_i ریشه‌ی مضاعف درجه‌ی ۳ باشد، $n^۲x_i^n$ نیز یک جواب رابطه‌ی بازگشتی است. در حالت کلی اگر x_i ریشه‌ی مضاعف درجه‌ی p باشد،

$$g(n) = t_۰x^n + t_۱nx^n + t_۲n^۲x^n + \cdots + t_{p-۱}n^{p-۱}x^n \qquad (۳۷-۳)$$

جوابی برای رابطه‌ی بازگشتی است.

مثال ۳-۱۹ رابطه‌ی بازگشتی زیر را حل کنید:

$$a_n = -a_{n-۱} + ۳a_{n-۲} + ۵a_{n-۲} + ۲a_{n-۳} \quad (n \geq ۵) \qquad (۳۸-۳)$$

حل: معادله‌ی سرشت‌نمای این رابطه به صورت زیر است:

$$x^۴ + x^۳ - ۳x^۲ - ۵x - ۲ = ۰$$

که برابر است با $(x-۲)(x+۱)^۳$. بنابراین

$$a_n = t_۱(-۱)^n + t_۲n(-۱)^n + t_۳n^۲(-۱)^n + t_۴۲^n$$

جواب کلی این رابطه است. با معلوم بودن مقادیر اولیه‌ی $a_۱$ تا $a_۴$، مقادیر $t_۱$ تا $t_۴$ به‌دست می‌آیند. مثلاً در حالت $a_۱ = ۴$، $a_۲ = -۳$، $a_۳ = ۲۲$ و $a_۴ = -۷$ از دستگاه‌های متناظر جواب‌های $t_۱ = -t_۳ = ۱$ و $t_۲ = -t_۴ = ۲$ به‌دست می‌آیند.

مثال بعد، حالتی را بررسی می‌کند که در آن معادله‌ی سرشت‌نما ریشه‌ی حقیقی ندارد.

مثال ۳-۲۰ رابطه‌ی بازگشتی

$$a_n = ۲a_{n-۱} - ۲a_{n-۲} \quad (a_۰ = ۱, a_۱ = ۰)$$

را حل کنید.

حل: معادله‌ی سرشت‌نمای این رابطه به‌صورت $x^۲ - ۲x + ۲ = ۰$ است که ریشه‌های غیرحقیقی $\alpha = ۱ + i$ و $\overline{\alpha} = ۱ - i$ دارد. پس داریم

$$\begin{cases} \alpha &= \sqrt{۲}\ \left(\cos(\frac{\pi}{۴}) + i\sin(\frac{\pi}{۴})\right) \\ \overline{\alpha} &= \sqrt{۲}\ \left(\cos(\frac{\pi}{۴}) - i\sin(\frac{\pi}{۴})\right) \end{cases}$$

بنابراین،

$$\begin{aligned} a_n &= A\alpha^n + B\overline{\alpha}^n \\ &= (\sqrt{۲})^n \left[A\left(\cos(\frac{n\pi}{۴}) + i\sin(\frac{n\pi}{۴})\right)\right] + \left[B\left(\cos(\frac{n\pi}{۴}) - i\sin(\frac{n\pi}{۴})\right)\right] \\ &= \sqrt{۲}\left[C\cos(\frac{n\pi}{۴}) + D\sin(\frac{n\pi}{۴})\right]. \end{aligned}$$

از مقادیر اولیه‌ی $a_۰ = ۱$ و $a_۱ = ۰$ نتیجه می‌گیریم که $C = ۱$ و $D = -۱$. پس

$$a_n = \sqrt{۲}\left((\cos(\frac{n\pi}{۴}) - \sin(\frac{n\pi}{۴})\right).$$

لازم به ذکر است که در این معادلات، از رابطه‌ی مهم زیر که با استقرا ثابت می‌شود، استفاده شده است.

$$(\cos\theta + i\sin\theta)^n = \cos(n\theta) + i\sin(n\theta)$$

حال با حل مسئله‌ای کاربردی، به روشی دیگر در حل رابطه‌های بازگشتی توجه کنید.

مثال ۳-۲۱ فرض کنید A و E دو رأس روبه‌روی یک ۸ ضلعی منتظم $ABCDEFGH$ هستند. قورباغه‌ای از رأس A شروع به جهیدن می‌کند و هر بار به رأس مجاور می‌پرد. ولی وقتی به رأس E رسید، همان‌جا متوقف می‌شود. a_n را تعداد مسیرهایی بگیرید که قورباغه با n جهش از طریق آن‌ها از A به E برسد. ثابت کنید

$$a_{۲n} = \frac{۱}{\sqrt{۲}}(x^{n-۱} - y^{n-۱})$$

که در آن $x = ۲ + \sqrt{۲}$ و $y = ۲ - \sqrt{۲}$.[۳۹]

اثبات: فرض کنید b_n تعداد مسیرهایی باشد که قورباغه از آن‌ها با n جهش از A به E برسد. اگر قورباغه بخواهد از A به E برود، در دو جهش اول یا به C می‌رسد، یا به G، و یا به A برمی‌گردد. به دو طریق می‌تواند به A بازگردد: $[A\ H\ A]$ یا $[A\ B\ A]$. حال از جایی که اکنون به آن رسیده باید شروع کند و با $n-۲$ جهش به E برسد، بنابراین داریم: $a_n = ۲b_{n-۲} + ۲a_{n-۲}$. در مورد b_n نیز به‌طور مشابه می‌توان دید که $b_n = ۲b_{n-۲} + a_{n-۲}$.

برای حل این رابطه‌ها دقت کنید که چون بین A و E، ۴ ضلع وجود دارد، و جهش‌های رفت و برگشت با هم تعداد زوجی می‌سازند، برای رفتن از A به E حتماً تعداد

[۳۹] این یکی از مسئله‌های بیست و یکمین المپیاد جهانی ریاضی بود.

زوجی حرکت لازم است. پس $a_{۲n-۱} = ۰$. برای حالت‌های زوج، دو رابطه‌ی بازگشتی $a_n = ۲b_{n-۲} + ۲a_{n-۲}$ و $b_n = ۲b_{n-۲} + a_{n-۲}$ را داریم.

برای حل، از تفاضل دو رابطه داریم: $b_{n-۲} = a_{n-۲} - a_{n-۴}$. در نتیجه با گذاشتن در رابطه‌ی اولی داریم: $a_n = ۴a_{n-۲} - ۲a_{n-۴}$. حال اگر $c_n = a_{۲n}$ را در نظر بگیریم، رابطه‌ی همگن $c_n = ۴c_{n-۱} - ۲c_{n-۲}$ $(n > ۲)$ به‌دست می‌آید که با حل معادله‌ی سرشت‌نما و توجه به حالت‌های اولیه‌ی $c_۱ = ۰$ و $c_۲ = ۲$ خواهیم داشت

$$a_{۲n} = c_n = \frac{۱}{\sqrt{۲}}((۲ + \sqrt{۲})^{n-۱} - (۲ - \sqrt{۲})^{n-۱})$$

مشابه مثال ۳-۱۸، به دلیل این‌که $۱ > \sqrt{۲} - ۲$، مشاهده می‌کنیم که

$$a_{۲n} = \langle \frac{(۲ + \sqrt{۲})^{n-۱}}{\sqrt{۲}} \rangle. \qquad (۳۹-۳)$$

□

تمرین‌های بخش ۳-۶

۳-۶.۱ رابطه‌ی بازگشتی زیر را به‌طور کامل حل کنید.

$$\begin{aligned} G_۰ &= ۱, \\ G_۱ &= ۲, \\ G_۲ &= ۴ \\ G_n &= G_{n-۱} + ۲G_{n-۲} - ۲G_{n-۳}, \quad n \geq ۳ \end{aligned}$$

۳-۶.۲ رابطه‌ی بازگشتی

$$T(n) = T(n-۱) + ۲T(n-۲) - ۲T(n-۳)$$

برای $n \geq ۳$ را تا حد امکان به‌صورت دقیق حل کنید. برای $۰ \leq n \leq ۲$ داریم: $T(n) = ۹n^۲ - ۱۵n + ۱۰۶$.

⋆ **۳-۶.۳** روی محیط دایره‌ای چند کارتِ کوچک سیاه و سفید قرار داده‌ایم. دو نفر به‌نوبت این عمل را انجام می‌دهند: اولی همه‌ی کارت‌های سیاهی را که در مجاورت یک کارت سفید باشند، بر می‌دارد و دومی برای کارت‌های سفید مجاور کارت‌های سیاه همین کار را انجام می‌دهد. اگر ۴۰ کارت وجود داشته باشد، آیا ممکن است پس از انجام ۲ حرکت فقط یک کارت باقی بماند؟ اگر کارت‌ها ۱۰۰۰ تا باشد، دست‌کم چند حرکت لازم است؟ اگر n کارت دور دایره باشد، دست‌کم چند حرکت لازم است؟

۳-۷ تحلیل سرشکنی

تا کنون با تحلیل الگوریتم‌ها در بدترین حالت و حالت میانگین آشنا شدیم. اما گاهی با دنباله‌ای از اعمال بر روی داده‌ساختاری سروکار داریم که هزینه‌ی برخی از آن‌ها در مواردی خیلی زیاد اما تعداد این موارد به نسبت کم است، یا اعمالی که هزینه‌ی آن‌ها کم است ولی تعداد آن‌ها زیاد است، یا این‌که انجام یک عمل که هزینه‌ی زیادی دارد موجب می‌شود تا همان عمل در مراحل بعد با هزینه‌ی کم‌تری انجام شود. در این موارد، هزینه‌ی یک عمل در بدترین حالتش زیاد است، اما اگر مجموع هزینه‌ها را برای همه‌ی اعمال موجود در دنباله حساب کنیم و بر تعداد آن‌ها تقسیم کنیم — یعنی میانگین هزینه‌ی اعمال در بدترین حالت را به‌دست بیاوریم — هزینه‌ی معقول‌تری نسبت به هزینه در بدترین حالت به‌دست آورده‌ایم، که به آن «هزینه‌ی سرشکن‌شده[۴۰]»، یا هزینه‌ی سرشکنی، و به روش محاسبه‌ی آن «تحلیل سرشکنی[۴۱]» می‌گوییم.

تعریف ۳-۷ یک داده‌ساختار و دنباله‌ای از اعمال مختلف بر روی آن را در نظر بگیرید. هزینه‌ی سرشکن‌شده‌ی هر عمل برابر میانگین هزینه‌ی بدترین حالت آن عمل در این دنباله تعریف می‌شود.

به تفاوت بین هزینه‌ی سرشکن‌شده و هزینه‌ی میانگین یک عمل توجه کنید. در هزینه‌ی میانگین، فرض بر آن است که ورودی با یک تابع توزیع مشخص (معمولاً یک‌نوا) ظاهر می‌شود. به‌ازای هر ورود، الگوریتم هزینه‌ای قابل محاسبه دارد. میانگین این هزینه‌ها برابر هزینه‌ی میانگین آن عمل بر روی ورودی با تابع توزیع داده شده است. در صورتی‌که در تحلیل سرشکنی، از توزیع داده خبری نیست و ما هزینه‌ی هر یک از اعمال موجود در دنباله‌ی اعمال را در بدترین حالت آن در نظر می‌گیریم؛ دنباله‌ی اعمال را هم در بدترین حالت فرض می‌کنیم.

مثال ۱: پشته با عمل «حذف چندگانه»

فرض کنید که بر روی پشته‌ی S علاوه بر اعمال عادی درج (PUSH) و حذف (POP)، عمل حذف چندگانه (MULTIPOP) هم انجام می‌شود. این عمل پارامتر k را می‌گیرد و

[۴۰] amortized cost
[۴۱] amortized analysis

به‌ترتیب k عنصر بالای پشته را حذف می‌کند؛ اگر کم‌تر از این تعداد عنصر در پشته موجود باشد، همه‌ی عناصر به‌ترتیب حذف می‌شوند.

```
MULTIPOP (S, k)
    ▷ عنصر بالای پشته‌ی S یا همه‌ی آن‌ها را به‌ترتیب حذف می‌کند k
1   while not STACK-EMPTY(S)  and k ≠ 0
2       do POP(S)
3          k ← k − 1
```

می‌دانیم که اعمال درج و حذف در بدترین حالت در زمان $\mathcal{O}(1)$ انجام می‌شود. بنابراین هزینه‌ی انجام عمل $\text{MULTIPOP}(S, k)$ برابر $\Theta(\min\{k, \text{SIZE}(S)\})$ است که می‌تواند در بدترین حالت $\mathcal{O}(n)$ باشد. بنابراین اگر دنباله‌ای از n تا از این اعمال داشته باشیم، کل هزینه در بدترین حالت $\mathcal{O}(n^2)$ می‌شود.

این نتیجه‌گیری درست است، اما دقیق نیست. ما در همین بخش نشان می‌دهیم که جمع هزینه‌ی n تا از این اعمال ــ هر ترتیبی که داشته باشند ــ حداکثر $\mathcal{O}(n)$ است. یعنی هزینه‌ی سرشکن‌شده برای هر کدام از آن‌ها، از جمله عمل حذف چندگانه، $\mathcal{O}(1)$ است.

مثال ۲: افزایش شمارنده‌ی دودویی

یک شمارنده‌ی دودویی k بیتی $A[k-1..0]$ را در نظر بگیرید (بیت با اندیس ۰ کم‌ارزش‌ترین بیت است) که مقدار اولیه‌ی آن صفر است. تنها عملی که بر روی این داده‌ساختار انجام می‌شود «افزایش» یک واحد به آن است. مثلاً اگر مقدار شمارنده 010010011111 باشد، پس از یک بار افزایش مقدار آن به 010010100000 تغییر می‌کند. در این مثال، یک بیت از 0 به 1 و پنج بیت از 1 به 0 تغییر می‌کنند. اگر هزینه‌ی هر تغییر بیت را ۱ واحد بگیریم، در این‌جا هزینه‌ی افزایش ۶ واحد خواهد بود.

رویه‌ی INCREMENT عمل افزایش را پیاده‌سازی می‌کند. چنان‌چه گفته‌شد، اندیس ۰ کم‌ارزش‌ترین بیت و اندیس $length[A] - 1$ پر ارزش‌ترین بیت در A است. توجه کنید که شمارنده به‌صورت چرخه‌ای عمل می‌کند؛ یعنی از آرایه‌ی تماماً 1 به آرایه‌ای که همه‌ی بیت‌های آن 0 است، می‌رود.

```
INCREMENT (A)
1  i ← 0
2  while i < length[A] and A[i] = 1
3      do A[i] ← 0
4         i ← i + 1
5  if i < length[A]
6     then A[i] ← 1
```

روشن است که هر عمل افزایش، در بدترین حالت همه‌ی k بیت را تغییر می‌دهد و بنابراین هزینه‌ی n بار عمل افزایش $\mathcal{O}(nk)$ است. در این‌جا نشان می‌دهیم که هزینه‌ی هر عمل افزایش به‌صورت سرشکنی $\mathcal{O}(۱)$ است.

۳-۷-۱ روش‌های تحلیل سرشکنی

تحلیل سرشکنی به سه روش انجام می‌شود:

- روش انبوهه[۴۲]: در این روش مطابق تعریف، جمع هزینه‌های اعمال محاسبه می‌شود و بر تعداد آن‌ها تقسیم می‌شود تا هزینه‌ی سرشکن‌شده به‌دست آید.
- روش حساب‌داری[۴۳]: در این روش به‌ازای هر عمل، مقداری پول پرداخت می‌شود. از این مقدار مبلغی صرف انجام واقعی عمل مورد نظر می‌شود و مازاد به مخزن پول اضافه می‌شود. اگر مقدار پول پرداختی برای انجام آن عمل کافی نباشد، از مخزن پول برداشت می‌شود. اگر مخزن هیچ‌وقت کم‌بود پول نداشته باشد، میزان پول پرداختی برای هر عمل با هزینه‌ی سرشکن‌شده‌ی آن عمل متناسب است.
- روش تابع پتانسیل[۴۴]: حالت کلی‌تر روش حساب‌داری است که در آن مخزن پول و میزان پول پرداختی برای هر عمل به‌صورت دقیق با فرمول‌های ریاضی بیان و تحلیل می‌شود. توضیحات بیش‌تر در ادامه خواهد آمد.

[۴۲] aggregate
[۴۳] accounting
[۴۴] potential function

تحلیل پشته با روش انبوهه

در این بخش نشان می‌دهیم که چگونه می‌توان پشته با عمل حذف چندگانه را با روش انبوهه، تحلیل سرشکنی کرد.

اگر n عمل داشته باشیم، می‌دانیم که حداکثر تعداد عناصر پشته، اگر همه درج باشند، n است. هم‌چنین می‌دانیم که هر عنصر داخل پشته دقیقاً یک‌بار درج و حداکثر یک بار حذف می‌شود و یک عنصر یا مستقیماً با عمل POP حذف می‌شود و یا با MULTIPOP. پس در مجموع حداکثر $2n$ بار عمل درج و حذف عادی انجام می‌شود. هزینه‌ی هر عمل PUSH و POP هم $\mathcal{O}(1)$ است، پس هزینه‌ی سرشکن‌شده‌ی هر عمل، متناسب با ۲ یا $\mathcal{O}(1)$ است.

تحلیل شمارنده با روش انبوهه

فرض کنیم که عمل افزایش را n بار بر روی یک شمارنده‌ی k بیتی که در ابتدا صفر است انجام می‌دهیم. می‌دانیم که هر عمل افزایش حداکثر یک بیت را از 0 به 1 و تعدادی بیت را از 1 به 0 تغییر می‌دهد. در نتیجه هزینه‌ی واقعی آن متناسب با جمع این دو عدد است. تعداد کل تغییرات بیت‌ها را با روش زیر می‌شماریم. مانند قبل، بیت‌ها را از ۰ تا $k-1$ شماره‌گذاری می‌کنیم که بیت ۰ کم‌ارزش‌ترین بیت است. روشن است که،

- بیت ۰ با هر افزایش تغییر می‌کند، یعنی در مجموع n بار،
- بیت ۱ یک در میان تغییر می‌کند، بنابراین در مجموع دقیقاً $\lceil n/2 \rceil$ بار،
- بیت ۲ هر ۴ بار یک دفعه تغییر می‌کند، بنابراین در مجموع دقیقاً $\lceil n/4 \rceil$ بار،
- به‌طور کلی بیت i ام پس از هر 2^i بار عمل افزایش، یک‌بار تغییر می‌کند. یعنی در مجموع دقیقاً $\lceil \frac{n}{2^i} \rceil$ بار.

پس تعداد تغییر بیت‌ها در مجموع برابر است با

$$\sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

یعنی هزینه‌ی سرشکن‌شده‌ی هر عمل افزایش، متناسب با ۲ یا $\mathcal{O}(1)$ است.

تحلیل پشته به روش حساب‌داری

در این روش فرض می‌کنیم یک مخزن پول داریم که در ابتدا خالی است. اگر برای هر عمل درج ۲ ریال پرداخت کنیم، می‌توانیم ۱ ریال آن‌را صرف عمل PUSH کنیم و ۱ ریال باقی‌مانده را در مخزن برای عنصر درج‌شده در پشته قرار دهیم. بدیهی است که پس از مدتی برای هر عنصر موجود در پشته ۱ ریال در مخزن داریم . این ۱ ریال می‌تواند صرف انجام عمل حذف آن عنصر شود، چه حذف در اثر POP باشد و چه به‌خاطر عمل MULTIPOP. یعنی در واقع عمل حذف مجانی انجام می‌شود. بنابراین اگر n عمل بر روی پشته انجام شود، حداکثر $۲n$ ریال هزینه شده است. بنابراین هزینه‌ی سرشکن‌شده‌ی هر عمل ۲ یا $\mathcal{O}(۱)$ است. توجه کنید که در این مثال ۲ ریال از مقدار $۲n/n$ در روش انبوهه به‌دست آمده است.

تحلیل شمارنده به روش حساب‌داری

برای این داده‌ساختار، به‌ازای هر عمل افزایش، ۲ ریال هزینه می‌کنیم. می‌دانیم که حداکثر یک بیت از 0 به 1 تغییر می‌کند (اگر همه‌ی بیت‌ها 1 باشند، پس از افزایش همه 0 می‌شوند)؛ ۱ ریال را صرف این تغییر می‌کنیم و ۱ ریال باقی‌مانده را در مخزن و برای همین بیت 1 قرار می‌دهیم. بنابراین پس از مدتی برای هر کدام از بیت‌های 1، ۱ ریال در مخزن قرار دارد. پس عمل تغییر 1 به 0 آن بیت مجانی انجام می‌شود. بنابراین اگر n بار عمل افزایش را انجام دهیم دقیقاً $۲n$ ریال هزینه کرده‌ایم و ممکن است در انتها مقداری پول هم در مخزن باقی بماند. در هر حال، هزینه‌ی سرشکن‌شده‌ی هر عمل افزایش، ۲ یا $\mathcal{O}(۱)$ است.

۳-۷-۲ روش تابع پتانسیل

در واقع این روش بیان ریاضی روش حساب‌داری برای تحلیل سرشکنی در حالت کلی است. فرض کنید $D_۰$ داده‌ساختار اولیه و D_i داده‌ساختار پس از عمل iام است. به‌عبارت دیگر،

$$D_۰ \xrightarrow{\text{عمل ۱}} D_۱ \xrightarrow{\text{عمل ۲}} D_۲ \ldots \xrightarrow{\text{عمل } n} D_n.$$

همچنین فرض کنید که c_i هزینه‌ی واقعی عمل iام است. در این‌صورت تابع پتانسیل $\Phi(D_i)$ را تعریف می‌کنیم که D_i را به یک عدد حقیقی نگاشت می‌کند. هم‌چنین $\hat{c}$ را مطابق

فرمول زیر تعریف می‌کنیم:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \qquad (۴۰-۳)$$

و آن‌را هزینه‌ی سرشکن‌شده‌ی عمل i ام می‌نامیم. در این‌صورت داریم

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0). \qquad (۴۱-۳)$$

اگر $\Phi(D_0) = 0$، داریم

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i. \qquad (۴۲-۳)$$

یعنی $\sum_{i=1}^{n} \hat{c}_i$ کران بالایی برای $\sum_{i=1}^{n} c_i$ است که می‌خواهیم به‌دست آوریم.
تناظر این روش با روش حساب‌داری به‌صورت زیر است:

- $\Phi(D_i)$: مقدار پول موجود در مخزن پس از عمل i ام.
- $\hat{c}_i$: مقدار پولی که برای عمل i ام پرداخت می‌کنیم.
- c_i: مقدار هزینه‌ی صرف‌شده برای عمل i ام.
- اگر $c_i < \hat{c}_i$، مابه‌التفاوت به مخزن اضافه می‌شود: $\Phi(D_i) = \Phi(D_{i-1}) + \hat{c}_i - c_i$.
- اگر $c_i > \hat{c}_i$، برای انجام عمل i لازم است به‌اندازه‌ی $c_i - \hat{c}_i$ از مخزن برداریم تا بتوانیم این عمل را انجام دهیم. پس موجودی در مخزن $\Phi(D_i) = \Phi(D_{i-1}) - (c_i - \hat{c}_i)$ می‌شود.

برای استفاده از این روش باید تابع پتانسیلی تعریف کنیم که دارای شرایط گفته‌شده باشد و با آن بتوانیم مقادیر $\hat{c}_i$ را برای اعمال مختلف پیدا کنیم. به‌عنوان مثال، به تحلیل داده‌ساختارهایی که قبلاً تعریف کرده‌ایم با این روش توجه کنید.

تحلیل پشته به روش پتانسیل

در این روش $\Phi(D_i)$ را برابر تعداد عناصر موجود در پشته می‌گیریم. بدیهی است که در ابتدا $\Phi(D_0) = 0$ و همیشه مقدارش مثبت است، پس Φ یک تابع پتانسیل است. حال به‌ازای هر عمل، هزینه‌ی سرشکن‌شده را به‌دست می‌آوریم:

- برای عمل PUSH، چون یک عنصر اضافه می‌شود، داریم: $\Phi(D_i) - \Phi(D_{i-1}) = 1$. و می‌دانیم که هزینه‌ی واقعی درج $c_i = 1$ است. پس،

$$\hat{c}_i = \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2.$$

- برای عمل POP، چون یک عنصر کم می‌شود، داریم

$$\Phi(D_i) - \Phi(D_{i-1}) = -1.$$

هزینه‌ی واقعی حذف هم $c_i = 1$ است، پس $\hat{c}_i = 1 - 1 = 0$.

- عمل MULTIPOP هم تعدادی POP است. پس هزینه‌ی سرشکن‌شده‌ی آن هم صفر است.

دقت کنید که این اعداد به‌دست آمده همان مقدار پولی است که در روش حساب‌داری برای انجام این اعمال می‌پرداختیم. بنابراین،

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i = 2n,$$

یعنی هزینه‌ی سرشکن‌شده‌ی هر عمل $\mathcal{O}(1)$ است.

تحلیل شمارنده به روش پتانسیل

برای این داده‌ساختار تابع پتانسیل $\Phi(D_i)$ را برابر تعداد یک‌ها در آرایه‌ی A می‌گیریم. (همان مقدار پولی که در مخزن در روش حساب‌داری موجود است.) به این ترتیب، $\Phi(D_0) = 0$ و همیشه $\Phi(D_i) \geq 0$. بنابراین، تعریف درست است.

یک عمل افزایش را در نظر بگیرید. فرض کنید که در این عمل t_i بیت از 1 به 0 و حداکثر یک بیت از 0 به 1 تبدیل می‌شود. پس،

$$\left.\begin{aligned} &\hat{c}_i = \Phi(D_i) - \Phi(D_{i-1}) + c_i \\ &\Phi(D_i) - \Phi(D_{i-1}) \leq -t_i + 1 \\ &c_i \leq t_i + 1 \end{aligned}\right\} \rightarrow \hat{c}_i \leq 2.$$

نتیجه می‌گیریم که هزینه‌ی سرشکن‌شده‌ی هر عمل افزایش، ۲ یا $\mathcal{O}(1)$ است.

در بخش ۵-۸، درهم‌سازی پویا را با این روش تحلیل می‌کنیم که مثال بزرگ‌تری است و اهمیت این روش را به‌خوبی نشان می‌دهد.

تمرین‌های بخش ۳-۷

۱.۷-۳ نشان دهید اگر به شمارنده‌ی دودویی عمل RESET (یعنی صفر کردن شمارنده) را هم اضافه کنیم باز هم هزینه‌ی سرشکن‌شده‌ی همه‌ی اعمال $\mathcal{O}(1)$ است.

۲.۷-۳ نشان دهید اگر عمل DECREMENT را نیز به شمارنده اضافه کنیم، دیگر هزینه‌ی سرشکنی لزوماً $\mathcal{O}(1)$ نیست. ولی اگر از یک سیستم نمایشی دیگر که در آن هر رقم ۰، ۱ یا ۲ است استفاده کنیم، می‌توان این دو عمل را طوری پیاده‌سازی کرد که هزینه‌ی سرشکنی هر یک ثابت باشد. مثلاً در این سیستم نمایش داریم

$$(۱۲۱۰) = ۰ \times ۲^۰ + ۱ \times ۲^۱ + ۲ \times ۲^۲ + ۱ \times ۲^۳ = (۱۰۰۱۰) = ۱۸.$$

۳.۷-۳ داده‌ساختاری از اعداد معرفی کنید که عمل درج یک عنصر و عمل حذف $n/۲$ عنصر بزرگ را به صورت سرشکن‌شده در زمان ثابت انجام دهد.

۴.۷-۳ یک شمارنده‌ی دودویی متفاوت m بیتی $b_m b_{m-۱} \dots b_۱$ را در نظر بگیرید که در آن تغییر بیت i (b_i) هزینه‌ای برابر i دارد. یعنی اگر شمارنده از عدد 10111 به 11000 افزایش یابد، هزینه‌ای برابر $۱۰ = ۴ + ۳ + ۲ + ۱$ خواهد داشت. نشان دهید که هزینه‌ی عمل افزایش در این شمارنده‌ی دودویی نیز $\mathcal{O}(1)$ است. برای این کار تابع

$$\Phi(b) = \sum_{i=۱}^{m} (i+۲) b_i$$

را به‌عنوان تابع پتانسیل پیش‌نهاد می‌کنیم که در آن $b = b_m b_{m-۱} \dots b_۱$. بقیه‌ی کار با شما!

⋆ **۵.۷-۳** m سطل و $n < m$ توپ در اختیار داریم. در ابتدا، هر کدام از n سطل یک توپ دارند و بقیه خالی هستند. شما می‌توانید با تعدادی «حرکت» توپ‌ها را بین سطل‌ها توزیع کنید. در هر حرکت، یک سطل به‌نام b_i با k_i توپ انتخاب و همه‌ی توپ‌های آن‌را بین k_i سطل دیگر توزیع می‌کنید تا به هر کدام یکی از این توپ‌ها برسد؛ یعنی b_i همه‌ی توپ‌هایش را از دست می‌دهد و به k_i سطل هر کدام دقیقاً یک توپ اضافه می‌شود. هزینه‌ی این حرکت را برابر k_i می‌گیریم. بدیهی است که مجموع توپ‌ها همیشه برابر n است ($\sum_{i=۱}^{n} k_i = n$).

الف) ترتیب نامحدودی از حرکت‌ها را معین کنید تا بعد از تعدادی حرکت اولیه، هزینه‌ی هر حرکت $\mathcal{O}(\sqrt{n})$ شود. (راهنمایی: راه‌حل شما ممکن است دقیقاً $\lfloor \sqrt{۲n + \frac{۱}{۴}} - \frac{۱}{۲} \rfloor$ شود.)

ب) با تعریف یک تابع پتانسیل مناسب ثابت کنید که هزینه‌ی سرشکن‌شده‌ی هر حرکت حداکثر $۲\sqrt{n}$ است. (راهنمایی: تابع پتانسیل برای هر سطل با k_i توپ را برابر $\max\{۰, k_i - t\}$ بگیرید که در آن t یک ثابت دل‌خواه برای همه‌ی سطل‌هاست.)

تمرین‌های فصل ۳

۱.۳ در برخی کتاب‌ها، Ω را با اندکی تفاوت به‌گونه‌ی دیگری تعریف می‌کنند. در این‌جا از نماد Ω_∞ برای این تعریف استفاده می‌کنیم. می‌گوییم $f(n) = \Omega_\infty(g(n))$ اگر یک ثابت مثبت c وجود داشته‌باشد به‌طوری‌که برای بی‌نهایت عدد صحیح n، داشته باشیم

$$f(n) \geq cg(n) \geq 0.$$

الف) نشان دهید برای هر دو تابع $f(n)$ و $g(n)$ که به‌طور مجانبی نامنفی هستند، $f(n) = O(g(n))$ یا $f(n) = \Omega_\infty(g(n))$ یا هر دو این روابط برقرار هستند، در حالی که اگر به‌جای Ω_∞ از Ω استفاده کنیم، وضع بدین گونه نخواهد بود.

ب) مزایا و معایب ممکنِ استفاده از Ω_∞ به‌جای Ω را برای مشخص‌کردن زمان اجرای برنامه‌ها بیان کنید.

در برخی از کتاب‌ها نیز O را اندکی متفاوت تعریف می‌کنند که ما آن‌را با نماد O' نمایش می‌دهیم. می‌گوییم $f(n) = O'(g(n))$ اگر و فقط اگر $|f(n)| = O(g(n))$.

پ) برای هر یک از جهت‌های «اگر و فقط اگر» در قضیه‌ی ۳-۱ اگر با همان تعریف قبلی Ω، O' را به‌جای O جای‌گزین کنیم چه اتفاقی خواهد افتاد؟

در برخی کتاب‌ها $\tilde{O}$ را برای بیان مفهوم O با نادیده گرفتن عوامل لگاریتمی استفاده می‌کنند، یعنی:

$$\tilde{O}(g(n)) = \{f(n) : \exists c > 0, k > 0, n_0 > 0, s.t. \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\lg^k(n)\}$$

ت) $\tilde{\Omega}$ و $\tilde{\Theta}$ را به‌طور مشابه تعریف و برای آن‌ها مشابه قضیه‌ی ۳-۱ را ثابت کنید.

۲.۳ عمل‌گر تکرار $*$ که در تابع $\lg^*$ به‌کار رفته است، می‌تواند به هر تابع یک‌نوای صعودی روی مجموعه اعداد حقیقی اعمال شود. برای ثابت داده شده‌ی $c \in R$، تابع مکرر f_c^* را به این صورت تعریف می‌کنیم:

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

که لزوماً در همه‌ی حالات خوش‌تعریف نیست. به‌عبارت دیگر، $f_c^*(n)$ تعداد دفعات تکرار تابع f است که ورودی آن را به c یا کم‌تر کاهش می‌دهد. برای هر یک از تابع‌های $f(n)$ و ثابت c که در جدول آمده است، کرانی تا حد ممکن دقیق برای $f_c^*(n)$ ارائه دهید.

	$f(n)$	c	$f_c^*(n)$
الف	$n-1$	۰	
ب	$\lg n$	۱	
پ	$\frac{n}{2}$	۱	
ت	$\frac{n}{2}$	۲	
ث	$\sqrt{n}$	۲	
ج	$\sqrt{n}$	۱	
چ	$n^{\frac{1}{3}}$	۲	
ح	$\frac{n}{\lg n}$	۲	

۳.۳ فرض کنید $A[1\cdots n]$ آرایه‌ای از n عدد متمایز باشد. زوج (i,j) را یک «وارونگی» می‌گوییم اگر $i<j$ و $A[i]>A[j]$.

الف) تعداد وارونگی‌های آرایه‌ی $[2,3,6,8,1,5,7,9]$ چند تاست؟

ب) حداکثر تعداد وارونگی‌های یک آرایه‌ی n عنصره چندتاست؟ محاسبه کنید.

پ) چه آرایه‌ای از اعداد $\{1,2,\ldots,n\}$ بیش‌ترین تعداد وارونگی را دارد؟ چرا؟

ت) چه تعداد آرایه‌ی n عنصره‌ی A با عناصر ۱ تا n بیش‌ترین تعداد وارونگی را دارند؟ چرا؟

ث) اگر تعداد وارونگی‌های آرایه‌ی n عنصری A برابر m باشد، زمان اجرای الگوریتم مرتب‌ساز درجی روی این آرایه در بدترین حالت چقدر است؟ اثبات کنید.

ج) الگوریتمی ارائه و تحلیل کنید که تعداد وارونگی‌های یک آرایه‌ی n عنصری A را در $O(n\lg n)$ تعیین کند؟

⋆ **۴.۳** تعدادی بطری نوشابه کاملاً مشابه و یک نردبان n پله‌ای داریم. می‌خواهیم بدانیم که از کدام پله به‌بالا اگر یک بطری را رها کنیم، بر روی زمین خواهد شکست. یک «آزمون» یعنی رها کردن یک بطری از یکی از پله‌های نردبان. اگر برای به‌دست آوردن جواب بخواهیم کم‌ترین تعداد بطری‌ها شکسته شود، می‌توانیم از پایین‌ترین پله شروع کنیم و بالا برویم و در هر پله، عمل آزمون را انجام دهیم. اولین پله‌ای که بطری بشکند، جواب است. اگر بخواهیم فقط تعداد آزمون‌ها کمینه شود، از روش دودویی جست‌وجو می‌توانیم جواب را به‌دست آوریم. به‌این ترتیب که ابتدا در پله‌ی وسط آزمون را انجام می‌دهیم. اگر بطری شکست که همین کار را در نیمه‌ی پایین‌تر وگرنه در نیمه‌ی بالاتر انجام می‌دهیم و همیشه پله‌ی وسطی را بین پله‌هایی که جواب در یکی از آن‌ها واقع است انتخاب می‌کنیم.

الف) تعداد دقیق آزمون‌ها در روش دوم را برحسب n به‌دست آورید.

ب) حداکثر چه تعداد بطری ممکن است شکسته شود؟ محاسبه کنید.

پ) فرض کنید برای محاسبه‌ی شماره‌ی پله‌ی بیان‌شده، روشی مدنظر است که حداکثر k تا آزمون انجام شود. فرض کنید $f_k(n)$ حداکثر تعداد آزمون‌ها در این مسئله است. بدیهی است

که $f_1(n) \leq n$. الگوریتمی را بیان کنید که $f_2(n) = o(n)$ (دقت کنید که o کوچک است).

ت) الگوریتمی در حالت کلی بیان کنید که در آن $f_k(n) = o(f_{k-1}(n))$.

۵.۳ پروفسور دیوجنز n تراشه‌ی بسیار مجتمع دارد که شبیه هم فرض می‌شوند و می‌توانند یک‌دیگر را تحت آزمون قرار دهند. در هر بار آزمون دو تراشه مورد آزمایش قرار می‌گیرند و هر یک از آن‌ها می‌گوید که دیگری سالم یا معیوب است. یک تراشه‌ی درست همیشه درست تشخیص می‌دهد اما به نتیجه‌ی آزمایش با یک تراشه‌ی معیوب نمی‌توان اعتماد کرد. بنابراین چهار حالت زیر ممکن است پیش آید:

A چه می‌گوید؟	B چه می‌گوید؟	نتیجه‌ای که می‌توان گرفت
B سالم است	A سالم است	هر دو سالم یا هر دو معیوب‌اند
B سالم است	A معیوب است	حداقل یکی معیوب است
B معیوب است	A سالم است	حداقل یکی معیوب است
B معیوب است	A معیوب است	حداقل یکی معیوب است

الف) نشان دهید که اگر بیش از $\frac{n}{4}$ تراشه‌ها معیوب باشند، با استفاده از این روش آزمون، نمی‌توان دقیقاً مشخص کرد که کدام تراشه‌ها سالم و کدام‌ها معیوب هستند. فرض کنید که ممکن است برای گمراه کردن پروفسور، تراشه‌ها بدترین حالت ممکن را داشته باشند.

ب) مسئله‌ی پیدا کردن یک تراشه‌ی سالم از میان n تراشه را در حالتی که بیش از $\frac{n}{4}$ تراشه‌ها سالم هستند درنظر بگیرید. نشان دهید که در این حالت $\lfloor \frac{n}{4} \rfloor$ آزمون دوبه‌دو برای کاهش مسئله به مسئله‌ای با اندازه‌ی تقریباً نصف آن کافی است.

پ) نشان دهید با فرض این‌که بیش از $\frac{n}{4}$ تراشه سالم هستند، تراشه‌های سالم را می‌توان با $\Theta(n)$ آزمون دوبه‌دو شناسایی کرد. رابطه‌ی بازگشتی که تعداد آزمایش‌ها را مشخص می‌کند پیدا و آن‌را حل کنید.

۶.۳ در این کتاب فرض کرده‌ایم که انتقال پارامترها در فراخوانی رویه‌ها زمان ثابتی به خود اختصاص می‌دهد حتی اگر یک آرایه‌ی N عنصری انتقال داده شود. این فرض در بیش‌تر سیستم‌ها معقول است زیرا به‌جای آرایه، اشاره‌گری به آن انتقال داده می‌شود. این مسئله مفهوم سه روش انتقال آرایه را مورد بررسی قرار می‌دهد:

a) آرایه به‌صورت یک اشاره‌گر انتقال داده می‌شود. زمان $= \Theta(1)$.

b) آرایه با ایجاد نسخه‌ی مشابه منتقل می‌شود. زمان $= \Theta(N)$ که N اندازه‌ی آرایه است.

c) آرایه تنها با نسخه‌برداری از محدوده‌ای از آن که ممکن است مورد دست‌یابی قرار گیرد انتقال می‌یابد. زمان $\Theta(q - p + 1)$ تنها وقتی لازم است که قسمت $A[p \cdots q]$ انتقال داده شود.

الف) الگوریتم بازگشتی جست‌وجوی دودویی برای یافتن یک عدد در آرایه‌ی مرتب‌شده را در نظر بگیرید. برای زمان اجرای الگوریتم در بدترین حالت در هر یک از سه حالت بیان شده برای انتقال آرایه، رابطه‌ای بازگشتی پیدا کنید و برای هر رابطه‌ی بازگشتی کران بالای مناسبی بیابید. برای نمایش اندازه‌ی آرایه‌ی اصلی از N و برای بیان اندازه‌ی زیرمسئله‌ها از n استفاده کنید.

ب) قسمت «الف» را برای الگوریتم مرتب‌سازی ادغامی نیز انجام دهید.

⋆ **۷.۳** یک ماتریس $m \times n$ از اعداد حقیقی یک «آرایه‌ی مانژ[۴۵]» است اگر برای هر i، j، k و l که $1 \le i < k \le m$ و $1 \le j < l \le n$، داشته باشیم

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

به عبارت دیگر، هر دو سطر و دو ستونی که از یک آرایه‌ی مانژ انتخاب شود، برای چهار عنصر محل تلاقی این سطرها و ستون‌ها، مجموع دو عنصر بالا-چپ و پایین-راست کم‌تر یا مساوی با مجموع دو عنصر پایین-چپ و بالا-راست خواهد بود. به عنوان مثال آرایه‌ی زیر مانژ 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

الف) ثابت کنید که یک آرایه مانژ است اگر و تنها اگر برای تمام $i = 1, 2, \ldots, m-1$ و $j = 1, 2, \ldots, n-1$ داشته باشیم

$$A[i,j] + A[i+1,j+1] \le A[i,j+1] + A[i+1,j].$$

(توجه: برای قسمت «تنها اگر» از استقرا روی سطرها و ستون‌ها به‌طور جداگانه استفاده کنید.)

ب) آرایه‌ی زیر مانژ نیست. یکی از عناصر آن را تغییر دهید تا آرایه‌ی مانژ شود. (راهنمایی: از قسمت «الف» استفاده کنید.)

۳۷	۲۳	۲۲	۳۲
۲۱	۶	۷	۱۰
۵۳	۳۴	۳۰	۳۱
۳۲	۱۳	۹	۶
۴۳	۲۱	۱۵	۸

پ) فرض کنید $f(i)$ شماره‌ی ستون شامل چپ‌ترین عنصر کمینه در سطر i باشد. ثابت کنید برای هر آرایه‌ی مانژ $m \times n$، $f(1) \le f(2) \le \ldots \le f(m)$.

[۴۵] Monge

ت) این الگوریتم چپ‌ترین عنصر کمینه در هر ستون از یک آرایه‌ی مانژ $m \times n$ را محاسبه می‌کند:

- یک زیرماتریس از A به‌نام A' که شامل سطرهای با شماره‌ی زوج از ماتریس A است بساز.
- به‌طور بازگشتی چپ‌ترین عنصر کمینه را برای هر سطر از A' پیدا کن.
- در سطرهای با شماره‌ی فرد ماتریس A، چپ‌ترین عنصر کمینه را پیدا کن.

توضیح دهید که چگونه می‌توان در زمان $\mathcal{O}(m+n)$ چپ‌ترین عنصر کمینه را در سطرهای با شماره‌ی فرد، با دانستن چپ‌ترین عناصر کمینه در سطرهای با شماره‌ی زوج محاسبه کرد.

ث) یک رابطه‌ی بازگشتی برای زمان اجرای الگوریتم شرح داده‌شده در قسمت «ت» بنویسید. نشان دهید که جواب آن $\mathcal{O}(m+n\log m)$ است.

فصل

۴

داده‌ساختارهای ساده

در این فصل ابتدا داده‌ساختارهایی را که در این کتاب بررسی می‌شوند دسته‌بندی می‌کنیم. «داده‌ساختارهای ساده[۱]» مانند لیست، پشته، صف و درخت‌های معمولی در اکثر کتاب‌های مبانی کامپیوتر هم بررسی می‌شوند. به آن‌ها «صف اولویت[۲]» را اضافه می‌کنیم که برای اعمال درج، حذف کوچک‌ترین (بزرگ‌ترین) عنصر و نیز کاهش (افزایش) کلید یک عنصر استفاده می‌شود و پیاده‌سازی کارا و ساده‌ای دارد. «فرهنگ داده‌ای[۳]» به مجموعه‌ای از عناصر گفته می‌شود که بر روی آن‌ها به‌تکرار، اعمال درج و حذف یک عنصر و نیز جست‌وجو برای یافتن یک عنصر انجام می‌شود. در این کتاب دو داده‌ساختار ساده را برای پیاده‌سازی فرهنگ داده‌ای در نظر می‌گیریم. «درخت دودویی جست‌وجو[۴]» این اعمال را در بدترین حالت برحسب تعداد عناصر به‌صورت خطی و در حالت میانگین به‌صورت لگاریتمی انجام می‌دهد. «جدول درهم‌سازی[۵]» روش دیگری برای پیاده‌سازی فرهنگ داده‌ای است که در بهترین وضعیت اعمال مورد نظر را در زمان میانگینِ ثابت انجام می‌دهد.

داده‌ساختارهای گفته‌شده، بجز روش درهم‌سازی که به‌دلیل اهمیت آن به‌صورت مجزا

---

[۱] elementary data structures
[۲] priority queue
[۳] data dictionary
[۴] binary search tree
[۵] hashing table

در فصل ۵ بیان می‌شود، در همین فصل مورد بررسی قرار می‌گیرند.

داده‌ساختارهای پیچیده‌تر را با عنوان «داده‌ساختارهای پیشرفته» در فصل ۷ ارائه می‌کنیم. این شامل داده‌ساختارهای «مجموعه‌های مجزا[۶]» و «درخت دودویی جست‌وجوی بهینه[۷]» است که در مورد آن‌ها مفصل توضیح خواهیم داد. پس از آن گونه‌هایی از درخت دودویی جست‌وجو با ارتفاع لگاریتمی بیان می‌شوند. همچنین این فصل شامل درخت‌های کاملاً متوازن است که کاربردشان در داده‌ساختارهای حافظه‌ی خارجی است.

۴-۱ دسته‌بندی داده‌ساختارها

۱) داده‌ساختارهای ساده

- ◁ لیست[۸] (لیست ساده‌ی یک‌سویه، دوسویه، حلقه‌ای)
- ◁ پُشته[۹]
- ◁ صف[۱۰]
- ◁ درخت معمولی، درخت عبارت
- ◁ صف اولویت

۲) داده‌ساختارهای ساده برای فرهنگ داده‌ای (درج، حذف و جست‌وجو)

- ◁ درخت دودویی جست‌وجو (د.د.ج): حداکثر $\mathcal{O}(n)$ و میانگین $\mathcal{O}(\lg n)$
- ◁ درهم‌سازی[۱۱]: به‌صورت میانگین و در بهترین حالت در $\mathcal{O}(1)$

۳) داده‌ساختارهای پیشرفته

- ◁ مجموعه‌های مجزا
- ◁ درخت دودویی جست‌وجوی بهینه
- ◁ درخت‌های دودویی جست‌وجو با ارتفاع $\mathcal{O}(\lg n)$

---

[۶] disjoint sets
[۷] optimal binary search tree
[۸] list
[۹] stack
[۱۰] queue
[۱۱] hashing

- درخت قرمز-سیاه[۱۲] و گسترش‌های آن: درخت مرتبه‌ی آماری[۱۳] و درخت بازه[۱۴]
- درخت اِی.وی.اِل[۱۵]
  - ◁ درخت‌های کاملاً متوازن (با ارتفاع $\mathcal{O}(\lg n)$)
- درخت ۳-۲[۱۶]
- درخت بی[۱۷]

۲-۴ لیست‌ها

در حالت کلی، یک لیست تعدادی عنصر را با ترتیب مشخصی ذخیره می‌کند. در این لیست عنصر اول، عنصر بعد و قبل از هر عنصر، و اغلب، عنصر آخر مشخص و قابل دست‌رسی هستند. معمولاً، اعمال زیر توسط داده‌ساختار لیست انجام می‌شود:

- ایجاد لیست تهی (CREATE)،
- محاسبه‌ی تعداد عناصر موجود در لیست، یا اندازه‌ی لیست (SIZE)،
- درج یک عنصر در ابتدا یا انتهای لیست (INSERT-FIRST، INSERT-LAST)،
- درج یک عنصر بعد از یک عنصر داده‌شده (INSERT-AFTER)، و
- حذف یک عنصر از ابتدای لیست (DELETE-FIRST) یا حذف عنصر بعد از یک عنصر داده‌شده در لیست (DELETE-AFTER).

بسته به کاربرد لیست، ممکن است اعمال دیگری هم به این‌ها اضافه شوند، مثلاً مرتب‌سازی یک لیست، ادغام دو لیست مرتب، حذف عناصر تکراری در لیست.

لیست‌ها را می‌توان به‌صورت پیوندی[۱۸]، یک‌سویه[۱۹]، حلقه‌ای[۲۰]، دوسویه[۲۱]،

[۱۲] red-black tree
[۱۳] order-statistic tree
[۱۴] interval tree
[۱۵] AVL tree
[۱۶] 2-3 tree
[۱۷] B tree
[۱۸] linked list
[۱۹] unidirectional
[۲۰] circular
[۲۱] bidirectional

سلسله‌مراتبی[۲۲] و یا در حالت کلی[۲۳] پیاده‌سازی کرد. به لیست پیوندی یک‌سویه، لیست خطی[۲۴] هم گفته می‌شود.

بر اساس مکان درج یا حذف عناصر، داده‌ساختار لیست به اسامی زیر شناخته می‌شود:

- پُشته: درج و حذف فقط در یک انتهای لیست انجام می‌شود. در این‌صورت، اولین عنصری که از آن خارج می‌شود، آخرین عنصر درج شده است[۲۵]، یا اولین عنصر درج شده، آخرین عنصری خواهد بود که از آن خارج می‌شود.[۲۶]
- صف: درج فقط در انتها و حذف فقط از ابتدای لیست امکان‌پذیر است. در نتیجه، عناصر به همان ترتیبی که در صف درج شده‌اند از آن نیز خارج می‌شوند.[۲۷]

در این بخش، ابتدا پیاده‌سازی لیست‌های پیوندی ساده را بیان می‌کنیم. سپس پشته و صف را، هم با استفاده از آرایه و هم با لیست پیوندی پیاده‌سازی می‌کنیم. در پایان، چند کاربرد مهم از لیست‌ها را بررسی خواهیم کرد. تحلیل بازار بورس، مرتب‌سازی ادغامی، کار با عبارت‌های کلی چندجمله‌ای ریاضی، و نیز تبدیل خودکار رویه‌های بازگشتی به غیربازگشتی نمونه‌هایی از این کاربردها هستند.

۴-۲-۱ پیاده‌سازی لیست‌های پیوندی

هر عنصر یک لیست ساده‌ی یک‌سویه به نام *node*، دارای دو مؤلفه‌ی *element* و *next* است. مؤلفه‌ی *element* حاوی تمام اطلاعاتی است که می‌خواهیم در آن عنصر ذخیره کنیم. مثلاً، اگر عنصر حاوی اطلاعات یک فرد باشد، ممکن است نام، نام خانوادگی، آدرس و دیگر مشخصات او مورد نظر باشد. همچنین ممکن است عناصر لیست هر کدام مؤلفه‌ای یگانه (مانند *key*) داشته باشند که در صورت نیاز بتوان آن‌ها را برحسب این مؤلفه مرتب کرد. مؤلفه‌ی *next* هر عنصر آدرس عنصر بعدی است و تنها از این طریق می‌توان به آن عنصر دست‌رسی داشت. این مؤلفه عناصر مختلف لیست را به هم «پیوند» می‌دهد.

یک لیست را می‌توان این‌چنین با زبان جاوا پیاده‌سازی کرد:

---

[۲۲] hierarchical
[۲۳] general lists
[۲۴] linear list
[۲۵] Last-In-First-Out (LIFO)
[۲۶] First-In-Last-Out (FILO)
[۲۷] First-In-First-Out (FIFO)

```
class Node {
    private Object element;
    private Node next;
    // constructors
    Node(){
       this(null,null);
    }
    public Node(Object e, Node n){
       element = e
       next = n;
    }
    void setElement(Object newElem){ element = newElem;}
    void setNext(Node newNext){ next = newNext;}
    Object getElement(){ return element;}
    Node getNext() {return next;}
}
```

در پیاده‌سازی ممکن است یک عنصر اضافی را به‌نام «سرلیست[۲۸]» در ابتدای لیست قرار دهیم تا از ایجاد حالت‌های خاصِ لیست تهی جلوگیری کنیم. شکل ۴-۱ چند نمونه از لیست‌های پیوندی را نشان می‌دهد.

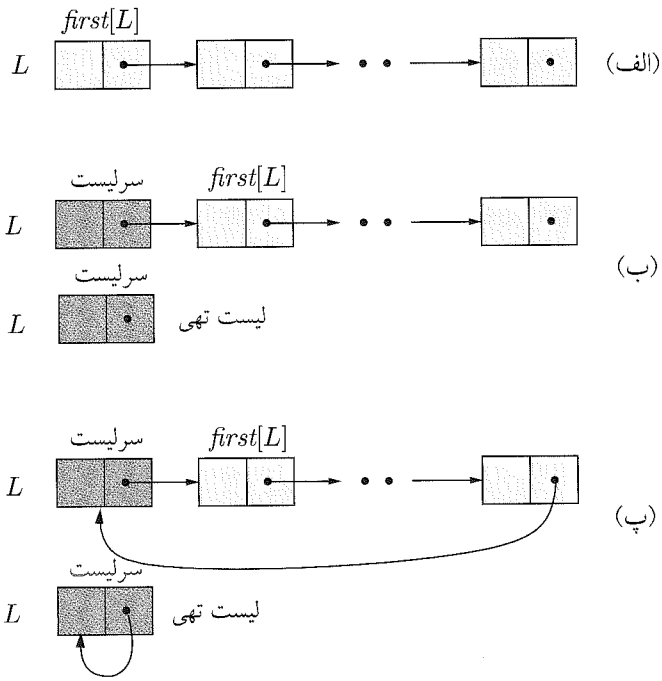
جدول زیر دستورهای به‌کار رفته در شبه‌کد استفاده‌شده در این کتاب (شبه‌کد CLRS) و نیز دستورهای معادل آن‌ها را در زبان جاوا نشان می‌دهد.

جاوا (Java)	شبه‌کد CLRS
`x = new Node()`	$x \leftarrow$ ALLOCATE-NODE()
`x = new Node(element e, next n)`	$x \leftarrow$ ALLOCATE-NODE(e, n)
`delete x`	FREE-NODE(x)
`x.getNext()`	$next[x]$
`x.setNext(n)`	$next[x] \leftarrow n$

۴-۲-۲ اعمال اصلی بر روی لیست خطی

یک لیست خطی L باید بتواند اعمال زیر را انجام دهد. در این‌جا فرض می‌شود x مقدار یا محتوای یک عنصر و n یک عنصر از لیست است. همچنین فرض می‌شود که مؤلفه‌ی $size[L]$ تعداد عناصر و $first[L]$ اولین عنصر لیست L را برمی‌گرداند.

[۲۸] sentinelt یا header



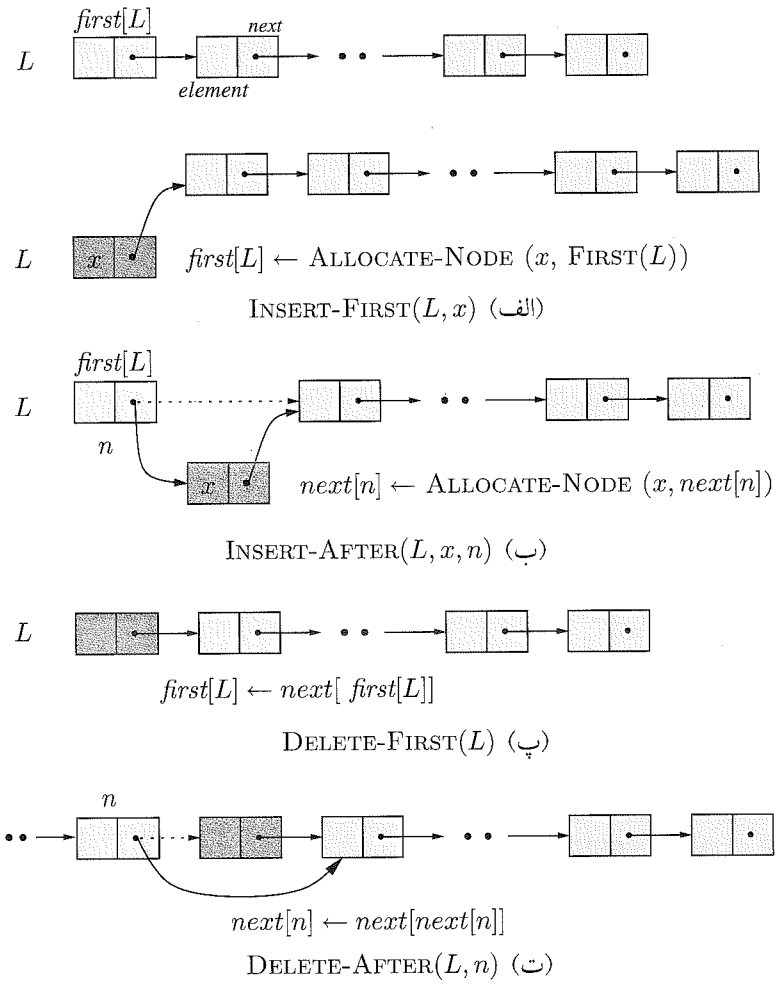
شکل ۴-۱ لیست‌های پیوندی خطی و حلقه‌ای. (الف) لیست پیوندی بدون سرلیست، (ب) لیست پیوندی با سرلیست، و (پ) لیست پیوندی حلقه‌ای با سرلیست.

- CREATE-LIST(L): لیست تهی L را ایجاد می‌کند.
- SIZE(L): تعداد عناصر لیست را برمی‌گرداند.
- FIRST(L): اولین عنصر لیست را برمی‌گرداند.
- ISEMPTY(L): مشخص می‌کند که آیا لیست خالی است یا خیر.
- INSERT-FIRST(L, x): یک عنصر با مقدار x را در ابتدای لیست درج می‌کند.
- INSERT-AFTER(L, x, n): یک عنصر با مقدار x را پس از عنصر n در L درج می‌کند.
- DELETE-FIRST(L): اولین عنصر لیست L را حذف می‌کند.
- DELETE-AFTER(L, n): عنصر پس از عنصر n را در L (در صورت وجود) حذف می‌کند.

رویه‌های مربوط به این اعمال در زیر نشان داده شده‌اند.

```
CREATE (L)
1  size [L] ← 0

SIZE (L)
1  return size[L]

FIRST (L)
1  if SIZE(L) ≠ 0
2     then return first[L]
3     else error list is empty

ISEMPTY (L)
1  return SIZE(L) = 0

INSERT-FIRST (L, x)
1  first[L] ← ALLOCATE-NODE (x, FIRST(L))
2  size[L] ← size[L] + 1

INSERT-AFTER (L, x, n)
1  if n = null
2     then error element is empty
3  next[n] ← ALLOCATE-NODE (x, next[n])
4  size[L] ← size[L] + 1

DELETE-FIRST (L)
1  if ISEMPTY (L)
2     then error list is empty
3  n ← FIRST(L)
4  first[L] ← next[n]
5  FREE-NODE(n)
6  size[L] ← size[L] − 1

DELETE-AFTER (L, n)
1  if ISEMPTY (L)  or n = null or next[n] = null
2     then error element does not exist
3  r ← next[n]
4  next[n] ← next[r]
5  FREE-NODE(r)
6  size[L] ← size[L] − 1
```

شکل ۴-۲ هم نحوه‌ی اجرای تعدادی از این رویه‌ها را نمایش می‌دهد. روشن است که هر یک از این اعمال در زمان $\mathcal{O}(1)$ انجام می‌شود.



شکل ۴-۲ انجام اعمال درج و حذف در یک لیست خطی یک‌سویه.

درج و حذف در لیست دوسویه‌ی خطی

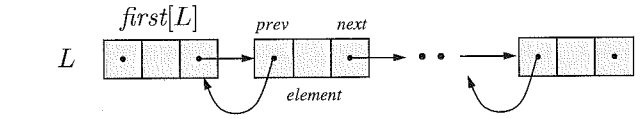
اعمال بیان‌شده را می‌توان بر روی گونه‌های دیگر لیست هم نوشت. در این‌جا برای مثال لیست دوسویه را درنظر می‌گیریم که در آن هر عنصر علاوه بر مؤلفه‌های لیست

یک‌سویه، مؤلفه‌ی *prev* هم دارد که به عنصر قبلی‌اش در لیست اشاره می‌کند. رویه‌ی ALLOCATE-NODE (x, p, n) عنصری با محتوای x ایجاد می‌کند که عنصر بعدی آن n و قبلی آن p باشد. برای جلوگیری از حالت‌های خاص، در این‌جا لیست را با سرلیست در نظر گرفته‌ایم. رویه‌های مربوط به INSERT-AFTER و DELETE-AFTER در زیر می‌آیند. شکل ۴-۳ هم نحوه‌ی انجام این اعمال را نشان می‌دهد.

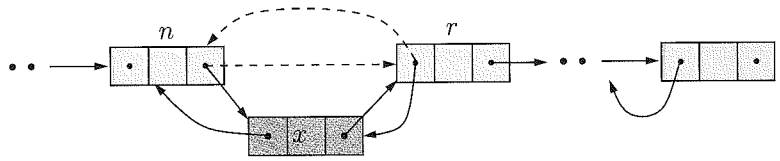
```
INSERT-AFTER (L, x, n)
    ▷ عنصری با محتوای x را پس از عنصر n در لیست دوسویه‌ی L درج می‌کند
 1  if ISEMPTY(L)  or n = null
 2     then error  element n does not exist
 3  r ← next[n]
 4  next[n] ← ALLOCATE-NODE (x, n, r)
 5  prev[r] ← next[n]
 6  size[L] ← size[L] + 1

DELETE-AFTER (L, n)
    ▷ عنصر بعدی n را در لیست دو سویه‌ی L حذف می‌کند
 1  if ISEMPTY(L)  or n = null or next[n] = null
 2     then error  element does not exist
 3  r ← next[n]
 4  if next[r] ≠ null
 5     then prev[next[r]] ← n
 6  next[n] ← next[r]
 7  FREE-NODE(r)
 8  size[L] ← size[L] − 1
```

۴-۲-۳ عملیات دیگر بر روی لیست‌ها

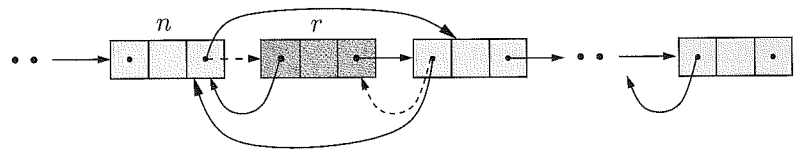
در این بخش چند الگوریتم دیگر بر روی لیست‌ها ارائه می‌کنیم. این اعمال را می‌توان با اعمال پایه‌ای که گفتیم پیاده‌سازی کرد، اما هدف پیاده‌سازی مستقیم آن‌هاست.



(الف) لیست خطی دوسویه‌ی



(ب) INSERT-AFTER(L, x, n)



(پ) DELETE-AFTER(L, n)

شکل ۴-۳ انجام اعمال درج و حذف بر روی لیست‌های خطی دوسویه.

حذف عناصر تکراری در یک لیست

هدف حذف عناصر تکراری از یک لیست است، که ترتیب عناصر باقی‌مانده تغییر نکند. مثلاً می‌خواهیم لیست $\langle ۲, ۳, ۱, ۱, ۲, ۱, ۱ \rangle$ به $\langle ۳, ۲, ۱ \rangle$ تبدیل شود. PURGELIST این‌کار را انجام می‌دهد.

```
PURGELIST (L)
    ▷ همه‌ی عناصر یکسان را جز یکی حذف می‌کند ◁
1   p ← FIRST (L)
2   while p <> null
3       do q ← p
4          while next[q] <> null
5              do if element[p] = element[next[q]]
6                     then DELETE-AFTER (L, q)
7                     else q ← next[q]
8          p ← next[p]
```

اگر تعداد عناصر لیست n باشد، این الگوریتمی از مرتبه‌ی $\mathcal{O}(n^2)$ است. این کار را می‌توان با مرتب‌سازی لیست و سپس با $\mathcal{O}(n)$ کار اضافی برای حذف عناصر تکراری در لیست مرتب هم حل کرد. البته در این‌صورت، لیست نهایی این الگوریتم (که عناصرش مرتب هستند) با لیستی که الگوریتم فوق تولید می‌کند متفاوت خواهد بود. در ادامه‌ی این بخش، نشان می‌دهیم که چگونه می‌توان مرتب‌سازی ادغامی را بر روی لیست‌ها پیاده‌سازی کرد.

وارون کردن یک لیست

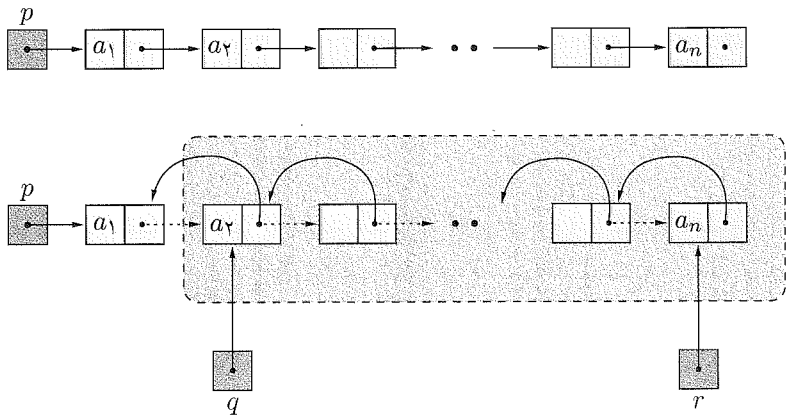
می‌خواهیم یک لیست را تنها با تغییر اشاره‌گرهای آن وارون کنیم، به‌طوری‌که عنصر iام لیست n عضوی پس از وارون‌سازی عنصر $n-i+1$ام لیست جدید شود. این‌کار را هم به‌صورت بازگشتی و هم غیربازگشتی انجام می‌دهیم. رویه‌ی بازگشتی RECURSIVE-REVERSE(L,p) زیرلیست شامل عناصر p به بعد را در لیست L، وارون‌شده برمی‌گرداند. برای وارون کل لیست p باید برابر $first[L]$ باشد.

فرض کنید که تعداد عناصر از p به بعد در لیست L برابر n باشد. الگوریتم بازگشتی ابتدا عنصر p را کنار می‌گذارد و بقیه‌ی لیست با $n-1$ عنصر (که با عنصر q آغاز می‌شود) را به‌صورت بازگشتی وارون می‌کند و از همان عناصر یک لیست r ایجاد می‌کند. در پایان، عنصر p را به انتهای لیست r وصل می‌کند تا کار کامل شود.

```
RECURSIVE-REVERSE (L, p)
    ▷ لیست L را از عنصر p به بعد وارون می‌کند و حاصل را برمی‌گرداند.
1   if p = null or next[p] = null
2      then return p
3   q ← next[p]
4   r ← RECURSIVE-REVERSE (L, q)
5   next[q] ← p
6   next[p] ← null
7   return r
```

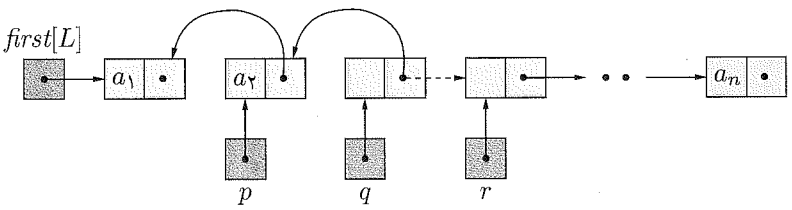
پس از وارون لیست $n-1$ عضوی، اشاره‌گر r برابر عنصر انتهایی لیست L خواهد بود و با فرض استقرا، همه‌ی اشاره‌گرهای لیستِ کوچک‌تر به عقب اشاره خواهند کرد. پس

برای این‌که عنصر p را به انتهای لیست r بچسبانیم، لازم است در ابتدا اشاره‌گر $next[q]$ را برابر p و سپس $next[p]$ را برابر **null** قرار دهیم. بدیهی است که این رویه از مرتبه‌ی $\mathcal{O}(n)$ است. شکل ۴-۴ نحوه‌ی کار این الگوریتم را نشان می‌دهد.



شکل ۴-۴ وارون کردن یک لیست به‌صورت بازگشتی.

الگوریتم غیربازگشتی برای این‌کار به سه اشاره‌گر p q و r نیاز دارد که p و q به‌ترتیب به سه عنصر متوالی در لیست اصلی (وارون نشده) اشاره می‌کنند. در زمان اجرا، q و p به‌ترتیب به اولین و دومین عنصر لیستی که تا این مرحله وارون شده اشاره می‌کنند و r به عنصر بعدی در لیست L، که هنوز وارون نشده است (شکل ۴-۵). در ابتدا، p تهی است و q و r به‌ترتیب، به عنصر اول و دوم لیست اصلی اشاره می‌کنند. در هر مرحله، کافی است که مؤلفه‌ی $next$ عنصر r را به عنصر q تغییر دهیم و هر سه اشاره‌گر را یک مرحله به‌جلو ببریم. روشن است که اگر r تهی شود، کار وارون‌سازی به‌پایان رسیده است.



شکل ۴-۵ وارون‌کردن یک لیست به‌صورت غیر بازگشتی.

رویه‌ی NR-Reverse همین الگوریتم را بیان می‌کند.

```
NR-Reverse (L)
 1  if Size(L) ≤ 1
 2     then return first[L]
 3  p ← null
 4  q ← First(L)
 5  r ← next[q]
 6  while r ≠ null
 7     do next[q] ← p
 8        p ← q
 9        q ← r
10        r ← next[r]
11  next[q] ← p
12  return q
```

روشن است که این‌کار در زمان خطی انجام می‌شود.

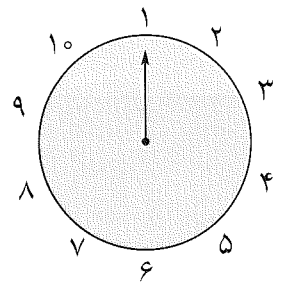
مسئله‌ی ژوزفوس

این یک مسئله‌ی قدیمی و گونه‌ای از مسئله‌ی باستانی منتسب به فلاویوس ژوزفوس[۲۹] تاریخ‌دان قرن اول میلادی است. او در زمان جنگ رومیان با یهودیان، یکی از ۴۱ یهودی‌ای بود که به‌وسیله‌ی رومیان در یک غار محاصره شده بودند. به‌جای تسلیم، این گروه از جنگ‌جویان تصمیم گرفتند که همگی به‌این صورت خودکشی کنند: آن‌ها قرار گذاشتند تا با شروع از نفر اول، و به‌صورت حلقوی، هر بار نفر دوم زنده‌ها خود را بکُشد و نوبت به نفر زنده‌ی بعدی برسد، تا این‌که هیچ‌کس باقی نماند. ولی ژوزفوس زرنگ‌تر از آن‌ها بود و مکان نشستن آخرین فردی را که قاعدتاً باید خود را به‌تنهایی می‌کشت محاسبه کرد و از ابتدا در آن مکان نشست و جان سالم به‌در برد.[۳۰]

مسئله در حالت کلی به این صورت است: اگر n نفر با شماره‌های ۱ تا n دور دایره‌ای قرار بگیرند و با شروع از شماره‌ی ۱ و در جهت ساعت‌گرد هر بار دومین نفر زنده (یا

[۲۹] Flavius Josephus

[۳۰] این مسئله در کتاب [۹] بررسی و تحلیل شده است.



شکل ۴-۶ مسئله‌ی ژوزفوس با ۱۰ نفر.

kامین نفر در حالت کلی) خودش را بکشد، آخرین نفر چه شماره‌ای دارد؟ مثلاً مطابق شکل ۴-۶، برای $n = ۱۰$ به‌ترتیب افراد

۲، ۴، ۶، ۸، ۱۰، ۳، ۷، ۱، ۹

خودکشی می‌کنند و ۵ زنده می‌ماند.

جواب این مسئله یا $J(n)$، از رابطه‌ی بازگشتی زیر قابل محاسبه است:

$$\begin{aligned} J(۱) &= ۱ \\ J(۲n) &= ۲J(n) - ۱, \quad \text{for } n \geq ۱, \\ J(۲n+۱) &= ۲J(n) + ۱ \quad \text{for } n \geq ۱. \end{aligned}$$

اگر n را به‌صورت یک عدد دودویی بنویسیم و آن‌را به‌چپ یک بیت شیفت دورانی دهیم $J(n)$ به‌دست می‌آید. مثلاً برای

$$n = ۱۰۰ = (۱۱۰۰۱۰)_۲,$$

جواب

$$J(n) = (۱۰۰۱۰۰۱)_۲ = ۷۳$$

است.[۳۱]

ما در این‌جا مسئله را با استفاده از یک لیست پیوندی حلقه‌ای پیاده‌سازی می‌کنیم و جواب را با شبیه‌سازی خودکشی‌ها به‌دست می‌آوریم. روشن است که این راه‌حلِ کُند، برای حالات دیگر این مسئله هم قابل استفاده است.

[۳۱] در سال ۱۳۸۱ آقای آرمین شمس براق از دانش‌جویان دانشگاه فردوسی مشهد راه‌حل ساده‌تری برای اثبات جواب این مسئله در حالتی خاص پیدا کرد که مورد تأیید و تشویق آقای کنوث قرار گرفت.

```
JOESEPHOUS (n)
    ▷ در ابتدا یک لیست پیوندی حلقوی با n گره ایجاد می‌کند ▷
 1  CREATE(L)
 2  first[L] ← q ← ALLOCATE-NODE(1, null)
 3  p ← q
 4  for i ← 2 to n
 5     do next[p] ← ALLOCATE-NODE(i, next[p])
 6        p ← next[p]
 7  next[p] ← q;    size[L] ← n
    ▷ و حالا راه‌حل کُند مسئله‌ی ژوزفوس ▷
 8  p ← FIRST(L)
 9  while next[p] ≠ p
10     do DELETE-AFTER(L, p)
11        p ← next[p]
12  PRINT element[p]
```

۴-۲-۴ پیاده‌سازی لیست‌ها با اشاره‌گرهای اندیسی

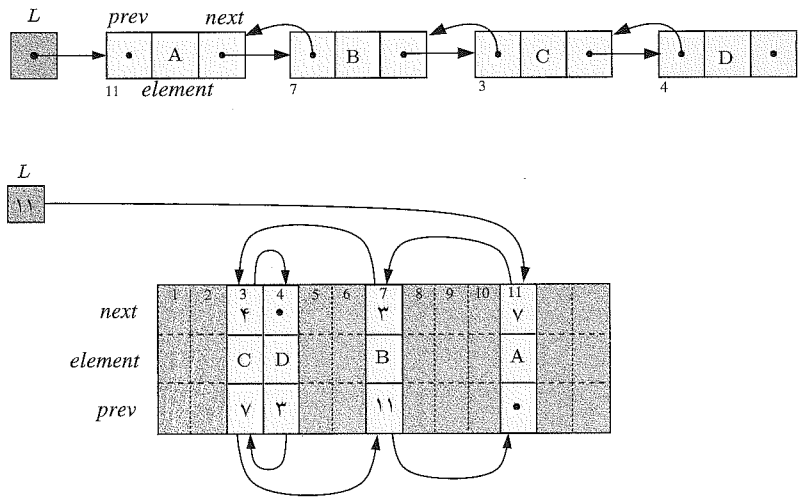
چگونه می‌توان لیست‌ها را بدون اشاره‌گر پیاده‌سازی کرد؟ در زبان‌های شیءگرا، محتوای اشاره‌گر به یک شیء آدرس اولین کلمه‌ی آن شیء در حافظه است. اما در مواردی می‌توانیم به‌جای آدرس یک شیء، اندیس اولین درایه‌ی آن شیء در یک آرایه‌ی بزرگ ــ که حافظه‌ی کامپیوتر را شبیه‌سازی می‌کند ــ را به‌کار ببریم. در این صورت ما از «اشاره‌گر اندیسی[۳۲]» استفاده کرده‌ایم.

نحوه‌ی کار با اشاره‌گرهای اندیسی، به فهم بهتر کار اشاره‌گرهای واقعی کمک می‌کند. به‌علاوه، در مواردی ممکن است این نوع پیاده‌سازی سریع‌تر هم باشد. در بخش ۷-۱ داده‌ساختارهایی را خواهیم دید که پیاده‌سازی آن‌ها با اشاره‌گر اندیسی کاراتر است.

در این بخش، به‌عنوان نمونه، نحوه‌ی پیاده‌سازی لیست‌ها با اشاره‌گر اندیسی را نشان می‌دهیم. از همین روش می‌توان داده‌ساختارهای پیچیده‌تر را هم با این اشاره‌گر پیاده‌سازی کرد. نکته‌ی مهم آن است که در این‌حالات باید مدیریت فضای آزاد را خود انجام دهیم.

---

[۳۲] cursor



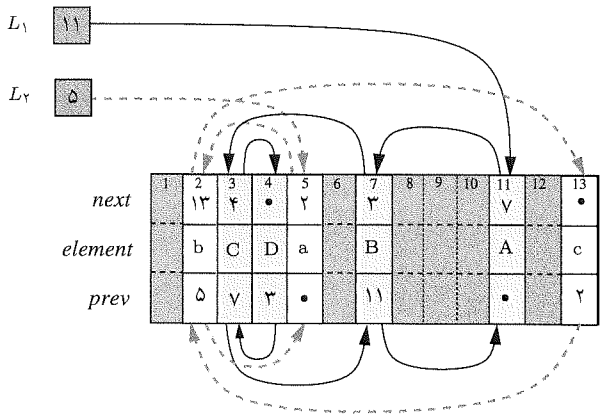
شکل ۴–۷ پیاده‌سازی یک لیست دوسویه با اشاره‌گر اندیسی.

استفاده از آرایه در پیاده‌سازی لیست‌ها

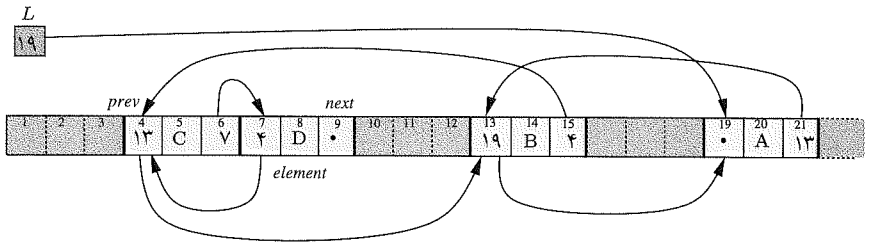
فرض کنید می‌خواهیم لیست دوسویه‌ی L را با آرایه پیاده‌سازی کنیم. مؤلفه‌های عناصر لیست *next*، *prev* و *element* هستند. در این‌صورت، مطابق شکل ۴–۷ سه آرایه‌ی *next*، *prev* و *element* با یک اندازه‌ی حداکثری تعریف می‌کنیم؛ اگر i اندیس یک عنصر از لیست باشد، $next[i]$، $prev[i]$ و $element[i]$ مقادیر مؤلفه‌های آن عنصر هستند.

می‌توان در یک مجموعه از آرایه‌ها بیش از یک لیست را پیاده‌سازی کرد، چنان‌چه شکل ۴–۸ دو لیست L_1 با عناصر A، B، C و D و L_2 با عناصر a، b و c را نشان می‌دهد. شماره‌ی درایه‌هایی که عناصر در آن‌ها قرار گرفته‌اند اصلاً مهم نیست بلکه اشاره‌گرها هستند که این لیست‌ها را می‌سازند.

همین کار را می‌توان در یک آرایه‌ی بزرگ انجام داد. این مطابق مدل حافظه‌ی کامپیوتر است که به‌صورت یک آرایه‌ی بزرگ از کلمات است. در بسیاری از زبان‌های برنامه‌سازی یک شیء تعدادی از کلمات پشتِ‌سرِهم را اشغال می‌کند. اشاره‌گر به یک شیء در واقع آدرس اولین کلمه‌ی آن شیء در حافظه است. آدرس شروع مؤلفه‌های آن شیء را می‌توان با افزودن مقدارهای ثابت به آدرس آن شیء به‌دست آورد. اگر همین روش را برای ذخیره‌ی لیست مورد نظر در یک آرایه‌ی A به‌کار ببریم، شکل ۴–۹ به‌دست می‌آید که در آن یک عنصر از لیست مثلاً درایه‌های $A[j \cdots k]$ را اشغال می‌کند. اشاره‌گر به این عنصر،



شکل ۴-۸ دو لیست L_1 و L_2 در یک آرایه.



شکل ۴-۹ پیاده‌سازی لیست دوسویه‌ی شکل ۴-۷ با یک آرایه.

اندیس j را دارد و برای شروع مؤلفه‌های آن باید مقادیر ۰ تا $j-k$ را به آن افزود.

در صورتی که مؤلفه‌های یک عنصر از داده‌گونه‌های مختلف باشند و یا اندازه‌های متفاوت داشته باشند، پیاده‌سازی به‌این روش و نیز مدیریت فضای آزاد پیچیده‌تر از استفاده از چند آرایه است. در این‌جا مدیریت فضای آزاد را در حالت چند آرایه‌ای ذکر می‌کنیم.

مدیریت فضای آزاد در پیاده‌سازی با اشاره‌گر اندیسی

مدیریت فضای آزاد در پیاده‌سازی با اشاره‌گرهای اندیسی به‌وسیله‌ی خود کاربر صورت می‌گیرد. در این‌جا همه‌ی آرایه‌ها در واقع کل حافظه‌ی در اختیار همه‌ی لیست‌هاست که عناصر یک‌سان دارند. چنان‌چه در شکل ۴-۸ نشان داده شد، می‌توان به این روش تعداد

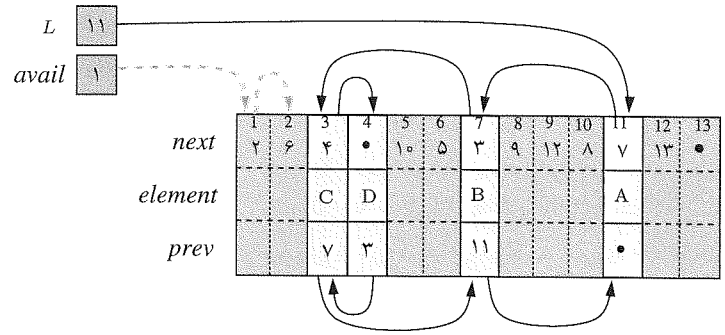
زیادی لیست را در یک فضای مشترک پیاده‌سازی کرد. در این روش، عناصر «آزاد» که توسط هیچ لیستی استفاده نمی‌شوند، با استفاده از یکی از مؤلفه‌های اندیسی، مثلاً *next* به‌هم وصل می‌شوند و تشکیل یک لیست جدید به‌نام «لیست فضای آزاد» می‌دهند. ما این لیست را *avail* می‌نامیم که شکل ۴-۱۰(الف) آن‌را نشان می‌دهد. در ابتدا که همه‌ی فضا آزاد است، *avail* برابر ۱ است و همه‌ی عناصر آرایه به‌ترتیب در آن لیست قرار می‌گیرند.

در اختیار قرار دادن یک عنصر از فضای آزاد (ALLOCATE-OBJECT)، در واقع حذف عنصر اول لیست *avail* و در اختیار قرار دادن آن عنصر است (البته هر عنصر دیگر *avail* نیز مناسب است، ولی حذف عنصر اول از همه ساده‌تر است). همچنین، آزادسازی یک عنصر x در واقع درج آن عنصر در ابتدای لیست *avail* است. این رویه‌ها در ادامه آمده‌اند.

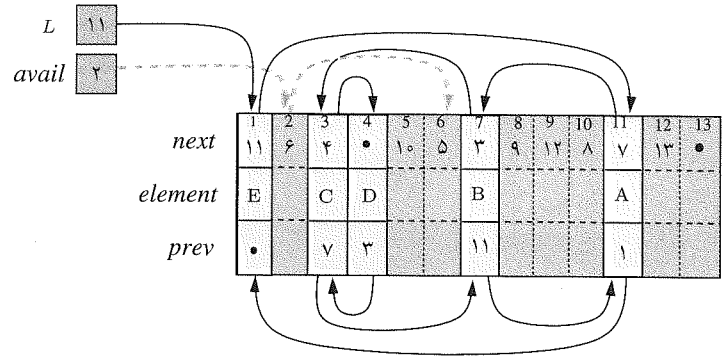
```
INITIALIZE ()
1  null ← 0
2  avail ← 1
3  for i ← 1 to M − 1
4     do next[i] ← i + 1
5  next[M] ← null

ALLOCATE-OBJECT ()
1  if avail = null
2     then error out of space
3  x ← avail
4  avail ← next[avail]
5  return x

FREE-OBJECT (x)
1  next[x] ← avail
2  avail ← x
```

مدیریت فضای آزاد در پیاده‌سازی با اشاره‌گر اندیسی

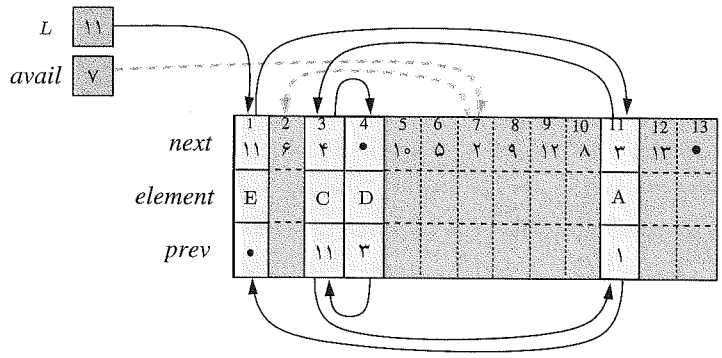
البته این روش ساده‌ترین روش مدیریت فضای آزاد است. در عمل، مدل تک آرایه‌ای استفاده می‌شود که در آن لیست‌های گوناگون در این فضا قرار می‌گیرند. در این‌صورت کار مدیریت فضای آزاد به‌وسیله‌ی کامپایلرها انجام می‌شود. یکی از مسائل مهم مدیریت چنین



(الف) لیست اصلی



(ب) پس از درج E به‌عنوان عنصر اول لیست (الف)



(پ) پس از حذف عنصر B در لیست (ب)

شکل ۴-۱۰ مدیریت فضای آزاد در پیاده‌سازی با اشاره‌گر اندیسی.

فضایی که چندین لیست کلی در آن قرار دارد، مسئله‌ی «زباله‌روبی[۳۳]» است. این مسئله وقتی مهم می‌شود که کاربر تغییراتی در اشاره‌گرها می‌دهد و در اثر آن تعداد زیادی از درایه‌ها غیر قابل استفاده می‌شود. این مشکل وقتی رخ می‌دهد که این عناصر با فراخوانی FREE-OBJECT به فضای آزاد بازگردانده نشده باشند.

هنگامی که میزان حافظه‌ی آزاد کم شود، رویه‌ی زباله‌روبی فراخوانده می‌شود که کارش بررسی فضای حافظه و تشکیل لیست جدیدی از فضای آزاد است که شامل همه‌ی حافظه‌ی غیر قابل دست‌رسی به‌وسیله‌ی اشاره‌گرهای موجود است. حافظه‌ای غیرقابل دست‌رسی است که با شروع از متغیرها و اشاره‌گرهای قابل استفاده یا از طریق اشاره‌گرهای اندیسی موجود نتوان به آن رسید.

نکته‌ی مهم آن است که عمل زباله‌روبی معمولاً وقتی انجام می‌شود که کامپایلر برای در اختیار گذاردن فضا برای یک شیء مورد تقاضای کاربر، دچار کم‌بود شود. بنابراین رویه‌ی زباله‌روبی هم باید طوری طراحی شود که نیاز به حافظه‌ی زیادی نداشته باشد. از این‌رو، معمولاً این رویه‌ها به‌صورت غیربازگشتی و با استفاده از تعداد کمی متغیر کمکی طراحی می‌شوند. در پروژه‌ای در انتهای این فصل، پیاده‌سازی این رویه را تمرین خواهید کرد.

تمرین‌های زیربخش ۴-۲-۴

۴-۲-۱.۴ معمولاً مطلوب است که تمام عناصر یک لیست پیوندی دوسویه را در حافظه‌ای فشرده ذخیره کنیم. به‌عنوان مثال این کار با استفاده از m مکانِ اول اندیس در نمایش چندآرایه‌ای صورت می‌گیرد. (این همان وضعیتی است که در محیط محاسباتی حافظه‌ی مجازی صفحه‌بندی شده وجود دارد.) توضیح دهید که چطور اعمال ALLOCATE-OBJECT و FREE-OBJECT را می‌توان پیاده‌سازی کرد به‌طوری که نمایش آن فشرده باشد. فرض کنید که خارج از لیست پیوندی، هیچ اشاره‌گری به عناصر لیست وجود ندارد. (راهنمایی: از پیاده‌سازی آرایه‌ای پشته استفاده کنید.)

۴-۲-۲.۴ فرض کنید L یک لیست پیوندی دوسویه به‌طول m است که در آرایه‌های *key*، *next* و *prev*، هر کدام با طولی برابر با n، ذخیره شده است. علاوه بر آن، فرض کنید این آرایه‌ها با رویه‌های ALLOCATE-OBJECT و FREE-OBJECT مدیریت می‌شوند و حاوی یک لیست پیوندی *avail* از عناصر آزاد است. همچنین فرض کنید که از این n عنصر، دقیقاً m تا در لیست L و $n - m$ تا در *avail* قرار دارند. یک رویه به‌نام COMPACTIFY-LIST(L, *avail*) بنویسید تا با داشتن لیست L و لیست *avail*، عناصر درون L را به‌گونه‌ای جابه‌جا کند که خانه‌های ۱، ۲ تا m از آرایه را اشغال کنند و همچنین لیست عناصر آزاد را طوری تغییر دهد که درست باقی بماند و درایه‌های $m+1$، $m+2$ تا n از آرایه را شامل شود. رویه‌ی شما باید دارای زمان اجرای $\Theta(m)$ باشد، و تنها از مقدار ثابتی حافظه‌ی اضافی استفاده کند. در مورد درستی رویه خود بحث کنید.

[۳۳] garbage collection

۴-۲-۵ پشته‌ها

پشته لیستی است که اعمال درج و حذف عناصر فقط در یک سوی آن ــ به‌نام بالای پشته ــ انجام می‌شود.

اعمال بر روی پشته‌ی S

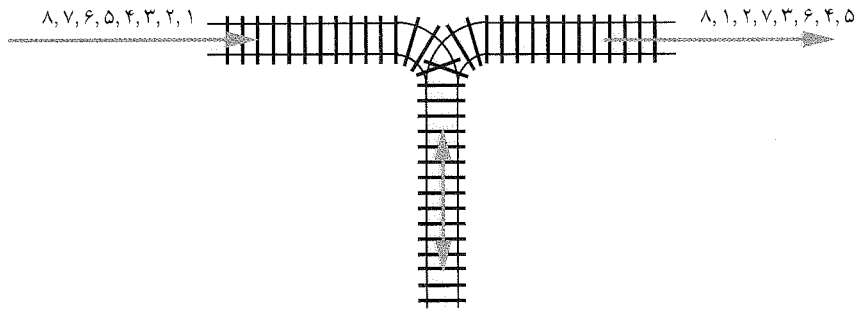
- CREATE-STACK(S): یک پشته‌ی تهی S را ایجاد می‌کند،
- PUSH(S, x): یک عنصر x را در بالای پشته‌ی S درج می‌کند،
- POP(S): عنصر بالای پشته را حذف و آن‌را برمی‌گرداند،
- SIZE(S): تعداد عناصر موجود در پشته را برمی‌گرداند،
- ISEMPTY(S): مشخص می‌کند که آیا پشته خالی است، و
- TOP(S): عنصر بالای پشته را برمی‌گرداند.

قبل از بحث در مورد جزییات این اعمال، به چند کاربرد پشته می‌پردازیم.

کاربرد پشته‌ها: جابه‌جایی قطارها

در ایستگاه‌های قطار برای جابه‌جا کردن قطارها از یک ریل اضافی مانند شکل ۴-۱۱ استفاده می‌کنند که مثل پشته عمل می‌کند. یعنی اولین قطار که وارد این ریل می‌شود، آخرین قطاری است که از آن خارج می‌گردد. عمل درج در پشته و حذف از آن مطابق تعریف انجام می‌شود. توجه کنید که پس از عمل حذف، بالاترین قطار در پشته فقط در ریل سمت راست قرار می‌گیرد و امکان بازگشت مجدد آن به پشته یا برگشت به ریل ورودی نیست.

حال فرض کنید که در ریل ورودی دنباله‌ای از قطارها با شماره‌های ۱ تا n پشت سرهم قرار دارند (۱ در ابتدای ردیف است). با ترکیبی مناسب از n بار درج و n بار حذف، جای‌گشتی از شماره‌های قطارها در خروجی به‌دست می‌آید که به آن «جای‌گشت قابل تولید» می‌گوییم. مثلاً جای‌گشت $\langle ۵, ۴, ۶, ۳, ۷, ۲, ۱, ۸ \rangle$ قابل تولید است چون با اعمال زیر



شکل ۴-۱۱ پشته‌ای از قطارها.

(از چپ‌به‌راست) ایجاد می‌شود (I برای درج و D برای حذف):

I, I, I, I, I, D, I, D, D, I, D, D, D, I, D

ولی

$$\langle ۴, ۳, ۷, ۸, ۶, ۲, ۵, ۱ \rangle$$

قابل تولید نیست چون برای این‌که زیردنباله‌ی ۷, ۸, ۶ تولید شود باید ابتدا قطارهای ۱ تا ۴ به پشته وارد و سپس ۴ و ۳ به‌ترتیب خارج شوند. سپس به‌ناچار باید قطارهای ۵ تا ۸ به‌همین ترتیب از ورودی به پشته درج شوند که سه قطار بالای پشته از بالا ۸، ۷ و ۶ خواهند بود که در آن‌صورت امکان تولید زیردنباله‌ی بیان‌شده نیست.

مسئله‌ی ارزیابی بازار بورس

در بازار بورس، آخرین قیمت سهام هر شرکت در انتهای هر روز تهیه می‌شود و بر روی این داده‌ها تحلیل‌های مختلف انجام می‌گیرد. فرض کنید می‌خواهیم در هر روز «دوره‌ی سهام» یک شرکت خاص را پیدا کنیم، به این صورت که اگر قیمت سهام آن شرکت در روز i برابر p_i باشد، دوره‌ی سهام در روز i برابر است با تعداد روزهای بلافاصله قبل از i (و شامل خود i) که قیمت سهام کم‌تر یا مساوی p_i باشد. اگر قیمت سهام یک شرکت در n روز متوالی در یک آرایه‌ی n تایی به‌نام P ذخیره شده باشد، می‌خواهیم در خروجی آرایه‌ی n تایی S را محاسبه کنیم که $S[i]$ دوره‌ی سهام آن شرکت در روز i باشد.

الگوریتم کُند

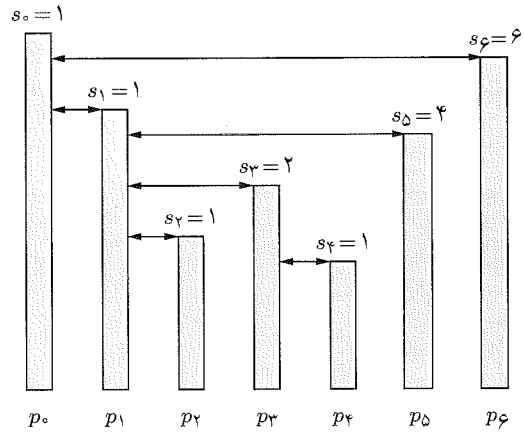
با یک الگوریتم کند از مرتبه‌ی $\mathcal{O}(n^2)$ می‌توان این مسئله را به‌صورت زیر حل کرد:

```
COMPUTESPANS (P)
    ▷ P آرایه‌ی n-عضوی
    ▷ S آرایه‌ی n-عضوی
 1  for i ← 0 to n − 1
 2    do k ← 0
 3       done ← false
 4       repeat if P[i − k] ≤ P[i]
 5              then k ← k + 1
 6              else done ← true
 7       until k = i or done
 8       S[i] ← k
 9  return S
```

الگوریتم خطی

نکته‌ی مهم در حل سریع این مسئله آن است که برای به‌دست آوردن دوره‌ی سهام در روز i، نیازی به بررسی قیمت سهام در همه‌ی روزهای قبل از آن نیست، بلکه کافی است از بین این قیمت‌ها آن‌هایی را نگه‌داریم که به‌ازای $t_1 < t_2 < \cdots < t_k = i - 1$ داشته باشیم $p_{t_1} > p_{t_2} > \cdots > p_{t_k}$، به‌طوری‌که همه‌ی مقادیر سهام بین روزهای t_i و t_{i+1} بین دو مقدار p_{t_i} و $p_{t_{i+1}}$ باشد (به شکل ۴-۱۲ مراجعه کنید). اگر این اطلاعات را داشته باشیم، می‌توان به‌سادگی S_i را به‌دست آورد: p_i را با p_{t_k} مقایسه می‌کنیم، اگر از آن کم‌تر بود که بازه‌ی iام به‌دست می‌آید، وگرنه باید با $p_{t_{k-1}}$ مقایسه کنیم، چرا که بقیه‌ی قیمت‌ها در این بازه از p_{t_k} کم‌ترند و نیازی به بررسی آن‌ها نیست. این کار را تکرار می‌کنیم تا به بزرگ‌ترین rای برسیم که $p_{t_r} > P_i > p_{t_{r+1}}$. در آن‌صورت $i - t_r$ دوره‌ی iام است. البته اگر $p_{t_1} \leq p_i$، در آن‌صورت i جواب است.

در هر مرحله اعداد را در یک پشته‌ی D ذخیره می‌کنیم به‌طوری‌که p_{t_k} در بالای پشته قرار گیرد. رویه‌ی COMPUTESPANS2 این کار را انجام می‌دهد.



شکل ۴-۱۲ قیمت روزانه‌ی سهام یک شرکت در بازار بورس (p_i) و دوره‌ی سهام آن در هر روز (s_i).

```
COMPUTESPANS2 (P)
    ▷ از پشته‌ی S استفاده می‌کنیم ▷
 1  for i ← 0 to n − 1
 2    do done ← false
 3       while ( not ISEMPTY(D) or not done)
 4          do if P[i] ≥ P[top[D]]
 5                then POP(D)
 6                else done ← true
 7       if ISEMPTY(D)
 8          then h ← −1
 9          else h ← TOP(S)
10       S[i] ← i − h
11       PUSH(D, i)
```

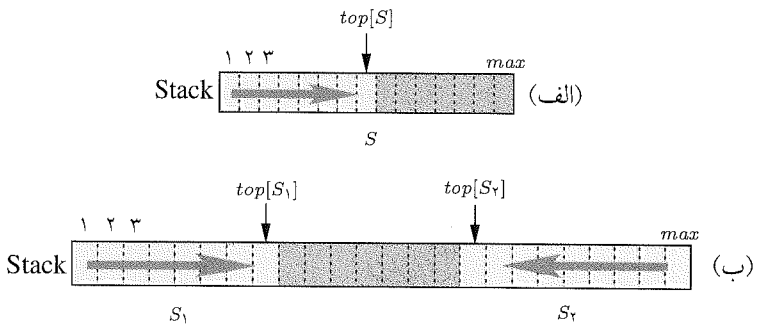
زمان اجرای این الگوریتم $O(n)$ است چرا که هر p_i دقیقاً یک‌بار وارد پشته و حداکثر یک بار از آن خارج می‌شود.

پیاده‌سازی پشته با آرایه

در این روش، یک آرایه‌ی سراسری به‌نام Stack به‌اندازه‌ی max از عناصری با داده‌گونه‌ی $element$ ایجاد می‌شود. پشته‌ی S حاوی یک مؤلفه‌ی $top[S]$ است که اندیس عنصر بالای پشته را در آرایه‌ی Stack نشان می‌دهد؛ پشته از اندیس ۱ تا $top[S]$ در آن آرایه قرار دارد. فرض می‌شود که این اندیس برای پشته‌ی خالی برابر ۰ است.

```
public class ArrayStack implements Stack {
    public static final int CAPACITY=1000;
    private int capacity;
    private object S[];
    private int top = -1;
    public ArrayStack(){
       this(CAPACITY);
    }
    public ArrayStack(int cap){
       capacity = cap;
       S = new object[capacity];
    }
}
```

شکل ۴-۱۳ نمایشی از این پیاده‌سازی است که در آن یک پشته و دو پشته با آرایه پیاده‌سازی شده‌اند. بدیهی است که محدودیت تعداد عناصر پشته نقطه‌ی ضعف این پیاده‌سازی است.



شکل ۴-۱۳ پیاده‌سازی پشته با آرایه: (الف) یک پشته و (ب) دو پشته.

اعمال مختلف با این پیاده‌سازی در زیر آمده است:

```
CREATE (S)
1  top[S] ← 0

SIZE (S)
1  return top[S]

ISEMPTY (S)
1  return SIZE(S) = 0

TOP (S)
1  if ISEMPTY(S)
2     then error stack is empty
3  return Stack[top[S]]

PUSH (S, x)
1  if SIZE(S) = max
2     then error stack is full
3  top[S] ← top[S] + 1
4  Stack[top[S]] ← x

POP (S)
1  if ISEMPTY(S)
2     then error stack is empty
3  e ← Stack[top[S]]
4  top[S] ← top[S] − 1
5  return e
```

اعمال بر روی پشته‌ی S

بدیهی است که هر یک از اعمال در $\mathcal{O}(1)$ انجام می‌شوند.

پیاده‌سازی پشته با لیست پیوندی

در این روش پشته‌ی S مانند یک لیست پیوندی پیاده‌سازی می‌شود، با این محدودیت که درج و حذف هر عنصر فقط در ابتدای لیست انجام شوند. در این‌جا دیگر محدودیتی برای تعداد عناصر پشته نیست.

```
SIZE (S)
 1  return  size[S]

ISEMPTY (S)
 1  return  top[S] =null

TOP (S)
 1  if ISEMPTY(S)
 2     then error stack is empty
 3  return  element[top[S]]

PUSH (S, x)
 1  n ← ALLOCATE-NODE()
 2  element[n] ← x
 3  next[n] ← top[S]
 4  top[S] ← n
 5  size[S] ← size[S] + 1

POP (S)
 1  if ISEMPTY(S)
 2     then error STACK IS EMPTY
 3  n ← top[S]
 4  temp ← element[top[S]]
 5  top[S] ← next[top[S]]
 6  size[S] ← size[S] − 1
 7  FREE-OBJECT(n)
 8  return  temp
```

پیاده‌سازی پشته با لیست پیوندی یک‌سویه

تمرین‌های زیربخش ۴-۲-۵

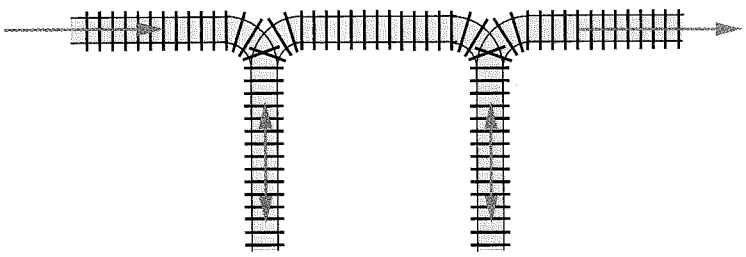
۴-۲-۵.۱ در مسئله‌ی پشته‌ای از قطارها، شرط لازم و کافی برای این‌که یک دنباله قابل تولید باشد چیست؟ آن‌را اثبات کنید.

۴-۲-۵.۲ الگوریتمی از $O(n)$ ارائه دهید تا قابل تولید بودن یک دنباله را تشخیص دهد.

۴-۲-۵.۳ فرض کنید یک ریل مستقیم نیز بین قطارهای ورودی و خروجی وجود دارد؛ یعنی قطار ورودی می‌تواند یا به داخل پشته رود و یا مستقیماً به ریل خروجی منتقل شود، و یا برعکس از

خروجی به ورودی برگردد. در این صورت الگوریتم تشخیص یک دنباله‌ی قابل تولید را ارائه دهید.

۴-۲-۴.۵ مسئله‌ی جابه‌جایی قطارها را مطابق شکل۴-۱۴ برای سیستم دو پشته‌ای حل کنید.



شکل ۴-۱۴ دو پشته از قطارها.

۴-۲-۶ صف

صف لیستی است که درج یک عنصر فقط در انتهای آن ممکن است و حذف از آن، عنصرِ ابتدایی لیست را حذف و برمی‌گرداند. اعمال پایه‌ای بر روی یک صف به شرح زیرند:

اعمال بر روی یک صف Q

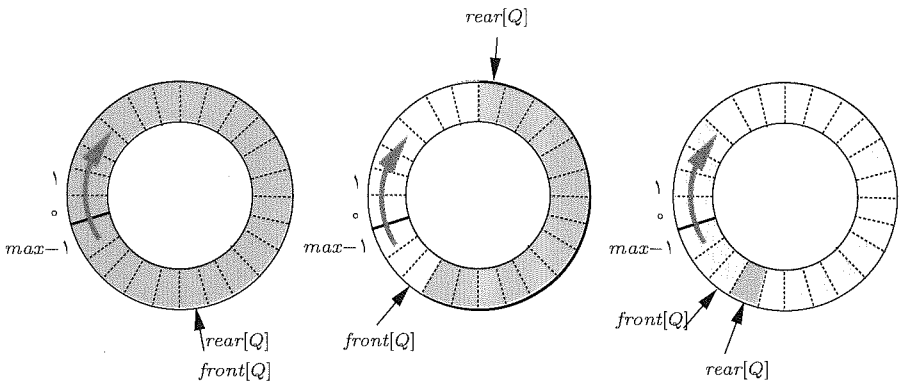
- CREATE-QUEUE(Q): یک صف تهی Q را ایجاد می‌کند،
- ENQUEUE(Q, x): یک عنصر x را در انتهای Q درج می‌کند،
- DEQUEUE(Q): یک عنصر را از ابتدای صف حذف و آن‌را برمی‌گرداند،
- SIZE(Q): تعداد عناصر موجود در صف را در اختیار قرار می‌دهد،
- ISEMPTY(Q): مشخص می‌کند که آیا Q خالی است، و
- FRONT-ELEMENT(Q): عنصر ابتدای صف را برمی‌گرداند.

پیاده‌سازی با آرایه‌ی حلقوی

در این پیاده‌سازی یک صف Q شامل یک آرایه، مثلاً به نام Queue، با اندازه‌ی max و اندیس‌های ۰ تا $max-۱$ است که عناصر آن به‌صورت چرخه‌ای و در جهت ساعت‌گرد در

آن ذخیره می‌شوند، یعنی Queue$[(i+1) \bmod max]$ عنصر بعدیِ Queue$[i]$ است.

مؤلفه‌های $front[Q]$ و $rear[Q]$ به‌ترتیب اندیس اولین عنصر و عنصر بعد از آخرین عنصرِ صف Q را ذخیره می‌کنند. $size[Q]$ هم تعداد عناصر Q را نشان می‌دهد. برای تمیز دادن سریع دو حالت «کاملاً پر» و «کاملاً خالی» صف از هم، حداکثر $max-1$ عنصر را در آن جای می‌دهیم. شکل ۴-۱۵ یک صف و حالت‌های مختلف آن‌را نشان می‌دهد.



شکل ۴-۱۵ یک آرایه‌ی حلقوی برای پیاده‌سازی صف.

حالت‌های مختلف صف

در ابتدا صف خالی است، اندازه‌ی آن برابر صفر و مؤلفه‌های آن

$$front[Q] = rear[Q] = 0$$

است. تعداد عناصر یک صف با این پیاده‌سازی همیشه برابر

$$(max - front[Q] + rear[Q]) \bmod max$$

است. اگر صف کاملاً خالی باشد داریم $front[Q] = rear[Q]$ و اگر کاملاً پر باشد، رابطه‌ی زیر برقرار است:

$$(max - front[Q] + rear[Q]) \bmod max = max - 1 \equiv (rear[Q] + 1) \bmod max = front[Q]$$

اعمال مختلف بر روی صف به صورت زیر و هزینه‌ی هرکدام $\mathcal{O}(1)$ است:

```
CREATE-QUEUE (Q)
1  ALLOCATE-NODE(Q)
2  rear[Q] = front[Q] ← size[Q] ← 0

SIZE (Q)
1  return (max − front[Q] + rear[Q]) mod max

ISEMPTY (Q)
1  return (front[Q] = rear[Q])

FRONT-ELEMENT (Q)
1  if ISEMPTY(Q)
2     then error queue is empty
3  return Queue[front[Q]]

ENQUEUE (Q, x)
1  if Size(Q) = max − 1
2     then error queue is full
3  Queue[rear[Q]] ← x
4  rear[Q] ← (rear[Q] + 1) mod max

DEQUEUE (Q)
1  if isEmpty()
2     then error queue is empty
3  temp ← Q[front[Q]]
4  Queue[front[Q]] ← null
5  front[Q] ← (front[Q] + 1) mod max
6  return temp
```

پیاده‌سازی صف با آرایه‌ی حلقوی

پیاده‌سازی صف با لیست پیوندی

برای پیاده‌سازی صف Q، می‌توانیم از یک لیست پیوندی یک‌سویه با دو اشاره‌گر $front[Q]$ و $rear[Q]$ که به‌ترتیب به عنصر اول و عنصر آخر لیست اشاره می‌کنند، استفاده کنیم. در

این صورت، عمل DEQUEUE عنصر اول لیست را حذف می‌کند و در اختیار قرار می‌دهد و عمل ENQUEUE یک عنصر به انتهای لیست درج می‌کند.

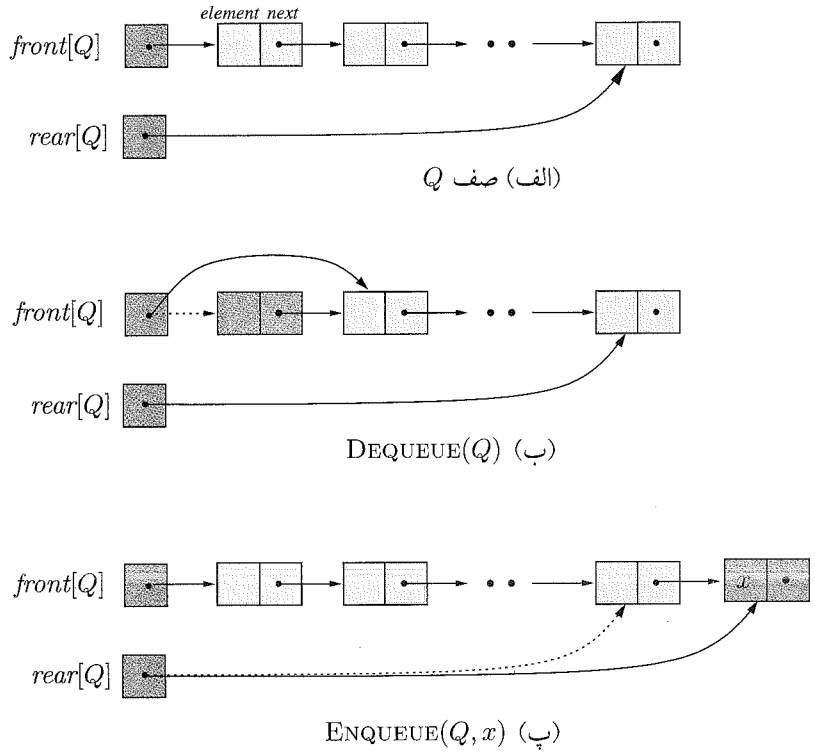
```
ISEMPTY (Q)
1  return size[Q] = 0

ENQUEUE (Q, x)
1  next[rear[Q]] ← ALLOCATE-NODE (x, null)
2  rear[Q] ← next[rear[Q]]
3  size[Q] ← size[Q] + 1

DEQUEUE (Q, x)
1  if ISEMPTY(Q)
2     then error QUEUE IS EMPTY
3  n ← front[Q]
4  front[Q] ← next[front[Q]]
5  temp ← element[n]
6  FREE-NODE(n)
7  size[Q] ← size[Q] − 1
8  return temp
```

پیاده‌سازی صف با لیست پیوندی یک‌سویه

شکل ۴-۱۶ نحوه‌ی انجام این اعمال را نشان می‌دهد. رویه‌های مورد نیاز در ادامه بیان شده‌اند.

۴-۳ کاربردهایی از لیست‌ها

در این بخش با چند کاربرد لیست آشنا می‌شویم. ابتدا نشان می‌دهیم که الگوریتم مرتب‌سازی ادغامی یک لیست را تنها با تغییر اشاره‌گرها و در زمان بهینه مرتب می‌کند. سپس لیست‌های کلی را بیان می‌کنیم که برای انجام اعمال مختلف بر روی عبارات ریاضی به‌کار می‌آید و نشان می‌دهیم که با استفاده از پشته می‌توان یک رویه‌ی بازگشتی را به معادلِ غیربازگشتی آن تبدیل کرد.



شکل ۴-۱۶ پیاده‌سازی صف با لیست پیوندی یک‌سویه. (الف) یک صف، (ب) پس از عمل حذف عنصر ابتدایی، و (پ) پس از عمل درج در انتها.

۴-۳-۱ مرتب‌سازی ادغامی

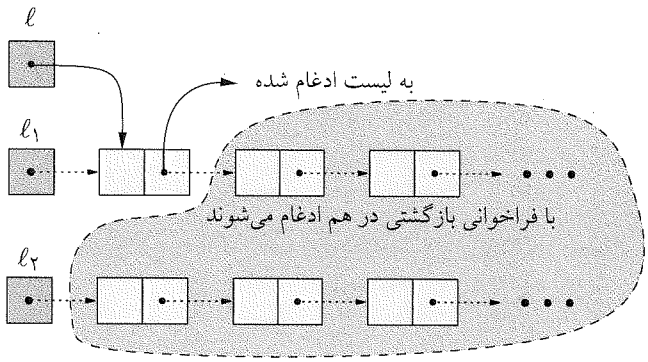
هدف از این بخش، مرتب‌سازی کارای یک لیست پیوندی، تنها با تغییر اشاره‌گرهای آن است، به‌طوری که عناصر لیست برحسب مؤلفه‌ی *key* آن‌ها مرتب شوند. نشان می‌دهیم که این‌کار را الگوریتم مرتب‌سازی ادغامی انجام می‌دهد.

در این روش، عناصر لیست ورودی را به دو لیست تقریباً هم‌اندازه (با اختلاف حداکثر ۱ عنصر) تقسیم می‌کنیم. هر لیست را به‌صورت بازگشتی مرتب و سپس آن‌ها را در هم ادغام می‌کنیم تا لیستی شامل عناصر مرتب به‌دست آید. رویه‌ی MERGESORT(L) این کار را انجام می‌دهد.

در این الگوریتم، SPLIT(ℓ) لیست n عضوی با عنصر اول ℓ را دریافت می‌کند و فقط با تغییر اشاره‌گرهای آن، دو لیست L_1 و L_2 به‌ترتیب با عناصر اول ℓ_1 و ℓ_2 تولید می‌کند. اندازه‌های این دو لیست به‌ترتیب برابر $\lceil \frac{n}{2} \rceil$ و $\lfloor \frac{n}{2} \rfloor$ است. پس از تقسیم، L_1 و L_2 به‌صورت بازگشتی مرتب و سپس با رویه‌ی MERGE($first[L_1]$, $first[L_2]$) در هم ادغام می‌شوند.

```
MERGESORT (L)
    ▷ یک لیست پیوندی را فقط با تغییر اشاره‌گرها مرتب می‌کند
 1  if SIZE(L) > 1
 2     then CREATE(L1);  CREATE(L2)
 3          first[L1], first[L2 ← SPLIT(first[L])
 4          MERGESORT (L1)
 5          MERGESORT (L2)
 6          first[L]←  MERGE(first[L1], first[L2])
 7          return L
```

شکل ۴-۱۷ نحوه‌ی ادغام دو لیست مرتب با عناصر اول ℓ_1 و ℓ_2 را نشان می‌دهد. این کار مانند ادغام دو آرایه‌ی مرتب است ولی فقط با تغییر اشاره‌گرها انجام می‌شود. نخست، دو عنصر اول دو لیست با هم مقایسه می‌شوند؛ بدیهی است که کوچک‌ترین آن‌دو اولین عنصر لیست نهایی است. فرض کنید که این عنصر ℓ_1 باشد. در این‌صورت، این عنصر را کنار می‌گذاریم و بقیه‌ی عناصر لیست L_1 و لیست L_2 را به‌صورت بازگشتی در هم ادغام می‌کنیم. سپس عنصر کنار گذاشته‌شده را به‌عنوان عنصر اول به آن اضافه می‌کنیم.

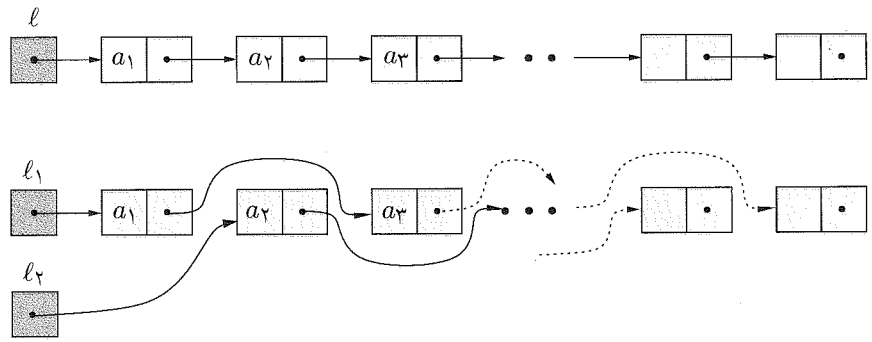


شکل ۴-۱۷ ادغام دو لیست مرتب با عناصر اول ℓ_1 و ℓ_2 و تولید یک لیست مرتب با عنصر اول ℓ.

مراحل مختلف ادغام دو لیست در رویه‌ی $\text{MERGE}(\ell_1, \ell_2)$ آمده است.

```
MERGE (ℓ1, ℓ2)
 1  if ℓ1 = null
 2     then return ℓ2
 3  if ℓ2 = null
 4     then return ℓ1
 5  if key[ℓ1] ≤ key[ℓ2]
 6     then next[ℓ1] ← MERGE(next[ℓ1], ℓ2)
 7          return ℓ1
 8     else next[ℓ2] ← MERGE(ℓ1, next[ℓ2])
 9          return ℓ2
```

برای تقسیم یک لیست n عنصری L با عنصر اول ℓ به دو لیست L_1 و L_2 (به‌ترتیب با عناصر اول ℓ_1 و ℓ_2) با اندازه‌های تقریباً یکسان (یا با حداکثر یک واحد اختلاف)، به این صورت عمل می‌کنیم: عنصر اول و دوم L را به‌ترتیب به‌عنوان عناصر اول لیست‌های L_1 و L_2 کنار می‌گذاریم، سپس عناصر باقی‌مانده‌ی L را به‌صورت بازگشتی به دو لیست هم‌اندازه تقسیم می‌کنیم. با اتصال عناصری که کنار گذاشته‌ایم، لیست‌های حاصل به‌دست می‌آیند. با استقرا می‌توان نشان داد که اندازه‌های لیست‌های L_1 و L_2 به‌ترتیب برابر $\lceil \frac{n}{2} \rceil$ و $\lfloor \frac{n}{2} \rfloor$ است. مراحل مختلف رویه‌ی $\text{SPLIT}(\ell)$ را می‌توانید در شکل ۴-۱۸ ببینید.



شکل ۴-۱۸ تقسیم یک لیست n عضوی به دو لیست $\lceil \frac{n}{2} \rceil$ و $\lfloor \frac{n}{2} \rfloor$ عضوی.

```
SPLIT (ℓ)
1  if ℓ = null
2     then return null, null
3  if next[ℓ] = null
4     then return null, ℓ
5  ℓ1 ← ℓ
6  ℓ2 ← next[ℓ]
7  next[ℓ], next[ℓ2] ← SPLIT(next[ℓ2])
8  return ℓ1, ℓ2
```

روشن است که زمان اجرای این الگوریتم با رابطه‌ی بازگشتی ۳-۱۸ قابل نمایش است. چنان‌چه در بخش ۳-۵-۱ نشان دادیم، این زمان از $\mathcal{O}(n \lg n)$ و بهینه است.

۴-۳-۲ لیست‌های کلی

در این بخش هدف کار با چندجمله‌ای‌ها در یک حالت کلی است به‌طوری‌که بتوان آن‌ها را به‌صورت کارا پیاده‌سازی کرد و اعمال مختلف را بر روی آن انجام داد. چندجمله‌ای‌های مورد نظر جمع عبارت‌هایی از نوع $cx^{e_x}y^{e_y}z^{e_z}\ldots$ هستند که c ضریب این عبارت و e_x، e_y، e_z، ...به‌ترتیب ضریب‌های توان متغیرهای x، y، z و ... هستند. مثلاً

$$P = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 6x^3y^4z + 2xyz \quad (۱-۴)$$

مثالی از چنین عبارت ریاضی است. هدف، طراحی داده‌ساختاری مناسب برای حالت کلی این عبارت‌هاست، به‌طوری که بتوان اعمال مختلفی مانند موارد زیر را در زمان مناسب بر روی آن انجام داد:

- چاپ یک عبارت،
- تعیین بیش‌ترین عمق یک عبارت ،
- کپی‌کردن یک عبارت،

- جمع یا تفریق دو عبارت، و
- مشتق‌گیری از عبارت برحسب یکی از متغیرها.

می‌توان از یک لیست پیوندی یک‌سویه برای پیاده‌سازی یک چندجمله‌ای استفاده کرد. در آن‌صورت هر عنصر این لیست باید ضریب و همه‌ی توان‌های متغیرهای آن عبارت را ذخیره کند، مثل این عنصر برای سه متغیر:

coef	*expx*	*expy*
expz	*link*	

روشن است که این روش برای حالت کلی که تعداد متغیرها از قبل مشخص نباشد قابل استفاده نیست.

روش دیگر، تعریف بازگشتی یک چندجمله‌ای و پیاده‌سازی آن با لیست‌های کلی است. در این روش اگر

$$P(z, y, x, n_z, n_y, n_x)$$

یک چندجمله‌ای از نوع فوق مثلاً برحسب متغیرهای x، y و z به‌ترتیب با توان‌های n_x، n_y و n_z باشد، و در آن متغیرها به‌ترتیب z، y و x اعمال شوند، را به‌صورت زیر تعریف می‌کنیم:

$$\begin{aligned} P(z, y, x, n_z, n_y, n_x) = c_{n_z}(P_{n_z}(y, x, n_y, n_x)z^{n_z} \\ + c_{n_z-1}(P_{n_z-1}(y, x, n_y, n_x)z^{n_z-1} \\ + \cdots + c_0(P_0(y, x, n_y, n_x)) \end{aligned} \qquad (۲-۴)$$

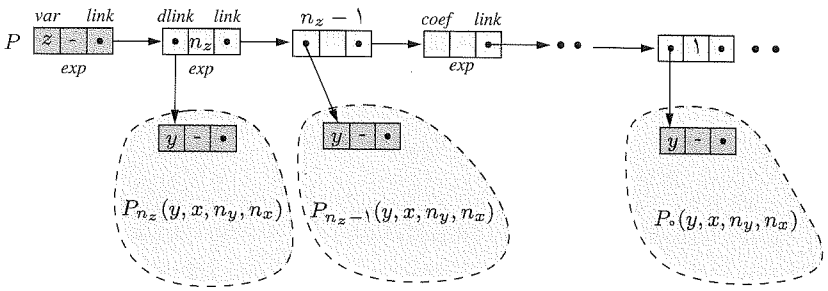
در این تعریف، c_iها ضرایب ثابت و P_iها هم خود چندجمله‌هایی این‌چنین هستند. اگر P_i ثابت نباشد، مقدار ضریب c_i را برابر ۱ قرار می‌دهیم، چرا که مقدار تمام ضرایب ثابت در ضرایب داخل P_i در نظر گرفته می‌شود.

مثلاً چندجمله‌ای ۴-۱ به‌صورت زیر نوشته می‌شود:

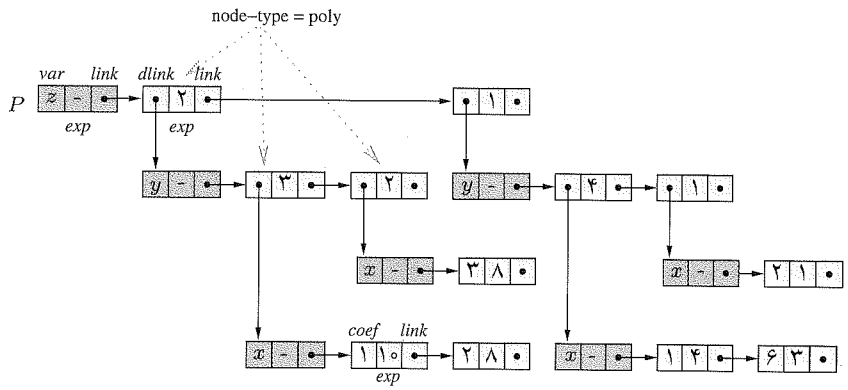
$$((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 + 6x^3)y^4 + 2xy)z \qquad (۳-۴)$$

با این تعریف، یک چندجمله‌ای $P = \sum^i c_i P_i z^i$ (که P_i برحسب z نیست) به‌صورت یک لیست کلی پیاده‌سازی می‌شود که عنصر اول آن حاوی متغیر اول (در این‌جا z) است. عناصر دیگر این لیست هر یک به‌ترتیب متناظر با $c_i P_i z^i$ است. اگر P_i یک عدد ثابت باشد، این عنصر لیست حاوی مقادیر c_i و i خواهد بود. اگر P_i خود یک چندجمله‌ای باشد، c_i برابر ۱ منظور می‌شود (مقدارش در ضرایب P_i منظور می‌شود) و عنصر مربوط به آن در لیست اصلی شامل i و اشاره‌گری به لیست P_i است که خود یک چندجمله‌ای با یک متغیر کم‌تر است که به‌صورت بازگشتی به‌همین صورت ساخته می‌شود.

در شکل ۴-۱۹ گونه‌ی کلی و نیز لیست مربوط به مثال این بخش نشان داده شده است. در این پیاده‌سازی عناصر لیست یا از گونه‌ی «چندجمله‌ای» هستند یا نه، که در شکل نشان داده شده است.



$$P(z, y, x, n_z, n_y, n_x) = P_{n_z}(y, x, n_y, n_x)z^{n_z} + P_{n_z-1}(y, x, n_y, n_x)z^{n_z-1} + \cdots + P_0(y, x, n_y, n_x)$$



شکل ۴-۱۹ لیست مربوط به $((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 + 6x^3)y^4 + 2xy)z$.

حال با داشتن این ساختار از لیست می‌توان اعمال مختلف را پیاده‌سازی کرد. مثلاً، رویه‌ی PRINT-PLIST(P) چندجمله‌ای P را به‌فرم بازگشتی شبیه رابطه‌ی ۴-۳ می‌نویسد. مثلاً اگر ورودی لیست شکل ۴-۱۹ باشد، خروجی این عبارت خواهد بود:

```
((x^10 + 2x^8) y^3 + 3x^8 y^2)z^2 + ((x^4 + 6x^3)y^4 + (2x^1)y^1)z
```

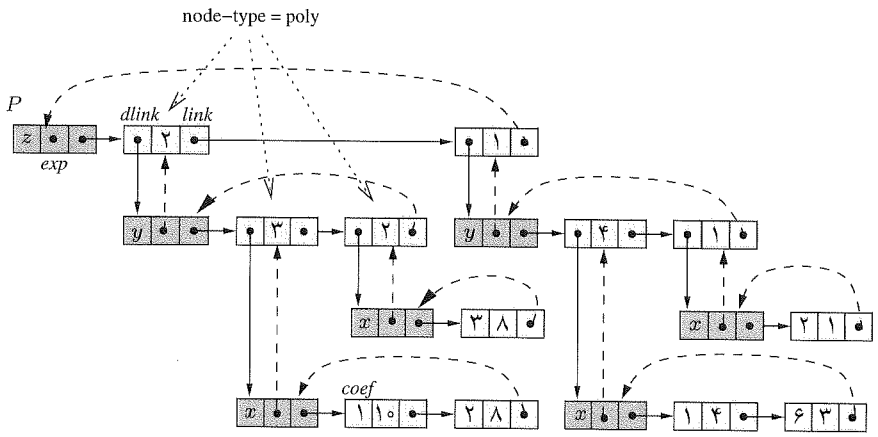
```
PRINT-PLIST (P)
    ▷ فرض می‌کنیم که لیست درست ساخته شده است
 1  p ← P
 2  X = var[p] ▷ متغیر اصلی چندجمله‌ای P
 3  while p ≠ null
 4     do if NODE-TYPE(p) = poly
 5           then PRINT '('
 6                PRINT-PLIST (dlink[p])
 7                PRINT ') ^ exp[p]'
 8           else
 9                ▷ با فرض درستی داده‌گونه‌ها
10                PRINT '+coef[p] X^exp[p]'
11        p ← link[p]
```

همچنین می‌توان با توجه به ساختار بازگشتی این لیست، عمق آن‌را با رویه‌ی DEPTH-PLIST به‌دست آورد.

```
DEPTH-PLIST (P)
    ▷ فرض می‌کنیم که لیست درست ساخته شده است
1  p ← P
2  depth ← 0
3  while p ≠ null
4     do if NODE-TYPE(p) = poly
5           then dp ← DEPTH-PLIST (dlink[p]) + 1
6                depth ← max{depth, dp}
7        p ← link[p]
8  return DEPTH
```

نخ‌کشی لیست‌ها

در لیست‌های کلی می‌توان از اشاره‌گرهای **null** هم استفاده کرد و آن‌ها را به گره‌های مشخصی اشاره داد به‌طوری که این کار موجب سهولت برخی از اعمال مورد نظر شود. در این‌صورت، به چنین اشاره‌گرهایی «نخ[۳۴]» گفته می‌شود. البته در پیاده‌سازی باید یک بیت به مؤلفه‌ها اضافه کرد تا بتوان بین اشاره‌گر واقعی و نخ تفاوت گذاشت. مثلاً، لیست کلی شکل ۴-۱۹ را می‌توان به‌صورت شکل ۴-۲۰ «نخ‌کشی» کرد، به‌طوری‌که اشاره‌گرهای تهیِ *link* به سرلیست افقی و اشاره‌گرهای تهیِ *exp* هر گره به گرهِ بالاییِ از نوع poly اشاره کنند.



شکل ۴-۲۰ لیست نخ‌کشی شده‌ی شکل ۴-۱۹. نخ‌ها به‌صورت خط‌چین نشان داده شده‌اند.

با این نخ‌کشی، می‌توان برخی از الگوریتم‌های بازگشتی قبل را به‌صورت غیر بازگشتی نوشت. مثلاً DEPTH-THREADED-PLIST گونه‌ی غیر بازگشتی DEPTH-PLIST است که عمق یک عبارت را مشخص می‌کند. چنان‌چه مشخص است، در این رویه گاهی لازم است نخ بودن یک اشاره‌گر مشخص شود.

البته لیست اصلی را می‌توان بدون نخ و بدون استفاده از پشته هم به‌صورت غیربازگشتی پیمایش کرد. در این روش هنگام جلو رفتن با یک متغیر کمکی، اشاره‌گرها را وارون می‌کنیم تا مسیر برگشت درست شود. از طریق این اشاره‌گرهای وارون به جای اول برمی‌گردیم و در مسیر بازگشت مجدداً اشاره‌گرها را به جهت اول خودشان باز می‌گردانیم. جزئیات این پیاده‌سازی که در زباله‌روبی استفاده می‌شود با شماست!

[۳۴] thread

```
DEPTH-THREADED-PLIST (P)
 1  p ← P
 2  depth ← 0
 3  maxdepth ← 0
 4  while true
 5     do if Node-Type[p] = poly
 6           then depth ← depth + 1
 7                p ← dlink[p]
 8           else while link[p] is a thread
 9                   do p ← link[p]
10                      maxdepth = max{maxdepth, depth}
11                      if exp[p] ≠ null
12                        then p ← exp[p]
13                             depth ← depth − 1
14                        else return maxdepth
15        if link[p] is pointer
16          then p ← link[p]
```

۴-۳-۳ تبدیل الگوریتم‌های بازگشتی به غیربازگشتی

شبیه‌سازی فراخوان‌های بازگشتی یکی دیگر از کاربردهای پشته است. این بحث عمدتاً برای فهم بهتر از رفتار الگوریتم‌های بازگشتی ارائه می‌شود و از آن می‌توان به‌عنوان شروعی برای تبدیل خودکار رویه‌های بازگشتی به رویه‌های غیربازگشتیِ معادل استفاده کرد.

هر فراخوانی شامل دو مرحله‌ی اصلی هستند:

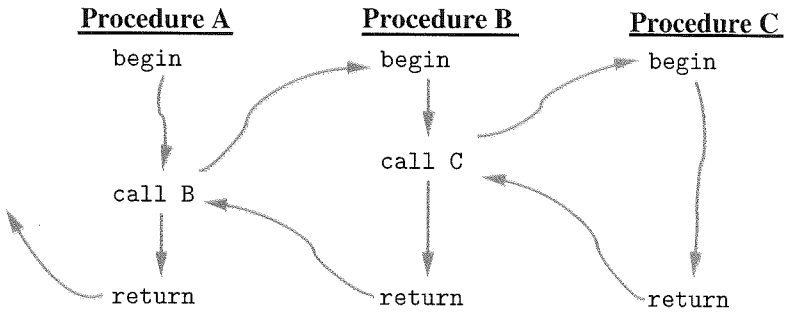
۱. عمل فراخوانی[۳۵]

۲. بازگشت[۳۶] از یک فراخوانی

---

[۳۵] call
[۳۶] return

شکل ۴-۲۱ مثالی از دو فراخوانی را نشان می‌دهد که در آن رویه‌ی A در برنامه‌ی خود رویه‌ی B را فرامی‌خواند. B هم در برنامه‌اش رویه‌ی C را فرامی‌خواند. رویه‌ی C در پایان عمل بازگشت را انجام می‌دهد و به دستور پس از فراخوانیِ C در رویه‌ی B باز می‌گردد. رویه‌ی B هم در پایان به رویه‌ی A بازگشت می‌کند و A هم پس از پایان کارش به رویه‌ی بالاتر باز می‌گردد.



شکل ۴-۲۱ انتقال کنترل برنامه در فراخوانی و بازگشت

فراخوانی مانند وقفه[۳۷] در سخت‌افزار عمل می‌کند؛ کلیه‌ی متغیرها و مقادیرشان در لحظه‌ی فراخوانی ذخیره می‌شوند، عملیاتی صورت می‌گیرد و هنگام بازگشت، وضعیت پیش از فراخوانی مجدداً بازسازی می‌شود. به‌عبارت دیگر، در هر فراخوانی:

۱. کلیه‌ی متغیرهای محلی (در حالت کلی، همه‌ی متغیرهای دسترس‌پذیر) و مقادیرشان در پشته‌ی سیستم قرار می‌گیرند (PUSH).

۲. آدرس بازگشت به پشته منتقل می‌شود (PUSH).

۳. عمل «انتقال پارامترها[۳۸]» انجام می‌شود. پارامترها ممکن است از نوع ارزشی[۳۹] یا آدرسی[۴۰] باشند.

۴. کنترل برنامه (ثبات شمارنده‌ی برنامه[۴۱]) به ابتدای رویه‌ی جدید اشاره می‌کند.

عمل بازگشت، عکس عملیات فوق را انجام می‌دهد:

---

[۳۷] interrupt
[۳۸] parameter passing
[۳۹] value
[۴۰] variable
[۴۱] program counter

۱. مقادیر متغیرهای محلی از رکورد بالای پشته برداشته و در آن متغیرها قرار داده می‌شوند.

۲. آدرس بازگشت را از بالای پشته می‌خواند.

۳. آخرین رکورد را از پشته برمی‌دارد (POP).

۴. کنترل برنامه، کار را از آدرس بازگشت (بند ۲) ادامه می‌دهد.

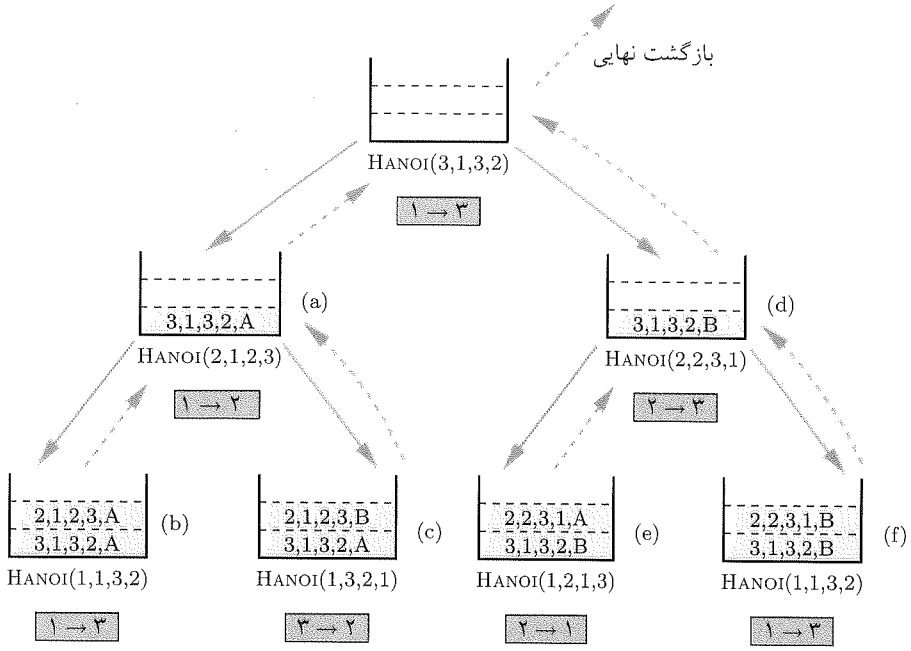
مثال: مسئله‌ی برج‌های هانوی

این مفاهیم را با الگوریتم بازگشتی مسئله‌ی برج‌های هانوی، که در بخش ۳-۴-۲ با جزئیات بررسی شد، دنبال می‌کنیم.

```
HONOI (n, f, t, h)
    ▷ سکه را از میله‌ی f به میله‌ی t و به کمک میله‌ی h منتقل می‌کند
1   if n = 1
2     then Print f ⟶ t   ▷ سکه‌ی بالایی را از میله‌ی f به میله‌ی t منتقل کن
3     else HONOI(n − 1, f, h, t)
4          A: Print f ⟶ t   ▷ سکه‌ی بالایی را از میله‌ی f به میله‌ی t منتقل کن
5          HONOI(n − 1, h, t, f)
6          B:
```

برای درک بهتر، مراحل مختلف فراخوانی و بازگشت برای اجرای HONOI(3,1,3,2) در شکل ۴-۲۲ نشان داده شده است.

در ابتدا پشته خالی است و اولین فراخوانی انجام می‌شود. مطابق الگوریتم، این موجب فراخوانی بازگشتیِ HONOI(2,1,2,3) می‌شود. برای شبیه‌سازی، نخست مقادیر فعلی متغیرهای n، f، t و h و سپس آدرس بازگشت در بالای پشته قرار می‌گیرند. در این حالت $n = ۳$، $f = ۱$، $t = ۳$ و $h = ۲$ و آدرس بازگشت را A فرض می‌کنیم که در رویه‌ی HONOI به عنوان آدرس ۳ نشان داده شده است. بنابراین رکورد بالای پشته (3,1,3,2,A) خواهد بود. انتقال پارامتر انجام می‌شود، یعنی گمارش‌های $n = ۲$، $f = ۱$، $t = ۲$ و $h = ۳$ اجرا و کار الگوریتم از سطر اول آن دنبال می‌شود. این با حالت (a) در شکل نشان داده



شکل ۴-۲۲ مراحل مختلف فراخوانی‌های بازگشتی و مقادیر پشته‌ی سیستم در HONOI(3,1,3,2).

شده است. HONOI(2,1,2,3) هم مانند یک فراخوانی مستقل اجرا می‌گردد که خود موجب فراخوانی HONOI(1,1,3,2) می‌گردد، ولی پیش از آن، رکورد (2,1,2,3,A) در بالای پشته قرار می‌گیرد (حالت (b)). چون $n = ۱$، این فراخوانی دستور شماره‌ی ۱ را اجرا می‌کند و در خروجی ۳ ⟵ ۱ را می‌نویسد تا بالاترین سکه از میله‌ی ۱ به میله‌ی ۳ منتقل شود.

سپس عمل بازگشت انجام می‌شود؛ به این صورت که مقادیر بالای پشته به متغیرها گمارده می‌شود (یعنی مجدداً به حالت (a) برمی‌گردیم) و کار از دستور A با این مقادیر دنبال می‌شود. یعنی ۲ ⟵ ۱ در خروجی نوشته می‌شود (چون $f = ۱$ و $t = ۲$ است) و HONOI(1,3,2,1) فراخوانده می‌شود. توجه کنید که آدرس بازگشت این فراخوانی B است و این مقادیر در بالای پشته درج می‌شود. این فراخوانی ۲ ⟵ ۱ را چاپ می‌کند و پس از POP مقادیر بالای پشته و گمارش به متغیرها، کار را از آدرس B، که خود یک بازگشت است، دنبال می‌کند. بنابراین پس از رفتن به حالت (a) از حالت (c)، مجدداً مقادیر بالای پشته POP شده (یعنی به حالت اول با پشته‌ی خالی باز می‌گردیم) و کار از آدرس A دنبال می‌شود.

سپس حالت پشته‌ها به‌ترتیب به (d)، (e) و (f) می‌رود و با سه بار بازگشت، در پایان به برنامه‌ای بازمی‌گردد که این رویه را فراخوانده است. اگر ترتیب حرکت‌ها را دنبال کنید، می‌بینید که حرکت‌ها به‌درستی انجام شده‌اند.

حال می‌خواهیم یک رویه‌ی غیربازگشتی بنویسیم که همین مراحل کار رویه‌ی بازگشتی را که گفتیم شبیه‌سازی کند. برای این کار یک پشته‌ی S را که در ابتدا تهی است ایجاد می‌کنیم. هر رکورد این پشته شامل ۴ مقدار عددی، برای متغیرهای n، f، t و h، و یک مقدار نویسه‌ای 'A' یا 'B' به‌عنوان آدرس بازگشت است. سطر با برچسب Rec-Call در این رویه به‌معنی شروع یک فراخوانی بازگشتی است که با **goto** Rec-Call آغاز می‌شود. سطر با برچسب Return-Label هم شروع عمل بازگشت است.

```
NonRecursive-Honoi (n, f, t)
 1  ▷ پشته‌ی S شامل آدرس بازگشت و مقادیر همه‌ی متغیرهای محلی است
 2  Create-Stack(S)
 3  h ← the other peg
 4  ▷ دستور شماره‌ی 4 آغاز یک فراخوانی بازگشتی است
 5  Rec-Call: if n = 1
 6     then Print f ⟶ t  ▷ سکه‌ی بالایی را از میله‌ی f به میله‌ی t منتقل کن
 7          goto Return-Label
 8     else Push (S, StackRec(n, f, t, h, 'A'))
 9          n, f, t, h ← n − 1, f, h, t  ▷ انتقال پارامترها با فرض ارزشی بودن
10          goto Rec-Call
    ▷ از این دستور عمل بازگشت شبیه‌سازی می‌شود
11  Return-Label: if not isEmpty(S)
12     then return-address, n, f, t, h ← Pop(S)
13          switch
14             case return-address = 'A'
15                do Print f ⟶ t
                     ▷ سکه‌ی بالایی را از میله‌ی f به میله‌ی t منتقل کن
16                   Push(S, StackRec(n, f, t, h, 'B'))
                     ▷ انتقال پارامترها
17                   n, f, t, h ← n − 1, h, t, f
18                   goto Rec-Call
19             case return-address = 'B'
20                do goto Return-Label
```

مانند الگوریتم بازگشتی، هر فراخوانی در این الگوریتم از سطر ۴ آغاز می‌شود. اگر $n = ۱$ باشد، تنها سکه (در عمل سکه‌ی رویی) را حرکت می‌دهد و باز می‌گردد، وگرنه مقادیر فعلی متغیرها و آدرس بازگشت A را در بالای پشته درج می‌کند و پس از انتقال پارامترها برای انجام یک فراخوانی بازگشتی دیگر به سطر مربوط می‌رود. سطر ۱۱ که عمل بازگشت را شبیه‌سازی می‌کند: اگر پشته تهی باشد، یعنی این آخرین بازگشت است و باید به برنامه‌ای که این رویه را فراخوانده است بازگردد. اگر پشته تهی نبود، باید مقادیر بالای پشته پس از دریافت و گمارش به متغیرها دور ریخته شود و بسته به آدرس بازگشت، دو کار مختلف انجام دهد: یکی انجام یک فراخوانی بازگشتی دیگر (با آدرس بازگشت B) و دیگری که خود یک بازگشت است با یک goto در سطر ۱۸ انجام می‌شود.

آن‌چه که گفته شد را می‌توان به‌صورت خودکار، برای رویه‌های بازگشتی با پارامترهای ارزشی هم پیاده‌سازی کرد. اگر پارامتری آدرسی باشد، باید به‌جای مقدار آن، آدرس آن‌را در پشته ذخیره کرد.

حذف آخرین بازگشت

آخرین فراخوانی بازگشتی‌ای را که در هیچ شرایطی پس از آن، دستوری که از مقادیر متغیرها استفاده کند اجرا نشود «آخرین بازگشت[۴۲]» می‌گوییم. نشان می‌دهیم که دستور این بازگشت را می‌توان بدون استفاده از پشته حذف کرد.

```
PROC-1 (a, b, c)
    ...
    ...
    ...
    PROC-1(A,B,C)      ▷ آخرین سطر؛ پس از آن هیچ دستوری اجرا نمی‌شود
A:
```

چرا که هنگام فراخوانی علاوه بر مقادیر متغیرها، آدرس بازگشت (A) نیز در پشته ذخیره و عمل فراخوانی انجام می‌شود که طی آن ممکن است مقادیر پارامترها تغییر کند.

[۴۲] tail recursion

ولی پس از بازگشت، مقادیر اولیه‌ی پارامترها در متغیرهای مربوط گمارده می‌شوند و از نقطه‌ی A کار دنبال می‌شود. اما A هم خود یک بازگشت است و قبل از آن هم از مقادیر قبلی پارامترها استفاده‌ای نشده است. بنابراین به‌جای رویه‌ی فوق می‌توان رویه‌ی غیربازگشتی زیر را استفاده کرد و نیازی هم به استفاده از پشته نیست.

```
PROC-2 (a, b, c)
1:
    ...
    ...
    ...
    a, b, c ← A, B, C      ▷ انتقال پارامترها
    goto 1
```

به‌عنوان مثالی از حذف آخرین فراخوانی، توجه کنید که الگوریتم برج هانوی زیر معادل الگوریتم اصلی است.

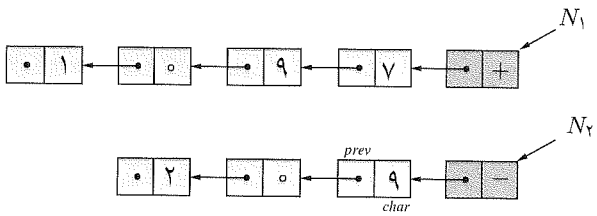
```
TOWERS-OF-HONOI2 (n, f, t, h)
    ▷ حذف آخرین بازگشت
1   1: if n = 1
2      then Print f ⟶ t   ▷ سکه‌ی بالایی را از میله‌ی f به میله‌ی t منتقل کن
3      else TOWERS-OF-HONOI2(n − 1, f, h, t)
4           Print f ⟶ t   ▷ سکه‌ی بالایی را از میله‌ی f به میله‌ی t منتقل کن
5           n, f, h ← n − 1, h, f      ▷ انتقال پارامترها
6           goto 1
```

حال چون TOWERS-OF-HONOI2 تنها یک فراخوانی بازگشتی (در سطر ۳) دارد، می‌توان آن‌را با پشته ولی بدون استفاده از آدرس بازگشت، به رویه‌ی غیربازگشتی معادل تبدیل کرد.

```
NonRecursive-Honoi2 (n, f, t)
 1  ▷ پشته‌ی S شامل آدرس بازگشت و مقادیر همه‌ی متغیرهای محلی است
 2  Create-Stack(S)
 3  h ← (غیر از f و t) میله‌ی سوم
 4  ▷ دستور بعد آغاز یک فراخوانی بازگشتی است
 5  Rec-Call: if n = 1
 6    then
 7         Print f ⟶ t  ▷ سکه‌ی بالایی را از میله‌ی f به میله‌ی t منتقل کن
 8         goto Return-Label
 9    else Push(S, StackRec(n, f, t, h))
10         n, f, t, h ← n − 1, f, h, t  ▷ انتقال پارامترها
11         goto Rec-Call
12  ▷ end recursive call
13  Return-Label: if not isEmpty(S)
14    then n, f, t, h ← Pop(S)
15         Print f ⟶ t  ▷ سکه‌ی بالایی را از میله‌ی f به میله‌ی t منتقل کن
16         n, f, t, h ← n − 1, h, t, f  ▷ انتقال پارامترها
17         goto Rec-Call
```

تمرین‌های بخش ۳-۴

۱.۳-۴ لیست پیوندی یکی از روش‌های پیاده‌سازی اعداد با تعداد ارقام زیاد و نامشخص است. مثلاً اعداد ۱۰۹۷+ و ۲۰۹− را می‌توان به‌صورت دو لیست زیر پیاده‌سازی کرد. مؤلفه‌های هر عنصر لیست، *char* (که یا + و یا − و یا یکی از ارقام ۰ تا ۹) است و *prev* که به رقم قبلی اشاره می‌کند. لیست‌ها سرلیست دارند و علامت عدد را در خود ذخیره می‌کنند.



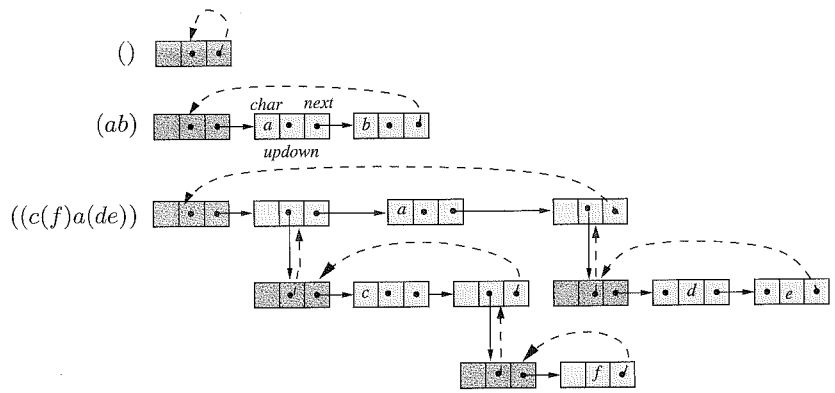
رویه‌ای بنویسید تا دو عدد (مثبت یا منفی) را با هم جمع کند و یک لیست جدید برای عدد مجموع بسازد.

۲.۳-۴ الگوریتم بازگشتی بنویسید تا یک لیست پیوندی دوسویه را وارون نماید.

۳.۳-۴ داده‌ساختاری طراحی کنید که بتواند اعمال PUSH، POP، FINDMIN (یافتن و برگرداندن کوچک‌ترین عنصر بین عناصر موجود) و FINDMAX را در $O(1)$ انجام دهد. اعمال را به‌زبان CLRS بنویسید و تحلیل کنید. ثابت کنید اگر عمل DELETEMIN هم اضافه شود، دست‌کم یکی از این اعمال باید در $\Omega(\lg n)$ انجام شود.

۴.۳-۴ n عدد طبیعی کوچک‌تر از k داده شده‌اند. با انجام پیش‌پردازشی که بیش‌تر از $O(n+k)$ نباشد، می‌خواهیم این پرسش را در زمان $O(1)$ انجام دهیم: «تعداد عناصر موجود در بازه‌ی $[a,b]$ چندتاست؟» داده‌ساختاری ساده برای این کار پیش‌نهاد کنید و الگوریتم‌های لازم را بنویسید.

۵.۳-۴ لیسپ[۴۳] یک زبان برنامه‌نویسی است که به‌صورت پرانتزی بیان می‌شود و برای نمایش و اجرای دستورهای این زبان از لیست استفاده می‌شود. گونه‌ی ساده‌ی این لیست‌ها به‌صورت شکل زیر نشان داده شده است. چنان‌چه مشخص است، این لیست‌ها نخ‌کشی شده و در آن‌ها از سرلیست استفاده شده است و مؤلفه‌های هر عنصر آن *char*، *next* و *updown* است.



الگوریتم‌هایی بنویسید تا

الف) به‌صورت بازگشتی فرم پرانتزی یک لیست را بنویسد.

ب) همین کار را به‌صورت غیر بازگشتی انجام دهد.

پ) به‌صورت بازگشتی از یک عبارت پرانتزی، لیست را ایجاد کند.

۶.۳-۴ رویه‌ی ISEQUAL(P_1, P_2) را بنویسید که دو چندجمله‌ای کلی P_1 و P_2 را با هم مقایسه کند و در صورت یک‌سان بودن، مقدار «صحیح» را برگرداند.

⋆ **۷.۳-۴** تعدادی مایکروفیش در اختیار داریم که هر کدام با یک کد دودویی k رقمی مشخص شده است. می‌خواهیم این مایکروفیش‌ها را به‌وسیله‌ی ماشینی با دو پشته‌ی ۰ و ۱ مرتب کنیم. این ماشین

[۴۳] Lisp: List Processing

فقط می‌تواند اعمال زیر را انجام دهد:

- $\text{POP}(a)$: از بالای پشته‌ی a یک کارت بردار،
- $\text{PUSH}(a, m)$: کارت m را در بالای پشته‌ی a قرار بده،
- $\text{COMBINE}(a)$: کل کارت‌های پشته‌ی $a-1$ را با همان ترتیب به بالای پشته‌ی a منتقل کن.
- $\text{PULL}(a, c)$: همه‌ی کارت‌های پشته‌ی a را که بیت c آن‌ها صفر است با همان ترتیب به پشته‌ی $a-1$ که باید خالی باشد منتقل کن. ترتیب کارت‌هایی که در a باقی می‌ماند با قبل از این عمل تفاوتی نمی‌کند.

یک الگوریتم خطی برای مرتب‌سازی کارت‌ها ارائه و آن‌را اثبات و تحلیل کنید.

⋆ **۸.۳-۴** فرض کنید کُد پیاده‌سازی پشته را در اختیار داریم و می‌خواهیم از آن برای پیاده‌سازی صف استفاده کنیم. برای این کار دو پشته‌ی S_1 و S_2 را ایجاد می‌کنیم. برای انجام «درج x در صف»، عمل $\text{PUSH}\ (S_1, x)$ انجام می‌دهیم و برای انجام $\text{DEQUEUE} \leftarrow x$ تک تک عناصر S_1 را POP کرده، در $\text{PUSH}\ S_2$ می‌کنیم. $x \leftarrow \text{POP}(S_2)$ کار را کامل می‌کند.

الف) فرض کنید که با یک صف تهی شروع می‌کنیم، و به‌ترتیب: ۳ درج، ۲ حذف، ۳ درج و ۲ حذف بر روی صف انجام می‌دهیم. جمع کل هزینه‌های این ۱۰ عمل چقدر است و در انتها چند عنصر در هر پشته قرار دارد؟

ب) فرض کنید که n درج و n حذف با یک ترتیب دل‌خواه انجام شده است. حداکثر هزینه‌ی یکی از این اعمال چقدر است؟ (جواب دقیق بدهید، نه O). دنباله‌ای از اعمال را که منجر به بدترین رفتار شده است نشان دهید.

پ) با فرض شروع از یک صف تهی، فرض کنید که دنباله‌ی دل‌خواهی از این اعمال بر روی این داده‌ساختار انجام می‌شود. هزینه‌ی سرشکن‌شده‌ی هرکدام از این اعمال را حساب کنید. جواب دقیق مد نظر است. می‌توانید برای حل این بند از هر روشی استفاده کنید.

۹.۳-۴ تعداد راه‌های مختلف ساختن یک لیست n عنصری از k لیست $\frac{n}{k}$ عنصری به‌طوری که ترتیب عناصر هر لیست کوچک در لیست نهایی هم یک‌سان باشد چند تاست؟ روش محاسبه‌ی خود را نشان دهید. ثابت کنید که $\Omega(n \lg k)$ کران پایین این عدد است.

۱۰.۳-۴ مجموعه‌ای از لیست‌ها با n عنصر داده شده‌اند. برروی هر لیست اعمال زیر انجام می‌شوند:

- $\text{INSERT}(x, L_k)$: عنصر x را در لیست L_k درج کن.
- $\text{SUM}\ (L_k)$: مجموع عناصر موجود در لیست L_k را به‌دست آور.

الف) با استفاده از روش حساب‌داری ثابت کنید که هر دو عمل در زمان سرشکنی $O(1)$ قابل انجام است.

ب) همین کار را با استفاده از روش تابع پتانسیل انجام دهید.

⋆ **۱۱.۳-۴** تمرین ۴-۲-۱.۴ از شما خواسته است نشان دهید چگونه می‌توان یک لیست n عضوی را به‌طور فشرده در n مکان اول یک آرایه جای داد. فرض کنیم تمام کلیدها متفاوت هستند و لیست فشرده نیز مرتب است؛ یعنی برای هر $i = ۱, ۲, \ldots, n$ که $next[i] \neq \textbf{null}$ داریم: $key[i] < key[next[i]]$. با این فرض‌ها، با حل این مسئله شما نشان خواهید داد که الگوریتم تصادفی زیر می‌تواند برای جست‌وجو در لیست در زمان میانگین $\mathcal{O}(\sqrt{n})$ به‌کار رود.

```
COMPACT-LIST-SEARCH (L, n, k)
 1  i ← head[L]
 2  while i ≠ null and key[i] < k
 3      do j ← RANDOM(1, n)
 4         if key[i] < key[j] and key[j] ≤ k
 5            then i ← j
 6                 if key[i] = k
 7                    then return i
 8         i ← next[i]
 9  if i = null or key[i] > k
10    then return null
11    else return i
```

اگر سطرهای ۳ تا ۷ این الگوریتم را نادیده بگیریم، یک الگوریتم معمولی برای جست‌وجو در لیست‌های پیوندی مرتب خواهیم داشت که در آن i به‌نوبت، به هر یک از مکان‌های لیست اشاره می‌کند. جست‌وجو زمانی پایان می‌یابد که اندیس i از انتهای لیست خارج شود، یا زمانی که $key[i] \geq k$. در حالت اخیر، اگر $key[i] = k$، واضح است که ما یک کلید با مقدار k یافته‌ایم. اگر $key[i] > k$ باشد، بعد از آن دیگر نمی‌توانیم کلیدی با مقدار k پیدا کنیم و بنابراین باید به جست‌وجو خاتمه دهیم. سطرهای ۳ تا ۷ مکان جلوتری از لیست مانند j را به طور تصادفی انتخاب می‌کند و به آن می‌پرد. چنین پرشی در صورتی مفید خواهد بود که $key[j]$ از $key[i]$ بزرگ‌تر باشد و از k بزرگ‌تر نباشد؛ در این صورت j مکانی از لیست را مشخص می‌سازد که i در طول یک جست‌وجوی معمولی در لیست باید به آن می‌رسید. از آن‌جا که لیست فشرده است، می‌دانیم که هر انتخابی از اندیس‌های بین ۱ و n به‌عنوان j، یکی از عناصر لیست را مشخص می‌کند و هیچ‌یک از آن‌ها از خانه‌های آزاد نیست.

به‌جای تحلیل مستقیم COMPACT-LIST-SEARCH، الگوریتمی مرتبط با آن‌را که COMPACT-LIST-SEARCH' می‌نامیم تحلیل می‌کنیم. این الگوریتم یک پارامتر دیگر به نام t نیز می‌گیرد که کران بالایی برای تعداد تکرارهای حلقه‌ی اول محسوب می‌شود.

```
COMPACT-LIST-SEARCH' (L, n, k, t)
 1  i ← head[L]
 2  for q ← 1 to t
```

```
3    do j ← RANDOM(1, n)
4       if key[i] < key[j] and key[j] ≤ k
5          then i ← j
6               if key[i] = k
7                  then return i
8  while i ≠ null and key[i] < k
9     do i ← next[i]
10 if i = null or key[i] > k
11    then return null
12    else return i
```

برای مقایسه‌ی اجرای الگوریتم‌های COMPACT-LIST-SEARCH(L, n, k) و COMPACT-LIST-SEARCH'(L, n, k, t)، فرض کنید دنباله‌ی اعداد صحیحی که با فراخوانی RANDOM$(۱, n)$ حاصل می‌شود برای هر دو الگوریتم یکسان است.

الف) فرض کنید COMPACT-LIST-SEARCH(L, n, k)، منجر به t بار تکرار حلقه‌ی **while** در سطرهای ۲ تا ۸ شود. استدلال کنید که COMPACT-LIST-SEARCH(L, n, k)، همان جواب را برمی‌گرداند و تعداد کل تکرارهای دو حلقه‌ی **while** و **for** در الگوریتم COMPACT-LIST-SEARCH'(L, n, k, t) دست‌کم t است.

در فراخوانی COMPACT-LIST-SEARCH'(L, n, k, t)، فرض کنید X_t متغیری تصادفی و نمایانگر فاصله‌ای در لیست پیوندی باشد (در زنجیره‌ی اشاره‌گرهای $next$) که بین i و کلید k ی مورد نظر ما پس از t بار اجرای حلقه‌ی **for** در سطرهای ۲ تا ۷ می‌ماند.

ب) بحث کنید که زمان اجرای میانگین COMPACT-LIST-SEARCH'(L, n, k, t)، $\mathcal{O}(t + E[X_t])$ است.

پ) نشان دهید $E[X_t] \leq \sum_{r=۱}^{n} (\frac{۱-r}{n})^t$.

ت) نشان دهید $\sum_{r=۰}^{n-۱} r^t \leq \frac{n^{t+۱}}{(t+۱)}$.

ث) ثابت کنید $E[X_t] \leq \frac{n}{(t+۱)}$.

ج) نشان دهید که COMPACT-LIST-SEARCH'(L, n, k, t) در زمان میانگین $\mathcal{O}(t + \frac{n}{t})$ اجرا می‌شود.

چ) نتیجه بگیرید که COMPACT-LIST-SEARCH در زمان میانگین $\mathcal{O}(\sqrt{n})$ اجرا می‌شود.

ح) چرا در COMPACT-LIST-SEARCH فرض می‌کنیم تمام کلیدها متمایز هستند؟ بحث کنید که‌وقتی لیست می‌تواند مقادیر تکراری داشته باشد پرش‌های تصادفی لزوماً در زمان اجرای مجانبی تغییری ایجاد نمی‌کند.

۴-۴ درخت‌ها

درخت یک مدل مناسب برای نمایش بسیاری از مفاهیم، پدیده‌ها و رابطه‌ی بین آن‌هاست. مثلاً، درخت فامیلی، سلسله‌مراتب اداری، عبارت‌های ریاضی و مراحل مختلف یک بازی به‌خوبی با انواع درخت‌ها مدل می‌شوند. به‌صورت انتزاعی درخت‌ها گراف‌های خاصی هستند که در مورد ویژگی‌های آن‌ها نتایج نظری زیادی موجود است.

تعریف ۱. درخت آزاد: درخت یک گراف هم‌بند[۴۴] و بدون دور است که به آن «درخت آزاد[۴۵]» هم می‌گوییم. رأس‌های این گراف «گره[۴۶]» نامیده می‌شوند.

نکته‌ی ۴-۱ یک گراف درخت است اگر بین هردو رأس آن تنها یک مسیر وجود داشته باشد. چنین درختی با n گره دقیقاً دارای $n-1$ یال است.

لم ۴-۱ اگر E تعداد یال‌های یک درخت آزاد و V تعداد گره‌های آن باشد، داریم $E = V - 1$.

اثبات: (با استقرا) برای $V = 1$ بدیهی است که $E = 0$. برای $V = k$، حتماً گره‌ای با درجه‌ی ۱ وجود دارد (وگرنه دور داریم). این گره و یال متصل به آن‌را حذف می‌کنیم. درختی با $V - 1$ گره و $E - 1$ یال به‌دست می‌آید. طبق فرض استقرا داریم، $E - 1 = V - 1 - 1$ که حکم را ثابت می‌کند. □

تعریف ۲. درخت جهت‌دار (ریشه‌دار): یک «درخت جهت‌دار[۴۷]» مجموعه‌ای از عناصر است که رابطه‌ی «پدر-فرزندی[۴۸]» بین آن‌ها برقرار است، به‌طوری که هر عنصر بجز «ریشه» دقیقاً یک پدر داشته باشد.

به‌عبارت دیگر، اگر بر روی یال‌های یک درخت آزاد جهت قرار دهیم به‌طوری‌که درجه‌ی ورودی هر گره، بجز گره ریشه، ۱ باشد درخت جهت‌دار ایجاد شده است. جهت‌ها نشان‌دهنده‌ی رابطه‌ی «پدر-فرزندی» است.

تعریف ۳ (تعریف بازگشتی درخت ریشه‌دار): «درخت ریشه‌دار[۴۹]» مطابق زیر تعریف می‌شود:

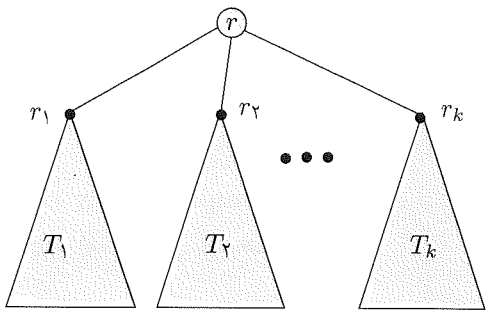
---

[۴۴] connected
[۴۵] free tree
[۴۶] node
[۴۷] directed tree
[۴۸] parent-child
[۴۹] rooted tree

الف) یک گره به‌تنهایی یک درخت ریشه‌دار است.

ب) اگر k درخت ریشه‌دار مستقل T_1 تا T_k به‌ترتیب با ریشه‌های r_1 تا r_k داشته باشیم، می‌توانیم یک درخت ریشه‌دار بزرگ‌تر به‌نام T با ریشه‌ی r بسازیم به‌طوری‌که r پدر r_1 تا r_k باشد. در این‌صورت T_1، T_2 تا T_k را «زیردرخت[۵۰]»های T می‌گوییم.

شکل ۴-۲۳ این تعریف را نشان می‌دهد.



شکل ۴-۲۳ یک درخت ریشه‌دار.

۴-۴-۱ تعریف‌های اولیه در درخت‌ها

ریشه (root) در درخت جهت‌دار: گره‌ای که دارای پدر نیست. این گره در هر درخت جهت‌دار یک‌تا است.

برگ (leaf): گره بدون فرزند.

برادر (sibling): گره‌هایی که یک پدر دارند، برادر هم هستند.

گره داخلی (interior node): گره غیر برگ.

ارتفاع گره v (height): طول بزرگ‌ترین مسیر از v به برگ w به‌طوری‌که w گره‌ای از زیردرختی به ریشه‌ی v باشد.

[۵۰] subtree

ارتفاع درخت: ارتفاع ریشه.

سطح (عمق) یک گره (depth, level): طول مسیری از ریشه‌ی درخت به آن گره.

درخت متوازن (balanced tree): درختی که سطح برگ‌های آن حداکثر یک واحد با هم اختلاف داشته باشد.

درخت کاملاً متوازن (completely balanced tree): درختی که سطح برگ‌های آن برابر است.

درخت k تایی (k-ary tree): درختی که بیشینه‌ی تعداد فرزندان یک گره k باشد.

درخت k تایی کامل (complete k-ary tree): درخت متوازنی که درآن تعداد فرزندان هر گره برابر k یا صفر (فقط برای برگ‌ها) باشد.

درخت مرتب (ordered tree): درختی است که ترتیب فرزندان هر گره مشخص است.

درخت برچسب‌دار (labeled tree): درختی است که هر گره آن یک برچسب دارد.

درخت دودویی (binary tree): درخت مرتبی است که هر عنصرآن حداکثر دارای دو فرزند به نام‌های فرزند چپ و راست باشد. اگر یک گره فقط یک فرزند داشته باشد باید مشخص شود که فرزند چپ است یا راست.

اولاد (نوادگان) یک گره v (descendents): گره‌های موجود در زیردرختی به ریشه‌ی v را اولاد (نوادگان) v می‌گوییم. با این تعریف هر گره اولاد خودش هم است.

اجداد (نیاکان) یک گره (ancestors): گره‌های موجود در مسیری از ریشه به گره v را اجداد (نیاکان) آن گره گوییم. بنابراین هر گره جزو اجداد خودش است.

اولاد واقعی (proper descendent): اولاد v بجز v را اولاد واقعی v می‌گوییم.

اجداد واقعی (proper ancestors): اجداد v بجز v را اجداد واقعی v می‌گوییم.

درخت پُر (full tree): درخت کامل و کاملاً متوازن.

جنگل (forest): تعدادی درخت!

مسئله‌ی ۴-۱ درختی با n گره که تعداد فرزندان هر گره صفر یا k باشد، می‌تواند چند برگ داشته باشد؟

حل: اگر b تعداد برگ‌ها و n تعداد کل گره‌ها باشد، تعداد یال‌ها برابر است با: $n-1=(n-b)*k$. پس $n-b=(n-1)/k$ و $b=n-(n-1)/k$ یا $b=[(k-1)n+1]/k$. یعنی باید $(k-1)n+1$ بر k بخش‌پذیر باشد.

۴-۴-۲ پیمایش درخت‌ها

فرض کنید درخت ریشه‌دار و مرتب T با ریشه‌ی r و k عدد زیر درخت T_1، T_2 تا T_k موجود است. می‌خواهیم با حرکت روی یال‌های درخت، همه‌ی گره‌های آن‌را هرکدام یک‌بار ملاقات کنیم. به این کار «پیمایش درخت[۵۱] T» می‌گوییم.

سه روش معمول برای پیمایش درخت عبارتند از:

پیمایش پیش‌ترتیب[۵۲]: ابتدا ریشه را ملاقات می‌کنیم، سپس زیردرخت‌ها را به‌ترتیب و با همین روش پیمایش می‌کنیم. یعنی:

$$Preorder(T) = r, Preorder(T_1), Preorder(T_2), \ldots, Preorder(T_k)$$

پیمایش میان‌ترتیب[۵۳]: نخست زیردرخت اول را با همین روش پیمایش می‌کنیم. سپس ریشه‌ی درخت را ملاقات می‌کنیم و آن‌گاه بقیه‌ی زیردرخت‌ها را به‌ترتیب و به‌صورت بازگشتی پیمایش می‌کنیم. یعنی:

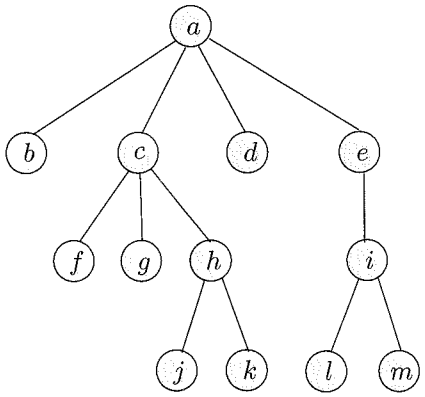
$$Inorder(T) = Inorder(T_1), r, Inorder(T_2), \ldots, Inorder(T_k)$$

پیمایش پس‌ترتیب[۵۴]: ابتدا زیردرخت‌ها را به‌ترتیب و با همین روش پیمایش و سپس ریشه را ملاقات می‌کنیم. به‌عبارت دیگر:

$$Postorder(T) = Postorder(T_1), Postorder(T_2), \ldots, Postorder(T_k), r$$

---

[۵۱] tree traversal
[۵۲] preorder
[۵۳] inorder
[۵۴] postorder



شکل ۴-۲۴ مثالی از یک درخت جهت‌دار و مرتب. جهت یال‌ها از بالا به پایین فرض می‌شود. همچنین ترتیب گره‌های برادر از سمت چپ به راست است.

مثلاً سه پیمایش فوق درخت شکل ۴-۲۴ به‌صورت زیر خواهد بود:

$Preorder(T) = a, b, c, f, g, h, j, k, d, e, i, l, m$
$Inorder(T) = b, a, f, c, g, j, h, k, d, l, i, m, e$
$Postorder(T) = b, f, g, j, k, h, c, d, l, m, i, e, a$

۴-۴-۳ درخت دودویی معادل

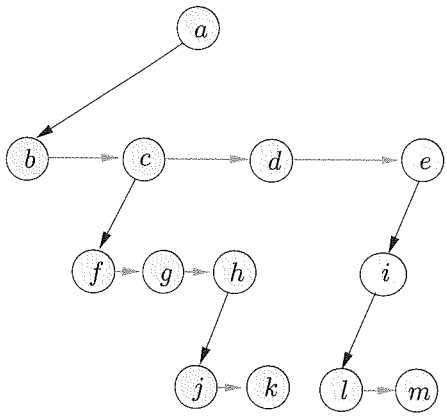
هر درخت را می‌توان به‌صورت یک درخت دودویی درآورد که به آن «درخت دودویی معادل[۵۵]» با درخت اصلی می‌گوییم. درخت دودویی معادل در پیاده‌سازی درخت‌های کلی به‌کار می‌رود.

الف) اولین (سمت چپ‌ترین) فرزند هر گره در درخت اصلی، فرزند چپ آن گره در درخت دودویی معادل است.

ب) برادر سمت راست هر گره در درخت اصلی، فرزند راست آن در درخت معادل است.

مثلاً شکل ۴-۲۵ درخت دودویی معادل درخت شکل ۴-۲۴ است.

[۵۵] equivalent binary tree



شکل ۴-۲۵ درخت دودویی معادلِ درخت شکل ۴-۲۴.

اگر درخت دودویی معادل را یک درخت دودویی مستقل (به‌نام T') در نظر بگیریم و آن‌را به‌همان روش‌های فوق پیمایش کنیم، ترتیب‌های زیر به‌دست می‌آیند.

$$Preorder(T') = a, b, c, f, g, h, i, j, d, e, k, l, m = Preorder(T)$$
$$Inorder(T') = b, f, g, i, j, h, c, d, l, m, k, e, a = Postorder(T)$$
$$Postorder(T') = j, i, h, g, f, m, l, k, e, d, c, b, a$$

۴-۴-۴ اعمال مختلف بر روی درخت

بر روی درخت T اعمال مختلفی را می‌توان انجام داد، ولی باید دست‌کم اعمال زیر در هر پیاده‌سازی فراهم شوند:

- CREATE(T): درخت تهی T با ویژگی‌های تعریف‌شده را ایجاد کن.
- ROOT(T): ریشه‌ی درخت T (یا تهی اگر T تهی باشد) را برگردان.
- PARENT(T, v): پدر گره v در درخت T (یا مقدار تهی) را برگردان.
- LEFT-MOST-CHILD(T, v): اولین فرزند گره v در درخت T (یا تهی) را برگردان.
- RIGHT-SIBLING(T, v): برادر سمت راست v در درخت T (یا تهی) را برگردان.
- SIZE(T): تعداد عناصر موجود در درخت را برگردان.
- ISEMPTY(T): مشخص کن که آیا درخت تهی است یا خیر.

- ELEMENT(T, n): برچسب عنصر در گره n را برگردان.

در صورتی‌که این اعمال پیاده‌سازی شده باشند، می‌توان رویه‌های پیچیده‌تری نوشت. مثلاً پیمایش‌های پیش‌ترتیب، میان‌ترتیب و پس‌ترتیب به‌صورت زیر خواهند بود. در این رویه‌ها پارامترهای (T, p) به این صورت در نظر گرفته می‌شود که پیمایش مورد نظر در زیردرختی به ریشه‌ی p در درخت T انجام می‌شود. این فراخوانی‌ها در ابتدا با پارامترهای $(T,$ ROOT$(T))$ انجام می‌شوند.

```
PREORDER (T, p)
 1  if p = null
 2     then return
 3  ELEMENT(T, p)
 4  p ← LEFT-MOST-CHILD(T, p)
 5  while p ≠ null
 6     do PREORDER(T, p)
 7        p ← RIGHT-SIBLING(T, p)

INORDER (T, p)
 1  if p = null
 2     then return
 3  n ← LEFT-MOST-CHILD(T, p)
 4  INORDER(T, n)
 5  ELEMENT(T, p)
 6  n ← RIGHT-SIBLING(T, n)
 7  while n ≠ null
 8     do INORDER (T, n)
 9        n ← RIGHT-SIBLING(T, n)
10

POSTORDER (T, p)
 1  if p = null
 2     then return
 3  n ← LEFT-MOST-CHILD(T, p)
 4  while n ≠ null
 5     do POSTORDER(T, n)
 6        n ← RIGHT-SIBLING(T, n)
 7  ELEMENT(T, p)
```

سه روش مختلف برای پیمایش یک درخت

همچنین می‌توان با فراخوانی CountNodes(T, Root(T)) تعداد گره‌های T و با فراخوانی NodeHeight(T, p) ارتفاع گره p در درخت T را به‌دست آورد. در رویه‌ی دوم، بدیهی است که ارتفاع p یک واحد بیش‌تر از بیشینه‌ی ارتفاع‌های فرزندان p است.

```
CountNodes (T, p)
    ▷ تعداد گره‌های موجود در درخت T با ریشه‌ی p را می‌شمارد
 1  if T = null
 2     then return 0
 3  count ← 1
 4  p ← Left-Most-Child(T, p)
 5  while p ≠ null
 6     do count ← count+ CountNodes(T, p)
 7        p ← Right-Sibling (T, p)

NodeHeight (T, p)
    ▷ ارتفاع گره p را در درخت T محاسبه می‌کند و آن‌را برمی‌گرداند
    ▷ از نحوه‌ی پیاده‌سازی درخت اطلاعی نداریم
 1  if isEmpty(T)
 2     then return -1
 3  height ← 0
 4  p ← Left-Most-Child(T, p)
 5  while p ≠ null
 6     do height ← max{height, NodeHeight (T, p)}
 7        p ← Right-Sibling(T, p)
 8  return height + 1
```

دو رویه بر روی درخت‌ها

یا مثلاً می‌توان پدر یک عنصر را با استفاده از اعمال دیگر به‌صورت زیر پیاده‌سازی کرد: Parent(T, r, p) پدر p را در زیردرختی به ریشه‌ی r در درخت T به‌دست می‌آورد. این رویه را از r آغاز می‌کند و در تمام زیردرخت‌های این ریشه، به‌نام q، و با فراخوانی بازگشتیِ Parent(T, q, p) به‌دنبال پدر p در آن زیردرخت می‌گردد، تا آن‌که یا آن‌را پیدا کند و یا یکی از این qها خودِ p باشد که در آن‌صورت r پدرِ p است.

```
PARENT (T, r, p)
    ▷ گره پدر یک گره p را در درخت T به ریشه‌ی r برمی‌گرداند
    ▷ در صورتی که عنصر در زیردرخت نباشد برمی‌گرداند
    ▷ فرض می‌کنیم که اشاره‌گر «پدر» وجود ندارد
 1  if p = r
 2     then return null
 3  q ← Left-Most-Child(T, r)
 4  while q ≠ null
 5     do if p = q
 6           then return r
 7        s ← PARENT(T, q, p)
 8        if s ≠ null
 9           then return s
10        q ← RIGHT-SIBLING(T, q)
11  return null
```

۴-۴-۵ پیاده‌سازی درخت‌ها

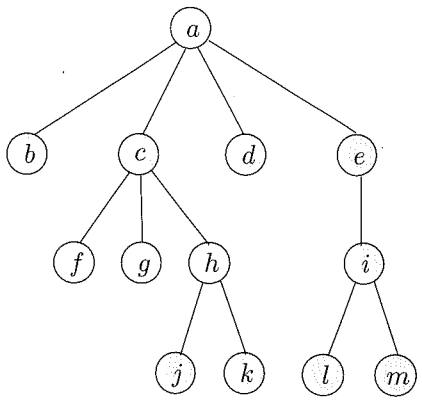
آن‌چه در بالا بیان شد، با این فرض بود که درخت پیاده‌سازی شده باشد و اعمال پایه‌ای در اختیار کاربر باشد. در این بخش نشان می‌دهیم که روش‌های متفاوتی برای پیاده‌سازی درخت‌ها وجود دارد و در نتیجه اعمال پایه‌ای آن‌ها هم به‌صورت مختلف پیاده‌سازی می‌شوند.

پیاده‌سازی درخت با آرایه‌ها

در این پیاده‌سازی، عناصر درخت را به‌صورت زیر در درایه‌های دو آرایه به‌نام‌های *element* و *parent* قرار می‌دهیم. *element*[i] برچسب و *parent*[i] شماره‌ی درایه (اندیس) پدر عنصر iام در درخت است. درایه‌ی ۱ حاوی ریشه است. اگر درخت مرتب باشد، فرزندان هر گره واقع در اندیس a به ترتیب در درایه‌هایی با اندیس بالاتر از i قرار می‌گیرند، به‌طوری که ترتیب فرزندان در این روش هم حفظ شود. توجه کنید که در این پیاده‌سازی

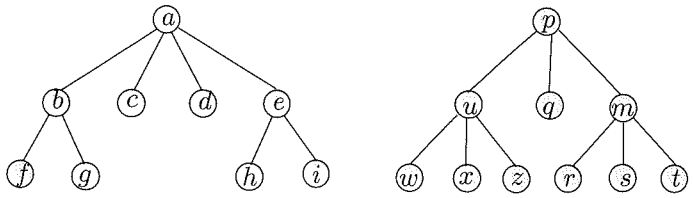
نیازی نیست که اندیس‌های فرزندان یک گره پشت سر هم باشند.

شکل ۴-۲۶ یک درخت مرتب و پیاده‌سازی آن را با استفاده از آرایه نشان می‌دهد. روشن است که با این روش می‌توان بیش از یک درخت را در یک آرایه پیاده‌سازی کرد. شکل ۴-۲۷ مثالی از پیاده‌سازی دو درخت را در یک آرایه نشان می‌دهد.



	1	2	3	4	5	6	7	8	9	10	11	12	13
key	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*
parent	۰	۱	۱	۱	۱	۲	۲	۲	۵	۸	۸	۹	۹

شکل ۴-۲۶ یک درخت مرتب و پیاده‌سازی آن با آرایه.



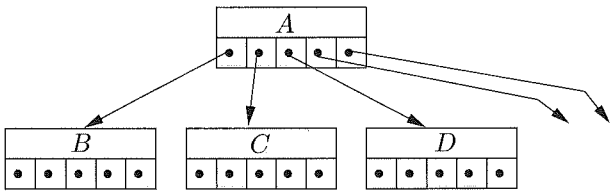
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	*b*	*c*	*d*	*e*	*f*	*p*	*u*	*g*	*h*	*i*	*q*	*m*	*r*	*s*	*t*	*w*	*x*	*z*
۰	۱	۱	۱	۱	۲	۰	۷	۲	۵	۵	۷	۷	۱۳	۱۳	۱۳	۸	۸	۸

شکل ۴-۲۷ پیاده‌سازی دو درخت در یک آرایه.

همچنین می‌توان دید که اعمال گفته‌شده را می‌توان به‌راحتی (البته با هزینه‌ای بیش‌تر از $O(1)$ برای برخی از اعمال) پیاده‌سازی کرد.

پیاده‌سازی درخت با اشاره‌گر

در این پیاده‌سازی برای هر گره یک رکورد با مؤلفه‌ی *label* برای برچسب آن گره و یک آرایه‌ی $Child[1 \cdots max\text{-}child]$ در نظر می‌گیریم. درایه‌ی $child[i]$ به رکورد فرزند i ام آن گره اشاره می‌کند (شکل ۴–۲۸). مقدار *max-child* حداکثر تعداد فرزندان هر گره در درخت است. البته، چنان‌چه بیان شد، هر پیاده‌سازی با اشاره‌گر را می‌توان با اشاره‌گرهای اندیسی هم پیاده‌سازی کرد.



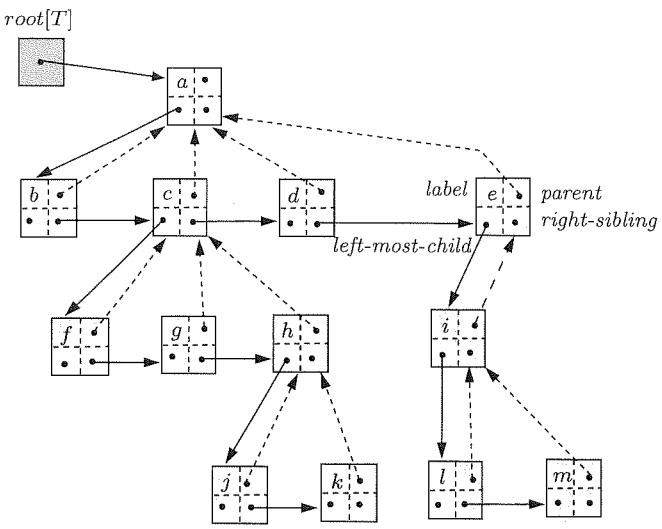
شکل ۴–۲۸ پیاده‌سازی درخت با اشاره‌گر در حالت کلی.

بدیهی است که این پیاده‌سازی برای حالتی که تعداد فرزندان گره‌ها خیلی متفاوت باشند، مناسب نیست و موجب اتلاف حافظه می‌شود. برای رفع این مشکل از پیاده‌سازی درخت دودویی معادل با اشاره‌گر استفاده می‌کنیم.

پیاده‌سازی درخت دودویی معادل

در این پیاده‌سازی، درخت اصلی T مرتب فرض می‌شود. هر گره، یک مؤلفه‌ی *label* یا برچسب دارد و سه اشاره‌گر *left-most-child*، *right-sibling* و *parent*. برای هر گره v، اشاره‌گر $left\text{-}most\text{-}child[v]$ به اولین فرزندِ v در T، $right\text{-}sibling[v]$ به برادر سمت راستِ v در T و $parent[v]$ به پدر v در T اشاره می‌کند. شکل ۴–۲۹ این پیاده‌سازی را برای درخت شکل ۴–۲۴ نشان می‌دهد که اشاره‌گر به پدر با نقطه‌چین نشان داده شده است.

در این پیاده‌سازی دست‌رسی به گره‌ها کمی سخت‌تر است، ولی تعداد زیاد فرزندان برای یک گره مشکل حافظه ایجاد نمی‌کند.



شکل ۴-۲۹ پیاده‌سازی درخت مرتب شکل ۴-۲۴ به روش درخت دودویی معادل.

پیاده‌سازی اعمال مختلف

اگر از پیاده‌سازی درخت دودویی معادل استفاده کنیم، اَعمال گفته‌شده در بخش ۴-۴-۴ به‌سادگی و در $O(1)$ انجام می‌شوند. برای ایجاد یک درخت T که برچسب ریشه‌ی آن x و دو زیر درخت T_1 و T_2 به‌ترتیب زیردرخت‌های اول و دوم آن باشد، می‌توان از CREATE2 استفاده کرد.

```
CREATE2 (x, T1, T2)
    ▷ درختی به ریشه‌ای که برچسب آن x است ایجاد می‌کند
    ▷ که زیر درخت‌های T1 و T2 به‌ترتیب زیردرخت‌های اول و دوم آن باشند
    ▷ فرض: T1 تهی نیست و زیردرخت‌ها به‌درستی با همین روش پیاده‌سازی شده‌اند
    r ← ALLOCATE-NODE (x, T1, null)
1   parent[ROOT(T1)] ← r
2   parent[ROOT(T2)] ← r
3   right-sibling[ROOT(T1)] ← ROOT(T2)
4   return r
```

این رویه را باید برای تعداد متفاوتی فرزند هم نوشت تا کامل شود. مثلاً برای ۳ فرزند باید CREATE3 را فراخواند.

```
CREATE3 (x, T1, T2, T3)
    ▷ همان کار CREATE2 را با سه زیر درخت T1، T2 و T3 انجام می‌دهد ◁
    r ← ALLOCATE-NODE (x, T1, null)
1   parent[ROOT(T1)] ← r
2   parent[ROOT(T2)] ← r
3   parent[ROOT(T3)] ← r
4   right-sibling[ROOT(T1)] ← ROOT(T2)
5   right-sibling[ROOT(T2)] ← ROOT(T3)
6   return r
```

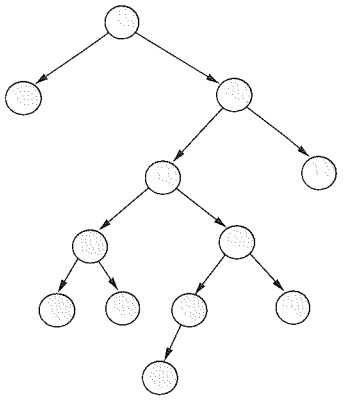
در مقایسه با روش‌های پیاده‌سازی گفته‌شده، این بهترین روش برای پیاده‌سازی درخت‌های کلی و مرتب است.

۴-۴-۶ درخت دودویی

درخت دودویی حالت خاصی از درخت‌های مرتب و کلی است که کاربردهای فراوان دارد. هر گره در درخت دودویی دو فرزند چپ و راست دارد که ممکن است یک یا هر دو فرزندش تهی باشند. درخت عبارت (بخش ۴-۴-۷) و درخت تصمیم (بخش ۶-۱) نمونه‌هایی از کاربردهای مهم درخت دودویی هستند. بسیاری از درخت‌ها که گره‌های آن‌ها حداکثر ۲ فرزند دارند هم دودویی فرض می‌شوند. شکل ۴-۳۰ یک درخت دودویی را نشان می‌دهد.

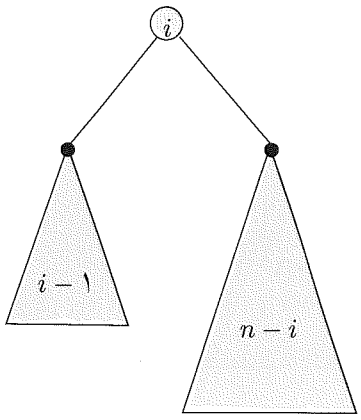
تعداد درخت‌های دودویی

می‌خواهیم تعداد درخت‌های دودویی متفاوت را که با n گره می‌توان ساخت به‌دست آوریم. فرض کنید که گره‌های هر کدام از این درخت‌ها را به‌روش میان‌ترتیب از ۱ تا n



شکل ۴-۳۰ یک درخت دودویی.

شماره‌گذاری کرده‌ایم. با این شماره‌گذاری، فرض کنید که گره شماره‌ی $1 \leq i \leq n$ ریشه‌ی درخت باشد. در آن صورت زیردرخت چپ ریشه حتماً باید $i - 1$ گره و زیردرخت راست آن حتماً باید $n - i$ گره داشته باشد (شکل ۴-۳۱ را ببینید). حال اگر $T(n)$ تعداد درخت‌های دودویی با n گره باشد، تعداد زیردرخت‌های چپ و راست ریشه به‌ترتیب برابر $T(i-1)$ و $T(n-i)$ خواهند بود. در آن‌صورت اگر گره‌ی i ریشه باشد، حاصل‌ضرب $T(i)T(n-i)$ برابر تعداد کل درخت‌هاست. برای در نظر گرفتن همه‌ی حالت‌های ریشه،



شکل ۴-۳۱ شمارش تعداد درخت‌های دودویی با n گره.

باید این مقدار را برای همه‌ی مقادیر i جمع بزنیم که رابطه‌ی بازگشتی ۴-۴ به‌دست می‌آید.

$$T(n) = \begin{cases} T(۰) = T(۱) = ۱ \\ T(n) = \sum_{i=۱}^{n} T(i-۱)T(n-i), \quad i > ۱ \end{cases} \tag{۴-۴}$$

جواب این رابطه‌ی بازگشتی

$$T(n) = \frac{۱}{n+۱}\binom{۲n}{n}$$

است که همان «عدد n ام کاتالان[۵۶]» است.

پیاده‌سازی درخت‌های دودویی

در پیاده‌سازی درخت دودویی از اشاره‌گر استفاده می‌کنیم. هر گره دو مؤلفه‌ی *left* و *right* دارد که به‌ترتیب به فرزندهای چپ و راست آن گره اشاره می‌کنند، و ممکن است برخی از آن‌ها تهی باشند. مانند قبل، معمولاً هر گره مؤلفه‌ی *parent* هم دارد که به پدر آن گره اشاره می‌کند. بدیهی است که پدر ریشه تهی است.

با این پیاده‌سازی پیمایش پیش‌ترتیب یک درخت دودویی T به ریشه‌ی r به‌صورت زیر خواهد بود:

```
PREORDER (T, r)
    ▷ پیمایش پیش‌ترتیب درخت T به ریشه‌ی r
1   if r = null
2      then return
3   meet element(T, r)
4   PREORDER(T, left[r])
5   PREORDER(T, right[r])
```

[۵۶] nth Catalan number: این عدد منتسب به یوجین چارلز کاتالان (۱۸۹۴-۱۸۱۴) ریاضی‌دان مشهور بلژیکی است.

۴-۴-۷ درخت‌های عبارت

یکی از کاربردهای مهم درخت دودویی «درخت‌های عبارت[۵۷]» است که برای عبارت‌های ریاضی به‌کار می‌رود.

نگارش‌های مختلف عبارت ریاضی

هر عبارت شامل تعدادی متغیر یا ثابت است که به آن‌ها «عملوند[۵۸]» می‌گوییم. یک «عمل‌گر[۵۹]» برروی عمل‌وندها یا زیرعبارت‌ها عمل می‌کند. یک عمل‌گر ممکن است دودویی باشد، مانند ضرب و جمع، و بر روی دو عمل‌وند اجرا شند، یا مانند تغییر علامت و لگاریتم، یگانی[۶۰] باشند و تنها بر روی یک عمل‌وند اِعمال شود. البته عمل‌گرهایی هم داریم که بر روی بیش از دو عمل‌وند تعریف می‌شوند، ولی ما در این‌جا فقط عمل‌گرهای دودویی و یگانی را در نظر می‌گیریم. چنان‌چه می‌دانیم، در یک عبارت ریاضی، عمل‌گرها نسبت به هم اولویت[۶۱]های مشخص و تعریف‌شده‌ای دارند. البته می‌توانیم با قرار دادن پرانتزهایی، اولویت‌های دیگری را اِعمال کنیم. عمل‌گرهایی که اولویت بالاتری دارند زودتر از بقیه اجرا می‌شوند.

در این بخش، ابتدا نگارش‌های مختلف یک عبارت ریاضی را توضیح می‌دهیم و سپس الگوریتم‌های تبدیل برخی نگارش‌ها را به‌هم ارائه می‌کنیم.

نگارش میان‌وندی عبارت

همه با نگارش میان‌وندی[۶۲] عبارت آشنا هستند. مثلاً

```
a+(b-c*d)^-f^g^(h/¬i*k).
```
(۵-۴)

در این نگارش با قرار دادن پرانتز، می‌توان اولویت برخی از عمل‌گرها را تغییر داد. به‌طور کلی، قاعده‌ای که با آن آشنا هستیم آن است که عبارتی که در عمق بیش‌تری از پرانتزها قرار گرفته است زودتر اجرا می‌شود. در یک عبارت کوچک‌تر و بدون پرانتز، ابتدا عمل‌گرهای

۵۷ expression trees
۵۸ operands
۵۹ operator
۶۰ uniary
۶۱ priority
۶۲ infix notation

یگانی اجرا می‌شوند و ترتیب آن‌ها مهم نیست. بین عمل‌گرهای دودویی اولویت‌ها به‌ترتیب زیر هستند:

۱. توان،

۲. ضرب و تقسیم (با اولویت یک‌سان)، و

۳. جمع و تفریق (با اولویت یک‌سان).

اگر عمل‌گرهای ضرب و تقسیم (یا جمع و تفریق) در عمق برابری از پرانتزها باشند، آن‌که سمت چپ است اول اجرا می‌شود. ولی در مورد عمل‌گر توان این‌چنین نیست؛ عمل‌گر توانِ سمت راست، زودتر اجرا می‌شود. یعنی a^b^c برابر a^{b^c} است و نه $(a^b)^c = a^{bc}$.

نگارش میان‌وندی با پرانتز کامل

در نگارش «میان‌وندی با پرانتز کامل[۶۳]» اولویت عمل‌گرهای یک عبارت ریاضی، با گذاردن پرانتز دور همه‌ی آن‌ها مشخص می‌شود. ما برای تعریف دقیق این نگارش عبارت ریاضی، از قاعده‌ای به‌نام «گرامر[۶۴]» استفاده می‌کنیم که در زیر می‌آید:

$$E \rightarrow (E\ \langle\beta\rangle\ E) \qquad (۶-۴)$$

$$\rightarrow (\ \langle\alpha\rangle E\ ) \qquad (۷-۴)$$

$$\rightarrow \langle \texttt{operand} \rangle \qquad (۸-۴)$$

$$\langle\alpha\rangle \rightarrow \neg \mid ! \mid Sin \mid Log \mid \cdots \quad \texttt{unary operators} \qquad (۹-۴)$$

$$\langle\beta\rangle \rightarrow - \mid + \mid * \mid / \mid \hat{} \mid \wedge \mid \vee \mid \cdots \quad \texttt{binary operators} \qquad (۱۰-۴)$$

این به‌معنی آن است که یک عبارت از این نگارش به یکی از این سه شکل است: یا شامل یک پرانتز باز، یک زیرعبارت E با همین نگارش، یک عمل‌گر دودویی β، یک زیرعبارت دیگر با همین نگارش و یک پرانتز بسته (قاعده‌ی ۴-۶) است، یا شامل یک پرانتز باز، یک عمل‌گر یگانی، یک زیرعبارت با همین نگارش و یک پرانتز بسته (قاعده‌ی ۴-۷) است، یا فقط یک عمل‌وند (قاعده‌ی ۴-۸). یک عمل‌گر دودویی β بر روی دو عمل‌وند اجرا

[۶۳] infix with complete paranthesis
[۶۴] grammer

می‌شود که ترتیب آن‌ها، مثلاً برای عمل تقسیم (/) یا تفریق (−) مهم است. در این‌جا عمل توان را با عمل‌گر ^ نشان داده‌ایم. مثلاً

```
((a+((b-(c*d))^e)-(f^(g^((h/(¬i))*k))))
```
(۱۱-۴)

نگارش میان‌وندی با پرانتز کامل برای عبارت میان‌وندی ۴-۵ است.

نگارش پس‌وندی عبارت

در نگارش پس‌وندی[۶۵] عمل‌گرها (که ممکن است خود عبارت‌های بزرگی باشند که با همین نگارش نوشته می‌شوند) بلافاصله و به‌ترتیب پس از عمل‌وند نوشته می‌شوند. گرامر این گونه از عبارت‌های ریاضی با عمل‌گرهای دودویی و یگانی به این صورت است:

$$\begin{aligned} E &\rightarrow EE\langle\beta\rangle \\ &\rightarrow E\langle\alpha\rangle \\ &\rightarrow \langle\texttt{operand}\rangle \end{aligned} \qquad (۱۲-۴)$$

مثلاً `abcd*-e^+fghi¬/k*^^-` نگارش پس‌وندی برای همان عبارت میان‌وندی ۴-۵ است. توجه کنید که ترتیب عمل‌گرها در این دو عبارت یک‌سان است. همچنین، در نگارش پس‌وندی یک عبارت نیازی به پرانتزگذاری نیست و ترتیب عمل‌گرها از جایگاه آن‌ها مشخص می‌شود.

نگارش پیش‌وندی عبارت

در نگارش پیش‌وندی[۶۶]، عمل‌وندها (که خود به همین نگارش نوشته می‌شوند) به‌ترتیب قبل از یک عمل‌گر نوشته می‌شوند.

$$\begin{aligned} E &\rightarrow \langle\beta\rangle EE \\ &\rightarrow \langle\alpha\rangle E \\ &\rightarrow \langle\texttt{operand}\rangle \end{aligned} \qquad (۱۳-۴)$$

مثلاً `-+a^-b*cde^f^g*/h¬ik` نگارش پیش‌وندی همان میان‌وندی ۴-۵ است. در این نگارش نیز، مانند نگارش پس‌وندی، پرانتز ظاهر نمی‌شود.

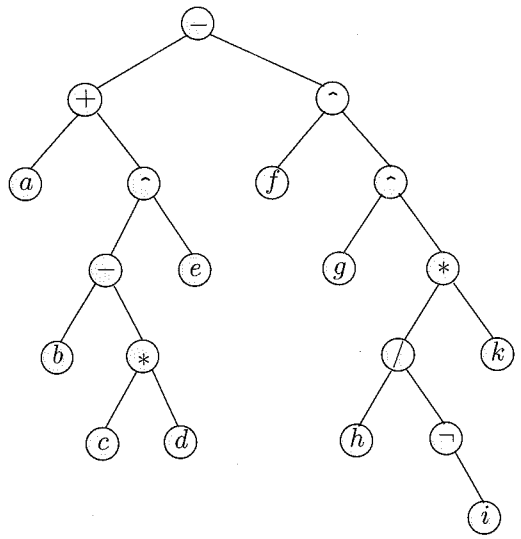
---

[۶۵] postfix
[۶۶] prefix

جدول ۴-۱ نگارش‌های مختلف عبارت ریاضیِ ۴-۵.

نام روش	نگارش عبارت
میان‌وندی	a+(b-c*d)^e-f^g^(h/¬i*k)
میان‌وندی با پرانتز کامل	((a+((b-(c*d))^e))-(f^(g^((h/(¬i))*k))))
پس‌وندی	abcd*-e^+fghi¬/k*^^-
پیش‌وندی	-+a^-b*cde^f^g*/h¬ik

در جدول ۴-۱ مثال‌هایی را که ذکر کردیم یک‌جا آورده‌ایم. شکل ۴-۳۲ هم درخت عبارتِ ۴-۵ را نشان می‌دهد.



شکل ۴-۳۲ درخت عبارت برای a+(b-c*d)^-f^g^(h/¬i*k).

روشن است که آخرین عمل‌گری که اجرا می‌شود در ریشه‌ی این درخت قرار می‌گیرد که در این مثال عمل‌گر تفریق است. ریشه عملاً عبارت اصلی را به دو زیرعبارت تقسیم می‌کند که درخت‌های عبارت هریک به صورت بازگشتی ساخته می‌شود. توجه کنید که اولویت عمل‌گرها کاملاً رعایت شده است. همچنین به‌دلیل آن‌که درخت دودویی است برای عمل‌گرهای یگانی (مانند ¬ در این مثال)، عمل‌وند مربوط به آن به‌طور قراردادی،

به‌عنوان فرزند راست آن در نظر گرفته می‌شود.

به‌سهولت می‌توان از یک درخت عبارت، نگارش کاملاً پرانتزی آن عبارت را با پیمایش میان‌ترتیب درخت و قراردادن پرانتزها تولید کرد. همچنین روشن است که پیمایش پیش‌ترتیب یک درخت عبارت، نگارش پیش‌وندی و پیمایش پس‌ترتیب آن نگارش پس‌وندی آن‌را تولید می‌کند.

۴-۴-۸ تبدیل نگارش‌های مختلف عبارت به هم

در این بخش نشان می‌دهیم که چگونه می‌توان یک عبارت را که به یک روش نگارش نوشته شده است، با روش‌های دیگر نگارش هم نوشت.

تبدیل نگارش میان‌وندی به درخت عبارت

به‌سادگی می‌توان نگارش میان‌وندی با پرانتز کامل را به درخت عبارت تبدیل کرد و یا به‌صورت مستقیم یا غیرمستقیم با روش‌های دیگر نوشت. برای نگارش غیردرختی عبارت، آرایه‌هایی به‌نام‌های *infix*، *par-infix*، *postfix* و *prefix* در نظر می‌گیریم که به‌ترتیب عبارت‌های میان‌وندی، میان‌وندی با پرانتز کامل، پس‌وندی و پیش‌وندی را نشان می‌دهند. هر درایه‌ی این آرایه‌ها یک عمل‌وند، یک عمل‌گر تک‌حرفی یا پرانتز است.

رویه‌ی P-INFIX-TO-TREE (l, r) با فرض آن‌که از اندیس l تا r در آرایه‌ی *par-infix* یک عبارت میان‌وندی با پرانتز کامل (درست مطابق تعریف ۴-۶) قرار دارد، درخت عبارت آن‌را می‌سازد. این رویه، ابتدا حالت‌های خاص را بررسی می‌کند، بعد حالتی که عمل‌گر این عبارت یگانی باشد و سپس حالت کلی با عمل‌گر دودویی را به‌طور بازگشتی می‌سازد. رویه‌ی ALLOCATE-NODE$(x, left, right)$ یک گره از درخت دودویی با برچسب x می‌سازد که اشاره‌گرهای چپ و راست آن به‌ترتیب $left$ و $right$ هستند و اشاره‌گر به گره ایجادشده را برمی‌گرداند. اگر گره پدر داشته باشد، پارامتر چهارم هم به این رویه اضافه می‌شود که اشاره‌گر به پدرِ این گره است.

با فرض آن‌که یک عمل‌وند نیز به‌تنهایی یک عبارت ریاضی است، در سطر ۳ اگر $l = r$ باشد، یک گره با فرزندان تهی ایجاد می‌شود. در سطر ۵، طبق فرض، حتماً در درایه‌ی *par-infix*$[l]$ پرانتز باز و در *par-infix*$[r]$ پرانتز بسته قرار دارد. اگر *par-infix*$[l+1]$ حاوی یک عمل‌گر یگانی باشد، در آن‌صورت عبارت از نوع αE است که درخت عبارت

معادل آن یک ریشه با برچسب α دارد که زیردرخت چپ آن تهی و E زیردرخت راست آن است. این درخت با فراخوانی سطر ۶ ساخته می‌شود. در حالت کلی که عبارت از نوع $E_1 \beta E_2$ باشد، ابتدا در سطر ۷ اندیس انتهایی عبارت E_1 (یا k) با فراخوانی $k \leftarrow \text{FINDMATCH}(l+1)$ به‌دست می‌آید. پس E_1 از اندیس $l+1$ تا k و E_2 از اندیس $k+2$ تا $r-1$ در آرایه قرار دارد. عمل‌گر β هم در درایه‌ی $k+1$ هست. بنابراین، فراخوانی سطر ۸ این درخت را به‌صورت بازگشتی می‌سازد. رویه‌ی FINDMATCH را هم می‌توان به‌راحتی با شمارش پرانتزهای باز و بسته نوشت.

```
P-INFIX-TO-TREE (l, r)
    ▷ یک عبارت میان‌وندی با پرانتز کامل که از اندیس l تا r در آرایه‌ی par-infix قرار دارد را
      به روش بازگشتی به درخت عبارت معادل آن تبدیل می‌کند
1   if l > r
2     then return null
3   if l = r
4     then return ALLOCATE-NODE (par-infix[l], null, null)
5   if par-infix[l + 1] in an unary operator
6     then return ALLOCATE-NODE (par-infix[l + 1],null,
                    P-Infix-to-Tree(l + 2, r − 1))
7   k ← FINDMATCH(l + 1)
8   return ALLOCATE-NODE (par-infix[k + 1],
          P-Infix-to-Tree(l + 1, k), P-INFIX-TO-TREE(k + 2, r − 1))
```

اگر n طول نگارش میان‌وندی باشد، زمان اجرای این رویه در بدترین حالت از $\mathcal{O}(n^2)$ است. این زمان اجرا خیلی کند است. رویه‌ی غیربازگشتی که در ادامه ارائه خواهیم داد مسئله را در زمان خطی حل می‌کند.

روش غیر بازگشتی برای تبدیل نگارش میان‌وندی به درخت عبارت

رویه‌ی غیربازگشتی شامل یک حلقه‌ی **for** $i \leftarrow 1$ **to** n است که i اندیس در آرایه‌ی *par-infix* و n طول این آرایه است. برای طراحی این رویه، ویژگی مستقل از حلقه‌ی این حلقه را به این صورت تعریف می‌کنیم: در ابتدای حلقه‌ی **for**، و به‌ازای یک مقدار i، اشاره‌گری به‌نام p به یک گره از درخت عبارتی که تاکنون ساخته‌شده به نام a اشاره می‌کند.

a ریشه‌ی یک زیردرخت مربوط به یک عبارت میان‌وندی با پرانتز کامل به نام E است که از اندیس i در آرایه‌ی *par-infix* آغاز شده است. این زیردرخت در ابتدا تنها همان گره a را دارد، اما در ادامه‌ی کار که i به‌جلو می‌رود، بخش‌هایی از E ساخته شده و p هم بر روی عناصر این زیردرخت حرکت می‌کند و مجدداً به گره a برمی‌گردد. این زمانی است که زیردرخت E به‌طور کامل ساخته شده است و i به آخرین اندیس این عبارت اشاره می‌کند. سپس p به پدر a برمی‌گردد، مقدار i برابر اندیس عنصری شده و دور دیگری از اجرای حلقه با این ویژگی آغاز می‌شود. در ابتدا $i = ۱$ و p ریشه‌ی درخت عبارت اصلی است. البته فرض می‌کنیم که عبارت میان‌وندی دست‌کم یک عنصر دارد و این عبارت دقیقاً از گرامر ۴-۶ تبعیت می‌کند.

رویه‌ی NR-P-INFIX-TO-TREE این کار را انجام می‌دهد. در ادامه، توضیحات بیش‌تر ارائه خواهد شد.

```
NR-P-INFIX-TO-TREE ()
    ▷ الگوریتم غیربازگشتی خطی برای تبدیل عبارت میان‌وندی به درخت
    ▷ فرض می‌شود که گره‌های درخت، اشاره‌گر به پدر دارند ▷
 1  CREATE-TREE (T)
 2  p ← ROOT (T)
 3  for i = 1 to length[par-infix]
 4    do switch
 5          case par-infix[i] = '('
 6            do if par-infix[i + 1] ≠ unary operator
 7                  then left[p] ← ALLOCATE-NODE (null, null, p)
 8                       p ← left[p]
 9          case par-infix[i] = ')'
10            do p ← parent[p]
11          case par-infix[i] is an operand
12            do label[p] ← infix[i]
13               p ← parent[p]
14          case par-infix[i] is a binary or unary operator
15            do label[p] ← par-infix[i]
16               right[p] ← ALLOCATE-NODE (null, null, p)
17               p ← right[p]
```

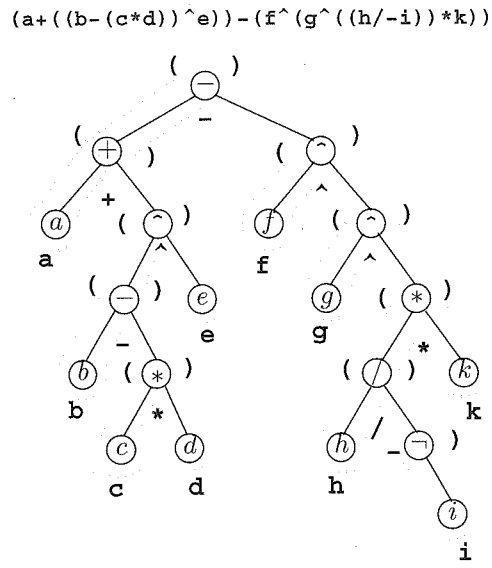
برای این‌که ببینیم ویژگی مستقل از حلقه‌ی ادعایی ما همیشه درست است به حالات مختلف عبارت E دقت می‌کنیم. اگر E فقط یک عمل‌وند باشد، ادعای ما درست است.

اگر E با یک '(' آغاز شود، بسته به نوع عمل‌گرِ استفاده شده (دودویی یا یگانی)، یا از نوع $(E_1 \beta E_2)$ است و یا از نوع (αE_3). در حالت اول، گره ریشه‌ی E_1 ایجاد می‌شود و p برابر آن خواهد شد و مطابق ویژگی مستقل از حلقه، و براساس استقرا، درخت E_1 به‌درستی ساخته می‌شود و در انتها p به ریشه‌ی E و i به β اشاره می‌کند. اگر عبارت از نوع دوم باشد، زیردرخت چپ E تهی است و E_3 زیردرخت راست آن خواهد بود.

اگر اندیس i یک عمل‌گر دودویی باشد، چنان‌چه گفتیم حتماً E_1 ساخته شده است. پس برچسب گره p برابر β و p به ریشه‌ی E_2 که در سطر ۷ ساخته می‌شود خواهد رفت. اگر عمل‌گر یگانی باشد هم همین‌کار را انجام می‌دهیم با این فرض که E_1 برابر **null** است.

اگر اندیس i برابر ')' باشد، یعنی یک زیرعبارت پایان یافته است. در این حالت p به پدرش منتقل شده و کار ادامه پیدا می‌کند. توجه کنید که شرط پایانی ویژگی حلقه در این‌جا مراعات شده است.

الگوریتم بسیار کارا و از $\mathcal{O}(n)$ است و بدون پشته و بازگشت درخت عبارت را می‌سازد. شکل ۴-۳۳ مراحل مختلف اجرای این رویه را برای یک عبارت نشان می‌دهد. در این شکل مسیر حرکت نقطه‌چین است و نویسه‌ای که در هر گره می‌بینند (چه در مسیر رفت و چه برگشت) نشان داده شده است.



شکل ۴-۳۳ مراحل اجرای رویه‌ی NR-P-INFIX-TO-TREE.

تبدیل عبارت میان‌وندی در حالت کلی به نگارش پس‌وندی

در این تبدیل اولویت عمل‌گرها را هم باید در نظر بگیریم، چرا که عبارت میان‌وندیِ ورودی لزوماً پرانتزی کامل نیست و ترتیب اجرای عمل‌گرها را باید از اولویت آن‌ها تعیین کنیم.

الگوریتم نویسه‌های عبارت میان‌وندی داده‌شده را یک‌به‌یک می‌خواند و با استفاده از یک پشته که ترتیب انجام عمل‌گرها را براساس اولویت‌های آن‌ها مشخص می‌کند، در خروجی عبارت پس‌وندی را تولید می‌کند. جدول ۴-۲ مراحل مختلف این الگوریتم را بر روی یک مثال نشان می‌دهد.

جدول ۴-۲ مراحل مختلف تبدیل یک عبارت میان‌وندی به نگارش پس‌وندی معادل.

نویسه‌های ورودی ←		→ پشته	خروجی
a	+(b-c*d)^e-f^g^(h/¬i*k)		
+	(b-c*d)^e-f^g^(h/¬i*k)		a
(	b-c*d)^e-f^g^(h/¬i*k)	+	a
b	-c*d)^e-f^g^(h/¬i*k)	+(	a
-	c*d)^e-f^g^(h/¬i*k)	+(	ab
c	*d)^e-f^g^(h/¬i*k)	+(-	ab
*	d)^e-f^g^(h/¬i*k)	+(-	abc
d	)^e-f^g^(h/¬i*k)	+(-*	abc
)	^e-f^g^(h/¬i*k)	+(-*	abcd
^	e-f^g^(h/¬i*k)	+	abcd*-
e	-f^g^(h/¬i*k)	+^	abcd*-
-	f^g^(h/¬i*k)	+^	abcd*-e
f	^g^(h/¬i*k)	-	abcd*-e^+
^	g^(h/¬i*k)	-	abcd*-e^+f
g	^(h/¬i*k)	-^	abcd*-e^+f
^	(h/¬i*k)	-^	abcd*-e^+fg
(	h/¬i*k)	-^^	abcd*-e^+fg
h	/¬i*k)	-^^(	abcd*-e^+fg
/	¬i*k)	-^^(	abcd*-e^+fgh
¬	i*k)	-^^(/	abcd*-e^+fgh
i	*k)	-^^(/¬	abcd*-e^+fgh
*	k)	-^^(/¬	abcd*-e^+fghi
k	)	-^^(*	abcd*-e^+fghi¬/
)		-^^(*	abcd*-e^+fghi¬/k
		-^^	abcd*-e^+fghi¬/k*
			abcd*-e^+fghi¬/k*^^-

الگوریتم با خواندن یک عمل‌وند مستقیماً آن‌را در خروجی می‌نویسد و در صورت خواندن یک عمل‌گر (یا پرانتز باز) پس از انجام پردازشی بر روی عناصر بالای پشته آن‌را وارد پشته می‌کند. این پردازش مهم‌ترین بخش این الگوریتم است که اولویت‌ها را اِعمال می‌کند و نتیجه‌ی آن این می‌شود که عناصر موجود در پشته به‌ترتیب صعودیِ اولویت‌هایشان قرار می‌گیرند. البته این نکته را باید اضافه کرد که پرانتز باز در پشته نشان‌دهنده‌ی شروع یک دنباله‌ی جدید از عمل‌گرهاست که به‌ترتیب اولویت‌هایشان مرتب هستند. پشته در واقع حاوی عمل‌گرهایی است که تا کنون دیده‌ایم ولی هنوز نمی‌توانیم تصمیم بگیریم که کدام‌یک را زودتر از بقیه اعمال کنیم.

اگر عمل‌گری را که خوانده‌ایم x بنامیم و عمل‌گر بالای پشته را $y = \text{TOP}(S)$ (شامل پرانتز باز)، الگوریتم در کلیات به‌این ترتیب عمل می‌کند: اگر اولویت x از اولویت y بیش‌تر باشد، x را در پشته درج می‌کند. ولی تا وقتی که پشته عنصر دارد و اولویت y از اولویت x بیش‌تر است، y را از پشته خارج و آن‌را در انتهای رشته‌ی خروجی می‌نویسد. این کار را تکرار می‌کند تا یا پشته خالی شود یا اولویت عنصر بالای آن از x کم‌تر شود. در هر دو صورت، x را در بالای پشته درج می‌کند. شهود این الگوریتم روشن است، سمت چپ‌ترین عمل‌گر، یا β را در نظر بگیرید که مستقیماً بر روی عمل‌وندهای کناری خود به‌نام‌های a و b اِعمال شود. یعنی در عبارت میان‌وندی داشته باشیم: $\cdots a\beta b \cdots$ و اولویت β از عمل‌گرهای مجاورش بیش‌تر باشد. در این‌صورت $a\beta b$ را می‌توانیم به $ab\beta$ تبدیل کنیم. β هنگامی قابل تشخیص است که عمل‌گر بعد از آن در عبارت ورودی اولویتی کم‌تر داشته باشد، و این‌کار را الگوریتم فوق انجام می‌دهد. با این تبدیل $a\beta b$ از ورودی حذف و به‌جای آن انگار که یک عمل‌وند $r = a\beta b$ جای‌گزین می‌شود. در نتیجه می‌توانیم بر روی بقیه‌ی عبارت ورودی استقرا بزنیم.

اما نکته‌ی مهم نحوه‌ی مقایسه‌ی اولویت عمل‌گرهاست. اگر پرانتز باز را عمل‌گری حساب کنیم که در ورودی ظاهر شده است، اولویتش از همه بیش‌تر است و همیشه باید وارد پشته شود (چون به یک عبارت با عمق بیش‌تر رسیده‌ایم)، ولی در داخل پشته اولویت آن از همه کم‌تر است و فقط به‌وسیله‌ی پرانتز بسته‌ی متناظر از پشته خارج می‌شود. اگر x و y دارای اولویت یک‌سان باشند (مثلاً یکی * و دیگری / باشد)، برای عمل‌گرهای جمع، تفریق، ضرب و تقسیم، می‌دانیم که اولویت y بیش‌تر از اولویت x است (چون در این صورت عمل‌گر سمت چپ زودتر اعمال می‌شود). ولی اگر این‌دو توان باشند، اولویت x از y بیش‌تر است.

نکته‌ی دیگر، دیدن پرانتز بسته در ورودی است. این نویسه نشان‌دهنده‌ی اتمام یک زیرعبارت است که از نزدیک‌ترین پرانتز باز به آن شروع شده است. چون ما طوری عمل

می‌کنیم که هیچ عمل‌گری نتواند پرانتز باز را از پشته خارج کند، به‌هنگام دیدن پرانتز بسته همه‌ی عمل‌گرهای پشته به‌ترتیب از بالای پشته یک‌به‌یک برداشته و در خروجی نوشته می‌شوند تا این‌که به پرانتز باز برسیم که آن نیز از پشته برداشته می‌شود و ما در عمل یک زیرعبارت را به‌طور کامل به گونه‌ی پس‌وندی آن تبدیل کرده‌ایم.

در انتها و پس از خواندن آخرین نویسه‌ی عبارت ورودی، باز عمل‌گرها را یک‌به‌یک از پشته خارج و در خروجی چاپ می‌کنیم.

رویه‌ی INFIX-TO-POSTIX گام‌های این الگوریتم را نشان می‌دهد. برای این‌که رفتار الگوریتم برحسب عمل‌گر ورودی و عمل‌گر بالای پشته خودکار شود جدول ۴-۳ ($action[i,j]$) را تعریف می‌کنیم که تعیین می‌کند اگر عمل‌گر i را خوانده باشیم و در بالای پشته عمل‌گر j باشد چه‌کاری باید انجام دهیم.

```
INFIX-TO-POSTFIX (infix)
    ▷ عبارت میان‌وندی را در حالت کلی به گونه‌ی پس‌وندی آن تبدیل می‌کند
    ▷ الگوریتم از پشته‌ی S و ماتریس action استفاده می‌کند
 1  INTIALIZE-ACTIONS()
 2  while there is token in infix
 3      do read token c from infix
 4         if c is an operand
 5           then write c at the end of postfix
 6           else done ← false
 7                while not done
 8                   do if ISEMPTY(S)
 9                         then PUSH(c, S)
10                              done ← true
11                         else t ← TOP(S)
12                              if action[c, t] ='PUSH'
13                                then PUSH (S, c)
14                                     done ← true
15                                else if t ≠ '('
16                                       then write t at postfix
17                                     POP(S)
18  while not isEmpty(S)
19      do write TOP(S) at the end of postfix
20         POP(S)
```

در این الگوریتم می‌توان به‌جای خروجی یک پشته‌ی دیگر تعریف کرد که اشاره‌گر به درخت‌هایی را که تولید می‌شوند در خود نگاه دارد. در این صورت، در الگوریتم فوق

جدول ۴-۳ جدول $action[c,t]$

عمل‌گر ورودی c:	عمل‌گری که بالای پشته است : t						
	(	−	+	×	/	^	¬
(	PUSH	PUSH	PUSH	PUSH	PUSH	PUSH	PUSH
−	PUSH	POP	POP	POP	POP	POP	POP
+	PUSH	POP	POP	POP	POP	POP	POP
*	PUSH	PUSH	PUSH	POP	POP	POP	POP
/	PUSH	PUSH	PUSH	POP	POP	POP	POP
^	PUSH	PUSH	PUSH	PUSH	PUSH	PUSH	POP
¬	PUSH	PUSH	PUSH	PUSH	PUSH	PUSH	POP
)	POP-more	POP	POP	POP	POP	POP	POP

به‌جای چاپ هر عمل‌وند در خروجی، یک گره تنها با آن عمل‌وند ایجاد می‌شود و اشاره‌گر آن در بالای پشته قرار می‌گیرد. به‌جای چاپ عمل‌گر در خروجی، یک گره با آن عمل‌گر ایجاد می‌شود. بسته به نوع آن عمل‌گر، یک یا دو درخت از بالای پشته برداشته می‌شود و به‌عنوان فرزندان چپ و راست (برای عمل‌گرهای دودویی) و یا فقط فرزند راست (برای عمل‌گرهای یگانی) به آن گره (البته با برچسب مناسب) متصل می‌شود (می‌شوند). زیردرخت حاصل از این کار در بالای پشته درج می‌شود.

درستی الگوریتم

درستی این الگوریتم را با استقرا ثابت می‌کنیم. پایه‌ی استقرا روشن است. اگر فرض کنیم که الگوریتم یک عبارت کوچک‌تر را به‌درستی به نگارش پس‌وندی تبدیل می‌کند و در انتهای این‌کار پشته خالی است، با این فرض استقرا می‌توان دید که همه‌ی زیرعبارت‌هایی که با یک پرانتز باز شروع و با پرانتز متناظر آن بسته می‌شوند به‌درستی به یک عبارت پس‌وندی معادل تبدیل می‌شوند. پس عبارت ورودی را می‌توان بدون پرانتز در نظر گرفت. در چنین عبارتی ادعا می‌کنیم که آخرین عمل‌گر مثلاً β (عمل‌گری که در ریشه‌ی درخت عبارت قرار می‌گیرد) به‌درستی تشخیص داده می‌شود. می‌دانیم که در عبارت $E_1 \beta E_2$، β کم‌اولویت‌ترین عمل‌گر است. پس وقتی β را از ورودی می‌خوانیم، موجب آن می‌شود که (مانند پرانتز بسته) همه‌ی عمل‌گرهای موجود در پشته یک‌به‌یک از پشته خارج و در خروجی نوشته شوند. یعنی طبق فرض استقرا، نگارش پس‌وندی E_1 در خروجی نوشته می‌شود. سپس β وارد پشته می‌شود و چون اولویتش از همه‌ی عمل‌گرهای E_2 کم‌تر است در پشته باقی می‌ماند تا E_2 هم طبق استقرا به نگارش پس‌وندی تبدیل شده و در خروجی

نوشته شود. در انتها، β هم در خروجی ظاهر می‌شود. بنابراین عبارت به‌درستی به‌صورت پس‌وندی نگارش می‌شود.

تبدیل نگارش پس‌وندی به نگارش پیش‌وندی

برای این تبدیل یک روش بازگشتی متفاوت ارائه می‌کنیم. با فرض داشتن یک زیرعبارت صحیح پس‌وندی به‌نام E، رویه‌ی POSTFIX-TO-PREFIX(i, j) فرض می‌کند i اندیس انتهایی یک زیرعبارت درستِ پس‌وندی به‌نام E' در آرایه‌ی *postfix* است و j هم آخرین اندیس عبارت پیش‌وندی معادل در آرایه‌ی *prefix*. در شروع $i = j = n$. این الگوریتم نگارش پیش‌وندی E را که در بخش $postfix[?\cdots i]$ قرار دارد تولید می‌کند و از اندیس بالا به‌پایین و در بخش $prefix[?\cdots j]$ می‌نویسد. نیازی نیست که ما اندیس شروع E و E' را داشته باشیم. با فرض آن‌که متغیرهای i و j از نوع آدرسی هستند و تغییرات در مقادیر آن‌ها در انتهای یک فراخوانی پابرجا خواهد ماند، رویه را طوری طراحی می‌کنیم که در انتهای فراخوانی، i اندیس شروع E و مقدار j اندیس E'، در آرایه‌ی *prefix* باشد.

```
POSTFIX-TO-PREFIX (i, j)
    ▷ زیرعبارت پس‌وندی postfix[?···i] را به نگارش پیش‌وندی
      در prefix[?···j] تبدیل می‌کند
    ▷ در انتها، i و j اندیس‌های شروع این دو زیرعبارت هستند
    ▷ postfix و prefix آرایه‌های سراسری فرض می‌شوند
 1  switch
 2     case postfix[i] is an operand
 3        do prefix[j] ← postfix[i]
 4     case postfix[i] is a binary operator
 5        do operator ← postfix[i]
 6           i ← i − 1
 7           POSTFIX-TO-PREFIX (i, j)
 8           i ← i − 1; j ← j − 1
 9           POSTFIX-TO-PREFIX (i, j)
10           prefix[j − 1] ← operator
11           j ← j − 1
12     case postfix[i] is an unary operator
13        do operator ← postfix[i]; i ← i − 1
14           POSTFIX-TO-PREFIX (i, j)
15           prefix[j − 1] ← operator
16           j ← j − 1
```

با این ترتیب مشخص است که $postfix[i]$ باید یک عمل‌گر باشد. اگر این عمل‌گر دودویی (β) باشد، حتماً عبارت به‌صورت $E = E_1 E_2 \beta$ است که E_1 و E_2 دو عبارت پس‌وندی هستند که بلافاصله قبل از β ظاهر می‌شوند. $E' = \beta E'_1 E'_2$ نگارش پیش‌وندی است که در آن E'_1 و E'_2 به‌ترتیب نگارش پیش‌وندی E_1 و E_2 هستند. با داشتن اندیس آخر E_2 به‌صورت بازگشتی می‌توانیم E'_2 را بنویسیم و با فرض استقرا، اندیس‌ها به ابتدای E_1 و E'_1 تغییر می‌کنند. با یک فراخوانی دیگر می‌توانیم E'_1 را بسازیم و در انتها β را قرار دهیم. اگر عمل‌گر یگانی باشد، کاری مشابه آن‌چه بیان شد انجام می‌دهیم.

روشن است که زمان اجرای این الگوریتم هم خطی است.

تبدیل نگارش پس‌وندی به درخت عبارت

رویه‌ی POSTFIX-TO-TREE(j) نگارش پس‌وندی یک عبارت را که آخرین اندیس آن j است به درخت عبارت تبدیل می‌کند. در انتهای این فراخوانی، مقدار j به اندیس ابتدایی زیرعبارت آن تغییر می‌کند. لذا j یک متغیر آدرسی است. در برنامه‌ی اصلی، باید ابتدا یک درخت ایجاد شود و ریشه‌ی آن با فراخوانی این رویه ساخته شود.

درستی این الگوریتم مانند POSTFIX-TO-PREFIX(i, j) اثبات می‌شود. زمان اجرای این رویه هم خطی است.

```
POSTFIX-TO-TREE-1 (j)
    ▷ یک درخت عبارت برای postfix[?..j] ایجاد می‌کند
    ▷ فرض می‌شود که j یک متغیر آدرسی است و در انتها برابر اندیس شروع خواهد بود
 1  n ← ALLOCATE-NODE(A[j], null, null)
 2  switch
 3     case postfix[j] is a binary operator
 4        do j ← j − 1
 5           right[n] ← POSTFIX-TO-TREE-1(j)
 6           j ← j − 1
 7           left[n] ← POSTFIX-TO-TREE-1(j)
 8     case postfix[j] is a unary operator
 9        do j ← j − 1
10           right[n] ← POSTFIX-TO-TREE-1(j)
11  return n
```

همین‌کار را می‌توان به‌صورت غیربازگشتی و به‌کمک رویه‌ی POSTFIX-TO-TREE-2(*postfix*) انجام داد. در این روش آرایه‌ی *postfix* از ابتدا تا انتها پیمایش می‌شود. با دیدن هر عمل‌وند تک حرفی، یک درخت با یک گره که برچسب آن عمل‌وند خوانده شده است، ایجاد شده و در بالای یک پشته به‌نام S درج می‌شود. اگر درایه‌ای که می‌بینند، یک عمل‌گر دودویی β باشد، حتماً به انتهای زیرعبارتی مثل $E_1 E_2 \beta$ رسیده است. طبق فرض استقرا درخت‌های عبارت E_1 و E_2 به‌نام‌های T_1 و T_2 قبلاً ساخته‌شده و در بالای پشته قرار دارند؛ T_2 در بالای پشته و T_1 بلافاصله قبل از آن. الگوریتم درخت جدیدی با ریشه‌ی r ایجاد می‌کند که T_1 زیردرخت چپ و T_2 زیردرخت راست آن است. این درخت جدید به‌جای T_1 و T_2 در بالای پشته قرار می‌گیرد تا کار به همین منوال دنبال شود.

```
POSTFIX-TO-TREE-2 (postfix)
    ▷ یک درخت عبارت برای postfix ایجاد می‌کند
 1  CREATE-STACK(S)
    ▷ عناصر پشته اشاره‌گر به یک زیردرخت هستند
 2  for i ← 1 to length[postfix]
 3    do if postfix[i] is an operand
 4          then PUSH(S, ALLOCATE-NODE(postfix[i], null, null))
 5       if postfix[i] is a unary operator
 6          then r ← ALLOCATE-NODE(postfix[i], null,POP(S))
 7               PUSH(S, r)
 8       if postfix[i] is a binary operator
 9          then t2 ← POP(S)
10               r ← ALLOCATE-NODE(postfix[i],POP(S), t1)
11               PUSH(S, r)
12  return TOP(S)
```

اگر عمل‌گر یگانی باشد، کاری مشابه انجام می‌شود و به‌جای درخت T که در بالای پشته قرار دارد درختی جدید با ریشه‌ی عمل‌گر یگانی و زیردرخت چپ تهی و زیر درختِ T ایجاد شده و در بالای پشته قرار می‌گیرد.

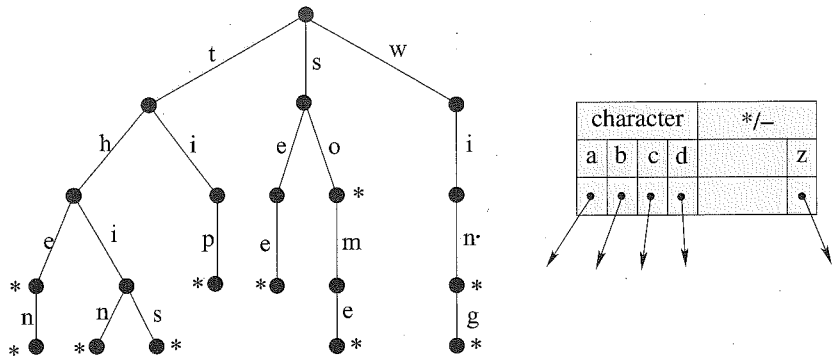
این رویه، هم سریع است و هم زمان اجرای آن نسبت به اندازه‌ی آرایه‌ی *postfix* خطی است.

۴-۴-۹ تِرای، درختی برای ذخیره‌ی رشته‌ها

تِرای[۶۷] یک داده‌ساختار درختی برای ذخیره‌ی مجموعه‌ای از کلمات (رشته[۶۸]ها) با طول مختلف از نویسه[۶۹]های یک زبان است، به‌طوری‌که بتوان به‌صورت کارا تشخیص داد که آیا یک کلمه در آن مجموعه وجود دارد یا خیر. این عملی است که به‌صورت مکرر در سیستم‌های غلط‌یاب[۷۰] انجام می‌شود، و تِرای می‌تواند یک داده‌ساختار مورد استفاده در این سیستم‌ها باشد؛ هرچند که در عمل داده‌ساختارهای کاراتری برای این‌کار وجود دارد.

یک تِرای برای کلمات انگلیسی به این صورت ساخته می‌شود: اگر کلمات انگلیسی فقط از نویسه‌های کوچک 'a' تا 'z' تشکیل شده باشند، هر گره تِرای، ۲۶ فرزند خواهد داشت؛ هرکدام برای یک نویسه و هر نویسه در عمل برچسب یکی از یال‌های خروجی آن گره است. به هر گره یک رشته نسبت داده می‌شود که از چسباندن نویسه‌های موجود در تنها مسیر از ریشه تا آن گره در تِرای به‌دست می‌آید. اگر رشته‌ی یک گره در مجموعه‌ی کلماتی که تِرای ذخیره می‌کند وجود داشته باشد، یک برچسب * به آن گره داده می‌شود.

شکل ۴-۳۴ یک تِرای و ساختار هر گره آن‌را نشان می‌دهد. این تِرای کلمات wing، win، so، some، see، tip، this، thin، then، و the را ذخیره می‌کند.



شکل ۴-۳۴ یک تِرای برای کلمات wing، win، so، some، see، tip، this، thin، then، و the. ساختار هر گره تِرای هم نشان داده شده است که هر کدام شامل چندین مؤلفه است.

---

[۶۷] Trie برگرفته‌شده از کلمه‌ی retrieval
[۶۸] string
[۶۹] حرف الفبا یا character
[۷۰] spell-check

ساختار گره‌ی یک ترای به‌زبان پاسکال به‌صورت زیر است:

```
type Lettertype =  'a' .. 'z';
     Node = ^   Nodetype;
     Nodetype = record
                   letter : Lettertype;
                   isword: ('*', '-');
                   children: array[Lettertype] of Node
                end;
```

اعمال بر روی ترای

اعمال زیر بر روی یک ترای اجرا می‌شود:

- درج یک رشته در ترای،
- حذف یک رشته از ترای،
- جست‌وجو برای یافتن یک رشته در ترای، و
- نوشتن تمام رشته‌های موجود در ترای.

به‌عنوان مثال، در این‌جا رویه‌ی درج یک رشته‌ی x در یک ترای T آمده است. این الگوریتم تک‌تکِ نویسه‌های x را بررسی می‌کند و با شروع از ریشه‌ی T بر اساس نویسه‌ای که می‌بیند، تا جایی‌که امکانش هست در T جلو می‌رود. اگر در این مسیر به یک پیوند **null** برسد، بقیه‌ی مسیر را با اجرای دستورات سطر ۴ ایجاد می‌کند. در انتها هم با گمارش یک برچسب * به آخرین گره، ثبت می‌کند که x در T وجود دارد.

```
TRIE-INSERT (T, x)
    ▷ کلمه‌ی x را در ترای T درج می‌کند
1  p ← ROOT(T)
2  for i ← 1 to length[x]
3     do if the value of x[i]th link of p is null
4           then x[i]th link of p ←
                        ALLOCATE-NODE(with x[i] as its value)
5        p ← x[i]th child of p
6  tag[p] ← '*'
```

رویه‌های مربوط به بقیه‌ی اعمال را به شما وا می‌گذاریم.

نقطه ضعف مهم ترای وجود تعداد زیادی اشاره‌گر تهی است که موجب اتلاف حافظه می‌شود. در کاربردهای عملی، مانند غلط‌یاب از گونه‌های دیگر ترای که اشاره‌گر تهی کم‌تری دارند یا از روش درهم‌سازی (که در فصل ۵ به آن خواهیم پرداخت) استفاده می‌شود.

تمرین‌های بخش ۴-۴

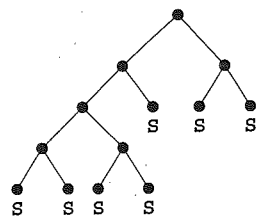
۱.۴-۴ یک درخت T با n عنصر را در نظر بگیرید که هر گره غیر برگ آن ۲ فرزند دارد. فرض کنید $E(T)$ و $I(T)$ به‌ترتیب مجموع عمق برگ‌ها و مجموع عمق داخلی (عناصر غیر برگ) در T باشد. چه رابطه‌ای بین $E(T)$ و $I(T)$ همواره برقرار است؟ حدس بزنید و با استقرا اثبات کنید.

۲.۴-۴ یک S-Term ترتیبی از تعدادی حرف S و پرانتز است که به صورت زیر تعریف می‌شود:

- S یک S-Term است.
- اگر M و N دو S-Term باشند، (MN) هم یک S-Term است.

مثال: ((((SS)(SS))S)(SS)).

یک S-Term را می‌توان به صورت یک درخت دودویی کامل نشان داد. برای مثال درخت مربوط به عبارت فوق به صورت زیر است:



((((SS)(SS)S)(SS))

الف) رویه‌ای بنویسید تا با دریافت یک رشته‌ی S-Term از ورودی، درخت مربوط به آن را ایجاد کند.

پرانتز بسته در نوشتن S-Term ها اطلاعات جدیدی به ما نمی‌دهد و می‌تواند حذف شود. در این صورت عبارت فوق SS)SSS(SS)))) خواهد بود. به این‌گونه از S-Term ها «فرم خلاصه‌شده» می‌گوییم.

ب) رویه‌ای بنویسید تا از فرم خلاصه‌شده‌ی یک S-Term درخت آن را ایجاد کند.

۳.۴-۴ عبارت $a - b * c + d - e/g/h$ را، با فرض اعمال اولویت عمل‌گرها، به چند طریق می‌توان به‌درستی پرانتزگذاری کرد به‌طوری‌که مقدار عبارت حاصل به‌ازای همه‌ی مقادیر a تا h با مقدار عبارت اصلی برابر شود؟

دقت کنید که دور یک متغیر تنها در عبارت، پرانتز گذاشته نمی‌شود و یک عبارت پرانتزگذاری‌شده باید صفر یا تعداد زوجی پرانتز داشته باشد. مثلاً $((a + b) + c)$، $a + (b + c)$ و $a + b + c$ سه‌عبارت پرانتزگذاری‌شده‌ی درست برای $a + b + c$ است و برای $(a) + b + c$ نیست.

۴.۴-۴ رویه‌ی بازگشتی PREFIX2TREE را بنویسید تا با دریافت فرم پیش‌وندی یک عبارت ریاضی به‌طول n که در آرایه‌ی $prefix[1 \cdots n]$ قرار دارد، درخت عبارت را تولید کند. الگوریتم خود را تحلیل کنید.

الف) آخرین فراخوان بازگشتی را در این رویه حذف کنید و رویه‌ی حاصل را بنویسید.

ب) رویه‌ی حاصل را به‌صورت غیر بازگشتی بنویسید.

۵.۴-۴ یک رویه با زمان اجرای خطی بنویسید تا از یک درخت عبارت، نگارش میان‌وندی آن‌را با حداقل پرانتزها تولید کند. پرانتزی مورد نیاز است که اگر برداشته شود، ترتیب اجرای عمل‌وندها تغییر کند.

۶.۴-۴ فرض کنید T یک درخت دودویی کامل با n گره و به‌ارتفاع $\lg n$ است. می‌خواهیم مسیر ساده‌ای بین یک رأس u به یک رأس دیگر به نام v پیدا کنیم. گره‌های u و v داده شده‌اند و می‌دانیم که هر گره از این درخت به گره‌های فرزند و گره‌ی پدر دسترسی دارد. این کار را با چه مرتبه‌ای می‌توان انجام داد [۷۱]؟

⋆ **۷.۴-۴** درخت مبنا درختی دودویی است که مانند تِرای، مجموعه‌ای از رشته‌های ساخته‌شده از ۰ و ۱ را نشان می‌دهد. در این درخت، هر گره متناظر با یک رشته است: برای ریشه این رشته تهی است. رشته‌ی هر گره برابر رشته‌ی پدر این گره به اضافه‌ی یک حرف است؛ این حرف برابر ۱ است اگر فرزند راست باشد و ۰ است اگر فرزند چپ باشد. هر گره علاوه بر اشاره‌گر به فرزندان راست و چپ حاوی یک متغیر منطقی است. اگر رشته‌ی متناظر با این گره در مجموعه‌ی کلمات درخت مبنا وجود داشته باشد، این متغیر ۱ است.

الف) رشته‌های ۱۱۰۱، ۰۱۰۰، ۰۰۱۱، ۰۰۱۰، ۰۰۰۱، ۱۰۰۰ و ۱۰۰۱ را به‌ترتیب در یک درخت مبنای تهی درج می‌کنیم. درخت حاصل را رسم کنید.

ب) الگوریتمی طراحی کنید که با گرفتن مجموعه‌ای از n رشته از ۰ و ۱، درخت مبنا را بسازد.

(توجه: مسئله‌ی ۹.۴ در صفحه‌ی ۲۵۰ هم در رابطه با درخت مبنا است.)

[۷۱] کنکور کارشناسی ارشد رشته‌ی مهندسی کامپیوتر، سال ۱۳۸۷

۴-۵ درخت دودویی جست‌وجو

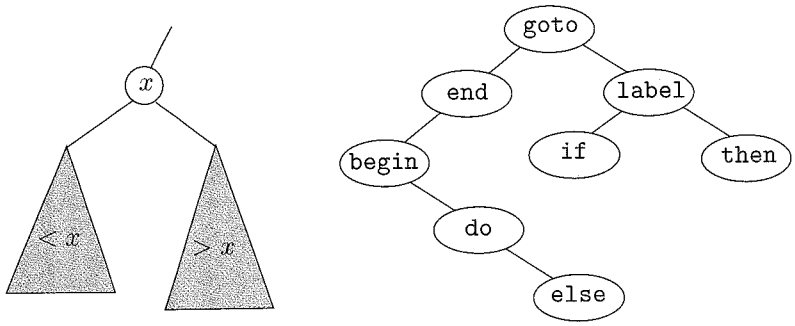
«درخت دودویی جست‌وجو[۷۲]» داده‌ساختاری است که اعمال مختلف فرهنگ داده‌ای (جست‌وجو، درج و حذف) را به‌طرز ساده‌ای انجام می‌دهد. پیاده‌سازی آن ساده و هزینه‌ی انجام اعمال مختلف در آن در حالت میانگین بهینه است. جدول نمادها[۷۳] یکی از کاربردهای مهم فرهنگ‌داده‌ای است که کلمات کلیدی یک زبان برنامه‌سازی را ذخیره می‌کند. کامپایلر این زبان باید هر کلمه‌ای را که کاربر در برنامه‌اش تعریف می‌کند در زمان کوتاهی بررسی کند که آیا جزو کلمات کلیدی هست یا خیر. درخت دودویی جست‌وجو برای این‌کار بسیار مناسب است.

این درخت مبنای گونه‌های پیش‌رفته‌تری از داده‌ساختارهاست که در فصل ۷ بررسی می‌شوند.

تعریف ۴-۱ درخت دودویی جست‌وجو (د.د.ج) یک درخت دودویی است که همه‌ی برچسب‌های گره‌های زیردرخت چپ هر گره آن با برچسب x، کوچک‌تر از x و همه‌ی برچسب‌های گره‌های زیردرخت راست آن بزرگ‌تر از x باشد.

مثلاً شکل ۴-۳۵ یک د.د.ج را برای ۸ گره که کلمات کلیدی زبان پاسکال هستند نشان می‌دهد.

روشن است که پیمایش میان‌ترتیب یک د.د.ج، مرتب‌شده‌ی عناصر را تولید می‌کند.

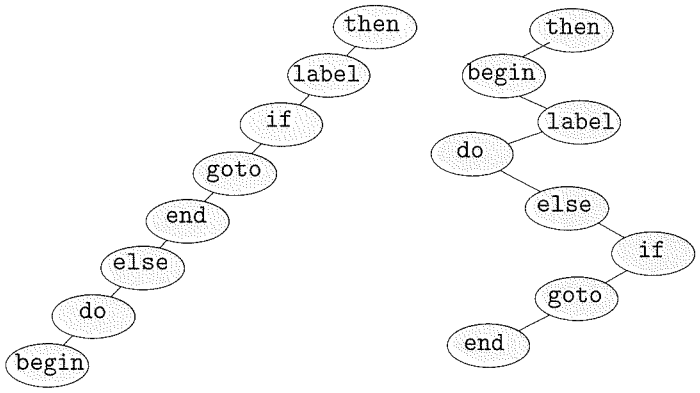


شکل ۴-۳۵ تعریف درخت دودویی جست‌وجو (د.د.ج) و یک نمونه‌ی آن برای یک جدول نمادها از ۸ کلمه‌ی کلیدی زبان پاسکال.

---

[۷۲] binary search tree
[۷۳] symbol tables

ارتفاع یک د.د.ج وابسته به ترتیب عناصری است که در آن درج می‌شود یا از آن حذف می‌گردد. مثلاً اگر در یک د.د.ج تهی همان ۸ کلمه‌ی شکل ۴-۳۵ از بزرگ به کوچک درج شود، درخت سمت چپ شکل ۴-۳۶ ایجاد می‌شود که بیش‌ترین ارتفاع ممکن (یعنی ۷) را دارد. البته تعداد زیادی د.د.ج با ارتفاع ۷ وجود دارد که شکل سمت راست یکی دیگر از آن‌هاست.



شکل ۴-۳۶ دو د.د.ج با ۸ عنصر با بیش‌ترین ارتفاع.

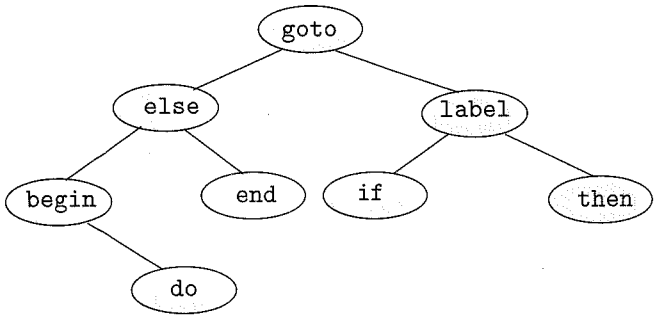
نکته‌ی ۴-۲ تعداد د.د.ج های متفاوت با n عنصر و یا ارتفاع $n-1$ برابر 2^{n-1} است.

دلیل این نکته آن است که هر یک از $n-1$ عنصر غیر برگ تنها یک فرزند دارد و هر کدام فقط می‌تواند یا فرزند چپ باشد یا راست. طبق اصل ضرب، تعداد کل حالات برابر 2^{n-1} می‌شود.

کم‌ترین ارتفاع برای یک د.د.ج با n عنصر برابر $\lfloor \lg n \rfloor$ است که یک نمونه‌ی آن برای ۸ عنصر مورد نظر در شکل ۴-۳۷ دیده می‌شود. اثبات این فرمول با استقرا و یا با روش‌های شمارشی دیگر ساده است.

مسئله‌ی ۴-۲ با n عنصر داده‌شده چند تا د.د.ج با ارتفاع $\lfloor \lg n \rfloor$ می‌توان ساخت؟

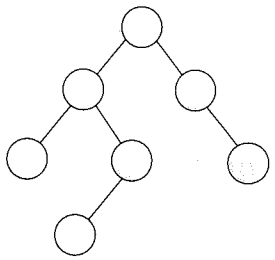
نکته‌ی جالب در د.د.ج آن است که اگر n عنصر آن با ترتیبی به‌صورت تصادفی درج شوند درختی ایجاد خواهد شد که میانگین ارتفاع آن $\mathcal{O}(\lg n)$ است. این نکته که در انتهای



شکل ۴-۳۷ د.د.ج با ۸ عنصر و با کم‌ترین ارتفاع.

این بخش ثابت می‌کنیم از آن نظر مهم است که در نتیجه‌ی آن اَعمال فرهنگ داده‌ای بر روی این درخت از همین مرتبه ــ که کم‌ترین مرتبه‌ی ممکن است ــ خواهند بود.

مسئله‌ی ۴-۳ اعداد $\{۶, ۲۲, ۹, ۱۴, ۱۳, ۱, ۸\}$ را به درخت زیر طوری نسبت دهید که د.د.ج شود.



به‌چند حالت می‌توان اعداد بیان‌شده را وارد یک درخت تهی کرد تا در انتها، درخت فوق حاصل شود؟ این تعداد را دقیقاً محاسبه کنید.

نکته‌ی ۴-۳ تعداد د.د.ج هایی که با $a_1 < a_2 < \cdots < a_n$ می‌توان ساخت برابر است با $\frac{1}{n+1}\binom{2n}{n}$.

هر درخت دودویی با n عنصر را می‌توان به‌صورت میان‌ترتیب پیمایش کرد و عناصر a_1 تا a_n را به تک‌تک گره‌های آن تخصیص داد و یک د.د.ج ساخت. بنابراین مطابق رابطه‌ی بازگشتی بیان‌شده در بخش ۴-۴-۶، تعداد د.د.ج هایی که ساخته می‌شوند همان عدد nام کاتالان است.

۴-۵-۱ اَعمال مختلف برروی درخت دودویی جست‌وجو

اَعمال پایه‌ای بر روی یک د.د.ج به‌صورت زیر است: فرض می‌کنیم که هر گره r در د.د.ج T یا مؤلفه‌ی یکتای $key[r]$ دارد که کلید آن گره است. همچنین سه اشاره‌گر $left[r]$، $right[r]$، $parent[r]$ که به‌ترتیب به فرزندان چپ، راست و پدر r اشاره می‌کند.

جست‌وجو

برای جست‌وجوی عنصر x در زیردرختی به ریشه‌ی r، اگر $x = key[r]$ بود که r جواب است، وگرنه بسته به این‌که کلید r کوچک‌تر یا بزرگ‌تر از x باشد، جست‌وجو به‌ترتیب در زیردرخت چپ یا راست r به همین روش دنبال می‌شود.

```
BST-SEARCH (r, x)
    ▷ r یک گره (یا ریشه‌ی) یک د.د.ج است
 1  if r = null or x = key[r]
 2     then return r
 3  if x < key[r]
 4     then return BST-SEARCH(left[r], x)
 5     else  return BST-SEARCH(right[r], x)
```

جست‌وجوی غیر بازگشتی

```
NR-BST-SEARCH (r, x)
    ▷ r یک گره (یا ریشه‌ی) یک د.د.ج است
 1  while r ≠ null and x ≠ key[r]
 2     do if x < key[r]
 3           then r ← left[r]
 4           else  r ← right[r]
 5  return r
```

مسئله‌ی ۴-۴ به مسیری که در د.د.ج طی می‌شود تا به x برسیم «مسیر جست‌وجوی x» و به دنباله‌ی کلیدهای ملاقات شده در مسیر جست‌وجوی x یک «دنباله‌ی جست‌وجوی x» می‌گوییم. مثلاً در یک د.د.ج با کلیدهای بین ۱ تا ۱۰۰۰ دو دنباله‌ی

$$\langle ۲, ۲۵۲, ۴۰۱, ۳۹۸, ۳۳۰, ۳۴۴, ۳۹۷, ۳۶۳ \rangle$$

$$\langle ۹۲۴, ۲۲۰, ۹۱۱, ۲۴۴, ۸۹۸, ۲۵۸, ۳۶۲, ۳۶۳ \rangle,$$

دنباله‌های جست‌وجو هستند، ولی

$$\langle ۹۲۵, ۲۰۲, ۹۱۱, ۲۴۰, ۹۱۲, ۲۴۵, ۳۶۳ \rangle$$

یک دنباله‌ی جست‌وجو نیست چون از چپ به راست که حرکت می‌کنیم، ۹۱۲ از ۹۱۱ بیش‌تر است.

یک مسئله این است که اگر دنباله‌ی

$$A = \langle a_۱, a_۲, \ldots, a_n \rangle$$

داده شده باشد، با چه الگوریتم کارایی می‌توان تشخیص داد که A یک دنباله‌ی جست‌وجو است؟

حل: یک راه‌حل خطی برای این مسئله به این صورت است که از سمت چپ به راست دنباله حرکت می‌کنیم و در هر مرحله بازه‌ی (l, r) را به‌روز می‌کنیم. این بازه‌ای است که xای که جست‌وجو می‌کنیم تا این مرحله در آن قرار دارد، یعنی $l \leq x \leq r$. در ابتدا و برای $i = ۱$ محدودیتی برای بازه نیست؛ یعنی بازه برابر $(-\infty, \infty)$ است. برای $i = k$ بازه‌ی (l, r) همچنین به این معنی است که همه‌ی کلیدهای زیردرختی به ریشه‌ی a_k در آن بازه قرار دارند. اگر $a_k > a_{k+۱}$، باید $l < a_{k+۱}$ و ما بازه را به $(a_{k+۱}, r)$ تغییر می‌دهیم. اگر $a_k < a_{k+۱}$، باید $a_{k+۱} < r$ و در آن‌صورت ما بازه را به $(l, a_{k+۱})$ تغییر می‌دهیم. اگر بتوانیم بدون مشکلی تا انتهای دنباله جلو برویم، دنباله‌ی جست‌وجو صحیح و در صورت بروز تضاد، ناصحیح است.

یافتن عنصر کمینه

اگر از ریشه با اشاره‌گر چپ حرکت کنیم و به گره‌ای برسیم که فرزند چپ ندارد، آن گره حاوی عنصر کمینه در د.د.ج است. به‌صورت بازگشتی، اگر اشاره‌گر سمت چپ r تهی باشد، در آن‌صورت r عنصر کمینه است، و گرنه عنصر کمینه در زیردرخت چپ r عنصر کمینه‌ی همه‌ی درخت است.

```
BST-MINIMUM (r)
1  if left[r] = null
2     then return r
3  BST-MINIMUM(left[r])
```

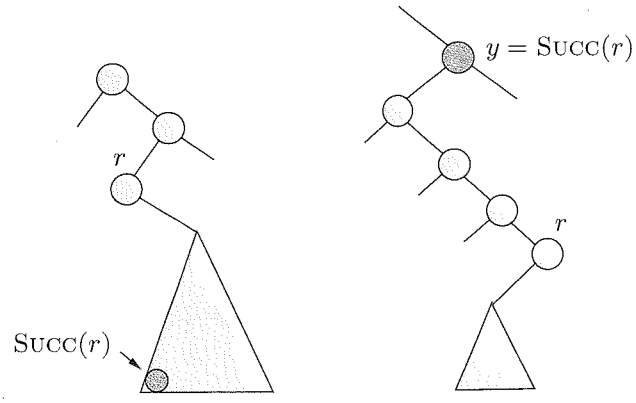
همین‌کار را می‌توان به‌صورت غیر بازگشتی نیز انجام داد.

```
NR-BST-MINIMUM (r)
1  while left[r] ≠ null
2     do r ← left[r]
3  return r
```

عنصر بعدی

چنان‌چه در شکل ۴-۳۸ نشان داده شده است، عنصر بعدی یک گره r در یک د.د.ج عنصر کمینه در زیردرخت راست r است. اگر چنین زیردرختی وجود نداشته باشد، در مسیر یک‌تای r به ریشه‌ی درخت، به‌دنبال نزدیک‌ترین جد r به‌نام y می‌گردیم که فرزند y، که در این مسیر است، فرزند چپ آن باشد.

```
BST-SUCCESSOR (r)
1  if right[r] ≠ null
2     then return NR-BST-MINIMUM(right[r])
3  y ← parent[r]
4  while y ≠ null  and r = right[y]
5     do r ← y
6        y ← parent[y]
7  return y
```



شکل ۴-۳۸ یافتن عنصر بعدی r در د.د.ج. راه‌حل برای دو حالتی که r فرزند راست داشته یا نداشته باشد، مشخص شده است.

درج یک عنصر

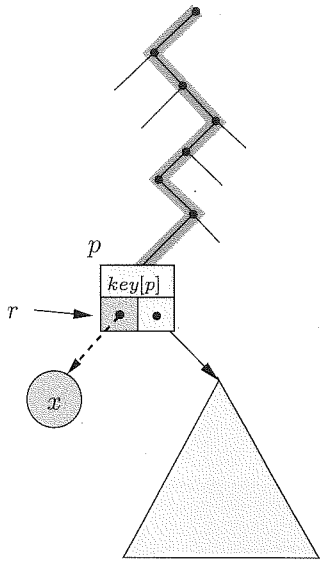
برای درج یک عنصر با برچسب x در زیردرختی به ریشه‌ی r از یک د.د.ج به‌صورت بازگشتی و مطابق زیر عمل می‌کنیم (در این‌جا فقط برای سادگی کار فرض می‌کنیم گره‌ها مؤلفه‌ی «پدر» ندارند): اگر r تهی باشد، چون به‌صورت بازگشتی به r رسیده‌ایم، r در واقع مؤلفه‌ی راست یا چپ یک عنصر از درخت است. بنابراین با فرض آن‌که r یک متغیر آدرسی است، با دستور ALLOCATE-NODE(x,**null**, **null**) یک گره با برچسب x ایجاد و در جای درست به درخت وصل می‌کنیم. توجه کنید که اگر متغیر ارزشی باشد، این اتصال برقرار نمی‌شود.

اگر r تهی نباشد، کلید r را با x مقایسه می‌کنیم. اگر $x < key[r]$، x باید به‌صورت بازگشتی در زیردرخت چپ r و اگر $x > key[r]$ x باید در زیردرخت راستِ r درج شود. اگر $x = key[r]$، کاری انجام نمی‌دهیم چون د.د.ج عنصر تکراری ندارد.

شکل ۴-۳۹ جزئیاتی از این عمل را نشان می‌دهد. مسیر پررنگ مسیر جست‌وجوی x در درخت د.د.ج است که به‌صورت بازگشتی دنبال می‌شود تا به گره p برسیم و فرض کرده‌ایم که مثلاً

$$key[p] > x \text{ و } left[p] = \textbf{null}.$$

در این فراخوانی عنصری با برچسب x ایجاد و از طریق این اشاره‌گر به عنوان فرزند چپ p به درخت وصل می‌شود.



شکل ۴-۳۹ درج یک عنصر با برچسب x در یک د.د.ج. گره با برچسب x حتماً از طریق یک اشاره‌گر تهی (که فرزند چپ یا راست یک گره p است) به درخت وصل می‌شود.

رویه‌ی BST-INSERT(r, x) مراحل این الگوریتم را نشان می‌دهد.

```
BST-INSERT (r, x)
    ▷ r یک گره (یا ریشه‌ی) د.د.ج است
    ▷ مهم: r باید یک پارامتر آدرسی باشد
    ▷ فرض می‌شود مؤلفه‌ی parent وجود ندارد
 1  if r = null
 2     then r ← ALLOCATE-NODE(x, null, null)
 3  if x < key[r]
 4     then BST-INSERT(left[r], x)
 5     else  if x > key[r]
 6              then BST-INSERT(right[r], x)
```

درج بازگشتی با نگه‌داری مؤلفه‌ی پدر

الگوریتم گفته‌شده برای درج کلید x را می‌توان با وجود مؤلفه‌ی پدر گسترش داد. برای این‌کار، در هر فراخوانی x را باید در زیردرختی به‌ریشه‌ی r که p پدر r است درج کنیم. بدیهی است که اولین فراخوانی به‌صورت

$$\text{BST-INSERT-2}(\text{ROOT}[T], \textbf{null}, x)$$

خواهد بود. الگوریتم مشابه قبل کار می‌کند، فقط در هر فراخوانی بازگشتی توجه داریم که اشاره‌گر پدر را هم منتقل کنیم.

```
BST-INSERT-2 (r, p, x)
    ▷ r یک گره (یا ریشه‌ی) د.د.ج
    ▷ p پدر r است
    ▷ مهم: r باید یک پارامتر آدرسی باشد
1   if r = null
2     then r ← ALLOCATE-NODE(x, null, null)
3          parent[r] ← p
4   if x < key[r]
5     then BST-INSERT-2(left[r], r, x)
6     else if x > key[r]
7            then BST-INSERT-2(right[r], r, x)
```

درج یک عنصر به‌صورت غیر بازگشتی

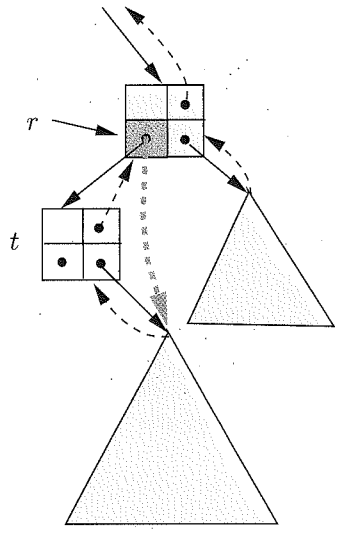
برای درج غیربازگشتی کلید x در یک د.د.ج به‌نام T مانند قبل، باید مسیر جست‌وجوی ناموفق x را در T پیدا کنیم تا به یک اشاره‌گر تهی برسیم. برای این‌که اشاره‌گر پدر را نیز در نظر بگیریم، باید کار جست‌وجو را با دو اشاره‌گر پشت‌سرهم p و $prep$ انجام دهیم که $prep$ همیشه پدر p در T است. مانند الگوریتم بازگشتی، هنگامی که به اشاره‌گر تهی می‌رسیم، گره ایجاد شده را به آن متصل می‌کنیم و اشاره‌گرهای دیگر، از جمله اشاره‌گر پدر را مقداردهی می‌کنیم.

```
NR-BST-INSERT (T, x)
 1  n ← ALLOCATE-NODE(x, null, null)
 2  prep ← null
 3  p ← root[T]
 4  while p ≠ null
 5      do prep ← p
 6         if x < key[p]
 7           then p ← left[p]
 8           else if x > key[p]
 9                  then p ← right[p]
10                  else return
11  parent[n] ← prep
12  if prep = null
13    then root[T] ← n
14    else if x < key[prep]
15           then left[prep] ← n
16           else right[prep] ← n
```

حذف عنصر کمینه

چنان‌چه گفته شد، عنصر کمینه عنصری است که اشاره‌گر چپ آن تهی است و می‌توان از ریشه با دنبال کردن اشاره‌گرهای چپ به آن رسید. با فراخوانی BST-DELETEMIN(r) می‌خواهیم عنصر کمینه‌ی زیردرختی به ریشه‌ی r را از درخت حذف کنیم و برچسب عنصر کمینه را برگردانیم.

بدیهی است که r نمی‌تواند تهی باشد. اگر اشاره‌گر چپ r تهی باشد، خودش عنصر کمینه است. در غیر این صورت، عنصر کمینه در زیردرخت چپ آن قرار دارد و همین کار را می‌توانیم به‌صورت بازگشتی با آن زیردرخت انجام دهیم. در نهایت، به گره‌ای می‌رسیم که اشاره‌گر r (که در شکل ۴-۴۰ نشان داده شده است) به گره کمینه اشاره می‌کند. برای آن‌که بتوانیم این عنصر را پس از حذف از درخت به فضای آزاد برگردانیم، اشاره‌گر t و برچسب r را نگه می‌داریم. دستور $r \leftarrow right[r]$ عملاً گره کمینه را از درخت حذف می‌کند، البته با این فرض که r متغیر آدرسی است. سپس گره t را آزاد می‌کنیم و برچسب آن‌را بر می‌گردانیم.



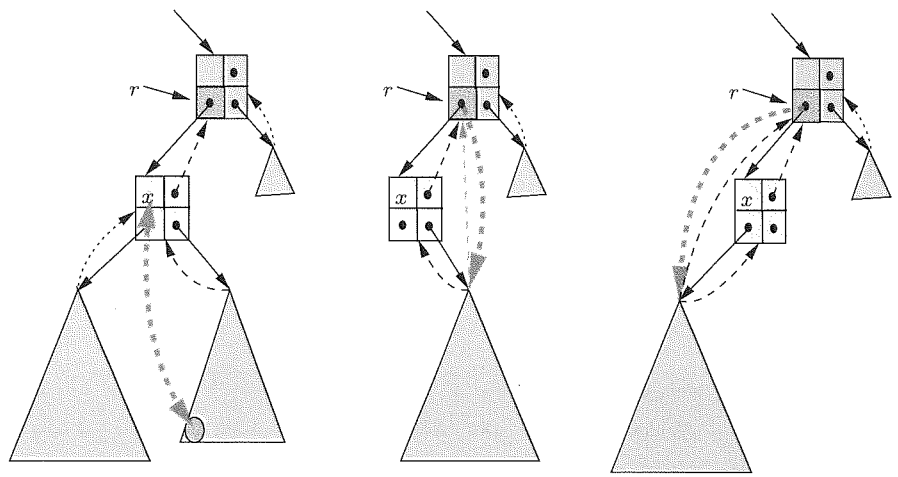
شکل ۴-۴۰ حذف عنصر کمینه در د.د.ج. عنصر کمینه (t) عنصری است که فرزند چپ ندارد و از دنبال کردن اشاره‌گرهای چپ از ریشه، و در آخرین مرحله از r به آن می‌رسیم. برای حذف کافی است که دستور $r \leftarrow right[r]$ انجام شود.

```
BST-DELETEMIN (r)
    ▷ r یک گره (یا ریشه‌ی) د.د.ج است ▷
    ▷ مهم: r باید یک پارامتر آدرسی باشد ▷
 1  if r = null
 2     then error tree is empty
 3  if left[r] = null
 4     then x ← label[r]
 5          t ← r
 6          r ← right[r]
 7          parent[r] ← parent[t]
 8          FREE-NODE (t)
 9          return x
10     else return BST-DELETEMIN(left[r])
```

الگوریتم بازگشتی برای حذف یک عنصر از د.د.ج

رویه‌ی R-BST-DELETE(r, x) عنصری با برچسب x را در یک د.د.ج به ریشه‌ی r حذف می‌کند. این رویه در ابتدا با پارامتر ROOT(T) برای r فراخوانی می‌شود.

رویه ابتدا گره r را با برچسب x پیدا می‌کند (اگر چنین گره‌ای نباشد، کاری انجام نمی‌شود). اگر r فرزند چپ یا راست نداشته باشد (حالت‌های (الف) و (ب) در شکل ۴-۴۱) آن عنصر را می‌توان به‌سادگی حذف، گره را آزاد و اشاره‌گر پدر را مقداردهی کرد. برای این‌کار، توجه کنید که مانند چند رویه‌ی قبل، متغیر r باید از نوع آدرسی باشد. اما اگر r هر دو فرزند را داشته باشد، عنصر کمینه‌ی زیردرخت راست r را حذف می‌کنیم و برچسب آن‌را به‌جای برچسب x در گره r قرار می‌دهیم. با پیمایش میان‌ترتیب درخت، به‌سهولت می‌توان دید که ترتیب عناصر پس از حذف نیز حذف می‌شود.

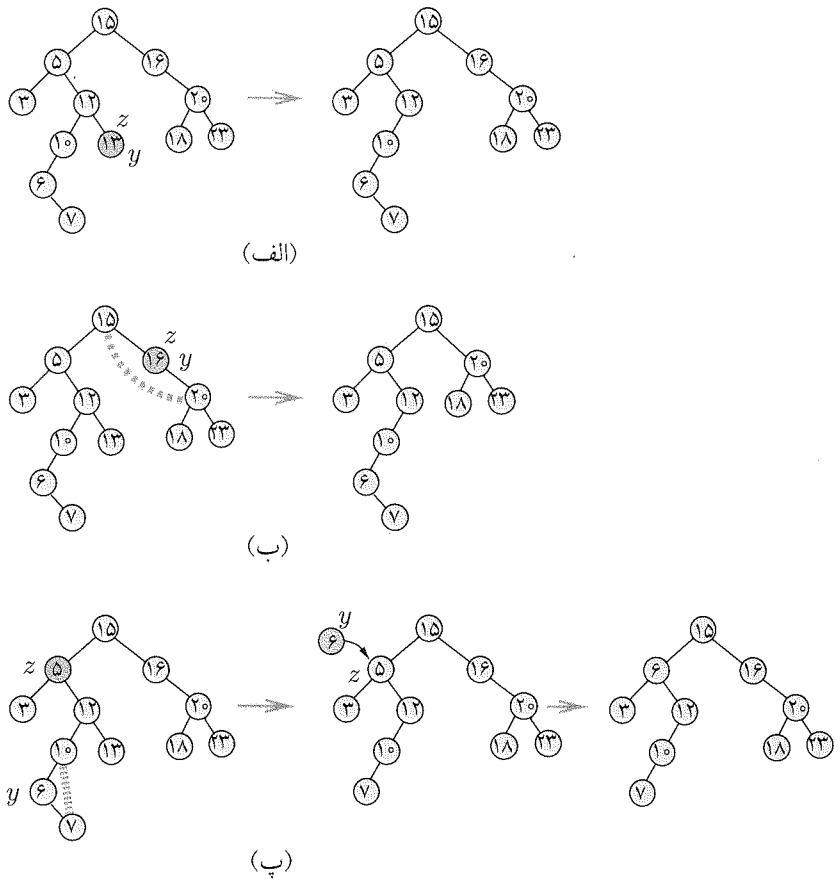


شکل ۴-۴۱ حالت‌های مختلف حذف عنصری با برچسب x از یک د.د.ج. عنصری که باید حذف شود، (الف) فرزند چپ ندارد، (ب) فرزند راست ندارد، یا (پ) هر دو فرزند را دارد. (راه‌حل برای هر حالت ذکر شده است.)

```
R-BST-DELETE (r, x)
    ▷ r یک گره (یا ریشه‌ی ) یک د.د.ج است ◁
    ▷ مهم: r باید یک پارامتر آدرسی باشد ◁
 1  if r = null
 2    then error tree is empty
 3  if x < label[r]
 4    then R-BST-DELETE(left[r], x)
 5  if x > label[r]
 6    then R-BST-DELETE(right[r], x)
 7  if x = label[r]
 8    then temp ← r
 9         if left[r] = null
10           then r ← right[r]
11                parent[r] ← parent[temp]
12                FREEE-NODE(temp)
13           else if right[r] = null
14                  then r ← left[r]
15                       parent[r] ← parent[temp]
16                       FREEE-NODE(temp)
17                  else label[r] ← BST-DELETEMIN (right[r])
```

الگوریتم غیربازگشتی برای حذف یک عنصر

رویه‌ی NR-BST-DELETE(T, z)، گره z را از د.د.ج T حذف می‌کند. اگر z برگ باشد یا فقط یک فرزند داشته باشد، حذف آن و مقداردهی جدید به تعدادی از اشاره‌گرهای پدر بسیار ساده است. این دو حالت در شکل‌های ۴-۴۲(الف) و (ب) نشان داده شده است. در هر دوی این حالت‌ها y برابر خود z خواهد شد و در سطرهای ۴ تا ۱۳، y عملاً حذف می‌شود، اشاره‌گرهایی که باید تغییر می‌کنند و در سطر ۱۰ اگر حذف y باعث تغییر ریشه‌ی درخت شود، این‌کار انجام می‌شود. اگر z دو فرزند داشته باشد نیز y برابر عنصر بعدی z می‌شود و چون با این حساب، y فرزند چپ ندارد، حذف آن در همان سطرهای ۴ تا ۱۳ انجام می‌شود. البته در حالت آخر (مانند الگوریتم بازگشتی) کلید عنصر y در عنصر z قرار داده می‌شود که در حالت (پ) شکل دیده می‌شود.



شکل ۴-۴۲ حالت‌های مختلف حذف غیر بازگشتی عنصر z از د.د.ج. (الف) اگر z برگ باشد. (ب) اگر z یک فرزند (در این مثال فرزند راست) داشته باشد. در این‌صورت y همان z خواهد بود و y حذف می‌شود. (پ) اگر z هر دو فرزند را داشته باشد. در این‌صورت، y که عنصر بعدی z است، حتماً فرزند چپ ندارد و به‌سادگی می‌توان آن‌را حذف کرد و سپس برچسب y در z قرار می‌گیرد.

```
NR-BST-DELETE (T, z)
 1  if left[z] = null or rigt[z] = null
 2     then y ← z
 3     else y ← BST-SUCCESSOR(z)
 4  if left[y] ≠ null
 5     then x ← left[y]
 6     else x ← right[y]
 7  if x ≠ null
 8     then parent[x] ← parent[y]
 9  if parent[y] = null
10     then root[T] ← x
11     else if y = left[parent[y]]
12             then left[parent[y]] ← x
13             else right[parent[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
```

قضیه‌ی ۴-۱ هزینه‌ی عملیات حذف، درج، جست‌وجو و دیگر عملیات بر روی یک د.د.ج T متناسب با ارتفاع T است.

۴-۵-۲ میانگین ارتفاع درخت دودویی جست‌وجو

فرض می‌کنیم که کلیدهای $a_1 < a_2 \cdots < a_n$ با یک ترتیب تصادفی و با فرض احتمال یک‌سان برای هر ترتیب وارد د.د.ج تهی T می‌شوند. ثابت می‌کنیم که میانگین ارتفاع T برابر $\mathcal{O}(\lg n)$ است. برای این‌کار سه متغیر تصادفی زیر را تعریف می‌کنیم:

- X_n: ارتفاع درختی که به‌صورت تصادفی برای n عنصر ایجاد می‌شود.
- $Y_n = 2^{X_n}$: برای سادگی کار از Y_n استفاده می‌کنیم.

- R_n: اندیس اولین عنصری که وارد T می‌شود و در ریشه قرار می‌گیرد.

اگر $R_n = i$، بدیهی است که زیردرخت چپ ریشه حتماً $i-1$ عنصر و زیردرخت راست آن $n-i$ عنصر خواهد داشت. همچنین روشن است که ارتفاع T یک واحد بیش‌تر از بیشینه‌ی ارتفاع‌های دو زیر درخت آن است. پس، $Y_n = ۲\max\{Y_{i-۱}, Y_{n-i}\}$ و پایه‌ی این رابطه‌ی بازگشتی $Y_۱ = ۱$ است.

سعی می‌کنیم رابطه‌ای برای مقدار میانگین Y_n، یا $E[Y_n]$ به‌دست آوریم و سپس $E[X_n]$ را که همان پاسخ مورد نظر است محاسبه کنیم. برای این‌کار، ابتدا حالتی را در نظر می‌گیریم که $R_n = i$ باشد و آن‌را با رخداد زیر نمایش می‌دهیم:

$$Z_{n,i} = \{R_n = i\}$$

$Z_{n,i} = ۱$ است اگر $R_n = i$ باشد، و گرنه مقدارش صفر است. طبق فرضِ احتمالات یک‌سان برای عناصر درخت، بدیهی است که

$$E[Z_{n,i}] = ۱/n. \qquad (۱۴-۴)$$

و داریم

$$Y_n = \sum_{i=۱}^{n} Z_{n,i}.(۲.\max\{Y_{i-۱}, Y_{n-i}\}). \qquad (۱۵-۴)$$

با استفاده از فرمول ۴-۱۴ و دیگر ویژگی‌های متغیرهای تصادفی، $E[Y_n]$ را مطابق زیر به‌دست می‌آوریم:

$$\begin{aligned}
E[Y_n] &= E\left[\sum_{i=۱}^{n} Z_{n,i}(۲.\max\{Y_{i-۱}, Y_{n-i}\})\right] \\
&= \sum_{i=۱}^{n} E[Z_{n,i}(۲.\max\{Y_{i-۱}, Y_{n-i}\})] \\
&= \sum_{i=۱}^{n} E[Z_{n,i}]E[(۲.\max\{Y_{i-۱}, Y_{n-i}\})] \\
&= \sum_{i=۱}^{n} \frac{۱}{n} E[(۲.\max\{Y_{i-۱}, Y_{n-i}\})] \\
&= \frac{۲}{n}\sum_{i=۱}^{n} E[(\max\{Y_{i-۱}, Y_{n-i}\})] \\
&\le \frac{۲}{n}\sum_{i=۱}^{n}(E[Y_{i-۱}] + E[Y_{n-i}]).
\end{aligned}$$

نامساوی آخر از این نتیجه‌گیری شده است که برای هر دو متغیر تصادفی نامنفی X و Y، نامعادله‌ی $E[\max(XmY)] \leq E[X] + E[Y]$ برقرار است. این نامساوی را می‌توان به‌صورت زیر هم نوشت:

$$E[Y_n] \leq \frac{۴}{n} \sum_{i=۱}^{n-۱} (E[Y_i]) \tag{۴-۱۶}$$

با استفاده از روش جای‌گذاری می‌توان نشان داد که حاصل رابطه‌ی بازگشتی ۴-۱۶ عبارت زیر است:

$$E[Y_n] \leq \frac{۱}{۴} \binom{n+۳}{۳} \tag{۴-۱۷}$$

برای اثبات این فرمول، در تمرین ۴-۱۴.۵ نشان می‌دهیم که:

$$\sum_{i=۰}^{n-۱} \binom{i+۳}{۳} = \binom{n+۳}{۴} \tag{۴-۱۸}$$

در ابتدا برای $Y_۱$ داریم

$$۱ = Y_۱ = E[Y_۱] \leq \frac{۱}{۴} \binom{۱+۳}{۳} = ۱$$

و این پایه‌ی استقرا است. برای حالت کلی داریم

$$\begin{aligned}
E[Y_n] &\leq \frac{۴}{n} \sum_{i=۰}^{n-۱} E[Y_i] \\
&= \frac{۴}{n} \sum_{i=۰}^{n-۱} \frac{۱}{۴} \binom{i+۳}{۳} \\
&= \frac{۱}{n} \sum_{i=۰}^{n-۱} \binom{i+۳}{۳} \\
&= \frac{۱}{n} \binom{n+۳}{۳} \\
&= \frac{۱}{n} \frac{(n+۳)!}{۴!(n-۱)!} \\
&= \frac{۱}{۴} \frac{(n+۳)!}{۳!n!} \\
&= \frac{۱}{۴} \binom{n+۳}{۳}.
\end{aligned}$$

از سوی دیگر می‌دانیم که

$$۲^{E[x_n]} \leq E[۲^{X_n}] = E[Y_n]$$

پس،

$$\begin{aligned} ۲^{E[x_n]} &\leq \frac{۱}{۴}\binom{n+۳}{۳} \\ &= \frac{۱}{۴}\cdot\frac{(n+۳)(n+۲)(n+۱)}{۶} \\ &= \frac{n^۳+۶n^۲+۱۱n+۶}{۲۴}. \end{aligned}$$

که اگر از طرفین لگاریتم بگیریم خواهیم داشت، $E[X_n] = O(\lg n)$، همان‌که می‌خواستیم ثابت کنیم.

قضیه‌ی ۴-۲ میانگین ارتفاع یک درخت دودویی جست‌وجو که به‌صورت تصادفی با n عنصر ساخته می‌شود برابر $O(\lg n)$ است.

تمرین‌های بخش ۴-۵

۴-۵.۱ چه تعداد درخت دودویی با n گره و با برچسب‌های ۱ تا n دارای ترتیب‌های یکسان در دو روش پس‌ترتیب و میان‌ترتیب هستند؟

۴-۵.۲ با ۲۵ عنصر چند د.د.ج با ارتفاع کمینه می‌توان ساخت؟

۴-۵.۳ آیا با داشتن دنباله‌ی میان‌ترتیب از عناصر یک د.د.ج می‌توان آن‌را به‌صورت یک‌تا ساخت؟ با پس‌ترتیب چه‌طور؟ با پیش‌ترتیب چه‌طور؟

۴-۵.۴ با توجه به این‌که مرتب‌سازی مقایسه‌ای n عنصر در بدترین حالت به $\Omega(n \lg n)$ زمان نیاز دارد، ثابت کنید هر الگوریتم مقایسه‌ای برای ساختن یک د.د.ج از لیستی از n کلید دل‌خواه در بدترین حالت به $\Omega(n \lg n)$ زمان نیاز دارد.

۴-۵.۵ فرض کنید جست‌وجو برای کلید k در یک د.د.ج به یک برگ ختم می‌شود. سه مجموعه‌ی زیر را در نظر بگیرید: A، کلیدهای سمت چپ مسیر جستجو؛ B، کلیدهای روی مسیر و C، کلیدهای سمت راست مسیر. ادعا می‌شود که به‌ازای هر سه کلید $a \in A$، $b \in B$ و $c \in C$، باید داشته باشیم $a \leq b \leq c$. یک مثال نقض برای این ادعا بیابید.

۴-۵.۶ نشان دهید اگر رأسی در یک د.د.ج دو فرزند داشته باشد، آن‌گاه عنصر بعدی آن، فرزند چپ و عنصر قبلی آن، فرزند راست ندارد.

۷.۵-۴ د.د.ج T با کلیدهای مجزا را در نظر بگیرید. نشان دهید اگر زیردرخت سمت راست گره x در T تهی و y عنصر بعدی x در درخت باشد، آن‌گاه y پایین‌ترین جد x است که فرزند چپ آن نیز جد x باشد. (توجه داشته باشید که هر گره، نیاکان خودش نیز هست.)

۸.۵-۴ پیمایش میان‌وندی یک د.د.ج n رأسی را می‌توان با پیدا کردن کوچک‌ترین عنصر درخت و سپس $n-1$ بار فراخوانی رویه‌ی TREE-SUCCESSOR پیاده‌سازی کرد. ثابت کنید این الگوریتم در زمان $\mathcal{O}(n)$ اجرا می‌شود.

۹.۵-۴ ثابت کنید در یک د.د.ج با ارتفاع h، بدون توجه به این‌که از کدام گره شروع کنیم، k بار فراخوانی متوالی رویه‌ی TREE-SUCCESSOR به زمان $\mathcal{O}(k+h)$ نیاز دارد.

۱۰.۵-۴ فرض کنید T یک د.د.ج با کلیدهای مجزا باشد و x یکی از برگ‌های این درخت و y پدر آن است. نشان دهید $key[y]$ یا کوچک‌ترین کلید بزرگ‌تر از $key[x]$ و یا بزرگ‌ترین کلید کوچک‌تر از $key[x]$ در T است.

۱۱.۵-۴ فرض کنید یک د.د.ج با درج متوالی مقادیر متفاوت در درخت ساخته شده باشد. نشان دهید تعداد گره‌هایی که برای جست‌وجوی یک مقدار در درخت دیده می‌شوند یکی بیش از تعداد گره‌های دیده‌شده در زمان درج آن مقدار در درخت است.

۱۲.۵-۴ آیا عمل حذف در د.د.ج جابه‌جایی‌پذیر است؟ به‌عبارت دیگر، آیا حذف x و سپس y از یک د.د.ج همان تأثیر حذف y و سپس x را دارد؟ جابه‌جایی‌پذیری را اثبات کنید یا برای آن یک مثال نقض بیاورید.

۱۳.۵-۴ در رویه‌ی BST-DELETE هنگامی که گره z دو فرزند داشته باشد، می‌توانیم به‌جای عنصر بعدی، عنصر قبلی آن را انتخاب کنیم. بعضی معتقدند که یک استراتژی منصفانه که در آن اولویت انتخاب عنصر بعدی مساوی با عنصر قبلی باشد، کارایی تجربی بهتری خواهد داشت. چگونه می‌توانیم BST-DELETE را تغییر دهیم تا به چنین استراتژی منصفانه‌ای برسیم؟

۱۴.۵-۴ فرمول ۴-۱۸ را ثابت کنید.

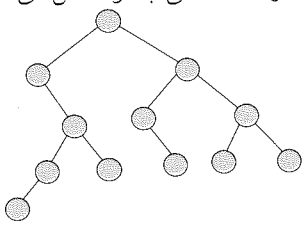
۱۵.۵-۴ یک د.د.ج بر روی n گره توصیف کنید به‌گونه‌ای که میانگین عمق هر گره در آن $\Theta(\lg n)$ و ارتفاع درخت $\omega(\lg n)$ باشد. یک کران بالای مجانبی برای ارتفاع یک د.د.ج با n گره که در آن میانگین عمق گره‌ها $\Theta(\lg n)$ است بیابید.

۱۶.۵-۴ نشان دهید که عبارت «د.د.ج با n گره که به تصادف و با احتمال مساوی از میان د.د.ج‌های با n گره انتخاب شده باشد» با عبارت «د.د.ج که به‌صورت تصادفی ساخته شده باشد» از نظر مفهومی متفاوت است. (راهنمایی: تمام حالت‌های ممکن برای $n = ۳$ را بنویسید.)

۱۷.۵-۴ چه تعداد از $۷!$ جای‌گشت از عناصر A تا G را اگر در یک د.د.ج تهی به‌ترتیب درج کنیم، همان درختی ایجاد می‌شود که دنباله‌ی درج‌های زیر (از چپ به راست) تولید می‌کند؟ توجیه کنید.

$$A, E, F, G, B, D, C$$

۱۸.۵-۴ تعداد حالات درج $a_{۱۲} > \cdots > a_۲ > a_۱$ در یک د.د.ج تهی را که منجر به تولید درخت زیر شود دقیقاً محاسبه کنید. مسئله در حالت کلی چگونه حل می‌شود؟



۱۹.۵-۴ می‌خواهیم عمل TREE-ENUMERATE (x, a, b) را بر روی زیردرخت دودویی جست‌وجو به ریشه‌ی x بنویسیم به‌طوری‌که تمام کلیدهایی را پیدا کند که مقدار آن‌ها بین a و b است. یک الگوریتم کارا از $O(h + m)$ برای این کار ارائه دهید (h ارتفاع درخت و m تعداد جواب است).

۲۰.۵-۴ n عنصر با کلیدهای مختلف را می‌توان با استفاده از یک د.د.ج به‌نام T که در ابتدا تهی است به صورت زیر مرتب کرد:

- عناصر را به‌ترتیبی که از ورودی دریافت می‌شوند در T درج کن
- T را به روش میان‌ترتیب پیمایش کن و عناصر را به ترتیب پیمایش در آرایه‌ی خروجی بنویس.

مرتبه‌ی زمان اجرای این الگوریتم به‌ترتیب در بهترین حالت، بدترین حالت، و حالت متوسط چیست؟

⋆ **۲۱.۵-۴** کلیدهای مساوی مشکلی را برای پیاده سازی د.د.ج ها به وجود می‌آورند.

الف) عمل‌کرد مجانبی NR-BST-INSERT هنگامی که برای درج n عنصر با کلیدهای برابر درون د.د.ج که در ابتدا تهی است استفاده شود چیست؟ برای عمل‌کرد بهتر، پیش‌نهاد می‌کنیم که قبل از سطر ۶، مساوی بودن x با $key[p]$ و قبل از سطر ۱۴ مساوی بودن x با $key[prep]$ بررسی شود. اگر شرط مساوی برقرار بود، یکی از استراتژی‌های زیر را به‌کار می‌گیریم: برای هر استراتژی عمل‌کرد مجانبی درج n عنصر با کلیدهای برابر، در د.د.ج در ابتدا تهی، را بیابید. (استراتژی‌ها برای سطر ۶ توضیح داده شده‌اند، با جای‌گذاری مناسب استراتژی‌های مربوط به سطر ۱۴ به‌دست می‌آیند.)

ب) پرچم بیتی $b[p]$ را برای گره p در نظر بگیرید و بسته به مقدار $b[p]$ که هرگاه p در زمان درج عنصری با کلیدی برابر p دیده می‌شود از «درست» به «نادرست» و برعکس عوض می‌شود، p را برابر $left[p]$ یا $right[p]$ قرار دهید.

پ) لیستی از گره‌های با کلید برابر را همراه با p نگه دارید و n را در این لیست درج کنید.

ت) p را به‌صورت تصادفی برابر $left[p]$ یا $right[p]$ قرار دهید. (عمل‌کرد بدترین حالت را اثبات کنید و به‌صورت غیر دقیق، عمل‌کرد حالت میانگین را به‌دست آورید.)

۴-۶ صف اولویت

«صف اولویت[۷۴]» داده‌ساختاری است که اَعمال درج، حذف کوچک‌ترین (یا بزرگ‌ترین) عنصر، کاهش و افزایش کلید یک عنصر از n عنصر را در زمانِ بهینه‌ی $\mathcal{O}(\lg n)$ و استفاده از حافظه‌ی بهینه‌ی خطی انجام می‌دهد.

دلیل این نام‌گذاری را می‌توان استفاده از آن در تخصیص یک منبع به تقاضاهای اولویت‌دار ذکر کرد. مثلاً در سیستم عامل که پردازه‌ها نسبت به هم اولویت متفاوت دارند در زمان‌های مختلف متقاضی استفاده از منبعی مانند پردازنده یا دیسک سخت هستند، یا بیماران اورژانسی که از یک درمانگاه خدمات دریافت می‌کنند. در چنین سیستمی، درخواست‌ها با اولویت‌هایشان (مثلاً اولویت پردازه در یک سیستم عامل یا بدی حال بیمار در یک اورژانس) وارد می‌شوند و خدمات به درخواستی داده می‌شود که اولویتش از بقیه بیش‌تر است. در چنین سیستمی، اَعمال درج (ورود به سیستم) و حذف بزرگ‌ترین عنصر (بدحال‌ترین بیمار، یا پردازه‌ای با بیش‌ترین اولویت) به‌طور دائم انجام می‌شود. بدیهی است که بخواهیم این کار با سرعت بالا صورت گیرد. اگر n عنصر در صف باشند، انجام این کار در زمانی کم‌تر از $\mathcal{O}(\lg n)$ امکان‌پذیر نیست.

به‌داده‌ساختار صف اولویت «درخت نیمه‌مرتب[۷۵]» هم می‌گوییم.

«هِرَم دودویی[۷۶]» داده‌ساختاری است که در پیاده‌سازی صف اولویت استفاده می‌شود. به این دلیل به صف اولویت، هرم هم گفته می‌شود. هرم به دو گونه تعریف می‌شود: «هرم کمینه[۷۷]» و «هرم بیشینه[۷۸]».

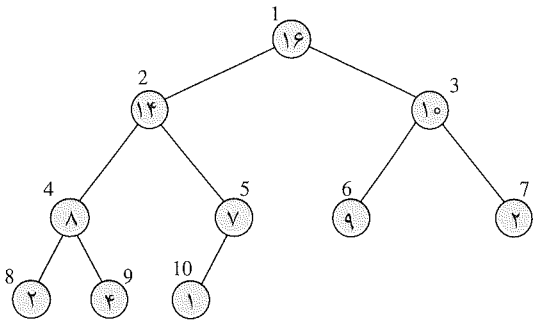
ما در این کتاب از واژه‌ی هرم برای صف اولویت استفاده می‌کنیم و گونه‌ی بیشینه‌ی آن‌را توصیف می‌کنیم.

۴-۶-۱ تعریف و ویژگی‌های هرم بیشینه

هرم بیشینه‌ی دودویی یک «درخت دودویی تقریباً کامل[۷۹]» است، به‌طوری که حداکثر یک گره فقط فرزند چپ دارد و بقیه‌ی گره‌های غیر برگ دو فرزند دارند. درخت متوازن است،

---

[۷۴] priority queue
[۷۵] partially ordered tree
[۷۶] binary heap
[۷۷] min-heap
[۷۸] max-heap
[۷۹] nearly complete binary tree



1	2	3	4	5	6	7	8	9	10
۱۶	۱۴	۱۰	۸	۷	۹	۲	۲	۴	۱

شکل ۴-۴۳ یک هرم بیشینه و پیاده‌سازی آن.

یعنی برگ‌ها حداکثر در دو سطح هستند. به‌علاوه، برگ‌های سطح آخر آن از سمت چپ در درخت ظاهر شده‌اند.[۸۰] ویژگی مهم دیگر این درخت آن است که کلید هر گره از کلیدهای فرزندانش کوچک‌تر نیست. توجه کنید که طبق این تعریف، در هرم می‌توان عناصری با کلید یکسان داشت. اگر رابطه‌ی کوچک‌تر و بزرگ‌تر را با هم عوض کنیم، هرم کمینه خواهد شد. شکل ۴-۴۳ یک نمونه از هرم بیشینه و پیاده‌سازی آن‌را نشان می‌دهد.

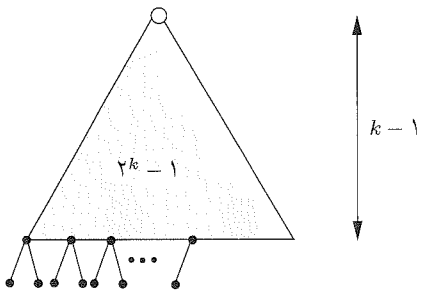
از تعریف هرم بیشینه می‌توان ویژگی‌های زیر را ملاحظه کرد:

لم ۴-۲ ریشه بزرگترین عنصر در هرم بیشینه است.

با استقرا بر روی ارتفاع درخت، به‌سادگی ثابت می‌شود که ریشه‌ی هر زیردرخت عنصرِ بیشینه‌ی آن زیردرخت است. پایه‌ی استقرا را یک گره تنها در نظر می‌گیریم.

لم ۴-۳ ارتفاع یک هرم با n عنصر برابر $h = \lfloor \lg n \rfloor$ است.

[۸۰] این محدودیت به دلیل سادگی در پیاده‌سازی آن در نظر گرفته شده است.



شکل ۴-۴۴ ارتفاع یک هرم با n عضو.

اثبات: طبق تعریف، اگر ارتفاع درخت h باشد، درخت تا سطح $h-1$ پر است. همچنین برای هر n، یک k موجود است به‌طوری‌که $2^k \le n < 2^{k+1}$، یا $n = 2^k + r - 1$ که در آن $1 \le r \le 2^k$. هرمی که با این تعداد عنصر می‌سازیم شامل یک درخت دودویی پر به ارتفاع $k-1$ و با $2^k - 1$ عنصر است که به آن r برگ در سطح k از سمت چپ اضافه شده است (شکل ۴-۴۴). واضح است که ارتفاع هرم برابر $\lfloor \lg n \rfloor$ می‌باشد. □

دیگر ویژگی‌های هرم بیشینه به‌شرح زیر است:

- kامین بزرگ‌ترین عنصر نمی‌تواند در سطحی بیش‌از k در درخت قرار گیرد.
- ساختار و شکل درخت برای هر تعداد ثابت عنصر از قبل مشخص است و مقدار کلیدهای عناصر در آن تأثیری ندارد.
- اَعمال درج، حذفِ بزرگ‌ترین عنصر و افزایش و کاهش کلید یک عنصر از مرتبه‌ی $O(\lg n)$ انجام می‌شود.

۴-۶-۲ پیاده‌سازی هرم بیشینه و انجام اَعمال مختلف

هرم طوری تعریف شده است که به‌سادگی می‌توان با استفاده از آرایه آن‌را پیاده‌سازی کرد. n عنصر را در آرایه‌ی n تایی $A[1 \cdots n]$ به‌صورت زیر پیاده‌سازی می‌کنیم:

- ریشه‌ی درخت در $A[1]$ قرار دارد،
- فرزند چپ عنصر iام در $A[2i]$ (اگر $2i \le n$) قرار دارد،

- فرزند راست عنصر iام در $A[2i+1]$ (اگر $2i+1 \leq n$) قرار دارد، و
- پدر عنصر iام در $A[\lfloor \frac{i}{2} \rfloor]$ ذخیره شده است ($i > 1$).

به‌این ترتیب، ساختار درختی هرم دودویی با اشاره‌گرهای فرزند چپ، راست و پدر، فقط با اندیس‌های آرایه پیاده‌سازی می‌شود.

با این پیاده‌سازی، انجام اَعمال زیر بسیار ساده است. البته فرض می‌کنیم که این رویه‌ها هنگامی فراخوانده می‌شوند که عنصر مورد نظر موجود باشد.

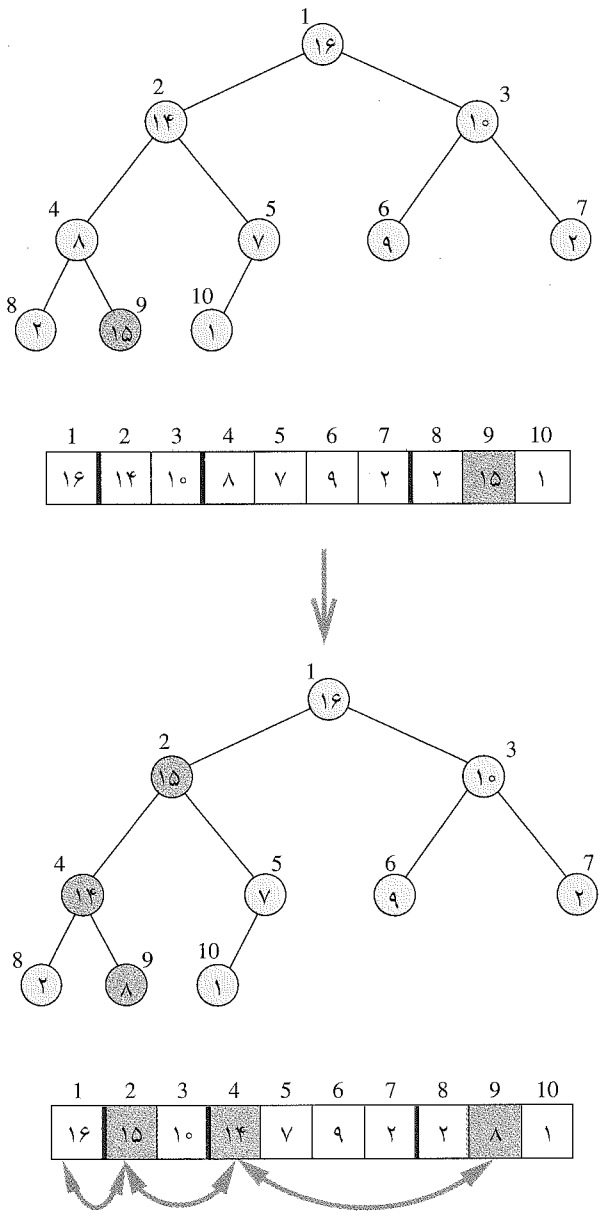
```
PARENT (i)
 1  return ⌊i/2⌋

LEFTCHILD (i)
 1  return 2i

RIGHTCHILD (i)
 1  return 2i + 1
```

افزایش کلید یک عنصر در هرم بیشینه

می‌خواهیم مقدار کلید یک عنصر (مثلاً عنصر i) را در یک هرم بیشینه افزایش دهیم تا کلیدش مقدار key را داشته باشد. برای این‌کار، key را با کلید پدرش مقایسه می‌کنیم. اگر key کم‌تر بود، هرم تغییری نمی‌کند؛ اگر key بیش‌تر بود، باید کلید این عنصر و پدرش عوض شود. با این‌کار کلید پدرش افزایش می‌یابد. پس همین الگوریتم را بر روی پدر اجرا می‌کنیم و آن‌را آن‌قدر تکرار می‌کنیم تا یا key از کلید پدرش کم‌تر باشد، یا به ریشه برسیم.

رویه‌ی MAX-HEAP-INCREASE-KEY(A, i, key) این‌کار را انجام می‌دهد. روشن است که این رویه حداکثر به‌اندازه‌ی ارتفاع درخت، یا $\lfloor \lg n \rfloor$ تعویض انجام می‌دهد.

شکل ۴-۴۵ هم جزئیات این الگوریتم را برای افزایش کلید یک عنصر از ۹ به ۱۵ نشان می‌دهد.



شکل ۴-۴۵ افزایش کلید ۹ به ۱۵ در هرم بیشینه.

```
MAX-HEAP-INCREASE-KEY (A, i, key)
1  if key < A[i]
2     then error new key is smaller than current key
3  A[i] ← key
4  while i > 1  and A[i] > A[PARENT(i)]
5     do swap(A[i], A[PARENT(i)] )
6        i ← PARENT(i)


HEAP-INSERT (A, x)
1  length[A] ← length[A] + 1
2  A[length[A]] ← −∞
3  HEAP-INCREASE-KEY(A, length[A], key)
```

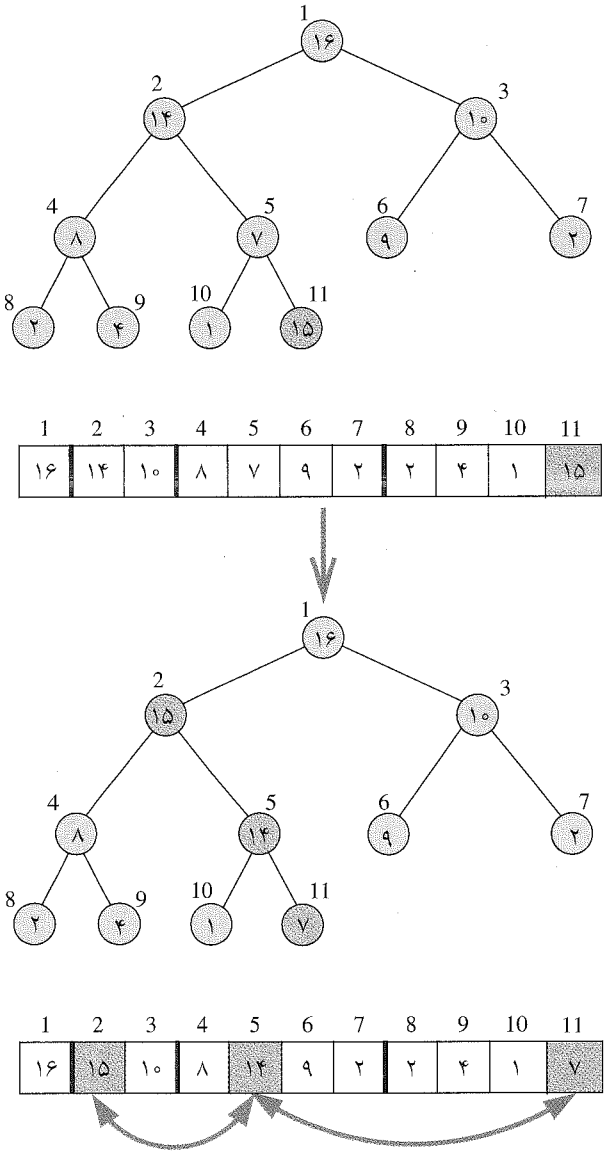
درج یک عنصر در هرم بیشینه

اگر هرم قبل از درج n عنصر داشته باشد، می‌دانیم که $A[n+1]$ مکانی است که عنصر جدید در هرم ظاهر می‌شود. ابتدا کلید این عنصر را برابر $-\infty$ قرار می‌دهیم تا ویژگی هرم حفظ شود. سپس این کلید را به مقدار *key* افزایش می‌دهیم. شکل ۴-۴۶ جزئیات درج کلید ۱۵ را در هرم شکل ۴-۴۳ نشان می‌دهد.

ساخت هرم بیشینه

برای ساخت هرم از n کلید که در یک آرایه‌ی A قرار دارند، از الگوریتم بازگشتی MAX-HEAPIFY استفاده می‌کنیم. این رویه با پارامتر (A, i) فراخوانده می‌شود. در ابتدای این فراخوانی فرض می‌کنیم که عناصر آرایه‌ی A از اندیس $i+1$ تا انتهای آرایه ($length[A]$) خاصیت هرم را دارند؛ یعنی عنصر k از فرزندانش (در صورت وجود) در درایه‌های $2k$ و $2k+1$ کوچک‌تر نیست. در انتهای این فراخوانی آرایه‌ی A از اندیس i تا انتهای آرایه به‌صورت هرم در می‌آید.

نحوه‌ی کار رویه‌ی MAX-HEAPIFY(A, i) به این صورت است که ابتدا اندیس بزرگ‌ترین فرزند i، یا *bigchild* را به‌دست می‌آوریم. اگر کلید i از کلید *bigchild* کم‌تر



شکل ۴-۴۶ درج عنصر ۱۵ در هرم شکل ۴-۴۳.

نباشد، کار به اتمام رسیده است. ولی اگر کلید i از کلید *bigchild* کم‌تر باشد، در آن‌صورت این دو کلید را با هم عوض می‌کنیم و با فراخوانی MAX-HEAPIFY $(A,\ bigchild)$ از اندیس *bigchild* تا آخر آرایه را به‌صورت هرم در می‌آوریم.

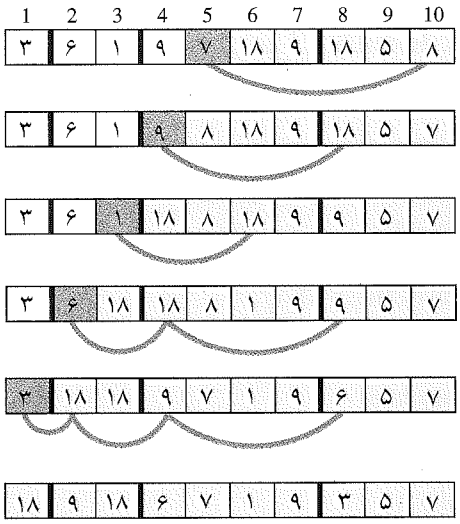
```
MAX-HEAPIFY (A, i)
 1  l ← LEFTCHILD (i)
 2  r ← RIGHTCHILD (i)
 3  if l ≤ length[A]  and A[l] > A[i]
 4     then bigchild← l
 5     else bigchild← i
 6  if r ≤ length[A]  and A[r] > A[bigchild]
 7     then bigchild← r
 8  if bigchild ≠ i
 9     then swap(A[i], A[bigchild])
10          MAX-HEAPIFY (A,bigchild)

BUILD-HEAP (A)
 1  for i ← ⌊length[A]/2⌋ downto 1
 2     do MAX-HEAPIFY(A, i)
```

رویه‌ی BUILD-HEAP(A) آرایه‌ی A را به‌صورت هرم بیشینه در می‌آورد. اگر A دارای n عنصر باشد، روشن است که عناصر واقع در اندیس $\lfloor \frac{n}{2} \rfloor + 1$ تا n برگ هستند و لذا نیازی نیست که رویه‌ی MAX-HEAPIFY برای آن‌ها فراخوانی شود. از این رو، سطر ۱ الگوریتم BUILD-HEAP از $\lfloor \frac{n}{2} \rfloor$ تا ۱ تکرار می‌شود. شکل ۴-۴۷ هم مراحل مختلف تبدیل یک آرایه به هرم بیشینه را نشان می‌دهد.

تحلیل MAX-HEAPIFY

اندازه‌ی زیردرخت‌های یک هرم با n عنصر حداکثر $\frac{2n}{3}$ است. این زمانی اتفاق می‌افتد که زیردرخت سمت چپ یک درخت پر به ارتفاع h و زیردرخت سمت راست یک درخت پر به ارتفاع $h-1$ باشد. در نتیجه $T(n)$ زمان اجرای MAX-HEAPIFY برابر است با

$$T(n) \leq T(\frac{2n}{3}) + \Theta(1).$$



شکل ۴-۴۷ مراحل مختلف تبدیل یک آرایه به هرم بیشینه.

طبق قضیه‌ی اصلی داریم، $T(n) = \mathcal{O}(\lg n)$. البته بدترین حالت هنگامی روی می‌دهد که رویه تا رسیدن به یک برگ ادامه پیدا کند. بنابراین $T(n) = \Omega(\lg n)$ و از این دو نتیجه می‌گیریم که $T(n) = \Theta(\lg n)$.

همچنین اگر $S(n)$ حداکثر تعداد تعویض‌ها در الگوریتم فوق باشد، داریم

$$\begin{aligned} S(n) &= \sum_{i=۲}^{n} [۱ + \lfloor \lg i - ۱ \rfloor] \\ &< n + \sum_{i=۲}^{n} \lg n = n + n \lg n. \end{aligned}$$

در بخش ۶-۳-۳ ثابت می‌کنیم که هزینه‌ی تبدیل یک آرایه‌ی n عضوی به هرم بیشینه $\mathcal{O}(n)$ است و نه $\mathcal{O}(n \lg n)$.

نکته‌ی ۴-۴ آیا می‌توان در حالت کلی آرایه‌ای تولید کرد که برای تبدیل آن به هرم بیشینه بیش‌ترین تعداد تعویض انجام شود؟

کاهش کلید یک عنصر در هرم بیشینه

اگر کلید عنصر i در یک هرم بیشینه کاهش یابد، کافی است رویه‌ی MAX-HEAPIFY(A, i) را فرابخوانیم.

حذف عنصر بیشینه در هرم بیشینه

با استفاده از MAX-HEAPIFY می‌توان رویه‌ی حذف عنصر بیشینه را نوشت. اولاً، می‌دانیم که عنصر بیشینه در ریشه است. ثانیاً، می‌دانیم که با حذف یک عنصر از درخت، سمت راست‌ترین برگ در آخرین سطح آن (که در آخرین درایه‌ی آرایه قرار دارد) حذف می‌شود. پس می‌توان کلید این برگ را در ریشه قرار داد و هرم را با فراخوانی MAX-HEAPIFY$(A, ۱)$ بازسازی کرد.

```
HEAP-DELETEMAX (A)
1  if length[A] < 1
2     then error heap underflow
3  max ← A[1]
4  A[1] ← A[length[A]]
5  length[A] ← length[A] − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

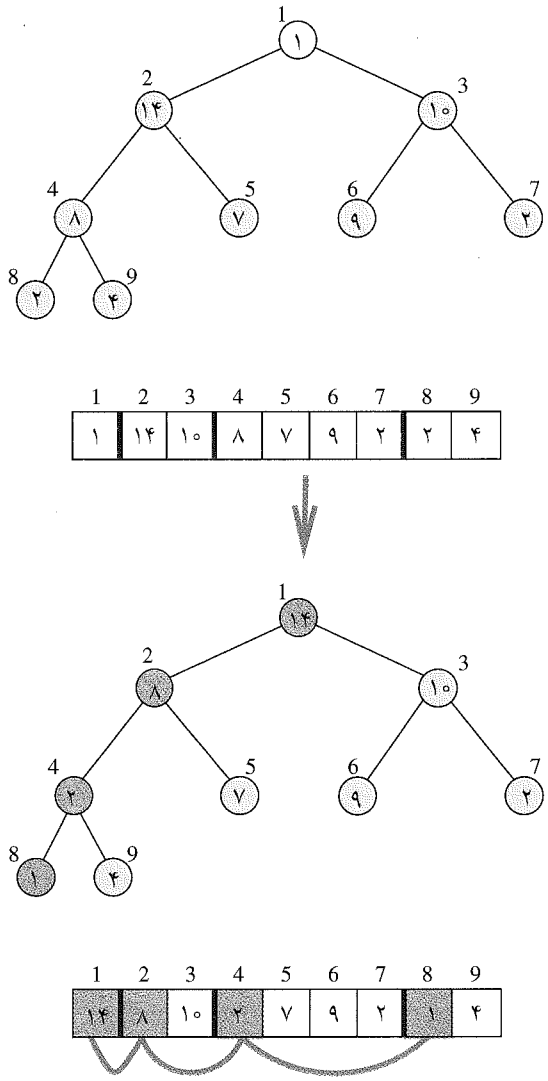
شکل ۴-۴۸ مراحل مختلف حذف عنصر بیشینه‌ی هرم شکل ۴-۴۳ را نشان می‌دهد.

مرتب‌سازی هرمی که در بخش ۶-۳-۳ خواهد آمد، از داده‌ساختار هرم استفاده می‌کند.

تمرین‌های بخش ۴-۶

۱.۶-۴ نشان دهید در نمایش آرایه‌ای یک هرم با n عضو، برگ‌ها با اعداد $\lfloor \frac{n}{۲} \rfloor + ۱$، $\lfloor \frac{n}{۲} \rfloor + ۱$ تا n اندیس‌گذاری شده‌اند.

۲.۶-۴ نشان دهید زمان اجرای MAX-HEAPIFY روی یک هرم در بدترین حالت $\Omega(\lg n)$ است. (راهنمایی: برای یک هرم با n گره، مقادیر هر گره را طوری تعیین کنید که باعث شود MAX-HEAPIFY بر روی هر گره در مسیر برگ تا ریشه به صورت بازگشتی فراخوانی شود.)



شکل ۴-۴۸ حذف عنصر بیشینه از هرم شکل ۴-۴۳.

۳.۶-۴ برای رویه‌های HEAP-MINIMUM، HEAP-EXTRACT-MIN، HEAP-DECREASE-KEY و MIN-HEAP-INSERT شبه‌کدی بنویسید که صف اولویت کمینه (min-priority queue) را با هرم کمینه پیاده‌سازی کند.

۴.۶-۴ فرض کنید یک هرم بیشینه حاوی همه‌ی اعداد ۱ تا ۱۰۲۳ است. چه تعداد از اعداد بیش‌تر از ۱۰۰۰ می‌توانند برگ باشند؟ حداکثر چه تعداد از این اعداد می‌توانند هم‌زمان برگ باشند؟

۵.۶-۴ فرض کنید که یک هرم بیشینه‌ی دودویی حاوی n عدد متمایز است. چهارمین بزرگ‌ترین عنصر این عناصر در کدام‌یک از درایه‌ها می‌تواند قرار بگیرد و در کدام درایه‌ها نمی‌تواند؟

۶.۶-۴ برای یک آرایه‌ی هرم بیشینه به‌طول n، اَعمال زیر را با چه مرتبه‌ای می‌توان به‌صورت کارا انجام داد؟ فقط با نماد O بنویسید. (توضیح لازم نیست.)

الف) جمع همه‌ی اعداد.

ب) جمع تعداد $\lg n$ بزرگ‌ترین عدد.

پ) جمع ۱۰ عدد بزرگ.

⋆ **۷.۶-۴** داده‌ساختار «صف اولویت میانه[۸۱]» شامل n عنصر مجزاست که اعمال زیر را می‌توان روی آن انجام داد:

- درج یک عنصر و حذف عنصر میانه، در بدترین حالت در $O(\lg n)$.
- MAKEMPQ: n عنصر داده شده را در زمان $O(n)$ به‌صورت MPQ در می‌آورد.

با استفاده از هرم، داده‌ساختار MPQ را طراحی کنید و نحوه‌ی انجام اعمال فوق را دقیقاً توضیح دهید و تحلیل نمایید.

۸.۶-۴ در مورد هرم بیشینه موارد زیر را ثابت کنید:

الف) حداکثر تعداد گره‌های با ارتفاع h در یک هرم با n عنصر برابر با $\lceil \frac{n}{2^{h+1}} \rceil$ است.

ب) فقط با استفاده از بند فوق ثابت کنید که الگوریتم ساختن هرم با n عنصر $O(n)$ است.

۹.۶-۴ یک رابطه‌ی بازگشتی برای تعداد هرم‌های متفاوتی که با $n = 2^k - 1$ عنصر متمایز ساخته می‌شوند، بنویسید.

⋆ **۱۰.۶-۴** یک هرم بیشینه با n عنصر متمایز از اعداد حقیقی و یک عدد x داده شده است. الگوریتمی از $O(k)$ ارائه دهید که مشخص کند آیا kامین کوچک‌ترین عنصر موجود در هرم از x کوچک‌تر است یا خیر. (لزومی ندارد که آن عنصر را حذف کنید.)

---

[۸۱] MPQ: Median-Priority-Queue

۴-۶.۱۱ فرض کنید H_1 و H_2 دو هرم بیشینه هستند که به‌صورت درختی (و نه با آرایه) پیاده‌سازی شده‌اند؛ بنابراین شما به ریشه‌ی هر هرم و به دو فرزند و پدر هر عنصر دست‌رسی دارید. الگوریتم MERGE-HEAP(H_1, H_2) را به‌طور کامل بنویسید تا در زمان $O(\lg n)$ این دو هرم را در هم ادغام کنید و آن‌ها را به یک هرم جدید تبدیل نمایید. در صورت نیاز، در الگوریتم خود می‌توانید از اعمال تعریف شده بر روی هرم‌ها استفاده کنید.

⋆ **۴-۶.۱۲** با این‌که بدترین حالت زمان اجرا برای مرتب‌سازی ادغامی $\Theta(n \lg n)$ و برای مرتب‌سازی درجی $\Theta(n^2)$ است، ضرایب ثابت کوچک در مرتب‌سازی درجی سبب شده‌اند که برای n های کوچک، مرتب‌سازی درجی سریع‌تر از مرتب‌سازی ادغامی عمل کند. بنابراین، این ایده مطرح می‌شود که در مرتب‌سازی ادغامی، هنگامی که زیرمسئله‌ها به اندازه‌ی کافی کوچک می‌شوند، از مرتب‌سازی درجی استفاده کنیم.

پس تغییری این‌چنین در مرتب‌سازی ادغامی در نظر بگیرید که در آن $\frac{n}{k}$ بخش از لیست هر یک به طول k، با استفاده از الگوریتم مرتب‌سازی درجی مرتب شود و سپس این بخش‌های مرتب با روش معمول در هم ادغام شوند. k مقداری است که از پیش تعیین می‌شود.

الف) نشان دهید که با استفاده از مرتب‌سازی درجی، $\frac{n}{k}$ بخش از لیست، هر یک به‌طول k، در بدترین حالت در زمان $\Theta(nk)$ مرتب می‌شوند.

ب) نشان دهید که ادغام این بخش‌ها در بدترین حالت در زمان $\Theta(n \lg(\frac{n}{k}))$ قابل انجام است.

پ) با فرض این‌که الگوریتم تغییریافته در بدترین حالت در زمان $\Theta(nk + n \lg(\frac{n}{k}))$ قابل انجام است، بزرگ‌ترین مقدار مجانبی k، که با نماد Θ و برحسب n نشان داده می‌شود، چیست که به‌ازای آن الگوریتم تغییریافته از نظر زمان اجرا به‌طور مجانبی مشابه با مرتب‌سازی ادغامی معمولی است؟

ت) k را در عمل چگونه باید انتخاب کنیم؟

۴-۶.۱۳ هرم‌های ادغام‌شدنی

یک هرم (کمینه‌ی) ادغام‌شدنی، از این اعمال پشتیبانی می‌کند: MAKE-HEAP (که یک هرم ادغام‌شدنی خالی ایجاد می‌کند)، INSERT، MINIMUM، EXTRACT-MIN و UNION. نشان دهید چگونه می‌توان یک هرم ادغام‌شدنی را با استفاده از یک لیست پیوندی در هر یک از حالات زیر پیاده‌سازی کرد. سعی کنید هر یک از اعمال را تا حد ممکن کارا کنید. زمان اجرای هر عمل را برحسب اندازه‌ی مجموعه‌ی داده‌ای پویایی که عمل روی آن انجام می‌شود تحلیل کنید.

الف) لیست‌ها مرتب هستند.

ب) لیست‌ها نامرتب هستند.

پ) لیست‌ها نامرتب هستند و مجموعه‌های داده‌ای پویا که عملیات روی آن‌ها انجام می‌شود مجزا می‌باشند.

تمرین‌های فصل ۴

۱.۴ فرض کنید $S = \{x | 0 \le x < N\}$ مجموعه‌ای از اعداد حقیقی بین صفر و عدد صحیح N باشد. همچنین فرض کنید I_j برای $j = 0 \cdots N-1$ زیرمجموعه‌هایی از S هستند به‌طوری‌که $I_j = \{x | j \le x < j+1\}$. به عبارت دیگر، I_j ها S را افراز می‌کنند، یعنی $S = \cap_{j=0}^{N-1} I_j$.

داده‌ساختاری برای ذخیره‌ی اعداد مجموعه‌ی S (و در نتیجه I_j ها) پیش‌نهاد کنید به‌طوری‌که بتوان اعمال زیر را با مرتبه‌های خواسته‌شده انجام داد:

- INSERT(x): x را به S و نیز به I_j مربوط (اگر وجود نداشته باشد) درج کن.
- DELETE (x): x را از S و نیز از I_j حذف کن.
- LIST (j) : همه‌ی عناصر موجود در I_j را بنویس.

داده‌ساختار شما باید طوری طراحی شود که دو عمل اول حداکثر در $\mathcal{O}(\lg(|S|))$ و عمل LIST حداکثر در $\mathcal{O}(|I_j|)$ انجام شود. جزئیات داده‌ساختار و نحوه‌ی انجام اعمال فوق را توضیح دهید.

⋆ **۲.۴** n نقطه با مختصات حقیقی بر روی محور x ها داده شده‌اند. می‌خواهیم با پیش‌پردازشی به‌اندازه‌ی $\mathcal{O}(n \lg n)$ و ساخت داده‌ساختاری مناسب، کاری کنیم که پرس‌وجوی RANGE-SEARCH(a, b) یعنی «یافتن همه‌ی نقاط بین بازه‌ی $[a, b]$ از اعداد حقیقی را در زمان $\mathcal{O}(\lg n + k)$ انجام دهد؛ k تعداد نقاط خروجی است. برای این‌کار یک د.د.ج پیشنهاد می‌کنیم که نقاط در n برگ آن قرار گیرند.

الف) توضیح دهید که چه اطلاعاتی باید در گره‌های میانی (غیر برگ) این درخت قرار داد تا جست‌وجو برای پیدا کردن یک عنصر، در زمانی متناسب با ارتفاع درخت انجام شود.

ب) توضیح دهید که چگونه می‌توان این درخت را ساخت به‌طوری‌که کم‌ترین ارتفاع را داشته باشد. این مقدار چقدر است؟

پ) الگوریتم RANGE-SEARCH(a, b) را به زبان شبه‌کد بنویسید و زمان اجرای آن را تحلیل کنید.

⋆ **۳.۴** داده‌ساختاری طراحی کنید تا بتواند n عنصر با اندیس‌های ۱ تا n را در فضایی با $\mathcal{O}(n)$ حافظه ذخیره کند و هر یک از اعمال زیر را در $\mathcal{O}(1)$ انجام دهد:

- INIT: داده‌ساختار را مقداردهی اولیه کن؛ در ابتدا عناصری ندارد و تهی است.
- SET(i, x): مقدار عنصر با اندیس i را برابر x قرار بده.
- GET(i): مقدار عنصر با اندیس i را برگردان. اگر عنصری با اندیس i قبلاً درج نشده باشد، جواب تهی است.

توجه کنید که نمی‌توان از امکانات زبان برنامه‌سازی برای این داده‌ساختار استفاده کرد. مثلاً نمی‌توان یک آرایه‌ی n تایی داشت که اعضای آن در ابتدا مقدار ۰ داشته باشند، زیرا عمل INIT آن $\mathcal{O}(n)$ است

و قابل قبول نیست. اگر شما از آرایه استفاده می‌کنید باید فرض کنید مقادیر اولیه‌ی درایه‌ها مشخص نیست. الگوریتم‌های شما باید صرف‌نظر از مقادیر اولیه همواره کار کند. دقت کنید که بجز اعمال فوق عمل دیگری (مثلاً حذف) بر روی داده‌ساختار تعریف نشده است.

⋆ **۴.۴** داده‌ساختاری برای پیاده‌سازی یک لیست مرتب از عناصر ارائه دهید تا بتواند اعمال زیر را در زمان‌های مشخص شده انجام دهد:

- ACCESS(k): kامین عنصر لیست را برگرداند.
- INSERT(k, x): x را به‌عنوان kامین عنصر لیست درج کند.
- REVERSE(i, j): ترتیب عناصر iام تا jام در لیست را تعویض کند.

اگر n تعداد عناصر لیست باشد، باید هزینه‌ی این اعمال به‌صورت سرشکنی $O(\lg n)$ شود.

⋆ **۵.۴** یک مجموعه‌ی پویای Q شامل عناصر x و کلید $key[x]$ است. این مجموعه اعمال زیر را انجام می‌دهد:

- INSERT(x, Q): درج عنصر x در Q
- $x \leftarrow$ EXTRACT-OLDEST (Q): عنصری از Q را که از بقیه زودتر درج‌شده است حذف می‌کند و آن‌را برمی‌گرداند.
- $x \leftarrow$ FIND-MAX (Q): عنصر x را پیدا می‌کند (ولی حذف نمی‌کند) که بیش‌ترین کلید $key[x]$ را در Q داشته باشد.

یک داده‌ساختار کارا برای مجموعه‌ی پویا طراحی کنید و بنویسید که هر یک از اعمال گفته شده چگونه انجام می‌شود و آن‌را تحلیل نمایید. راه‌حل کارا اعمال فوق را به‌صورت سرشکنی در $O(1)$ حل می‌کند. (بخشی از نمره به راه‌حل $O(\lg n)$ اختصاص داده شده است.)

۶.۴ ساخت هرم با استفاده از درج

رویه‌ی BUILD-MAX-HEAP را می توان به‌صورت دیگری با استفاده‌ی مکرر از HEAP-INSERT پیاده‌سازی کرد. پیاده‌سازی زیر را در نظر بگیرید:

```
BUILD-MAX-HEAP2 (A)
1  heap-size[A] ← 1
2  for i ← 2 to length[A]
3     do MAX-HEAP-INSERT (A, A[i])
```

الف) آیا همیشه با اجرای دو رویه‌ی BUILD-MAX-HEAP و BUILD-MAX-HEAP2 بر روی یک آرایه‌ی ورودی یک‌سان، هرم‌های مشابه به‌دست می‌آید؟ ادعای خود را ثابت کنید یا آن را با مثال نقض رد کنید.

ب) نشان دهید در بدترین حالت BUILD-MAX-HEAP2 به $\Theta(n \lg n)$ زمان برای ساخت یک هرم احتیاج دارد.

۷.۴ تحلیل هرم‌های d تایی

یک هرم d تایی مانند یک هرم دودویی است با این تفاوت که گره‌های غیر برگ، به جای ۲ فرزند d فرزند دارند.

الف) چگونه می‌توان یک هرم d تایی را با آرایه نمایش داد؟

ب) ارتفاع یک هرم d تایی بر حسب n و d چیست؟

پ) پیاده‌سازی مناسبی از EXTRACT-MAX برای هرم بیشینه‌ی d تایی ارائه دهید. زمان اجرای آن را بر حسب n و d تحلیل کنید.

ت) پیاده‌سازی مناسبی از درج برای هرم بیشینه‌ی d تایی ارائه دهید. زمان اجرای آن را برحسب n و d تحلیل کنید.

ث) پیاده‌سازی مناسبی از INCREASE-KEY(A, i, k) ارائه دهید که در آن ابتدا $A[i] \leftarrow \max(A[i], k)$ شود و سپس ساختار هرم بیشینه‌ی d تایی به‌شکل مناسبی به‌روز شود. زمان اجرای آن را بر حسب n و d تحلیل کنید.

⋆ ۸.۴ جداول یانگ

یک جدول یانگ[۸۲] ماتریسی به ابعاد $m \times n$ است که عناصر سطرهای آن از چپ به راست و عناصر ستون‌هایش از بالا به پایین مرتب شده‌اند. بعضی از درایه‌های جدول یانگ مقدار ∞ دارند که با آن‌ها همانند عناصر ناموجود برخورد می‌کنیم. بنابراین جدول یانگ می‌تواند برای نگه‌داری $r \leq mn$ عدد متناهی استفاده شود.

الف) یک جدول یانگ با ابعاد ۴×۴ را که شامل اعداد $\{۹, ۱۶, ۳, ۲, ۴, ۸, ۵, ۱۴, ۱۲\}$ باشد رسم کنید.

ب) در مورد این‌که در جدول یانگ Y با ابعاد $m \times n$، اگر $Y[۱, ۱] = \infty$، جدول خالی و اگر $Y[m, n] < \infty$، جدول پر است، بحث کنید.

پ) الگوریتمی با زمان اجرای $O(m + n)$ برای پیاده‌سازی EXTRACT-MIN بر روی جدول یانگ ارائه دهید. الگوریتم شما باید از رویه‌ی بازگشتی استفاده کند که مسئله‌ی با اندازه‌ی $m \times n$ را با حل بازگشتی زیرمسئله‌ی $(m - ۱) \times n$ یا $m \times (n - ۱)$ حل کند. (راهنمایی: از MAX-HEAPIFY کمک بگیرید. $T(p)$ را که $p = m + n$ است به‌عنوان زمان اجرای EXTRACT-MIN روی یک جدول یانگ با ابعاد $m \times n$ تعریف کنید. یک رابطه‌ی بازگشتی برای $T(p)$ به‌دست آورید و آن‌را حل کنید تا کران زمان $O(m + n)$ به‌دست آید.)

ت) نشان دهید چگونه یک عنصر در زمان $O(m + n)$ در یک جدول غیر پر درج می‌شود.

---

[۸۲] Young

ث) بدون استفاده از هیچ روش مرتب‌سازی دیگر، نشان دهید که چگونه می‌توان از یک جدول یانگ برای مرتب‌سازی $n^۲$ عنصر در زمان $\mathcal{O}(n^۳)$ استفاده کرد.

ج) الگوریتمی با زمان اجرای $\mathcal{O}(m+n)$ ارائه دهید که نشان دهد آیا عدد داده‌شده درجدول یانگ وجود دارد یا خیر.

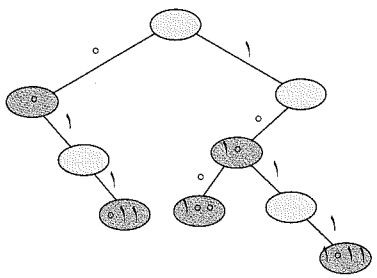
۹.۴ دو رشته‌ی $a = a_۰a_۱\cdots a_p$ و $b = b_۱b_۲\cdots b_q$ را در نظر بگیرید که هر یک از a_iها و b_iها عضو مجموعه‌ای مرتب از نویسه‌هاست. می‌گوییم رشته‌ی a به‌صورت الفبایی کوچک‌تر از b است اگر یکی از دو شرط زیر اتفاق بیفتد:

۱. عدد صحیح j وجود داشته باشد که $۰ \le j \le min(p,q)$، به‌طوری‌که $a_j < b_j$ و برای $i = ۰, ۱, \cdots, j-۱$ داشته باشیم $a_i = b_i$،

۲. $p < q$ و برای $i = ۰, ۱, \cdots, p$ داشته باشیم $a_i = b_i$.

به‌عنوان مثال، اگر a و b دو رشته از بیت‌ها باشند، آن‌گاه طبق قاعده‌ی اول (با قرار دادن $j = ۳$)، ۱۰۱۱۰ < ۱۰۱۰۰ و طبق قاعده‌ی دوم ۱۰۱۰۰۰ < ۱۰۱۰۰. این مانند ترتیبی است که در فرهنگ لغات به‌کار می‌رود.

داده‌ساختار «درخت مبنا[۸۳]»ی نشان داده شده در شکل ۴-۴۹، رشته بیت‌های ۱۰۰، ۰۱۱، ۱۰، ۱۰۱۱ و ۰ را نگه می‌دارد. هنگام جست‌وجو برای کلید $a = a_۰a_۱\cdots a_p$ در گره‌ای با عمق i اگر $a_i = ۰$ باشد، به سمت چپ و در غیر این صورت به راست می‌رویم. فرض کنید S مجموعه‌ای از رشته‌های دودویی مجزا با مجموع طول n باشد. نشان دهید چگونه با استفاده از درخت مبنا، S را به‌صورت الفبایی در زمان $\Theta(n)$ مرتب کنیم. برای مثال در شکل ۴-۴۹ خروجی مرتب‌سازی باید (از چپ‌به‌راست) رشته‌های {۰, ۰۱۱, ۱۰, ۱۰۰, ۱۰۱۱} باشد.



شکل ۴-۴۹ یک درخت مبنا که رشته‌های ۱۰۱۱، ۱۰، ۰۱۰، و ۰ را ذخیره می‌کند. اگر کلید گره‌ای در درخت نباشد، آن گره با رنگ تیره نشان داده شده است.

[۸۳] radix tree

پروژه‌های برنامه‌نویسی فصل ۴

۱ کار با درخت‌های کلی

برنامه‌ای بنویسید تا با دریافت مشخصات یک درخت کلی و مرتب، آن‌را به روش درخت دودویی معادل پیاده‌سازی کند و سپس با استفاده از چند رویه‌ی بازگشتی، آن درخت را به روش‌های پیش‌ترتیب، میان‌ترتیب، و پس‌ترتیب پیمایش نماید.
برای این‌کار:

الف) ورودی شامل چند درخت به‌صورت زیر است: ورودی مربوط به یک درخت با تعداد گره‌های آن درخت (حداکثر ۱۰۰) شروع شده و در هر یک از سطرهای بعد، برچسب یک گره و سپس برچسب‌های گره‌های فرزندان آن گره به‌ترتیب از چپ به راست نوشته می‌شوند. توجه کنید که گره‌های درخت لزوماً به‌ترتیب سطح آن‌ها در ورودی ظاهر نمی‌شوند. اطلاعات ورودی مربوط به‌چند درخت پشت سر هم قرار می‌گیرند. به‌عنوان مثال، ورودی زیر مربوط به درخت شکل ۴-۲۴ است:

```
13
a   b c d e
c   f g h
e   i
h   j k
i   l m
b
d
f
g
j
k
l
m
```

ب) فرض کنید درخت‌های ورودی حداکثر چهارتایی هستند. بنابراین لازم است رویه‌های CREATE2، CREATE3 و CREATE4 را بنویسید، به‌طوری‌که درخت حاصل نخ‌کشی‌شده باشد.

پ) رویه‌های بازگشتی زیر را بنویسید به‌طوری‌که با دریافت یک گره p از درخت T، گره بعدی آن را در یکی از پیمایش‌های پیش‌ترتیب، میان‌ترتیب، و پس‌ترتیب پیدا کند و بازگرداند.

- NEXT-PREORDER (T, p)

- NEXT-INORDER (T, p)
- NEXT-POSTORDER (T, p)

ت) با استفاده از رویه‌های فوق، ترتیب پیمایش درخت را به هر سه روش ذکر شده در خروجی بنویسید.

۲ ترجمه‌ی یک عبارت به زبان ماشین

برنامه‌ای بنویسید تا یک یا چند عبارت ورودی از نگارش میان‌وندی دریافت کند و اَعمال زیر را انجام دهد. عبارت ورودی از عمل‌گرهای +، -، *، /، ^ و ~ (~ به‌معنی تغییر علامت) استفاده می‌کند و ممکن است پرانتز داشته باشد. عمل‌وندها اعداد ثابتِ صحیح، اعشاری یا متغیرهای یک یا چند حرفی هستند. تعداد فاصله‌های خالی استفاده‌شده در ورودی متغیر است.

برنامه‌ی شما باید عبارت ورودی را دریافت کند و پس از اطمینان از درستی آن، آن‌را به‌صورت پس‌وندی و نیز پرانتزی کامل نگارش کند.

درانتها، برنامه‌ی شما باید عبارت ورودی را به‌دستورهای زبان ماشین یک ماشین فرضی و با یک انباشت‌گر[۸۴] ترجمه کند. فرض کنید دستورهای این ماشین به‌صورت زیر هستند:

LDA A	یعنی	$AC \leftarrow A$
STR A	یعنی	$A \leftarrow AC$
ADD A	یعنی	$AC \leftarrow AC + A$
SUB A	یعنی	$AC \leftarrow AC - A$
NEG	یعنی	$AC \leftarrow -AC$
DIV A	یعنی	$AC \leftarrow AC/A$
MUL A	یعنی	$AC \leftarrow AC * A$
PWR A	یعنی	$AC \leftarrow AC^A$

در این دستورها، A ممکن است یک عدد صحیح یا اعشاری، اسم یک متغیر، یا یک متغیر کمکیِ TEMPi باشد (در این حالت، i یک عدد صحیح و AC ثبات انباشت‌گر است).

مثال: یک خروجی نمونه برای یک عبارت ورودی به‌صورت زیر است:

```
Input expression: 12.5 + (sum - E)/A^~5^~0.17
Postfix:          12.5 sum E - A  5 ~ 0.17 ~ ^ ^ / +
Fully parenthesized infix:
          (12.5+((sum-E)/(A^((~5)^(~0.17)))))
Translation:
           LDA 0.17
           NEG
```

[۸۴] accumulator

```
STR TEMP1
LDA 5
NEG
PWR TEMP1
STR TEMP1
LDA A
PWR TEMP1
STR TEMP1
LDA SUM
SUB E
DIV TEMP1
ADD 12.5
```

در این برنامه، پُشته باید به‌صورت داده‌گونه‌ی انتزاعی تعریف شود. در گونه‌ی کاملاً پرانتزی پرانتزهای اضافی و واضح را نگذارید. سعی کنید از متغیرهای TEMP کم‌تری استفاده کنید. استفاده از کم‌ترین تعداد ممکن این متغیرها امکان‌پذیر است. (راهنمایی: حداقل تعداد متغیرهای کمکی برابر است با یک واحد کم‌تر از ارتفاع بزرگ‌ترین زیردرختِ پر در درخت عبارت.)

۳ چندجمله‌ای‌های چندمتغیره

این پروژه در ارتباط با لیستِ کلی است که در بخش ۴-۳-۲ مورد بحث قرار گرفت. قبل از انجام این پروژه، این بخش را بخوانید. در این‌جا داده‌ساختار پیش‌نهادی را کمی تغییر می‌دهیم و از شما می‌خواهیم که اعمال بیان‌شده را پیاده‌سازی کنید.

داده‌ساختار

یک چندجمله‌ای P برحسب متغیر z را درحالت کلی به‌صورت زیر می‌نویسیم:

$$P = g_0 + g_1 z^{e_1} + g_2 z^{e_2} + \ldots + g_n z^{e_n}$$

به‌طوری‌که $0 < e_1 < e_2 < \cdots < e_n$ و g_i خود یک چندجمله‌ای است که به‌صورت بازگشتی نوشته می‌شود و برحسب متغیری است که به‌صورت الفبایی از z کم‌تر است. مثلاً چند جمله‌ای

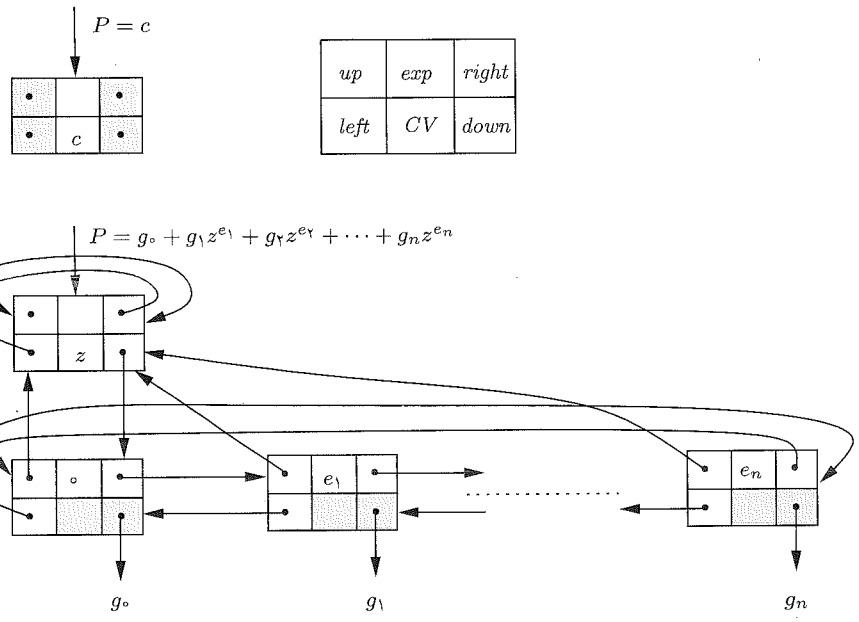
$$P = 3 + x^2 + xyz(xy + y - x) + z^3(1 - 3x)$$

را با این تعریف به‌صورت زیر می‌نویسیم:

$$P = ((3) + (1)x^2) + (((-1)x^2)y + ((1)x + (1)x^2)y^2)z + ((1) + (-3)x)z^3$$

در رابطه‌ی اخیر، هیچ‌گونه خلاصه‌سازی انجام نشده است و در آن هر عبارت واقع بین دو پرانتز باز و بسته‌ی متناظر، خود یک چندجمله‌ای است.

در شکل ۴-۵۰ یک داده‌ساختار برای پیاده‌سازی چندجمله‌ای P از نوع فوق پیش‌نهاد شده است. در این شکل، مؤلفه‌های استفاده‌شده برای هر گره نشان داده شده است: مؤلفه‌ی exp توان و CV یا متغیر چندجمله‌ای زیرین یا یک ثابت است . اگر $P = c$، چندجمله‌ای تنها یک گره دارد. اما در حالت کلی چندجمله‌ای به‌صورت بازگشتی و مطابق شکل تعریف می‌شود. برای سهولتِ کارهایی که می‌خواهیم انجام دهیم، لیست‌ها حلقوی و دوسویه در نظر گرفته شده‌اند.



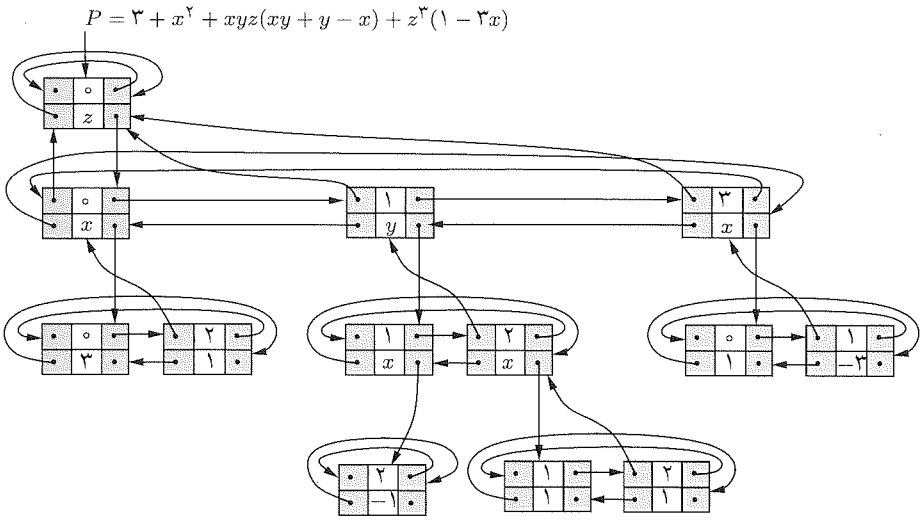
شکل ۴-۵۰ داده‌ساختار پیش‌نهادی برای یک چندجمله‌ای در حالت کلی. مؤلفه‌های هر گره در ساختار بالا و سمت راست نشان داده شده است.

شکل ۴-۵۱ داده‌ساختار مربوط به چندجمله‌ای زیر را نشان می‌دهد:

$$P = 3 + x^2 + xyz(xy + y - x) + z^3(1 - 3x)$$

اعمالی که شما باید پیاده‌سازی کنید

۱. READPOLY: یک چندجمله‌ای را از ورودی به‌صورت کلی دریافت کنید. در ورودی ضریب‌ها اعداد صحیح یا اعشاری هستند، ولی توان‌ها اعداد صحیح هستند. همچنین فرض



شکل ۴-۵۱ داده‌ساختار مربوط به چندجمله‌ای $P = ۳ + x^۲ + xyz(xy + y - x) + z^۳(۱ - ۳x)$.

کنید متغیرها همه تک‌حرفی هستند. مثلاً برنامه‌ی شما باید ورودی زیر را بپذیرد:

```
P = 3+ X^2 + XYZ(XY + Y -X) + (1 -3.0 X) Z^3
```

۲. READPOLY: رشته‌ی ورودی برای هر چندجمله‌ای را به‌صورت بازگشتی که بیان شد درآورد.

۳. CONVERT: از فرم بازگشتی ورودی و به‌کمک یک رویه‌ی بازگشتی، داده‌ساختار آن چندجمله‌ای را ایجاد نماید.

۴. PRINT: از داده‌ساختارِ یک چندجمله‌ای، فرم بازگشتی آن‌را در خروجی چاپ نماید. (در این‌جا پرانتزهای اضافی گذاشته نشوند.)

۵. ADD: با داشتن دو چندجمله‌ای P و Q، چندجمله‌ای مجموع این دو (یعنی $P + Q$) را به‌دست آورد.

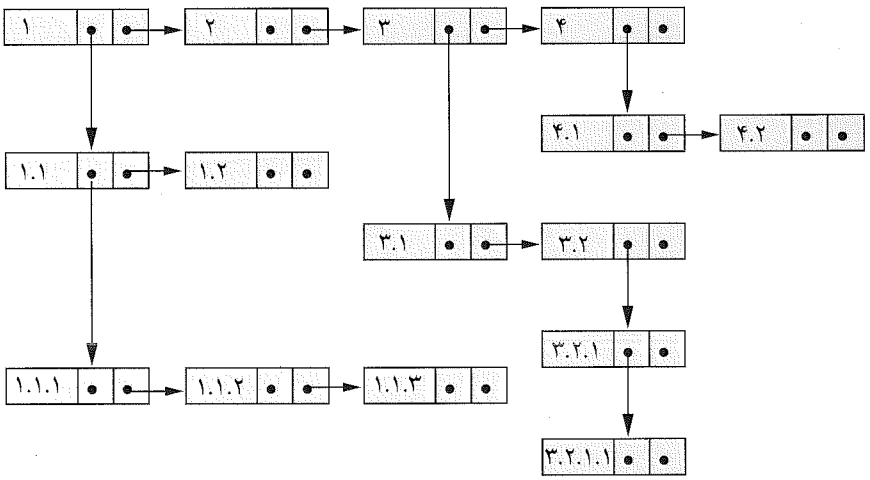
۶. MULT: با داشتن دو چندجمله‌ای P و Q، چندجمله‌ای $P * Q$ را به‌دست آورد.

در انجام عمل جمع و ضرب فرض بر این است که چندجمله‌ای‌های قبلی، دیگر مورد نیاز نیستند، بنابراین باید از عناصر آن‌ها در چندجمله‌ای حاصل استفاده شود.

اعمال PRINT و ADD را به‌صورت غیر بازگشتی و بدون استفاده از پشته پیاده‌سازی کنید. این کار با استفاده از اشاره‌گر *up* میسر خواهد بود.

۴ زباله‌روبی

در این پروژه می‌خواهیم با الگوریتم «زباله‌روبی» در کامپایلرها و کار با لیست‌ها و نیز اشاره‌گرهای اندیسی آشنا شویم. لیست‌هایی که با آن‌ها کار می‌کنیم دارای ساختاری مشابه شکل زیرند:



در این شکل برای راحتی توضیح، از برچسب‌های سلسله‌مراتبی برای رأس‌ها استفاده شده است. این برچسب‌ها در عمل پیاده‌سازی نمی‌شوند.

هر لیست را به‌صورت یک عبارت پرانتزی کامل هم نشان می‌دهیم. مثلاً شکل فوق متناظر با عبارت زیر است:

```
(((***)*)*(*((*)))(**))
```

در این نمایش، هر عنصر این لیست با یک عبارت تک قابل نمایش است. مثلاً گره‌های با برچسب‌های ۴.۱، ۳.۲.۱ و ۱.۱.۳ را به‌ترتیب با عبارت‌های *)***)، *)*)**) و ***)) نشان می‌دهیم.

برای این پروژه، یک آرایه‌ی بزرگ را با درایه‌هایی از مؤلفه‌های *label*، *tag*، *next* و *down* در نظر بگیرید. در این ساختار باید بتوانید چند لیست کلی از جمله لیست *avail* (برای فضای آزاد) را پیاده‌سازی کنید. همه‌ی اعمالی که در پایین گفته می‌شود باید با اشاره‌گر اندیسی پیاده‌سازی شوند. مؤلفه‌ی *tag* یک بیتی است که فقط برای زباله‌روبی استفاده می‌شود.

برنامه‌ی شما باید فرمان‌های زیر را اجرا کند:

۱. با فرمان *expression* = Li یک لیست به‌نام Li و متناظر با عبارت کاملاً پرانتزی *expression* (مطابق مثال فوق) ایجاد کند. این کار باید به‌وسیله‌ی یک رویه‌ی بازگشتی نوشته شود.

۲. با فرمان Make Li Child of Lj at *node-expr* لیست ایجاد شده‌ی Li به عنوان فرزند گره *node-expr* در لیست Lj (از طریق مؤلفه‌ی *down* آن گره) قرار گیرد. فرض می‌شود که قبل از انجام این کار مؤلفه‌ی *down* این گره مقدار ندارد.

۳. با فرمان Delete Li from *node-expre* مقدار اشاره‌گر *down* گره *node-expr* در لیست Li برابر **null** می‌شود. این فرمان گره‌ها را به فضای آزاد باز نمی‌گرداند.

۴. فرمان Print Li عبارت کاملاً پرانتزی لیست Li را در خروجی چاپ می‌کند. این کار باید به‌صورت غیر بازگشتی و فقط با استفاده از یک پشته نوشته شود.

۵. فرمان Garbage-Collection عناصر غیر قابل دسترسی در لیست‌ها را به انتهای فضای آزاد (لیست *avail*) اضافه می‌کند.
توجه مهم: این کار باید به‌صورت غیر بازگشتی و بدون استفاده از پشته انجام شود.

برنامه‌ای بنویسید که یک فایل ورودی حاوی تعداد زیادی از فرمان‌های فوق را دریافت کند و تک‌تک فرمان‌ها را انجام دهد. فرض کنید که همه‌ی فرمان‌ها درست‌اند.

۵ درخت فامیلی

هدف از این پروژه طراحی و ایجاد یک داده‌ساختار برای ذخیره‌ی کلیه‌ی روابط فامیلی ممکن بین تعدادی از افراد یک فامیل است. به این منظور پیش‌نهاد می‌شود که گره اصلی این داده‌ساختار دارای مؤلفه‌هایی مطابق با جدول ۴-۴ باشد.

در این داده‌ساختار برای هر فرد یک گره ایجاد می‌شود که اشاره‌گرهایی به پدر و مادر او دارد. اشاره‌گرهای prev-child-for-dad و prev-child-for-mom هم به‌ترتیب به فرزند قبلیِ پدر و مادر این گره اشاره می‌کنند. همچنین، اشاره‌گر latest-child به آخرین فرزند این فرد است. این گره حاوی اطلاعات فرد (شامل نام و شماره‌شناسنامه که یک کلید تک است) می‌باشد. latest-mar اشاره‌گری به گره آخرین ازدواج این فرد است. تاریخ تولد و مرگ هم دو مؤلفه‌ی دیگر این گره هستند. با اشاره‌گرهای left و right، گره‌های افراد به‌صورت یک درخت دودویی جست‌وجو در می‌آیند که جست‌وجو را ساده می‌کنند. کلید این درخت شماره‌ی شناسنامه‌هاست.

«ازدواج» هم گره‌ای است که با همین ساختار و به‌ازای هر ازدواج ایجاد می‌شود. مؤلفه‌ی type این گره برابر v فرض می‌شود. اشاره‌گرهای hus و wife به‌ترتیب به گره‌های شوهر و زن در این ازدواج اشاره می‌کنند. همچنین، prev-mar-hus و prev-mar-wife (در صورت وجود) به‌ترتیب به گره‌های ازدواج‌های قبلی شوهر و زن اشاره می‌کنند. تاریخ‌های ازدواج و یا جدایی (که ممکن است با طلاق یا مرگ یکی از دو زوج رخ دهد)، نیز در گره ازدواج ثبت می‌شود.

جدول ۴-۴ مؤلفه‌های گره اصلی در این داده‌ساختار

dad/hus	اشاره‌گر به پدر یا شوهر این گره
mom/wife	اشاره‌گر به مادر یا زن این گره
prev-child-for-dad	اشاره‌گر به فرزند قبلیِ پدر این گره
prev-child-for-mom	اشاره‌گر به فرزند قبلیِ مادر این گره
prev-mar-hus	اشاره‌گر به گره قبلیِ ازدواج شوهر
prev-mar-wife	اشاره‌گر به گره قبلیِ ازدواج زن
latest-child	اشاره گر به آخرین فرزند این گره
sex/type	نوع گره (مرد، زن، ازدواج)
SSN	شماره شناسنامه (تک است)
name	نام صاحب گره
birth/date-of-mar	تاریخ تولد یا تاریخ ازدواج
death/date-mar-end	تاریخ فوت یا جدایی (طلاق)
latest-mar	اشاره‌گر به آخرین گره ازدواج این فرد
left و right	اشاره‌گرهای مربوط به درخت دودویی جست‌وجو

این داده‌ساختار به‌صورت پویا و بر اساس وقوع رخ‌دادهای تولد، مرگ، ازدواج، و طلاق رشد می‌کند.

به‌عنوان مثال، داده‌ساختار شکل ۴-۵۲ پس از وقوع رخ‌دادهای ذکر شده در جدول ۴-۵ به‌وجود آمده است:

در این پروژه شما باید پرونده‌ای از رخ‌دادها (تولد، مرگ، ازدواج، طلاق) را بخوانید و داده‌ساختارها را بسازید. نحوه‌ی ورود و قالب اعداد ورودی را شما خود تعیین کنید.

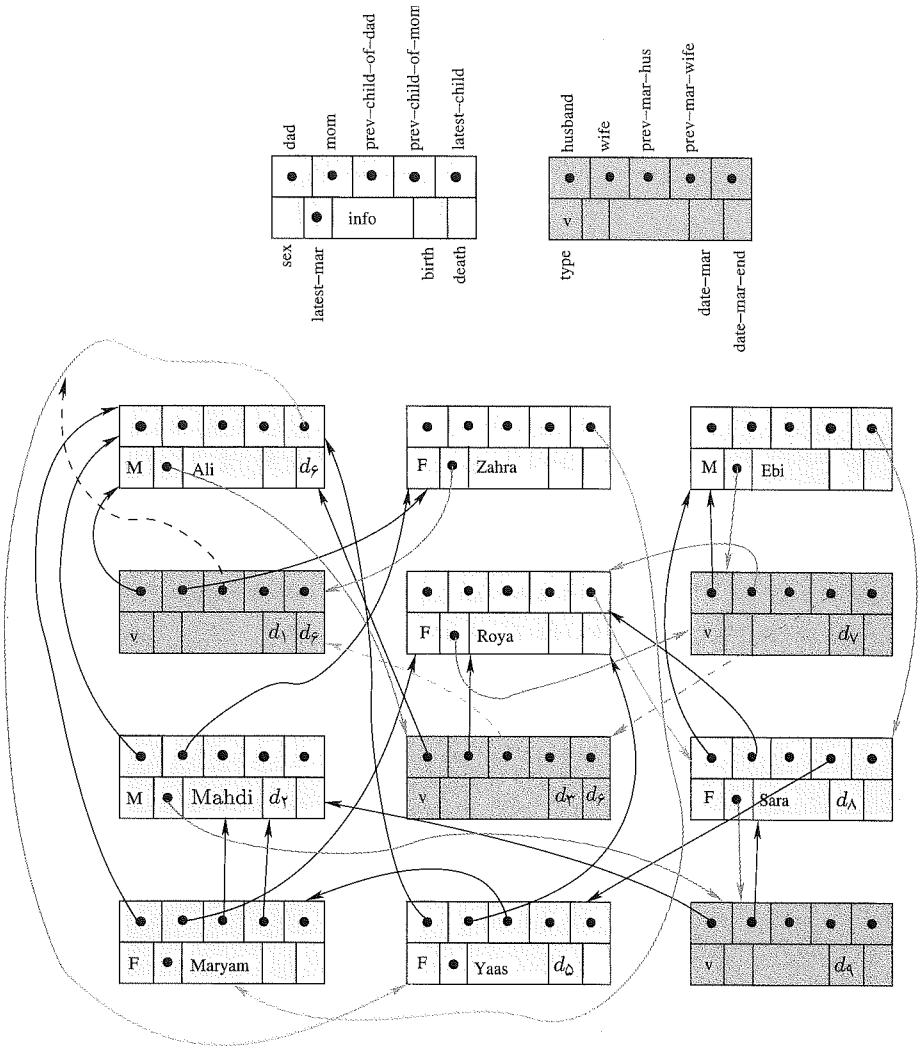
برنامه‌ی شما باید بتواند کلیه‌ی روابط ممکن فامیلی بین افراد موجود را تشخیص دهد.

نمونه‌هایی از این روابط به قرار زیرند: فرزند، همسر، برادر، خواهر، نتیجه، نبیره، ندیده (!)، پسر (دختر) عمه (خاله، عمو)، جد، باجناق، مادرزن ، پدرزن ، مادر (پدر) خوانده، و ...

این روابط محدود نیستند و شما هرچه بیش‌تر بتوانید روابط فامیلی را تشخیص دهید، کار بهتری انجام داده‌اید.

به‌صورت محاوره‌ای، سیستم شما باید بتواند با دادن اسامی A و B و رابطه‌ی R، به سؤالاتی مانند زیر جواب دهد:

- رابطه‌ی A با B چیست؟
- کلیه‌ی افرادی را که با A رابطه‌ی R دارند نام ببرید.



شکل ۴-۵۲ یک درخت فامیلی.

جدول ۴-۵ رخ‌دادهای مختلف در درخت فامیلی

تاریخ	رخ‌داد
d_1	علی با زهرا ازدواج می‌کند، این ازدواج دوم علی و ازدواج اول زهرا است.
d_2	مهدی فرزند علی و زهرا متولد می‌شود.
d_3	علی با رؤیا ازدواج می‌کند، سومین ازدواج علی و اولین ازدواج رؤیا.
d_4	مریم فرزند دوم علی و زهرا متولد می‌شود.
d_5	یاس فرزند علی و رؤیا متولد می‌شود.
d_6	علی فوت می‌کند.) :
d_7	ابراهیم (Ebi) با رؤیا ازدواج می‌کند، اولین ازدواج ابراهیم و دومین ازدواج رؤیا.
d_8	سارا فرزند ابراهیم و رؤیا متولد می‌شود.
d_9	مهدی با سارا ازدواج می‌کند.

علاوه بر مطالب فوق آن‌چه که به نظرتان جالب می‌رسد و بتوانید در پروژه‌ی خود اضافه کنید قابل قبول و تشویق است.

مرجع:

F. Mavaddat, and B. Parhami, *A data Structure for Family Relations,* The Computer Journal Vol. 22, No. 2, PP. 11O-113, May 1979.

فصل

۵

درهم‌سازی

در بسیاری از کاربردها نیاز است که بر روی مجموعه‌ی پویایی از داده‌ها اعمال فرهنگ داده‌ای، یعنی جست‌وجو، درج و حذف انجام شود. مجموعه‌ای را پویا می‌گوییم که تعداد عناصر آن در طول زمان تغییر کنند. مثلاً، چنان‌چه در فصل ۴ هم گفتیم، یک کامپایلر نیاز دارد کلمات کلیدی خود را در یک جدول نمادها سازمان‌دهی کند تا بتواند در ترجمه‌ی یک برنامه، بررسی کند که آیا تک‌تکِ کلمات استفاده شده در آن برنامه در جدول نمادها هست یا خیر.

تا کنون ما با داده‌ساختارهای ساده (مثل لیست‌ها یا درخت دودویی جست‌وجو) آشنا شدیم که در آن‌ها هزینه‌ی اعمال فرهنگ داده‌ای بر روی مجموعه‌ای از n عنصر در بدترین حالت $\mathcal{O}(n)$ و بسیار کند است. بعداً در فصل ۷ خواهیم دید که درخت دودویی جست‌وجو را می‌توان بهبود بخشید تا اعمال بیان‌شده را در بدترین حالت در $\mathcal{O}(\lg n)$ انجام دهد.

در این بخش، با داده‌ساختاری به‌نام جدول درهم‌سازی[۱] و روش درهم‌سازی[۲] آشنا می‌شویم که هم پیاده‌سازی آن نسبتاً ساده است و هم با طراحی درست، هزینه‌ی هر کدام از این اعمال، در بدترین حالت و نیز در حالت میانگین می‌تواند $\mathcal{O}(۱)$ شود.

---

[۱] hashing table
[۲] hashing

فرض می‌کنیم مجموعه‌ی عناصر حداکثر شامل n عنصر است. کلید عنصر x را $keyA[x]$ می‌نامیم و می‌دانیم که کلیدها متمایز و عناصری از مجموعه‌ای به‌نام K هستند. همچنین می‌دانیم که $K \subseteq U$ که در آن U مجموعه‌ی جهانی مقادیر کلیدهاست.

۵-۱ جدول آدرس‌دهی مستقیم

در «آدرس‌دهی مستقیم[۳]» هر عنصر را برحسب مقدار کلیدش مستقیماً در یک آرایه‌ی T ذخیره می‌کنیم. در کاربردی که از این روش استفاده می‌کند، فرض می‌شود که مجموعه‌ی جهانی U اندازه‌ای نسبتاً کوچک دارد. همچنین $K \subseteq \{0, 1, \cdots, m-1\}$، که در آن مقدار m خیلی بزرگ نیست. برای نمایش مجموعه‌ی پویایی از این عناصر، از آرایه‌ی $T[0 \cdots m-1]$ به نام «جدول آدرس‌دهی مستقیم[۴]» استفاده می‌کنیم. در درایه‌ی i ام آرایه‌ی T تنها عنصری با کلید i می‌تواند جای گیرد و درایه‌های دیگر خالی یا **null** هستند. یعنی،

$$T[k] = \begin{cases} x & \text{اگر } k \in K \text{ و } key[x] = k \\ \textbf{null} & \text{وگرنه} \end{cases}$$

به‌این ترتیب، اعمال فرهنگ داده‌ای در جدول آدرس‌دهی مستقیم به‌سادگی و به‌صورت زیر قابل انجام است.

```
DIRECT-ADDRESS-SEARCH (T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT (T, x)
1  T[key(x)] ← x

DIRECT-ADDRESS-DELETE (T, x)
1  T[key(x)] ← null
```

[۳] direct address
[۴] direct-address table

بدیهی است که هر یک از این اعمال در $\mathcal{O}(1)$ انجام می‌شود.

مهم‌ترین مشکل آدرس دهی مستقیم آن است که m می‌تواند بسیار بزرگ باشد. مثلاً اگر کلید ۶۴ بیتی باشد، می‌تواند

۱۸,۴۴۶,۷۴۴,۰۷۳,۷۰۹,۵۵۱,۶۱۶

مقدار متفاوت داشته باشد! و می‌دانیم که ساخت یک جدول با این اندازه ناممکن است. یا اگر کلید رشته باشد، تعداد حالت‌های آن می‌تواند حتی بیش‌تر از این باشد.

برای حل این مشکل از جدول درهم‌سازی استفاده می‌کنیم.

تمرین‌های بخش ۵-۱

۵-۱.۱ فرض کنید مجموعه‌ی پویای S از متغیرها با آدرس‌دهی مستقیم در جدول T به‌طول m ذخیره شده است. رویه‌ای پیشنهاد کنید تا بزرگترین عنصر S را پیدا کند. زمان اجرای این رویه در بدترین‌حالت چیست؟

۵-۱.۲ آرایه‌ای به‌طول m از بیت‌ها (صفر و یک) را در نظر بگیرید. واضح است که این آرایه در مقایسه با آرایه‌ای از اشاره‌گرها به‌همین طول، حافظه‌ی کم‌تری مصرف می‌کند. چگونه می‌توان از آرایه‌ای از بیت‌ها برای نمایش مجموعه‌ی پویایی از عناصر متمایز و بدون اطلاعات اضافی استفاده کرد؟ اعمال فرهنگ داده‌ای باید در زمان $\mathcal{O}(1)$ اجرا شوند.

۵-۱.۳ روشی برای پیاده‌سازی یک جدول آدرس‌دهی مستقیم ارائه کنید که در آن کلیدهای ذخیره شده لزوماً متمایز نیستند و عناصر می‌توانند اطلاعات اضافه هم داشته باشند. اعمال فرهنگ داده‌ای (درج، حذف و جست‌وجو) باید هر یک در زمان $\mathcal{O}(1)$ اجرا شود. (فراموش نکنید که عمل حذف، به‌عنوان ورودی یک اشاره‌گر به عنصر مورد نظر می‌گیرد نه مقدار آن عنصر را.)

۵-۱.۴ می‌خواهیم یک فرهنگ لغات را با آدرس‌دهی مستقیم بر روی یک آرایه‌ی بزرگ پیاده‌سازی کنیم. در آغاز، درایه‌های آرایه دارای مقادیر مشخصی نیستند و مقداردهی اولیه‌ی کل آرایه هم به‌دلیل اندازه‌ی بزرگ آن غیر عملی است. روشی برای پیاده‌سازی این فرهنگ ارائه کنید، به‌طوری‌که هر عنصر ذخیره‌شده فضایی از مرتبه‌ی $\mathcal{O}(1)$ اشغال کند. هرکدام از اعمال درج، حذف و جست‌وجو باید در زمان $\mathcal{O}(1)$ انجام گیرد و مقداردهی اولیه‌ی داده‌ساختار نیز باید به‌زمان $\mathcal{O}(1)$ نیاز داشته باشد. (راهنمایی: برای بررسی این‌که مقدار یکی از درایه‌ها در آرایه‌ی بزرگ معتبر هست یا خیر، از یک پشته‌ی اضافه استفاده کنید که اندازه‌ی آن برابر تعداد عناصری باشد که واقعاً در فرهنگ لغات ذخیره کرده‌ایم.)

۵-۲ جدول‌های درهم‌سازی

برای حل مشکلی که در آدرس‌دهی مستقیم گفتیم، جدول را به‌اندازه‌ی قابل پیاده‌سازی m در نظر می‌گیریم و عنصر x با کلید $k = key[x]$ را در درایه‌ی $h(k)$ ذخیره می‌کنیم. در این‌جا h «تابع درهم‌سازی[۵]» است و به‌ازای یک کلید k، مقدار $h(k)$ یک عدد صحیح بین ۰ و $m-1$ است. یعنی

$$h : U \to \{0, 1, \cdots, m-1\}.$$

چون معمولاً $m < |U|$، ممکن است بیش از یک عنصر به یک درایه‌ی خاص نگاشته شوند. به این مشکل «برخورد[۶]» می‌گوییم که باید آن‌را به‌روشی برطرف کنیم. شکل ۵-۱ یک جدول درهم‌سازی، مجموعه‌ی جهانی U، مجموعه‌ی کلیدهای K و نیز تابع h را برای ۶ کلید نشان می‌دهد. در این مثال، کلیدهای k_5 و k_6 برخورد دارند چون $h(k_5) = h(k_6)$.

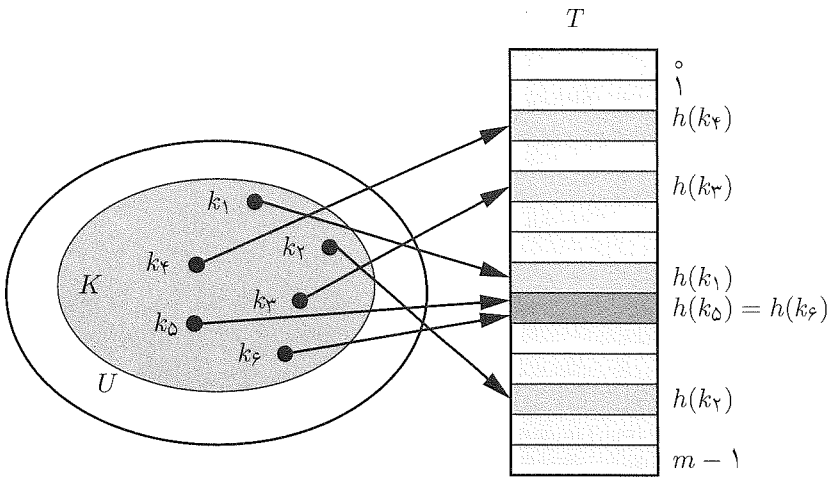
بدیهی است که می‌خواهیم تابع درهم‌سازی طوری انتخاب شود که اصلاً برخوردی صورت نگیرد. اما این در اغلب موارد امکان‌پذیر نیست. به‌ناچار، باید به‌روشی این مشکل را حل کنیم. یک راه، انتخاب یک تابع درهم‌سازی است که به‌صورت تصادفی عناصر را در درایه‌های مختلف قرار دهد (البته باید یک کلید خاص را همیشه به یک درایه‌ی مشخص نگاشت کند)، به این امید که تعداد برخوردها کم شود. البته طراحی چنین تابعی ممکن است کار ساده‌ای نباشد، و در هر حال شاهد برخورد خواهیم بود. در بخش‌های بعد، با استفاده از روش‌های زنجیره‌ای یا آدرس‌دهی باز سعی می‌کنیم مشکل برخورد را حل کنیم.

۵-۳ روش زنجیره‌ای برای حل برخورد

در روش «زنجیره‌ای[۷]» عناصری را که به‌وسیله‌ی تابع درهم‌سازی به یک درایه از جدول درهم‌سازی، مثلاً aS، نگاشته می‌شوند به‌صورت یک لیست خطی یک‌سویه در می‌آوریم و آدرس شروع آن را در همان درایه، یعنی $T[i]$ قرار می‌دهیم. شکل ۵-۲ یکی از این لیست‌ها را در جدول نشان می‌دهد که در آن، تابع درهم‌سازی برای کلیدهای ۴۹، ۸۶ و ۵۹ مقدار یک‌سان i را برمی‌گرداند. در حالت کلی، عناصر با کلیدهای مختلف k که مقدار $h(k)$ آن‌ها برابر است در لیست پیوندی $T[h(k)]$ قرار می‌گیرند. درایه‌های خالی جدول هم

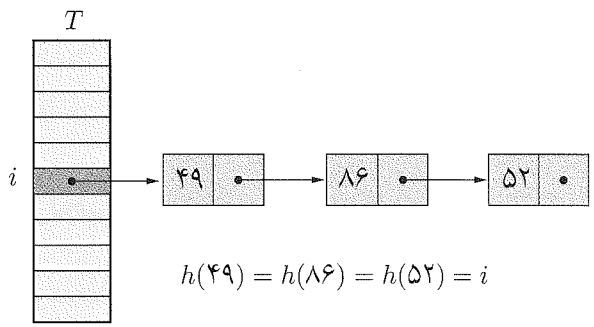
---

[۵] hash function
[۶] collision
[۷] chaining



شکل ۵-۱ یک جدول درهم‌سازی با تابع درهم‌سازی h. کلیدهای $k_۵$ و $k_۶$ برخورد دارند چون به یک درایه نگاشته شده‌اند.

لیست‌های تهی هستند. به چنین جدولی، «جدول درهم‌سازی با روش زنجیره‌ای» می‌گوییم. شکل ۵-۲ مثالی از این روش است.



شکل ۵-۲ روش زنجیره‌ای برای حل مشکل برخورد.

در این روش، اَعمال جست‌وجو، درج و حذف به‌صورت زیر و با استفاده از دستورات بیان‌شده در لیست‌ها پیاده‌سازی می‌شوند:

```
CHAINED-HASH-SEARCH (T, k)
1  search an element whith key k in list T[h(k)]

CHAINED-HASH-INSERT (T, x)
1  insert x at the head of list T[h(key[x])]

CHAINED-HASH-DELETE (T, x)
1  delete x from the list T[h(key[x])]
```

روشن است که درج در زمان $\mathcal{O}(1)$ انجام می‌شود، اما عمل جست‌وجو و حذف (که ابتدا باید با انجام یک جست‌وجو عنصر مورد نظر را یافت) ممکن است در بدترین حالت هزینه‌ای متناسب با طول لیست داشته باشد. می‌توان حالت‌های بدی را تصور کرد که تعداد $\mathcal{O}(n)$ عنصر با هم برخورد داشته باشند. بنابراین، هزینه‌ی جست‌وجو و حذف ممکن است در بدترین حالت هزینه‌ی $\mathcal{O}(n)$ داشته باشد، که خیلی زیاد است و قابل قبول نیست. البته هزینه‌ی انجام محاسبه‌ی تابع درهم‌سازی هم باید به آن اضافه شود.

در بخش بعد، نشان می‌دهیم که اگر تابع درهم‌سازی به‌صورت یک‌نوا عناصر را در جدول درهم‌سازی توزیع کند، هزینه‌ی میانگین جست‌وجو $\mathcal{O}(1)$ خواهد بود.

تحلیل روش زنجیره‌ای

برای تحلیل دقیق درهم‌سازی به روش زنجیره‌ای، فرض می‌کنیم که جدول درهم‌سازی T به‌اندازه‌ی m، در خود n عنصر را ذخیره می‌کند. پارامتر زیر را به این منظور تعریف می‌کنیم:

تعریف ۵-۱ α: ضریب بارگذاری[۸] T، یعنی $\alpha = \frac{n}{m}$.

[۸] load factor

برای تحلیل حالت میانگین، فرض می‌کنیم که تابع درهم‌سازی ساده و یک‌نوا[۹] است. یعنی هر عنصر با احتمال مساوی در هر یک از درایه‌های جدول قرار می‌گیرد. بنابراین اگر n_j اندازه‌ی لیست واقع در درایه‌ی $T[j]$ یا به صورت ساده‌تر T_j (برای $j = ۰ \cdots m-۱$) باشد، داریم $n = n_۰ + n_۱ + \cdots + n_{m-۱}$. مقدار میانگین n_j برابر است با $\alpha = \frac{n}{m}$. همچنین، فرض می‌کنیم که مقدار $h(k)$ را می‌توان در $\mathcal{O}(۱)$ حساب کرد. پس، زمان مورد نیاز برای جست‌وجوی یک عنصر با کلید k به‌صورت خطی به $n_{h(k)}$ بستگی دارد.

یک عمل جست‌وجو یا موفق است (یعنی، عنصر مورد جست‌وجو موجود است) یا ناموفق (یعنی، عنصر در جدول نیست). بنابراین، باید هزینه‌ی میانگین را برای هرکدام از این دو نوع جست‌وجو محاسبه کنیم.

قضیه‌ی ۵-۱ در یک جدول درهم‌سازی با روش زنجیره‌ای، با فرض استفاده از تابع درهم‌سازی ساده‌ی یک‌نوا، یک جست‌وجوی ناموفق به زمان میانگین $\mathcal{O}(۱ + \alpha)$ نیاز دارد.

اثبات: طبق فرض ساده و یک‌نوا بودن تابع درهم‌سازی، هر کلید k که هنوز در جدول ذخیره نشده باشد، با احتمال برابر می‌تواند در هر یک از درایه‌های جدول قرار گیرد. همچنین، می‌دانیم که زمان لازم برای انجام یک جست‌وجوی ناموفق برای کلید k برابر است با زمان لازم برای پیمایش تا انتهای لیست $T[h(k)]$ که دارای طول میانگین α است. بنابراین، اگر زمان محاسبه‌ی $h(k)$ برابر $\mathcal{O}(۱)$ فرض شود، زمان جست‌وجوی ناموفق برابر است با $\mathcal{O}(۱ + \alpha)$. □

قضیه‌ی ۵-۲ دریک جدول درهم‌سازی با روش زنجیره‌ای، با فرض استفاده از تابع درهم‌سازی ساده‌ی یک‌نوا، یک جست‌وجوی موفق به زمان میانگین $\mathcal{O}(۱ + \alpha)$ نیاز دارد.

اثبات: فرض کنید x_i برابر iامین عنصری است که در جدول درج می‌شود $(i = ۱, ۲, \cdots, n)$، و نیز $k_i = key[x_i]$. همچنین، متغیر تصادفی X_{ij} را برابر $X_{ij} = I(h(k_i) = h(k_j))$ (یعنی وقوع رخ‌دادی که در آن $h(k_i) = h(k_j)$) می‌گیریم. با فرض ساده و یک‌نوا بودن تابع درهم‌سازی، می‌دانیم که $prob\{(h(k_i) = h(k_j))\} = \frac{۱}{m}$، بنابراین $E[X_{ij}] = \frac{۱}{m}$.

[۹] simple uniform hashing

از طرفی، میانگین تعداد عناصر مورد بررسی برای جست‌وجوی موفق برابر است با

$$\begin{aligned} E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\ &= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\ &= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \\ &= 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}. \end{aligned}$$

بنابراین، میانگین زمان جست‌وجوی‌موفق (با احتساب زمان لازم برای محاسبه‌ی تابع درهم‌سازی) برابر است با

$$\Theta(2+\alpha/2-\alpha/2n) = \Theta(1+\alpha).$$

□

تمرین‌های بخش ۵-۳

۵-۳.۱ با فرض استفاده از تابع ساده و یک‌نوای h برای درهم‌سازی n کلید متمایز در آرایه‌ی T به طول m، امید ریاضی تعداد برخوردها را به‌دست آورید. به‌طور دقیق‌تر، امید ریاضی اندازه‌ی مجموعه‌ی $\{\{k,l\} : k \neq l \text{ and } h(k) = h(l)\}$ چقدر است؟

۵-۳.۲ با استفاده از تابع درهم‌سازی $h(k) = k \bmod 9$، اعداد ۵، ۲۸، ۱۹، ۱۵، ۲۰، ۳۳، ۱۲، ۱۷، و ۱۰ را به‌ترتیب در جدولی به‌اندازه‌ی ۹ درج کنید. برای حل مشکل برخوردها از روش زنجیره‌ای استفاده کنید.

۵-۳.۳ پروفسور مارلی اعتقاد دارد که اگر روش زنجیره‌ای را تغییر دهیم که هر لیست به‌صورت مرتب‌شده نگه‌داری شود، می‌توان کارایی را به‌طرز چشم‌گیری افزایش داد. تغییرات پروفسور چه تأثیری بر زمان اجرای جست‌وجوی موفق، جست‌وجوی ناموفق، درج و حذف دارد؟

۵-۳.۴ اگر درایه‌های استفاده نشده از جدول درهم‌سازی را درون یک لیست پیوندی به‌نام «لیست آزاد» نگه داریم، چگونه می‌توان از این لیست برای خانه‌های جدول درهم‌سازی حافظه تخصیص داد و آن‌ها را آزاد کرد؟ فرض کنید که هر درایه علاوه بر یک پرچم یک بیتی و دو یا یک اشاره‌گر،

می‌تواند یک عنصر را در خود ذخیره کند. پیاده‌سازی شما باید طوری باشد تا میانگین زمان اجرای تمام اعمال فرهنگ داده‌ای و عملیات بر روی لیست آزاد $\mathcal{O}(1)$ باشد. آیا لیست آزاد باید دوسویه باشد یا یک لیست پیوندی یک‌سویه کفایت می‌کند؟

۵.۳-۵ ثابت کنید که اگر $|U| > nm$ باشد، آن‌گاه زیرمجموعه‌ای n عضوی از U وجود دارد که همه‌ی اعضای آن به یک خانه فرستاده می‌شوند. بدین ترتیب زمان جست‌وجو برای درهم‌سازی زنجیره‌ای در بدترین حالت $\mathcal{O}(n)$ است.

۴-۵ توابع درهم‌سازی

یک تابع درهم‌سازی خوب تابعی است که مطابق تعریف بالا ساده و یک‌نوا باشد. بسیاری از توابع درهم‌سازی فرض می‌کنند کلید عناصر اعداد طبیعی هستند، بنابراین اگر کلید عناصری که می‌خواهیم ذخیره کنیم طبیعی نباشد، باید روشی برای متناظر کردن آن‌ها با اعداد طبیعی بیابیم. برای مثال، اگر کلید عناصر رشته‌هایی از نویسه‌ها باشد، می‌توانیم کلید را مجموع کُد اَسکی[۱۰] نویسه‌های آن رشته در نظر بگیریم.

در ادامه روش‌های ساده‌ی مختلف برای طراحی تابع درهم‌سازی را بیان می‌کنیم.

۱-۴-۵ روش تقسیم

در این روش، تابع درهم‌سازی برابر باقیمانده‌ی تقسیم یک کلید k بر m در نظر گرفته می‌شود. یعنی، $h(k) = k \bmod m$.

بعضی مقادیر برای m مناسب‌ترند و باید از انتخاب مقادیر خاصی هم اجتناب کنیم. مثلاً m نباید توانی از ۲ باشد، چرا که اگر مثلاً $m = 2^p$، در آن‌صورت $h(k)$ معادل p بیت کم‌ارزش k خواهد بود. در صورتی‌که بهتر است از مقادیر همه‌ی بیت‌های کلید برای محاسبه‌ی $h(k)$ استفاده شود. به‌طور کلی، یک عدد اول که نزدیک به توانی از ۲ نباشد انتخاب مناسبی برای m است. مثلاً اگر تقریباً ۲۰۰۰ کلید رشته‌ای داشته باشیم و حاضر باشیم که به‌طور میانگین ۳ عنصر را برای پیدا کردن کلید مورد نظر خود بگردیم، بهتر است اندازه‌ی جدول را برابر ۷۰۱ بگیریم، چرا که ۷۰۱ یک عدد اول نزدیک به $\frac{2000}{3}$ است. در

[۱۰] American Standard Code for Information Interchange (ASCII)

این صورت، یک رشته را به یک عدد k تبدیل و از تابع درهم‌سازی $h(k) = k \bmod ۷۰۱$ استفاده می‌کنیم.

مقدار و توزیع کلیدها هم در خوبی یا بدی تابع درهم‌سازی نقش دارد. به‌عنوان یک مثال ساده، فرض کنید همه‌ی ۱۰۰ کلید داده‌شده مجذور کامل باشند، یعنی $k_i = i^۲$ برای $i = ۱, ۲, \ldots, ۱۰۰$. اگر $m = ۷$ باشد، می‌دانیم که

$$xy \bmod m = (x \bmod m)(y \bmod m) \bmod m$$

درنتیجه، $h(k_i) = i^۲ \bmod ۷ = (i \bmod ۷)^۲ \bmod ۷$. بنابراین،

i	$i \bmod ۷$	$(i \bmod ۷)^۲ \bmod ۷$
۱	۱	۱
۲	۲	۴
۳	۳	۲
۴	۴	۲
۵	۵	۴
۶	۶	۱
۷	۰	۰

داریم $۱۰۰ = ۲ + ۷ * ۱۴$. در نتیجه، تعداد برخوردها در درایه‌های مختلف به‌صورت زیر است و می‌بینیم که کلیدها به‌صورت یک‌نوا توزیع نشده‌اند.

درایه	تعداد برخورد
۰	۱۴
۱	$۱۴ * ۲ + ۱ = ۲۹$
۲	$۱۴ * ۲ = ۲۸$
۳	۰
۴	$۱۴ * ۲ + ۱ = ۲۹$
۵	۰
۶	۰

چنین محاسباتی برای مقادیر بزرگ‌تر نیز قابل انجام است.

۵-۴-۲ روش ضرب

در روش ضرب[۱۱]، ابتدا کلید k را در یک عدد ثابت A که $۰ < A < ۱$ است ضرب می‌کنیم و قسمت اعشاری آن‌را به‌دست می‌آوریم. یعنی

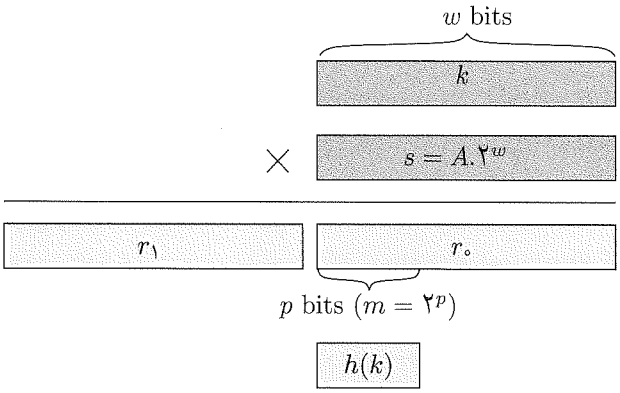
$$h(k) = \lfloor m(kA \bmod ۱) \rfloor$$

[۱۱] multiplication method

که در آن $kA \bmod ۱$ برابر $kA - \lfloor kA \rfloor$ یعنی برابر قسمت اعشاری عدد kA است.

مزیت مهم روش ضرب آن است که این تابع به مقدار m حساس نیست. ما معمولاً m را توان ۲ می‌گیریم ($m = ۲^p$ برای یک عدد صحیح p)، چون به‌سادگی می‌توانیم تابع درهم‌سازی را به‌صورت زیر پیاده‌سازی کنیم.

فرض کنید که کلمه‌ی کامپیوتر w بیتی است و k در یک کلمه جای می‌گیرد. A را طوری انتخاب می‌کنیم که برابر $s/۲^w$ باشد، که در آن s یک عدد صحیح در محدوده‌ی $۰ < s < ۲^w$ باشد. مطابق شکل ۵-۳، ابتدا k را در عدد صحیح w بیتی $s = A.۲^w$ ضرب می‌کنیم. حاصل‌ضرب یک عدد $۲w$ بیتی به‌صورت $r_۱ ۲^w + r_۰$ است که $r_۱$ کلمه‌ی پرارزش و $r_۰$ کلمه‌ی کم‌ارزش حاصل‌ضرب است. عدد p بیتی تابع درهم‌سازی مورد نظر، p بیت پرارزش $r_۰$ است. شکل ۵-۴ مثالی را برای $r = ۳$ نشان می‌دهد که حاصل کار، عدد ۳ (011) خواهد شد.



شکل ۵-۳ نحوه‌ی محاسبه‌ی تابع درهم‌سازی به‌روش ضرب.

$$
\begin{array}{rl}
1101011 & k \\
1011001 & A \\
\hline
1001010011011 &
\end{array}
$$

شکل ۵-۴ مثالی از محاسبه‌ی تابع درهم‌سازی به‌روش ضرب.

هرچند که این روش با هر مقدار A درست است، ولی برای برخی از مقادیر بهتر کار می‌کند. انتخاب بهینه وابسته به نوع داده‌ای است که قرار است درهم‌سازی شود. دونالد کنوث پیشنهاد می‌کند که مقدار A برابر زیر باشد [۱۲]:

$$A \approx (\sqrt{5}-1)/2 = ۰/۶۱۸۰۳۳۹۸۸۷.. \qquad (۱-۵)$$

مثلاً، فرض کنید که $k = ۱۲۳۴۵۶$، $p = ۱۴$، $m = ۲^{۱۴} = ۱۶,۳۸۴$ و $w = ۳۲$ باشد. اگر پیشنهاد کنوث را دنبال کنیم، مقدار A را طوری انتخاب می‌کنیم که به‌صورت $s/۲^{۳۲}$ باشد و به مقدار ۵-۱ نزدیک باشد. یعنی $A = (۲,۶۵۴,۴۳۵,۷۶۹)/۲^{۳۲}$ را در نظر می‌گیریم. بنابراین

$$ks = ۳۲۷,۷۰۶,۰۲۲,۲۹۷,۶۶۴ = (۷۶,۳۰۰ \times ۲^{۳۲}) + ۱۷,۶۱۲,۸۶۴.$$

در نتیجه، $r_۱ = ۷۶,۳۰۰$ و $r_۰ = ۱۷,۶۱۲,۸۶۴$. ۱۴ بیت پرارزش $r_۰$ جواب مورد نظر ماست، پس $h(k) = ۶۷$.

۵-۵ درهم‌سازی سراسری

اگر تابع درهم‌سازی را از قبل داشته باشیم، می‌توان مجموعه‌ای از n کلید را پیدا کرد که همه به یک درایه فرستاده شوند و در نتیجه زمان بازیافت عناصر $\Theta(n)$ گردد که بدترین حالت ممکن است. هر تابع درهم‌سازی مشخصی نیز ممکن است در بدترین حالت چنین رفتار بدی را از خود بروز دهد. تنها راه مؤثر برای بهبود وضعیت، انتخاب تصادفی تابع درهم‌سازی است به‌گونه‌ای که مستقل از مقادیری باشد که عملاً می‌خواهیم ذخیره کنیم. این روش را «درهم‌سازی سراسری[۱۲]» می‌نامیم و نشان می‌دهیم که این روش بدون درنظر گرفتن چگونگی انتخاب مقادیر، در حالت میانگین عمل‌کرد مناسبی خواهد داشت.

ایده‌ی اصلی درهم‌سازی سراسری این است که در شروع کار، تابع درهم‌سازی را از یک خانواده‌ی توابع که با دقت طراحی شده است انتخاب کنیم. در مثال جدول نمادهای یک کامپایلر، مشاهده خواهیم کرد که با درهم‌سازی سراسری، انتخاب شناسه‌ها به‌وسیله‌ی کاربر دیگر همیشه نمی‌تواند باعث عمل‌کردی ضعیف شود. عمل‌کرد ضعیف تنها هنگامی اتفاق می‌افتد که کامپایلر به‌صورت تصادفی یک تابع درهم‌سازی را انتخاب کند که باعث شود مجموعه‌ی شناسه‌ها به طرز نامناسبی درهم‌سازی شوند، اما نشان می‌دهیم که احتمال رخ دادن این حالت کم و برای تمام مجموعه‌های شناسه‌های با اندازه‌ی مساوی، برابر است.

[۱۲] universal hashing

فرض کنید $\mathcal{H}$ خانواده‌ای محدود از توابع درهم‌سازی است که هر کدام از آن‌ها تمام کلیدهای ممکن (U) را به بازه‌ی $[۰, ۱, \cdots, m-۱]$ نگاشت می‌کند. چنین خانواده‌ای را «سراسری» می‌گوییم اگر برای هر دو کلید متفاوت $k, l \in U$، تعداد توابع درهم‌سازی $h \in \mathcal{H}$ که برای آن‌ها $h(k) = h(l)$ است، حداکثر برابر $|\mathcal{H}|/m$ باشد. به بیان دیگر، اگر یک تابع درهم‌سازی h به صورت تصادفی از $\mathcal{H}$ انتخاب شود، احتمال وقوع برخورد بین کلیدهای متفاوت k و l با تابع h بیش‌تر از $۱/m$ برابر شانس برخورد نیست، اگر مقدار $h(k)$ و $h(l)$ به‌صورت تصادفی و مستقل از هم از مجموعه‌ی $[۰, ۱, \cdots, m-۱]$ انتخاب شده باشند.

قضیه‌ی ۵-۳ نشان می‌دهد که یک خانواده‌ی سراسری از توابع درهم‌سازی رفتار مطلوبی را در حالت میانگین خواهد داشت. به‌خاطر بیاورید که n_i طول لیست $T[i]$ در درهم‌سازی به‌روش زنجیره‌ای است.

قضیه‌ی ۵-۳ فرض کنید تابع درهم‌سازی h از یک خانواده‌ی سراسری از توابع درهم‌سازی انتخاب شده و از آن برای درهم‌سازی n عنصر در جدول T با اندازه‌ی m استفاده شده است، و نیز برای حل مشکل برخوردها از روش زنجیره‌ای استفاده کرده‌ایم. اگر کلید k در جدول موجود نباشد، آن‌گاه $E[n_{h(k)}]$، یا امید ریاضی طول لیستی که k به آن فرستاده می‌شود، حداکثر برابر α است. اگر k در جدول موجود باشد، آن‌گاه $E[n_{h(k)}]$، یا امید ریاضی طول لیست شامل k، حداکثر برابر $\alpha + ۱$ است.

اثبات: توجه کنید که امید ریاضی در این‌جا تنها وابسته به انتخاب تابع درهم‌سازی است و به فرضیات ما راجع به توزیع کلیدها ارتباطی ندارد. برای هر جفت کلید متفاوت k و l، نماینده‌ی متغیر تصادفی $X_{kl} = I(h(k) = h(l))$ را تعریف می‌کنیم. از آن‌جا که بنا بر تعریف، با تابع h، هر جفت از کلیدها با احتمال حداکثر $۱/m$ برخورد می‌کنند، داریم $Pr\{h(k) = h(l)\} \leq ۱/m$ و در نتیجه داریم $E[X_{kl}] \leq ۱/m$. برای هر کلید k نیز متغیر تصادفی Y_k را تعریف می‌کنیم و برابر تعداد برخوردهای کلید k با بقیه‌ی کلیدها می‌گیریم، بنابراین

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{lk},$$

و از آن نتیجه می‌گیریم

$$\begin{aligned} E[Y_k] &= E\left[\sum_{\substack{l\in T\\ l\neq k}} X_{lk}\right] \\ &= \sum_{\substack{l\in T\\ l\neq k}} E[X_{lk}] \\ &= \sum_{\substack{l\in T\\ l\neq k}} \frac{1}{m}. \end{aligned}$$

ادامه‌ی اثبات به این بستگی دارد که در جدول T مقدار k وجود دارد یا خیر.

- اگر $k \notin T$، آن‌گاه $n_{h(k)} = Y_k$ و نیز $n = |\{l : l \in T \text{ and } l \neq k\}|$. بنابراین، $E[n_{h(k)}] = E[Y_k] \leq n/m = \alpha$.
- اگر $k \in T$، آن‌گاه چون k در لیست $T[h(k)]$ ظاهر می‌شود ولی Y_k مقدار k را حساب نکرده است، داریم $n_{h(k)} = Y_k + 1$ و $|\{l : l \in T \text{and } l \neq k\}| = n - 1$. بنابراین، $E[n_{h(k)}] = E[Y_k] + 1 \leq (n-1)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha$.

□

نتیجه‌ی زیر نشان می‌دهد که حاصل درهم‌سازی سراسری مطلوب است. یعنی، با این روش برای حریف غیر ممکن است که دنباله‌ای از عملیات را انتخاب کند که سبب ایجاد بدترین حالت شود. اگر انتخاب تابع درهم‌سازی در هنگام اجرا به‌قدر کافی تصادفی باشد، تضمین می‌کنیم که می‌توان هر دنباله از اعمال را به‌طور میانگین در زمان اجرای مناسب انجام داد.

نتیجه‌ی ۵-۱ با استفاده از درهم‌سازی سراسری در جدولی با m درایه، اگر برخوردها با روش زنجیره‌ای حل شوند، هر دنباله از n عمل درج، جست‌وجو و حذف که $O(m)$ عمل درج را شامل شود، به زمان میانگین $\Theta(n)$ نیاز دارد.

اثبات: چون تعداد عملیات درج $O(m)$ است، باید $n = O(m)$ باشد، بنابراین $\alpha = O(1)$. هر درج و حذف به زمان ثابت نیاز دارد، و طبق قضیه‌ی ۵-۳، میانگین زمان اجرای عمل

جست‌وجو نیز $O(1)$ است. پس به دلیل خطی بودن فرضیات ما، امید ریاضی زمان اجرای کل عملیات $O(n)$ است. □

ساخت یک خانواده‌ی سراسری از توابع درهم‌سازی

با کمک گرفتن از نظریه‌ی اعداد، می‌توان یک خانواده‌ی سراسری از توابع درهم‌سازی ساخت. ابتدا عدد اول p را به‌اندازه‌ی کافی بزرگ انتخاب می‌کنیم که تمامی مقادیر ممکن کلیدها (k) در بازه‌ی بسته‌ی $[0, p-1]$ قرار گیرند. فرض کنید Z_p معرف مجموعه‌ی $\{0, 1, \ldots, p-1\}$ باشد، و Z_p^* مجموعه‌ی $\{1, 2, \ldots, p-1\}$ را نشان دهد. از آن‌جا که فرض کرده‌ایم اندازه‌ی فضای مقادیر بزرگ‌تر از اندازه‌ی جدول درهم‌سازی است، داریم $p > m$.

اکنون تابع درهم‌سازی $h_{a,b}$ را برای هر $a \in Z_p^*$ و هر $b \in Z_p$ به‌صورت زیر تعریف می‌کنیم:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m. \qquad (۵-۲)$$

به‌عنوان مثال، اگر $p = 17$ و $m = 6$ باشد، داریم $h_{3,4}(8) = 5$. خانواده‌ی تمامی این توابع درهم‌سازی مانند زیر است:

$$\mathcal{H}_{p,m} = \{h_{a,b} : a \in Z_p^* \text{ and } b \in Z_p\} \qquad (۵-۳)$$

هر تابع درهم‌سازی $h_{a,b}$، Z_p را به Z_m می‌نگارد. این خانواده از توابع درهم‌سازی این خاصیت مفید را دارد که m، اندازه‌ی برد توابع، مقداری دل‌خواه است و لزومی ندارد که اول باشد؛ خاصیتی که از آن بعداً در بخش ۵-۷ مربوط به درهم‌سازی کامل استفاده می‌کنیم. به‌دلیل این‌که $p-1$ انتخاب برای a، و p انتخاب برای b داریم، $\mathcal{H}_{p,m}$ شامل $p(p-1)$ تابع می‌باشد.

قضیه‌ی ۵-۴ خانواده‌ی $\mathcal{H}_{p,m}$ از توابع درهم‌سازی که توسط معادلات ۵-۲ و ۵-۳ تعریف شده است، سراسری است.

اثبات: دو مقدار مختلف k و l از Z_p را درنظر بگیرید. برای تابع درهم‌سازی داده‌شده

$h_{a,b}$، r و s را مانند زیر تعریف می‌کنیم:

$$\begin{aligned} r &= (ak+b) \bmod p, \\ s &= (al+b) \bmod p. \end{aligned}$$

مشاهده می‌کنیم که چون $r-s \equiv a(k-l) (\bmod p)$، p یک عدد اول است و a و $k-l$ به هنگ p ناصفر هستند، داریم $r \neq s$. بنابراین، هنگام محاسبه‌ی هر $h_{a,b}$ از $\mathcal{H}_{p,m}$، مقادیر متفاوت k و l به دو مقدار متفاوت r و s به هنگ p نگاشته می‌شوند. پس در مرحله‌ی گرفتن باقی‌مانده بر p، برخوردی نداریم. به‌علاوه، هر یک از $p(p-1)$ انتخاب ممکن برای جفت (a,b) که $a \neq 0$، جفت (r,s) متفاوتی را نتیجه می‌دهد که $r \neq s$. و چون با داشتن s و r می‌توانیم a و b را مانند زیر به‌دست بیاوریم

$$\begin{aligned} a &= ((r-s)((k-l)^{-1} \bmod p)) \bmod p, \\ b &= (r-ak) \bmod p, \end{aligned}$$

که $(k-l)^{-1} \bmod p$ وارون ضربی یک‌تای $k-l$ به هنگ p است، و از آن‌جا که تنها $p(p-1)$ جفت ممکن (r,s) که $r \neq s$ وجود دارد، تناظری یک‌به‌یک بین جفت‌های (a,b) که $a \neq 0$ و جفت‌های (r,s) که $r \neq s$ وجود دارد. بنابراین، برای هر جفت ورودی داده‌شده‌ی k و d، اگر (a,b) را به‌صورت تصادفی و یک‌نوا از $Z_p^* \times Z_p$ انتخاب کنیم، جفت حاصل (r,s) به‌احتمال مساوی می‌تواند هر جفت با مقادیر متفاوت به هنگ p باشد.

پس، نتیجه می‌گیریم احتمال این‌که کلیدهای متفاوت k و l برخورد داشته باشند، برابر احتمال این است که $r \equiv s (\bmod m)$، که در آن r و s مقادیر متفاوتی به هنگ p هستند که به‌تصادف انتخاب شده‌اند. برای مقدار داده‌شده‌ی r، تعداد مقادیر ممکن برای s از بین $p-1$ مقدار باقی‌مانده که $s \neq r$ و $s \equiv r (\bmod p)$ حداکثر برابر زیر است:

$$\lceil p/m \rceil - 1 \leq (p+m-1)/m - 1 = (p-1)/m.$$

و احتمال این‌که s و r بعد از این‌که باقی‌مانده‌ی آن‌ها را به هنگ m گرفتیم، با هم برخورد داشته باشند حداکثر برابر $1/m = \lceil (p-1)/m \rceil / (p-1)$ است.

بنابراین برای هر جفت مقدار متفاوت $k, l \in Z_p$

$$Pr\{h_{a,b}(k) = h_{a,b}(l)\} \leq 1/m$$

و $\mathcal{H}_{p,m}$ واقعاً سراسری است. □

تمرین‌های بخش ۵-۵

۱.۵-۵ فرض کنید می‌خواهیم عمل جست‌وجو را در یک لیست پیوندی به‌طول n انجام دهیم، که در آن هر عنصر دارای کلید k با مقدار درهم‌سازی $h(k)$ است. هر کلید یک رشته‌ی نویسه‌ی بلند می‌باشد. یک عنصر و کلید آن داده شده است. چگونه می‌توانیم از مقادیر درهم‌سازی برای جست‌وجوی این عنصر کمک بگیریم؟

۲.۵-۵ فرض کنید برای درهم‌سازی رشته‌ای با r نویسه در آرایه‌ای با m درایه، آن‌را به‌صورت عددی در مبنای ۱۲۸ در نظر می‌گیریم و سپس از روش تقسیم استفاده می‌کنیم. m را می‌توان به‌وسیله‌ی یک کلمه‌ی ۳۲ بیتی نمایش داد، ولی برای نمایش رشته‌ی فوق، به‌تعداد زیادی کلمه نیاز خواهیم داشت. چگونه می‌توان روش تقسیم را برای محاسبه‌ی مقدار درهم‌سازی رشته با استفاده از تعداد ثابتی متغیر جدید در حافظه به‌کار برد؟

۳.۵-۵ نوعی از روش تقسیم را در نظر بگیرید که در آن $h(k) = k \bmod m$ که $m = 2^p - 1$ و k رشته‌ای از نویسه‌هاست که به‌صورت عددی در مبنای 2^p است. نشان دهید اگر رشته‌ی x جای‌گشتی از رشته‌ی y باشد، آن‌گاه x و y مقدار درهم‌سازی یک‌سانی دارند. کاربردی را مثال بزنید که در آن این خاصیت برای یک تابع درهم‌سازی مناسب نباشد.

۴.۵-۵ جدول درهم‌سازی با اندازه‌ی $m = 1000$ و تابع درهم‌سازی $h(k) = \lfloor m(kA \bmod 1) \rfloor$ را در نظر بگیرید که در آن $A = (\sqrt{5} - 1)/2$. محاسبه کنید که مقادیر ۶۱، ۶۲، ۶۳، ۶۴ و ۶۵ به کدام درایه‌ها فرستاده می‌شوند.

**** ۵.۵-۵** خانواده‌ی $\mathcal{H}$ از توابع درهم‌سازی از مجموعه‌ی محدود U به مجموعه‌ی محدود B را ϵـ سراسری می‌نامیم اگر برای تمام جفت کلیدهای متمایز k و l از U داشته باشیم $Pr\{h(k) = h(l)\} \leq \epsilon$. این احتمال با انتخاب تصادفی تابع درهم‌سازی h از میان خانواده‌ی $\mathcal{H}$ به‌دست می‌آید. نشان دهید که در یک خانواده ϵـ سراسری از توابع درهم‌سازی حتماً داریم $\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}$.

*** ۶.۵-۵** فرض کنید U مجموعه‌ای از n تایی‌هایی است که مقادیر آن‌ها از Z_p انتخاب شده باشند، و $B = Z_p$ که p عددی اول است. تابع درهم‌سازی $h_b : U \to B$ را برای $b \in Z_p$ و ورودی n تایی $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ از U، به‌صورت زیر تعریف می‌کنیم:

$$h(\langle a_0, a_1, \ldots, a_{n-1} \rangle) = \sum_{j=0}^{n-1} a_j b^j$$

اگر $\mathcal{H} = \{h_b : b \in Z_p\}$، با توجه به تعریف ϵـ سراسری در تمرین قبل، ثابت کنید که $\mathcal{H}$، $(n-1)/p$ـ سراسری است.

۵-۶ آدرس‌دهی باز

در «آدرس‌دهی باز[۱۳]»، یک آرایه به‌اندازه‌ی m به‌عنوان جدول درهم‌سازی استفاده می‌شود که حداکثر m عنصر در آن جای می‌گیرد و در هر درایه، فقط یک عنصر ذخیره می‌شود. بنابراین، برای استفاده از این روش، تعداد بیشینه‌ی عناصر را باید از قبل بدانیم، یا آن‌که از «جدول درهم‌سازی پویا[۱۴]» استفاده کنیم که در آن اندازه‌ی جدول برحسب نیاز، کم یا زیاد (معمولاً نصف یا ۲ برابر) می‌شود.

برای درج یک عنصر در آدرس‌دهی باز، باید تعدادی از درایه‌های جدول «وارسی[۱۵]» شوند تا جای مناسب برای درج یک عنصر پیدا شود. برای این‌کار تابع درهم‌سازی را به این صورت تعریف می‌کنیم:

$$h : U \times \{۰, ۱, \cdots, m-۱\} \to \{۰, ۱, \cdots, m-۱\}.$$

در آدرس‌دهی باز و برای یک کلید k، به درایه‌های زیر که به‌صورت متوالی وارسی می‌شوند، «دنباله‌ی وارسی[۱۶]» k می‌گوییم.

$$\langle h(k,۰), h(k,۱), \cdots, h(k,m-۱)\rangle$$

برای جست‌وجوی یک کلید، دنباله‌ی وارسی آن دنبال می‌شود تا آن کلید یافت شود، وگرنه کلید در جدول موجود نیست.

برای درج، دنباله‌ی وارسی بررسی می‌شود تا اولین درایه‌ی خالی پیدا شود و عنصر مورد نظر در آن‌جا درج می‌شود. اگر چنین جای خالی‌ای پیدا نشد، در آن‌صورت پیام خطای overflow چاپ می‌شود. توجه کنید که تابع درهم‌سازی $h(k,i)$ ممکن است طوری طراحی شده باشد که دنباله‌ی وارسی یک کلید، به یک درایه بیش از یک بار برود و در نتیجه، به بعضی از درایه‌ها اصلاً رجوع نکند. یعنی ممکن است برای یک کلید خاص، جا برای درج آن وجود داشته باشد، اما هیچ‌گاه پیدا و امکان درج آن کلید فراهم نشود.

رویه‌ی HASH-INSERT این‌کار را انجام می‌دهد. دقت کنید که اگر به‌جای **until** $i = m$ دستور **until** $A[j] =$**null or** $i = m$ را قرار دهیم، ظاهراً موجب تسریع عمل درج می‌شود؛ چرا که اگر عنصر در جدول نباشد لزومی به گشتن همه‌ی دنباله‌ی وارسی نیست و اولین درایه‌ی خالی جواب منفی را تولید می‌کند. اما در صورت این تغییر، اگر جست‌وجو با حذف همراه شود، چنان‌چه خواهیم دید، ممکن است مشکل ایجاد کند.

---

[۱۳] open hashing
[۱۴] dynamic hash table
[۱۵] probing
[۱۶] probe sequence

```
HASH-INSERT (T, k)
 1  i ← 0
 2  repeat j ← h(k, i)
 3      if T[j] = null
 4          then T[j] ← k
 5               return j
 6          else i ← i + 1
 7  until i = m
 8  error hash table overflow


HASH-SEARCH (T, k)
 1  i ← 0
 2  repeat j ← h(k, i)
 3      if T[j] = k
 4          then return j
 5      i ← i + 1
 6  until i = m
 7  return null
```

برای حذف یک عنصر، ابتدا آن‌را پیدا می‌کنیم و مقدار آن درایه را برابر تهی قرار می‌دهیم.

```
HASH-DELETE (T, k)
 1  i ← HASH-SEARCH (T, k)
 2  if i ≠ null
 3     then T[i] ← null
 4     else error key k not found
```

اما با انجام تغییرات زیر می‌توانیم کار جست‌وجو را کمی سریع‌تر کنیم. پس از هر عمل حذف، به‌جای این‌که در درایه عنصر حذف‌شده، مقدار **null** قرار دهیم، مقدار خاص "deleted" را می‌نویسیم. در این صورت، درایه‌ی "deleted" برای جست‌وجو مانند درایه‌ی پُر است. اما برای درج، اگر عنصر قبلاً وجود نداشته باشد، این درایه مانند درایه‌ی تهی محسوب می‌شود.

```
HASH-SEARCH-NEW (T, k)
1  i ← 0
2  repeat j ← h(k, i)
3      if T[j] = k
4          then return j
5      i ← i + 1
6  until T[j] = null or i = m
7  return null

HASH-INSERT-NEW (T, k)
1  if HASH-SEARCH(T, k) ≠ null
2     then error item already exits
3  i ← 0
4  repeat j ← h(k, i)
5      if T[j] = null or T[j] = "deleted"
6          then T[j] ← k
7                return j
8          else i ← i + 1
9  until i = m
10 error hash table overflow

HASH-DELETE-NEW (T, k)
1  i ← HASH-SEARCH (T, k)
2  if i ≠ null
3     then T[i] ← "deleted"
4     else error key k not found
```

در تحلیل آدرس‌دهی باز فرض می‌کنیم که درهم‌سازی به‌صورت یک‌نوا عمل می‌کند. یعنی دنباله‌ی وارسی هر کلید، با احتمال یکسان برابر هر یک از $m!$ جای‌گشت $\langle ۰, ۱, \ldots m-۱ \rangle$ است. این تعریف، البته حالت کلی تابع درهم‌سازی یک‌نواست که قبلاً تعریف شد. پیاده‌سازی درهم‌سازی یک‌نوا کار مشکلی است و در عمل روش‌های

تقریبی مانند «درهم‌سازی دوگانه[۱۷]» که بعداً گفته خواهد شد، استفاده می‌شود.

سه روش برای محاسبه‌ی دنباله‌ی وارسی به‌کار می‌رود. این روش‌ها تضمین می‌کنند که برای هر کلیدِ k دنباله‌ی

$$\langle h(k,0), h(k,1), \cdots, h(k,m-1)\rangle$$

یک جای‌گشت از $\langle 0, 1, \ldots m-1\rangle$ باشد. البته هیچ‌یک از این روش‌ها، فرض درهم‌سازی یک‌نوا را ارضا نمی‌کند، چون هیچ‌کدام قادر به تولید بیش از m^2 دنباله‌ی وارسی نیست، که بسیار کم‌تر از $m!$ است. درهم‌سازی دوگانه بیش‌ترین تعداد دنباله‌ها را تولید می‌کند و گاهی از دو روش دیگر بهتر است.

۵-۶-۱ وارسی خطی

با فرض داشتن یک تابع درهم‌سازی عادی $h' : U \to \{0, 1, \cdots, m-1\}$، که از آن به نام تابع درهم‌سازی کمکی[۱۸] یاد می‌کنیم، روش «وارسی خطی[۱۹]» از تابع زیر برای $i = 0, 1, \cdots, m-1$ استفاده می‌کند:

$$h(k,i) = (h'(k) + i) \bmod m \qquad (۴-۵)$$

به‌ازای یک کلید k، ابتدا $T[h'(k)]$ را وارسی می‌کنیم، سپس به $T[h'(k)+1]$ و بعد به $T[h'(k)+2]$ می‌رویم و این کار را به‌صورت حلقوی ادامه می‌دهیم تا به $T[h'(k)-1]$ برسیم. روشن است که این روش فقط m دنباله‌ی وارسی مجزا تولید می‌کند.

پیاده‌سازی وارسی خطی ساده است. ولی این روش مشکلی به‌نام «خوشه‌بندی اولیه[۲۰]» دارد که موجب می‌شود تا در جدول، بخش‌های پر، دنبال هم ظاهر شوند که هر کدام را یک خوشه می‌گوییم. یک درایه‌ی خالی که بلافاصله بعد از i درایه‌ی پرِ دنبالِ هم قرار گیرد، در مرحله‌ی بعد و با احتمال $(i+1)/m$ پر می‌شود و به این ترتیب، به مرور زمان خوشه‌ها بزرگ‌تر شده و زمان جست‌وجو هم طولانی‌تر می‌شود.

---

[۱۷] double hashing
[۱۸] auxilary
[۱۹] linear probing
[۲۰] primary clustering

۵-۶-۲ وارسی درجه‌ی ۲

«وارسی درجه‌ی ۲[۲۱]» از تابع درهم‌سازی زیر برای $(i = ۰, ۱, \cdots, m-۱)$ استفاده می‌کند:

$$h(k,i) = (h'(k) + c_۱ i + c_۲ i^۲) \bmod m \qquad (۵-۵)$$

که در آن h' تابع درهم‌سازی کمکی و $c_۱, c_۲$ $(c_۲ \neq ۰)$ ثابت‌های کمکی هستند.

این روش بسیار بهتر از وارسی خطی عمل می‌کند و مشکل خوشه‌بندی را ندارد. البته برای این‌که همه‌ی m درایه‌ی جدول مورد استفاده قرار گیرند، مقادیر $c_۱$، $c_۲$ و m باید به‌صورت خاص انتخاب شوند. همچنین، اگر وارسی اولیه‌ی دو کلید مشابه باشند، دنباله‌ی وارسی‌های آن‌ها هم یک‌سان است

$$h(k_۱, ۰) = h(k_۲, ۰) \Longrightarrow h(k_۱, i) = h(k_۲, i).$$

۵-۶-۳ درهم‌سازی دوگانه

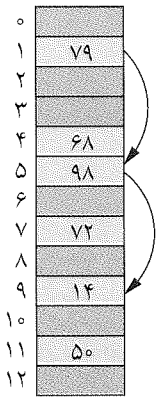
درهم‌سازی دوگانه یک روش وارسی است که در آن تابع درهم‌سازی به‌صورت زیر است:

$$h(k,i) = (h_۱(k) + i h_۲(k)) \bmod m$$

که $h_۱$ و $h_۲$ تابع‌های درهم‌سازی کمکی هستند. اولین وارسی در $T[h_۱(k)]$ انجام می‌شود و وارسی‌های بعدی به درایه‌ای با فاصله‌ی $h_۲(k)$ (البته به هنگ m) می‌رود. شکل ۵-۵ مثالی از درج کلید ۱۴ را با استفاده از این روش نشان می‌دهد. در این مثال، $m = ۱۳$ و $h_۱(k) = k \bmod ۱۳$ و $h_۲(k) = ۱ + (k \bmod ۱۱)$. چون $۱۴ \equiv ۱ (\bmod ۱۳)$ و $۱۴ \equiv ۳ (\bmod ۱۱)$، بنابراین کلید ۱۴، پس از آن‌که درایه‌های ۱ و ۵ وارسی شد و پر بودند، در درایه‌ی خالی ۹ درج می‌شود.

در طراحی این درهم‌سازی، باید مقدار $h_۲(k)$ نسبت به اندازه‌ی جدول، یا m، اول باشد تا این‌که کلیه‌ی درایه‌های جدول وارسی شوند. یک روش مناسب برای اِعمال این خاصیت، انتخاب m به‌صورت توان ۲ و طراحی $h_۲$ به‌طوری است که همیشه یک عدد فرد تولید کند.

[۲۱] quadratic probing



شکل ۵-۵ درج با وارسی دوگانه.

یک روش دیگر آن است که m عدد اول باشد و h_2 همیشه یک عدد صحیح و مثبت کم‌تر از m را تولید کند. مثلاً، m می‌تواند یک عدد اول و h_1 و h_2 به‌صورت زیر باشند:

$$h_1(k) = k \bmod m$$
$$h_2(k) = 1 + (k \bmod m'),$$

که در آن m' عددی کوچک‌تر و نزدیک به m (مثلاً $m-1$) است. مثلاً اگر $k = 123456$، $m = 701$ و $m' = 700$ باشد، داریم $h_1(k) = 80$ و $h_2(k) = 257$. بنابراین ابتدا درایه‌ی ۸۰ و سپس درایه‌های با فاصله‌ی ۲۵۷ از آن (به هنگ m) را وارسی می‌کنیم.

درهم‌سازی دوگانه نسبت به وارسی خطی و وارسی درجه‌ی ۲ بهتر عمل می‌کند، چون به تعداد $\Theta(m^2)$ (در مقایسه با $\Theta(m)$) دنباله‌ی وارسی تولید می‌کند. بنابراین به‌نظر می‌رسد کارایی این درهم‌سازی به کارایی آرمانی درهم‌سازی یک‌نوا نزدیک باشد.

۵-۶-۴ تحلیل آدرس‌دهی باز

در آدرس‌دهی باز در هر درایه‌ی جدول حداکثر یک عنصر قرار می‌گیرد، بنابراین $n \leq m$ و در نتیجه $\alpha = \frac{n}{m} \leq 1$. برای تحلیل، فرض می‌کنیم که تابع درهم‌سازی یک‌نواست. دراین صورت دنباله‌ی وارسی $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ در درج یا جست‌وجوی کلید

k، معادل یکی از جای‌گشت‌های $\langle ۰, ۱, \ldots, m-۱ \rangle$ است.

قضیه‌ی ۵-۵ حداکثر تعداد وارسی‌ها در یک جدول درهم‌سازی باز با $\alpha < ۱$ برای یک جست‌وجوی ناموفق به‌طور میانگین برابر است با $\frac{۱}{۱-\alpha}$.

اثبات: فرض کنید X یک متغیر تصادفی است که تعداد وارسی‌ها در یک جست‌وجوی ناموفق را نشان می‌دهد و A_i رخ‌دادی است که i امین وارسی به درایه‌ی پُر برود.

رخ‌داد $\{X \geq i\}$ یعنی: یک جست‌وجوی ناموفق که حتماً $i-۱$ وارسی اولش پُر و پس از آن یکی از وارسی‌هایش (i ام یا بیش‌تر) تهی بوده است. پس

$$\{X \geq i\} = A_۱ \cap A_۲ \cap \ldots \cap A_{i-۱}.$$

داریم

$$\begin{aligned}
\Pr\{X \geq i\} &= \Pr\{A_۱ \cap A_۲ \cap \ldots \cap A_{i-۱}\} \\
&= \Pr\{A_۱\} \times \Pr\{A_۲|A_۱\} \times \Pr\{A_۳|A_۱ \cap A_۲\} \ldots \\
&= \frac{n}{m} \times \frac{n-۱}{m-۱} \times \frac{n-۲}{m-۲} \times \ldots \times \frac{n-i+۲}{m-i+۲} \\
&\leq \left(\frac{n}{m}\right)^{i-۱} \\
&= \alpha^{i-۱}
\end{aligned}$$

و همچنین

$$\begin{aligned}
E[X] &= \sum_{i=۱}^{\infty} i \Pr\{X = i\} \\
&= \sum_{i=۱}^{\infty} i(\Pr\{X \geq i\} - \Pr\{X \geq i+۱\}) \\
&= \sum_{i=۱}^{\infty} \Pr\{X \geq i\} \\
&\leq \sum_{i=۱}^{\infty} \alpha^{i-۱} \\
&= \sum_{i=۰}^{\infty} \alpha^{i} \\
&= \frac{۱}{۱-\alpha}
\end{aligned}$$

□

نتیجه‌ی ۵-۲ درج یک عنصر در یک جدول درهم‌سازی باز در حالت میانگین حداکثر به $\frac{1}{1-\alpha}$ وارسی نیاز دارد.

اثبات: اگر دست‌کم یک فضای خالی وجود داشته باشد، یک عنصر درج می‌شود. بنابراین $\alpha < 1$. درضمن درج یک عنصر مستلزم انجام یک جست‌وجو و قرار دادن آن در اولین درایه‌ی خالی است. پس تعداد میانگین وارسی‌ها با توجه به قضیه‌ی ۵-۵ برابر $\frac{1}{1-\alpha}$ خواهد بود. □

قضیه‌ی ۵-۶ مقدار میانگین تعداد وارسی‌ها در یک جدول درهم‌سازی باز با $\alpha < 1$ برای یک جست‌وجوی موفق حداکثر برابر است با $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.

اثبات: دنباله‌ی وارسی جست‌وجو برای k مانند دنباله‌ی وارسی برای درج k است.

اگر k برابر $(i+1)$امین کلیدی باشد که درج می‌شود، میانگین وارسی‌ها برابر است با

$$\frac{1}{(1-i/m)} = \frac{m}{m-i}.$$

پس

$$\begin{aligned}\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\ &= \frac{1}{\alpha}(H_m - H_{m-n})\end{aligned}$$

که چنان‌چه می‌دانیم، $H_i = \sum_{j=1}^{i} 1/j$ تابع هارمونی iام است. از طرفی

$$\begin{aligned}\frac{1}{\alpha}(H_m - H_{m-n}) &= \frac{1}{\alpha}\sum_{k=m-n+1}^{m} 1/k \\ &\le \frac{1}{\alpha}\int_{m-n}^{m}(1/x)dx \\ &= \frac{1}{\alpha}\ln\frac{m}{m-n} \\ &= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}.\end{aligned}$$

□

تمرین‌های بخش ۵-۶

۵-۶.۱ درج اعداد ۱۰، ۲۲، ۳۱، ۴، ۱۵، ۲۸، ۱۷، ۸۸ و ۵۹ در یک جدول درهم‌سازی به‌اندازه‌ی $m = ۱۱$ و با استفاده از آدرس‌دهی باز با تابع درهم‌سازی اولیه‌ی $h'(k) = k \bmod m$ را درنظر بگیرید. نتیجه‌ی حاصل از درج این کلیدها را با استفاده از وارسی خطی، وارسی درجه‌ی دو با $c_۱ = ۱$ و $c_۲ = ۳$ و نیز درهم‌سازی دوگانه با $h_۲(k) = ۱ + (k \bmod (m - ۱))$ نشان دهید.

۵-۶.۲ فرض کنید برای جلوگیری از برخوردها از درهم‌سازی دوگانه استفاده می‌کنیم. یعنی، تابع درهم‌سازی $h(k, i) = (h_۱(k) + ih_۲(k)) \bmod m$ را به‌کار می‌بریم. نشان دهید اگر برای مقدار دل‌خواه k بزرگترین مقسوم‌علیه مشترک m و $h_۲(k)$ برابر $d \geq ۱$ باشد، آن‌گاه هر جست‌وجوی ناموفق برای k، قبل از برگشتن به خانه‌ی $h_۱(k)$، تعداد $۱/d$ خانه از جدول درهم‌سازی را بررسی می‌کند. بنابراین اگر m و $h_۲(k)$ نسبت به هم اول باشند (که در آن صورت $d = ۱$ می‌شود)، جست‌وجو ممکن است کل جدول درهم‌سازی را بررسی کند.

۵-۶.۳ یک جدول درهم‌سازی با درهم‌سازی یک‌نوا را درنظر بگیرید. کران بالایی برای امید ریاضی تعداد وارسی‌ها در یک جست‌وجوی ناموفق و نیز موفق پیدا کنید، اگر ضریب بار برابر $\frac{۳}{۴}$ و $\frac{۷}{۸}$ باشد.

⋆ **۵-۶.۴** یک جدول درهم‌سازی با آدرس دهی باز و ضریب بار α را درنظر بگیرید. مقدار غیر صفر α را طوری پیدا کنید که امید ریاضی تعداد وارسی‌ها در یک جست‌وجوی ناموفق دوبرابر امید ریاضی تعداد وارسی‌ها در یک جست‌وجوی موفق باشد.

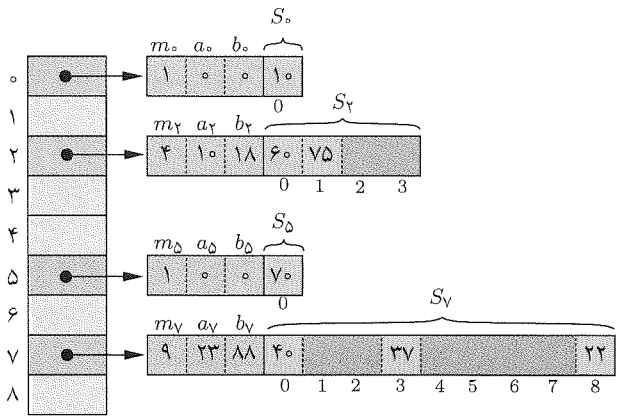
۵-۷ درهم‌سازی کامل

با این‌که روش درهم‌سازی به‌دلیل عمل‌کرد مطلوبش در حالت میانگین مورد استفاده قرار می‌گیرد، اگر مجموعه‌ی کلیدهای داده شده «ایستا[۲۲]» باشند، یعنی از قبل معلوم باشند، می‌توان کاری کرد که در بدترین حالت نیز عمل‌کردی عالی داشته باشند. در برخی از کاربردها با کلیدهای ایستا سروکار داریم؛ مانند، مجموعه‌ی کلمات کلیدی در یک زبان برنامه‌نویسی، یا مجموعه‌ی اسامی موجود در یک لوح فشرده.

یک روش درهم‌سازی را کامل[۲۳] می‌نامیم اگر در بدترین حالت تعداد دست‌رسی‌ها به حافظه برای یک عمل جست‌وجو $\mathcal{O}(۱)$ باشد. ایده‌ی اصلی درهم‌سازی کامل ساده است: استفاده از دو سطح درهم‌سازی و یک تابع درهم‌سازی سراسری برای هر سطح.

---

[۲۲] static
[۲۳] perfect hashing



شکل ۵-۶ مثالی از درهم‌سازی کامل که در آن کلیدهای $K = \{۱۰, ۲۲, ۳۷, ۴۰, ۶۰, ۷۰, ۷۵\}$ با دو تابع درهم‌سازی سراسری به جدول درج می‌شوند.

شکل ۵-۶ این روش را نشان می‌دهد. در این شکل، برای ذخیره‌ی مجموعه‌ی $K = \{۱۰, ۲۲, ۳۷, ۴۰, ۶۰, ۷۰, ۷۵\}$ از درهم‌سازی کامل استفاده شده است. تابع درهم‌سازی اولیه $h(k) = ((ak + b) \bmod p) \bmod m$ است که در آن $a = ۳$، $b = ۴۲$، $p = ۱۰۱$ و $m = ۹$. مثلاً، $h(۷۵) = ۲$. یعنی کلید ۷۵ به درایه‌ی ۲ از جدول T نگاشته می‌شود. جدول ثانویه‌ی درهم‌سازی S_j، تمامی کلیدهای فرستاده شده به درایه‌ی j را ذخیره می‌کند. اندازه‌ی S_j برابر m_j و تابع درهم‌سازی مربوط به آن $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$ است. از آن‌جا که در این مثال $h_۲(۷۵) = ۱$، مقدار ۷۵ در خانه‌ی اول جدول درهم‌سازی ثانویه‌ی $S_۲$ قرار می‌گیرد. نشان می‌دهیم که با انتخاب درست تابع درهم‌سازی، در هیچ‌کدام از جدول‌های درهم‌سازی ثانویه برخوردی رخ نمی‌دهد، بنابراین درهم‌سازی در بدترین حالت به‌زمان ثابت نیاز دارد.

سطح اول در واقع همان درهم‌سازی زنجیره‌ای است: n کلید به‌وسیله‌ی تابع درهم‌سازی h که با دقت از خانواده‌ای از توابع درهم‌سازی سراسری انتخاب شده است به m درایه نگاشته می‌شود. به‌جای ساختن لیستی از کلیدهای فرستاده شده به خانه‌ی j، از جدول درهم‌سازی ثانویه‌ی کوچکی به نام S_j و با تابع درهم‌سازی h_j استفاده می‌کنیم. با انتخاب دقیق h_j، می‌توان تضمین کرد که هیچ برخوردی در سطح دوم رخ نمی‌دهد.

به‌منظور تضمین این ادعا، m_j، اندازه‌ی جدول درهم‌سازی S_j را برابر مجذور تعداد مقادیر فرستاده شده به جدول S_j، یا $n_j^۲$، قرار می‌دهیم. هرچند ممکن است به نظر برسد که

این رابطه‌ی درجه‌ی دو بین m_j و n_j موجب استفاده‌ی زیاد از حافظه می‌شود، اما نشان خواهیم داد که با انتخاب خوب تابع درهم‌سازی اولیه، امید ریاضی کل حافظه‌ی استفاده شده همچنان $\mathcal{O}(n)$ خواهد ماند.

توابع درهم‌سازی خود را از خانواده‌های سراسری توابع درهم‌سازی مربوط به بخش ۵-۵ انتخاب می‌کنیم. تابع سطح اول از خانواده‌ی $\mathcal{H}_{p,m}$ انتخاب می‌شود که در آن p عددی اول و بزرگ‌تر از تمام مقادیر داده‌هاست. مقادیری که به خانه‌ی j فرستاده می‌شوند دوباره به‌وسیله‌ی تابع درهم‌سازی h_j که از خانواده‌ی $\mathcal{H}_{p,m_j}$ انتخاب شده است، به جدول درهم‌سازی S_j با اندازه‌ی m_j ارسال می‌شوند.

کار خود را در دو گام ادامه می‌دهیم. ابتدا برای اطمینان از این‌که در جدول‌های درهم‌سازی ثانویه برخوردی نخواهیم داشت راهی پیدا می‌کنیم. سپس نشان خواهیم داد که امید ریاضی مقدار حافظه‌ای که در کل ــ برای جدول درهم‌سازی اولیه و تمامی جدول‌های درهم‌سازی ثانویه ــ استفاده کرده‌ایم $\mathcal{O}(n)$ است.

قضیه‌ی ۵-۷ اگر n مقدار را در یک جدول درهم‌سازی با اندازه‌ی $m = n^2$ با استفاده از تابع درهم‌سازی h که به‌صورت تصادفی از یک خانواده‌ی سراسری توابع درهم‌سازی انتخاب شده است ذخیره کنیم، آن‌گاه احتمال این‌که برخوردی رخ بدهد کم‌تر از $\frac{1}{2}$ است.

اثبات: $\binom{n}{2}$ جفت مقدار وجود دارند که می‌توانند با هم برخورد داشته باشند. اگر h به‌صورت تصادفی از بین خانواده‌ی سراسری $\mathcal{H}$ از توابع درهم‌سازی انتخاب شده باشد، هر جفت با احتمال $\frac{1}{m}$ برخورد خواهد داشت. فرض کنید X متغیری تصادفی است که تعداد برخوردها را می‌شمارد. با فرض $m = n^2$، امید ریاضی تعداد برخوردها برابر است با

$$E[x] = \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n^2 - n}{2} \cdot \frac{1}{n^2} < \frac{1}{2}.$$

با استفاده از نامساوی مارکوف خواهیم داشت: $Pr\{X \geq t\} \leq E[x]/t$ که اگر t را برابر ۱ قرار دهیم، اثبات کامل می‌شود. □

از توضیح داده شده در قضیه‌ی ۵-۷ که $m = n^2$ است، نتیجه می‌گیریم برای تابعی که به‌صورت تصادفی از $\mathcal{H}$ انتخاب شده باشد، احتمال برخورد نداشتن، بیش‌تر از احتمال برخورد داشتن است. بنابراین با داشتن مجموعه‌ی K با n عنصر، چون K ثابت است، به‌راحتی می‌توانیم با چند انتخاب تصادفی، تابع درهم‌سازی h را پیدا کنیم که هیچ

برخوردی نداشته باشد.

اما هنگامی که n بزرگ باشد، یک جدول درهم‌سازی با اندازه‌ی $m = n^۲$ زیاد از حد بزرگ است. بنابراین از روش دوسطحی درهم‌سازی استفاده می‌کنیم، و از قضیه‌ی ۵-۷ تنها در درهم‌سازی مقادیر درون هر خانه بهره می‌بریم. تابع درهم‌سازی بیرونی یا سطح اول h برای فرستادن مقادیر به $m = n$ خانه استفاده می‌شود. سپس اگر n_j کلید به درایه‌ی j فرستاده شود، برای داشتن ارجاع‌هایی بدون برخورد و در زمان ثابت، از جدول درهم‌سازی ثانویه‌ی S_j با اندازه‌ی $m_j = n_j^۲$ استفاده می‌شود.

اکنون به اثبات این‌که کل حافظه‌ی استفاده شده $O(n)$ است برمی‌گردیم. چون اندازه‌ی m_j مربوط به جدول درهم‌سازی ثانویه‌ی S_j رشدی درجه دو نسبت به n_j دارد، این خطر وجود دارد که مقدار کل حافظه‌ی استفاده‌شده زیاد باشد.

اگر اندازه‌ی جدول سطح اول $m = n$ باشد، نشان می‌دهیم که کل مقدار حافظه‌ی استفاده‌شده برای جدول درهم‌سازی اولیه، جدول‌های ثانویه با اندازه‌های m_j و نیز برای پارامترهای a_j و b_j از $O(n)$ است. دقت کنید که a_j و b_j تابع درهم‌سازی ثانویه‌ی h_j از خانواده‌ی $\mathcal{H}_{p,m_j}$ را معرفی می‌کند (بجز وقتی که $n_j = ۱$ که در آن‌صورت a و b را مساوی ۰ می‌گیریم). لم زیر و نتیجه‌ای که به‌دنبال دارد کران بالایی برای امید ریاضی مجموع کل حافظه‌های ثانویه به‌دست می‌آورد. یک نتیجه‌ی بعدی نیز احتمال این را که مجموع اندازه‌ی تمام جداول درهم‌سازی ثانویه فوق‌خطی شود محدود می‌کند.

قضیه‌ی ۵-۸ اگر تعداد n کلید را به‌وسیله‌ی تابع درهم‌سازی h که به‌تصادف از یک خانواده‌ی سراسری از توابع درهم‌سازی انتخاب شده است، در یک جدول درهم‌سازی با اندازه‌ی $m = n$ ذخیره کنیم، آن‌گاه

$$E\left[\sum_{j=۰}^{m-۱} n_j^۲\right] < ۲n$$

که در آن n_j تعداد کلیدهایی است که به درایه‌ی j نگاشته شده‌اند.

اثبات: کار را با اتحاد زیر، که برای تمام اعداد صحیح غیر منفی درست است، شروع می‌کنیم:

$$a^۲ = a + ۲\binom{a}{۲}$$

داریم

$$\begin{aligned} E\left[\sum_{j=0}^{m-1} n_j^2\right] &= E\left[\sum_{j=0}^{m-1} (n_j + 2\binom{n_j}{2})\right] \\ &= E\left[\sum_{j=0}^{m-1} n_j\right] + E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] \\ &= E[n] + E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] \\ &= n + E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right]. \end{aligned}$$

مقدار $\sum_{j=0}^{m-1} \binom{n_j}{2}$ برابر تعداد کل برخوردهاست. از خواص درهم‌سازی سراسری و از آن‌جا که $m = n$ است، نتیجه می‌گیریم که امید ریاضی این مقدار دست بالا برابر است با

$$\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2}.$$

پس $E\left[\sum_{j=0}^{m-1} n_j^2\right] \leq n + 2\frac{n-1}{2} = 2n - 1 < 2n$. □

نتیجه‌ی ۵–۳ اگر بر اساس درهم‌سازی کامل، تعداد n کلید را به‌وسیله‌ی تابع درهم‌سازی h که به‌تصادف از یک خانواده‌ی سراسری توابع درهم‌سازی انتخاب شده است، در یک جدول درهم‌سازی با اندازه‌ی $m = n$ ذخیره کنیم و (برای $j = 0, 1, \cdots, m-1$) اندازه‌ی هریک از جدول‌های ثانویه را $m_j = n_j^2$ قرار دهیم، آن‌گاه امید ریاضی حافظه‌ی مورد نیاز برای کل جدول‌های ثانویه کم‌تر از $2n$ است.

اثبات: از آن‌جا که برای $j = 0, 1, \cdots, m-1$ $m_j = n_j^2$ است و با استفاده از قضیه‌ی ۵–۸ خواهیم داشت:

$$E\left[\sum_{j=0}^{m-1} m_j\right] = E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n, \qquad (۵–۶)$$

که اثبات را کامل می‌کند. □

نتیجه‌ی ۵-۴ اگر بر اساس درهم‌سازی کامل، n کلید را به‌وسیله‌ی تابع درهم‌سازی h که به‌تصادف از یک خانواده‌ی سراسری توابع درهم‌سازی انتخاب شده است، در یک جدول درهم‌سازی با اندازه‌ی $m = n$ ذخیره کنیم و (برای $j = ۰, ۱, \cdots, m-۱$) اندازه‌ی هریک از جدول‌های ثانویه را برابر $m_j = n_j^۲$ قرار دهیم، آن‌گاه احتمال این‌که میزان کل حافظه‌ی استفاده‌شده برای توابع درهم‌سازی ثانویه از $۴n$ بیش‌تر شود، کم‌تر از $\frac{۱}{۲}$ است.

اثبات: بار دیگر از نامساوی مارکوف استفاده می‌کنیم: $Pr\{X \geq t\} \leq E[X]/t$. در نامساوی ۵-۶ اگر $X = \sum_{j=۰}^{m-۱} m_j$ و $t = ۴n$، داریم

$$\begin{aligned} Pr\{\sum_{j=۰}^{m-۱} m_j \geq ۴n\} &\leq \frac{E\left[\sum_{j=۰}^{m-۱} m_j\right]}{۴n} \\ &< \frac{۲n}{۴n} \\ &= \frac{۱}{۲}. \end{aligned}$$

با استفاده از نتیجه‌ی ۵-۳ می‌بینیم که با امتحان کردن چند تابع درهم‌سازی که به‌تصادف از خانواده‌ی سراسری توابع انتخاب شده باشند، به سرعت می‌توانیم تابعی پیدا کنیم که مقدار مناسبی حافظه استفاده کند. □

تمرین بخش ۵-۷

⋆ **۱.۷-۵** فرض کنید n کلید را با استفاده از آدرس‌دهی باز و درهم‌سازی یک‌نوا در یک جدول درهم‌سازی با اندازه‌ی m درج می‌کنیم. هم‌چنین فرض کنید $p(n,m)$ احتمال این باشد که هیچ برخوردی رخ ندهد. نشان دهید که

$$p(n,m) \leq e^{\frac{-n(n-۱)}{۲m}}.$$

همچنین نشان دهید که با $n > \sqrt{m}$، احتمال رخ ندادن برخورد به‌سرعت به صفر میل می‌کند.

۵-۸ درهم‌سازی پویا

در روش‌های درهم‌سازی که تا کنون دیده‌ایم، فرض می‌شود که تعداد عناصر حداکثر برابر اندازه‌ی جدول (روش باز) یا تقریباً برابر آن است (روش زنجیره‌ای). در حالی که در بسیاری از کاربردها، تعداد عناصر از قبل مشخص نیست و هر لحظه با اعمال درج و حذف این تعداد تغییر می‌کند. بنابراین نمی‌توان کران بالای خوبی برای اندازه‌ی جدول درهم‌سازی پیش‌بینی کرد که حافظه تلف نشود.

در روش «درهم‌سازی پویا[۲۴]» اندازه‌ی جدول درهم‌سازی برحسب نیاز زیاد و کم می‌شود. در این بخش نشان می‌دهیم که اگر زمان زیاد و کم شدن اندازه‌ی جدول را به‌درستی تعیین کنیم، و در صورت انتخاب تابع درهم‌سازی یک‌نوا، هزینه‌ی سرشکن‌شده‌ی هر یک از اعمال حذف و درج هنوز $\mathcal{O}(1)$ باقی می‌ماند. در تحلیل این داده‌ساختار از روش تابع پتانسیل که در بخش ۳-۷-۲ گفته شد استفاده می‌کنیم.

در پیاده‌سازی، فرض می‌کنیم که T جدول درهم‌سازی، $table[T]$ مؤلفه‌ی جدول، $size[T]$ اندازه‌ی جدول، و $num[T]$ تعداد عناصر موجود در جدول T باشد. اعمالی که بر روی این جدول انجام می‌دهیم به‌قرار زیرند:

- درج ساده: جدول جای خالی دارد و عنصر مانند قبل در جدول درج می‌شود.
- درج با گسترش[۲۵]: قبل از درج، جدول پر است. در این‌صورت اندازه‌ی جدول دو برابر می‌شود، عناصر موجود به جدول جدید منتقل می‌شوند و سپس عنصر مورد نظر در جدول جدید درج می‌شود.
- حذف ساده: حذفی که موجب تغییر در اندازه‌ی جدول نمی‌شود.
- حذف و فشرده‌سازی[۲۶]: پس از حذف نسبت تعداد عناصر به‌اندازه‌ی جدول کم می‌شود و لازم است اندازه‌ی جدول نصف شود و همه‌ی عناصر موجود به جدول جدید منتقل شوند.

برای تحلیل، نمادهای زیر را تعریف می‌کنیم:

- c_i هزینه‌ی عمل i ام،

---

[۲۴] dynamic hashing
[۲۵] expansion
[۲۶] compaction

- s_i اندازه‌ی جدول پس از عمل i ام،
- n_i تعداد عناصر جدول پس از عمل i ام، و
- $0 \leq \alpha_i = \frac{n_i}{s_i} \leq 1$ ضریب بار پس از عمل i ام.

۵-۸-۱ فقط درج

اگر در درهم‌سازی پویا فقط درج داشته باشیم، فرض می‌کنیم که اندازه‌ی جدول همیشه توانی از ۲، و در ابتدا برابر صفر است. رویه‌ی TABLE-INSERT کار درج در این جدول را انجام می‌دهد.

```
TABLE-INSERT (T, x)
 1  if size[T] = 0
 2     then allocate table[T] with one slot
 3          size[T] ← 1
 4  if num[T] = size[T]
 5     then allocate new-table with 2.size[T] slots
 6          insert all items from table[T] into new-table
 7          free table[T]
 8          table[T] ← new-table
 9          size[T] ← 2 · size[T]
10  insert x into table[T]
11  num[T] ← num[T] + 1
```

اگر اندازه‌ی جدول صفر باشد، جدولی با یک درایه ایجاد می‌شود و x را در آن قرار می‌دهد. وگرنه، اگر تعداد عناصر موجود در جدول برابر اندازه‌ی جدول بود، یعنی قبل از درج، $\alpha_{i-1} = 1$، در آن صورت «درج با گسترش» انجام می‌شود. برای این‌کار، جدول جدید *new-table* با اندازه‌ی ۲ برابر جدول موجود (*table*) ایجاد می‌شود، سپس همه‌ی عناصر موجود در *table* با تابع درهم‌سازی جدید (با احتساب اندازه‌ی جدید جدول) در *new-table* درج می‌شود، جدول قبلی آزاد می‌شود و جدول جدید تغییر نام داده و پارامترهای آن تنظیم می‌شود. در سطرهای ۱۰ و ۱۱ عمل واقعی درج (یا در جدول قبلی یا

در جدول گسترش یافته) انجام می‌شود.

به‌نظر می‌رسد که هزینه‌ی انجام درج، به‌علت هزینه‌ی گسترش یک جدول به‌اندازه‌ی n، زیاد است، در بدترین حالت $O(n)$ است. این تحلیل در بدترین حالت درست است، اما روشن است که بعد از هر گسترش جدول، تعداد زیادی از درج‌ها با همان هزینه‌ی $O(1)$ انجام می‌شود (البته با فرض آن‌که تابع درهم‌سازی ساده و یک‌نواست). این مشاهده ما را به تحلیل سرشکنی که در بخش ۳-۷ گفته شد، هدایت می‌کند.

تحلیل سرشکنی به‌روش انبوهه

جدول زیر مثالی از نحوه‌ی گسترش جدول برای درج‌های متوالی است که در آن اندازه‌ی جدول و تعداد گسترش‌های انجام شده از ابتدا، برای درج iام نشان داده شده است.

i	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱	...
s_i	۱	۲	۴	۴	۸	۸	۸	۸	۱۶	۱۶	۱۶	۱۶
تعداد گسترش‌ها	۱	۲	۳	۳	۳	۴	۴	۴	۵	۵	۵	۵

روشن است که تعداد کل گسترش‌ها برای n درج، برابر $\lceil \lg n \rceil + ۱$ است. هزینه‌ی درج iام اگر گسترشی در کار نباشد برابر ۱ است و اگر گسترش انجام شود برابر $n_{i-۱} + ۱$ است، چرا که قبل از درج آخر باید $n_{i-۱}$ عنصر موجود در جدول قبلی به جدول جدید منتقل شوند. یعنی

$$c_i = \begin{cases} ۱ & \text{درج ساده} \\ n_{i-۱} + ۱ & \text{درج با گسترش} \end{cases}$$

بنابراین مجموع هزینه‌ی n بار عمل درج برابر است با

$$\sum_{i=۱}^{n} c_i = n + \sum_{k=۰}^{\lceil \lg n \rceil} ۲^k = n + \frac{۲^{\lceil \lg n \rceil + ۱} - ۱}{۲ - ۱} \leq ۳n.$$

یعنی هزینه‌ی سرشکن‌شده‌ی هر درج حداکثر ۳ واحد یا $O(1)$ است.

تحلیل به روش حساب‌داری

از تحلیل فوق حدس می‌زنیم که به‌روش حساب‌داری باید به‌ازای هر درج ۳ ریال خرج کنیم: ۱ ریال آن‌را صرف خود عمل درج می‌کنیم و ۱ ریال دیگر را بر روی عنصر درج‌شده

قرار می‌دهیم. سپس با استقرا نشان می‌دهیم که می‌توانیم ۱ ریال باقی‌مانده را طوری بر روی یک عنصر «نظیر» عنصر درج‌شده قرار دهیم تا در ابتدای هر گسترش، هر کدام از عناصر یک ریال داشته باشند (یعنی گسترش بتواند مجانی انجام شود) و همه‌ی پول‌های روی عناصر صرف عمل گسترش شود و هیچ عنصری پولی نداشته باشد.

اثبات این امر بسیار ساده است: در ابتدا این ادعا درست است. فرض کنید که بلافاصله پس از یک گسترش (و قبل از درج واقعی عنصر) اندازه‌ی جدول برابر 2^k و تعداد عناصر موجود 2^{k-1} باشد. بنابراین برای درج همین تعداد عنصر تا گسترش بعدی، فضا موجود است. چنان‌چه گفته شد، هنگام درج هر کدام از این عناصر ۱ ریال بر روی آن و ۱ ریال بر روی یکی از عناصر موجود در جدول قرار می‌دهیم. بنابراین وقتی این جدول پر می‌شود، هر عنصر جدول ۱ ریال دارد، پس در درج بعدی، عمل گسترش هزینه‌ای ندارد و بنابراین گام استقرا هم ثابت می‌شود.

تحلیل به‌روش تابع پتانسیل

برای این مسئله تابع پتانسیل Φ تعریف می‌کنیم که $\Phi(D_i) \geq 0$. در این‌صورت، اگر عمل i موجب گسترش شود، $\Phi(D_{i-1}) \geq s_{i-1}$. تعریف می‌کنیم $\Phi(D_i) = 2n_i - s_i$. به‌وضوح $\Phi(D_i) \geq 0$ و اگر عمل i موجب گسترش شود، داریم

$$\begin{aligned} s_{i-1} &= n_{i-1} \\ \Phi(D_{i-1}) &= s_{i-1} \\ s_i &= 2s_{i-1} \\ n_i &= n_{i-1} + 1. \end{aligned}$$

بنابراین مطابق تعریف ۳-۴۰ از بخش ۳-۷-۲،

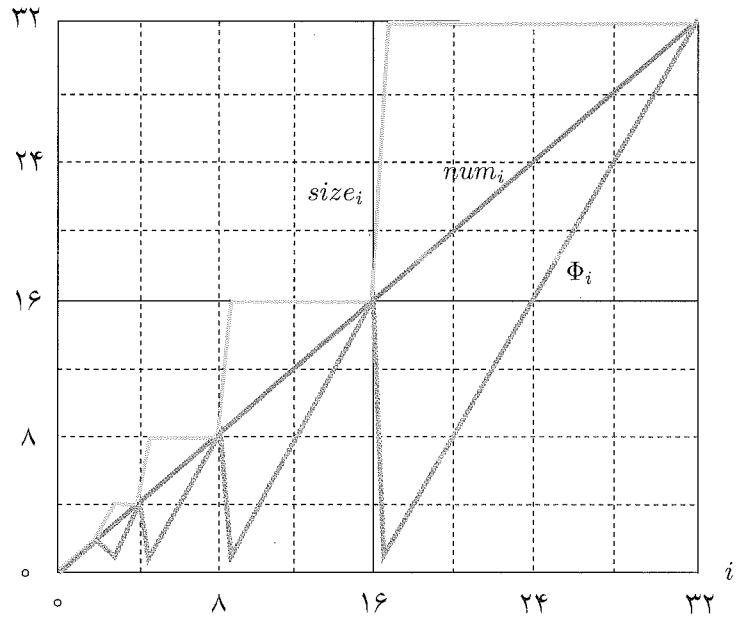
$$\begin{aligned} \hat{c}_i &= 1 + \Phi(D_i) - \Phi(D_{i-1}) \ . \\ &= n_{i-1} + 1 + [2n_i - s_i] - [2n_{i-1} - s_{i-1}] \\ &= n_{i-1} + 1 + [2n_{i-1} + 2 - 2s_{i-1}] - [2n_{i-1} - s_{i-1}] \\ &= n_{i-1} - s_{i-1} + 1 + 2 = 3. \end{aligned}$$

برای عمل درج عادی i ام، $s_i = s_{i-1}$ و $n_i = n_{i-1} + 1$، پس

$$\hat{c}_i = 1 + \Phi(D_i) - \Phi(D_{i-1})$$

$$
\begin{aligned}
&= ۱ + [۲n_i - s_i] - [۲n_{i-۱} - s_{i-۱}] \\
&= ۱ + [۲n_{i-۱} + ۲ - s_{i-۱}] - [۲n_{i-۱} - s_{i-۱}] \\
&= ۱ + ۲ = ۳,
\end{aligned}
$$

یعنی در هر صورت هزینه‌ی سرشکن‌شده‌ی هر درج حداکثر ۳ واحد یا $O(۱)$ است. شکل ۵-۷ منحنی‌های اندازه، تعداد عناصر و تابع پتانسیل را به‌ازای درج‌های متوالی مختلف نشان می‌دهد.



شکل ۵-۷ منحنی‌های اندازه، تعداد عناصر و تابع پتانسیل به‌ازای درج‌های مختلف در جدول درهم‌سازی پویا، در حالتی‌که عمل حذف نداشته باشیم.

۵-۸-۲ درج و حذف با هم

در صورتی که بخواهیم عمل حذف را نیز در کنار درج در نظر بگیریم، باید عمل فشرده‌سازی جدول را که با یک حذف انجام می‌شود، تعریف کنیم. در این حالت، عنصر مورد نظر را حذف می‌کنیم، سپس اگر تعداد عناصر موجود نسبت به اندازه‌ی جدول از

حدی کم‌تر شد، اندازه‌ی جدول را نصف می‌کنیم و همه‌ی عناصر دیگر را به جدول جدید منتقل می‌کنیم.

فرض کنید عمل فشرده‌سازی را وقتی انجام می‌دهیم که

- عمل i حذف باشد و
- ضریب بار $\alpha_{i-1} = \frac{n_{i-1}}{s_{i-1}} = \frac{1}{2}$ باشد.

در این‌صورت می‌توان سناریوی زیر را تصور کرد (I برای درج و D برای حذف):

$$\underbrace{I, \ldots, I}_{n/2}, \underbrace{I, D, D, I, I, D, D, \ldots}_{n/2}$$

در این صورت، پس از $n/2$ بار درج، با یک درج دیگر عمل گسترش انجام می‌شود و با یک عمل حذف پس از آن، عمل فشرده‌سازی، و نیز در سومین درج عمل گسترش و در چهارمین حذف عمل فشرده‌سازی، و این کار ادامه می‌یابد. یعنی در کل به‌تعداد $\Theta(n)$ بار عمل گسترش و فشرده‌سازی انجام می‌شود، که هر کدام به‌اندازه‌ی $\Theta(n)$ هزینه دارد. پس در کل هزینه $\Theta(n^2)$ می‌شود که قابل قبول نیست.

برای حل این مشکل، عمل فشرده‌سازی را وقتی انجام می‌دهیم که عمل iام، حذف و نیز ضریب بار برابر $\alpha_{i-1} = \frac{n_{i-1}}{s_{i-1}} \leq \frac{1}{4}$ باشد.

در این صورت، اگر با حذف iام فشرده‌سازی انجام شود، داریم $n_i = n_{i-1} - 1$ و $s_i = \frac{s_{i-1}}{2}$. پس $\alpha_i = \frac{2(n_{i-1}-1)}{s_{i-1}} < \frac{1}{2}$ $\frac{1}{4} <$.

تحلیل

به‌دلیل پیچیدگی این تحلیل، از روش تابع پتانسیل استفاده می‌کنیم. باید تابع پتانسیلی تعریف کنیم که ویژگی‌های زیر را داشته باشد:

- درست قبل از گسترش برای عمل درج iام، داشته باشیم $\Phi(T_{i-1}) \geq n_{i-1}$ که عمل گسترش بدون هزینه انجام شود.
- درست قبل از فشرده‌سازی برای عمل حذف iام داشته باشیم: $\Phi(T_{i-1}) \geq n_{i-1}$.
- $\Phi(T_i) \geq 0$

بنابراین، تابع پتانسیل زیر را در نظر می‌گیریم:

$$\Phi(T_i) = \begin{cases} ۲n_i - s_i & \alpha_i \geq \frac{۱}{۲} \\ s_i/۲ - n_i & \alpha_i < \frac{۱}{۲} \end{cases} \qquad (۷-۵)$$

وضعیت لحظه‌ی قبل از گسترش را در نظر بگیرید: داریم $\alpha_{i-۱} = ۱$، $s_{i-۱} = n_{i-۱}$ و $\Phi(T_{i-۱}) = ۲n_{i-۱} - s_{i-۱} = n_{i-۱}$. اگر حالت قبل از فشرده‌سازی را در نظر بگیریم، داریم $\alpha_{i-۱} = ۱/۴$، $s_{i-۱} = ۴n_{i-۱}$ و $\Phi(T_{i-۱}) = s_{i-۱}/۲ - n_{i-۱} = n_{i-۱}$.

پس این تابع مناسبی است. همچنین، چون $\frac{۱}{۴} \leq \alpha_i \leq ۱$، می‌توان نتیجه گرفت که $\Phi(T_i) \geq ۰$. اما برای تحلیل، باید همه‌ی حالت‌ها را در نظر بگیریم.

تحلیل: درج همراه با گسترش

طبق فرض داریم $\alpha_{i-۱} = ۱$، یعنی: $s_{i-۱} = n_{i-۱}$. همچنین، چون $\alpha_i > ۱/۲$، داریم: $s_i = ۲s_{i-۱} = ۲n_{i-۱} = ۲(n_i - ۱)$. بنابراین، $\Phi(T_{i-۱}) = ۲n_{i-۱} - s_{i-۱}$ و $\Phi(T_i) = ۲n_i - s_i$. و می‌دانیم: $s_i = ۲s_{i-۱}$ و $n_i = n_{i-۱} + ۱$. پس،

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-۱}) \\ &= n_{i-۱} + ۱ + [۲n_i - s_i] - [۲n_{i-۱} - s_{i-۱}] \\ &= n_{i-۱} + ۱ + [۲n_{i-۱} + ۲ - ۲s_{i-۱}] - [۲n_{i-۱} - s_{i-۱}] \\ &= n_{i-۱} - s_{i-۱} + ۱ + ۲ = ۳. \end{aligned}$$

تحلیل: درج عادی

برای این حالت می‌دانیم $\frac{۱}{۴} < \alpha_{i-۱} \leq ۱$، $\frac{۱}{۴} < \alpha_i < ۱$، $s_i = s_{i-۱}$ و $n_i = n_{i-۱} + ۱$. سه حالت مختلف را باید در نظر بگیریم: یکی $\alpha_{i-۱} \geq \frac{۱}{۲}$، که در آن‌صورت $\alpha_i \geq \frac{۱}{۲}$. بنابراین مانند حالت گسترش، داریم:

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-۱}) \\ &= ۱ + [۲n_i - s_i] - [۲n_{i-۱} - s_{i-۱}] \\ &= ۱ + [۲n_{i-۱} + ۲ - s_{i-۱}] - [۲n_{i-۱} - s_{i-۱}] \\ &= ۱ + ۲ = ۳. \end{aligned}$$

یعنی هزینه‌ی سرشکن‌شده ثابت است.

حالت دیگر آن‌که $\alpha_{i-1} < \frac{1}{2}$ و $\alpha_i < \frac{1}{2}$. در این‌صورت،

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + [s_i/2 - n_i] - [s_{i-1}/2 - n_{i-1}] \\ &= 1 + [s_i/2 - n_i] - [s_i/2 - (n_i - 1)] \\ &= 0.\end{aligned}$$

حالت سوم آن است که $\alpha_{i-1} < \frac{1}{2}$ و $\alpha_i \geq \frac{1}{2}$. در این حالت، با توجه به تعریف تابع پتانسیل داریم

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + [2n_i - s_i] - [s_{i-1}/2 - n_{i-1}] \\ &= 1 + [2(n_{i-1} + 1) - s_{i-1}] - [s_i/2 - (n_i - 1)] \\ &= 3n_{i-1} - \frac{3}{2}s_{i-1} + 3 \\ &= 3\alpha_{i-1}s_{i-1} - \frac{3}{2}s_{i-1} + 3 \\ &< \frac{3}{2}s_{i-1} - \frac{3}{2}s_{i-1} + 3 \\ &= 3.\end{aligned}$$

یعنی در همه‌ی حالت‌ها، هزینه‌ی سرشکن‌شده ثابت است.

تحلیل: حذف بدون فشرده‌سازی

اگر عمل i حذف عادی باشد، در آن‌صورت دو حالت را در نظر می‌گیریم:

حالت اول: $\alpha_i < \frac{1}{2}$، یعنی $\alpha_{i-1} < \frac{1}{2}$. و نیز داریم $s_i = s_{i-1}$ و $n_i = n_{i-1} - 1$. پس

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + [s_i/2 - n_i] - [s_{i-1}/2 - n_{i-1}] \\ &= 1 + [s_i/2 - n_i] - [s_i/2 - (n_i + 1)] \\ &= 0.\end{aligned}$$

حالت دوم: $\alpha_{i-1} \geq \frac{1}{2}$ (این حالت به عنوان تمرین ۵-۸.۳ داده شده است).

تحلیل: حذف همراه با فشرده‌سازی

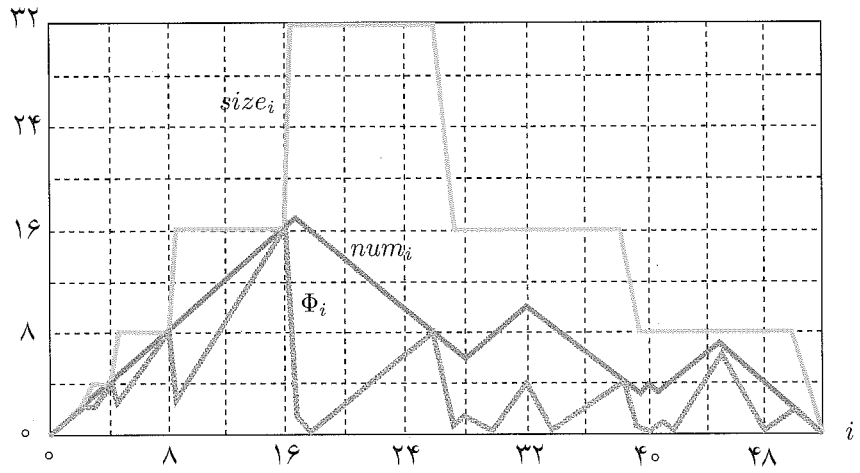
اگر عمل حذف i موجب فشرده‌سازی شود، در آن‌صورت $\alpha_i < \frac{۱}{۲}$، یعنی $\alpha_{i-۱} = \frac{۱}{۴}$. همچنین می‌دانیم که

$$s_i = s_{i-۱}/۲ \text{ و } ۱ - n_{i-۱} = n_i.$$

پس،

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-۱}) \\
&= n_{i-۱} + ۱ + [s_i/۲ - n_i] - [s_{i-۱}/۲ - n_{i-۱}] \\
&= n_{i-۱} + [s_{i-۱}/۴ - (n_{i-۱} - ۱)] - [s_{i-۱}/۲ - n_{i-۱}] \\
&= n_{i-۱} + [n_{i-۱} - (n_{i-۱} - ۱)] - [۲n_{i-۱} - n_{i-۱}] \\
&= ۱.
\end{aligned}$$

شکل ۵–۸ منحنی‌های اندازه، تعداد عناصر و تابع پتانسیل را به‌ازای درج و حذف‌های مختلف در جدول درهم‌سازی پویا نشان می‌دهد. چنان‌چه می‌بینیم، تابع پتانسیل همواره نامنفی است و برای هر عمل گسترش و فشرده‌سازی به‌اندازه‌ی کافی در مخزن پول هست.



شکل ۵–۸ منحنی‌های اندازه، تعداد عناصر و تابع پتانسیل به‌ازای درج و حذف‌های مختلف در جدول درهم‌سازی پویا.

تمرین‌های بخش ۸-۵

۱.۸-۵ در جدول درهم‌سازی پویا با عمل درج و حذف یک عنصر، فرض کنید که عمل فشرده‌سازی را وقتی انجام می‌دهیم که عمل iام حذف و ضریب بار $\alpha_{i-1} \leq \frac{1}{4}$ باشد. با تعریف یک تابع پتانسیل نشان دهید که هزینه‌ی سرشکن‌شده‌ی عمل حذف (چه عادی باشد و یا موجب فشرده‌سازی شود) $\mathcal{O}(1)$ است.

۲.۸-۵ فرض کنید که می‌خواهیم یک جدول درهم‌سازی باز و پویا را پیاده‌سازی کنیم. چرا ممکن است جدول را پر فرض کنیم در حالی که ضریب بار آن اکیداً کم‌تر از ۱ است؟ به اختصار توضیح دهید که چگونه می‌توان در چنین جدولی عمل درج را انجام داد به‌طوری‌که میانگین هزینه‌ی سرشکن‌شده برای هر عمل درج $\mathcal{O}(1)$ شود. چرا میانگین مقدار هزینه‌ی واقعی هر درج لزوماً برای هر عمل $\mathcal{O}(1)$ نیست؟

۳.۸-۵ نشان دهید که اگر $\alpha_{i-1} \geq 1/2$ و عمل iام بر روی یک جدول پویا حذف باشد، آن‌گاه کران بالای هزینه‌ی سرشکن‌شده‌ی این عمل با توجه به تابع پتانسیل ۵-۷ ثابت است.

۴.۸-۵ در فشرده‌سازی جدول که اندازه‌ی آن را هنگامی که ضریب بار کم‌تر از ۱/۴ می‌شود نصف می‌کنیم، فرض کنید که اگر ضریب بار کم‌تر از ۱/۳ شود اندازه‌ی جدول را این‌بار در ۲/۳ ضرب می‌کنیم. با استفاده از تابع پتانسیل

$$\Phi(T) = |1 \cdot num[T] - size[T]|$$

نشان دهید که در این حالت نیز کران بالای هزینه‌ی سرشکن‌شده‌ی عمل حذف ثابت است.

تمرین‌های فصل ۵

۱.۵ در روش درهم‌سازی باز با وارسی خطی، تابع درهم‌سازی برای عناصر به‌صورت زیر است:

```
key:  A B C D E F G
hash: 3 5 3 4 5 6 3
```

اگر جدول درهم‌سازی به اندازه‌ی ۷ در ابتدا تهی باشد، به چند حالت مختلف این عناصر می‌توانند در جدول درج شوند تا در نهایت جدول H[0..6]=[E F G A C B D] تولید شود؟
این مسئله را در حالت کلی چگونه حل می‌کنید؟

⋆ **۲.۵ کران طولانی‌ترین وارسی**
از یک جدول درهم‌سازی با اندازه‌ی m برای نگه‌داری n عنصر استفاده کرده‌ایم. فرض کنید که $n \leq m/2$. برای مشکل برخوردها از آدرس‌دهی باز استفاده کرده‌ایم.

الف) با فرض تابع درهم‌سازی یک‌نوا، نشان دهید که برای $i = 1, 2, \cdots, n$، احتمال این‌که iامین درج بیش از k وارسی نیاز داشته باشد حداکثر 2^{-k} است.

ب) نشان دهید که برای $i = 1, 2, \ldots, n$ احتمال این‌که iامین درج بیش از $2\lg n$ وارسی نیاز داشته باشد، حداکثر $1/n^2$ است.

فرض کنید متغیر تصادفی X_i بیان‌گر تعداد وارسی‌های لازم برای iامین درج باشد. در قسمت (ب) نشان دادید $Pr\{X_i > 2\lg n\} \leq 1/n$. هم‌چنین فرض کنید متغیر تصادفی $X = \max_{1 \leq i \leq n} X_i$ نشان‌دهنده‌ی بیشینه‌ی تعداد وارسی‌های لازم برای کل درج‌ها است.

پ) نشان دهید $Pr\{X > 2\log n\} \leq 1/n$.

ت) نشان دهید که $E[X]$، امید ریاضی طول بزرگ‌ترین دنباله‌ی وارسی‌ها از $\mathcal{O}(\lg n)$ است.

⋆ **۳.۵ کران اندازه‌ی جدول در روش زنجیره‌ای**
فرض کنید یک جدول درهم‌سازی به‌اندازه‌ی n داریم که در آن مشکل برخوردها با روش زنجیره‌ای حل شده است. n عنصر را در این جدول درج کرده‌ایم به‌طوری‌که هر عنصر با احتمال مساوی می‌تواند به‌تمام درایه‌ها برود. اگر M بیشینه‌ی تعداد عناصر در یک درایه پس از درج تمام عناصر باشد، می‌خواهیم کران بالای $\mathcal{O}(\log n / \log\log n)$ را برای $E[M]$، امید ریاضی مقدار M، ثابت کنیم.

الف) ثابت کنید که Q_k، احتمال این که دقیقاً k عنصر به یک درایه‌ی مشخص فرستاده شوند، برابر است با

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}$$

ب) اگر احتمال $M = k$، یا احتمال این‌که درایه‌ای را که دارای بیش‌ترین تعداد عنصر است k عنصر داشته باشد، را با P_k نمایش دهیم، نشان دهید $P_k \leq nQ_k$.

پ) با استفاده از تقریب استرلینگ ثابت کنید $Q < e^k/k^k$.

ت) ثابت کنید عدد 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 $k \geq k_0 = c \lg n / \lg \lg n$ داریم $P_k < 1/n^2$.

ث) نشان دهید

$$E[M] \leq Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n + Pr\left\{M \leq \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n}$$

و نتیجه بگیرید که $E[M] = O(\lg n / \lg \lg n)$.

⋆ ۴.۵ وارسی درجه‌ی ۲

فرض کنید می‌خواهیم درون یک جدول درهم‌سازی با خانه‌های $\{0, 1, \cdots, m-1\}$ دنبال عنصر داده‌شده‌ی k بگردیم. فرض کنید تابع درهم‌سازی h را که فضای عناصر ما را به $\{0, 1, \ldots, m-1\}$ می‌فرستد نیز در اختیار داریم. روش جست‌وجوی ما در زیر آمده است:

۱) مقدار $i \leftarrow h(k)$ را محاسبه کن و $j \leftarrow 0$ قرار بده.

۲) در درایه‌ی i به‌دنبال k بگرد. اگر آن‌را یافتی یا اگر درایه خالی بود جست‌وجو را متوقف کن.

۳) $j \leftarrow (j+1) \bmod m$ و $i \leftarrow (i+j) \bmod m$ قرار بده و به مرحله‌ی ۲ برگرد.

فرض کنید که m توانی از ۲ است.

الف) نشان دهید که این روش یک مورد از روش کلی «وارسی درجه‌ی ۲» است. این‌کار را با تعیین مقدارهای مناسب برای ثابت‌های c_1 و c_2 در رابطه‌ی ۵-۵ انجام دهید.

ب) ثابت کنید که این الگوریتم در بدترین حالت هر درایه از جدول را وارسی می‌کند.

⋆ ۵.۵ درهم‌سازی kـسراسری

فرض کنید $\mathcal{H} = \{h\}$ خانواده‌ای از توابع درهم‌سازی باشد که در آن هر تابع h، فضای U از عناصر را به $\{0, 1, \cdots, m-1\}$ می‌فرستد. می‌گوییم $\mathcal{H}$، kـسراسری است اگر برای هر دنباله‌ی ثابت از k عنصر متمایز $\langle x^{(1)}, x^{(2)}, \ldots, x^{(k)} \rangle$ و برای هر h که به‌صورت تصادفی از $\mathcal{H}$ انتخاب شده باشد، دنباله‌ی $\langle h(x^{(1)}), h(x^{(2)}), \ldots, h(x^{(k)}) \rangle$ بتواند با احتمال یکسان هر یک از m^k دنباله‌ی با عناصر انتخاب شده از $\{0, 1, \ldots, m-1\}$ باشد.

الف) نشان دهید اگر $\mathcal{H}$، ۲ـسراسری باشد آن‌گاه سراسری نیز هست.

ب) فرض کنید U مجموعه‌ی n تایی‌ها با مقادیر انتخاب‌شده از Z_p باشد و B را برابر Z_p در نظر بگیرید که p عددی اول است. برای هر n تایی $a = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ با مقادیر عضو Z_p و هر $b \in Z_p$، تابع درهم‌سازی $h_{a,b} : U \to B$ را روی ورودی n تایی $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ به‌صورت زیر تعریف کنید:

$$h_{a,b}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$

اگر $\mathcal{H} = \{h_{a,b}\}$ باشد، نشان دهید که $\mathcal{H}$، ۲ـسراسری است.

پ) فرض کنید آلیس و باب به‌طور مخفیانه برای خود تابع درهم‌سازی $h_{a,b}$ را از خانواده‌ی ۲ـسراسری $\mathcal{H}$ از توابع درهم‌سازی انتخاب کرده‌اند. آلیس بعداً پیام m را (که $m \in U$) از طریق اینترنت به باب می‌فرستد. او برای تصدیق پیامش، پیوست $t = h_{a,b}(m)$ را نیز ضمیمه می‌کند. باب پس از گرفتن زوج (m,t)، بررسی می‌کند که آیا t برابر $h_{a,b}(m)$ است یا نه. فرض کنید یک مزاحم در سر راه، (m,t) را استراق سمع کرده و برای فریبِ باب، آن‌را با زوج متفاوت (m',t') عوض می‌کند. نشان دهید هر قدر هم قدرت محاسباتی مزاحم زیاد باشد، احتمال این‌که او موفق شود و باب (m',t') را بپذیرد حداکثر $1/p$ است.

۶.۵ فرض کنید n مقدار را با استفاده از آدرس‌دهی باز و درهم‌سازی یک‌نوا در یک جدول درهم‌سازی به اندازه‌ی m وارد کنیم. هم‌چنین فرض کنید $p(n,m)$ احتمال این باشد که هیچ برخوردی رخ ندهد. نشان دهید که $p(n,m) \leq e^{\frac{-n(n-1)}{2m}}$ است. (راهنمایی: از معادله‌ی $e^x \geq 1 + x$ استفاده کنید.) نشان دهید که وقتی n از $\sqrt{m}$ بیش‌تر می‌شود، احتمال رخ ندادن برخورد به‌سرعت به صفر میل می‌کند.

فصل

۶

مرتب‌سازی و مرتبه‌ی آماری

در فصل ۳ با چند الگوریتم مرتب‌سازی آشنا شدیم: مرتب‌سازی درجی (مستقیم و دودویی) و مرتب‌سازی حبابی که برای مرتب‌سازی n عنصر در عمل هزینه‌ای برابر $\mathcal{O}(n^2)$ داشتند، و نیز مرتب‌سازی ادغامی که با صرف $\mathcal{O}(n)$ حافظه‌ی اضافی، در زمان $\mathcal{O}(n \lg n)$ عناصر را مرتب می‌کرد.

در این فصل، ابتدا پس از تعریف دقیقی از مسئله، گونه‌های مختلف مرتب‌سازی را بیان می‌کنیم، و کران پایین الگوریتم‌های مرتب‌سازی را چه در بدترین حالت و چه در حالت میانگین به‌دست می‌آوریم. نشان می‌دهیم که در شرایط خاصی مرتب‌سازی را می‌توان در زمان خطی انجام داد و در بخش‌های ۶-۲ الگوریتم‌های مرتب‌سازی خطی ـ شمارشی، سطلی و مبنایی را ارائه و تحلیل می‌کنیم.

در بخش ۶-۳ چند الگوریتم مرتب‌سازی مقایسه‌ای را بررسی می‌کنیم که کار خود را با مقایسه‌ی کلیدهای عناصر انجام می‌دهند. در بخش ۶-۳-۱ مرتب‌سازی سریع را مورد بررسی دقیق قرار می‌دهیم که در حالت میانگین، سریع‌ترین الگوریتم مرتب‌سازی است. در بخش ۶-۳-۳ مرتب‌سازی هرمی بیان می‌شود که رفتاری بهینه در بدترین حالت دارد. مرتب‌سازی فورد ـ جانسون که در بخش ۶-۴ ارائه می‌شود، تعداد مقایسه‌هایی که بین کلیدهای عناصر انجام می‌دهد، به مقدار بهینه نزدیک است.

در بخش ۶-۵، برای مسئله‌ی مرتبه‌ی آماری الگوریتم‌هایی مبتنی بر مرتب‌سازی سریع ارائه می‌کنیم که یکی در حالت میانگین و دیگری در بدترین حالت بهینه است.

در انتها و در بخش ۶-۶، چند الگوریتم مرتب‌سازی خارجی را ارائه می‌دهیم. این الگوریتم‌ها زمانی استفاده می‌شوند که امکان ذخیره‌ی کل عناصر در حافظه‌ی داخلی نباشد.

۶-۱ دسته‌بندی و کران پایین

مرتب‌سازی را به‌طور دقیق به‌صورت زیر تعریف می‌کنیم:

مسئله‌ی مرتب‌سازی: n عنصر با کلیدهای $a_1, a_2, \cdots, a_n$ و رابطه‌ی ترتیب کامل[1] به‌نام $\leq$ داده شده‌اند. مسئله‌ی مرتب‌سازی، یافتن جای‌گشت π از دنباله‌ی $\langle 1, 2, \cdots, n \rangle$ داده‌شده است به‌طوری‌که

$$a_{\pi_1} \leq a_{\pi_2} \leq \cdots \leq a_{\pi_n},$$

که در آن π_i، iامین عدد در جای‌گشت π است.

الگوریتم‌های مرتب‌سازی[2] را از سه جنبه‌ی زیر می‌توان دسته‌بندی کرد:

۱. از نظر موقعیت داده‌ها در زمان مرتب‌سازی،

- **مرتب‌سازی داخلی**[3]، که در آن همه‌ی داده‌ها برای مرتب‌سازی در حافظه قرار دارند. بنابراین دست‌یابی به داده‌ها سریع است.
- **مرتب‌سازی خارجی**[4]، که برای مرتب‌سازی، تنها مقدار ثابتی از داده‌ها در حافظه‌ی اصلی است و دست‌رسی به بقیه‌ی داده‌ها تنها از طریق حافظه‌ی جانبی ــ که بسیار کند است ــ امکان‌پذیر می‌باشد. در این الگوریتم‌ها تعداد دست‌یابی‌ها به حافظه‌ی جانبی، شاخص زمان اجراست.

۲. از نظرحفظ ترتیب نسبی عناصر،

- **مرتب‌سازی پایدار**[5]، که در آن ترتیب نسبی عناصری که کلید یک‌سان دارند، قبل و بعد از عمل مرتب‌سازی تغییر نکند.

---

[1] total order یک رابطه‌ی ترتیب جزیی است اگر نامتقارن، بازتابی و تراگذری باشد. رابطه‌ی ترتیب جزیی R بر روی مجموعه‌ی A یک ترتیب کامل است، اگر برای هر $a, b \in A$ داشته باشیم، aRb یا bRa.

[2] sorting algorithms

[3] internal sorting algorithms

[4] external sorting algorithms

[5] stable

- **مرتب‌سازی ناپایدار**[۶]، که در آن ترتیب نسبی عناصر با کلیدهای یک‌سان در انتها ممکن است تغییر کند.

۳. از نظر نحوه‌ی مرتب‌سازی داده‌ها،

- **مرتب‌سازی مقایسه‌ای**[۷]، که بر اساس مقایسه‌ی کلیدهای عناصر عمل می‌کند. یک مرتب‌سازی مقایسه‌ای اطلاعات دیگری از داده‌های ورودی ندارد و فقط با انجام مقایسه بین کلیدهای عناصر می‌تواند ترتیب نسبی آن‌ها را پیدا کند.
- **مرتب‌سازی غیر مقایسه‌ای**[۸]، که بدون مقایسه‌ی کلیدهای عناصر با هم، می‌توان مرتب‌سازی را انجام داد. این الگوریتم‌ها با استفاده از اطلاعاتی که از قبل در خصوص نوع کلیدها موجود است، و در شرایط خاص، از روش‌های سریعی استفاده می‌کنند که در آن‌ها کلیدهای عناصر با هم مقایسه نمی‌شوند.

بدیهی است که کران پایین تعداد مقایسه‌ی هر الگوریتم مرتب‌سازی، از جمله الگوریتم‌هایی که مقایسه‌ای نیستند $\Omega(n)$ است. نشان خواهیم داد که برای الگوریتم‌های مقایسه‌ای، کران پایین ــ هم در بدترین حالت و هم درحالت میانگین ــ $\Omega(n \lg n)$ است.

ابتدا مرتب‌سازی غیر مقایسه‌ای را بررسی می‌کنیم که از آن‌ها، مرتب‌سازی شمارشی[۹]، مرتب‌سازی مبنایی[۱۰] و مرتب‌سازی سطلی[۱۱] را می‌توان نام برد. اما قبل از آن، به یک تعریف توجه کنید.

تعریف ۶-۱ ترتیب الفبایی[۱۲]، اگر $\vec{b}$ و $\vec{a}$ دو رشته‌ی به‌ترتیب n و m نویسه‌ای زیر باشند و داشته باشیم $n \leq m$،

$$\vec{a} = a_1 a_2 \dots a_i \dots a_n$$
$$\vec{b} = b_1 b_2 \dots b_i \dots b_m,$$

تعریف می‌کنیم $\vec{a} < \vec{b}$ اگر یک k $(1 \leq k \leq n)$ وجود داشته باشد که $a_k < b_k$ و $a_i = b_i$ برای تمام $1 \leq i < k$.

---

[۶] unstable
[۷] comparison sort
[۸] non-comparison sort
[۹] count sort
[۱۰] radix sort
[۱۱] bucket sort
[۱۲] lexicographic ordering

در مواری که کلیدهای عناصر رشته باشند، مرتب‌سازی آن‌ها بر اساس ترتیب الفبایی و مطابق تعریف گفته‌شده است.

در بخش بعد، کران پایین تعداد مقایسه‌ها را در مرتب‌سازی مقایسه‌ای، در بدترین حالت و حالت میانگین ثابت می‌کنیم.

درخت تصمیم

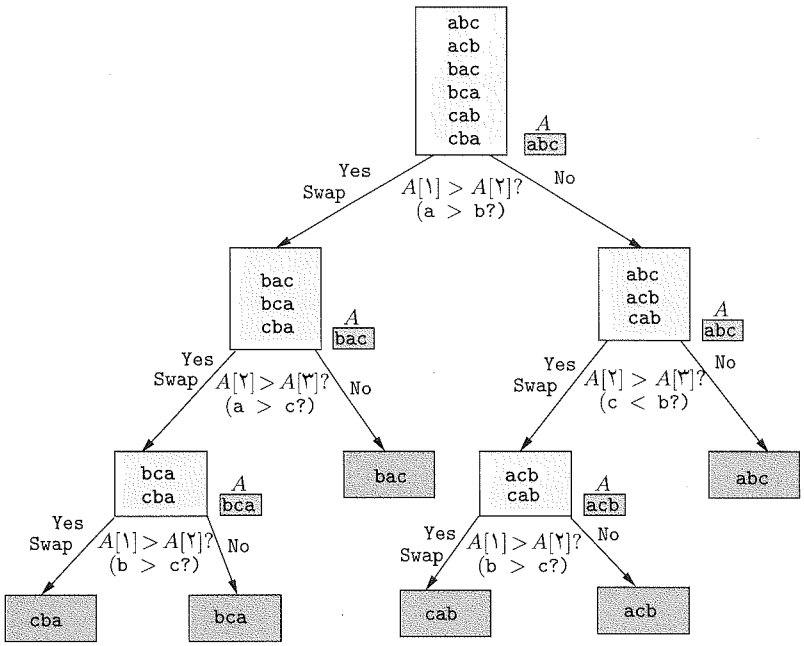
یک الگوریتم مرتب‌سازی $\mathcal{A}$ را در نظر بگیرید که با مقایسه و تعویض عناصر، آرایه‌ی ورودی را مرتب می‌کند. رفتار $\mathcal{A}$ بر روی تعدادی عنصر ورودی را می‌توان با یک «درخت تصمیم[۱۳]» $T_{\mathcal{A}}$ مدل کرد. در این مدل هر «وضعیت[۱۴]» الگوریتم با یک گره نمایش داده می‌شود. طبق تعریف، یک وضعیت s_i حاوی زیرمجموعه‌ی P_i از $n!$ جای‌گشت ورودی است، به‌طوری‌که اگر $\mathcal{A}$ در وضعیت s_i باشد، خروجی نهایی الگوریتم، یکی از جای‌گشت‌های موجود در P_i است. ریشه‌ی درخت تصمیم، یا وضعیت اولیه‌ی الگوریتم، شامل همه‌ی $n!$ جای‌گشت است.

با هر مقایسه، $\mathcal{A}$ دو عنصر از ورودی را مقایسه می‌کند (مثلاً $\alpha > \beta$) و در صورت «درست» (Y) یا «نادرست» (N) بودن این مقایسه، s_i به‌ترتیب به یکی از دو وضعیت s_j یا s_k منتقل می‌شود، به‌طوری‌که $P_j \subset P_i$ شامل همه‌ی جای‌گشت‌های P_i است که در آن‌ها $\alpha > \beta$ و نیز P_k شامل همه‌ی جای‌گشت‌های P_i است که در آن‌ها $\alpha \geq \beta$.

به‌عنوان مثال، اگر $\mathcal{A}$ مرتب‌سازی درجی زیر باشد (این همان الگوریتم بیان‌شده در صفحه‌ی ۵۷ در بخش ۳-۱-۱ با کمی تغییر است.)

```
INSERTION-SORT' (A, n)
    for k ← 2 to n
1     do i ← k
2        while i > 1 and A[i − 1] > A[i]
3           do SWAP (A[i], A[i − 1])
4              i ← i − 1
```

[۱۳] decision tree
[۱۴] state



شکل ۶-۱ درخت تصمیم برای الگوریتم 'INSERTION-SORT با ورودی $A[۱ \cdots ۳] = \text{abc}$. هر گره این درخت (یا وضعیت) شامل تعدادی جای‌گشت است که یکی از آن‌ها جای‌گشت مرتب خروجی است. در کنار هر وضعیت، مقدار آرایه‌ی A نوشته شده است.

درخت تصمیم برای آرایه‌ی ورودی $A[۱ \cdots ۳] = \text{abc}$ مطابق شکل ۶-۱ در خواهد آمد که در آن جای‌گشت‌های هر وضعیت درون مستطیل آن نوشته شده است. در کنار هر مستطیل هم عناصر آرایه‌ی A مشخص است. یک برگ در این درخت وضعیت نهایی یک ورودی خاص را نشان می‌دهد. برای هر مقدار ورودی، رفتار الگوریتم مسیری است که از ریشه تا یک برگ ادامه دارد و در هر وضعیت این مسیر، دو عنصر این آرایه را با هم مقایسه می‌کند و سپس ممکن است این دو عنصر را با هم تعویض کند.

ارتباط بین وضعیت‌های فوق، یک گراف وضعیت را ایجاد می‌کند. روشن است که این گراف نمی‌تواند دور داشته باشد. همچنین، هر وضعیت که شامل بیش از یک جای‌گشت باشد، دقیقاً دو وضعیت خروجی دارد، و نیز حالتی که تنها یک جای‌گشت داشته باشد، خروجی ندارد و برگ است. پس این گراف وضعیت یک درخت دودویی پُر و با دست‌کم $n!$ برگ است (چون ممکن است برخی از برگ‌ها تکرار شوند).

قضیه‌ی ۶-۱ یک درخت دودویی به ارتفاع h حداکثر ۲^h برگ دارد.

این قضیه با استقرا روی h ثابت می‌شود (امتحان کنید).

قضیه‌ی ۶-۲ ارتفاع یک درخت تصمیم که n عنصر را مرتب می‌کند دست‌کم $\lceil \lg n! \rceil$ است.

اثبات: درخت تصمیم دست‌کم $n!$ برگ دارد، بنابراین ارتفاعش دست‌کم $\lceil \lg n! \rceil$ است. □

نتیجه‌ی ۶-۵ هر الگوریتم مرتب‌سازی مقایسه‌ای که n عنصر را مرتب می‌کند، در بدترین حالت، باید دست‌کم $\lceil \lg n! \rceil$ مقایسه بین عناصر ورودی انجام دهد.

می‌دانیم که $n! \leq n^n$، پس $\lceil \lg n! \rceil \leq n \lg n$. ولی این کران بالای ضعیفی است. چنان‌چه در بخش ۲ ذکر شد، تقریب استرلینگ کران بهتری را به‌دست می‌دهد. براساس این تقریب داریم

$$n! = \sqrt{۲\pi n} \left(\frac{n}{e}\right)^n \left(۱ + \Theta\left(\frac{۱}{n}\right)\right) \qquad (۶\text{-}۱)$$

با استفاده از این رابطه می‌توان ثابت کرد که

$$n! = o(n^n) \qquad (۶\text{-}۲)$$

$$n! = \omega(۲^n) \qquad (۶\text{-}۳)$$

$$\lg(n!) = \Theta(n \lg n) \qquad (۶\text{-}۴)$$

نتیجه‌ی ۶-۵، کران پایین تعداد مقایسه‌ها (و در نتیجه زمان اجرای) هر الگوریتم مرتب‌سازی را در بدترین حالت بیان می‌کند.

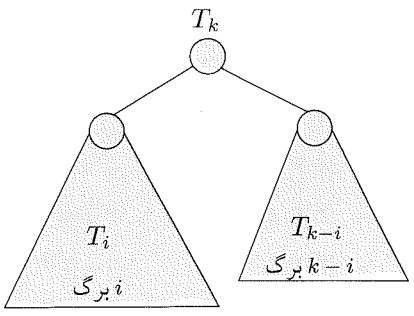
حال می‌خواهیم کران پایین تعداد مقایسه‌های هر الگوریتم مرتب‌سازی مقایسه‌ای را در حالت میانگین نیز به‌دست آوریم. توجه داریم که هر الگوریتم مرتب‌سازی را می‌توان با یک درخت تصمیم مدل کرد. ارتفاع این درخت بیان‌گر رفتار آن الگوریتم در بدترین حالت است؛ یعنی یک ورودی برای آن الگوریتم وجود دارد که موجب پرهزینه‌ترین رفتار از سوی الگوریتم می‌شود. اما منظور از رفتار میانگین چیست؟

فرض می‌کنیم که با احتمال مساوی، ورودی یکی از $n!$ جای‌گشت ممکن است. در این صورت، مطابق آن‌چه در مورد درخت تصمیم بیان شد، الگوریتم با احتمال یک‌سان به یکی از برگ‌های درخت تصمیم خواهد رفت و تعداد مقایسه‌های لازم به‌ازای هر برگ، عمق آن برگ در درخت تصمیم است. بنابراین میانگین هزینه‌ی الگوریتم برابر است با میانگین عمق برگ‌های درخت تصمیم برای آن الگوریتم. حال باید ثابت کنیم که این مقدار برای هر الگوریتم مرتب‌سازی مقایسه‌ای نمی‌تواند از یک کران کم‌تر باشد.

قضیه‌ی ۶-۳ اگر هر یک از جای‌گشت‌های یک ترتیب n تایی، با احتمال یک‌سان در ورودی ظاهر شود، آن‌گاه میانگین عمق برگ‌های درخت تصمیم برای هر الگوریتم مرتب‌سازی مقایسه‌ای، دست‌کم $\lg n!$ خواهد بود.

اثبات: فرض کنید T_m درخت تصمیمی باشد که m برگ دارد. همچنین برای کلیه‌ی درخت‌های دودویی T_m که m عدد برگ دارند، فرض کنید $D(T_m)$ مجموع عمق برگ‌های T_m، و $D(m)$ مقدار کمینه‌ی $D(T_m)$ است. با استقرا ثابت می‌کنیم که $D(m) \geq m \lg m$.

پایه‌ی استقرا: حکم برای $m = ۱$ واضح است.
فرض استقرا: برای $m < k$ درست است.
حکم استقرا: درخت T را با k برگ در نظر بگیرید (شکل ۶-۲). فرض کنید که زیردرخت چپ ریشه‌ی T_k i برگ داشته باشد. چون درخت تصمیم دودویی کامل است، $i \geq ۱$ و همچنین زیردرخت راست T_k دارای $k - i$ برگ است. روشن است که عمق هر برگ T_i



شکل ۶-۲ اثبات قضیه‌ی ۶-۳.

یک واحد کم‌تر از عمق آن برگ در T_k است. بنابراین،

$$D(T) = i + D(T_i) + (k-i) + D(T_{k-i}).$$

اما با توجه به تعریف داریم:

$$D(k) = \min_{1 \le i \le k} \{k + D(i) + D(k-i)\}.$$

و با فرض استقرا

$$D(k) \ge k + \min_{1 \le i \le k} \{i \lg i + (k-i) \lg (k-i)\}.$$

با فرض آن‌که i یک عدد طبیعی و k ثابت است، می‌دانیم که تابع $i \lg i$ برحسب i اکیداً صعودی و تابع $(k-i)\lg(k-i)$ اکیداً نزولی است. مقدار کمینه‌ی جمع این دو تابع هنگامی است که این دو تابع با هم برابر شوند؛ یعنی برای $i = \frac{k}{2}$. بنابراین،

$$D(k) \ge k \lg k \text{ و } D(m) \ge m \lg m$$

و حکم استقرا ثابت می‌شود.

با توجه به آن‌که احتمال هر ورودی برابر $\frac{1}{n!}$ است، و هر درخت تصمیم $n!$ برگ دارد، میانگین عمق برگ‌ها برابر $\lg n!$ خواهد بود. □

تمرین‌های بخش ۶-۱

۶-۱.۱ اعداد $1, 2, 3, \ldots, n$ به‌ترتیب دل‌خواهی بر روی یک ردیف قرار گرفته‌اند ($n > 1$). می‌خواهیم با استفاده از تکرار عمل «وارون»، آن‌ها را به‌ترتیب صعودی از چپ به راست مرتب کنیم. با هر عمل «وارون i» می‌توانیم i رقم سمت چپ را وارون کنیم ($1 < i \le n$). به‌عنوان مثال، در ۶ مرحله می‌توانیم ترتیب $\langle ۴, ۱, ۳, ۵, ۶, ۲ \rangle$ را مرتب کنیم:

$$\begin{aligned}
&\langle ۴, ۱, ۳, ۵, ۶, ۲ \rangle && i = ۲ \\
\rightarrow\ &\langle ۱, ۴, ۳, ۵, ۶, ۲ \rangle && i = ۶ \\
\rightarrow\ &\langle ۲, ۶, ۵, ۳, ۴, ۱ \rangle && i = ۵ \\
\rightarrow\ &\langle ۴, ۳, ۵, ۶, ۲, ۱ \rangle && i = ۲ \\
\rightarrow\ &\langle ۳, ۴, ۵, ۶, ۲, ۱ \rangle && i = ۴ \\
\rightarrow\ &\langle ۶, ۵, ۴, ۳, ۲, ۱ \rangle && i = ۶ \\
\rightarrow\ &\langle ۱, ۲, ۳, ۴, ۵, ۶ \rangle
\end{aligned}$$

الف) ثابت کنید هر ترتیبی از اعداد ۱ تا n را می‌توان در حداکثر $2n - 3$ مرحله تکرار عمل وارون مرتب کرد.

ب) ثابت کنید که برای $n \geq 5$، ترتیبی وجود دارد که نمی‌توان آن را در کم‌تر از n مرحله انجام عمل وارون مرتب کرد.[15]

۶-۱.۲ روشی را نشان دهید که بدون افزایش درجه‌ی پیچیدگی بتوان هر مرتب‌سازی ناپایدار را به‌گونه‌ی پایدار آن تبدیل کرد.

۶-۱.۳ ثابت کنید که هر الگوریتم مقایسه‌ای که نزدیک‌ترین عنصر موجود در یک آرایه‌ی n تایی را به عدد ورودی x به‌دست آورد، از $\Omega(\lg n)$ است. $A[i]$ نزدیک‌ترین عنصر است اگر $|x - A[i]|$ کم‌ترین مقدار را داشته باشد.

۶-۱.۴ هنگامی که ممکن است کلیدهای عناصر مقدارهای یک‌سان داشته باشند، هر مقایسه بین کلیدهای K_1 و K_2، یکی از سه پاسخ زیر را دارد: $K_1 < K_2$، $K_1 > K_2$ و $K_1 = K_2$. بنابراین درخت تصمیم برای چنین حالتی یک درخت سه‌تایی است و هر عنصر آن سه فرزند دارد. یک درخت تصمیم سه‌تایی برای سه عنصر بکشید. این درخت باید ۱۳ برگ (حالت نهایی) داشته باشد.

۶-۱.۵ اگر بخواهیم n عنصر را با کلیدهای ۱، ۱- و ۰ (صفر) مرتب کنیم، کمینه‌ی تعداد مقایسه‌ها در بدترین حالت چندتاست؟

۶-۲ مرتب‌سازی خطی

چنان‌چه بیان شد، اگر کلیدهای عناصر شرایط خاصی داشته باشند، می‌توان آن‌ها را در بهترین زمان ممکن، یعنی به‌صورت خطی مرتب کرد. مثلاً یک فرهنگ با n واژه را (به‌شرط محدود بودن طول واژه‌ها) می‌توان در $\mathcal{O}(n)$ مرتب کرد که این برای فرهنگ‌های بزرگ بسیار سریع‌تر از الگوریتم‌های مقایسه‌ای است. در این بخش سه الگوریتم مرتب‌سازی غیرمقایسه‌ای به نام‌های شمارشی، مبنایی و سطلی را ارائه می‌کنیم.

۶-۲-۱ مرتب‌سازی شمارشی

اگر کلید هر کدام از n داده‌ی ورودی یک عدد صحیح بین ۱ تا m (یک عدد ثابت) باشد، می‌توان با استفاده از مرتب‌سازی شمارشی بدون مقایسه و فقط با جابه‌جا کردن عناصر، آن‌ها را مرتب کرد. رویه‌ی COUNT-SORT این‌کار را انجام می‌دهد. ورودی الگوریتم

[15] سؤال ۲، مرحله‌ی اول پنجمین المپیاد کامپیوتر ایران

آرایه‌ی A حاوی اعداد ۱ تا m است، که مرتب‌شده‌ی آن در آرایه‌ی B نوشته خواهد شد (پس الگوریتم درجا نیست).

```
COUNT-SORT (A, B, m)
1  for i ← 1 to m
2     do C[i] ← 0
3  for i ← 1 to length[A]
4     do C[A[i]] ← C[A[i]] + 1
5  for i ← 2 to m
6     do C[i] ← C[i] + C[i − 1]
7  for i ← length[A] downto 1
8     do B[C[A[i]]] ← A[i]
9        C[A[i]] ← C[A[i]] − 1
```

این الگوریتم از یک آرایه‌ی کمکی دیگر به‌نام C استفاده می‌کند. ابتدا و در سطرهای ۳ و ۴، به‌ازای هر کلید r، تعداد عناصر A را که کلیدشان برابر r است می‌شمارد و در $C[r]$ می‌نویسد. سپس در سطرهای ۵ و ۶، به‌ازای هر کلید r بزرگ‌ترین اندیس در آرایه‌ی خروجی B را پیدا می‌کند که کلید r باید در آن‌جا نوشته شود، و این مقدار را در $C[r]$ می‌نویسد. در سطرهای ۷ تا ۹، با استفاده از مقادیر جدید آرایه‌ی C، درایه‌های A را از اندیس بزرگ به کوچک بررسی می‌کند و $k = A[i]$ را در $B[C[A[i]]]$ می‌نویسد و یک واحد از آخرین اندیس B که کلید k باید در آن قرار گیرد کم می‌کند تا اطلاعات C درست بماند. دقت کنید که الگوریتم طوری طراحی شده تا پایدار باشد.

روشن است که زمان اجرای کل الگوریتم $\mathcal{O}(n+m)$ است. اگر m از $\mathcal{O}(n)$ باشد، زمان اجرای کل $\mathcal{O}(n)$ خواهد بود. همچنین این الگوریتم به $\mathcal{O}(n+m)$ حافظه‌ی اضافی نیاز دارد.

حالت خاص

اگر کلیدهای عناصر اعداد ۱ تا n باشند و از هر کلید دقیقاً یک عدد داشته باشیم، می‌توان با الگوریتم زیر آن‌ها را به‌صورت درجا (بدون استفاده از آرایه‌ی کمکی) مرتب کرد.

```
COUNT-SORT' (A, n)
    ▷ حالت خاص: اگر کلیدهای n عنصر همه‌ی اعداد ۱ تا n باشند ◁
1   for i ← 1 to n
2     do while key[A[i]] ≠ i
3           do SWAP(A[i], A[key[A[i]]])
```

روشن است که این الگوریتم با هر بار تعویض، دست‌کم یک عنصر را در جای نهایی خودش قرار می‌دهد و آن عنصر دیگر جابه‌جا نمی‌شود. بنابراین پس از n بار تکرار، همه‌ی عناصر مرتب می‌شوند. دقت کنید که اگر از یک کلید بیش از یک بار داشته باشیم الگوریتم فوق ممکن است در حلقه بیفتد.

۶-۲-۲ مرتب‌سازی مبنایی

الگوریتم مرتب‌سازی مبنایی یک الگوریتم قدیمی است که در ماشین‌های مکانیکی برای مرتب‌سازی کارت‌ها از آن استفاده می‌شد. در این الگوریتم (اگر برای مرتب‌سازی اعداد به کار گرفته شود) ابتدا ورودی‌ها را بر اساس کم‌ارزش‌ترین رقم عدد ورودی (رقم شماره‌ی ۱) مرتب می‌کنیم، سپس بر اساس کم‌ارزش‌ترین رقم دوم (رقم شماره‌ی ۲)، و ... و در انتها بر اساس پرارزش‌ترین رقم مرتب می‌کنیم. برای این‌کار، باید از یک الگوریتم مرتب‌سازی پایدار مانند مرتب‌سازی شمارشی استفاده کنیم.

فرض کنید d تعداد رقم‌های ورودی است. اگر تعداد ارقام برای همه‌ی عناصر ورودی یک‌سان نباشد، با گذاشتن صفر در سمت چپ، آن‌ها را برابر می‌کنیم. رویه‌ی RADIX-SORT این کار را انجام می‌دهد.

```
RADIX-SORT (A, d)
1  for i ← 1 to d
2    do sort array A on digit i using a stable sort algorithm
```

درستی این الگوریتم را می‌توان با استقرا بر روی i ثابت کرد. ادعا می‌کنیم که در انتهای مرحله‌ی iام، عناصر ورودی برحسب i رقم اولشان (از کم‌ارزش به پرارزش) مرتب می‌شوند. پایه‌ی استقرا برای $i = ۰$ روشن است. اگر این امر برای $i = k$ درست باشد، با توجه به این‌که از یک الگوریتم پایدار استفاده می‌کنیم، در انتهای مرحله‌ی $k + ۱$ام نیز ادعا درست است.

زمان اجرای مرتب‌سازی مبنایی به الگوریتم ثانویه‌ی به‌کار رفته نیز بستگی دارد. اگر از مرتب‌سازی شمارشی استفاده کنیم زمان اجرا $\mathcal{O}(dn)$ خواهد بود.

مرتب‌سازی مبنایی را می‌توان به‌حالتی تعمیم داد که ورودی، آرایه‌ای از رکوردها باشد که در آن هر رکورد کلیدی با k مؤلفه داشته باشد و هر مؤلفه‌ی آن از یک داده‌گونه با حالت‌های قابل شمارش باشد. به این تعمیم، مرتب‌سازی سطلی گفته می‌شود.

۶-۲-۳ مرتب‌سازی سطلی

مرتب‌سازی سطلی حالت کلی مرتب‌سازی مبنایی است. فرض می‌کنیم ورودی دنباله یا لیست A با n رکورد است که هر رکورد آن کلیدی با k مؤلفه به‌نام‌های $f_۱$، $f_۲$ تا f_k دارد. فرض کنید که (برای $۱ \le i \le k$) f_i از داده‌گونه‌های t_i است و نیز این‌که تعداد مقادیر مختلفی که t_i می‌تواند داشته باشد، و ما آن‌را با s_i نشان می‌دهیم، مقداری ثابت و مستقل از n است. همچنین فرض می‌کنیم که f_i بر $f_{i-۱}$ اولویت دارد.

به‌عنوان مثال، A می‌تواند یک دسته از کارت‌های سوراخ‌شده[۱۶] باشد به‌طوری‌که مؤلفه‌های هر کارت در ستون‌های مختلف آن سوراخ شده است. برای توضیح ساده‌ترِ الگوریتم، فرض می‌کنیم ورودی یک دسته کارت است.

همچنین برای هر یک از k داده‌گونه‌ی t_i (برای $۱ \le i \le k$) مجموعه‌ای از سطل‌ها به نام B_i تهیه کرده‌ایم. B_i در واقع آرایه‌ای است که اندیس‌های آن مقادیر مختلف t_i است و هر درایه‌ی آن سطلی (لیستی) است که عناصر لیست اصلی (یا همان کارت‌ها) را در خود جای می‌دهد. در واقع، $B_i[v]$ سطلی است که کارت‌هایی که مقدار مؤلفه‌ی f_i آن‌ها برابر $v \in t_i$ است در آن قرار می‌گیرند. به زبان شبه‌پاسکال، B_i به‌صورت زیر تعریف می‌شود:

$$B_i : \texttt{array}[t_i] \quad \texttt{of list-type;}$$

---

[۱۶] punched cards

رویه‌ی BUCKET-SORT (A) لیست نامرتب A با مشخصات بیان‌شده را با این الگوریتم مرتب می‌کند.

```
BUCKET-SORT (A)
    ▷ مرتب‌سازی سطلی که لیست A را مرتب می‌کند
 1  for i ← 1 to k
 2    do for each value v of type t_i
 3         do make B_i[v] empty
 4       for each record r in list A in order
 5         do let v be the value of f_i of the key for r
 6              move r from A to the end of B_i[v]
 7       for each value v of type t_i from lowest to highest
 8         do concatenate B_i[v] to the end of A
```

در این رویه، در ابتدا همه‌ی کارت‌ها به‌صورت نامرتب در یک دسته قرار دارند. الگوریتم، کارت‌ها را به‌ترتیب مورد بررسی قرار می‌دهد و در انتهای مرحله‌ی i ام (برای $1 \leq i \leq k$) آن‌ها را برحسب مقادیر مختلف مؤلفه‌ی f_i شان به‌صورت زیر مرتب می‌کند (این کار به‌دلیل آن است که مؤلفه‌ی i بر $i-1$ اولویت دارد).

در مرحله‌ی i ام الگوریتم کارت‌های ورودی را نگاه می‌کند و اگر مقدار مؤلفه‌ی i برای کارتی که نگاه می‌کند v باشد، آن کارت را در انتهای کارت‌های سطل $B_i[v]$ قرار می‌دهد. (یا کارت را برعکس در آن سطل می‌اندازد.) در انتهای این مرحله، دسته‌کارت‌های همه‌ی سطل‌های B_i را به‌ترتیب پشت سر هم قرار می‌دهد (همان عمل الحاق[۱۷]) تا دسته‌کارت جدید به‌دست آید و در مرحله‌ی بعد مورد استفاده قرار گیرد.

ادعا می‌کنیم که در انتهای مرحله‌ی i ام، دسته‌کارت‌ها به‌ترتیب، برحسب مؤلفه‌های f_1 تا f_i شان مرتب شده‌اند. این ادعا به‌سادگی و با استقرا قابل اثبات است. اثبات این نکته با اهمیت است که کارت‌ها برعکس در سطل‌ها قرار می‌گیرند و الحاق این کارت‌ها باعث می‌شود که ترتیب دسته‌کارت‌ها و نیز الگوریتم پایدار بماند.

---

[۱۷] concatenation

اگر هر سطل را به‌صورت یک صف پیاده‌سازی کنیم، هر عمل الحاق در زمان ثابت قابل اجراست. در این صورت، زمان اجرای این الگوریتم

$$\sum_{i=1}^{k} \mathcal{O}(s_i + n) = \mathcal{O}(kn + \sum_{i=1}^{k} s_i)$$

است. اگر فرض کنیم $k, s_i = \mathcal{O}(n)$، آن‌گاه

$$T(n) = \Theta(kn + \sum_{i=1}^{k} n) = \Theta(n + kn) = \Theta(n).$$

مثال ۶-۱ فرض کنید هر کلید شامل سه مؤلفه با مقادیر {a...z}، {۱۰۰...۱۱۰۰} و {۱۳۰۰...۱۴۰۰} هستند و ۷ رکورد زیر داده شده‌اند.

عنصر	f_3	f_2	f_1
a_1	a	۵	۱۳۲۰
a_2	c	۱۲	۱۳۱۰
a_3	b	۱۲	۱۳۰۵
a_4	a	۸	۱۴۰۰
a_5	z	۱۰	۱۳۰۸
a_6	b	۱۲	۱۳۰۴
a_7	a	۶	۱۳۱۰

دسته‌کارت ورودی را با $\langle a_1, a_2, a_3, a_4, a_5, a_6, a_7 \rangle$ نشان می‌دهیم.

در اولین مرحله کارت‌ها به‌ترتیب بررسی می‌شوند و هر کارت بر اساس مقدار مؤلفه‌ی اولش در انتهای یکی از سطل‌های $B_1[.]$ قرار می‌گیرد، چنان‌چه در جدول ۶-۱ نشان داده شده است. پس از الحاق کارت‌های سطل‌ها، دسته‌کارت

$$\langle a_6, a_3, a_5, a_2, a_7, a_1, a_4 \rangle$$

به‌دست می‌آید. اگر این کارت‌ها را به‌ترتیب برحسب مقدار f_2 شان در سطل‌های $B_2[.]$ قرار دهیم و سپس آن‌ها الحاق کنیم، دسته‌کارت زیر به‌دست می‌آید:

$$\langle a_1, a_7, a_4, a_5, a_6, a_3, a_2 \rangle$$

پس از تکرار این کار بر روی مؤلفه‌ی f_3، کارت‌ها با اولویت‌های مورد نظر، برحسب کلیدهایشان مرتب می‌شوند.

جدول ۶-۱ مراحل مختلف اجرای الگوریتم مرتب‌سازی سطلی.

ورودی	$B_1[.]$	خروجی ۱	$B_2[.]$	خروجی ۲	$B_3[.]$	خروجی ۳
a_1	[۱۳۰۴] a_6	a_6	[۵] a_1	a_1	['a'] a_1, a_7, a_4	a_1
a_2	[۱۳۰۵] a_3	a_3	[۶] a_7	a_7	['b'] a_6, a_3	a_7
a_3	[۱۳۰۸] a_5	a_5	[۸] a_4	a_4	['c'] a_2	a_4
a_4	[۱۳۱۰] a_2, a_7	a_2	[۱۰] a_5	a_5	['z'] a_5	a_6
a_5	[۱۳۲۰] a_1	a_7	[۱۲]a_6, a_3, a_2	a_6		a_3
a_6	[۱۴۰۰] a_4	a_1		a_3		a_2
a_7		a_4		a_2		a_5

تمرین‌های بخش ۶-۲

۶-۲.۱ فرض کنید که A آرایه‌ای نامرتب از n عنصر است به‌طوری‌که کلید آن‌ها فقط k عدد متمایز دارند (فرض کنید $k < \sqrt{n}$). می‌خواهیم A را با الگوریتمی پایدار از مرتبه‌ی $\mathcal{O}(n \lg k)$ مرتب کنیم. برای این‌کار،

الف) آرایه‌ی مرتب B را از k مقدار متفاوت در A بسازید.

ب) آرایه‌ی A را به کمک B در زمان $\mathcal{O}(n)$ مرتب کنید.

الگوریتم‌های هر دو مرحله‌ی بالا را تشریح کنید. داده‌ساختارهای مورد نیاز را نام ببرید. الگوریتم بند (ب) را به‌صورت شبه‌کد بنویسید و آن‌را تحلیل کنید.

۶-۲.۲ دنباله‌ای را در نظر بگیرید که در آن هر عنصر با اندیس زوج از هر دو عنصر مجاورش کوچک‌تر باشد. آیا می‌توان این‌چنین دنباله‌ای را در $\mathcal{O}(n)$ مرتب کرد؟

۶-۲.۳ n عدد صحیح با مقادیر بین ۰ تا k داده شده‌اند. با انجام پیش‌پردازشی که بیش‌تر از $\mathcal{O}(n+k)$ نباشد، کاری کنید که بتوان پرس‌وجوی زیر را در زمان $\mathcal{O}(1)$ انجام داد:

به‌ازای مقادیر صحیح a و b $(0 \le a, b \le k)$، تعداد عناصری که دارای مقادیر در بازه‌ی $[a, b]$ هستند پیداکنید.

⋆ **۶-۲.۴** برای $n = 2^k - 1$، فرض کنید آرایه‌ی $A[1 \cdots n]$ شامل همه‌ی اعداد k بیتی بجز یک عدد ناشناخته است. می‌خواهیم این عدد را پیدا کنیم. تنها عمل مجاز بر روی A، دست‌رسی به jامین بیت درایه‌ی $A[i]$ است که در زمان ثابت انجام می‌شود. نشان دهید که با $\Theta(n)$ بار انجام این عمل مجاز، می‌توان عدد ناموجود در آرایه را پیدا کرد.

۶-۲.۵ n آرایه هر کدام با سه عنصر و هر عنصر بین ۱ تا ۲۰ داده شده‌اند. الگوریتمی با $\mathcal{O}(n)$ ارائه دهید تا تشخیص دهد که آیا در میان این آرایه‌ها دو آرایه‌ی دقیقاً یکسان هستند یا خیر، و اگر جواب

مثبت است دو آرایه را به‌دست آورد.

۶-۲.۶ الگوریتمی بیابید تا با داشتن n عدد صحیح در بازه‌ی ۰ تا k و با پیش‌پردازش ورودی، به این پرسش‌ها در زمان $O(1)$ پاسخ دهد: «چه تعداد از این n عدد صحیح در بازه $[a, b]$ قرار می‌گیرند؟» مرتبه زمانی پیش‌پردازش در الگوریتم شما باید $\mathcal{O}(n+k)$ باشد.

۶-۲.۷ نشان دهید چگونه می‌توان n عدد صحیح در بازه‌ی ۰ تا $n^2 - 1$ را در زمان $\mathcal{O}(n)$ مرتب کرد.

۶-۲.۸ در اولین الگوریتم مرتب‌سازی سطلی که در این بخش شرح داده شد، در بدترین حالت دقیقاً چند مرحله برای مرتب‌سازی اعداد دهدهی d رقمی لازم است؟ در بدترین حالت فرد مرتب‌کننده چند دسته کارت را باید دنباله‌گیری کند؟

۶-۲.۹ مرتب‌سازی درجا در زمان خطی

آرایه‌ای از n رکورد فقط با کلیدهای ۰ یا ۱ داده شده است. الگوریتمی که چنین آرایه‌ای را مرتب می‌کند ممکن است زیرمجموعه‌ای از ویژگی‌های مطلوب زیر را دارا باشد:

۱. زمان اجرای الگوریتم $\mathcal{O}(n)$ باشد،
۲. مرتب‌سازی پایدار باشد و
۳. مرتب‌سازی درجا باشد و به حافظه‌ی اضافی بیش از مقداری ثابت احتیاج نداشته باشد.

الف) الگوریتمی بیابید که ویژگی‌های ۱ و ۲ را داشته باشد.

ب) الگوریتمی بیابید که ویژگی‌های ۱ و ۳ را داشته باشد.

پ) الگوریتمی بیابید که ویژگی‌های ۲ و ۳ را داشته باشد.

ت) آیا در میان الگوریتم‌هایی که در قسمت‌های (الف) تا (پ) ارائه کرده‌اید، الگوریتمی هست که n رکورد با کلیدهای b بیتی را با استفاده از مرتب‌سازی مبنایی در زمان $\mathcal{O}(bn)$ مرتب نماید؟ آن‌را توضیح دهید و اگر وجود ندارد دلیل آن‌را بیان کنید.

ث) فرض کنید این n رکورد، کلیدهایی در بازه‌ی ۱ تا k داشته باشند. نشان دهید چگونه الگوریتم مرتب‌سازی شمارشی را می‌توان تغییر داد تا بتوانیم رکوردها را به‌طور درجا در زمان $\mathcal{O}(n+k)$ مرتب کنیم. شما ممکن است به $\mathcal{O}(k)$ حافظه علاوه بر آرایه ورودی نیاز داشته‌باشید. آیا الگوریتم شما پایدار است؟ (راهنمایی: برای $k = 3$ چگونه عمل می‌کنید؟)

۶-۲.۱۰ نشان دهید که در شرایط زیر مرتب‌سازی را می‌توان در زمان خطی انجام داد:

الف) آرایه‌ای از اعداد صحیح داده شده است که تعداد ارقام اعداد مختلف در آن ممکن است متفاوت باشد، اما تعداد کل ارقام تمام اعداد در آن n است. نشان دهید که می‌توان این آرایه را در زمان $\mathcal{O}(n)$ مرتب کرد.

ب) آرایه‌ای از رشته‌ها داده شده است که هر یک ممکن است تعداد نویسه‌های مختلفی داشته باشد، اما تعداد کل نویسه‌های تمام رشته‌ها در آن n است. نشان دهید که می‌توان این آرایه را

در زمان $\mathcal{O}(n)$ مرتب کرد. (در این جا ترتیب مورد نظر ترتیب الفبایی است؛ مثلاً داریم: $a < ab < b$.)

⋆ ۶-۲.۱۱ لیوان‌های آب

فرض کنید n لیوان به‌رنگ آبی و n لیوان به‌رنگ قرمز داده شده‌اند که اندازه و شکل آن‌ها با یک‌دیگر متفاوت است. ظرفیت همه‌ی لیوان‌های قرمز با یکدیگر و نیز ظرفیت لیوان‌های آبی با هم متفاوت است. می‌دانیم که هر لیوان قرمز یک لیوان آبی متناظر دارد که ظرفیت‌های آن‌دو دقیقاً با هم مساوی‌اند. می‌خواهیم لیوان‌ها را به جفت‌های آبی‌ـقرمز دسته‌بندی کنیم طوری‌که لیوان‌های هر جفت ظرفیت یک‌سانی داشته باشند. برای این کار از روش زیر استفاده می‌کنیم:

یک لیوان قرمز و یک لیوان آبی را بر می‌داریم، لیوان قرمز را از آب پر می‌کنیم و آب آن را درون لیوان آبی می‌ریزیم. با این کار، مشخص می‌شود که آیا دو لیوان ظرفیت یک‌سانی دارند یا خیر، و در صورت تفاوت، می‌فهمیم که ظرفیت کدام بیش‌تر است. فرض کنید زمان انجام چنین مقایسه‌ای یک واحد زمانی است.

هدف این است که الگوریتمی بیابیم که این دسته‌بندی را با کم‌ترین تعداد مقایسه انجام دهد. توجه داشته باشید که مقایسه‌ی مستقیم دو لیوان هم‌رنگ امکان‌پذیر نیست.

الف) یک الگوریتم قطعی که $\Theta(n^2)$ مقایسه برای یافتن جفت لیوان‌ها نیاز دارد ارائه دهید.

ب) ثابت کنید که کران پایین تعداد مقایسه‌ها در هر الگوریتم برای حل این مسئله برابر با $\Omega(n \log n)$ است.

پ) یک الگوریتم تصادفی پیش‌نهاد کنید که میانگین تعداد مقایسه‌های آن $\mathcal{O}(n \log n)$ باشد و درستی این کران را ثابت کنید. تعداد مقایسه‌های الگوریتم شما در بدترین حالت چقدر است؟

⋆ ۶-۲.۱۲

فرض کنید می‌خواهیم به‌جای این‌که آرایه‌ای را کاملاً مرتب کنیم، اعضای آن را به‌گونه‌ای جابه‌جا کنیم که به‌طور میانگین صعودی باشند. به بیان دقیق‌تر، آرایه‌ی n عضوی A را kـمرتب گوییم اگر برای هر $i = 1, 2, \ldots, n-k$ رابطه زیر برقرار باشد:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}$$

الف) ۱ـمرتب بودن یک آرایه به چه معناست؟

ب) یک جای‌گشت از اعداد ۱، ۲ تا ۱۰ پیدا کنید که ۲ـمرتب باشد ولی ۱ـمرتب نباشد.

پ) ثابت کنید یک آرایه‌ی n عضوی kـمرتب است اگر و فقط اگر برای هر i، $i = 1, 2, \ldots, n-k$ داشته باشیم: $A[i] \leq A[i+k]$.

ت) الگوریتمی ارائه کنید که یک آرایه‌ی n عنصری را در زمان $\mathcal{O}(n \lg(\frac{n}{k}))$، kـمرتب کند.

همچنین می‌توان برای ایجاد یک آرایه‌ی kـمرتب که در آن k عددی ثابت است، کران پایینی به‌دست آورد.

ث) نشان دهید که یک آرایه‌ی kـمرتب به‌طول n را می‌توان در زمان $\mathcal{O}(n \lg k)$ مرتب کرد.

ج) نشان دهید که اگر k ثابت باشد، برای kـمرتب کردن یک آرایه به‌طول n، $\Theta(n \lg n)$ زمان نیاز داریم. (راهنمایی: از پاسخ قسمت قبل و کران پایین مرتب‌سازی مقایسه‌ای استفاده کنید.)

*** ۶-۲-۱۳ کران پایین برای ادغام لیست‌های مرتب**

مسئله‌ی ادغام دو لیست مرتب در مسئله‌های گوناگونی بروز می‌کند؛ مثلاً به‌عنوان یک زیرروبه از مرتب‌سازی ادغامی است. در این مسئله، نشان می‌دهیم که ادغام دو لیست مرتبِ n عضوی، در بدترین حالت دست‌کم به $۲n - ۱$ مقایسه نیاز دارد. ابتدا با استفاده از درخت تصمیم، کران پایین $۲n - o(n)$ مقایسه را ثابت می‌کنیم:

الف) نشان دهید که اگر $۲n$ عدد داشته باشیم، به $\binom{۲n}{n}$ روشِ مختلف می‌توان آن‌ها را به دو لیست n عضوی تقسیم کرد.

ب) با استفاده از یک درخت تصمیم، نشان دهید هر الگوریتمی که دو لیست مرتب را به درستی ادغام کند، دست‌کم به $۲n - o(n)$ مقایسه نیاز خواهد داشت.

حال کران دقیق‌تر $۲n - ۱$ را ثابت می‌کنیم.

پ) نشان دهید که اگر دو عنصر موجود در دو لیست مختلف، در لیست مرتبِ نهایی متوالی باشند، حتماً با یکدیگر مقایسه می‌شوند.

ت) با استفاده از پاسخ خود به قسمت قبل، ثابت کنید که کران پایین تعداد مقایسه‌ها برای ادغام دو لیست n عضوی $۲n - ۱$ است.

۶-۳ مرتب‌سازی مقایسه‌ای

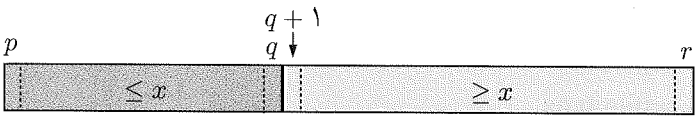
اگر تعداد حالات کلیدهای داده‌ی ورودی محدود نباشد، تنها با مقایسه‌ی کلیدهای عناصر ورودی می‌توان آن‌ها را مرتب کرد. چنان‌چه دیدیم کران پایین الگوریتم‌های مقایسه‌ای برای مرتب‌سازی n عنصر، هم در بدترین حالت و هم در حالت میانگین، برابر $\mathcal{O}(n \lg n)$ است. در این بخش، مرتب‌سازی سریع و مرتب‌سازی هرمی را توضیح می‌دهیم. البته داده‌ساختار مورد نیاز برای الگوریتم مرتب‌سازی هرمی قبلاً در بخش ۴-۶ از فصل ۴ بیان شد و در این بخش از آن استفاده می‌شود. برخی از مرتب‌سازی‌های مقایسه‌ای هم، مانند مرتب‌سازی درجی و حبابی و ادغامی، قبلاً در فصل ۳ ارائه شدند. مرتب‌سازی صدفی هم در همان فصل به‌عنوان یک تمرین بیان شد.

۶-۳-۱ مرتب‌سازی سریع

مرتب‌سازی سریع[۱۸] الگوریتمی است که در سال ۱۹۶۰ توسط سی.اِ. هُر[۱۹] طراحی شد. این الگوریتم براساس روش تقسیم‌وحل کار می‌کند و آرایه‌ی n عنصری را در بدترین حالت در زمان $O(n^۲)$ و در حالت میانگین در $O(n \lg n)$ مرتب می‌کند. البته ضریب ثابت این درجه‌ی پیچیدگی در حالت میانگین، نسبت به دیگر الگوریتم‌های بهینه، کوچک است و از این‌رو، در عمل سریع‌تر از الگوریتم‌های بهینه‌ی دیگر مرتب می‌کند. گونه‌ی حافظه‌ی خارجی این الگوریتم نیز کاراست و پیاده‌سازی آن به‌صورت موازی هم امکان‌پذیر است.

فرض کنید می‌خواهیم آرایه‌ی $A[p \cdots r]$ را مرتب کنیم. مانند دیگر الگوریتم‌های تقسیم‌وحل، مرتب‌سازی سریع شامل سه مرحله‌ی زیر است:

۱. **تقسیم:** آرایه‌ی $A[p \cdots r]$ به دو زیرآرایه‌ی ناتهی $A[p \cdots q]$ و $A[q+۱ \cdots r]$ بخش‌بندی[۲۰] می‌شود، به‌طوری‌که هر عنصر در $A[p \cdots q]$ از هیچ عنصری در $A[q+۱ \cdots r]$ بیش‌تر نباشد (شکل ۶-۳).



شکل ۶-۳ نمایی از الگوریتم مرتب‌سازی سریع.

۲. **حل:** هریک از زیرآرایه‌ی $A[p \cdots q]$ و $A[q+۱ \cdots r]$ به‌صورت بازگشتی مرتب می‌شوند.

۳. **ترکیب:** با توجه به نحوه‌ی بخش‌بندی، نیازی به ترکیب عناصر دو بخش نیست و آرایه در نهایت خودبه‌خود مرتب می‌شود.

رویه‌ی QUICKSORT(A, p, r) همین کار را انجام می‌دهد.

---

۱۸ quicksort
۱۹ C.A. Hoare
۲۰ partition

```
QUICKSORT (A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT (A, p, q)
4          QUICKSORT(A, q + 1, r)
```

این رویه در ابتدا به‌صورت QUICKSORT$(A, ۱, length[A])$ فراخوانی می‌شود.

رویه‌ی PARTITION که بخش‌بندی را انجام می‌دهد، مهم‌ترین قسمت این مرتب‌سازی است. این کار پیاده‌سازی‌های مختلفی دارد؛ یکی از آن‌ها چنین است:

```
PARTITION (A, p, r)
1  x ← A[p]
2  i ← p − 1
3  j ← r + 1
4  while true
5     do repeat j ← j − 1
6        until A[j] ≤ x
7        repeat i ← i + 1
8        until A[i] ≥ x
9        if i < j
10          then SWAP(A[i], A[j])
11          else return j
```

در این پیاده‌سازی، برای سرعت بخشیدن به الگوریتم و بدون انجام پردازش اضافی، ابتدا عنصرِ اول با کلیدی به‌نام x را به عنوان محور[۲۱] انتخاب می‌کنیم؛ در این جا محور $A[p]$ است، ولی می‌توان هرکدام از عناصر $A[p]$ تا $A[q - ۱]$ را نیز برای این کار انتخاب کرد.

درستی PARTITION را با استقرا ثابت می‌کنیم. در ابتدا فرض می‌کنیم که $A[p - ۱] = -\infty$ و $A[q + ۱] = \infty$. شرط مستقل از حلقه‌ی زیر برای این رویه را ثابت

[۲۱] pivot

می‌کنیم:

> در هر مرحله از حلقه‌ی **while** در ابتدا، مقدار هر عنصر با اندیس کم‌تر یا مساوی i، کم‌تر یا مساوی محور و مقدار هر عنصر با اندیس بیش‌تر یا مساوی j بزرگ‌تر یا مساوی محور است.

این شرط در ابتدای الگوریتم برقرار است زیرا داریم $i = p - 1$ و $j = r + 1$. در اولین گام حلقه و هنگام توقف، $i = p$ است و j در اندیسی متوقف می‌شود که $A[j] \leq x$ و حتماً اندیس j با خصوصیت $j \geq i$ وجود دارد. اگر $i < j$ باشد، اولین تعویض $A[i]$ و $A[j]$ انجام می‌شود. اما اگر $i = j$ باشد، رویه پایان می‌یابد و بخش چپ، تنها شامل عنصر $A[p]$ خواهد بود. در هر صورت نتیجه می‌گیریم که در انتها، دو بخش چپ و راست هر کدام دست‌کم یک عنصر خواهند داشت.

برای اثبات شرط مستقل از حلقه‌ی فوق در انتهای هر حلقه، فرض می‌کنیم که شرط در ابتدای آن حلقه برقرار است. توجه می‌کنیم که در هر گام، j به سمت چپ حرکت می‌کند و از روی عناصر $A[j] > x$ عبور می‌کند تا به یک عنصر $A[j] \leq x$ برسد. طبق فرض، چنین عنصری حتماً (دست‌کم در بخش غیر تهی در سمت چپ آرایه) وجود دارد. همچنین i به سمت راست حرکت می‌کند، از روی عناصر $A[i] < x$ عبور کرده و در اندیسی که برای آن $A[i] \geq x$ باشد متوقف می‌شود. مانند قبل، حتماً چنین عنصری وجود دارد. در پایان این گام از حلقه، سه حالت برای مقادیر اندیس i و j ممکن است:

الف) $i < j$

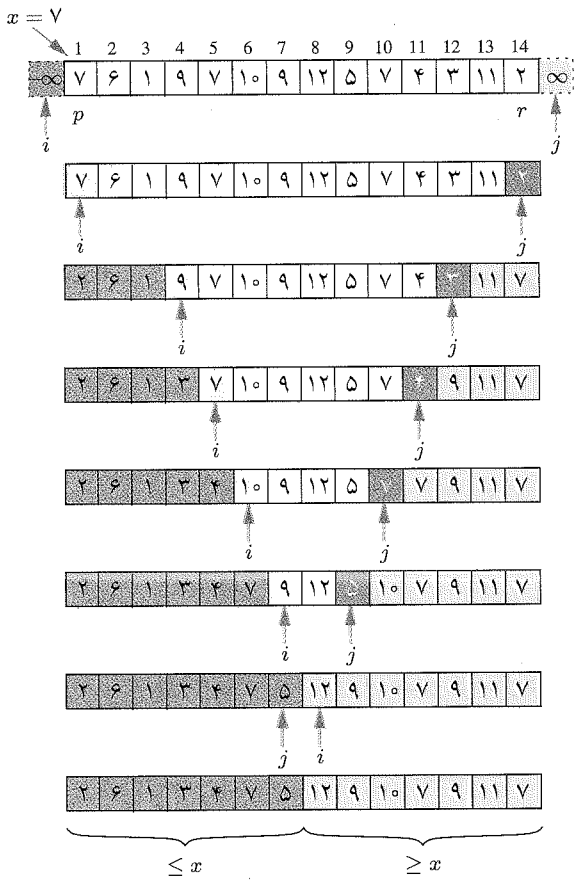
ب) $i = j$ (اگر در انتها $A[i] = A[j] = x$) و

پ) $j = i - 1$ (اگر در انتها $A[i] > x$ و $A[i - 1] < x$).

توجه کنید که $j < i - 1$ امکان‌پذیر نیست.

در حالت (الف)، دو عنصر $A[i]$ و $A[j]$ با هم تعویض می‌شوند. بدیهی است که در این صورت شرط مستقل از حلقه نیز برقرار است. در دو حالت (ب) و (پ)، ضمن برقراری شرط مستقل از حلقه، PARTITION پایان می‌یابد. در نهایت، آرایه به دو بخش ناتهی تقسیم می‌شود که یکی حاوی عناصر کم‌تر یا مساوی x و دومی حاوی عناصر بیش‌تر یا مساوی x است.

بنابراین مرتب‌سازی مستقل هر بخش موجب مرتب‌سازی کل آرایه می‌شود و رویه‌ی QUICKSORT به‌درستی ختم می‌شود. شکل ۶-۴ مراحل مختلف این رویه را برای یک مثال نشان می‌دهد.



شکل ۶-۴ مثالی از رویه‌ی PARTITION.

نکته‌های ۶-۱

۱. اگر یک آرایه‌ی n تایی با کلیدهای نامساوی در ابتدا به‌صورت صعودی مرتب باشد، پس از عمل بخش‌بندی، بدون تعویض به دو بخش به اندازه‌های ۱ و $n-1$ تقسیم می‌شود و این کار تکرار می‌گردد. پس، الگوریتم مرتب‌سازی سریع برای آرایه‌ی مرتب $O(n^2)$ زمان صرف می‌کند، ولی تعویضی انجام نمی‌دهد.

۲. اگر یک آرایه‌ی n تایی با کلیدهای نامساوی در ابتدا به‌صورت نزولی مرتب باشد، هر دو بار عمل بخش‌بندی یک عدد تعویض داریم ولی در هر بخش‌بندی با k عنصر، بخش‌ها به اندازه‌ی ۱ و $k-1$ تقسیم می‌شوند. پس زمان اجرای الگوریتم در این

حالت هم $\mathcal{O}(n^۲)$ است. تعداد تعویض‌ها در این صورت $\mathcal{O}(n)$ است. به‌طور دقیق‌تر، در این صورت بعد از اولین اجرای PARTITION آرایه مرتب می‌شود و دقیقاً تعداد تعویض‌ها برابر $\lfloor \frac{n}{۲} \rfloor$ خواهد بود.

۳. اگر همه‌ی عناصر آرایه دارای کلیدهای مساوی باشند، هر بار عمل بخش‌بندی با $\lfloor \frac{n}{۲} \rfloor$ بار تعویض آرایه را به دو بخش مساوی (حداکثر با اختلاف ۱) تقسیم می‌کند. پس زمان اجرای این الگوریتم در این حالت $\mathcal{O}(n \lg n)$ است، و تعداد تعویض‌ها نیز برابر $\mathcal{O}(n \lg n)$ خواهد بود.

تحلیل الگوریتم

ثابت می‌کنیم که هزینه‌ی الگوریتم مرتب‌سازی سریع در بدترین حالت برابر $\Theta(n^۲)$ و در حالت میانگین برابر $\Theta(n \lg n)$ است و این کارایی به نحوه‌ی بخش‌بندی وابسته است. می‌دانیم که هزینه‌ی هر عمل بخش‌بندی یک آرایه به‌اندازه‌ی n برابر $\mathcal{O}(n)$ با ثابت بسیار کوچک است.

بخش‌بندی در بدترین حالت

بدترین رفتار مرتب‌سازی سریع هنگامی اتفاق می‌افتد که بخش‌بندی در تمام مراحل الگوریتم r عنصر را به $r-۱$ و ۱ (یا $r-k$ و k برای هر k ثابت) عنصر تقسیم کند، و اگر آرایه از قبل مرتب باشد، این اتفاق می‌افتد. در این حالت، پیچیدگی زمانی الگوریتم به‌صورت $T(n) = T(n-۱) + \Theta(n)$ خواهد بود که با حل آن، به‌سادگی دیده می‌شود که $T(n) = \Theta(n^۲)$.

بخش‌بندی در بهترین حالت

بهترین حالت هنگامی اتفاق می‌افتد که رویه‌ی بخش‌بندی در هر مرحله r عنصر آرایه را به دو آرایه با تعداد عناصر $\lceil \frac{r}{۲} \rceil$ و $\lfloor \frac{r}{۲} \rfloor$ تقسیم کند. مثلاً اگر کلیدها همه برابر باشند، این حالت اتفاق می‌افتد. پیچیدگی الگوریتم در این حالت چنین محاسبه می‌شود

$$T(n) = ۲T(\frac{n}{۲}) + \Theta(n) \Longrightarrow T(n) = \Theta(n \lg n)$$

بخش‌بندی متوازن

در بخش‌های بعد، به‌طور کامل ثابت می‌کنیم که رفتار این الگوریتم در حالت میانگین خیلی بهتر از رفتار آن در بدترین حالت است. آن‌چه موجب این تفاوت در زمان اجرای الگوریتم می‌شود، نحوه‌ی بخش‌بندی بازگشتی آرایه و میزان توازن آن است. آن‌چه در این بخش می‌آوریم شهودی نسبت به تأثیر میزان توازن در بخش‌بندی با رفتار این الگوریتم است.

برای مثال، یک الگوریتم بخش‌بندی را در نظر بگیرید که همیشه آرایه را به نسبت ۹ به ۱ تقسیم می‌کند. در نگاه اول این الگوریتم نامتوازن به‌نظر می‌رسد. ولی چنین نیست! رابطه‌ی بازگشتی هزینه‌ی الگوریتم در این صورت $T(n) = T(\frac{9n}{10}) + T(\frac{n}{10}) + n$ و حل آن $T(n) = \mathcal{O}(n \lg n)$ است. درخت بازگشت این رابطه‌ی بازگشتی در شکل ۶-۵ دیده می‌شود. توجه کنید که حتی اگر این نسبت برای یک مقدار ثابت برای $\alpha < 1$ همیشه α به $1 - \alpha$ باشد، الگوریتم همچنان از مرتبه‌ی $\mathcal{O}(n \lg n)$ خواهد بود.

تحلیل در بدترین حالت

در بحث قبل مثالی از بدترین حالت مرتب‌سازی سریع دادیم که پیچیدگی الگوریتم برای آن $\Theta(n^2)$ است. البته این رفتار در حالت‌های کلی‌تر دیگر هم پیش می‌آید که در این بخش آن‌را بررسی می‌کنیم. نشان می‌دهیم که زمان اجرای مرتب‌سازی سریع در بدترین حالت $T(n) = \mathcal{O}(n^2)$ و نیز $T(n) = \Omega(n^2)$.

فرض کنید $T(n)$ زمان الگوریتم در بدترین حالت باشد و رویه‌ی PARTITION آرایه‌ی n عضوی را به زیرآرایه‌های q و $n - q$ عضوی تقسیم کند. در آن صورت

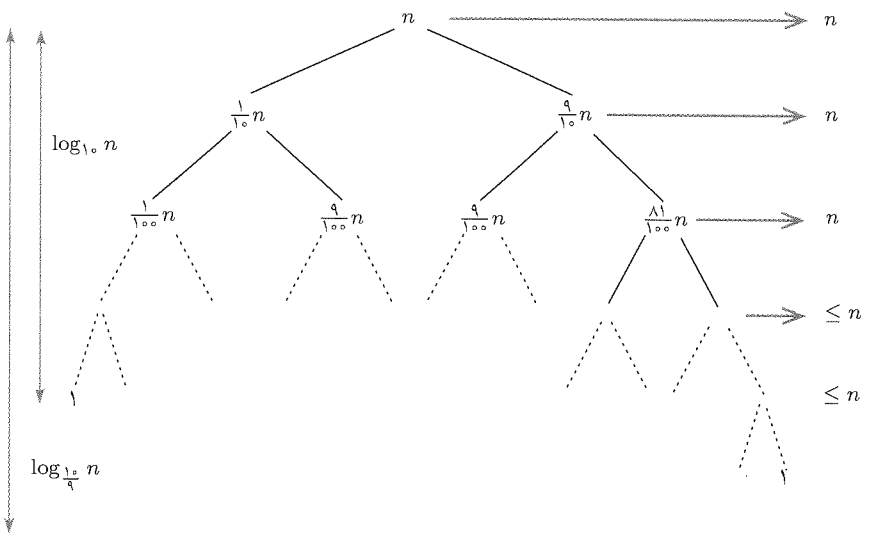
$$T(n) = \max_{1 \le q \le n-1} \{T(q) + T(n-q)\} + \Theta(n). \qquad (۶\text{-}۵)$$

توجه داریم که هر یک از دو ناحیه‌ی ایجاد شده پس از بخش‌بندی دست‌کم یک عضو دارد.

با استقرا اثبات می‌کنیم که $T(n) = \mathcal{O}(n^2)$. اگر برای $m < n$ فرض کنیم $T(m) \le cm^2$، داریم

$$T(n) \le \max\{cq^2 + c(n-q)^2\} + \Theta(n).$$

مقدار بیشینه‌ی $q^2 + (n-q)^2$ روی نواحی مرزی اتفاق می‌افتد. پس در حالت $q = 1$ داریم



شکل ۶-۵ درخت بازگشت برای $T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + n$

$$T(n) \leq c(1 + (n-1)^2) + \Theta(n) = cn^2 - 2c(n-1) + \Theta(n)$$
$$T(n) \leq cn^2,$$

به شرطی که c را آن‌قدر بزرگ بگیریم که $\Theta(n)$ را بپوشاند.

حال نشان می‌دهیم که $T(n) = \Omega(n^2)$. برای این کار، طبق استقرا فرض می‌کنیم برای $m \leq n$ داریم $T(m) \geq dm^2$. ثابت می‌کنیم که این رابطه برای $T(n)$ هم برقرار است. از این فرض و رابطه‌ی ۶-۵ نتیجه می‌گیریم

$$T(n) \geq \max_{1 \leq q \leq n-1} \{dq^2 + d(n-q)^2\} + \Theta(n).$$

بنابراین،

$$T(n) \geq dn^2 - 2d(n-1) + \Theta(n) \geq dn^2,$$

به‌شرطی که d آن قدر کوچک باشد که $\Theta(n) - 2d(n-1)$ مثبت شود. پس

$$T(n) = \Omega(n^2) = \mathcal{O}(n^2) \Rightarrow T(n) = \Theta(n^2),$$

و ادعا ثابت می‌شود.

۶-۳-۲ مرتب‌سازی سریع تصادفی

برای تحلیل الگوریتم مرتب‌سازی سریع در حالت میانگین، فرض می‌کنیم که هر کدام از جای‌گشت‌های عناصر ورودی با احتمال یک‌سان ظاهر می‌شود. اگر این فرض در مورد داده‌ی ورودی درست باشد، نشان می‌دهیم که از نظر میانگینِ زمان اجرا، مرتب‌سازی سریع یک الگوریتم بهینه است.

یک الگوریتم را «تصادفی[۲۲]» گوییم که رفتار آن علاوه بر آن‌که توسط ورودی مشخص می‌شود، به مقداری که مولد عدد تصادفی تولید می‌کند هم وابسته باشد. در تحلیل این الگوریتم فرض می‌کنیم یک تابع RANDOM(a, b) وجود دارد که به‌صورت تصادفی یک عدد بین a و b را تولید می‌کند.

در گونه‌ی تصادفی مرتب‌سازی سریع، قبل از صدا زدن رویه‌ی PARTITION، $A[p]$ با یک درایه‌ی تصادفی که بین p و r قرار دارد تعویض می‌شود.

```
RANDOMIZED-PARTITION (A, p, r)
1  i ← RANDOM(p, r)
2  SWAP(A[p], A[i])
3  return PARTITION(A, p, r)


RANDOMIZED-QUICKSORT (A, p, r)
1  if p < r
2     then q ← RANDOMIZED-PARTITION (A, p, r)
3          RANDOMIZED-QUICKSORT(A, p, q)
4          RANDOMIZED-QUICKSORT(A, q + 1, r)
```

تحلیل میانگین مرتب‌سازی سریع تصادفی

برای حالت میانگین، توجه کنید که محور همیشه عنصر اول آرایه‌ی A است که به‌طور تصادفی انتخاب می‌شود. بنابراین، احتمال این‌که مرتبه‌ی محور k $(1 \le k \le n)$ باشد (یعنی

[۲۲] randomized

محور k امین عنصر در حالت مرتب باشد) برابر $\frac{1}{n}$ است. فرض می‌کنیم که عناصر آرایه مقادیر متفاوت دارند. با این فرض هزینه‌ی الگوریتم ممکن است بالاتر رود و ما آن‌را در حالت سخت‌تری تحلیل می‌کنیم. توجه کنید که برای هر مقدار k، آرایه‌ی A به‌اندازه‌های مختلفی تقسیم می‌شود. طبق الگوریتم بخش‌بندی که گفتیم، در هر دو حالت $k = 1$ و $k = 2$، آرایه‌ی A به دو بخش با اندازه‌های ۱ و $n - 1$ تقسیم می‌شود. بنابراین اگر $\bar{T}(n)$ میانگین زمان اجرای الگوریتم RANDOMIZED-QUICKSORT باشد، داریم

$$\bar{T}(n) = \frac{1}{n}\left[\bar{T}(1) + \bar{T}(n-1) + \sum_{q=1}^{n-1}[\bar{T}(q) + \bar{T}(n-q)]\right] + \Theta(n). \qquad (۶\text{-}۶)$$

دو جمله‌ی اول برای $k = 1$ و هرکدام از $n - 1$ زوج‌جمله‌ی داخل جمع برای مقادیر $k = 2, \ldots, n$ است، چون با فرض نامساوی بودن عناصر، اگر محور k امین عنصر باشد $1 < k \leq n$، آرایه به دو بخش $k - 1$ و $n - k + 1$ عضوی تقسیم می‌شود.

در بحث‌های قبلی دیدیم که در بدترین حالت $T(n-1) = \Theta(n^2)$ است و $T(1) = \Theta(1)$. همچنین از آن‌جا که $T(n)$ بدترین زمان اجرای ممکن است، داریم $\bar{T}(n) \leq T(n)$. پس،

$$\frac{1}{n}[\bar{T}(1) + \bar{T}(n-1)] \leq \frac{1}{n}[\Theta(1) + \Theta(n^2)] = \Theta(n).$$

با توجه به فرمول ۶-۶ داریم:

$$\bar{T}(n) = \frac{1}{n}\sum_{q=1}^{n-1}[\bar{T}(q) + \bar{T}(n-q)] + \Theta(n) = \frac{2}{n}\sum_{k=1}^{n-1}\bar{T}(k) + \Theta(n)$$

برای حل این رابطه‌ی بازگشتی، حدس می‌زنیم برای مقادیری از $a, b > 0$، رابطه‌ی $\bar{T}(n) \leq an\lg n + b$ برقرار است، و آن‌را ثابت می‌کنیم.

$$\begin{aligned}
\bar{T}(n) &= \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \\
&\leq \frac{2}{n}\sum_{k=1}^{n-1} ak\lg k + b + \Theta(n) \\
&= \frac{2}{n}\sum_{k=1}^{n-1} ak\lg k + \frac{2}{n}\sum_{k=1}^{n-1} b + \Theta(n) \\
&= \frac{2a}{n}\sum_{k=1}^{n-1} k\lg k + \frac{2b}{n}(n-1) + \Theta(n).
\end{aligned}$$

از تقریب انتگرال ثابت می‌شود

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{4} n^2.$$

از این نامساوی و رابطه‌ی بالا داریم

$$\begin{aligned} \bar{T}(n) &\leq \frac{2a}{n}\left(\frac{n^2}{2}\lg n - \frac{n^2}{4}\right) + \frac{2b}{n}(n-1) + \Theta(n) \\ &\leq an\lg n + b + [\Theta(n) + b - \frac{an}{2}] \end{aligned} \qquad (۶-۷)$$

a و b را در فرمول ۶-۶ می‌توانیم طوری انتخاب کنیم تا مقدار درون پرانتز منفی شود. بنابراین $\bar{T}(n) \leq an\lg n + b$، و در نتیجه

$$\bar{T}(n)\Theta(n\lg n).$$

تمرین‌های زیربخش ۶-۳-۲

۶-۳-۲-۱ در الگوریتم مرتب‌سازی سریع اگر n عنصر مقادیر متفاوت داشته باشند، بزرگ‌ترین عنصر حداکثر چند بار جابه‌جا می‌شود؟

۶-۳-۲-۲ آیا همیشه می‌توان ورودی الگوریتم مرتب‌سازی سریع تصادفی را طوری ایجاد کرد که در زمان $O(n^2)$ اجرا شود؟

۶-۳-۲-۳ در گونه‌ی جدیدی از مرتب‌سازی سریع، برای انتخاب محور از میان n عنصر، $2\sqrt{n}+1$ عنصر اول آرایه را انتخاب می‌کنیم و با یک الگوریتم ساده مانند مرتب‌سازی درجی، آن‌ها را مرتب می‌کنیم. محور، عنصر میانه‌ی این تعداد عنصر مرتب‌شده است. بقیه‌ی الگوریتم مانند قبل عمل می‌کند. بدترین زمان اجرای این الگوریتم را محاسبه کنید؟

⋆ **۶-۳-۲-۴** در الگوریتم مرتب‌سازی سریع، یک بخش‌بندی «خوب» است اگر نسبت تعداد عناصر در دو بخش آرایه بین ۱/۴ و ۴ باشد. ما به‌صورت تصادفی محور را انتخاب می‌کنیم و اگر بخش‌بندی با این محور خوب نبود این کار را تکرار می‌کنیم تا یک بخش‌بندی خوب به‌دست آید. زمان اجرای این الگوریتم جدید را دقیقاً تحلیل کنید. ثابت کنید که این الگوریتم با احتمال بیش‌تر از $1/n - 1$ اعداد ورودی را به‌درستی و در زمان $c(n\lg n)$ برای یک c ثابت، مرتب می‌کند.

۶-۳-۲-۵ n عدد پیچ و n عدد مهره هر کدام با اندازه‌های متمایز داده شده‌اند. می‌دانیم که دقیقاً یک عدد از پیچ‌ها به یک عدد از مهره‌ها می‌خورد. تنها کار مجاز انتخاب یک عدد پیچ و یک عدد مهره و

«آزمون» این دو با هم است که آیا پیچ و مهره به هم می‌خورند یا این‌که کدام‌یک کوچک‌تر از دیگری است.

الف) می‌خواهیم با انجام تعدادی آزمون، کوچک‌ترین پیچ و کوچک‌ترین مهره (که مسلماً به هم می‌خورند) را پیدا کنیم. نشان دهید که برای $n = ۲$ مسئله را در بدترین حالت می‌توان با دو آزمون حل کرد. روشی ارائه دهید تا بتوان مسئله را در حالت کلی با $۲n - ۲$ آزمون حل کرد.

ب) می‌خواهیم با میانگین $O(n \lg n)$ عدد آزمون، پیچی را که به یک مهره داده‌شده می‌خورد پیدا کنیم. الگوریتمی برای این مسئله ارائه دهید و تحلیل کنید.

۶-۳-۶.۲ آرایه‌ی $A[۱ \cdots ۳n]$ از اعداد داده شده است. می‌خواهیم با مقایسه‌ی اعداد آرایه، دو عدد x و y $(x < y)$ را به‌دست آوریم به‌طوری‌که n عنصر A مقداری کم‌تر از x، n عنصر A مقداری بین x و y و n عنصر بقیه مقداری بیشتر از y داشته باشند. یک الگوریتم کارا برای حل این مسئله به‌میزان $M(n)$ حافظه‌ی اضافی (علاوه بر حافظه‌ی A) مصرف می‌کند و به زمان $T(n)$ نیاز دارد. کم‌ترین مقدار برای $M(n)$ و $T(n)$ چقدر است؟

۶-۳-۷.۲ احتمال این‌که RANDOMIZED-QUICKSORT به زمان $\Omega(n^۲)$ نیاز داشته باشد تا n عنصر مجزا از هم را مرتب کند چقدر است؟

۶-۳-۸.۲ یک لیست kـعددی، لیستی با n عنصر از k عدد مختلف است و هر کدام از این اعداد دقیقاً n/k بار در لیست ظاهر می‌شود. مثلاً $\langle ۱, ۱, ۲, ۲, ۳, ۳, ۴, ۴, ۵, ۵ \rangle$ یک لیست مرتب ۵ـعددی با اندازه‌ی ۱۰ است. یک الگوریتم $O(n \lg k)$ برای مرتب‌سازی یک لیست kـعددی با اندازه‌ی n ارائه دهید. برای سادگی فرض کنید که $n = ۲^i$ و $k = ۲^j$.

۶-۳-۳ مرتب‌سازی هرمی

مرتب‌سازی هرمی که در سال ۱۹۶۴ توسط ج.دبلیو. ویلیامز معرفی شد [۱۷]، یک الگوریتم کارای مرتب‌سازی درجا در حالت کلی است، که در بدترین حالت پیچیدگی بهینه دارد، هرچند که در عمل مرتب‌سازی سریع از آن سریع‌تر عمل می‌کند. این الگوریتم بر اساس داده‌ساختار هرم یا صف اولویت است که در بخش ۶-۴ معرفی شد. چنان‌که در آن بخش گفته شد، هرم بیشینه ساختاری است که در آرایه پیاده‌سازی می‌شود و برای n عنصر، می‌توان عمل درج، حذف بزرگ‌ترین عنصر و افزایش و کاهش کلید را در زمان کمینه‌ی $O(\lg n)$ و بدون حافظه‌ی اضافی انجام داد. در این بخش از رویه‌های تعریف شده در بخش ۶-۴ استفاده می‌کنیم.

در مرحله‌ی اول، مرتب‌سازی هرمی آرایه‌ی A با n عنصر را با فراخوانی BUILD-HEAP(A) به‌صورت درجا به شکل هرم بیشینه در می‌آورد. برای این‌کار، روشن است که عناصر با اندیس‌های $\lfloor \frac{n}{2} \rfloor + 1$ تا n چون فرزندی ندارند خودبه‌خود خاصیت هرم دارند. با این فرض، رویه‌ی MAX-HEAPIFY(A, i) به‌ترتیب برای مقادیر $\lfloor \frac{n}{2} \rfloor + 1$ تا ۱ فراخوانده می‌شود. روشن است که در انتها، کل آرایه‌ی A هرم است.

پس از آن، الگوریتم با استفاده از عمل «حذف بزرگ‌ترین عنصر» بر روی هرم A، عنصر اول (بزرگ‌ترین عنصر آرایه) را به‌جای حذف، با عنصر آخر جابه‌جا می‌کند و عناصر باقی‌مانده را مجدداً به‌صورت هرم در می‌آورد. اگر این کار را $n - 1$ بار تکرار کنیم، آرایه‌ی اصلی بدون نیاز به حافظه‌ی کمکی مرتب می‌شود.

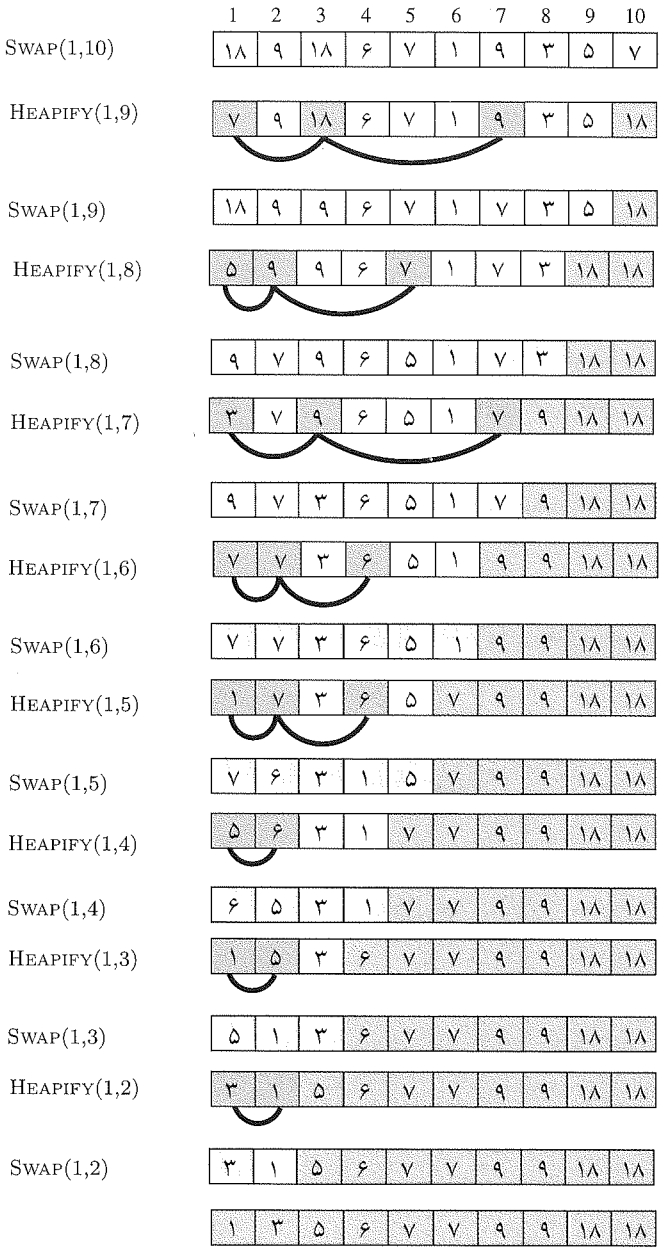
رویه‌های اصلی مرتب‌سازی هرمی به‌صورت زیر است:

```
BUILD-HEAP (A)
1  for i ← ⌊length[A]/2⌋ downto 1
2    do MAX-HEAPIFY(A, i)


HEAPSORT (A)
1  BUILD-HEAP(A)
2  for i ← length[A] downto 2
3    do swap(A[1], A[length[A]])
4       length[A] ← length[A] − 1
5       MAX-HEAPIFY (A, 1)
```

شکل ۶-۶ مراحل مختلف این الگوریتم را در مورد یک مثال و پس از ساخت هرم بیشینه نشان می‌دهد.

تحلیل ساخت هرم

چنان‌چه می‌دانیم، هزینه‌ی هر یک از اعمالی که بر روی هرم اجرا می‌شود برابر ارتفاع آن و یا $\mathcal{O}(\lg n)$ است. اگر رویه‌ی BUILD-HEAP را متناظر با n بار عمل MAX-HEAPIFY بگیریم، زمان اجرای آن $\mathcal{O}(n \lg n)$ به‌دست می‌آید. در این بخش نشان می‌دهیم که زمان



شکل ۶-۶ نمونه‌ی اجرای الگوریتم مرتب‌سازی هرمی پس از ساخت هرم بیشینه.

اجرای این رویه $\mathcal{O}(n)$ است نه $\mathcal{O}(n \lg n)$.

لم ۶-۱ بیشینه‌ی تعداد گره‌های با ارتفاع h در یک هرم با n عنصر برابر است با $\lceil \frac{n}{2^{h+1}} \rceil$.

اثبات: این لم را با استقرا بر روی h اثبات می‌کنیم. برای پایه‌ی استقرا، می‌دانیم که تعداد برگ‌های یک هرم برابر $\lceil \frac{n}{2} \rceil$ است، چون در نمایش آرایه‌ای، عناصر با اندیس i و $2i > n$ برگ هستند. فرض کنید در درخت T با n گره، n_h تعداد گره‌ها در ارتفاع h و نیز حکم برای گره‌های با ارتفاع $h-1$ درست باشد. اگر T' با n' گره همان T باشد که برگ‌های آن را برداشته باشیم، داریم $n' = n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$. همچنین می‌دانیم که گره‌هایی که ارتفاع‌شان در T برابر h است، در T' در ارتفاع $h-1$ قرار دارند. بنابراین،

$$n_h = n'_{h-1} \le \lceil n'/2^h \rceil = \lceil \lfloor n/2 \rfloor / 2^h \rceil \le \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil,$$

و حکم ثابت می‌شود. □

می‌دانیم که هزینه‌ی MAX-HEAPIFY بر روی گره‌های با ارتفاع h از $\mathcal{O}(h)$ است. با توجه به این‌که $0 \le h \le \lfloor \lg n \rfloor$، داریم

$$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \mathcal{O}(h) = \mathcal{O}(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}).$$

و از

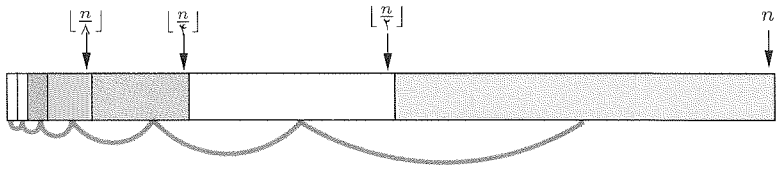
$$\sum_{h \ge 0} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2, \qquad (۶-۸)$$

نتیجه می‌گیریم که

$$T(n) = \mathcal{O}(n \sum_{h \ge 0} \frac{h}{2^h}) = \mathcal{O}(n).$$

راه‌حل دیگر

به‌عنوان راه‌حل دیگر، اگر تعداد کل تعویض‌ها را در بدترین حالت بشماریم، جواب دقیقی از کل هزینه‌ی ساخت هرم به‌دست می‌آوریم. فرض کنید $S(i)$ بیشینه‌ی تعداد تعویض‌های



شکل ۶-۷ تحلیل ساخت یک هرم از یک آرایه.

این الگوریتم برای اندیس i باشد. مطابق شکل ۶-۷، برای یک آرایه‌ی n تایی عناصری که اندیس آن‌ها بین $\lfloor n/4 \rfloor + 1$ تا $\lfloor n/2 \rfloor$ است، فقط یک فرزند در درخت نیمه‌مرتب دارند و بنابراین هنگام درج آن‌ها حداکثر یک تعویض داریم. تعداد این عناصر برابر $\lfloor n/2^2 \rfloor$ است. همچنین عناصری که اندیس آن‌ها بین $\lfloor n/8 \rfloor + 1$ تا $\lfloor n/4 \rfloor$ (به تعداد $\lfloor n/2^3 \rfloor$) است، حداکثر ۲ فرزند دارند، پس حداکثر ۲ تعویض خواهند داشت. در حالت کلی به‌تعداد $\lfloor n/2^{k+1} \rfloor$ عنصر (از اندیس $\lfloor n/2^{k+1} \rfloor + 1$ تا $\lfloor n/2^k \rfloor$) هر کدام حداکثر موجب k تعویض می‌شوند. این محاسبه‌ها در زیر نشان داده شده‌اند:

$$S(i) = \begin{cases} 1 & \text{برای } i = \lfloor n/2^2 \rfloor + 1 \ldots \lfloor n/2 \rfloor & \text{تعداد } = \lfloor n/2^2 \rfloor \\ 2 & \text{برای } i = \lfloor n/2^3 \rfloor + 1 \ldots \lfloor n/2^2 \rfloor & \text{تعداد } = \lfloor n/2^3 \rfloor \\ \ldots & \ldots & \\ k & \text{برای } i = \lfloor n/2^{k+1} \rfloor + 1 \ldots \lfloor n/2^k \rfloor & \text{تعداد } = \lfloor n/2^{k+1} \rfloor \\ \ldots & \ldots & \\ \lfloor \log n \rfloor & \text{برای } i = 1 & \text{تعداد } = 1 \end{cases}$$

حال مجموع بیشینه‌ی تعداد تعویض‌ها، یا $T(n)$ را می‌توانیم حساب کنیم:

$$T(n) \leq \sum_{k=1}^{\lfloor \log n \rfloor} k \frac{n}{2^{k+1}} \leq n. \qquad (۶-۹)$$

چرا که

$$\sum_{i=1}^{\infty} i/2^i = (1/2) + (1/4 + 1/4) + (1/8 + 1/8 + 1/8) + \ldots = 2.$$

و این رابطه را از طریق زیر هم می‌توان حساب کرد.

$$\begin{aligned} 1/2 + 1/4 + 1/8 + 1/16 + \cdots &= 1, \\ 1/4 + 1/8 + 1/16 + \cdots &= 1/2, \end{aligned}$$

$$۱/۸ + ۱/۱۶ + \cdots = ۱/۴,$$
$$\cdots = \cdots.$$

پس نتیجه می‌گیریم که تبدیل یک آرایه‌ی n تایی به یک هرم حداکثر n تعویض خواهد داشت و بنابراین از $\mathcal{O}(n)$ است. از سوی دیگر، این محاسبات نشان می‌دهد که در ساخت هرم، هزینه‌ی سرشکن‌شده‌ی درج هر عنصر $\mathcal{O}(۱)$ است.

البته چنان‌که در الگوریتم HEAPSORT آمد، در مرحله‌ی بعد $n - ۱$ تعویض داریم و اگر $h(i)$ هزینه‌ی عمل MAX-HEAPIFY$(1,i)$ باشد، داریم $h(i) \leq c \lg i$. پس قسمت دوم الگوریتم مرتب‌سازی هرمی، هزینه‌ای برابر زیر خواهد داشت:

$$\sum_{i=۲}^{n-۱} c \lg i \leq cn \lg n. \qquad (۱۰-۶)$$

یعنی مرتب‌سازی هرمی از $\mathcal{O}(n \lg n)$ است، و انتظاری کم‌تر از آن نمی‌توان داشت.

تمرین‌های زیربخش ۶-۳-۳

۶-۳-۱.۳ نشان دهید اگر تمام عناصر متفاوت باشند، زمان اجرای مرتب‌سازی هرمی در بهترین حالت $\Omega(n \lg n)$ است.

۶-۳-۲.۳ مشابه‌ی مسئله‌ی ۷.۴، می‌خواهیم ایده‌ی هرم دودویی را برای مرتب‌سازی تعمیم دهیم به‌طوری‌که گره‌های میانی از درخت به‌جای ۲ فرزند، d فرزند داشته باشد (به این داده‌ساختار هرم d تایی می‌گوییم).

الف) این هرم با آرایه چگونه پیاده‌سازی می‌شود؟ پدر و فرزندان $A[i]$ در کدام درایه قرار دارند؟ ارتفاع هرم d تایی با n عنصر چقدر است؟

ب) مانند هرم دودویی، ما از هرم d تایی برای مرتب‌سازی استفاده می‌کنیم. برای این کار، $d = ۱$ را انتخاب می‌کنیم. با این استدلال که در این حالت تنها عمل هزینه‌بر ساختن هرم است که آن هم با $\mathcal{O}(n)$ امکان‌پذیر است. بنابراین می‌توانیم با $\mathcal{O}(n)$، n عدد را مرتب کنیم. اشکال این استدلال چیست؟

پ) اگر $T(n,d)$ تعداد مقایسه‌های مرتب‌سازی n عنصر با استفاده از هرم d تایی باشد، با فرض آن‌که ساخت هرم $\mathcal{O}(n)$ است، $T(n,d)$ را به‌صورت $\Theta(.)$ و برحسب n و d به‌دست آورید. از این فرمول مقدار بهینه‌ی d را به‌دست آورید که با آن مقدار $T(n,d)$ کمینه شود.

۶-۳-۳.۳ آیا می‌توان الگوریتمی از مرتبه‌ی $\mathcal{O}(n)$ داشت که n عدد موجود در یک هرم را به‌ترتیب در یک آرایه بنویسد؟ در صورت پاسخ مثبت، الگوریتم خود را به‌طور کامل بیان و تحلیل کنید. برای پاسخ منفی نیز باید اثبات شما کامل باشد.

۶-۴ الگوریتم فورد-جانسون

می‌دانیم که کمینه‌ی تعداد مقایسه‌ی لازم (بین عناصر) برای مرتب‌سازی n عنصر برابر $\lceil \lg n! \rceil$ است. حال سؤالی که مطرح است آن است که آیا الگوریتمی با کم‌ترین تعداد مقایسه‌ی ممکن وجود دارد؟ در این بخش این مسئله را برای nهای کوچک بررسی می‌کنیم.

برای n برابر ۱، ۲ و ۳ الگوریتم مرتب‌ساز «مرتب‌سازی درج دودویی[۲۳]» که در صفحه‌ی ۶۲ بخش ۳-۱-۱ توضیح داده شد، تعداد مقایسه‌ی بهینه دارد (به‌ترتیب برابر ۰، ۱، ۳ و ۴) ولی برای $n > 3$ بهینه نیست. اگر $B(n)$ تعداد دقیق مقایسه‌های الگوریتم درج دودویی (در بدترین حالت ورودی) برای مرتب‌سازی n عنصر باشد، مقادیر $B(n)$ برای nهای کوچک و مقایسه‌ی آن با $\lceil \lg n! \rceil$ در جدول ۶-۲ آمده است.

جدول ۶-۲ تعداد مقایسه‌های مرتب‌سازی درج دودویی $B(n)$ برای تعداد کم عناصر و مقایسه‌ی آن با کران پایین.

$n =$	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱	۱۲	۱۳	۱۴	۱۵	۱۶	۱۷
$\lceil \lg n! \rceil =$	۰	۱	۳	۵	۷	۱۰	۱۳	۱۶	۱۹	۲۲	۲۶	۲۹	۳۳	۳۷	۴۱	۴۵	۴۹
$B(n) =$	۰	۱	۳	۵	۸	۱۱	۱۴	۱۷	۲۱	۲۵	۲۹	۳۳	۳۷	۴۱	۴۵	۴۹	۵۴

سؤالی که مطرح می‌شود آن است که آیا الگوریتمی با ۷ مقایسه برای مرتب‌سازی ۵ عنصر (در بدترین حالت ورودی) وجود دارد یا خیر؟ پاسخ به این سؤال مثبت است. اگر این عناصر را به‌دل‌خواه a، b، c، d و e بنامیم، روش این کار مطابق زیر است:

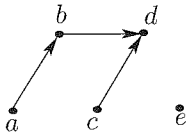
۱. a را با b و نیز c را با d مقایسه کن و عناصر کوچک‌تر و بزرگ‌تر هر مقایسه را پیدا کن. فرض می‌کنیم $a \leq b$ و $c \leq d$ باشد.

۲. با یک مقایسه‌ی دیگر دو عنصر کوچک بند فوق (یعنی b و d) را با هم مقایسه کن، فرض می‌کنیم که $b \leq d$ باشد. در انتهای این مرحله، ۵ عنصر به‌صورت شکل ۶-۸ در می‌آیند. در این شکل‌ها، $a \longrightarrow b$ یعنی $a \leq b$.

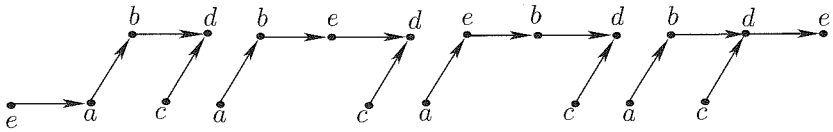
۳. ابتدا e را در زنجیره‌ی $a \leq b \leq d$ به‌روش دودویی درج کن، این کار حداکثر ۲ مقایسه نیاز دارد و یکی از حالات شکل ۶-۹ حاصل می‌شود.

---

[۲۳] binary insertion sort



شکل ۶-۸ مرحله‌ی اول مرتب‌سازی ۵ عنصر.



شکل ۶-۹ مرتب‌سازی ۵ عنصر: حالات مختلف پس از درج e.

۴. با توجه به‌این که $c \leq d$، درج c در زنجیره‌ی سه‌تایی (شامل a، b و e) یا دوتایی $a \leq b$، به‌روش دودویی حداکثر ۲ مقایسه‌ی دیگر نیاز دارد.

بنابراین ۵ عنصر حداکثر با ۷ مقایسه مرتب می‌شود و این بهینه است.

تعمیم الگوریتم

آقایان فورد[۲۴] و جانسون[۲۵] در سال ۱۹۵۹ الگوریتم بالا را به‌طرز جالبی تعمیم دادند [۶]. ایده‌ی اصلی این الگوریتم تشکیل یک «زنجیره‌ی اصلی» از عناصری است که مرتب هستند و تعیین یک ترتیب مشخص برای درج دیگر عناصر در این زنجیره به‌طوری‌که در انتها، زنجیره‌ی اصلی شامل همه‌ی عناصر شود.

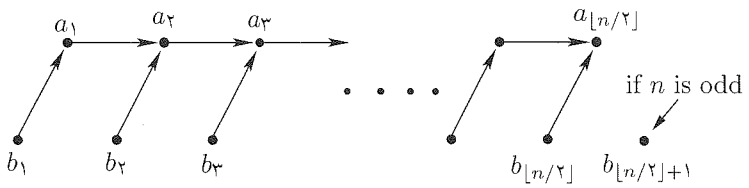
الگوریتم برای مرتب‌سازی n عنصر به‌صورت زیر است:

۱. عناصر را به $\lfloor \frac{n}{2} \rfloor$ جفت‌عنصر و یک عنصر اضافی (اگر n فرد باشد) تقسیم کن.

۲. هر دو عنصر یک جفت را با هم مقایسه و مرتب کن.

۳. به‌صورت بازگشتی $\lfloor \frac{n}{2} \rfloor$ عنصر بزرگ‌تر (از جفت‌عناصر) را نسبت به هم مرتب کن و پس از آن عناصر را مطابق شکل ۶-۱۰ نام‌گذاری کن. عناصر

[۲۴] Lester Ford Jr.
[۲۵] Slemer Johnson



شکل ۶-۱۰ مراحل اولیه‌ی الگوریتم فورد-جانسون.

$a_1 \le a_2 \le \ldots \le a_{\lfloor \frac{n}{2} \rfloor}$ را «زنجیره‌ی اصلی» می‌نامیم.

۴. عناصر b در شکل را با ترتیب زیر و به‌روش دودویی در بخشی از زنجیره‌ی اصلی درج کن. این ترتیب طوری انتخاب شده است که زنجیره‌ای که عنصر b در آن درج می‌شود حداکثر $2^k - 1$ عنصر داشته باشد تا این درج حداکثر k مقایسه نیاز داشته باشد:

الف) ابتدا b_3 را به زنجیره‌ی $b_1 \le a_1 \le a_2$ و سپس b_2 را به زنجیره‌ی $b_1 \le a_1$ و احتمالاً b_3 درج کن. تعداد عناصر این زنجیره هم حداکثر ۳ و درج با ۲ مقایسه امکان‌پذیر است.

ب) ابتدا b_5 را به زنجیره‌ی شامل ۷ عنصر a_1 تا a_4 و b_1 تا b_3 درج کن. سپس b_4 را به زنجیره‌ی a_1 تا a_3، b_1 تا b_3 و احتمالاً b_5 درج کن. تعداد عناصر این زنجیره حداکثر ۷ و هر درج با ۳ مقایسه امکان‌پذیر است.

پ) این روند را ادامه بده. در حالت کلی اگر $t_1, t_2, \ldots = 1, 3, 5, 11, \ldots$، در مرحله‌ی $k > 1$ ام هر عنصر b_{t_k} تا $b_{t_{k-1}+1}$ را به‌همین ترتیب در زنجیره‌ی عناصری که حتماً کوچک‌تر یا مساوی با آن عنصر هستند درج کن (شکل ۶-۱۱). آخرین مرحله شامل عنصر $b_{\lfloor \frac{n}{2} \rfloor+1}$ (در صورت وجود) می‌شود.

شکل ۶-۱۲ مثالی از ترتیب درج‌های عناصر b در زنجیره‌های مختلف را نشان می‌دهد.

برای محاسبه‌ی دقیق‌تر، فرض کنید

$$w_k = t_0 + t_1 + \cdots + t_{k-1} = \left\lfloor \frac{2^{k+1}}{3} \right\rfloor.$$

در این صورت، $(w_0, w_1, w_2, w_3, w_4, \cdots) = (0, 1, 2, 5, 10, 21, \cdots)$.

می‌توان ثابت کرد که

$$F(n) - F(n-1) = k \quad \Longleftrightarrow \quad w_k < n \le w_{k+1}, \tag{۶-۱۳}$$

و شرط آخر معادل است با

$$\frac{2^{k+1}}{3} < n < \frac{2^{k+2}}{3},$$

یعنی $k + 1 < \lg(3n) \le k + 2$. بنابراین

$$F(n) - F(n-1) = \left\lceil \lg\left(\frac{3}{4}n\right) \right\rceil.$$

جواب دقیق این رابطه‌ی بازگشتی، یا زمان اجرای الگوریتم فورد–جانسون، به‌صورت زیر است:

$$F(n) = \sum_{k=1}^{n} \left\lceil \lg\left(\frac{3}{4}k\right) \right\rceil. \tag{۶-۱۴}$$

جدول ۶-۳ تعداد مقایسه‌های الگوریتم فورد–جانسون را برای تعداد کم عناصر نشان می‌دهد.

جدول ۶-۳ تعداد مقایسه‌های الگوریتم فورد–جانسون $F(n)$ برای تعداد کم عناصر و مقایسه‌ی آن با کران پایین.

$n =$	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱	۱۲	۱۳	۱۴	۱۵	۱۶	۱۷
$\lceil \lg n! \rceil =$	۰	۱	۳	۵	۷	۱۰	۱۳	۱۶	۱۹	۲۲	۲۶	۲۹	۳۳	۳۷	۴۱	۴۵	۴۹
$F(n) =$	۰	۱	۳	۵	۷	۱۰	۱۳	۱۶	۱۹	۲۲	۲۶	۳۰	۳۴	۳۸	۴۲	۴۶	۵۰

$n =$	۱۸	۱۹	۲۰	۲۱	۲۲	۲۳	۲۴	۲۵	۲۶	۲۷	۲۸	۲۹	۳۰	۳۱	۳۲
$\lceil \lg n! \rceil =$	۵۳	۵۷	۶۲	۶۶	۷۰	۷۵	۸۰	۸۴	۸۹	۹۴	۹۸	۱۰۳	۱۰۸	۱۱۳	۱۱۸
$F(n) =$	۵۴	۵۸	۶۲	۶۶	۷۱	۷۶	۸۱	۸۶	۹۱	۹۶	۱۰۱	۱۰۶	۱۱۱	۱۱۶	۱۲۱

تمرین‌های بخش ۶-۴

۱.۴-۶ می‌دانیم که الگوریتم فورد-جانسون ۵ عنصر را با حداکثر ۷ مقایسه مرتب می‌کند.

الف) مراحل مختلف مرتب‌سازی ۱۰ عنصر با همین الگوریتم را با رسم شکل‌هایی نشان دهید. حداکثر تعداد مقایسه‌ها چند تاست؟

ب) با استفاده از بند «الف» نشان دهید که این الگوریتم ۲۱ عنصر را با حداکثر چند مقایسه مرتب می‌کند.

۲.۴-۶ رابطه‌ی ۶-۱۳ را ثابت کنید.

۳.۴-۶ ثابت کنید که برای $n \geq 22$، $F(n) > \lceil \lg n! \rceil$.

۶-۵ میانه‌ها و مرتبه‌های آماری

مرتبه‌ی آماری[۲۶] iام از یک مجموعه‌ی nعضوی، iامین عضو کوچک‌تر این مجموعه است. برای مثال، کمینه‌ی یک مجموعه، مرتبه‌ی آماری اول و بیشینه‌ی آن مرتبه‌ی آماری nام آن مجموعه است.

میانه به‌صورت شهودی «عضو وسطی» مجموعه است. برای nهای فرد، میانه یکی است و مرتبه‌ی آماری آن $i = (n+1)/2$ است. ولی اگر n زوج باشد، دو میانه داریم که در مرتبه‌های آماری $i = n/2$ و $i = n/2 + 1$ هستند. بنابراین بدون در نظر گرفتن زوجیت n، میانه‌ها در مرتبه‌های آماری $i = \lfloor (n+1)/2 \rfloor$ و $i = \lceil (n+1)/2 \rceil$ هستند.

در این بخش، روش‌های انتخاب مرتبه‌ی آماری iام از یک مجموعه با nعدد متفاوت را مورد بررسی قرار می‌دهیم. با این‌که برای راحتی اعداد را متفاوت در نظر می‌گیریم، نتایجی را که در این‌جا به‌دست می‌آوریم می‌توان به‌صورت شهودی به حالتی تعمیم داد که اعداد تکراری باشند. مسئله‌ی انتخاب[۲۷] را می‌توان به‌صورت زیر بیان کرد:

ورودی: مجموعه‌ی A از nعدد (متفاوت) و یک عدد i، که $1 \leq i \leq n$.
خروجی: عضو $x \in A$ که دقیقاً از $i-1$ عضو در A بزرگ‌تر است.

مسئله‌ی انتخاب را می‌توان در $O(n \lg n)$ حل کرد، به‌این ترتیب که اعداد را با استفاده از

---

[۲۶] order statistics
[۲۷] selection problem

یک الگوریتم بهینه (مانند مرتب‌سازی هرمی) مرتب می‌کنیم. در این صورت عنصر iام در آرایه‌ی مرتب‌شده جواب مسئله است، ولی الگوریتم‌های سریع‌تری نیز وجود دارند.

در این بخش، ابتدا مسئله‌ی یافتن هم‌زمان عنصر کمینه و بیشینه‌ی یک مجموعه را بررسی می‌کنیم، سپس به حل مسئله‌ی انتخاب می‌پردازیم. در بخش ۶-۵-۳ یک الگوریتم با زمان میانگین $O(n)$ را برای مسئله‌ی انتخاب پیشنهاد می‌کنیم و سپس در بخش ۶-۵-۴ الگوریتمی بیان می‌کنیم که این مسئله را در $O(n)$ حل می‌کند.

۶-۵-۱ کمینه و بیشینه

برای یافتن کمینه‌ی یک مجموعه nعضوی چند مقایسه لازم است؟ به‌راحتی می‌توانیم کران بالای $n-1$ را برای تعداد مقایسه‌ها پیدا کنیم: اعداد مجموعه را یک‌به‌یک بررسی می‌کنیم و کوچک‌ترین عضوی را که تا کنون دیده‌ایم نگه می‌داریم. در این روش فرض کرده‌ایم که مجموعه درون آرایه‌ی A ذخیره شده است، که $length[A] = n$.

```
MINIMUM (A)
1  min ← A[1]
2  for i ← 2 to length[A]
3     do if min > A[i]
4           then min ← A[i]
5  return min
```

البته، با تغییر عمل‌گر $>$ به $<$ می‌توان با $n-1$ مقایسه، عنصر بیشینه را به‌همین صورت پیدا کرد.

آیا این بهترین راه حل است؟ بله، برای این‌که می‌توانیم یک کران پایین از $n-1$ مقایسه را برای یافتن کمینه به‌دست آوریم. الگوریتمی تصور کنید که یافتن کمینه را به‌صورت یک دوره مسابقه بین عناصر در نظر می‌گیرد. هر مقایسه یک مسابقه است که در آن عضو کوچک‌تر پیروز می‌شود. نکته‌ی کلیدی این است که هر عنصر، بجز برنده‌ی نهایی باید دست‌کم در یک مسابقه شکست خورده باشد. بنابراین، دست‌کم $n-1$ مقایسه برای

یافتن کمینه لازم است، و الگوریتم MINIMUM از نظر تعداد کل مقایسه‌ها کمینه است.

۶-۵-۲ یافتن هم‌زمان بیشینه و کمینه

در برخی کاربردها، لازم است بیشینه و کمینه‌ی یک مجموعه‌ی n عضوی را با هم پیدا کنیم. برای مثال، ممکن است یک برنامه‌ی گرافیکی نیاز داشته باشد یک سری داده‌های به‌صورت (x, y) را طوری مقیاس کند که درون یک صفحه‌ی نمایش یا هر دستگاه گرافیکی مستطیل‌شکل دیگر قرار بگیرند. برای انجام این کار، برنامه باید ابتدا بیشینه و کمینه را در هر مختصه پیدا کند.

یافتن الگوریتمی که بتواند بیشینه و کمینه‌ی n عنصر را به‌طور هم‌زمان و با تعداد مقایسه‌هایی از مرتبه‌ی $\Omega(n)$ که از نظر آماری بهینه نیز هست پیدا کند، چندان مشکل نیست. کافی است تا بیشینه و کمینه را به‌صورت جدا از هم، و هر کدام را با $n - ۱$ مقایسه پیدا کنیم؛ و در مجموع $۲n - ۲$ مقایسه انجام داده‌ایم.

در حقیقت، تنها $۳\lceil n/۲ \rceil - ۲$ مقایسه برای یافتن هم‌زمان کمینه و بیشینه لازم است. در بخش بعد این کران پایین را ثابت می‌کنیم.

اثبات کران پایین

این اثبات براساس «استدلال رقابتی[۲۸]» است و مانند یک بازی بین ما (طراح الگوریتم) و حریف هوشمندی است که می‌خواهد ادعای بهینه بودن الگوریتم ما را رد کند. در هر مرحله، ما دو عنصر x و y را با هم مقایسه می‌کنیم و حریف که می‌خواهد موجب شود بیش‌ترین تعداد مقایسه‌ها را انجام دهیم، برای هر یک از این عناصر که تا کنون مقدارش مشخص نشده است، یکی از عناصر ورودی را انتخاب می‌کند و جواب می‌دهد که از دو عنصر x و y، کدام‌یک کوچک‌تر است. البته حریف دروغ نمی‌گوید و به‌تدریج داده‌ای برای الگوریتم ما تولید می‌کند که بدترین رفتار الگوریتم را موجب شود. بدیهی است اگر بتوانیم با فرض چنین حریفی با k مقایسه مسئله را حل کنیم، کران پایین تعداد مقایسه‌های مورد نیاز هر راه‌حل مسئله k است.

خلاصه‌ی اثبات چنین است که برای یافتن عنصر بیشینه، چنان‌چه در بالا هم گفته شد،

[۲۸] adverdary argument

حتماً به $n-1$ اطلاع نیاز داریم که هر اطلاع هم نتیجه‌ی مقایسه‌ی دو عنصر ورودی است. برای یافتن عنصر کمینه هم حتماً $n-1$ (و در مجموع $2n-2$) اطلاع لازم داریم. نکته این است که اگر مقایسه‌ها را درست انتخاب کنیم، با هر مقایسه ممکن است بتوانیم به‌جای یک اطلاع، دو اطلاع کسب کنیم.

با فرض مجزا بودن عناصر، اگر دو عنصر را که قبلاً مورد مقایسه قرار نگرفته باشند، با هم مقایسه کنیم و جواب را از حریف بخواهیم، با هر جوابی که او بدهد، یکی از این دو عنصر نمی‌تواند عنصر بیشینه و دیگری نمی‌تواند عنصر کمینه باشد (در واقع ۲ اطلاع با هر مقایسه به‌دست می‌آوریم). پس در مجموع با $\lfloor n/2 \rfloor$ مقایسه می‌توانیم $2\lfloor n/2 \rfloor$ اطلاع کسب می‌کنیم. با توجه به این‌که هر عنصر درگیر یک مقایسه شده است، می‌توان نشان داد که پرسش‌های بعدی هر کدام فقط یک اطلاع جدید به‌ما می‌دهند. بنابراین دست‌کم به

$$2n - 2 - 2\lfloor n/2 \rfloor$$

مقایسه‌ی دیگر نیاز داریم که در مجموع $2 - \lceil n/2 \rceil 3$ می‌شود.

راه‌حل تقسیم‌وحل

با استفاده از روش تقسیم‌وحل و مانند مرتب‌سازی ادغامی، آرایه را به دو بخش با $\lfloor \frac{n}{2} \rfloor$ و $\lceil \frac{n}{2} \rceil$ عنصر تقسیم می‌کنیم. به‌طور بازگشتی دو عنصر بیشینه و کمینه را در هر بخش پیدا می‌کنیم. روشن است که بیشینه‌ی دو عنصر بیشینه و کمینه‌ی دو عنصر کمینه جواب مسئله است.

اگر تعداد مقایسه‌های مورد نیاز روش $T(n)$ باشد، داریم،

$$T(n) = \begin{cases} 0 & n = 1, \\ 1 & n = 2, \\ T(\lfloor n/2 \rfloor + T(\lceil n/2 \rceil) + 2 & n > 2. \end{cases}$$

جواب این رابطه برای $n = 2^k$ همان مقدار بهینه است. این مطلب به‌سادگی با استقرا ثابت می‌شود

$$T(2^k) = 2[3 \times 2^{k-2} - 2] + 2 = 3 \times 2^{k-1} - 2,$$

اما این راه‌حل در برخی حالات بهینه نیست؛ مثلاً $T(6) = 8$ در حالی‌که نشان خواهیم داد می‌توان آن‌را با ۷ مقایسه $(2 - \lceil \frac{18}{3} \rceil)$ حل کرد.

راه‌حل بهینه

آرایه را به دو بخش با اندازه‌های ۲ و $n-2$ تقسیم می‌کنیم. با $T(n-2)+1$ مقایسه، کمینه و بیشینه‌ی هر بخش را پیدا می‌کنیم و با ۲ مقایسه‌ی دیگر، جواب نهایی را به‌دست می‌آوریم. در این صورت

$$T(n)=\begin{cases}0 & n=1,\\ 1 & n=2,\\ T(n-2)+3 & n>2.\end{cases}$$

که جواب آن همان مقدار بهینه است و این مطلب با استقرا به‌سادگی اثبات می‌شود.

۶-۵-۳ انتخاب در زمان میانگین خطی

به‌نظر می‌رسد که حل مسئله‌ی عمومی انتخاب مشکل‌تر از مسئله‌ی یافتن عنصر کمینه است. ولی، به‌طور غیر منتظره‌ای، مرتبه‌ی زمان اجرای هر دو مسئله برابر $\Theta(n)$ است. در این بخش، یک الگوریتم تقسیم‌وحل برای مسئله‌ی انتخاب ارائه می‌دهیم.

الگوریتم تصادفی انتخاب، یا RANDOMIZED-SELECT، مشابه‌ی مرتب‌سازی سریع عمل می‌کند. ایده‌ی ما، مانند مرتب‌سازی سریع، بخش‌بندی آرایه‌ی ورودی به دو بخش چپ و راست است به‌طوری‌که هر عنصر از بخش چپ از هر عنصر موجود در بخش راست بزرگ‌تر نباشد. ولی بر خلاف آن الگوریتم، که برروی هر دو بخش به‌صورت بازگشتی فراخوانده می‌شود، الگوریتم انتخاب کار را تنها در یکی از دو بخش دنبال می‌کند.

این تفاوت، در بررسی زمان اجرا خود را نشان می‌دهد: با وجود این‌که زمان اجرای میانگین مرتب‌سازی از $\Theta(n\lg n)$ است، زمان اجرای RANDOMIZED-SELECT در حالت میانگین، $\Theta(n)$ است.

این الگوریتم از رویه‌ی RANDOMIZED-PARTITION که در بخش ۶-۳-۱ گفته شد استفاده می‌کند. بنابراین، مشابه RANDOMIZED-QUICKSORT، یک الگوریتم تصادفی است، زیرا بخشی از رفتار آن با استفاده از خروجی تولیدکننده‌ی عدد تصادفی تعیین می‌شود.

رویه‌ی RANDOMIZED-SELECT زیر، iامین عنصر کمینه را در آرایه‌ی $A[p\cdots r]$ برمی‌گرداند.

```
RANDOMIZED-SELECT (A, p, r, i)
 1  if i > r − p + 1
 2     then error overflow
 3  if p = r
 4     then return A[p]
 5  q ← RANDOMIZED-PARTITION (A, p, r)
 6  k ← q − p + 1
 7  if i ≤ k
 8     then return RANDOMIZED-SELECT (A, p, q, i)
 9     else return RANDOMIZED-SELECT (A, q + 1, r, i − k)
```

پس از اجرای RANDOMIZED-PARTITION در سطر ۵، آرایه‌ی $A[p\cdots r]$ به دو زیرآرایه‌ی غیر تهی $A[p\cdots q]$ و $A[q+1\cdots r]$ تقسیم می‌شود به‌طوری‌که هر عضو $A[p\cdots q]$ از هر عضو $A[q+1\cdots r]$ کوچک‌تر است. در سطر ۶، عدد k یعنی تعداد عناصر در زیرآرایه‌ی $A[p\cdots q]$ محاسبه می‌شود. حال می‌توان تشخیص داد که iامین کمینه در کدام‌یک از این دو بخش است. اگر $i \le k$، آن‌گاه عنصر مطلوب در بخش چپ است و در سطر ۸ به‌صورت بازگشتی از زیرآرایه‌ی مذکور انتخاب می‌شود. اما اگر $i > k$ باشد، آن‌گاه عنصر مورد نظر $(i-k)$امین عنصر کمینه از بخش راست است، که به‌صورت بازگشتی در سطر ۹ پیدا می‌شود.

بدترین زمان اجرای الگوریتم RANDOMIZED-SELECT از مرتبه‌ی $\Theta(n^2)$ است، زیرا ممکن است بسیار بدشانس باشیم و محور یک زیرآرایه همیشه عنصر کمینه‌ی آن انتخاب شود. در نتیجه، ممکن است یکی از بخش‌ها فقط یک عنصر داشته باشد و لازم باشد کار را در بخش بعدی دنبال کنیم. البته، به‌خاطر تصادفی بودن این الگوریتم، هیچ ورودی خاصی به این بدی نخواهد بود.

می‌توانیم یک کران بالا برای $T(n)$، زمان اجرای میانگین RANDOMIZED-SELECT، روی آرایه‌ای از n عنصر را به‌صورت زیر پیدا کنیم.

در بخش ۶-۳-۱ دیدیم که الگوریتم RANDOMIZED-PARTITION یک بخش‌بندی تولید می‌کند که بخش کوچک‌تر آن به‌احتمال $2/n$ دارای ۱ عنصر است و به احتمال $1/n$ دارای i $(i = 2, 3, \cdots, n-1)$ عنصر است. با فرض این‌که دنباله‌ی ورودی صعودی یک‌نوا باشد، بدترین حالت RANDOMIZED-SELECT همیشه بدشانس است، یعنی iامین کمینه در

بخش بزرگ‌تر محور می‌افتد. بنابراین، به رابطه‌ی بازگشتی زیر می‌رسیم.

$$T(n) \leq \frac{1}{n}\left[T(\max(1,n-1)) + \sum_{k=1}^{n-1} T(\max(k,n-k))\right] + \mathcal{O}(n), \quad (۶-۱۵)$$

$$\leq \frac{1}{n}\left[T(n-1) + 2\sum_{k=\lceil n/2\rceil}^{n-1} T(k)\right] + \mathcal{O}(n), \quad (۶-۱۶)$$

$$= \frac{2}{n}\sum_{k=\lceil n/2\rceil}^{n-1} T(k) + \mathcal{O}(n). \quad (۶-۱۷)$$

می‌دانیم $\max(1,n-1) = n-1$، در نتیجه، رابطه‌ی ۶-۱۶ از ۶-۱۵ نتیجه می‌شود. به‌طور کلی،

$$\max(k,n-k) = \begin{cases} k & \text{اگر } k \geq \lceil n/2\rceil\ , \\ n-k & \text{اگر } k < \lceil n/2\rceil\ . \end{cases}$$

اگر n فرد باشد، آن‌گاه هر کدام از جمله‌های

$$T(\lceil n/2\rceil),\ T(\lceil n/2\rceil+1),\cdots,T(n-1)$$

دوبار در مجموع ظاهر می‌شود، و اگر n زوج باشد هر کدام از جمله‌های

$$T(\lceil n/2\rceil+1), T(\lceil n/2\rceil+2),\cdots,T(n-1)$$

دوبار و جمله‌ی $T(\lceil n/2\rceil)$ یک بار ظاهر می‌شود. در هر دو حالت، مجموع در رابطه‌ی ۶-۱۵ از بالا به مجموع رابطه‌ی ۶-۱۶ محدود می‌باشد. عبارت ۶-۱۷ هم از این نتیجه می‌شود که $T(n-1) = \mathcal{O}(n^2)$، و بنابراین جمله‌ی $\frac{1}{n}T(n-1)$ را می‌توان در مقابل $\mathcal{O}(n)$ نادیده گرفت.

رابطه‌ی بازگشتی ۶-۱۷ را با استقرا حل می‌کنیم. فرض کنید که برای ثابتی مانند c، داشته باشیم $T(n) \leq cn$ که شرایط اولیه‌ی رابطه‌ی بازگشتی را برآورده سازد. با استفاده از فرض استقرا خواهیم داشت،

$$T(n) \leq \frac{2}{n}\sum_{k=\lceil n/2\rceil}^{n-1} ck + \mathcal{O}(n)$$

$$\leq \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2\rceil-1}\right) + \mathcal{O}(n)$$

$$
\begin{aligned}
&= \frac{۲c}{n}\left(\frac{۱}{۲}(n-۱)n - \frac{۱}{۲}\left(\left\lceil\frac{n}{۲}\right\rceil - ۱\right)\left\lceil\frac{n}{۲}\right\rceil\right) + \mathcal{O}(n)\\
&\le c(n-۱) - \frac{c}{n}\left(\frac{n}{۲} - ۱\right)\left(\frac{n}{۲}\right) + O(n)\\
&= c\left(\frac{۳}{۴}n - \frac{۱}{۲}\right) + \mathcal{O}(n)\\
&\le cn,
\end{aligned}
$$

زیرا می‌توانیم c را به‌قدر کافی بزرگ انتخاب کنیم تا $c(n/۴ + ۱/۲)$ از $\mathcal{O}(n)$ پیشی بگیرد. بنابراین هر مرتبه‌ی آماری، و در حالت خاص میانه را می‌توان با زمان میانگین خطی یافت.

تمرین‌های زیربخش ۶-۵-۳

۱.۳-۵-۶ نشان دهید که دومین کمینه از میان n عنصر را می‌توان در بدترین حالت با $n + \lceil\lg n\rceil - ۲$ مقایسه یافت.

۲.۳-۵-۶ یک نسخه‌ی غیربازگشتی از RANDOMIZED-SELECT را بنویسید.

۳.۳-۵-۶ فرض کنید از RANDOMIZED-SELECT برای یافتن عنصر کمینه در آرایه‌ی $A = \langle ۳, ۲, ۹, ۰, ۷, ۵, ۴, ۸, ۶, ۱\rangle$ استفاده کنیم. دنباله‌ای از تقسیم‌های متوالی را مشخص کنید که به بدترین حالت کارایی RANDOMIZED-SELECT منتهی شود.

۴.۳-۵-۶ به‌یاد بیاورید که در صورت وجود عناصر برابر، رویه‌ی RANDOMIZED-PARTITION زیرآرایه‌ی $A[p\cdots r]$ را به دو زیرآرایه‌ی ناتهی $A[p\cdots q]$ و $A[q+۱\cdots r]$ تقسیم می‌کند به‌طوری‌که هر عنصر از $A[p\cdots q]$ کوچک‌تر یا مساوی هر عنصر از $A[q+۱\cdots r]$ است.

بررسی کنید که اگر عناصر برابر داشته باشیم، آیا RANDOMIZED-SELECT درست کار می‌کند؟

⋆ **۵.۳-۵-۶** فرض کنید ماشینی هست که عنصر میانه‌ی n عنصر را در زمان $\mathcal{O}(\sqrt{n})$ محاسبه می‌کند. آیا می‌توان با استفاده از این ماشین n عنصر را در زمان $\mathcal{O}(n)$ مرتب کرد؟

۶.۳-۵-۶ آیا در یک آرایه‌ی n تایی، می‌توان تعداد عناصر مجزا از هم را در زمان میانگین $\mathcal{O}(n)$ به‌دست آورد؟

۷.۳-۵-۶ n دانش‌جو در یک سالن ورزشی بزرگ به‌صورت تصادفی روی زمین نشسته‌اند. دانش‌جوی i در مختصات (x_i, y_i) قرار دارد. مربی می‌خواهد در نقطه‌ای بایستد که مجموع فاصله‌هایش تا همه‌ی دانش‌جویان کمینه شود. الگوریتمی از $\mathcal{O}(n)$ برای پیدا کردن مکان مربی پیدا کنید.

۶-۵-۸.۳ یک الگوریتم از مرتبه‌ی $O(n \lg n)$ ارائه دهید تا از میان n عدد داده‌شده در یک آرایه، یک عدد را که دقیقاً k بار تکرار شده است به‌دست آورد. k در ورودی داده می‌شود و مستقل از n است.

۶-۵-۴ انتخاب خطی در بدترین حالت

در این بخش یک الگوریتم انتخاب را بررسی می‌کنیم که زمان اجرای آن در بدترین حالت $O(n)$ است. مانند RANDOMIZED-SELECT، الگوریتم SELECT عنصر دل‌خواه را با بخش‌بندی بازگشتی آرایه‌ی ورودی پیدا می‌کند. ولی ایده‌ی اساسی این الگوریتم، تضمین یک بخش‌بندی خوب از آرایه است. SELECT از الگوریتم بخش‌بندی قطعی PARTITION در مرتب‌سازی سریع استفاده می‌کند، با کمی تغییر که در آن عنصر محور به‌عنوان پارامتر به آن داده شود.

الگوریتم SELECT، i امین عنصر کمینه آرایه‌ی ورودی از n عنصر را با اجرای مراحل زیر پیدا می‌کند.

۱. n عنصر آرایه‌ی ورودی را به $\lfloor n/5 \rfloor$ گروه، هر کدام شامل ۵ عنصر و حداکثر یک گروه با $n \bmod 5$ عنصر باقی‌مانده، تقسیم کن.

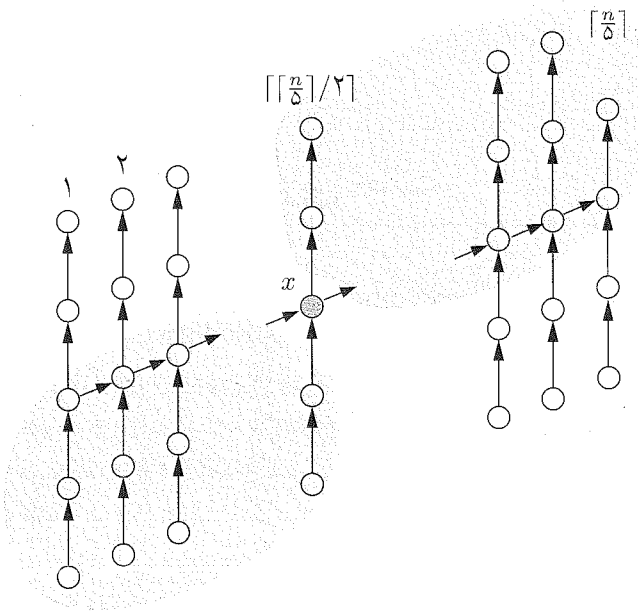
۲. عنصر میانه‌ی هر کدام از $\lceil n/5 \rceil$ گروه را با انجام یک الگوریتم ساده، مثلاً مرتب‌سازی درجی پیدا کن. (اگر گروه آخر دارای تعداد زوجی عنصر بود، میانه‌ی بزرگ‌تر را انتخاب کن.) شکل ۶-۱۳ این مراحل از الگوریتم را نشان می‌دهد.

۳. از رویه‌ی SELECT به‌صورت بازگشتی استفاده کن تا میانه‌ی x از $\lceil n/5 \rceil$ میانه‌ی به‌دست آمده در مرحله‌ی ۲ به‌دست آید.

۴. با استفاده نسخه‌ی تغییریافته‌ی الگوریتم بخش‌بندی آرایه‌ی ورودی را حول محور x تقسیم کن. فرض کن k تعداد عناصر در بخش پایین نقطه‌ی تقسیم باشد، پس تعداد عناصر بخش دیگر $n - k$ است.

۵. اگر $i \leq k$ از رویه‌ی SELECT به‌صورت بازگشتی برای یافتن i امین عنصر کمینه در بخش پایینی استفاده کن، وگرنه $(i - k)$ امین عنصر کمینه را در بخش دیگر پیدا کن.

برای بررسی زمان اجرای الگوریتم SELECT، ابتدا یک کران پایین برای تعداد عناصر بزرگ‌تر از محور انتخابی، یا x، پیدا می‌کنیم. به شکل ۶-۱۳ برای تجسم این استدلال مراجعه کنید.



شکل ۶-۱۳ انتخاب با بدترین زمان $\mathcal{O}(n)$ و تحلیل آن.

دست‌کم نصف میانه‌هایی که در مرحله‌ی ۲ پیدا شدند بزرگ‌تر یا مساوی x هستند. بنابراین دست‌کم $\lceil \frac{n}{10} \rceil - ۱$ گروه ۳ عنصر دارند که بزرگ‌تر از x هستند (در این‌جا گروهی که شامل x هست حساب نکرده‌ایم). به‌طور مشابه دست‌کم $\lceil \frac{n}{10} \rceil - ۲$ گروه ۳ عنصر دارند که کوچک‌تر از x هستند (در این‌جا نیز گروه شامل x و گروه آخر را که ممکن است کم‌تر از ۵ عنصر داشته باشد، حساب نکرده‌ایم). کمینه‌ی این دو مقدار برابر زیر است:

$$۳\left(\left\lceil \frac{۱}{۲}\left\lceil \frac{n}{۵} \right\rceil \right\rceil - ۲\right) \geq \frac{۳n}{۱۰} - ۶$$

بنابراین، در بدترین حالت، SELECT به‌صورت بازگشتی در بخشی کار را ادامه می‌دهد که حداکثر $n - (\frac{۳n}{۱۰} - ۶) = \frac{۷n}{۱۰} + ۶$ عنصر دارد.

با این ترتیب، رابطه‌ی بازگشتی برای زمان بدترین حالت اجرای SELECT، یا $T(n)$ به این صورت به‌دست می‌آید: مراحل ۱، ۲، و ۴ زمان $\mathcal{O}(n)$ مصرف می‌کنند. (مرحله‌ی ۲ شامل $\mathcal{O}(n)$ بار اجرای مرتب‌سازی درجی بر روی اندازه‌ی $\mathcal{O}(۱)$ است.) مرحله‌ی ۳ به‌اندازه‌ی $T(\lceil n/۵ \rceil)$ و مرحله‌ی ۵ حداکثر $T(۷n/۱۰ + ۶)$ زمان مصرف می‌کند. توجه کنید که برای $n > ۲۰$ داریم $۷n/۱۰ + ۶ < n$ و هم‌چنین هر ورودی مرکب از ۸۰ یا تعداد

کم‌تری عنصر به زمان $\mathcal{O}(۱)$ نیاز دارد. بنابراین به رابطه‌ی بازگشتی زیر می‌رسیم:

$$T(n) \leq \begin{cases} \Theta(۱) & \text{اگر} \quad n \leq ۸۰ \\ T(\lceil n/۵ \rceil) + T(۷n/۱۰ + ۶) + \mathcal{O}(n) & \text{اگر} \quad n > ۸۰ \end{cases} \qquad (۱۸-۶)$$

با استقرا نشان می‌دهیم که این زمان خطی است. فرض کنید که برای یک c مفروض و هر $n \leq ۸۰$ داشته باشیم $T(n) \leq cn$. با استقرا در سمت راست این رابطه خواهیم داشت

$$\begin{aligned} T(n) &\leq c\lceil n/۵ \rceil + c(۷n/۱۰ + ۶) + \mathcal{O}(n), \\ &\leq cn/۵ + c + ۷cn/۱۰ + ۶c + \mathcal{O}(n), \\ &\leq ۹cn/۱۰ + ۷c + \mathcal{O}(n), \\ &\leq cn, \end{aligned}$$

زیرا می‌توانیم برای هر $n > ۸۰$، c را به‌قدر کافی بزرگ اختیار کنیم تا $c(n/۱۰ - ۷)$ بزرگ‌تر از تابعی باشد که با جمله‌ی $\mathcal{O}(n)$ مشخص می‌شود. بنابراین زمان اجرای بدترین حالت رویه‌ی SELECT خطی است.

مانند مرتب‌سازی مقایسه‌ای، SELECT و RANDOMIZED-SELECT اطلاعاتی راجع به مکان نسبی عناصر را فقط با مقایسه پیدا می‌کنند. بنابراین، ویژگی خطی بودن، مستقل از ترتیب ورودی است، درست مانند حالتی که در الگوریتم‌های مرتب‌سازی داشتیم.

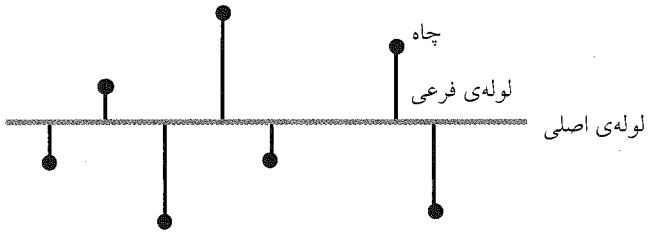
تمرین‌های بخش ۶-۵

۱.۵-۶ روی یک جاده تعدادی روستا قرار دارند که پراکندگی آن‌ها از قاعده‌ی خاصی پیروی نمی‌کند. می‌خواهیم یک پمپ بنزین در یکی از روستاها بسازیم تا مجموع فاصله‌ی همه‌ی روستاها از پمپ بنزین کمینه شود. نشان دهید که بهترین روستا برای این کار روستای میانه است.

۲.۵-۶ در الگوریتم SELECT، عناصر ورودی را به گروه‌های ۵ تایی تقسیم کردیم. اگر اندازه‌ی گروه‌ها را ۷ انتخاب می‌کردیم، آیا الگوریتم باز هم در زمان خطی اجرا می‌شد؟ اگر گروه‌ها ۳ تایی بودند چطور؟

۳.۵-۶ الگوریتم SELECT را تحلیل کنید و نشان دهید که تعداد عناصر بزرگ‌تر از میانه‌ی میانه‌ها، یعنی x، و تعداد عناصر کوچک‌تر از x دست‌کم $\lceil n/۲ \rceil$ است، البته به‌شرط این‌که $n \geq ۳۸$.

۴.۵-۶ نشان دهید که چگونه می‌توان کاری کرد که الگوریتم مرتب‌سازی سریع در بدترین حالت در زمان $\mathcal{O}(n \lg n)$ اجرا شود.



شکل ۶-۱۴ شکل مربوط به مسئله‌ی ۶-۵.۹.

⋆ **۶-۵.۵** فرض کنید که یک الگوریتم برای یافتن i امین عنصر کمینه در یک مجموعه از n عنصر، فقط از مقایسه استفاده می‌کند. نشان دهید که این الگوریتم می‌تواند بدون انجام مقایسه‌ی اضافی، $i-1$ عنصر کوچک‌تر و $n-i$ عنصر بزرگ‌تر را نیز پیدا کند.

۶-۵.۶ یک زیربرنامه که به‌عنوان یک «جعبه‌ی سیاه» داده شده است، در بدترین حالت با زمان خطی میانه را پیدا می‌کند. با استفاده از این جعبه‌ی سیاه، یک الگوریتم خطی ساده برای حل مسئله‌ی انتخابِ عنصری با هر مرتبه‌ی آماریِ داده‌شده طراحی کنید.

۶-۵.۷ یک الگوریتم با زمان اجرای $O(n)$ پیدا کنید، به‌طوری‌که با دریافت مجموعه‌ی S از n عضو متمایز و یک عدد مثبت $k \le n$، k امین عدد در S را که به میانه‌ی S نزدیک است بیابد.

۶-۵.۸ فرض کنید که $X[1\cdots n]$ و $Y[1\cdots n]$ دو آرایه‌ی مرتب با n عنصر هستند. یک الگوریتم با زمان اجرای $O(\lg n)$ طراحی کنید که بتواند میانه‌ی تمام $2n$ عنصر موجود در آرایه‌های X و Y را پیدا کند.

۶-۵.۹ پروفسور «اولی» با یک شرکت نفتی هم‌کاری می‌کند. این شرکت در حال طراحی یک خط لوله‌ی عظیم است که از شرق به غرب در یک منطقه‌ی نفتی با n چاه می‌گذرد. از هر چاه، یک خط لوله‌ی فرعی وجود دارد که مستقیماً و از نزدیک‌ترین راه (شمال یا جنوب) به لوله‌ی اصلی وصل می‌شود (مانند شکل ۶-۱۴). با در دست داشتن مختصه‌های x و y چاه‌ها، پروفسور اولی محل بهینه‌ی عبور لوله‌ی اصلی را چگونه باید تعیین کند تا مجموع طول لوله‌های فرعی کمینه شود؟ نشان دهید که محل بهینه را می‌توان در زمان خطی پیدا کرد.

۶-۵.۱۰ الگوریتم زیر برای یافتن k امین عنصر بین n عنصر در آرایه‌ی A پیش‌نهاد می‌شود:

۱. A را به دو بخش تقریباً مساوی تقسیم کن،

۲. میانه‌ی هر بخش را به‌صورت بازگشتی به‌دست آور،

۳. محور x را کوچک‌ترین میانه قرار بده

۴. براساس محور x، A را به سه بخش تقسیم‌کن (بخش کم‌تر، مساوی و بزرگ‌تر از x)،

۵. براساس اندازه‌ی این سه بخش، کار یافتن عنصر مطلوب را در یکی از این سه بخش دنبال کن.

الف) چگونه می‌توان بخش‌بندی گفته‌شده را در زمان خطی انجام داد؟

ب) نشان دهید که زمان اجرای این الگوریتم $\mathcal{O}(n^{\log_{۴}۳})$ است.

۶-۵.۱۱ n وزنه با وزن‌های متفاوت و نامشخص با شماره‌های ۱ تا n و یک ترازوی دو کفه‌ای بدون وزنه‌ی دیگر داده شده‌اند. نشان دهید که می‌توان حداکثر با $n + \lceil \lg n \rceil - ۲$ بار توزین، سبک‌ترین و دومین سبک‌ترین وزنه را پیدا کرد. (یک بار توزین یعنی قرار دادن دو وزنه‌ی i و j در دو کفه‌ی ترازو و یادداشت این‌که $i < j$ است یا $j < i$.)

۶-۵.۱۲ لیست مرتب از i عدد بزرگ‌تر

با داشتن مجموعه‌ای از n عدد، می‌خواهیم با استفاده از یک الگوریتم مقایسه‌ای، i عدد بزرگ‌تر را به‌ترتیب نزولی به‌دست آوریم. برای هر یک از روش‌های زیر، الگوریتمی بیابید که زمان اجرای آن در بدترین حالت به‌طور مجانبی کم‌ترین مقدار ممکن باشد، و زمان اجرای آن را بر حسب n و i تحلیل کنید.

الف) عددها را مرتب کن و i عدد بزرگ‌تر را فهرست کن.

ب) یک صف اولویت با استفاده از اعداد بساز و i بار عمل «حذف بزرگ‌ترین» را فراخوانی کن.

پ) با استفاده از الگوریتم مرتبه‌ی آماری، iامین بزرگ‌ترین عدد را پیدا کن و آن را محور در الگوریتم بخش‌بندی قرار بده، سپس بخش بزرگ‌تر را مرتب‌کن.

*** ۶-۵.۱۳ میانه‌ی وزن‌دار**

میانه‌ی وزن‌دار برای n عدد متفاوت $x_۱$، $x_۲$ تا x_n، به‌ترتیب با وزن‌های $w_۱$، $w_۲$ تا w_n، به‌طوری‌که $\sum_{i=۱}^{n} w_i = ۱$، عنصر x_k است که خاصیت‌های زیر را داشته باشد:

$$\sum_{x_i < x_k} w_i < \frac{۱}{۲},$$

$$\sum_{x_i > x_k} w_i \leq \frac{۱}{۲}$$

الف) استدلال کنید که میانه‌ی $x_۱$، $x_۲$ تا x_n، همان میانه‌ی وزن‌دار x_iها با وزن‌های $w_i = \frac{۱}{n}$ برای $i = ۱, ۲, \ldots, n$ است.

ب) نشان دهید که با استفاده از یک الگوریتم مرتب‌سازی، می‌توان در بدترین حالت در زمان $\mathcal{O}(n \lg n)$، میانه وزن‌دار n عنصر را پیدا کرد.

پ) نشان دهید که با استفاده از الگوریتم یافتن میانه در زمان خطی، می‌توان میانه‌ی وزن‌دار را در بدترین حالت در $\Theta(n)$ پیدا کرد؟

«مسئله‌ی مکان‌یابی اداره‌ی پست» به این صورت تعریف می‌شود: n نقطه p_1 p_2 تا p_n با وزن‌های w_1، w_2 تا w_n داده شده‌اند. می‌خواهیم نقطه‌ی p (نه لزوماً از میان نقاط داده شده) را به‌گونه‌ای بیابیم که $\sum_{i=1}^{n} w_i d(p,p_i)$ کمینه شود ($d(a,b)$ فاصله‌ی بین دو نقطه‌ی a و b است).

ت) بحث کنید که میانه‌ی وزن‌دار بهترین جواب برای حالت یک بعدی مسئله‌ی مکان‌یابی اداره پست است، که در آن نقاط تنها اعدادی حقیقی هستند و فاصله‌ی بین نقاط a و b برابر $d(a,b) = |a-b|$ تعریف می‌شود.

ث) بهترین جواب را برای حالت دو بعدی مسئله‌ی مکان‌یابی اداره پست پیدا کنید که در آن نقاط در مختصات مختلف (x,y) داده می‌شوند و فاصله‌ی بین دو نقطه‌ی $a = (x_1,y_1)$ و $b = (x_2,y_2)$، فاصله منهتنی[۲۹] بین این دونقطه، یا $d(a,b) = |x_1 - x_2| + |y_1 - y_2|$ تعریف می‌شود.

⋆ ۶-۵-۱۴ مرتبه‌های آماری کوچک

در بخش ۶-۵-۴ ثابت کردیم که $T(n)$، تعداد مقایسه‌های رویه‌ی SELECT برای یافتن عنصری با مرتبه‌ی i از میان n عدد در بدترین حالت $T(n) = \Theta(n)$ است. اما ضریب ثابت نهفته در این نماد Θ نسبتاً بزرگ است. هنگامی که i نسبت به n کوچک باشد، می‌توانیم رویه‌ی دیگری را اجرا کنیم که SELECT را به‌عنوان یک زیررویه فرا می‌خواند، ولی در بدترین حالت تعداد مقایسه‌های کم‌تری انجام می‌دهد.

الف) الگوریتمی پیش‌نهاد کنید که برای یافتن عنصر با مرتبه‌ی i، $U_i(n)$ مقایسه انجام دهد، که در آن:

$$U_i(n) = \begin{cases} T(n) & i \geq \frac{n}{2} \\ \lfloor \frac{n}{2} \rfloor + U_i(\lceil \frac{n}{2} \rceil) + T(2i) & \text{دیگر موارد} \end{cases}$$

(راهنمایی: با مقایسه‌ی دو به دوی $\lfloor \frac{n}{2} \rfloor$ زوج مجزا شروع کنید و روی مجموعه‌ای که عنصر کوچک‌تر هر زوج را دارد به‌طور بازگشتی ادامه دهید.)

ب) نشان دهید که اگر $i < \frac{n}{2}$، رابطه‌ی $U_i(n) = n + O(T(2i)\lg(\frac{n}{i}))$ برقرار است.

پ) نشان دهید که اگر a ثابتی کم‌تر از $\frac{n}{2}$ باشد، داریم $U_i(n) = n + O(\lg(n))$.

ت) نشان دهید که اگر $k \geq 2$، $i = \frac{n}{k}$ آن گاه $U_i(n) = n + O(T(\frac{2n}{k})\lg k)$.

---

[۲۹] Manhattan distance

۶-۶ مرتب‌سازی خارجی

در این نوع مرتب‌سازی، فرض می‌کنیم که اندازه‌ی داده‌ی ورودی آن‌قدر بزرگ است که یک‌جا در حافظه جای نمی‌گیرد و به‌ناچار فقط می‌توان بخش کوچکی از ورودی را در حافظه‌ی اصلی قرار داد و بقیه باید در حافظه‌ی خارجی ذخیره شود. الگوریتم‌هایی را که بر روی چنین داده‌هایی کار می‌کنند، «الگوریتم‌های خارجی[۳۰]» می‌گوییم. در این حالت، زمان دست‌رسی به یک بلوک اطلاعات بر روی حافظه‌ی جانبی و خواندن آن‌ها در حافظه‌ی اصلی، هزاران برابر بیش‌تر از زمان دست‌رسی در حافظه‌ی اصلی است. از این‌رو، معیار سنجش کارایی الگوریتم‌های خارجی، تعداد دسترسی‌ها به حافظه‌ی خارجی (دیسک) در نظر می‌گیریم. همچنین، زمان پردازش مورد نیاز پس از هر خواندن از حافظه‌ی جانبی، یک مقدار ثابت در نظر گرفته می‌شود، چون اندازه‌ی میان‌گیر[۳۱]ی که در حافظه‌ی اصلی این داده را در خود ذخیره می‌کند، مستقل از اندازه‌ی داده است و ثابت فرض می‌شود.

الگوریتم‌های خارجی زیادی وجود دارند ولی در این بخش از کتاب، فقط الگوریتم‌های مرتب‌سازی را مطالعه می‌کنیم.

در بررسی الگوریتم‌های مرتب‌سازی خارجی فرض می‌کنیم:

- اطلاعات بر روی فایل‌های حافظه‌ی خارجی به‌صورت ترتیبی ذخیره شده است؛ یعنی، برای خواندن رکورد mام باید $m-1$ رکورد قبلی آن را خوانده باشیم.
- هر فایل شامل n رکورد است و هر رکورد یک کلید یک‌تا دارد.
- هدف ایجاد فایلی است که رکوردهای آن براساس کلیدشان مرتب باشند.
- با هر دسترسی به دیسک، یک بلوک به‌اندازه‌ی k رکورد خوانده می‌شود.
- تعداد فایل‌هایی که در یک زمان می‌توانند باز باشند حداکثر برابر مقدار ثابت r است.
- اندازه‌ی حافظه‌ی اصلی قابل استفاده (یا میان‌گیرها) از مرتبه‌ی $\mathcal{O}(1)$ است.
- عملیات مقایسه و محاسبات دیگر فقط در حافظه‌ی اصلی انجام می‌شود.

دراین بخش چند الگوریتم مرتب‌سازی خارجیِ مبتنی بر ادغام را ارائه و تحلیل می‌کنیم.

---

[۳۰] external algorithms

[۳۱] buffer

۶-۶-۱ مرتب‌سازی ادغامی خارجی

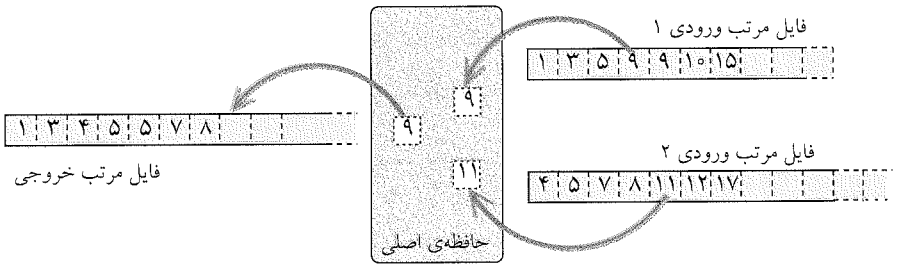
این روش مبتنی بر ادغام دو قطعه‌ی مرتب از دو فایل ورودی و ایجاد یک قطعه‌ی مرتب خروجی از آن‌دو است. هر قطعه می‌تواند بخشی از یک فایل باشد.

ادغام دو قطعه‌ی مرتب

فرض کنید دو قطعه‌ی مرتب از دو فایل بر روی حافظه‌ی خارجی داده شده‌اند. می‌خواهیم این دو قطعه را ادغام کنیم و حاصل را به‌عنوان قطعه‌ای مرتب در بخشی از یک فایل در حافظه‌ی خارجی بنویسیم. این‌کار مشابه ادغام دو آرایه‌ی مرتب است.

در حافظه‌ی اصلی به دو عدد میان‌گیر هرکدام به‌اندازه‌ی یک رکورد نیاز داریم تا بتوانیم رکوردهای قطعه‌های ورودی را در آن بخوانیم. یک میان‌گیر هم برای نوشتن رکورد در فایل خروجی مورد نیاز است.

در ابتدا، اولین رکورد هر دو قطعه (کوچک‌ترین کلید) را می‌خوانیم و در میان‌گیر مربوط به آن‌ها می‌نویسیم. هر بار رکوردهای میان‌گیرها را با هم مقایسه می‌کنیم و رکورد کوچکتر را در میان‌گیر خروجی می‌نویسیم تا در انتهای قطعه‌ی خروجی نوشته شود. سپس، رکورد بعدی را از همان قطعه‌ای که رکوردش را نوشته‌ایم می‌خوانیم و در همان میان‌گیر می‌نویسیم. این کار را تا خواندن و نوشتن همه‌ی رکوردهای دو قطعه‌ی ورودی ادامه می‌دهیم. ادغام دو فایل مرتب هم مانند همین الگوریتم است که شکل ۶-۱۵ آن را نشان می‌دهد.



شکل ۶-۱۵ ادغام دو قطعه‌ی مرتب.

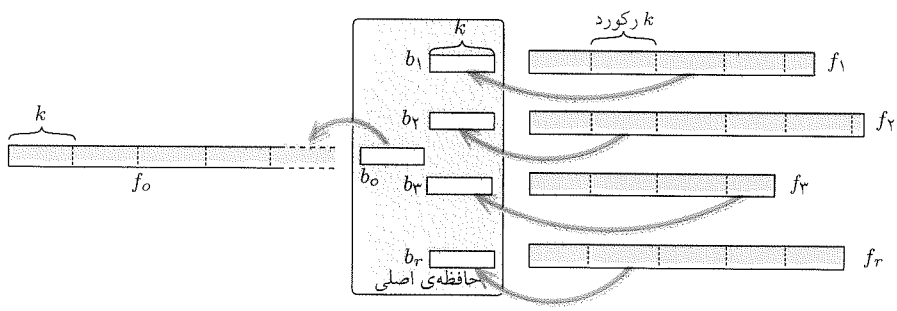
اگر قطعه‌ی اول n_1 رکورد و قطعه‌ی دوم n_2 رکورد داشته باشد و $k = 1$، با $n_1 + n_2$ بار خواندن و همین تعداد نوشتن می‌توان ادغام را انجام داد. اگر با هر دست‌رسی به حافظه‌ی خارجی $k > 1$ رکورد نوشته یا خوانده شود، این الگوریتم به‌تعداد $\lceil \frac{n_1}{k} \rceil + \lceil \frac{n_2}{k} \rceil$ بار دست‌رسی به حافظه‌ی خارجی کار ادغام را انجام می‌دهد.

ادغام r قطعه‌ی مرتب

در حالت کلی، الگوریتم گفته‌شده در بالا را نیز می‌توان برای r فایل مرتب انجام داد، به‌طوری‌که با هر بار دست‌رسی به حافظه‌ی خارجی یک بلوک متشکل از k رکورد از فایل خوانده یا نوشته شود. در این صورت، $r + 1$ میان‌گیر هر کدام به‌طول یک بلوک یا k رکورد نیاز است.

در ابتدا، از هر یک از فایل‌های ورودی اولین بلوک آن را می‌خوانیم. به‌کمک یک الگوریتم داخلی، این بلوک‌ها را در هم ادغام کرده و در بلوک خروجی یا b_o می‌نویسیم. در این الگوریتم، هر زمان که b_o پر شود، آن را در انتهای فایل خروجی f_o می‌نویسیم و هر زمان که به‌انتهای هر یک از میان‌گیرهای ورودی مثلاً b_i برسیم، یک بلوک دیگر را از فایل f_i می‌خوانیم و کار را دنبال می‌کنیم.

شمایی از این الگوریتم در شکل ۶-۱۶ نشان داده شده است. این الگوریتم به‌تعداد $\lceil \frac{\sum_{i=1}^{r} n_i}{k} \rceil$ بار از فایل‌های ورودی می‌خواند و به‌تعداد $\sum_{i=1}^{r} \lceil \frac{n_i}{k} \rceil$ بار در فایل خروجی می‌نویسد. مقدار حافظه‌ی مورد نیاز برای میان‌گیرها هم به‌اندازه‌ی $(r + 1)k$ رکورد است.



شکل ۶-۱۶ ادغام r فایل مرتب. با هر بار دست‌رسی به دیسک k رکورد خوانده یا نوشته می‌شود.

الگوریتم کلی

برای راحتی کار، این الگوریتم را برای $k = ۱$ بیان می‌کنیم. یک فایل ورودی f_{in} و چهار فایل $f_۱$، $f_۲$، $g_۱$ و $g_۲$ در حافظه‌ی خارجی مورد نیاز است.

الگوریتم به صورت زیر عمل می‌کند (شکل ۶-۱۷ را ببینید):

۱. فایل ورودی را به دو فایل $f_۱$ و $f_۲$ با حداکثر یک رکورد اختلاف، تقسیم کن.

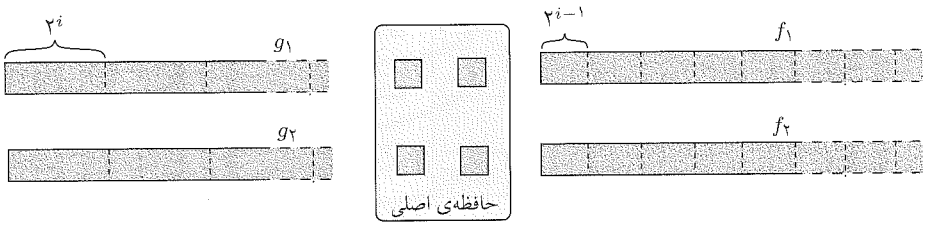
۲. برای $i = ۱, \ldots, M$ مراحل زیر را تکرار کن:
در هر مرحله فرض می‌کنیم که $f_۱$ و $f_۲$ (یا $g_۱$ و $g_۲$) شامل قطعات مرتبی هر یک به‌طول $۲^{i-۱}$ هستند و تعداد قطعات دو فایل ورودی حداکثر یک واحد اختلاف دارد.

۲-۱) $f_۱$ و $f_۲$ را به‌عنوان فایل‌های ورودی در نظر بگیر. هر بار، قطعات با شماره‌های یکسان $f_۱$ و $f_۲$ را با یک‌دیگر ادغام کن. حاصل هر ادغام قطعه‌ای مرتب به‌طول ۲^i است (بجز قطعه‌ی آخر که ممکن است طولش کم‌تر باشد). این قطعات را یک‌بار در میان در $g_۱$ و $g_۲$ بنویس.

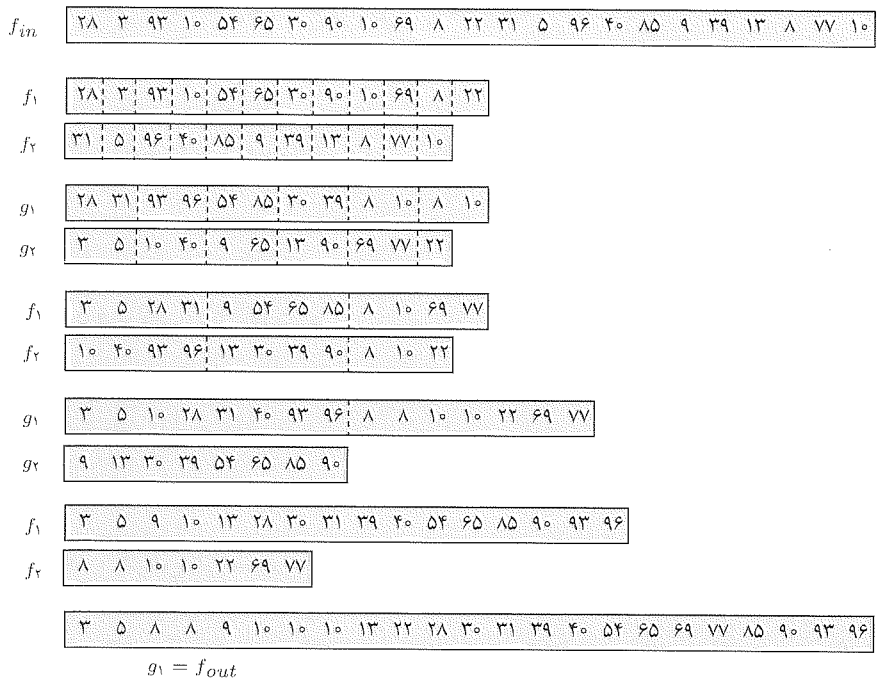
۲-۲) $g_۱$ و $g_۲$ را به‌عنوان فایل‌های ورودی و $f_۱$ و $f_۲$ را به عنوان فایل‌های خروجی در نظر بگیر و مراحل بالا را تکرار کن.

به‌سادگی و با استقرا می‌توان نشان داد که در انتهای مرحله‌ی iام هر فایل خروجی دارای قطعات مرتب و به‌طول ۲^i است (بجز حداکثر یک قطعه که طول آن از ۲^i کم‌تر است). همچنین، تعداد قطعه‌های دو فایل خروجی حداکثر یک واحد اختلاف دارند.

بنابراین در انتهای مرحله‌ی $M = \lceil \log n \rceil$ (تعداد تکرار در گام ۲ الگوریتم فوق)، یکی از فایل‌های خروجی حاوی تنها یک قطعه‌ی مرتب از تمام n رکورد فایل ورودی و دیگری



شکل ۶-۱۷ الگوریتم مرتب‌سازی ادغامی خارجی.



شکل ۶-۱۸ مثالی از الگوریتم مرتب‌سازی ادغامی خارجی.

خالی است. با توجه به‌این‌که در هر تکرار همه‌ی n رکورد موجود یک‌بار خوانده و یک‌بار هم نوشته می‌شوند، تعداد دست‌رسی‌ها به دیسک در مجموع برابر $۲n(\lceil \log n \rceil + ۱)$ است ($۲n$ بار خواندن و نوشتن برای تقسیم فایل اصلی منظور شده است).

این الگوریتم را می‌شود برای حالت $k > ۱$ هم تعمیم داد. در این صورت، تعداد دست‌رسی‌ها برابر $۲\lceil \frac{n}{k} \rceil (\lceil \log n \rceil + ۱)$ خواهد بود. شکل ۶-۱۸ مثالی از این الگوریتم را نشان می‌دهد.

با تقسیم فایل ورودی به r فایل با اندازه‌های یکسان و با استفاده از r حافظه نیز می‌توان فایل را مرتب کرد. در آن صورت، تعداد دست‌رسی‌ها به دیسک $۲n\lceil \log_r n \rceil + ۲n$ می‌شود و در حالت کلی برابر $۲\lceil \frac{n}{k} \rceil (\lceil \log_r n \rceil + ۱)$ خواهد بود. البته در این حالت، به حافظه‌ای به‌اندازه‌ی $k(r + ۱)$ نیاز است. این حالت کلی را روش «ادغام چندگانه[۳۲]» می‌گویند.

[۳۲] multiway merge

۶-۶-۲ مرتب‌سازی خارجی چندفازه

مرتب‌سازی چندفازه[۳۳] همان مرتب‌سازی ادغامی است که به‌جای استفاده از ۴ فایل کمکی، به‌طرز جالبی فقط ۳ فایل استفاده می‌کند. مراحل مختلف این الگوریتم به‌صورت زیر است:

۱. به فایل ورودی کم‌ترین تعداد رکورد را با کلید ∞ اضافه‌کن تا طول آن n برابر iامین عدد فیبوناچی (یا F_i) شود.

۲. فایل ورودی را به دو فایل با اندازه‌های F_{i-1} و F_{i-2} تقسیم کن (می‌دانیم که $F_i = F_{i-1} + F_{i-2}$).

۳. به‌تعداد M بار تکرار کن.

الف) فرض می‌کنیم f_1 دارای F_m قطعه‌ی مرتب است (در ابتدا $F_m = F_i$) که اندازه‌ی هر قطعه‌ی آن F_r است (در ابتدا $F_r = F_0 = 1$). همچنین f_2 دارای F_{m+1} قطعه‌ی مرتب است که اندازه‌ی هر قطعه‌ی آن F_{r+1} است، و f_3 خالی است.

ب) F_m قطعه‌ی مرتب از فایل f_1 را با همین تعداد قطعه‌ی مرتب از فایل f_2 در هم ادغام کن و حاصل را در f_3 بنویس.

پ) حال f_1 خالی، f_2 دارای $F_{m-1} = F_{m+1} - F_m$ قطعه به‌اندازه‌ی F_{r+1} و f_3 دارای F_m قطعه‌ی مرتب به‌اندازه‌ی $F_{r+2} = F_r + F_{r+1}$ است.

ت) نام‌گذاری فایل‌ها را به تناسب تغییر بده.

جدول ۶-۴ مراحل مختلف این الگوریتم را برای مرتب‌سازی یک فایل به اندازه‌ی ۳۴ نشان می‌دهد. در ابتدا، این فایل به دو فایل با اندازه‌های ۲۱ و ۱۳ تقسیم می‌شود، چون $F_9 = F_8 + F_7 = 21 + 13 = 34$. در این جدول، اگر فایلی حاوی k قطعه‌ی مرتب هر کدام به‌اندازه‌ی r باشد، آن را به‌صورت $k(r)$ نمایش داده‌ایم. ۱۳ قطعه‌ی f_1 و ۱۳ قطعه‌ی f_2 را با هم ادغام می‌کنیم تا ۱۳ قطعه‌ی به‌اندازه‌ی ۲ در فایل f_3 تولید شوند. ۸ قطعه به‌اندازه‌ی ۱ در f_2 باقی می‌ماند. با توجه به این‌که قطعات فایل f_1 همه خوانده شده‌اند، در گام بعدی این فایل به‌عنوان خروجی عمل می‌کند. در این گام، ۸ قطعه به‌اندازه‌ی ۱ از فایل f_2 را در ۸ قطعه به‌اندازه‌ی ۲ از فایل f_3 ادغام می‌کنیم و ۸ قطعه‌ی

[۳۳] polyphase merge sort

جدول ۶-۴ مثال مرتب‌سازی خارجی چندفازه برای $n = ۳۴$ عنصر.

پس از گام	$f_۱$	$f_۲$	$f_۳$
ابتدا	۱۳(۱)	۲۱(۱)	--
۱	--	۸(۱)	۱۳(۲)
۲	۸(۳)	--	۵(۲)
۳	۳(۳)	۵(۵)	--
۴	--	۲(۵)	۳(۸)
۵	۲(۱۳)	--	۱(۸)
۶	۱(۱۳)	۱(۲۱)	--
۷	--	--	۱(۳۴)

مرتب به‌اندازه‌ی ۳ در فایل $f_۱$ می‌نویسیم. فایل $f_۲$ خالی می‌شود و ۵ قطعه به‌اندازه‌ی ۲ در فایل $f_۳$ باقی می‌ماند. این کار را تکرار می‌کنیم؛ در انتهای گام ششم فایل $f_۱$ و $f_۲$ هر کدام فقط یک قطعه دارند، ولی اندازه‌ی قطعات مرتب آن‌ها به‌ترتیب ۲۱ و ۱۳ (همان $F_۷$ و $F_۸$) است. با یک‌بار دیگر ادغام این دو قطعه، فایل خروجی حاوی ۱ قطعه‌ی مرتب ولی به‌اندازه‌ی ۳۴ به‌دست می‌آید، که همان مرتب‌شده‌ی فایل ورودی است.

اثبات درستی و تحلیل

هر مرحله از این الگوریتم را با

$$a(b) + c(d) \longrightarrow e(f) + g(h)$$

نشان می‌دهیم، یعنی یک فایل حاوی a قطعه‌ی مرتب هرکدام به‌اندازه‌ی b، با فایلی که c قطعه‌ی مرتب هرکدام به‌اندازه‌ی d دارد ادغام و دو فایل دیگر با مشخصات ذکر شده ایجاد می‌کنیم. به‌عنوان ویژگی مستقل از حلقه، ثابت می‌کنیم که در هر مرحله اعداد a تا h فوق اعداد فیبوناچی هستند و در ابتدا و انتهای هر مرحله یکی از فایل‌ها تهی است. و نیز این‌که اگر F_i کوچک‌ترین عدد فیبوناچی بزرگ‌تر یا مساوی n باشد، در گام $i - r - ۱$ام (برای $۳ \le r \le i - ۲$) از الگوریتم، رابطه‌ی زیر بین این اعداد برقرار است:

$$F_r(F_{i-r+۱}) + F_{r-۱}(F_{i-r}) \longrightarrow F_{r-۱}(F_{i-r+۲}) + F_{r-۲}(F_{i-r+۱}) \qquad (۱۹-۶)$$

برای اثبات پایه‌ی استقرا داریم

$$F_{i-1}(F_2) + F_{i-2}(F_1) \longrightarrow F_{i-2}(F_3) + F_{i-3}(F_2)$$

چون $F_i = F_{i-1} + F_{i-2}$، $F_1 = F_2 = 1$ و $F_3 = 2$. همچنین چون $F_{i-2} < F_{i-1}$، به‌تعداد F_{i-2} رکورد از دو فایل ورودی می‌خوانیم و در یکی از فایل‌های خروجی می‌نویسیم. بدیهی است که تعداد قطعات این فایل خروجی برابر F_{i-2} و اندازه‌ی هر قطعه‌ی آن برابر $F_1 + F_2 = F_3$ خواهد بود. همچنین از فایل بزرگ‌تر به‌تعداد $F_{i-1} - F_{i-2} = F_{i-3}$ قطعه به‌اندازه‌ی F_2 باقی می‌ماند.

با همین استدلال و با فرض درستی پایه‌ی استقرا، می‌توان فرمول ۶-۱۹ را ثابت کرد. همچنین روشن است که اگر این کار را ادامه دهیم، در انتهای گام $i-4$ام (برای $r=3$) دو فایل خروجی هر کدام یک قطعه دارند و اندازه‌هایشان به‌ترتیب برابر F_{i-1} و F_{i-2} است. بنابراین، در انتهای گام $i-3$ام همه‌چیز به‌خوبی در یک فایل مرتب می‌شود.

برای تحلیل، روشن است که تعداد گام‌های الگوریتم برابر $i-4$ است و مطابق فرمول ۶-۱۹ در گام $i-r-1$ام الگوریتم به‌تعداد $2F_{i-2}F_{i-r+2}$ رکورد می‌خواند و می‌نویسد. با احتساب تقسیم اولیه‌ی فایل که به‌اندازه‌ی $2F_i$ دسترسی به دیسک نیاز دارد، در مجموع الگوریتم به تعداد

$$2F_i + \sum_{r=3}^{i-2} 2F_{i-2}F_{i-r+2}$$

به حافظه‌ی خارجی دسترسی خواهد داشت.

تمرین‌های بخش ۶-۶

۶-۶.۱ الگوریتم مرتب‌سازی خارجی چندفازه را برای فایلی به اندازه‌ی ۱۰۰ دنبال کنید. مشخص کنید در مجموع چند عمل خواندن و نوشتن روی دیسک انجام می‌شود.

۶-۶.۲ یک DVD حاوی n رکورد با کلیدهای نامرتب داده شده است. می‌خواهیم رکورد با کلید میانه را به‌دست آوریم. امکان کپی‌کردن همه‌ی فایل در حافظه‌ی اصلی نیست. فقط حداکثر می‌توانیم از مقدار $O(\lg n)$ حافظه‌ی اصلی استفاده کنیم. DVD غیر قابل نوشتن است. الگوریتمی با متوسط زمان $O(n \lg n)$ برای این‌کار ارائه دهید.

تمرین‌های فصل ۶

⋆ **۱.۶** ثابت کنید که می‌توان n عدد مجزا از هم را که بین ۱ تا $n^{\lg n}$ هستند در $\mathcal{O}(n \lg\lg n)$ مرتب کرد.

۲.۶ مرتب‌سازی خسته‌کننده

یک الگوریتم مرتب‌سازی را «خسته‌کننده» می‌گوییم اگر جای‌گشتی از n عنصر مجزا از هم به‌عنوان ورودی وجود داشته باشد که یک عنصر خاص از آن ورودی، به‌وسیله‌ی الگوریتم مورد نظر، در مجموع $\Omega(n)$ بار با بقیه‌ی عناصر مقایسه شود. کدام‌یک از الگوریتم‌های مرتب‌سازی که می‌شناسید خسته‌کننده است؟[۳۴]

۳.۶ فرض کنید A آرایه‌ای نامرتب از n عدد است به‌طوری‌که فقط k عدد متمایز دارد (فرض کنید $k < \sqrt{n}$). می‌خواهیم A را با الگوریتمی از مرتبه‌ای کم‌تر از $\mathcal{O}(n \lg n)$ مرتب کنیم. برای این‌کار،

الف) آرایه‌ی مرتب B را از k مقدار متفاوت در A بسازید.

ب) آرایه‌ی A را با کمک B در زمان $o(n \lg n)$ مرتب کنید.

الگوریتم‌های هر دو مرحله‌ی بالا را تشریح کنید. داده‌ساختارهای مورد نیاز را نام ببرید. الگوریتم بند «ب» را به‌صورت شبه‌کد بنویسید و آن‌را تحلیل کنید.

۴.۶ دنباله‌ی زیگزاگی

دنباله‌ی $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ از اعداد را «زیگزاگی[۳۵]» می‌نامیم اگر یک $0 \le t < n$ وجود داشته باشد به‌طوری‌که ترتیب $\langle a_t, a_{t+1}, \ldots, a_{t+n-1} \rangle$ شامل دو قطعه، یکی اکیداً صعودی و دیگری اکیداً نزولی باشد (یعنی $a_t < a_{t+1} < \cdots < a_k > a_{k+1} > a_{k+2} > \cdots > a_{t+n-1}$). البته توجه کنید که اندیس‌ها به‌هنگ n هستند. یعنی اگر دنباله‌ی $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ را t محل به‌چپ شیفت دورانی دهیم، دنباله‌ی حاصل شامل دو قطعه، یکی اکیداً صعودی و دیگری اکیداً نزولی خواهد بود.

الگوریتمی کارا برای به‌دست آوردن بزرگ‌ترین عدد در یک دنباله‌ی زیگزاگی پیش‌نهاد کنید و آن را ثابت و تحلیل کنید.

۵.۶ می‌خواهیم یک آرایه‌ی A با n عنصر را به k زیرآرایه‌ی $A_1, \ldots, A_k$ با تعداد $\lfloor n/k \rfloor$ یا $\lceil n/k \rceil$ عنصر تقسیم کنیم، به‌طوری‌که برای همه‌ی $i < j$ و هر $a_i \in A_i$ و هر $a_j \in A_j$ داشته باشیم $a_i < a_j$. الگوریتمی از $\mathcal{O}(n \lg k)$ برای این‌کار پیش‌نهاد کنید.

۶.۶ یک ماتریس $A[1 \cdots n, 1 \cdots m]$ شامل $r \le mn$ عدد است به‌طوری‌که عناصر هر سطر و هر ستون مستقلاً به‌صورت صعودی مرتب هستند؛ سطرها از چپ به راست و ستون‌ها از بالا به پایین.

---

[۳۴] آزمون ورودی کارشناسی ارشد، رشته‌ی مهندسی کامپیوتر، سال ۱۳۸۷

[۳۵] bitonic

تعدادی از عناصر ماتریس ∞ هستند. نشان دهید که عمل جست‌وجو در این جدول را می‌توان در $O(n+m)$ انجام داد. (جست‌وجو برای x یعنی آیا x در جدول وجود دارد یا خیر.)

۷.۶ گرسنه‌ها و سوپ

n نفر گرسنه و از خودراضی و n ظرف سوپ موجودند. گرمای ظرف i برابر b_i است و این دما در طول حل این مسئله تغییر نمی‌کند. نفر iام ظرفی دقیقاً با دمای t_i می‌خواهد. خوش‌بختانه می‌دانیم t_iها مجزا و متفاوت هستند و برای هر دمای خواسته‌شده‌ی t_i دقیقاً یک ظرف وجود دارد. تنها کار مجاز آن است که یک نفر و یک ظرف را انتخاب کنیم و او با چشیدن محتوای آن ظرف اعلام کند که آیا ظرف (نسبت به دمای مورد علاقه‌ی او) خیلی سرد است، خیلی داغ است یا این‌که دمایش کاملاً مناسب است. این کار را «چشیدن» می‌نامیم. می‌خواهیم با متوسط تعداد $O(n \lg n)$ بار چشیدن، ظرف مورد علاقه‌ی هر فرد را پیدا کنیم.

الف) الگوریتم این کار را ارائه و تحلیل کنید.

ب) اگر جواب هر چشیدن فقط «خوب» یا «بد» (هم برای داغ و هم برای سرد) باشد، سریع‌ترین الگوریتم برای این مسئله را ارائه دهید و آن‌را تحلیل کنید.

۸.۶ فرض کنید که آرایه‌ی $A[1 \cdots n]$ از اعداد زیگزاگی است (به تعریف مربوط در مسئله‌ی ۶-۴ مراجعه کنید).

الف) نشان دهید که پس از انجام عمل زیر، دنباله‌های $A[1 \cdots \lfloor \frac{n}{2} \rfloor]$ و $A[\lceil \frac{n}{2} \rceil \cdots n]$ نیز مستقلاً زیگزاگی هستند.

```
BITONIC-COMPARE (A)
1  for i ← 1 to ⌊n/2⌋
2    do if A[i] > A[i + ⌊n/2⌋]
3         then SWAP (A[i], A[i + ⌊n/2⌋])
```

ب) نشان دهید که یک آرایه‌ی زیگزاگی را می‌توان در $O(n)$ مرتب کرد.

۹.۶ مرتب‌سازی پن‌کیکی

الگوریتم Pancake Sort[۳۶] بر اساس رفتار آشپزها با پن‌کیک‌ها بنا شده است. فرض کنید n پن‌کیک برروی هم قرار گرفته‌اند. بر روی هر یک عددی نوشته شده که قابل رؤیت است. تنها کار ممکن فرو کردن کف‌گیر قبل از یک پن‌کیک خاص و وارونه‌کردن همه‌ی پن‌کیک‌های بالای کف‌گیر است. این کار را عمل «وارون» می‌گوییم که می‌تواند از هر جا انجام شود. هدف این الگوریتم، مرتب‌سازی صعودی اعداد روی پن‌کیک‌ها با استفاده از کم‌ترین تعداد عمل «وارون» است.

---

[۳۶] پن‌کیک غذایی است به شکل کلوچه‌ی گرد و نازک که به‌عنوان صبحانه صرف می‌شود. آشپزها معمولاً تعداد زیادی از پن‌کیک‌ها را روی هم قرار می‌دهند و با کف‌گیر، دسته‌ای از آن‌ها را برمی‌گردانند.

اگر تعداد پن‌کیک‌ها $n \geq ۶$ باشد، نشان دهید که با $۲n - ۲$ بار انجام عمل «وارون» می‌توان یقیناً اعداد را مرتب کرد.

⋆ ۱۰.۶ ادغام دنباله‌های مرتب

می‌خواهیم k دنباله‌ی مرتب هر کدام به‌اندازه‌ی n/k عنصر را در هم ادغام کنیم و از آن‌ها یک دنباله‌ی n عضوی به‌دست بیاوریم.

الف) برای این کار، ابتدا دنباله‌ی ۱ را با دنباله‌ی ۲، سپس حاصل را با دنباله‌ی ۳، و ...، و در پایان، حاصل را با دنباله‌ی nام ادغام می‌کنیم. زمان اجرای این الگوریتم را در بدترین حالت برحسب n و k تحلیل کنید.

ب) الگوریتمی از $\mathcal{O}(n \lg k)$ برای این کار ارائه دهید.

پ) ثابت کنید که $\frac{n!}{((n/k)!)^k}$ راه مختلف برای درهم تنیدن[۳۷] k دنباله‌ی ورودی در یک دنباله به‌طول n است.

ت) ثابت کنید که $\Omega(n \lg k)$ کران پایین هر الگوریتم برای مسئله‌ی فوق است.

۱۱.۶ n پردازنده در اختیار داریم که بعضی درست و بعضی خراب هستند. همچنین دستگاه آزمایش‌کننده‌ای داریم که نحوه‌ی استفاده از آن به این شکل است: دو پردازنده‌ی دل‌خواه A و B را در دستگاه می‌گذاریم. A و B هر یک دیگری را آزمایش می‌کند و گزارش می‌دهد که آیا دیگری درست است یا خراب. گزارش یک پردازنده‌ی درست حتماً صحیح است، ولی نمی‌توان به گزارش یک پردازنده‌ی خراب اعتماد کرد.

فرض کنید تعداد پردازنده‌های خراب کم‌تر از تعداد پردازنده‌های درست است.

الف) مسئله‌ی پیدا کردن یک پردازنده‌ی درست از بین n پردازنده را در نظر بگیرید. نشان دهید با $\lfloor \frac{n}{۲} \rfloor$ بار استفاده از دستگاه آزمایش‌کننده، می‌توان مسئله را به مسئله‌ای مشابه با تعداد پردازنده‌های تقریباً برابر $\frac{n}{۲}$ تبدیل کرد، به‌طوری‌که هم‌چنان تعداد پردازنده‌های خراب کم‌تر از تعداد پردازنده‌های درست باشد.

ب) با استفاده از قسمت قبل، نشان دهید که می‌توان تمام پردازنده‌های درست را با $\Theta(n)$ بار استفاده از دستگاه آزمایش‌کننده شناسایی کرد.

Stooge-Sort ۱۲.۶

پروفسور هاروارد و پروفسور فاین، الگوریتم STOOGE-SORT را پیش‌نهاد کرده‌اند.

```
STOOGE-SORT (A, i, j)
1  if A[i] > A[j]
2     then exchange A[i] ↔ A[j]
3  if  i + 1 ≥ j
```

[۳۷] interleave

```
4    then return
5  k ← ⌊((j−i+1)/3)⌋
6  STOOGE-SORT (A, i, j − k)
7  STOOGE-SORT (A, i + 1, j)
8  STOOGE-SORT (A, i, j − k)
```

الف) ثابت کنید اگر $n = length[A]$ باشد، آن‌گاه STOOGE-SORT آرایه‌ی ورودی را به‌درستی مرتب می‌کند.

ب) یک رابطه‌ی بازگشتی برای زمان اجرای بدترین حالت STOOGE-SORT ارائه دهید و جواب آن‌را با نماد Θ پیدا کنید.

پ) زمان اجرای بدترین حالت STOOGE-SORT را با مرتب‌سازی‌های درجی، ادغامی، هرمی و سریع مقایسه کنید.

⋆ ۱۳.۶ تحلیل دیگر از مرتب‌سازی سریع

این تحلیل به‌جای تأکید بر تعداد مقایسه‌های انجام‌شده، بر زمان اجرای مورد انتظار در هر فراخوانی بازگشتی رویه‌ی QUICKSORT تأکید دارد.

الف) ثابت کنید در هر آرایه به اندازه‌ی n، احتمال این‌که هر عنصر به‌عنوان میانه انتخاب شود $\frac{1}{n}$ است. از این موضوع برای تعریف معرف متغیرهای تصادفی

$$X_i = \{\text{امین کوچک‌ترین عنصر به‌عنوان محور انتخاب شود}\} \quad i$$

استفاده کنید. $E[X_i]$ چقدر است؟

ب) فرض کنید $T(n)$ یک متغیر تصادفی است که زمان اجرای رویه‌ی QUICKSORT را روی آرایه با اندازه‌ی n نشان می‌دهد. ثابت کنید:

$$E[T(n)] = E\left[\sum_{q=1}^{n} X_q \times (T(q-1) + T(n-q) + \Theta(n))\right] \qquad (۶\text{-}۲۰)$$

پ) نشان دهید که معادله‌ی ۶-۲۰ به صورت زیر ساده می‌شود:

$$E[T(n)] = \frac{2}{n}\sum_{q=0}^{n-1} E[T(q)] + \Theta(n) \qquad (۶\text{-}۲۱)$$

ت) نشان دهید

$$\sum_{k=1}^{n-1} k \lg k \le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 \qquad (۶\text{-}۲۲)$$

(راهنمایی: سری را به دو بخش تقسیم کنید: یکی برای $k = 1, 2, \cdots, \lceil \frac{n}{2} \rceil - 1$ و دیگری برای $k = \lceil \frac{n}{2} \rceil, \cdots, n$.)

ث) با استفاده از یک کران در معادله‌ی ۶-۲۱ نشان دهید که رابطه‌ی بازگشتی دارای جواب $E[T(n)] = \Theta(n \lg n)$ است. (راهنمایی: با جای‌گذاری نشان دهید که برای ثابت‌های a و b داریم: $E[T(n)] \leq an \lg n - bn$.)

۱۴.۶ عمق پشته برای مرتب‌سازی سریع

مرتب‌سازی سریع دو فراخوانی بازگشتی دارد. بعد از انجام بخش‌بندی، زیرآرایه‌ی چپ و سپس زیرآرایه‌ی راست به‌صورت بازگشتی مرتب می‌شوند. دومین فراخوانی بازگشتی در حقیقت لازم نیست و با استفاده از روش حذف آخرین بازگشت که در بخش ۴-۳-۳ گفته شد، می‌توان آن‌را به‌صورت زیر شبیه‌سازی کرد:

```
QUICK-SORT' (A, p, r)
1  while p < r
2      do ▷ Partition and sort left subarray.
3          q ← PARTITION (A, p, r)
4          QUICK-SORT'(A, p, q − 1)
5          p ← q + 1
```

الف) ثابت کنید که QUICK-SORT'$(A, ۱, length[A])$ به‌درستی آرایه‌ی A را مرتب می‌کند.

عمق پشته، بیش‌ترین مقدار فضای استفاده شده‌ی پشته در طول محاسبه است.

ب) سناریویی را توضیح دهید که در آن عمق پشته‌ی QUICK-SORT' برای آرایه به‌اندازه‌ی n، $\Theta(n)$ است.

پ) شبه‌کد QUICK-SORT' را طوری تغییر دهید که عمق پشته در بدترین حالت $\Theta(\lg n)$ شود. زمان اجرای مورد انتظار $\Theta(n \lg n)$ را برای الگوریتم حفظ کنید.

۱۵.۶ بخش‌بندی «میانه‌ی ۳ عنصر»

یک راه برای بهبود رویه‌ی بخش‌بندی در مرتب‌سازی سریع تصادفی این است که محور در بخش‌بندی با دقت بیش‌تری نسبت به انتخاب تصادفی یک عنصر انتخاب شود. یک روش معمول «میانه‌ی ۳ عنصر» است: محور را میانه‌ی سه عنصر که به‌صورت تصادفی از زیرآرایه انتخاب شده‌اند قرار می‌دهیم. برای این مسئله، فرض کنید عناصر آرایه‌ی ورودی $A[۱ \cdots n]$ متفاوت هستند و $n \geq ۳$. آرایه‌ی خروجی مرتب شده را با $A'[۱ \cdots n]$ نمایش می‌دهیم، محوری را که انتخاب می‌شود x می‌نامیم و تعریف می‌کنیم $p_i = Pr\{x = A'[i]\}$.

الف) یک فرمول دقیق برای p_i برحسب n و i برای $i = ۲, ۳, \cdots, n-۱$ ارائه دهید. (توجه کنید که $p_۱ = p_n = ۰$.)

ب) در مقایسه با پیاده‌سازی معمولی، شانس انتخاب $x = A'[\frac{n+۱}{۲}]$، میانه‌ی $A[۱ \cdots n]$، به‌عنوان محور چه‌قدر افزایش می‌یابد؟ با فرض $n \to \infty$، نسبت حدی این احتمال را بیابید.

پ) اگر «بخش‌بندی خوب» را این طور تعریف کنیم که محور $x = A'[i]$ باشد که در آن

$\frac{۲n}{۳} \le i \le \frac{n}{۳}$، احتمال تقسیم بندی خوب را، در مقایسه با پیاده‌سازی معمولی، چه میزان افزایش داده‌ایم؟ (راهنمایی: مجموع را به کمک انتگرال تخمین بزنید.)

ت) نشان دهید که با این روش زمان اجرای مرتب‌سازی هم‌چنان $\Omega(n \lg n)$ است و این روش بخش‌بندی فقط روی ضریب ثابت تأثیر می‌گذارد.

⋆ ۱۶.۶ مرتب‌سازی فازی بازه‌ها

حالتی از یک مسئله‌ی مرتب‌سازی را در نظر بگیرید که در آن اعداد دقیقاً مشخص نیستند. در عوض، برای هر عدد یک بازه روی محور اعداد حقیقی داریم که عدد به آن تعلق دارد. در واقع، n بازه‌ی بسته به شکل $[a_i, b_i]$ داریم که $a_i \le b_i$. هدف این است که این بازه‌ها را به صورت فازی مرتب کنیم. یعنی یک جای‌گشت $\langle i_۱, i_۲, \ldots, i_n \rangle$ از بازه‌ها تولید می‌کنیم، به‌طوری‌که به‌ازای هر $۱ \le i \le n$، $c_i \in [a_{ij}, b_{ij}]$ هایی وجود داشته باشند که در $c_۱ \le c_۲ \le \cdots \le c_n$ صدق کنند.

الف) یک الگوریتم برای مرتب‌سازی فازی n بازه طراحی کنید. الگوریتم شما باید ساختار کلی یک الگوریتم مرتب‌سازی سریع را داشته باشد که نقاط شروع (a_i ها) را مرتب کند، ولی از تداخل بازه‌ها برای زمان اجرای بهتر بهره ببرد. (هر چه تداخل بازه‌ها بیش‌تر باشد، مرتب‌سازی فازی بازه‌ها آسان‌تر می‌شود. الگوریتم شما باید تا آن‌جایی که ممکن است از تداخل بازه‌ها استفاده کند.)

ب) ثابت کنید که میانگین زمان مورد انتظار الگوریتم شما در حالت کلی $\Theta(n \lg n)$ است، ولی هنگامی که تمام بازه‌ها متداخل باشند، زمان اجرا، $\Theta(n)$ می‌شود. (یعنی هنگامی که مقدار x وجود داشته باشد که برای تمام i ها داشته باشیم $x \in [a_i, b_i]$)، الگوریتم شما نباید این مورد را مشخصاً بررسی کند، بلکه به‌طور طبیعی هنگامی که میزان تداخل‌ها بیش‌تر شود، الگوریتم شما باید کارایی بیش‌تر داشته باشد.

⋆ ۱۷.۶ میانگین عمق یک گره در د.د.ج

در این مسئله ثابت می‌کنیم که میانگین عمق یک گره در یک د.د.ج n رأسی که به‌صورت تصادفی ساخته شده باشد $\mathcal{O}(\lg n)$ است. اگر چه این نتیجه ضعیف‌تر از نتیجه‌ی قضیه‌ی ۴-۲ است، روشی که برای اثبات استفاده می‌کنیم، شباهت جالبی بین ساخت یک د.د.ج و اجرای الگوریتم مرتب‌سازی سریع تصادفی را آشکار می‌کند.

مجموع طول مسیرهای د.د.ج T را با $P(T)$ نشان می‌دهیم که برابر مجموع عمق همه‌ی گره‌های x عضو T، یا $d(T, x)$، است.

الف) نشان دهید میانگین عمق یک گره در T برابر است با $\frac{۱}{n} \sum_{x \in T} d(x, T) = \frac{۱}{n} P(T)$.

بنابراین قصد داریم نشان دهیم که امید ریاضی مقدار $P(T)$ برابر $\mathcal{O}(n \lg n)$ است.

ب) فرض کنید T_L و T_R به‌ترتیب معرف زیردرخت‌های چپ و راست T باشند. نشان دهید اگر T، n گره داشته باشد، آن‌گاه $P(T) = P(T_L) + P(T_R) + n - ۱$.

پ) فرض کنید $P(n)$ بیان‌گر میانگین مجموع طول مسیرهای تمام د.د.ج‌های ساخته شده با n گره باشد. نشان دهید:

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i+1) + n - 1).$$

ت) نشان دهید می‌توان $P(n)$ را به‌صورت زیر بازنویسی کرد:

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n).$$

ث) با یادآوری تحلیل دیگر نسخه‌ی تصادفی مرتب‌سازی سریع، نشان دهید $P(n) = O(n \lg n)$. در هر بار اجرای بازگشتی مرتب‌سازی سریع ما یک عنصر را به صورت تصادفی به‌عنوان محور انتخاب می‌کنیم تا مجموعه‌ی عناصری را که می‌خواهیم مرتب کنیم تقسیم کند. هر گره یک د.د.ج مجموعه‌ی گره‌هایی را که در زیردرخت آن گره قرار می‌گیرند تقسیم می‌کند.

ج) روشی برای پیاده‌سازی مرتب‌سازی سریع ارائه دهید که در آن مقایسه‌ها برای مرتب‌سازی مجموعه‌ای از عناصر دقیقاً همانند مقایسه‌های مربوط به درج عناصر درون یک د.د.ج باشند. (ترتیبی که در آن مقایسه‌ها انجام می‌شوند ممکن است متفاوت باشد اما مقایسه‌های مشابه‌ای باید انجام شود.)

⋆ ۱۸.۶ مرتب‌سازی مارپیچی ماتریس

یک ماتریس $n \times m$ از اعداد حقیقی را در نظر بگیرید. می‌خواهیم آن‌را از سطر اول به‌صورت مارپیچی مرتب کنیم. یعنی سطرهای با شماره‌ی فرد به‌صورت صعودی از چپ به راست، و سطرهای زوج به‌صورت صعودی ولی از راست به چپ مرتب باشند. نکته‌ی مهم این است که اگر از سطر اول همه‌ی سطرهای ماتریس را مطابق جهت‌های بیان‌شده به‌دنبال هم طی کنیم همه‌ی اعداد مرتب باشند.

الگوریتم زیر را برای مرتب‌سازی مارپیچی پیش‌نهاد می‌کنیم:

مراحل زیر را k بار تکرار کن:

۱. تک‌تکِ سطرها را مطابق جهت‌شان به‌صورت مستقل مرتب کن.

۲. تک‌تکِ ستون‌ها را از بالا به پایین مرتب کن.

الف) کم‌ترین مقدار k را پیدا کنید که در انتها الگوریتم فوق ماتریس را به‌درستی به‌صورت مارپیچی مرتب کند.

ب) درستی الگوریتم خود را اثبات و تحلیل کنید.

پروژه‌های برنامه‌نویسی فصل ۶

۱ مرتب‌سازی با لیست‌ها

برنامه‌ای بنویسید که N (بین ۱۰ تا ۱۰۰۰۰۰) عدد را که به‌صورت تصادفی تولید می‌کنید دریافت کند، آن‌ها را در یک لیست دوطرفه‌ی خطی با سرلیست قرار دهد و سپس با هریک از الگوریتم‌های زیر مرتب نماید. مرتب‌سازی تنها باید با تغییر اشاره‌گرها صورت گیرد.

۱. مرتب‌سازی ادغامی

۲. مرتب‌سازی حبابی

۳. مرتب‌سازی صدفی[۳۸]

الگوریتم اول در کتاب بیان شده است. الگوریتم دوم بر روی آرایه را می‌دانید و لازم است آن‌را به‌صورت مناسب تغییر دهید.

اگر الگوریتم مرتب‌سازی حبابی را فقط بر روی عناصری که به فاصله‌های k از هم هستند (یعنی عناصر اول، $k+1$ام، $2k+1$ام، $3k+1$ام و ...) انجام دهیم آن را k-sort می‌گوییم. مرتب‌سازی صدفی یعنی تکرار k-sort برای kهای $1, 4, 13, 40, 121, \ldots$ (از بزرگ به کوچک). این مقادیر در رابطه‌ی بازگشتی $h_{k-1} = 3h_k + 1$ و $h_t = 1$ برای $t = \lfloor \log_3 n \rfloor - 1$ صدق می‌کنند.

شما باید رده‌ی List را با اعمال مورد نیاز به‌طور کامل تعریف کنید و فقط از آن‌ها استفاده نمایید.

هم‌چنین باید برنامه‌ی خود را ۱۰۰۰ بار تکرار کنید و هر بار تعداد مقایسه‌ی کلیدها با هم را در هر الگوریتم (که همان هزینه‌ی اجرای آن الگوریتم است) بشمارید و برای هر الگوریتم منحنی زمان اجرا برای Nهای مختلف را رسم کنید. اگر رسم به‌وسیله‌ی برنامه‌ی شما انجام شود نمره‌ی اضافه خواهید گرفت. منحنی را دستی هم می‌توانید رسم کنید، ولی در هر حال هزینه‌های اجرای سه الگوریتم فوق در اندازه‌های مختلف باید در خروجی چاپ شود.

۲ پیاده‌سازی الگوریتم فورد ـ جانسون

برنامه‌ای بنویسید که اعداد $a_1, a_2, \ldots, a_n$ را با الگوریتم فورد ـ جانسون مرتب کند. هدف این الگوریتم، مرتب‌سازی با تعداد مقایسه‌هایی نزدیک به مقدار بهینه ($\lceil \log n! \rceil$) است.

---

[۳۸] shellsort

برای راحتی کار، زمان اجرای کل الگوریتم می‌تواند $\mathcal{O}(n^۲)$ باشد ولی تعداد مقایسه‌ها باید کم باشد. برای بررسی درستی برنامه، آرایه‌ای که باید مرتب شود به برنامه‌ی شما داده نمی‌شود. بلکه، برنامه‌ی شما با برنامه‌ی آزمون‌کننده به مکالمه خواهد پرداخت. به این شکل که خروجی برنامه‌ی شما، ورودی برنامه‌ی آزمون‌کننده، و ورودی آن، خروجی برنامه‌ی آزمون‌کننده است.

برنامه‌ی شما بعد از گرفتن n از ورودی، باید سؤالات خود را که در حقیقت مقایسه‌ها هستند در خروجی بنویسد و پاسخ آن‌ها را از ورودی بخواند. در پایان نیز جای‌گشتی از اعداد $۱, \cdots, n$ را در خروجی بنویسد، به‌طوری‌که اگر روی آرایه‌ی اولیه اِعمال شود آن را مرتب کند.

مکالمه:

۱. از ورودی استاندارد بخوانید و در خروجی استاندارد بنویسید.
۲. در ابتدا تنها n را (در یک سطر) از ورودی بخوانید.
۳. برای مقایسه‌ی «$a_i < a_j$?» (آیا a_i از a_j کم‌تر است؟) تنها در یک سطر i و j را به ترتیب بنویسید و پاسخ آن را (در یک سطر) از ورودی بخوانید. پاسخ می‌تواند 0 یا 1 باشد. اگر $a_i < a_j$ بود، پاسخ 1، و در غیر این صورت 0 خواهد بود.
۴. در پایان، جای‌گشت جواب را در یک سطر به این شکل بنویسید: ابتدا عدد 1-، و سپس، n عددِ $\pi_۱, \pi_۲, \ldots, \pi_n$ با یک فاصله از هم. π_i ها باید به‌گونه‌ای باشند که

$$a_{\pi_۱} < a_{\pi_۲} < \ldots < a_{\pi_n}.$$

نکته‌ی مهم در نوشتن خروجی: در نوشتن خروجی به این نکته حتماً توجه کنید که پس از نوشتن هر پرسش، خروجی را فلاش کنید. در غیر این صورت، ممکن است خروجی شما در میان‌گیر خروجی‌تان باقی بماند و عملاً به برنامه‌ی مقابل نرسد، که در این صورت، اگر منتظر خواندن پاسخ شوید هیچ‌گاه پاسخی نخواهید گرفت و در نتیجه برنامه‌تان از ادامه‌ی اجرا خواهد ایستاد تا این‌که زمان مجاز اجرای آن به پایان برسد.

برای مشاهده‌ی چگونگی فلاش‌کردن خروجی به جدول زیر مراجعه کنید:

C	`printf("%d %d\n", i, j);` `fflush(stdout);`

C++	`cout << i << " " << j << endl;`
	`------- or --------`
	`cout << i << " " << j << "\n" << flush;`

Pascal	`WriteLn(i, ' ', j);` `Flush(Output);`

Java	`System.out.println(`
	`  Integer.toString(i)+" "+Integer.toString(j));`
	`System.out.flush();`

محدودیت:

- هر گونه زیباسازی خروجی‌ها (از جمله در هنگام پرسش) مثلاً با نوشتن توضیحات اضافه یا قرار دادن «,» منجر به نادرست شمرده شدن خروجی می‌شود.
- برنامه‌ی خود را کاملاً استاندارد بنویسید چون برنامه‌ی شما در محیط shell لینوکس ترجمه و اجرا خواهد شد.
- به‌برنامه‌ی شما ۵ ثانیه وقت برای اجرا داده می‌شود.
- $n \leq ۵۰۰۰$ و همه‌ی a_i ها متفاوت‌اند.
- با توجه به این‌که حداکثر تعداد مقایسه‌ها (پرسش‌ها)ی لازم برای مرتب‌سازی n عنصر در الگوریتم فورد–جانسون مشخص است، تنها در صورتی نمره‌ی هر آزمون به برنامه تعلق می‌گیرد که تعداد پرسش‌ها (مقایسه‌ها)ی انجام شده از حد مجاز بیش‌تر نباشد. همچنین، اگر در پایان با پرسش‌هایی که انجام‌شده و پاسخ‌هایی که داده‌شده بتوان بیش از یک جای‌گشت جواب داشت، مشخص است که مرتب‌سازی درست انجام نشده است، در این حالت هم برنامه نمره‌ی آزمون مورد نظر را نخواهد گرفت.

ورودی و خروجی نمونه

مکالمه‌ی مربوط به دنباله‌ی $\{a_i\}$=(30, 1, 28, 5, 19, 10) را در زیر می‌بینید.

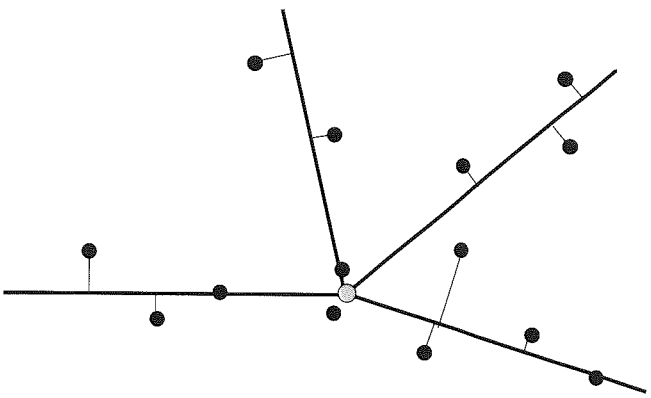
input	output
6	
	1 2
0	
	3 4
0	
	5 6
0	
	2 4
1	
	6 2
0	
	6 4
0	
	1 6
0	
	1 5
0	
	3 5
0	
	3 1
1	
	-1 2 4 6 5 3 1

راهنمایی:

- برنامه را می‌توانید با زمان اجرای $\mathcal{O}(n^۲)$ بنویسید.
- برای راحتی کار، ضروری نیست که از پیاده‌سازی الگوریتم با درخت دودویی استفاده کنید.
- تنها داده‌ساختار لازم برای پیاده‌سازی، آرایه است! کافی است عناصر زنجیره‌ی اصلی در یک آرایه باشند.
- به‌راحتی می‌توان عناصر دیگر را با جست‌وجوی دودویی در این زنجیره درج کرد.

۳ اتصال به چاه‌های نفت

مختصات n چاه نفت داده شده‌اند. می‌خواهیم با کشیدن حداقل تعداد لوله‌ها، نفت‌های این چاه‌ها را به یک پالایشگاه که در مرکز مختصات قرار دارد منتقل کنیم. می‌خواهیم این لوله‌ها طوری کشیده شوند که فاصله‌ی هر چاه تا نزدیک‌ترین لوله از d بیش‌تر نشود.



ورودی: سطر اول تعداد ورودی‌های مسئله است. سطر اول هر ورودی به‌ترتیب دو عدد n و d نوشته شده‌اند. ($۰ \le d \le ۱۵۰$ و $۱ \le n \le ۵۰۰$). در n سطر بعد، مختصات چاه‌ها بدون ترتیب خاصی قرار دارند. هر چاه با دو عدد x و y که اولی مختصات x و دومی مختصات y آن چاه است نشان داده می‌شود. فرض کنید

$$-۱۰۰ \le x, y \le ۱۰۰.$$

خروجی: برای هر ورودی یک عدد در یک سطر نوشته می‌شود که برابر با حداقل تعداد خطوط لوله‌ی مورد نیاز است.

مثال:

Standard Input	Standard Output
2 7 1 1 4 3 1 3 1 2 3 2 4 2 2 6 2 4 0 0 4 12 18 0 27 34 51	4 2 15

۴ زیرمجموعه‌ی مثلثی

مجموعه‌ی A از اعداد طبیعی داده شده است. زیرمجموعه‌ی $T \subseteq A$ را «مثلثی» می‌گوییم اگر برای هر سه عنصر متفاوت (ولی نه لزوماً نامساوی) $x, y, z \in T$ رابطه‌ی زیر برقرار باشد:

$$x + y > z,\ x + z > y,\ \text{and}\ y + z > x$$

برنامه‌ای بنویسید تا با دریافت اعداد موجود در A، بزرگ‌ترین زیرمجموعه‌ی مثلثی A (یعنی زیرمجموعه‌ای با بیش‌ترین تعداد عضو) را به‌دست آورد. (مجموعه A و زیر مجموعه‌هایش می‌توانند عضو تکراری داشته باشند.)

ورودی: در سطر اول n، و در n سطر بعد اعداد موجود در A به این صورت آمده است: در سطر iام سه عدد $s_i\ t_i\ k_i$ آمده، به این معنی که همه‌ی اعداد $s_i + j \times t_i$ $(0 \le j < k)$ عضو A هستند. تمام اعداد ورودی صحیح می‌باشند.

خروجی: در یک سطر دو عدد نوشته شود. عدد اول M تعداد عناصر T که بزرگ‌ترین زیرمجموعه‌ی مثلثی A است و عدد دوم مجموع عناصر موجود در T. اگر بیش از یک زیرمجموعه‌ی مثلثی با M عنصر وجود داشت عدد مربوط به مجموعه‌ای را که جمع عناصرش کم‌ترین است بنویسد.

محدودیت: $1 \le n \le 100$ و $1 \le \Sigma k_i \le 1000000$ (همه‌ی اعضای A کوچکتر از 10^9 می‌باشند.)

مثال:

Standard Input	Standard Output
3 1 2 10 4 2 6 3 5 3	10 105

فصل

۷

داده‌ساختارهای پیشرفته

در این فصل داده‌ساختارهای پیشرفته را بررسی می‌کنیم. ابتدا در بخش ۷-۱ داده‌ساختاری به‌نام «مجموعه‌های مجزا» را معرفی خواهیم کرد که بر روی تعدادی مجموعه‌ی مجزا از هم، هر یک از دو عمل «یافتن مجموعه‌ی حاوی یک عنصر» و «ادغام دو مجموعه» را که بهترین پیاده‌سازی آن در زمانی نزدیک به $O(1)$ است انجام می‌دهد. این داده‌ساختار در کاربردهای مختلفی استفاده می‌شود. در بخش ۷-۲، «درخت دودویی جست‌وجوی بهینه» را بررسی می‌کنیم که گونه‌ی جالبی از درخت عادی دودویی جست‌وجو است که برای ذخیره‌ی یک فرهنگ داده‌ای یا جدول نمادها، عمدتاً به‌منظور جست‌وجو در آن استفاده می‌شود. در این داده‌ساختار، میانگین زمان جست‌وجوی موفق و ناموفق کمینه است.

در بخش ۷-۳، د.د.ج را گسترش می‌دهیم تا ارتفاع آن همیشه لگاریتمی بماند. در بخش ۷-۳-۱ درخت قرمز-سیاه را توضیح می‌دهیم که با اضافه کردن یک بیتِ «رنگ» (قرمز یا سیاه) به هر گره، کاری می‌کند که همیشه ارتفاع درخت حداکثر حدود $2\lg n$ شود. درخت مرتبه‌ی آماری که در بخش ۷-۳-۲ گفته می‌شود، گسترش داده‌شده‌ی درخت قرمز-سیاه است که (علاوه بر دیگر اعمال فرهنگ داده‌ای)، اعمال «یافتن عنصری با مرتبه‌ی داده‌شده» و نیز «یافتن مرتبه‌ی یک عنصر» را در زمان لگاریتمی انجام می‌دهد. درخت بازه هم که در بخش ۷-۳-۳ مورد بررسی قرار می‌گیرد گونه‌ی گسترش‌یافته‌ی دیگری از درخت قرمز-سیاه است که با بازه‌ها کار می‌کند و اعمال فرهنگ داده‌ای بر روی مجموعه‌ای

از بازه‌ها و نیز عمل «یافتن بازه‌ی هم‌پوشان با یک بازه‌ی داده‌شده» را در زمان لگاریتمی انجام می‌دهد.

بخش ۷-۳-۴ درخت ای.وی.ال را معرفی می‌کند که به‌صورت متفاوتی تضمین می‌کند که ارتفاع د.د.ج همیشه لگاریتمی باشد.

بخش‌های ۷-۴ و ۷-۵ چند درخت کاملاً متوازن و با ارتفاع لگاریتمی را تشریح می‌کند. درخت ۳-۲ که ساده‌تر است ابتدا گفته می‌شود. سپس درخت «بی» که گونه‌ی عمومی‌تری از درخت ۳-۲ است ارائه می‌شود. درخت «بی» عمدتاً برای ذخیره‌ی داده‌ها بر روی حافظه‌ی خارجی مورد استفاده قرار می‌گیرد و نمونه‌ی خوبی از داده‌ساختارهای خارجی است.

۷-۱ مجموعه‌های مجزا

«مجموعه‌های مجزا[۱]» یکی از داده‌گونه‌های انتزاعی مفید و جالب برای ذخیره‌ی تعدادی مجموعه‌ی مجزا از عناصر است که بر روی آن اعمال «یافتن یک عنصر[۲]» و «ادغام دو مجموعه» انجام می‌شود. مجموعه‌های مجزا در زبان انگلیسی به‌نام‌های دیگری هم‌چون UNION-FIND، FIND-MERGE یا DISJOINT-FIND-MERGE مشهور است.

به‌طور دقیق‌تر، فرض کنید $U = \{1, 2, \ldots, n\}$ مجموعه‌ی عناصر و S_1، S_2، تا S_k $(k \leq n)$ زیرمجموعه‌های مجزایی از U باشند $(S_i \cap S_j = i \neq j)$ که $\cup_{i=1}^{k} S_i = U$ (یا S_i ها U را افراز می‌کنند). اعمال زیر را بر روی این مجموعه‌ها تعریف می‌کنیم:

ایجاد مجموعه CREATE(i): $S_i = \{i\}$ را ایجاد می‌کند.

یافتن عنصر FIND(i): مجموعه‌ی S_j را پیدا می‌کند که عنصر i عضو آن است، یعنی شماره‌ی j ای را برمی‌گرداند که $i \in S_j$.

ادغام دو مجموعه MERGE(i, j): دو مجموعه‌ی S_i و S_j را ادغام می‌کند، یعنی $S_i \cup S_j$ را جای‌گزین S_i یا S_j می‌کند و دیگری حذف می‌شود.

هدف از این بخش بررسی داده‌ساختارهای مناسب برای پیاده‌سازی این داده‌گونه‌ی انتزاعی است، به‌طوری که اعمال فوق را بتوان در زمان کوتاهی انجام داد.

---

[۱] disjoint sets
[۲] find

یکی از مهم‌ترین کاربردهای این داده‌گونه، استفاده از آن در یافتنِ «رده‌های هم‌ارزی[۳]» در رابطه‌های هم‌ارزی است که در مسئله‌های زیادی کاربرد دارد. مثلاً، یافتن اجزای هم‌بند در یک گراف از این نوع مسئله‌هاست. در این حالت دو رأس v و u از گرافِ بدون جهت با هم رابطه‌ی «هم‌بندی[۴]» R دارند، اگر این دو رأس به‌هم وصل باشند (یعنی مسیری از یکی به دیگری در گراف باشد). روشن است که این یک رابطه‌ی هم‌ارزی است و بر اساس آن عناصر این رابطه (یعنی رأس‌ها) به رده‌های هم‌ارزی افراز می‌شوند که هر یک را یک «جزء هم‌بند[۵]» گراف می‌گوییم. ما با استفاده از الگوریتم کلی‌ای که در زیر بیان می‌کنیم می‌توانیم این اجزا را به‌دست آوریم.

به عنوان مثال دیگر، گراف جهت‌دار G را در نظر بگیرید. می‌گوییم رأس‌های u و v با هم رابطه‌ی R دارند (یعنی uRv و vRu)، اگر این دو رأس در یک دور جهت‌دار از گراف G ظاهر شوند. بدیهی است که R یک رابطه‌ی هم‌ارزی است، که در نتیجه رأس‌های گراف را به مجموعه‌های متمایزِ هم‌ارزی افراز می‌کند که طبق تعریف، به هر یک از زیرمجموعه‌ها «جزء قویاً هم‌بند[۶]» می‌گوییم.

مثال سوم، رابطه‌ی هم‌ارزی R بین یال‌های یک گراف بدون جهت G تعریف شده است. دو یال e_1 و e_2 با هم رابطه‌ی R دارند اگر این دو یال در یک دور در گراف واقع شده باشند. با فرض آن‌که طبق تعریف، e با خودش این رابطه را دارد، بدیهی است که R یک رابطه‌ی هم‌ارزی است و در نتیجه یال‌های گراف را به رده‌های هم‌ارزی افراز می‌کند. به هر یک از این رده‌ها یک «جزءِ دو هم‌بند[۷]» گراف G می‌گوییم.

روش کلی برای محاسبه‌ی رده‌های هم‌ارزیِ یک رابطه‌ی هم‌ارزی R بر روی عناصر $U = \{1, 2, \ldots, n\}$ به‌صورت زیر است:

۱. به‌ازای هر عنصر $i \in U$ مجموعه‌ی $S_i = \{i\}$ را ایجاد کن.

۲. برای هر عضو $(b, c) \in R$

الف) b را پیدا کن، فرض کن $b \in S_i$ $(i \leftarrow \text{FIND}(b))$

ب) c را پیدا کن، فرض کن $c \in S_j$ $(j \leftarrow \text{FIND}(c))$

پ) اگر $i \neq j$ S_i و S_j را در هم ادغام کن $(\text{MERGE}(i, j))$.

---

[۳] equivalence classes
[۴] connectivity
[۵] connected component
[۶] strongly connected component
[۷] biconnected component

۳. در انتها، هر مجموعه‌ی باقی‌مانده یک رده‌ی هم‌ارزی برای R است.

بدیهی است که در الگوریتم فوق، دقیقاً n بار عمل ایجاد مجموعه، $2|R|$ بار عمل یافتن یک عنصر و حداکثر $n-1$ بار عمل ادغام انجام می‌شود (چرا که هر بار ادغام از تعداد مجموعه‌های موجود یک عدد کم می‌کند).

ساده‌ترین پیاده‌سازی برای این داده‌گونه می‌تواند هر یک از اعمال یافتن و ادغام را در $\mathcal{O}(n)$ انجام دهد که بسیار کند است. ما در این بخش چند داده‌ساختار بسیار کاراتر پیشنهاد می‌کنیم. در اولین پیاده‌سازی که مبتنی بر لیست‌های پیوندی است، عمل یافتن در $\mathcal{O}(1)$ و n عمل ادغام در مجموع در $\mathcal{O}(n \lg n)$ انجام می‌شود (یعنی هزینه‌ی سرشکن‌شده‌ی هر ادغام $\mathcal{O}(\lg n)$ است).

دومین پیاده‌سازی مبتنی بر درخت است که در آن هر عمل ادغام در $\mathcal{O}(1)$ و هر عمل پیدا کردن حداکثر در $\mathcal{O}(\lg n)$ انجام می‌شود. می‌توان با انجام یک تغییر در این پیاده‌سازی کاری کرد که m عمل ادغام در زمان $\mathcal{O}(m\,\alpha(m,n))$ انجام شود. $\alpha(m,n)$ معکوس تابع «اَکِرمَن[۸]» است که بسیار کند رشد می‌کند، تا جایی که برای مقادیر بسیار بزرگ n، $\alpha(n,n) \leq ۴$. تابع اکرمن و معکوس آن در بخش ۷-۱-۳ تعریف می‌شوند.

۷-۱-۱ داده‌ساختار مبتنی بر لیست

در این پیاده‌سازی، هر مجموعه یک لیست پیوندی از اعضای آن مجموعه است. اگر اشاره‌گری به‌انتهای هر لیست نگه داریم، ادغام فیزیکی دو مجموعه را می‌توان به‌سادگی در $\mathcal{O}(1)$ و با تغییر چند اشاره‌گر انجام داد. اگر به هر عنصر i دست‌رسی مستقیم داشته باشیم، می‌توانیم با ذخیره‌ی شماره‌ی مجموعه‌ی j که $i \in S_j$ در عنصر i، عمل یافتن را نیز در $\mathcal{O}(1)$ انجام دهیم. مشکل این‌جاست که به‌هنگام ادغام دو مجموعه باید این شماره را برای تک‌تک عناصر یک مجموعه به شماره‌ی مجموعه‌ی حاصل (همان مجموعه‌ی دوم) تغییر داد. این کار ممکن است حداکثر از مرتبه‌ی تعداد عناصر یکی از مجموعه‌های درگیر در ادغام و یا از $\mathcal{O}(n)$ باشد.

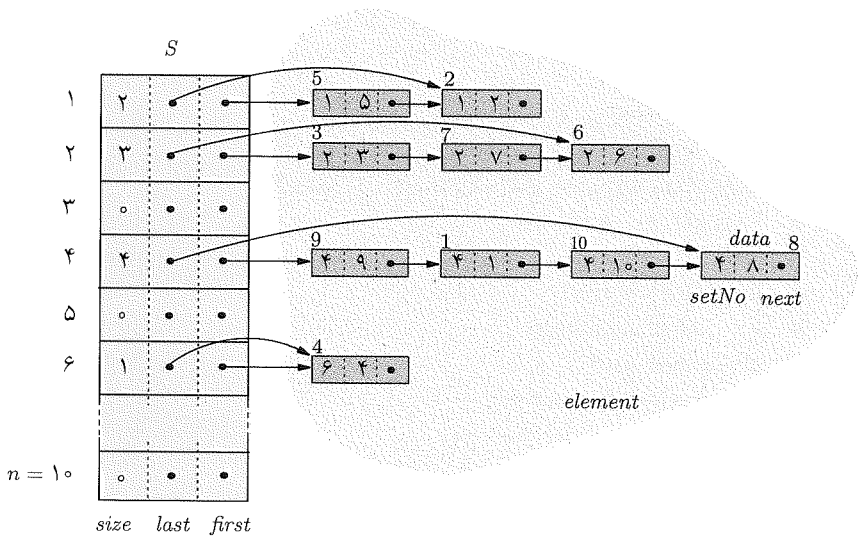
این مقدار هزینه برای یک عمل ادغام قابل قبول نیست. اما اگر با کمی دقت در ادغام دو مجموعه‌ی S_i و S_j، شماره‌ی مجموعه‌ی حاصل را آن مجموعه‌ای بگیریم (مثلاً i) که تعداد عناصرش بیش‌تر است (یعنی $|S_i| \geq |S_j|$)، هزینه‌ی n عدد ادغام برابر $\mathcal{O}(n \lg n)$

[۸] Ackermann function

خواهد شد. به‌عبارت دیگر هزینه‌ی سرشکن‌شده‌ی هر ادغام $O(\lg n)$ خواهد شد.

جزئیات پیاده‌سازی

با فرض این‌که تعداد عناصر و حداکثر تعداد مجموعه‌ها n است، برای پیاده‌سازی این داده‌گونه از اشاره‌گرهای اندیسی استفاده می‌کنیم. یک آرایه‌ی S به‌طول n که هر درایه‌ی آن، مثلاً $S[i]$ شامل مؤلفه‌های $size[S[i]]$، $first[S[i]]$ و $last[S[i]]$ است که به‌ترتیب، تعداد عناصر، اولین عنصر و آخرین عنصر لیست متناظر با S_i را ذخیره می‌کند. آرایه‌ی $element$ به‌اندازه‌ی n هم محل ذخیره‌ی کل عناصر است که آن‌ها را با اشاره‌گرهای اندیسی‌ای که دارند به‌صورت لیست در می‌آورد. $element[i]$ عنصر iام است و اگر عضوی از لیست $S[j]$ باشد ($i \in S_j$)، مؤلفه‌ی $setNo[element[i]]$ برابر j خواهد بود. مؤلفه‌ی $next[element[i]]$ اشاره‌گر اندیسی به عنصر بعدی این عنصر است که در لیست $S[j]$ قرار دارد، یا مقدار **null** را دارد، اگر عنصر آخر باشد. شکل ۷-۱ مثالی از این پیاده‌سازی را که شامل ۱۰ عنصر است نشان می‌دهد.

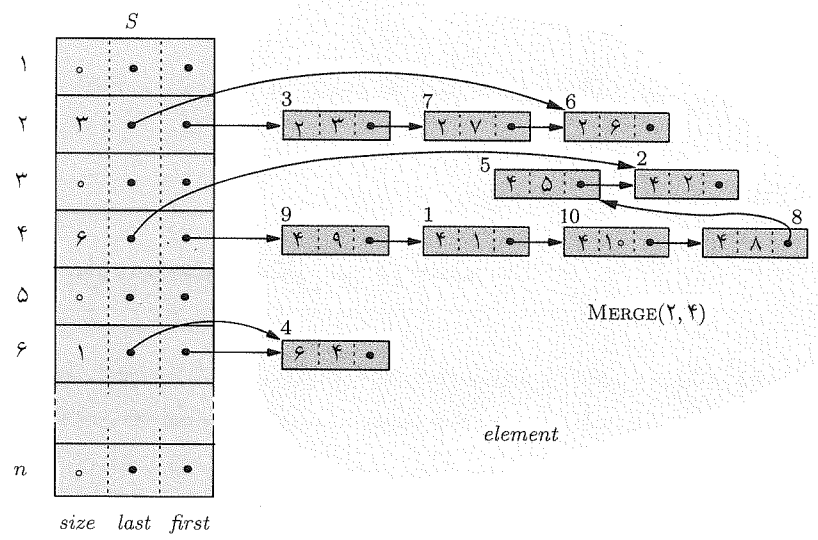


شکل ۷-۱ مثالی از پیاده‌سازی مبتنی بر لیست شامل ۱۰ عنصر و مجموعه‌های $S_1 = \{5,2\}$، $S_2 = \{3,7,6\}$، $S_4 = \{9,1,10,8\}$ و $S_6 = \{4\}$. عناصر ۱ تا ۱۰ در آرایه‌ی $element$ قرار دارند. در این شکل، شماره‌ی هر درایه در کنار آن عنصر آمده است.

رویه‌های زیر مقداردهی اولیه و نیز نحوه‌ی انجام دو عمل FIND و MERGE را بیان می‌کنند.

```
INITIALIZE ()
1  for i ← 1 to lenght[element]
2    do first[S[i]] ← i
3       last[S[i]] ← i
4       size[S[i]] ← 1
5       setNo[element[i]] ← i
6       next[element[i]] ← null

FIND (x)
    ▷ finds the set number that x in an element of
1  return setNo[element[x]]

MERGE (i,j)
    ▷ S_k ← S_i ∪ S_j, k is either i or j
1  if size[S[i]] > size[S[j]]
2    then k ← i; r ← j
3    else k ← j; r ← i
4  p ← last[S[k]]
5  next[element[p]] ← first[S[r]]
6  last[S[k]] ← last[S[r]]
7  last[S[r]] ← null
8  first[S[r]] ← null
9  p ← next[element[p]]
10 while p ≠ null
11    do setNo[element[p]] ← k
12       p ← next[element[p]]
```

شکل ۷-۲ مثال شکل ۷-۱ را پس از انجام عمل MERGE(۱, ۴) نشان می‌دهد که در آن مجموعه‌ی S_1 در S_4 ادغام شده و حاصل آن مجموعه‌ی جدید $S_4 = \{9, 1, 10, 8, 5, 2\}$ خواهد بود. مقدار هزینه‌ی این ادغام (علاوه‌بر هزینه‌های ثابت ادغام فیزیکی لیست‌ها) متناسب با ۲ است.



شکل ۷-۲ همان داده‌ساختار شکل ۷-۱ پس از انجام عمل MERGE(۱, ۴) که مجموعه‌های $S_۲ = \{۳, ۷, ۶\}$، $S_۴ = \{۹, ۱, ۱۰, ۸, ۵, ۲\}$ و $S_۶ = \{۴\}$ حاصل می‌شوند.

لم ۷-۱ هزینه‌ی n عمل ادغام در داده‌ساختار مبتنی بر لیست برابر $\mathcal{O}(n \lg n)$ است.

اثبات: هزینه‌ی یک عمل ادغام $S_۱$ با $S_۲$، متناسب است با تعداد تغییر شماره‌ی مجموعه‌ها در آن عمل. پس اگر بتوانیم تعداد کل تغییر شماره‌ی مجموعه‌های حاصل از عمل ادغام را بشماریم، هزینه‌ی کل ادغام‌ها به‌دست آورده‌ایم.

برای این‌کار، حداکثرِ تعداد دفعاتی را به‌دست می‌آوریم که ممکن است شماره‌ی مجموعه‌ی یک عنصر دل‌خواه به‌نام i تغییر کند. با توجه به این‌که همواره شماره‌ی عناصر مجموعه‌ی با تعداد عنصر کم‌تر، مثلاً $S_۱$ را تغییر می‌دهیم، روشن است که تعداد عناصر مجموعه‌ی $S_۲$ که حاصل می‌شود، دست‌کم دو برابر تعداد عناصر $S_۱$ است. با توجه به این نکته در می‌یابیم که حداکثر تعداد دفعاتی که شماره‌ی مجموعه‌ی یک عنصر i تغییر می‌کند نمی‌تواند از $\lg n$ بار بیش‌تر باشد، چرا که در نهایت یک مجموعه با حداکثر تعداد عناصر n ایجاد می‌شود. بنابراین، با n عنصر، هزینه‌ی کل تغییر شماره‌های مجموعه‌ها در مجموع از $n \lg n$ بیش‌تر نیست. □

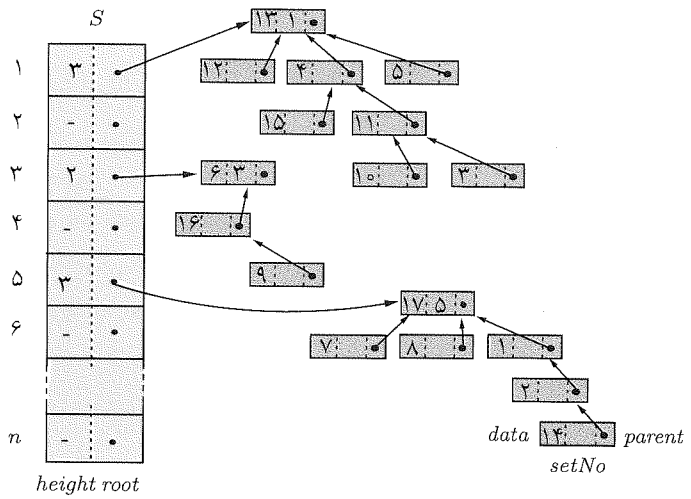
نتیجه‌ی ۷-۶ هزینه‌ی سرشکن‌شده‌ی هر عمل ادغام $\mathcal{O}(\lg n)$ است.

۷-۱-۲ داده‌ساختار مبتنی بر درخت

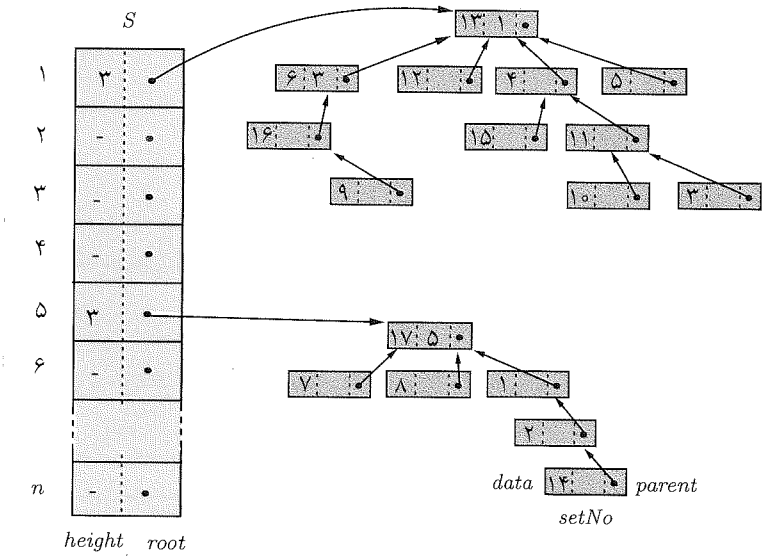
در این پیاده‌سازی، مانند قبل، آرایه‌ی *elements* عناصر را ذخیره می‌کند که مؤلفه‌های عنصر i، *setNo*، *parent* و عدد i است. مجموعه‌ها هم در آرایه‌ی S ذخیره می‌شوند. هر مجموعه‌ی S_j به‌صورت یک درخت عمومی پیاده‌سازی شده، به‌طوری‌که اگر $i \in S_j$ باشد، $element[i]$ یک گره از آن درخت است. در این درخت محدودیتی در تعداد فرزندان هر گره نیست. بنابراین با اشاره‌گر *parent* هر گره به پدرش وصل می‌شود. مجموعه‌ی S_i در درایه‌ی $S[i]$ ذخیره می‌شود که شامل مؤلفه‌های $rootS[i]$ و $height[S[i]]$ است که به‌ترتیب شماره‌ی عنصری که ریشه‌ی درخت S_i است و هم‌چنین ارتفاع آن درخت را ذخیره می‌کند. (شکل ۷-۳ را ببینید.)

در این صورت، برای انجام عمل ادغام S_i با S_j، کافی است ریشه‌ی یکی از این دو درخت، مثلاً S_i (یا $root[S[i]]$) را فرزند ریشه‌ی درخت دوم، یعنی S_j قرار داد و اطلاعات دیگر درخت حاصل را اصلاح کرد. روشن است که زمان انجام این کار $\mathcal{O}(1)$ است. شکل ۷-۴ نتیجه‌ی ادغام مجموعه‌های S_1 و S_3 از شکل ۷-۳ را نشان می‌دهد.

برای انجام عمل $\textsc{Find}(i)$، کافی است از عنصر $element[i]$ شروع و مؤلفه‌های *parent* آن‌را دنبال کنیم تا به ریشه‌ی یک درخت، مثلاً S_j برسیم (عنصری که مؤلفه‌ی *parent* آن **null** است). کافی است که مقدار مؤلفه‌ی *setNo* ریشه به‌درستی برابر j باشد، که در آن‌صورت j جواب عمل یافتن i است. توجه کنید که لزومی به روزآمد بودن مؤلفه‌های *setNo* برای بقیه‌ی عناصر نیست. به‌این ترتیب، روشن است که عمل یافتن یک عنصر از مرتبه‌ی ارتفاع درختی است که آن عنصر، عضو آن است. اگر در زمان انجام ادغام دقت نکنیم، ممکن است ارتفاع این درخت‌ها در بدترین حالت $\mathcal{O}(n)$ شود. در حالی‌که، مانند قبل، اگر در ادغام S_i با S_j، درختی که ارتفاع‌اش کم‌تر است، مثلاً S_i را زیر درخت درخت دوم (S_j) قرار دهیم، می‌توان ثابت کرد که ارتفاع درخت‌ها حداکثر $\lg n$ می‌شود و لذا عمل یافتن از $\mathcal{O}(\lg n)$ خواهد بود. به‌همین دلیل، مؤلفه‌ی *height* هر درخت هم ذخیره و هنگام انجام عمل ادغام اصلاح می‌شود.



شکل ۷-۳ پیاده‌سازی مبتنی بر درخت برای مجموعه‌های $S_1 = \{13,12,4,5,15,11,10,3\}$، $S_3 = \{6,16,9\}$ و $S_5 = \{17,7,8,1,2,14\}$.



شکل ۷-۴ ادغام مجموعه‌های S_1 و S_3 از شکل ۷-۳.

رویه‌های زیر الگوریتم‌های این اعمال را نشان می‌دهد.

```
INITIALIZE ()
1  for i ← 1 to lenght[element]
2     do height[S[i]] ← 0
3        root[S[i]] ← i
4        setNo[element[i]] ← i
5        parent[element[i]] ← null

FIND (x)
     ▷ finds the set number that x is an element of
1  p ← i while parent[element[p]] ≠ null
2     do p ← parent[element[p]]
3  return setNo[element[p]]

MERGE (i,j)
     ▷ S_x ← S_i ∪ S_j, x is either i or j
1  if height[S[i]] < height[S[j]]
2     then k ← i; r ← j
3     else k ← j; r ← i
4  parent[element[root[S[k]]]] ← root[S[r]]
5  if height[S[k]] = height[S[r]]
6     then height[S[r]] ← height[S[r]] + 1
7  root[S[k]] ← null
```

لم ۷-۲ هزینه‌ی هر عمل یافتن در پیاده‌سازی مبتنی بر درخت $\mathcal{O}(\lg n)$ است.

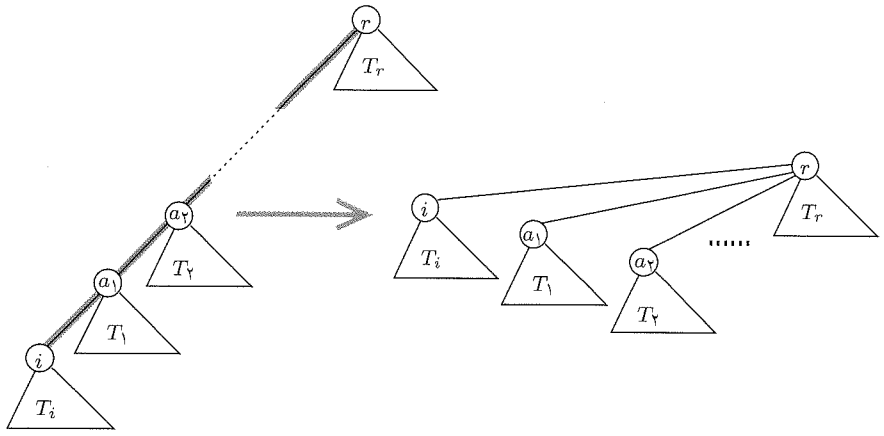
اثبات: با استقرا ثابت می‌کنیم که در درختی به ارتفاع h که با این الگوریتم ساخته می‌شود دست‌کم ۲^h عنصر وجود دارد. پایه‌ی استقرا بدیهی است. فرض کنید دو درخت به‌ارتفاع‌های $h_۱ \leq h_۲$ با هم ادغام شوند. اگر $h_۱ < h_۲$ باشد، ارتفاع درخت حاصل همان $h_۲$ است و با فرض استقرا، تعداد عناصر درخت حاصل دست‌کم $۲^{h_۲} + ۲^{h_۱}$ می‌شود و در

نتیجه از $۲^{h_۲}$ کم‌تر نیست. اگر $h_۱ = h_۲$ باشد، ارتفاع درخت حاصل $h_۲ + ۱$ خواهد بود. در این صورت نیز تعداد عناصر درخت حاصل از $۲^{h_۱+۱} = ۲^{h_۱} + ۲^{h_۲}$ کم‌تر نیست و حکم ثابت می‌شود.

با استفاده از این مطلب نتیجه می‌شود که در این داده‌ساختار، حداکثر ارتفاع یک درخت با m عنصر برابر $\lfloor \lg m \rfloor$ است. □

۷-۱-۳ پیاده‌سازی با «فشرده‌سازی مسیر»

در پیاده‌سازی با درخت، اگر در هر عمل یافتن عنصر i تغییراتی به شرح زیر در درختی که i عضو آن است بدهیم، می‌توان پیاده‌سازی را به‌صورت چشم‌گیری کاراتر کرد. فرض کنید که مطابق شکل ۷-۵، مسیر i به ریشه‌ی درخت r از گره‌های $a_۱$ تا a_k می‌گذرد. $a_۱$ تا a_k را (با تغییر مؤلفه‌ی *parent* آن‌ها) فرزندان مستقیم r می‌کنیم. با این کار، ارتفاع درخت به‌شدت کم و موجب می‌شود که اعمال بعدی یافتن بسیار سریع‌تر انجام شود. به این‌کار «فشرده‌سازی مسیر[۹]» می‌گوییم.



شکل ۷-۵ فشرده‌سازی مسیر در پیاده‌سازی مبتنی بر درخت.

با این تغییر ساده، می‌توان اثبات کرد که m عمل یافتن را می‌توان حداکثر در زمان

[۹] path compression

$O(m\alpha(m,n))$ انجام داد. چنان‌چه گفتیم $\alpha(m.n)$ معکوس تابع «اَکِرمَن» است. تابع α بسیار کند رشد می‌کند و حتی برای مقادیر بسیار بزرگ n، $\alpha(m,n) \leq ۴$.

تابع اکرمن و معکوس آن

برای فهم بهتری از تابع اکرمن و معکوس آن، نماد زیر را برای تکرار توان ۲ تعریف می‌کنیم.

$$g(i) \equiv ۲^{۲^{\cdot^{\cdot^{۲}}}}\Big\}^{i}$$

این تابع به‌صورت بازگشتی زیر نیز تعریف می‌شود

$$g(i) = \begin{cases} ۲^{۱} & \text{اگر} \quad i = ۰, \\ ۲^{g(i-۱)} & \text{اگر} \quad i > ۰. \end{cases}$$

مثلاً

$$g(۴) = ۲^{۲^{\cdot^{\cdot^{۲}}}}\Big\}^{۴} = ۲^{۲^{۲^{۲}}} = ۲^{۶۵۵۳۶}.$$

هم‌چنین، چنان‌چه در پیوست «ب» هم آمده است، برای تعریف $\lg^*$ از تابع تکرار زیر استفاده می‌کنیم:

$$\lg^{(i)} n = \begin{cases} n & \text{اگر} \quad i = ۰ \\ \lg(\lg^{(i-۱)} n) & \text{اگر} \quad i > ۰ \text{ and } \lg^{(i-۱)} n > ۰, \\ \text{undefined} & \text{اگر} \quad i > ۰ \text{ and } \lg^{(i-۱)} n) \leq ۰ \text{ or } \lg^{(i-۱)} n \text{ is undefined.} \end{cases}$$

و

$$\lg^* n = \min\{i \geq ۰ : \lg^{(i)} n \leq ۱\} \tag{۷–۱}$$

حال تابع اکرمن را برای مقادیر $i, j \geq ۱$ به‌صورت زیر تعریف می‌کنیم:

$$\begin{array}{lcll} A(۱,j) & = & ۲^j & \text{برای} \quad j \geq ۱, \\ A(i,۱) & = & A(i-۱,۲) & \text{برای} \quad i \geq ۲, \\ A(i,j) & = & A(i-۱, A(i,j-۱)) & \text{برای} \quad i,j \geq ۲. \end{array} \tag{۷–۲}$$

جدول ۷–۱ مقادیر تابع اکرمن $A(i,j)$ را برای مقادیر کوچکِ i و j نشان می‌دهد که رشد بسیار سریع این تابع کاملاً مشخص است.

جدول ۷-۱ مقادیر تابع اکرمن $A(i,j)$ برای مقادیر کوچک i و j.

	$j=۱$	$j=۲$	$j=۳$	$j=۴$
$i=۱$	$۲^{۱}$	$۲^{۲}$	$۲^{۳}$	$۲^{۴}$
$i=۲$	$۲^{۲}$	$۲^{۲^{۲}}$	$۲^{۲^{۲^{۲}}}$	$۲^{۲^{۲^{۲^{۲}}}}$
$i=۳$	$۲^{۲^{۲}}$	$۲^{\cdots^{۲}\}۱۶}$	$۲^{۲^{\cdots^{۲}\}۲^{۲}}\}۱۶}$	$۲^{۲^{\cdots^{۲}\}۲^{۲^{\cdots^{۲}\}۲^{۲}}}\}۱۶}$

حال معکوس تابع اکرمن را به‌صورت زیر تعریف می‌کنیم:

$$\alpha(m,n) = \min\{i \geq ۱ : A(i, \lfloor m/n \rfloor) > \lg n\}. \qquad (۷-۳)$$

نشان می‌دهیم که برای اعداد معمولی، $\alpha(m,n) \leq ۴$. اولاً روشن است که چون $m \geq n$ مقدار $\lfloor m/n \rfloor$ دست‌کم ۱ است. چون تابع اکرمن اکیداً صعودی است، به‌ازای $\lfloor m/n \rfloor \geq ۱$ داریم $A(i, \lfloor m/n \rfloor) > A(i,۱)$. بنابراین $A(۴, \lfloor m/n \rfloor) \geq A(۴,۱)$. ولی داریم

$$A(۴,۱) = A(۳,۲) = ۲^{۲^{\cdots^{۲}\}۱۶}}$$

که بسیار بیش‌تر از تخمین تعداد کل اتم‌ها در جهان (حدود $۱۰^{۸۰}$) است. فقط برای مقادیر غیر عملی بزرگ n ممکن است $A(۴,۱) \leq \lg n$. بنابراین برای همه‌ی مقادیر عملی $\alpha(m,n) \leq ۴$.

تمرین‌های بخش ۷-۱

۷-۱.۱ ثابت کنید که در داده‌ساختار «مجموعه‌های مجزا» با n عنصر که مبتنی بر درخت و با فشرده‌سازی مسیر پیاده‌سازی شده است، هزینه‌ی n عمل ادغام و بعد از آن m عمل یافتن $O(n+m)$ است.

۷-۱.۲ روش فشرده‌سازی مسیر را طوری تغییر دهید تا علاوه بر اعمال CREATE، FIND و MERGE، بتوان عمل HISTORICAL-FIND(x,y,t) را هم با همان هزینه انجام داد. این عمل به این معنی است که آیا x و y در زمان t در یک مجموعه بوده‌اند یا خیر. منظور از «در زمان t» وضعیت داده‌ساختار پس از انجام عمل tام است. فرض کنید یک شمارنده‌ی قابل دسترسی به‌وسیله‌ی هر عمل وجود دارد که شماره‌ی عمل در حال اجرا را نشان می‌دهد.

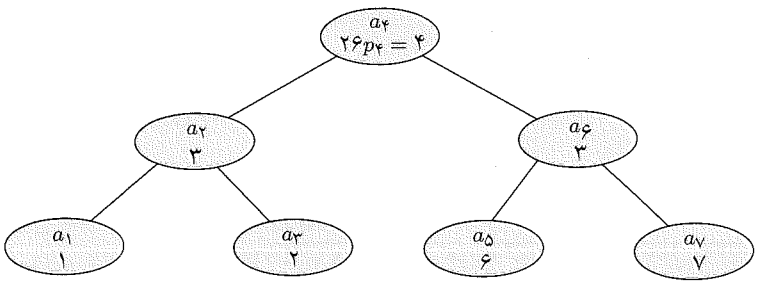
۳.۱-۷ در مسئله‌ی قبل نشان دهید که هر ترتیبی از n عمل ادغام و بعد از آن m عمل یافتن $\mathcal{O}(n \lg n + m)$ است.

۴.۱-۷ فرض کنید بر روی یک داده‌ساختار مجموعه‌ی مجزا که در ابتدای کار تهی است و با فشرده‌سازی مسیر کار می‌کند، تعداد m عمل یافتن، ادغام و ایجاد انجام گرفته است که n تا از آن‌ها ایجاد مجموعه و f تا از آن‌ها یافتن است. نشان دهید که هزینه‌ی کل اعمال یافتن روی هم‌رفته از $\mathcal{O}(n + f \lg \lg n)$ می‌باشد. (راهنمایی: از تحلیل سرشکن استفاده کنید.)

۲-۷ درخت دودویی جست‌وجوی بهینه

چنان‌چه در بخش ۴-۵ دیدیم، درخت دودویی جست‌وجو (د.د.ج) یک داده‌ساختار مناسب برای پیاده‌سازی فرهنگ‌های داده‌ای است که برای فرهنگی با n عنصر، اعمال درج، حذف و جست‌وجو را با میانگین $\mathcal{O}(\lg n)$ ولی با هزینه‌ی حداکثر $\mathcal{O}(n)$ انجام می‌دهد.

می‌دانیم که اگر داده‌ها از پیش آماده باشند، می‌توانیم آن‌ها را به‌گونه‌ای در د.د.ج قرار دهیم، تا درختی متوازن به‌دست آید. بدین ترتیب می‌توان در این صورت هریک از اعمال فرهنگ داده‌ای را در بدترین حالت با $\mathcal{O}(\lg n)$ انجام داد. برای ساختن چنین درختی، باید عنصر میانه را در ریشه قرار داد که با این کار نیمی از عناصر در زیردرخت چپ و نیمی دیگر در زیر درخت راست قرار می‌گیرند. همین کار را نیز به‌صورت بازگشتی بر روی زیردرخت‌های چپ و راست انجام می‌دهیم. درخت شکل ۷-۶ چنین درختی برای ۷ عنصر $a_1 < a_2 < \cdots < a_7$ است.



شکل ۷-۶ درخت متوازن با میانگین زمان جست‌وجوی $\frac{64}{26}$.

یک د.د.ج را در نظر بگیرید که عمل جست‌وجو در مقایسه با بقیه‌ی اعمال بسیار زیاد اتفاق می‌افتد. مثلاً، اگر اطلاعات مربوط به یک کتاب‌خانه را با این درخت پیاده‌سازی کنیم، بخش عمده‌ی کار بر روی آن عمل جست‌وجو است و این جست‌وجوها ممکن است «موفق» یا «ناموفق» باشند؛ یعنی کتاب در کتاب‌خانه موجود باشد یا خیر.

در عمل، تعداد تقاضاهای جست‌وجو برای هر عنصر این درخت (کتاب‌های مختلف) بسیار متفاوت است، و این داده را می‌توان از سابقه‌ی انجام این اعمال در مدت زمانی به‌دست آورد و آن‌را هر چند مدت به‌روز کرد. با توجه به آن‌که زمان انجام جست‌وجو برای یک عنصر متناسب با عمق آن عنصر در د.د.ج است، اگر بتوانیم عناصری را که تعداد جست‌وجو برای آن‌ها بیش‌تر است به ریشه نزدیک‌تر کنیم و آن‌هایی را که کم‌تر مورد جست‌وجو قرار می‌گیرند دورتر از ریشه قرار دهیم، میانگین زمان جست‌وجو حتی کم‌تر از حالت متوازن خواهد شد. این مطلب، ایده‌ی اصلی درخت دودویی جست‌وجوی بهینه است.

در درخت شکل ۷-۶ فرض کنید $P(a)$ احتمال آن باشد که یک جست‌وجوی دل‌خواه برای یافتن عنصر a انجام شود. هم‌چنین به‌عنوان مثال، فرض کنید

$$\begin{aligned} P(a_۱) &= \frac{۱}{۲۶}, & P(a_۲) &= \frac{۳}{۲۶} & P(a_۳) &= \frac{۲}{۲۶} \\ P(a_۴) &= \frac{۴}{۲۶}, & P(a_۵) &= \frac{۶}{۲۶} \\ P(a_۶) &= \frac{۳}{۲۶}, & P(a_۷) &= \frac{۷}{۲۶} \end{aligned}$$

در شکل، احتمال جست‌وجو برای هر عنصر (ضرب‌در ۲۶) نشان داده شده است. با توجه به آن‌که زمان جست‌وجو برای یک عنصر a_i متناسب است با عمق a_i در درخت به‌اضافه‌ی ۱ (که برابر است با تعداد مقایسه‌ها بین a_i و کلیدهای گره‌های درخت در مسیر جست‌وجوی a_i)، میانگین زمان جست‌وجو در این درخت برابر است با

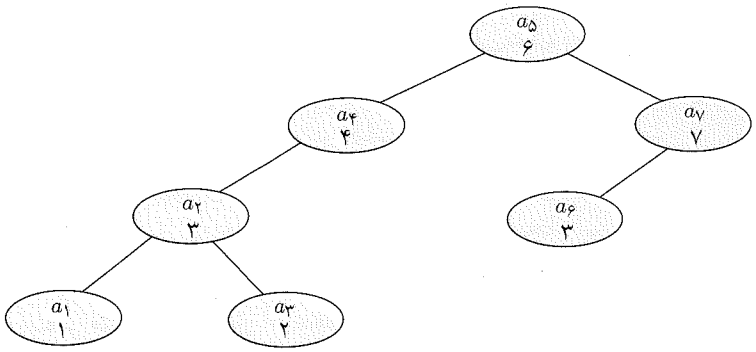
$$\frac{۱}{۲۶}[۴ \times ۱ + (۳+۳) \times ۲ + (۱+۲+۶+۷) \times ۳] = \frac{۶۴}{۲۶}.$$

اگر به‌جای این درخت، درخت شکل ۷-۷ را برای این عناصر ایجاد کنیم، با این‌که ارتفاع درخت بیش‌تر شده است، اما میانگین زمان جست‌وجو کم‌تر و برابر مقدار

$$\frac{۱}{۲۶}[۶ \times ۱ + (۴+۷) \times ۲ + (۳+۳) \times ۳ + (۱+۲) \times ۴] = \frac{۵۸}{۲۶}$$

می‌شود. دقت کنید که در این مثال جست‌وجوی ناموفق نداریم و احتمال آن صفر است.

به‌عنوان مثالی دیگر، توجه کنید که هر کلمه‌ی استفاده‌شده در متن یک زبان برنامه‌نویسی، بررسی می‌شود که آیا جزءِ کلمات کلیدی آن زبان (که در جدول نمادها در



شکل ۷–۷ درخت نامتوازن با میانگین زمان جست‌وجوی $\frac{58}{26}$.

آن کامپایلر موجود است) هست یا خیر. این کار به دفعات زیاد در زمان ترجمه‌ی یک برنامه تکرار می‌شود و بنابراین لازم است سریع باشد. مثلاً در زبان پاسکال، ممکن است یک جدول نماد کوچک از ۸ کلمه‌ی کلیدی به‌صورت شکل ۷–۸ ساخته شود.

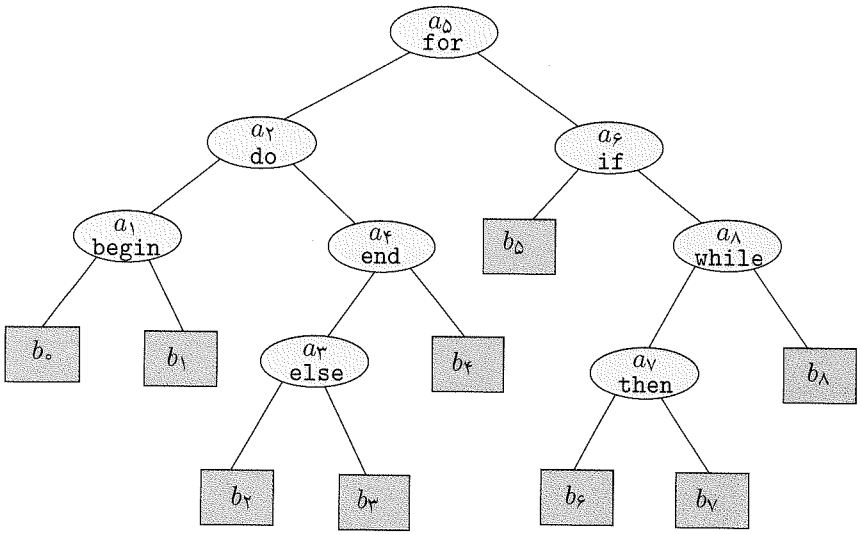
جست‌وجوهای ناموفق برای یافتن عناصری انجام می‌شوند که در درخت نیستند. این جست‌وجوها هم بسیار مهم هستند و در شکل بهینه‌ی درخت تأثیر می‌گذارند و باید آن‌ها را نیز در نظر گرفت. این جست‌وجوها را در یک د.د.ج با عناصر خارجی آن نشان می‌دهیم که در شکل ۷–۸ با عناصر مربعی‌شکل نشان داده شده‌اند. به‌ازای هر اشاره‌گر **null** در درخت اصلی، یک عنصر خارجی فرض می‌کنیم. مثلاً اگر عناصر خارجی را b_i بنامیم، b_0 نشان‌دهنده‌ی جست‌وجو برای کلیه‌ی کلمات کوچک‌تر از a_1 یا `begin` است، b_1 نشان‌دهنده‌ی جست‌وجو برای کلیه‌ی کلماتی است که از `begin` بزرگ‌تر و از `do` کم‌تر هستند، به‌همین صورت برای دیگر bها. همه‌ی جست‌وجوها برای عناصری با مقدار بیش‌تر از `while` (بزرگ‌ترین کلمه‌ی موجود در فرهنگ) به b_8 ختم می‌شوند.

مسئله‌ی ۷–۱ ثابت کنید تعداد عناصر خارجی، برای یک د.د.ج با n عنصر، برابر $n+1$ است.

اثبات: به سه روش زیر می‌توان این مسئله‌ی ساده را حل کرد:

۱. با استفاده از استقرا و حذف یک عنصر برگ و دو عنصر خارجی آن.

۲. می‌دانیم که به‌ازای هر عنصر غیر ریشه، یک پدر و در نتیجه یک یال متناظر



شکل ۷-۸ یک درخت دودویی جست‌وجو برای جدول نمادها.

وجود دارد (در کل $n-1$ یال). از طرف دیگر، هر عنصر دو فرزند دارد که برخی از فرزندها عناصر خارجی هستند. بنابراین $2n$ یال به عناصر داخلی و خارجی وارد می‌شوند. پس $2n-(n-1)=n+1$ عنصر به عنوان عنصر خارجی باقی می‌ماند.

۳. اگر n عنصر درخت $a_1 < a_2 < \cdots < a_n$ باشند، آن‌گاه $n+1$ بازه بین آن‌ها برای عناصر خارجی b_i موجود است: $b_0 < a_1 < b_1 < a_2 < \cdots < a_n < b_n$.

□

ساخت درخت دودویی جست‌وجوی بهینه

فرض کنید که

- p_i برابر است با احتمال جست‌وجو برای a_i (جست‌وجوی موفق) و
- q_i برابر است با احتمال جست‌وجو برای b_i (جست‌وجوی ناموفق برای عنصر x که $a_i < x < a_{i+1}$، $x < a_1$ یا $a_n < x$).

با داشتن p_iها و q_jها (برای $i = ۱, ۲, \ldots, n$ و $j = ۰, ۱, \ldots, n$) هدف، ساختن یک د.د.ج برای عناصر $a_۱ < a_۲ < \cdots < a_n$ است به‌طوری‌که میانگین زمان جست‌وجو (اعم از موفق یا ناموفق) کمینه شود. به چنین درختی «دودویی جست‌وجوی بهینه[۱۰]» می‌گوییم.

در این‌جا، زمان جست‌وجو برای مقدار x را تعداد مقایسه‌های بین x و کلیدهای درخت دودویی جست‌وجو تعریف می‌کنیم.

در یک د.د.ج T، عنصر a_i را درنظر بگیرید. جست‌وجو برای a_i به‌تعداد $depth(a_i) + ۱$ مقایسه بین کلیدهای عناصر در T نیاز دارد که $depth(a_i)$ عمق این عنصر در T است. اگر جست‌وجوی x ناموفق باشد، بسته به مقدار x، الگوریتم جست‌وجو در یکی از اشاره‌گرهای **null** متوقف می‌شود، به‌عبارت دیگر، به یکی از عناصر b_i می‌رسد. بنابراین برای جست‌وجوهای ناموفق می‌توانیم بگوییم که جست‌وجو به یکی از b_iها ختم می‌شود. با توجه به این‌که عناصر b_i در عمل پیاده‌سازی نمی‌شوند، برای رسیدن به b_i تنها تعداد $depth(b_i)$ مقایسه بین کلید عناصر T با x انجام می‌شود.

به‌طور خلاصه،

$$\text{میانگین زمان جست‌وجو} = \sum_{i=۱}^{n} p_i[۱ + depth(a_i)] + \sum_{j=۰}^{n} q_j\, depth(b_j). \qquad (۴\text{-}۷)$$

همچنین، می‌دانیم که $\sum_{i=۱}^{n} p_i + \sum_{j=۰}^{n} q_j = ۱$.

۷-۲-۱ راه‌حل بازگشتی

ابتدا سعی می‌کنیم مسئله را به‌صورت بازگشتی و بر اساس درخت‌های حاصل از زیرمسئله‌های مشابه ولی کوچک‌تر حل کنیم. نشان می‌دهیم که زمان اجرای این راه‌حل نمایی است. سپس نشان می‌دهیم که اگر جواب‌های همه‌ی زیرمسئله‌ها را به‌تدریج در جدولی ذخیره کنیم و به‌هنگام نیاز به حل یک زیرمسئله، از جواب موجود در آن جدول استفاده کنیم، یک راه‌حل چندجمله‌ای برای مسئله پیدا کرده‌ایم.

به راه‌حل‌های این‌چنینی، «برنامه‌ریزی پویا[۱۱]» (یا روش پویا) می‌گوییم. این یک روش مهم برای حل برخی مسئله‌های بهینه‌سازی است که در کتاب‌هایی مانند [۵] و [۱۰] به‌طور مفصل مورد بحث قرار گرفته است.

[۱۰] optimal binary search tree
[۱۱] dynamic programming

آن‌چه در هر دو روش بازگشتی و پویا از اهمیت برخوردار است، تعریف دقیق زیرمسئله در حالت کلی و یافتن رابطه‌ای بین جواب یک زیرمسئله‌ی بزرگ‌تر با جواب‌های زیرمسئله‌های کوچک‌تر است. البته باید نشان داد که برای آن‌که مسئله‌ی بزرگ‌تر به‌صورت بهینه حل شود لازم و کافی است که زیرمسئله‌ها هم به‌صورت بهینه حل شوند.

تعریف‌ها

برای حل مسئله به تعریف‌های زیر توجه کنید:

زیرمسئله‌ی T_{ij}: د.د.ج بهینه برای عناصر $a_j > \cdots > a_{i+1}$ که با داشتن $\langle q_i, p_{i+1}, q_{i+1}, \cdots, p_j, q_j \rangle$ ایجاد می‌شود. درخت مسئله‌ی اصلی $T_{0,n}$ است.

C_{ij}: هزینه‌ی بهینه، یا میانگین زمان جست‌وجوی بهینه در T_{ij} که مطابق رابطه‌ی زیر (مشابه ۷-۴) محاسبه می‌شود.

$$C_{ij} = \sum_{r=i+1}^{j} p_r[1 + depth(a_r)] + \sum_{r=i}^{j} q_r \; depth(b_r) \qquad (۵-۷)$$

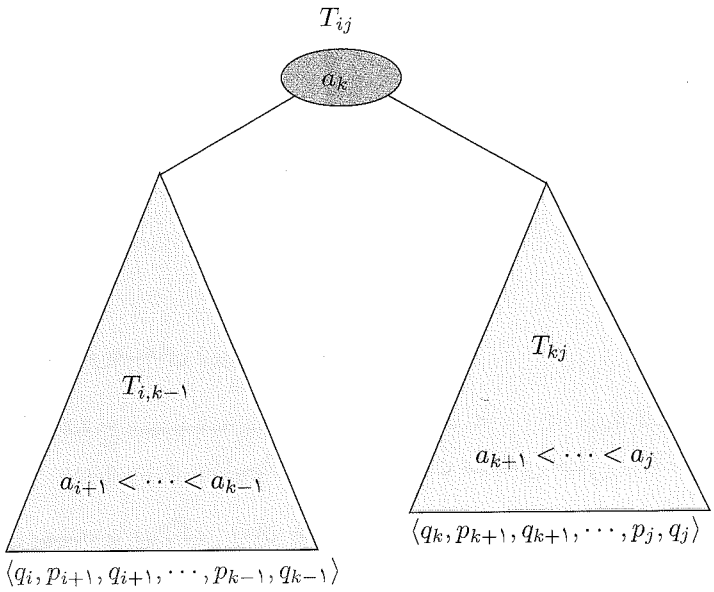
w_{ij}: مجموع احتمال‌ها در T_{ij}، یعنی $w_{ij} = q_i + \sum_{r=i+1}^{j}(p_r + q_r)$.

r_{ij}: شماره‌ی ریشه‌ی T_{ij}.

برای ساختن T_{ij} ریشه‌ی آن‌را مشخص می‌کنیم. فرض کنید a_k $(i < k \leq j)$ ریشه‌ی درخت باشد. در این‌صورت، C^k_{ij} را هزینه‌ی بهینه‌ی این درخت تعریف می‌کنیم. اگر a_k ریشه باشد، عناصر $a_{i+1} < \cdots < a_{k-1}$ با احتمال‌های $\langle q_i, p_{i+1}, q_{i+1}, \cdots, p_{k-1}, q_{k-1} \rangle$ (یعنی زیرمسئله‌ی $T_{i,k-1}$) حتماً زیردرخت چپ را تشکیل می‌دهند. همچنین، $a_{k+1} < \cdots < a_j$ با احتمال‌های $\langle q_k, p_{k+1}, q_{k+1}, \cdots, p_j, q_j \rangle$ (یعنی زیرمسئله‌ی T_{kj}) زیردرخت راست را تشکیل می‌دهند.

بدیهی‌است برای آن‌که T_{ij} بهینه باشد، این دو زیردرخت هم باید بهینه باشند. یعنی هزینه‌های زیردرخت‌های چپ و راست به‌ترتیب باید $C_{i,k-1}$ و C_{kj} باشند. (به شکل ۷-۹ مراجعه کنید.)

توجه کنید که $C_{i,k-1}$ هزینه‌ی بهینه‌ی $T_{i,k-1}$ است، اگر به‌صورت مستقل طراحی شود. هنگامی‌که این درخت به‌عنوان زیردرخت چپ T_{ij} محسوب شود، عمق همه‌ی عناصر آن (چه aها و چه bها) هریک، یک واحد افزایش می‌یابد. پس با توجه به رابطه‌ی ۷-۵،



شکل ۷-۹ درخت دودویی جست‌وجوی بهینه‌ی T_{ij} شامل عناصر $a_{i+1} < \cdots < a_j$. اگر a_k ریشه باشد، $T_{i,k-1}$ زیردرخت سمت چپ و T_{kj} زیردرخت سمت راست و هر دو بهینه هستند.

مقدار هزینه‌ی زیردرخت $T_{i,k-1}$ در درخت T_{ij} برابر $C_{i,k-1} + w_{i,k-1}$ خواهد بود. با همین استدلال مقدار هزینه‌ی زیردرخت راست (T_{kj}) در درخت T_{ij} برابر $C_{kj} + w_{kj}$ خواهد بود. بنابراین،

$$C_{ij}^k = (C_{i,k-1} + w_{i,k-1}) + (C_{kj} + w_{kj}) + p_k = C_{i,k-1} + C_{kj} + w_{ij}.$$

و در نتیجه

$$C_{ij} = \begin{cases} q_i & \text{اگر} \quad i = j \\ \min_{i<k\leq j}\{C_{i,k-1} + C_{kj}\} + w_{ij} & \text{اگر} \quad i < j. \end{cases} \qquad (۶\text{-}۷)$$

۷-۲-۲ راه‌حل پویا

رابطه‌ی ۷-۶ اساس الگوریتم بازگشتی برای این مسئله است. اگر از روش بازگشتی استفاده کنیم، زیرمسئله‌های تکراری خواهیم داشت، یعنی ممکن است یک زیرمسئله‌ی خاص را

چندین بار حل کنیم. روش بازگشتی در واقع همه‌ی درخت‌های دودویی جست‌وجویی که با این n عنصر می‌توان ساخت (همان عدد کاتالان) را در نظر می‌گیرد و بین آن‌ها بهینه را تشخیص می‌دهد، که تعداد این درخت‌ها و در نتیجه زمان اجرای الگوریتم نمایی است. در حالی‌که اگر نتیجه‌ی راه‌حل بهینه‌ی هر زیرمسئله را فقط یک‌بار حل کنیم و حاصل آن‌را در جایی ذخیره کنیم، برای حل زیرمسئله‌ی بزرگ‌تر ضرورتی به فراخوانی زیرمسئله‌های کوچک‌تر نیست، چون که آن زیرمسئله‌ها قبلاً حل شده و جواب ذخیره‌شده‌اش قابل استفاده است. این همان راه‌حل پویا است.

برای راه‌حل پویا ماتریس‌های

$$C[\circ\cdots n, \circ\cdots n] \text{ ،} w[\circ\cdots n, \circ\cdots n] \text{ و } r[\circ\cdots n, \circ\cdots n]$$

را تعریف می‌کنیم، تا برای درخت T_{ij} $(i \le j)$، $C[i,j]$، $w[i,j]$ و $r[i,j]$ به‌ترتیب مقادیر C_{ij}، w_{ij} و r_{ij} را که در بالا تعریف شدند ذخیره کنند.

MAKE-OBST رویه‌ی مورد نظر است. در سطرهای ۱ تا ۳، ماتریس‌ها برای درخت‌های T_{ii} مقداردهی می‌شوند. یعنی، $C_{ii} = \circ$ و $w_{ii} = q_i$ و این عناصر قطر ماتریس‌ها را تشکیل می‌دهند. در ادامه، فرض می‌کنیم زیرمسئله l گره دارد (برای $1 \le l \le n$). توجه کنید که این عناصر l عنصر مرتب و متوالی ورودی هستند. این امر از نحوه‌ی بازگشتی تقسیم مسئله به زیرمسئله‌ها به‌دست می‌آید. بنابراین مطابق تعریف، هر کدام از آن‌ها یک زیرمسئله‌ی مستقل هستند.

```
MAKE-OBST (p, q)
    ▷ a_1 < ··· < a_n یک د.د.ج بهینه برای عناصر
      و احتمال‌های ⟨q_0, p_1, q_1, ···, p_n, q_n⟩ می‌سازد
1   for i ← 0 to n
2     do w[i,i] ← q_i
3        C[i,i] ← 0
4   for l ← 1 to n
5     do for i ← 0 to n − l
6          do j ← i + l
7             w[i,j] ← w[i,j−1] + p[j] + q[j]
8             C[i,j] ← min_{i<k≤j} {C[i,k−1] + C[k+1,j]} + w[i,j]
                  ▷ فرض کنید m مقدار k است که کمینه‌ی بالا را می‌سازد
9             r[i,j] ← a_m
```

برای این‌که T_{ij} شامل l گره‌ی متوالی از $a_1 < \cdots < a_n$ باشد، i می‌تواند مقادیر ۰ تا $n-l$ را بگیرد، که در سطر ۵ هر حالت آن بررسی می‌شود. به‌ازای هر مقدار i مقدار j برابر $i+l$ خواهد شد (سطر ۶). در سطر ۷ مقدار $w[i,j]$ را از مقدار $w[i,j-1]$ که قبلاً محاسبه شده است به‌دست می‌آوریم. سطر ۸ همان رابطه‌ی ۷-۶ است که شماره‌ی ریشه‌ی درخت بهینه‌ی T_{ij} را به‌دست می‌آورد که این مقدار در $r[i,j]$ قرار می‌گیرد.

نتیجه‌ی ۷-۷ الگوریتم MAKE-OBST از $\Theta(n^3)$ است.

مثالی از ساخت د.د.ج بهینه

می‌خواهیم یک د.د.ج بهینه برای چهار عنصر

$$a_1 < a_2 < a_3 < a_4$$

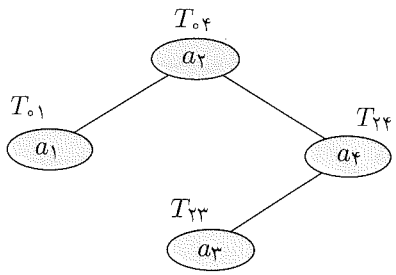
با احتمال‌های زیر ایجاد کنیم.

$$\begin{aligned} p_1 &= \frac{1}{4}, & p_2 &= \frac{1}{8}, & p_3 = p_4 &= \frac{1}{16} \\ q_0 &= \frac{1}{8}, & q_1 &= \frac{3}{16}, & q_2 = q_3 = q_4 &= \frac{1}{16} \end{aligned}$$

برای این مثال، مقادیر C_{ij}، w_{ij} و r_{ij} در انتهای الگوریتم در جدول ۷-۲ نشان داده شده است. این ماتریس به‌صورت قطری پر می‌شود. قطر اصلی مقادیر اولیه هستند، قطر بعدی هزینه‌های همه‌ی درخت‌های بهینه با فقط یک عنصر را نشان می‌دهد. هر قطر نسبت به قطر قبلی، یک عنصر اضافه دارد که از اندیس آن درایه معین می‌شود. در انتها، درایه‌ی $C_{0,4}$ حاوی همه‌ی اطلاعات درخت بهینه‌ی نهایی است. توجه کنید که از روی این ماتریس می‌توان درخت را ساخت. چون، در این مثال از $r_{04} = a_2$ معلوم می‌شود که a_2 ریشه‌ی درخت بهینه است. بنابراین T_{01} زیردرخت چپ و T_{34} زیردرخت راست در درخت نهایی است. جدول محاسبه شده برای هرکدام از این زیردرخت‌ها هم اطلاعات کافی برای ساخت آن را دارد و این کار به‌صورت بازگشتی دنبال می‌شود. شکل ۷-۱۰ درخت بهینه‌ی این مثال است.

جدول ۷-۲ مقادیر (ضرب‌در ۱۶) C_{ij}، w_{ij} و r_{ij} برای مثال.

	۰	۱	۲	۳	۴
۰	$C_{۰۰}=۰$ $w_{۰۰}=۲$	$C_{۰۱}=۹$ $w_{۰۱}=۹$ $r_{۰۱}=a_۱$	$C_{۰۲}=۱۸$ $w_{۰۲}=۱۲$ $r_{۰۲}=a_۱$	$C_{۰۳}=۲۵$ $w_{۰۳}=۱۴$ $r_{۰۳}=a_۱$	$C_{۰۴}=۳۳$ $w_{۰۴}=۱۶$ $r_{۰۴}=a_۲$
۱		$C_{۱۱}=۰$ $w_{۱۱}=۳$	$C_{۱۲}=۶$ $w_{۱۲}=۶$ $r_{۱۲}=a_۲$	$C_{۱۳}=۱۱$ $w_{۱۳}=۸$ $r_{۱۳}=a_۲$	$C_{۱۴}=۱۸$ $w_{۱۴}=۱۰$ $r_{۱۴}=a_۲$
۲			$C_{۲۲}=۰$ $w_{۲۲}=۱$	$C_{۲۳}=۳$ $w_{۲۳}=۳$ $r_{۲۳}=a_۳$	$C_{۲۴}=۸$ $w_{۲۴}=۵$ $r_{۲۴}=a_۴$
۳				$C_{۳۳}=۰$ $w_{۳۳}=۱$	$C_{۳۴}=۳$ $w_{۳۴}=۳$ $r_{۱۴}=a_۴$
۴					$C_{۴۴}=۰$ $w_{۴۴}=۱$



شکل ۷-۱۰ درخت دودویی جست‌وجوی بهینه برای مثال.

تمرین‌های بخش ۷-۲

۱.۲-۷ گام‌های مختلف ایجاد یک د.د.ج بهینه با چهار کلید $a_۱ < a_۲ < a_۳ < a_۴$ را با رسم جدول و تعیین هزینه نشان دهید و درخت نهایی را رسم کنید. فرض کنید که احتمال‌های جست‌وجو (ضرب در ۲۵) به قرار زیرند:

$$p_۱=۳, \quad p_۲=۲, \quad p_۳=۱, \quad p_۴=۴$$
$$q_۱=۲, \quad q_۲=۳, \quad q_۳=۴, \quad q_۴=۱, \quad q_۵=۲.$$

۲.۲-۷ یک درخت دودویی بهینه برای $a_{۲۴} < a_{۲۳} < \cdots < a_۲ < a_۱$ بسازید، با این فرض که: $p_۱ = ۵ \times ۲۸$ و $p_i = ۱ \times ۲۸$ برای $۱ < i \leq ۲۴$، و $q_i = ۰$ برای $۰ < i \leq ۲۴$. درخت را رسم و برهان خود را به‌اختصار ذکر کنید.

۳.۲-۷ نشان دهید که اگر r_{ij} شماره‌ی ریشه‌ی د.د.ج بهینه‌ی T_{ij} باشد، داریم $r_{i,j-۱} < r_{ij} < r_{i+۱,j}$.

⋆ **۴.۲-۷** ثابت کنید که می‌توان الگوریتم ساخت را بر اساس نتیجه‌ی تمرین ۷-۳.۲ تغییر داد و در آن‌صورت هزینه‌ی الگوریتم به‌جای $\Theta(n^۳)$، $\Theta(n^۲)$ خواهد شد.

۵.۲-۷ مجموع تعداد خواندن از درایه‌های جدول C را برای محاسبه‌ی د.د.ج بهینه دقیقاً برحسب n حساب کنید. این محاسبه درواقع هزینه‌ی دقیق الگوریتم بیان‌شده در این بخش است.

۶.۲-۷ در یک د.د.ج به هر عنصر a_i یک احتمال نسبت می‌دهیم. میانگین عمق برگ‌های این درخت برابر $\sum_{\text{leaf } v} p_v depth(v)$ تعریف می‌شود. فرض کنید که $a_۱ < a_۲ < \cdots < a_{۱۰}$، $p_۱ = p_{۱۰} = ۲$ و $p_۲ = \cdots = p_۹ = ۱$. مقدار کمینه‌ی میانگین عمق برگ‌ها چند است؟

۷-۳ درخت‌های دودویی جست‌وجو با ارتفاع لگاریتمی

گسترش‌های زیادی بر د.د.ج پیشنهاد شده است، عمدتاً با این هدف که در اعمال مختلف، ارتفاع آن همیشه $\mathcal{O}(\lg n)$ باقی بماند، و این‌کار با هزینه‌ی اضافی کمی صورت گیرد. از آن‌جمله، داده‌ساختارهای زیر هستند:

- درخت قرمز-سیاه.
- درخت اِی.وی.اِل[۱۲]، و
- درخت اسپلِی[۱۳].

در این بخش، ابتدا درخت قرمز-سیاه را با جزئیات بیش‌تری تشریح می‌کنیم. سپس خصوصیات درخت اِی.وی.اِل را به اختصار ذکر می‌کنیم. در بخش ۷-۳-۲، گسترش‌های دیگری به درخت قرمز-سیاه پیشنهاد می‌کنیم تا در کاربردهای مشخصی استفاده شوند. به‌این منظور، «درخت بازه[۱۴]» و «درخت مرتبه‌ی آماری[۱۵]» معرفی خواهند شد. درخت اسپلِی در مراجع مختلف از جمله در [۵] تشریح شده است.

[۱۲] A.V.L tree
[۱۳] splay tree
[۱۴] range tree
[۱۵] order-statistic tree

۷-۳-۱ درخت قرمز-سیاه

درخت قرمز-سیاه با n گره یک د.د.ج است که با اضافه کردن یک رنگ قرمز یا سیاه (یک بیت) به هر گره، کاری می‌کنیم که ارتفاع آن همیشه $O(\lg n)$ شود. در این درخت، بسیاری از اعمال درج و حذف، به‌صورت معمولی (مانند د.د.ج ساده) انجام می‌شوند و ارتفاع آن از حالت لگاریتمی خارج نمی‌شود. اما در برخی موارد لازم است که با انجام اعمالی با هزینه‌ی کل ثابت، ارتفاع درخت را متوازن کنیم. در نتیجه در این درخت، هر یک از اعمال درج، حذف، جست‌وجو و دیگر اعمالی را که د.د.ج فراهم می‌آورد می‌توان در زمان لگاریتمی انجام داد.

تعریف درخت قرمز-سیاه و خواص آن

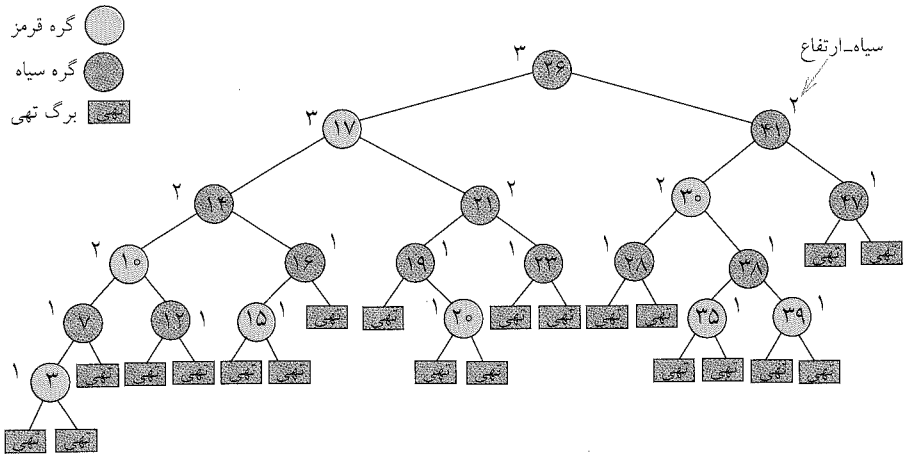
درخت قرمز-سیاه T با n عنصر خاصیت «قرمز-سیاه» زیر را دارد:

- یک د.د.ج است که هر عنصر آن (علاوه بر پیوندهای فرزند چپ، فرزند راست، و پدر) یک مؤلفه‌ی یک بیتی «رنگ» با مقادیر «سیاه» یا «قرمز» دارد.
- اشاره‌گرهای تهی در این درخت را «برگ‌های تهی[۱۶]» و سایر عناصر را «گره‌های داخلی[۱۷]» می‌نامیم. برگ‌های تهی سیاه هستند.
- پدر یک گره قرمز حتماً سیاه است.
- به‌ازای هر گره داخلی $x \in T$، تعداد گره‌های سیاه (بجز خود x) موجود که از x تا هر یک از نواده‌ی برگِ تهی آن برود، برابر است. این تعداد را «سیاه-ارتفاع[۱۸]» آن گره می‌گوییم و آن‌را با $bh(x)$ نمایش می‌دهیم.
- رنگ ریشه «سیاه» فرض می‌شود، هرچند این شرط ضروری نیست.

شکل ۷-۱۱ یک درخت قرمز-سیاه را با ۲۰ گره (و ۲۱ برگ تهی) نشان می‌دهد. کلیدهای گره‌ها در داخل آن نوشته شده‌اند و خاصیت د.د.ج را دارا هستند. رنگ گره‌ها مشخص شده است. در کنار هر گره، سیاه-ارتفاع آن هم نشان داده شده است.

---

۱۶ null leaves
۱۷ internal nodes
۱۸ black-height



شکل ۷-۱۱ مثالی از یک درخت قرمز‌ـسیاه با ۲۰ گره. عدد داخل هر گره کلید و عدد کنار آن، سیاه‌ـارتفاع آن گره است. اشاره‌گرهای تهی یا «برگ‌های تهی» هم نشان داده شده‌اند.

قضیه‌ی ۷-۱ بیشینه‌ی ارتفاع یک درخت قرمز‌ـسیاه که دارای n گره داخلی باشد، برابر $2\lg(n+1)$ است.

اثبات: ابتدا نشان می‌دهیم که هر زیردرخت دل‌خواهی به ریشه‌ی x، دست‌کم تعداد $2^{bh(x)}-1$ گره داخلی دارد. برای اثبات، از استقرا بر روی ارتفاع x استفاده می‌کنیم.

پایه: اگر ارتفاع x صفر باشد، x برگ و در نتیجه تهی است، و می‌دانیم که چنین زیردرختی گره داخلی ندارد و این همان $2^{bh(x)}-1=0$ است.

گام استقرا: سیاه‌ـارتفاع x عددی مثبت است. بنابراین x دو فرزند دارد که سیاه‌ـارتفاع هر کدام از آن‌ها برحسب این‌که x قرمز باشد یا سیاه، به‌ترتیب برابر $bh(x)$ یا $bh(x)-1$ است. چون ارتفاع هر فرزند کم‌تر از ارتفاع x است، از فرض استقرا نتیجه می‌گیریم که زیردرخت‌های به ریشه‌ی هر فرزند x هر کدام دست‌کم $2^{bh(x)-1}-1$ گره داخلی دارند. بنابراین زیردرخت به ریشه‌ی x هم دست‌کم حاوی $2^{bh(x)-1}-1+2^{bh(x)-1}-1+1=2^{bh(x)}-1$ گره داخلی است.

از طرف دیگر، می‌دانیم که در یک درخت به ارتفاع h، دست‌کم نیمی از گره‌های موجود بر روی هر مسیر ساده از ریشه به برگ (با نشمردن ریشه)، سیاه هستند. زیرا در

غیراین‌صورت باید پدر یک گره قرمز، قرمز باشد، و این ممکن نیست. نتیجه می‌گیریم که سیاه‌ـارتفاع ریشه‌ی درخت دست‌کم برابر $\frac{h}{۲}$ است. بنابراین

$$n \geq ۲^{\frac{h}{۲}} - ۱ \Rightarrow ۲^{\frac{h}{۲}} \leq n + ۱ \Rightarrow h \leq ۲\lg(n+۱)$$

□

نتیجه‌ی ۷-۸ ارتفاع یک درخت قرمز‌ـسیاه با n گره داخلی $O(\lg n)$ است.

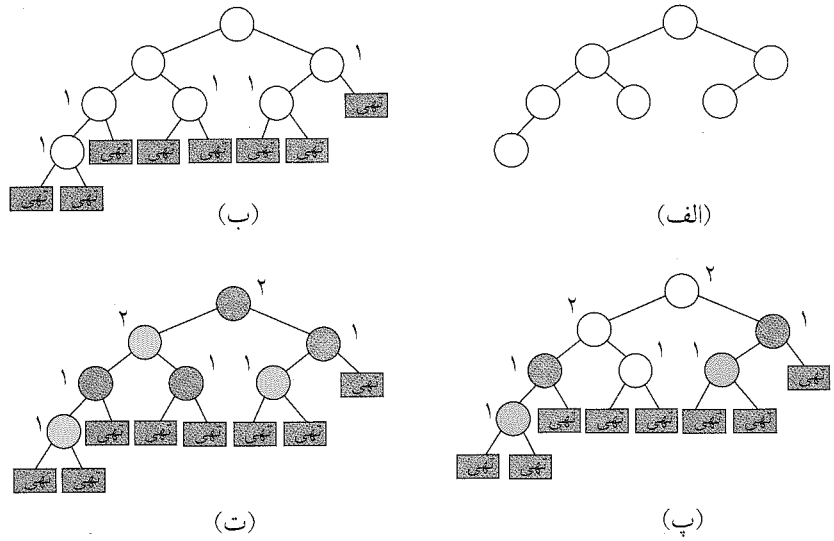
نکته‌ی ۷-۱ یک د.د.ج را چگونه می‌توان با رنگ‌های قرمز و سیاه رنگ کرد تا قرمز‌ـسیاه شود؟ فرض کنید که برای یک گره x، دو مسیر مختلفِ $p_۱$ و $p_۲$ از x به دو نواده‌ی برگِ تهیِ آن داریم، و طول هر مسیر p را تعداد گره‌های آن (بدون شمارش خودِ x) بگیریم و با $d(p)$ نشان دهیم. اگر $d(p_۱) > ۲d(p_۲)$ باشد، آن درخت را نمی‌توان به‌صورت قرمز‌ـسیاه رنگ‌آمیزی کرد، چون می‌دانیم که پدر گره قرمز نمی‌تواند قرمز باشد و باید تعداد گره‌های سیاه در $p_۱$ و $p_۲$ برابر باشند. اما اگر $d(p_۱) = ۲d(p_۲)$، می‌توان همه‌ی گره‌های $p_۲$ را سیاه و گره‌های $p_۱$ را یک‌درمیان قرمز و سیاه کرد تا $bh(x)$ دست‌کم در این دو مسیر درست شود.

کار رنگ‌آمیزی یک د.د.ج را از برگ‌های تهی که باید سیاه باشند آغاز می‌کنیم، و به سمت ریشه بالا می‌رویم و در هر مرحله سعی می‌کنیم سیاه‌ـارتفاع هر گره را به‌دست آوریم. در این حرکت رنگ گره‌های درخت مشخص می‌شوند. این الگوریتم شهودی را می‌توانید در شکل ۷-۱۲ دنبال کنید. البته بر پایه‌ی این توضیح، می‌توان الگوریتمی بازگشتی طراحی کرد که یک رنگ‌آمیزی درست را در صورت وجود پیدا کند.

ساختار و گره عمو

برای یک گره x، مؤلفه‌های $left[x]$، $right[x]$، $p[x]$، و $color[x]$ را تعریف می‌کنیم تا به‌ترتیب، فرزند چپ، فرزند راست، پدر و رنگ آن گره را نشان دهند. در آن‌صورت عموی گره x ($uncle[x]$) به این‌صورت قابل بیان است:

```
if p[x] = right[p[p[x]]]
   then uncle[x] ← left[p[p[x]]]
   else uncle[x] ← right[p[p[x]]]
```



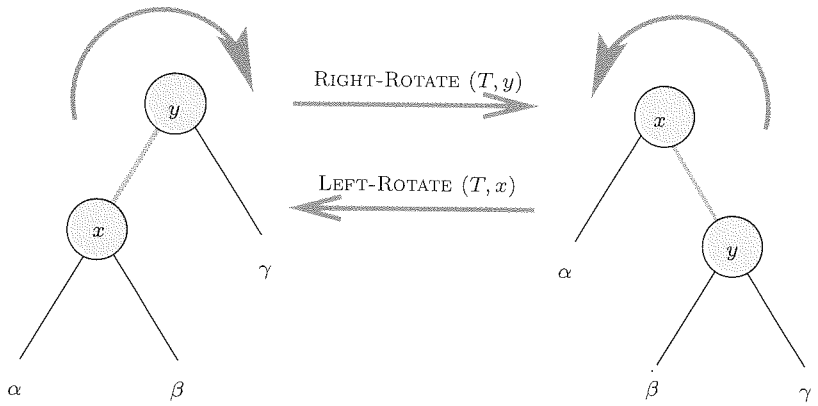
شکل ۷-۱۲ مثالی از یک الگوریتم شهودی برای رنگ‌آمیزی یک درخت د.د.ج به‌صورت قرمز-سیاه. (الف) درخت د.د.ج، (ب) با در نظر گرفتن برگ‌های تهی، سیاه-ارتفاع تعدادی از گره‌ها مشخص می‌شوند. با این اطلاعات می‌توان برخی از گره‌ها را به‌صورت تک رنگ کرد (پ)، و کار را ادامه داد تا این‌که یا به تناقض رسید یا همه‌ی درخت رنگ شود (ت). البته رنگ‌آمیزی در صورت وجود ممکن است تک نباشد.

دَوَران

اگر اعمال درج و حذف را به‌صورت ساده انجام دهیم، ممکن است درخت قرمز-سیاه نامتوازن شود. با انجام «دوران[۱۹]» می‌توانیم این مشکل را اصلاح کنیم. این عمل پایه‌ای است که در متوازن‌سازی دیگر ساختارهای د.د.ج متوازن (مانند ای.وی.ال) هم استفاده می‌شود.

دوران یا «راست‌گرد[۲۰]» است یا «چپ‌گرد[۲۱]» و این‌دو در شکل ۷-۱۳ نشان داده شده‌اند. اگر α، β و γ د.د.ج های کوچک‌تر در شکل باشند، و اگر $\langle\alpha\rangle$ را ترتیب میان‌وندی (مرتب) زیردرخت α بگیریم، روشن است که، هم قبل و هم بعد از دوران، رابطه‌ی $\langle\alpha\rangle < x < \langle\beta\rangle < y < \langle\gamma\rangle$ برقرار است. بنابراین درخت پس از دوران هم خاصیت د.د.ج

[۱۹] rotation
[۲۰] right-rotate
[۲۱] left-rotate



شکل ۷-۱۳ دوران‌های راست‌گرد و چپ‌گرد.

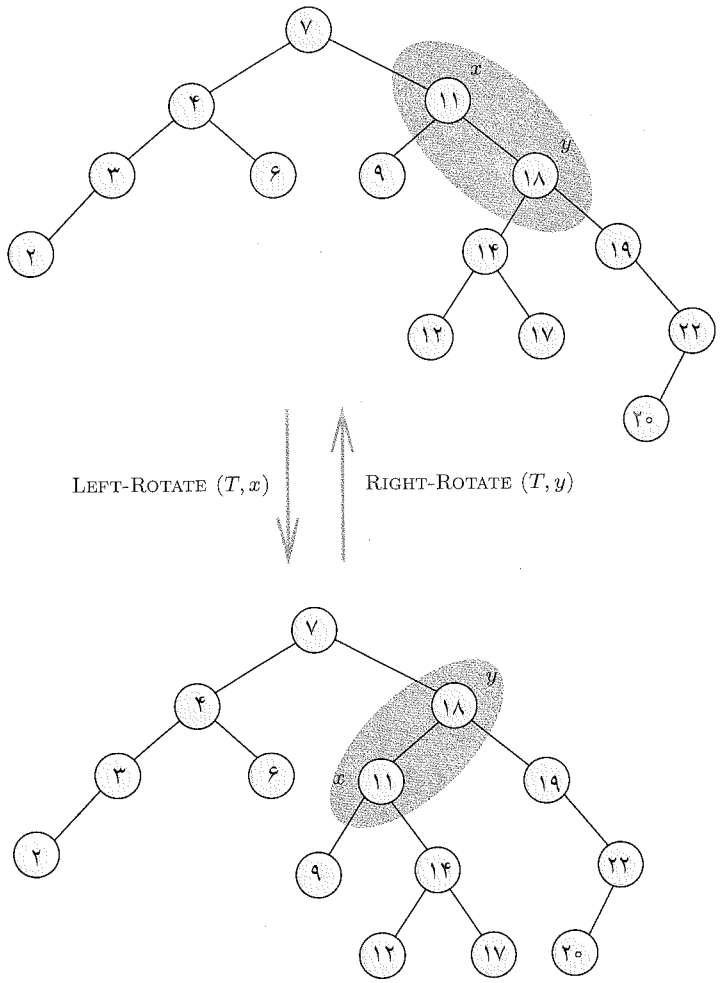
را دارد. توجه به این نکته مهم است که ارتفاع درخت α در کل درخت و پس از عمل دوران راست‌گرد، یک واحد کم و ارتفاع درخت γ یک واحد زیاد می‌شود (و در دوران چپ‌گرد این تغییرات برعکس می‌شود). بنابراین می‌توان ارتفاع زیردرخت‌ها را تنظیم کرد.

نکته‌ی ۷-۲ دو د.د.ج مختلف با تعداد عناصر یک‌سان را می‌توان فقط با انجام عمل دوران به‌هم تبدیل کرد.

(این مطلب در مسئله‌ی ۷-۳.۲ آمده است.)

رویه‌ی LEFT-ROTATE(T, x) درخت T را حول گره x به‌صورت چپ‌گرد دوران می‌دهد. برای این کار و طبق تعریف، یال خروجی «راست» x را در نظر می‌گیریم و مطابق شکل ۷-۱۳ در جهت چپ آن‌را دوران می‌دهیم. دوران راست‌گرد نیز شبیه همین کار است که در آن حول یال خروجی «چپ» x و در جهت راست دوران را انجام می‌دهیم. روشن است که این کار فقط با تغییر چند اشاره‌گر و در $\mathcal{O}(1)$ انجام می‌شود، و سایر مؤلفه‌های گره‌ها تغییری نمی‌کنند.

شکل ۷-۱۴ یک مثال واقعی از انجام LEFT-ROTATE (T, x) و نیز RIGHT-ROTATE (T, y) است.



شکل ۷-۱۴ مثالی از دوران چپ‌گرد و راست‌گرد.

```
LEFT-ROTATE (T, x)
 1  y ← right[x]
 2  right[x] ← left[y]
 3  if left[y] ≠ null
 4     then p[left[y]] ← x
 5  p[y] ← p[x]
 6  if p[x] = null
 7     then Root[T] ← y
 8     else if x = left[p[x]]
 9             then left[p[x]] ← y
10             else right[p[x]] ← y
11  left[y] ← x
12  p[x] ← y
```

درج یک عنصر در درخت قرمز-سیاه

مراحل مختلف درج یک عنصر x در یک درخت قرمز-سیاه T به‌صورت زیر است:

۱. x را به‌صورت ساده در د.د.ج T درج کن.

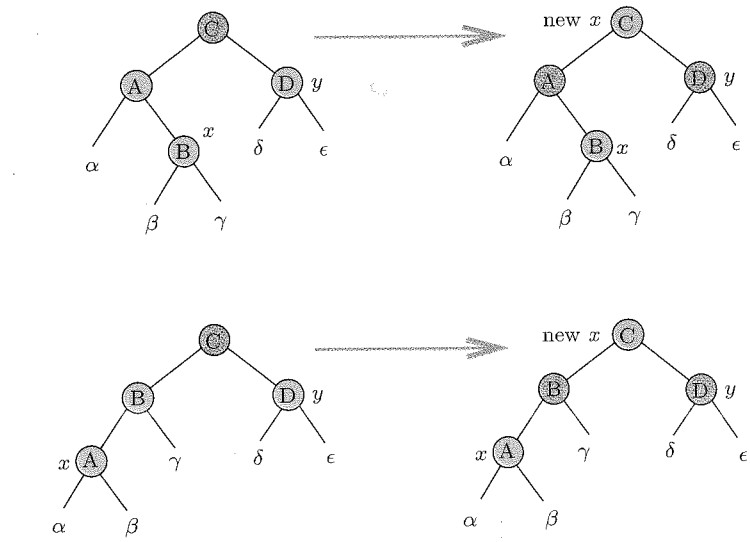
۲. رنگ x را قرمز قرار بده. در نتیجه، سیاه-ارتفاع گره‌ها تغییر نمی‌کند.

۳. اگر پدر x سیاه باشد درج به پایان رسیده است.

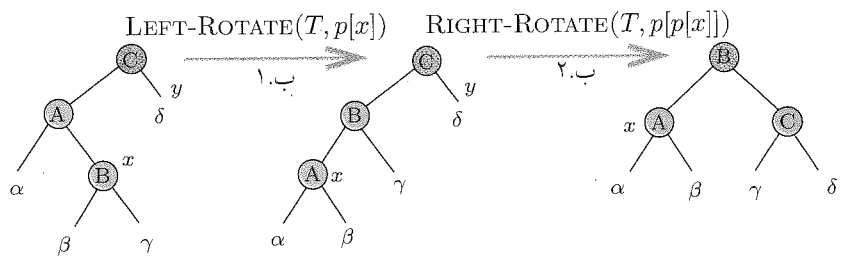
۴. اگر پدر x قرمز باشد، با استفاده از عموی x به‌نام y، کارهای زیر را انجام بده:

الف) (حالت ب.۱) اگر y قرمز باشد (شکل ۷-۱۵(الف))، پدر او (گره با برچسب C در شکل) حتماً سیاه است. در این صورت پدر x و y را از رنگ قرمز به سیاه و رنگ C را به قرمز تغییر می‌دهیم. دقت کنید که با این تغییرات، خاصیت سیاه-ارتفاع گره‌ها هنوز برقرار است. اما ممکن است پدر C قرمز باشد. اگر چنین باشد، همین الگوریتم را به‌صورت بازگشتی برروی گره new x اجرا می‌کنیم.

ب) (حالت ب.۲) اگر y سیاه باشد، مطابق شکل ۷-۱۵(ب) عمل می‌کنیم: اگر x فرزند راست پدرش باشد، با یک LEFT-ROTATE$(T, p[x])$ آن‌را به‌حالتی تبدیل می‌کنیم که x فرزند چپ پدرش است (شکل وسط) که در آن‌صورت



(الف)



(ب)

شکل ۷-۱۵ حالت‌های مختلف برای RB-INSERT.

با یک عمل RIGHT-ROTATE$(T, p[p[x]])$ و تغییر رنگ گره‌های B و C به حالت پایانی می‌رسیم.

شبه‌کد RB-INSERT (T, x) یک گره x را در قرمز-سیاه T درج می‌کند. در این رویه دستورهای مربوط به حالت‌های (الف)، (ب.۱) و (ب.۲) تفکیک شده‌اند.

```
RB-INSERT (T, x)
 1  color[x] ← red
 2  while x ≠ root[T]  and color[p[x]] = red
 3     do if p[x] = left[p[p[x]]]
 4           then y ← right[p[p[x]]]
 5                if color[y] =red
 6                 then color[p[x]] ← black                    .... الف
 7                      color[y] ← black                       .... الف
 8                      color[p[p[x]]] ← red                   .... الف
 9                      x ← p[p[x]]                            .... الف
10                 else  if x = right[p[x]]
11                            then x ← p[x]                    .... ب.۱
12                                 LEFT-ROTATE(T, x)           .... ب.۱
13                            color[p[x]] ← black              .... ب.۲
14                            color[p[p[x]]] ← red             .... ب.۲
15                            RIGHT-ROTATE(T, p[p[x]])         .... ب.۲
16           else  (same as then clause
                   with "right" and "left" exchanged)
17  color[root[T]] ← black
```

اثبات درستی: مراحل مختلف بیان‌شده در بالا، درستی الگوریتم درج را از نظر رنگ گره‌ها به خوبی نشان می‌دهد؛ یعنی در انتها پدر هیچ گره قرمز، رنگ قرمز ندارد. باید نشان دهیم که پس از درج، سیاه-ارتفاع‌ها هم به درستی رعایت می‌شوند. برای اثبات دقیق این مطلب دو حالتی را که در بالا بیان شد در نظر بگیرید:

الف) در این حالت، خاصیت سیاه-ارتفاع‌ها برای گره‌های با برچسب‌های A، B، و D برقرار است و مقدار آن تغییری نمی‌کند. مقدار سیاه-ارتفاع گره C یک واحد افزایش می‌یابد، ولی خاصیت مورد نظر برقرار است. توجه کنید که مقدار

سیاه‌ـارتفاع گره‌های پدر و اجداد C، به‌دلیل آن‌که رنگ این گره از سیاه به قرمز تبدیل شده است، تغییری نمی‌کند، و بنابراین همه درست هستند.

ب) در این حالت، اگر زیردرخت‌ها قرمز‌ـسیاه باشند، باید ریشه‌ی آن‌ها سیاه و سیاه‌ـارتفاع آن‌دو برابر باشد. پس از انجام حالت (ب.۱)، مقادیر تغییری نمی‌کنند. در صورت انجام حالت (ب.۲)، خواص مربوط به سیاه‌ـارتفاع برای گره‌های A و C به‌وضوح برقرار است. مقدار سیاه‌ـارتفاع B هم تغییری نمی‌کند. همین مطلب نیز برای همه‌ی گره‌های جد B نیز صادق است.

شکل ۷–۱۶ مراحل مختلف الگوریتم درج را برای عنصری با کلید ۴ نشان می‌دهد.

حذف یک گره از درخت قرمز‌ـسیاه

برای ساده‌سازی شرایط مرزی در پیاده‌سازی حذف، یک گره‌ی حفاظتی به‌نام $null[T]$ برای درخت T تعریف می‌کنیم که همه‌ی اشاره‌گرهای برگ تهی در درخت اصلی (از جمله پدر ریشه) به این گره اشاره کنند. این گره حاوی همان مؤلفه‌هایی است که در گره‌های معمولی درخت وجود دارد. مؤلفه‌ی رنگ این گره، سیاه فرض می‌شود ولی دیگر مؤلفه‌های آن می‌توانند مقادیر دل‌خواه بگیرند.

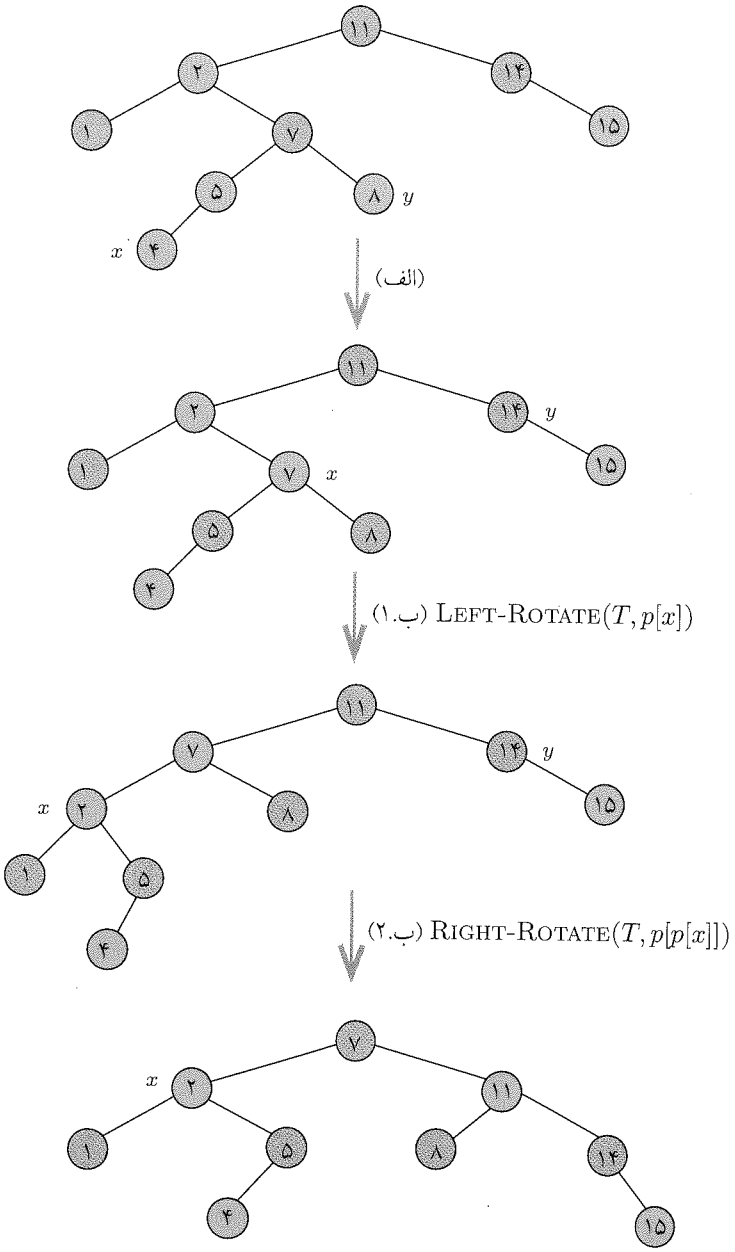
با استفاده از این گره با یک فرزند یا پدر تهی مانند یک گره عادی درخت رفتار می‌شود و این موجب ساده‌تر شدن الگوریتم حذف می‌شود.

حذف یک عنصر z در یک درخت قرمز‌ـسیاه T که این تغییر در آن انجام شده است، با فراخوانی RB-DELETE(T, z) انجام می‌شود. این رویه در زیر می‌آید.

این رویه مانند الگوریتم حذف در د.د.ج، z را از درخت حذف می‌کند. برای این کار یک گره به نام y به صورت فیزیکی از T حذف می‌شود. اگر یکی از فرزندان z برگ تهی باشد، y همان z است (سطرهای ۱ و ۲). وگرنه، y عنصر «بعدی[۲۲]» z است که در سطر ۳ و با فراخوانی TREE-SUCCESSOR (z) به‌دست می‌آید. x فرزند غیر تهی برای y است (یا $null[T]$ اگر هر دو فرزند y تهی باشند). y در سطرهای ۷ تا ۱۲ این رویه از درخت حذف می‌شود. اگر در سطر ۱۳، y همان z نبود، کلید y و دیگر مؤلفه‌هایش در z کپی می‌شود. در انتهای رویه، ما هم‌چنان به x دست‌رسی داریم.

در انتها، آن‌چه باید بررسی شود، خاصیت سیاه‌ـارتفاع‌هاست. اگر رنگ y قبل از حذف قرمز بوده باشد، حذف آن هیچ مشکلی ایجاد نمی‌کند. اما در صورت سیاه بودن باید

[۲۲] succesor



شکل ۷-۱۶ مراحل مختلف پس از درج عنصری با کلید ۴.

خاصیت قرمز-سیاه با فراخوانی رویه‌ی RB-DELETE-FIXUP(T, x) مجدداً برقرار شود.

برای حل این مشکل، در ابتدا فرض می‌کنیم که x (فرزند y) علاوه بر رنگ خود یک رنگِ سیاهِ اضافی هم دارد. با این فرض، همه‌ی سیاه-ارتفاع‌ها درست می‌شوند. هدف از رویه‌ی RB-DELETE-FIXUP آن است که این رنگ سیاهِ اضافی را به‌سمت ریشه هدایت کنیم تا در نهایت، یا جای‌گزین یک رنگ قرمز شود، یا به ریشه برسد و دور ریخته شود. برای بررسی حالت‌های مختلف این کار، به قراردادهای زیر توجه کنید:

قراردادها

- در هر مرحله x گره‌ای است که دو برچسب سیاه دارد.
- w برادر گره x خواهد بود.

```
RB-DELETE (T, z)
    ▷ گره داده‌شده‌ی z را از درخت قرمز- سیاه T حذف می‌کند
 1  if left[z] = null[T] or right[z] = null[T]
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ null[T]
 5     then x ← left[y]
 6     else x ← right[y]
    ▷ y گره‌ای است که به‌صورت فیزیکی از درخت T حذف می‌شود
 7  p[x] ← p[y]
 8  if p[y] = null[T]
 9     then root[T] ← x
10     else if y = left[p[y]]
11             then left[p[y]] ← x
12             else right[p[y]] ← x
13  if y ≠ z
14     then key[z] ← key[y]
15          if y has other fields, copy them too
16  if color[y] = Black
17     then RB-DELETE-FIXUP(T, x)
18  return y
```

حالت‌های ممکن به قرار زیر است:

حالت ۱: w قرمز است (بنابراین فرزندانش سیاه‌اند)،

حالت ۲: w سیاه است و هر دو فرزندش نیز سیاه هستند،

حالت ۳: w سیاه است و فرزند چپش قرمز و فرزند راستش سیاه است، یا

حالت ۴: w سیاه است و فرزند راستش قرمز است.

این ۴ حالت در شکل ۷-۱۷ نشان داده شده‌اند که در آن گره‌ها یا سیاه هستند یا قرمز که در شکل مشخص‌اند، یا این‌که دارای رنگ c یا c' هستند که آن‌هم با برچسب مشخص شده است.

راه‌حل زیر برای هر حالت، مشکل رنگ سیاه اضافی را حل خواهد کرد:

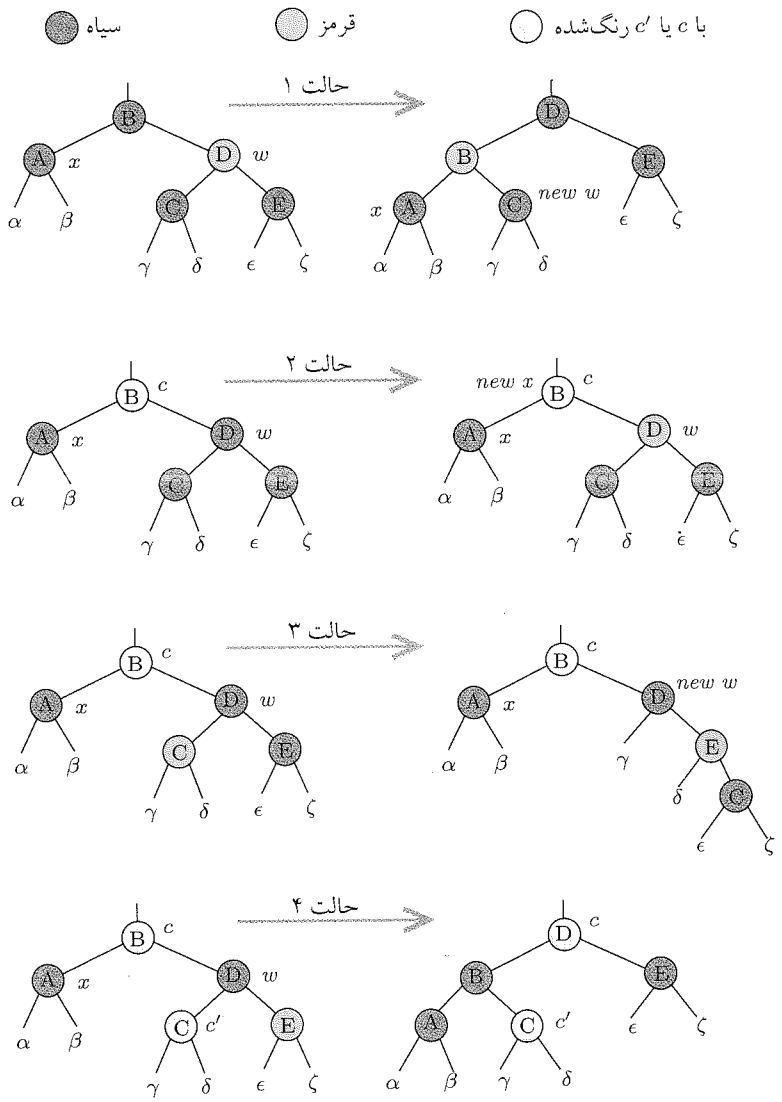
حالت ۱: در این حالت با تعویض رنگ گره‌های D و B و انجام یک دوران چپ‌گرد LEFT-ROTATE $(T, p[x])$ و با نام‌گذاری برادر جدید x، مسئله به یکی از حالت‌های ۲، ۳ یا ۴ تبدیل می‌شود. دقت کنید که با احتساب رنگ سیاهِ اضافی در x، همه‌چیز درست است.

حالت ۲: w را قرمز می‌کنیم و رنگ سیاه اضافی در x را به‌پدرش که در حال حاضر رنگ c دارد منتقل می‌کنیم. اگر c قرمز باشد به سیاه تغییر رنگ می‌دهد و کار تمام است. وگرنه، الگوریتم به‌همین صورت دنبال می‌شود. اگر پدر x ریشه باشد هم کار به پایان رسیده است.

حالت ۳: این حالت با تعویض رنگ گره‌های C و D و انجام یک دوران راست‌گرد RIGHT-ROTATE(T, w) به حالت ۴ تبدیل می‌شود. دقت کنید که اگر رنگ سیاه اضافی x را در نظر بگیریم، سیاه‌ـارتفاع‌ها هم‌چنان درست‌اند.

حالت ۴: در این حالت می‌توانیم با تغییر چند رنگ و یک دوران چپ‌گرد LEFT-ROTATE $(T, p[x])$ بدون آن‌که به خواص درخت لطمه‌ای وارد شود، رنگ سیاه اضافیِ x را حذف کنیم. در انتهای این حالت، حلقه به‌پایان می‌رسد.

رویه‌ی RB-DELETE-FIXUP(T, x) کارهای توضیح داده‌شده را حداکثر در زمانی متناسب با ارتفاع درخت انجام می‌دهد.



شکل ۷-۱۷ حالت‌های مختلف ایجاد شده در مراحل حذف در درخت قرمز-سیاه.

```
RB-DELETE-FIXUP (T, x)
 1  while x ≠ root[T] and color[x] =black
 2    do if x = left[p[x]]
 3       then w ← right[p[x]]
 4            if color[w] =red
 5              then color[w] ← black                          .... حالت ۱
 6                   color[p[x]] ← red                         .... حالت ۱
 7                   LEFT-ROTATE (T, p[x])                     .... حالت ۱
 8                   w ← right[p[x]]                           .... حالت ۱
 9            if color[left[w]] =black and color[right[w]] =black
10              then color[w] ← red                            .... حالت ۲
11                   x ← p[x]                                  .... حالت ۲
12              else      if color[right[w]] =black
13                        then color[left[w]] ← black           .... حالت ۳
14                             color[w] ← red                   .... حالت ۳
15                             RIGHT-ROTATE(T, w)               .... حالت ۳
16                             w ← right[p[x]]                  .... حالت ۳
17                        color[w] ← color[p[x]]                .... حالت ۴
18                        color[p[x]] ← black                   .... حالت ۴
19                        color[right[w]] ← black               .... حالت ۴
20                        LEFT-ROTATE (T, p[x])                 .... حالت ۴
21                        x ← root[T]                           .... حالت ۴
22              else      (same as then with right and left exchanged.)
23  color[x] ← black
```

تمرین‌های زیربخش ۷-۳-۱

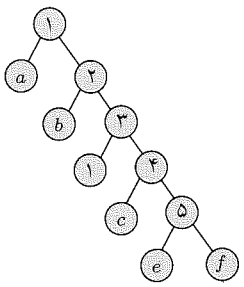
۷-۳-۱.۱ در یک درخت قرمز-سیاه که در ابتدا تهی است، اعمال درج و حذف زیر را انجام می‌دهیم. درخت حاصل پس از هر عمل را رسم‌کنید.

الف) درج این عناصر (به ترتیب از چپ به راست): `A, F, E, D, C, B, G, H, I`

ب) حذف این عناصر از درخت قسمت (الف): `A, F, I, H, B, C, G`

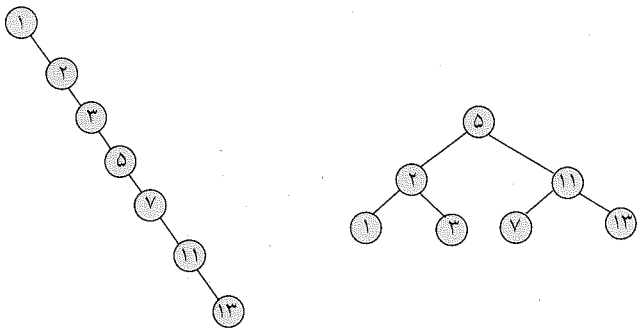
۷-۳-۱.۲ به‌طور کامل نشان دهید که هر دو درخت قرمز-سیاه با n رأس را می‌توان با حداکثر $2n - 2$ دوران به‌هم تبدیل کرد.

۷-۳-۳.۱ د.د.ج زیر داده شده است:



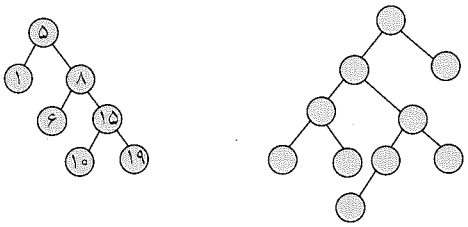
کم‌ترین تعداد دوران‌هایی را مشخص کنید که ارتفاع درخت را از ۵ به ۳ تقلیل دهد؟

۷-۳-۴.۱ چه دنباله‌ای از دوران‌ها درخت چپ را به درخت راست تبدیل می‌کند؟



زمان اجرای چنین الگوریتمی برای زنجیره‌ی $n = 2^k - 1$ تایی چیست؟

۷-۳-۵.۱ هر یک از درخت‌های زیر را به‌صورت قرمز-سیاه رنگ‌آمیزی کنید (یا استدلال کنید که چنین رنگ‌آمیزی امکان ندارد).



درخت قرمز-سیاه حاصل از درج ۱۶ در درخت سمت چپ را رسم کنید.

۷-۳-۶.۱ فرض کنید یک جست‌وجو برای پیدا کردن یک عنصر خاص در یک درخت قرمز-سیاه پس از ۲۰ شاخه از ریشه‌ی درخت به‌صورت ناموفق خاتمه می‌یابد. بیشینه و کمینه‌ی تعداد شاخه‌هایی که در هر جست‌وجوی ناموفق در این درخت ملاقات می‌شوند چقدر است؟

۷-۳-۷.۱ الگوریتمی بنویسید تا با زمان بهینه یک د.د.ج را رنگ کند تا قرمز-سیاه شود یا این‌که تشخیص دهد این کار ممکن نیست.

۷-۳-۸.۱ با فرض آن‌که در ساختار درخت قرمز-سیاه مؤلفه‌ی «پدر» برای گره‌ها وجود ندارد، رویه‌های RB-INSERT و RB-DELETE را در زمان بهینه بنویسید.

۷-۳-۹.۱ مطابق الگوریتم بیان‌شده در این بخش، اگر یک کلید را به یک درخت قرمز-سیاه اضافه و بلافاصله آن‌را حذف کنیم، آیا درخت حاصل با درخت اولیه برابر است؟ اگر پاسخ شما مثبت است ثابت کنید، وگرنه، مثال نقض بیاورید.

۷-۳-۱۰.۱ ثابت کنید در درخت قرمز-سیاه طولانی‌ترین مسیر از گره x به برگ نواده‌ی آن، دست‌کم دو برابر کوتاه‌ترین مسیر از همان گره به برگ نواده‌ی آن می‌باشد.

۷-۳-۱۱.۱ فرض کنید که یک جست‌وجو برای پیدا کردن یک عنصر خاص در یک درخت قرمز-سیاه پس از ۲۰ شاخه از ریشه‌ی درخت به‌صورت ناموفق متوقف می‌شود. حداکثر و حداقل تعداد شاخه‌هایی که در هر جست‌وجوی ناموفق در این درخت ملاقات می‌شوند چقدر است؟

⋆ ۷-۳-۱۲.۱ یک د.د.ج را در نظر بگیرید که به هر عنصر x از آن یک مؤلفه‌ی $size[x]$ اضافه کرده‌ایم که تعداد عناصر موجود در زیردرخت به ریشه‌ی x را ثبت می‌کند. فرض کنید $\frac{1}{2} \leq \alpha < 1$ یک ثابت است. می‌گوییم یک گره x، α-متوازن است اگر $size[left[x]] \leq \alpha.size[x]$ و $size[right[x]] \leq \alpha.size[x]$. کل درخت هم α-متوازن است اگر هر گره در آن مطابق تعریف بالا α-متوازن باشد.

الف) یک گره x در یک د.د.ج معمولی (با مؤلفه‌ی $size$، ولی نه لزوماً متوازن) داده شده است. نشان دهید که چگونه می‌توان زیر درخت به ریشه‌ی x را بازسازی کرد به‌طوری‌که آن زیردرخت $\frac{1}{2}$-متوازن شود. الگوریتم شما باید در زمان $\Theta(size[x])$ اجرا شود و حداکثر حافظه‌ی کمکی‌ای به‌همین اندازه مصرف کند.

ب) به‌طور کامل نشان دهید که انجام عمل جست‌وجو در یک د.د.ج α-متوازن در بدترین حالت به‌اندازه‌ی $O(\log_{1/\alpha} n)$ هزینه دارد.

پ) برای درج و حذف از این درخت باید اگر زیردرختی از حالت α-متوازن خارج شود، آن زیردرخت مطابق بند (الف) بازسازی شود. ثابت کنید که هزینه‌ی درج و حذف یک عنصر در یک درخت α-متوازن به‌صورت سرشکنی $O(\lg n)$ است. فرض کنید α ثابت و اکیداً بیش‌تر از $\frac{1}{2}$ است.

۷-۳-۲ گسترش درخت قرمز-سیاه: درخت مرتبه‌ی آماری

در این بخش دو داده‌ساختار که گسترش‌یافته‌ی درخت قرمز-سیاه هستند ارائه می‌شود. اولین ساختار، «درخت مرتبه‌ی آماری[۲۳]» است که مجموعه‌ای از n عنصر را که به‌صورت پویا به آن درج یا از آن حذف می‌شوند، طوری پیاده‌سازی می‌کند که اعمال مرتبه‌ی آماری (یافتن مرتبه‌ی یک عنصر، یا یافتن عنصری با مرتبه‌ی خاص) را در هر زمان بتوان در $O(\lg n)$ انجام داد. دومین داده‌ساختار «درخت بازه[۲۴]» است که بر روی تعدادی عناصر از نوع «بازه» که به‌صورت پویا وارد یا خارج می‌شوند ساخته می‌شود. با این داده‌ساختار می‌توان بازه‌ای را که با بازه‌ی داده‌شده هم‌پوشانی دارد، در زمان لگاریتمی نسبت به تعداد بازه‌های موجود، به‌دست آورد.

هدف، طراحی داده‌ساختاری است که بر روی تعدادی عنصر پویا و با کلیدهای متفاوت (که در طول زمان به این داده‌ساختار اضافه یا از آن حذف می‌شوند) ساخته شود، و علاوه بر اعمال درج، حذف و جست‌وجو، بتوان مرتبه‌ی یک عنصر، و نیز عنصری با مرتبه‌ی داده‌شده را در زمان لگاریتمی به‌دست آورد.

درخت مرتبه‌ی آماری، یک درخت قرمز-سیاه است که به‌صورت زیر گسترش داده شده و برای انجام کارای این اعمال مناسب است. در این گسترش، هر گره x علاوه بر اطلاعات مربوط به خاصیت قرمز-سیاه یک مؤلفه‌ی $size[x]$ هم دارد که تعداد عناصر موجود در زیردرختی به‌ریشه‌ی x را نشان می‌دهد. یعنی،

$$size[x] = size[left[x]] + size[right[x]] + ۱.$$

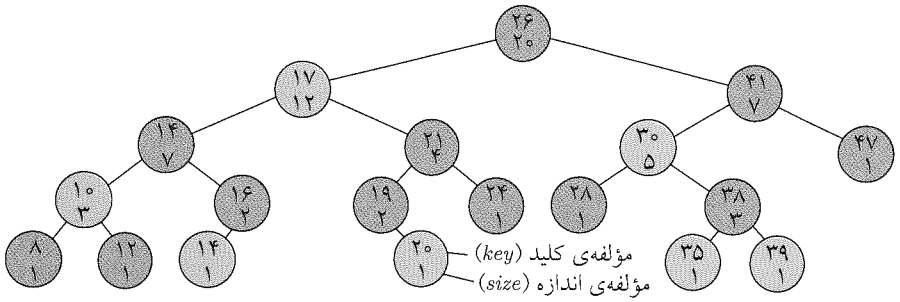
شکل ۷-۱۸ یک نمونه از درخت مرتبه‌ی آماری را نشان می‌دهد.

یافتن عنصری با مرتبه‌ی داده‌شده

رویه‌ی OS-SELECT(x, i)، در زیردرختی به ریشه‌ی x عنصری با iامین کلید بین آن عناصر (یا عنصر با مرتبه‌ی i) را به‌دست می‌آورد. این الگوریتم با استفاده از مؤلفه‌ی $size$ با هزینه‌ای متناسب با ارتفاع زیردرخت به‌ریشه‌ی x این کار را انجام می‌دهد.

الگوریتم ابتدا مرتبه‌ی ریشه‌ی x یا r را که برابر با تعداد عناصر در زیردرخت چپ x به‌اضافه‌ی ۱ است، به‌دست می‌آورد. اگر $i = r$ در آن‌صورت خود x جواب مطلوب

[۲۳] order statistic tree
[۲۴] interval tree



شکل ۷-۱۸ یک درخت مرتبه‌ی آماری.

است، وگرنه، اگر $i < r$، باید iامین عنصر در زیردرخت چپ را پیدا کنیم، و اگر $i > r$، $i - r$امین عنصر زیردرخت راست جواب است. این دو فراخوانی به‌صورت بازگشتی انجام می‌شوند. نشان می‌دهیم که با تغییرات داده‌شده، می‌توان خاصیت قرمز-سیاه درخت را همیشه حفظ کرد. به این دلیل، زمان این الگوریتم $O(\lg n)$ است.

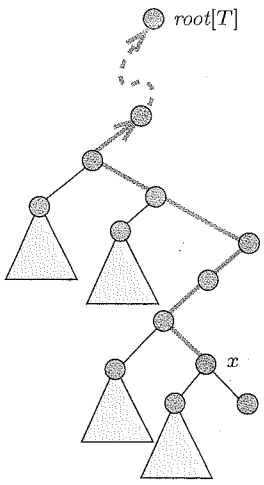
```
OS-SELECT (x, i)
1  r ← size[left[x]] + 1
2  if i = r
3     then return x
4     else if i < r
5             then return OS-SELECT(left[x], i)
6             else return OS-Select(right[x], i − r)
```

یافتن مرتبه‌ی یک عنصر داده‌شده

رویه‌ی OS-RANK(T, x) مرتبه‌ی عنصر x را در درخت مرتبه‌ی آماری T به‌دست می‌آورد. الگوریتم با استفاده از مؤلفه‌ی «پدر» مسیری مثل $x = a_1 \to a_2 \to \cdots \to a_k = root[T]$ را از x به ریشه‌ی درخت T پیمایش می‌کند. در این مسیر اگر a_i فرزند راست a_{i+1} باشد، اندازه‌ی زیر درخت چپ a_{i+1} را می‌شمارد و با اندازه‌ی زیردرخت چپ x جمع می‌کند. با توجه به خاصیت د.د.ج، عدد حاصل‌جمع، مرتبه‌ی آماری x است.

```
OS-RANK (T, x)
1  r ← size[left[x]] + 1
2  y ← x
3  while y ≠ root[T]
4      do if y = right[p[y]]
5             then r ← r + size[left[p[y]]] + 1
6          y ← p[y]
7  return r
```

شکل ۷–۱۹ نحوه‌ی اجرای این الگوریتم را نشان می‌دهد. بدیهی است که هزینه‌ی این الگوریتم متناسب با ارتفاع درخت است.



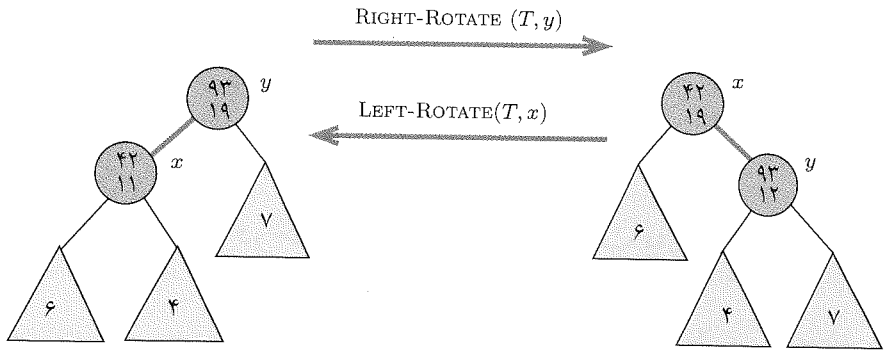
شکل ۷–۱۹ یافتن مرتبه‌ی عنصر x در یک درخت مرتبه‌ی آماری.

نگه‌داشت اندازه‌ی زیردرخت‌ها

دو رویه‌ی فوق به‌شرطی در زمان $O(\lg n)$ کار می‌کنند که به‌درستی بتوانیم مؤلفه‌ی اندازه‌ی زیردرخت‌ها را به‌هنگام انجام درج و حذف در درخت قرمز–سیاه با همان هزینه‌ی قبل به‌هنگام کنیم.

در انجام اعمال حذف و درج، اگر عمل دوران انجام نشود، به‌راحتی می‌توانیم در مسیر گره درج‌شده و یا گره‌ای که حذف می‌شود تا ریشه‌ی درخت، مؤلفه‌ی اندازه را به‌ترتیب یک واحد زیاد یا کم کنیم. بنابراین در حالت کلی کافی است نشان دهیم که عمل دوران را هم با کمی تغییر می‌توان طوری انجام داد که مؤلفه‌ی اندازه هم به‌درستی حساب و نگه‌داشته شود.

شکل ۷-۲۰ این‌کار را برای دوران‌های راست‌گرد و چپ‌گرد نشان می‌دهد.



شکل ۷-۲۰ نگه‌داشت اندازه‌های زیردرخت‌ها در عمل دوران.

مثلاً در رویه‌ی LEFT-ROTATE(T, x)، دستورهای زیر را باید اضافه کنیم:

$size[y] \leftarrow size[x]$

$size[x] \leftarrow size[left[x]] + size[right[x]] + ۱$

تمرین‌های زیربخش ۷-۳-۲

۷-۳-۱.۲ فرض کنید $A[۱ \ldots n]$ آرایه‌ای از n عدد متمایز باشد. زوج (i, j) را یک وارونگی می‌گوییم اگر $i < j$ و $A[i] > A[j]$.

الف) با استفاده از درخت مرتبه‌ی آماری، الگوریتمی از $\mathcal{O}(n \lg n)$ برای محاسبه‌ی تعداد وارونگی‌ها ارائه دهید.

ب) بدون استفاده از داده‌ساختار درخت مرتبه‌ی آماری، الگوریتمی از $\mathcal{O}(n \lg n)$ برای محاسبه‌ی تعداد وارونگی‌ها ارائه دهید.

۷-۳-۲.۲ فرض کنید n سیم موجودند. این سیم‌ها به‌صورتی روی یک دایره چیده شده‌اند که دو سر

هر سیم بر روی محیط دایره قرار دارد. الگوریتمی از $O(n \lg n)$ ارائه دهید تا با گرفتن مختصات دو سرِ تمام سیم‌ها، تعداد سیم‌هایی را که با هم دوبه‌دو تقاطع دارند بیابد.

۷-۳-۳.۲ در یک درخت مرتبه‌ی آماری با n عضو، عنصر x مفروض است. الگوریتمی از مرتبه‌ی $O(\lg n)$ ارائه دهید تا iاُمین نواده‌ی x را پیدا کند. فرض کنید که پیمایش درخت خطی است.

⋆ **۷-۳-۴.۲** آرایه‌ی n تایی مرتب A داده شده است. k درایه از این آرایه را به مقادیر دل‌خواه تغییر می‌دهیم تا آرایه‌ی A به‌دست آید. فرض کنید آرایه‌ی بیتی B هم داده شده است که درایه‌ی $B[i]$ برابر ۱ است اگر $A[i]$ تغییر کرده باشد، وگرنه صفر است. یک الگوریتم با زمان اجرای $O(n \lg k)$ برای مرتب‌سازی مجدد A ارائه دهید. حافظه‌ی اضافی مورد نیاز نباید از $O(n)$ بیش‌تر باشد. (بخش کمی از نمره به راه‌حل $O(nk)$ تعلق خواهد گرفت.)

۷-۳-۳ گسترش درخت قرمز-سیاه: درخت بازه

درخت بازه داده‌ساختاری برای کار با بازه‌هاست. یک «بازه‌ی بسته[۲۵]» که با $[t_1, t_2]$ نشان داده می‌شود، یک زوج مرتب از اعداد حقیقی t_1 و t_2 است که $t_1 \le t_2$. بازه‌ی $[t_1, t_2]$ شامل همه‌ی اعداد حقیقی $\{t \in R : t_1 \le t \le t_2\}$ است. یک «بازه‌ی باز[۲۶]» و یا «بازه‌ی نیمه‌باز[۲۷]» به‌ترتیب شامل هر دو نقطه یا یکی از نقطه‌های انتهایی نیست.

برای یک بازه‌ی $i = [t_1, t_2]$، $low[i] = t_1$ را نقطه‌ی ابتدایی و $high[i] = t_2$ را نقطه‌ی انتهایی بازه می‌نامیم. دو بازه‌ی i و i' نسبت به‌هم سه حالت مختلف زیر را دارند:

۱. i و i' هم‌پوشانی دارند،

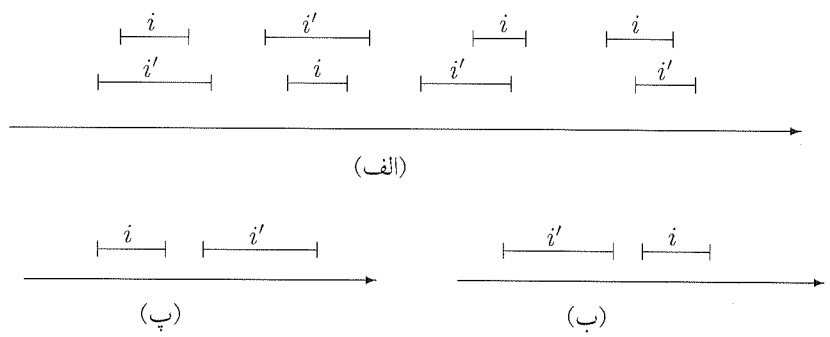
۲. $high[i] < low[i']$،

۳. $high[i'] < low[i]$.

شکل ۷-۲۱ این سه‌حالت را نشان می‌دهد. هم‌پوشانی دو بازه نیز به چهار حالت ممکن است که در همین شکل نشان داده شده است.

داده‌ساختار مطلوب را «درخت بازه» می‌نامیم که باید اعمال زیر را به‌صورت کارا پاسخ دهد. در این‌جا، T درخت بازه، x یک عنصر از آن و $int[x]$ بازه‌ی موجود در x است. هم‌چنین، i یک بازه‌ی ورودی است.

[۲۵] closed interval
[۲۶] open interval
[۲۷] semi-open interval



شکل ۷-۲۱ (الف) چهار حالت برای هم‌پوشانی بازه‌های i و i'. (ب) و (پ) حالت‌های ناهم‌پوشان را نشان می‌دهند.

INTERVAL-INSERT(T, x): درج عنصر x در درخت بازه‌ی T،

INTERVAL-DELETE(T, x): حذف عنصر x از درخت بازه‌ی T،

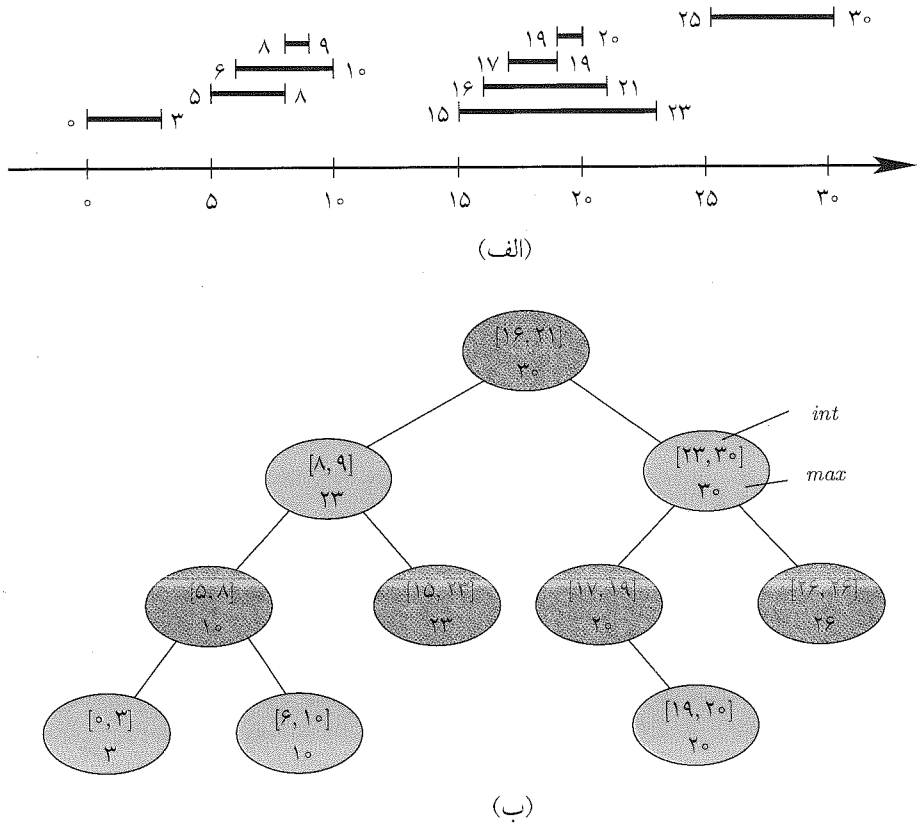
INTERVAL-SEARCH(T, i): اشاره‌گر به یک عنصر x از T را باز می‌گرداند که بازه‌ی آن، یا $int[x]$، با بازه‌ی i هم‌پوشانی دارد. اگر چنین عنصری در درخت نباشد، رویه مقدار «تهی» را بر می‌گرداند.

داده‌ساختار درخت بازه

داده‌ساختار مناسب برای انجام اعمال ذکرشده، یک درخت قرمز-سیاه است که عناصر آن بازه‌های ورودی هستند. کلید هر عنصر x نقطه‌ی شروع $int[x]$ (یا $low[int[x]]$) است. هم‌چنین، مؤلفه‌ی $max[x]$ به هر گره x اضافه می‌شود که نشان‌دهنده‌ی بیش‌ترین مقدار نقطه‌ی بالای کلیه‌ی بازه‌هایی است که در زیردرختی به ریشه‌ی x قرار دارند. شکل ۷-۲۲(ب) درخت بازه‌ی ساخته‌شده برای بازه‌های نشان داده شده است.

درج و حذف بازه‌ها

اعمال درج و حذف مانند درخت قرمز-سیاه انجام می‌شود. می‌توان دید که مانند درخت مرتبه‌ی آماری مؤلفه‌ی max عناصر را می‌توان محاسبه و ثبت کرد. عمل دوران هم می‌تواند این مقادیر را طوری اصلاح کند که مقدار این مؤلفه به‌درستی محاسبه شود.



شکل ۷-۲۲ (الف) بازه‌ها و (ب) درخت بازه‌ی مربوط به بازه‌های (الف).

یافتن بازه‌ی هم‌پوشان

رویه‌ی INTERVAL-SEARCH(T, i) در صورت وجود، یک بازه از درخت T را که با بازه‌ی i هم‌پوشانی دارد به‌دست می‌آورد. الگوریتم از ریشه‌ی درخت، یا x شروع می‌کند و در مسیری در درخت به جلو می‌رود تا گره مطلوبی را پیدا کند که بازه‌ی آن با i هم‌پوشانی دارد، یا این‌که به اشاره‌گر تهی برسد که در آن‌صورت مسئله جواب ندارد.

قبل از ارائه‌ی اثبات درستی، روشن است که این الگوریتم از مرتبه‌ی $O(\lg n)$ است، چرا که در حلقه هر بار یا به فرزند راست و یا به فرزند چپ یک عنصر می‌رود و این به‌معنی آن است که هزینه، متناسب با ارتفاع درخت است.

```
INTERVAL-SEARCH (T, i)
1  x ← root[T]
2  while x ≠ null and i does not overlap int[x]
3      do if left[x] ≠ null and max[left[x]] ≥ low[i]
4            then x ← left[x]
5            else x ← right[x]
6  return x
```

به‌عنوان مثال، اگر این الگوریتم را بر روی درخت شکل ۷-۲۲ و مقدار $i = [۲۲, ۲۵]$ اجرا کنیم، ابتدا x ریشه‌ی درخت است، اما به‌دلیل آن‌که $low[i] < max[left[x]]$، x برابر فرزند چپ ریشه می‌شود که حاوی بازه‌ی $[۸, ۹]$ است که با i هم‌پوشانی ندارد. در مرحله‌ی بعد، چون $low[i] > max[left[x]]$، به فرزند راست x که حاوی بازه‌ی $[۱۵, ۲۳]$ است می‌رویم. چون این بازه با i هم‌پوشانی دارد آن‌را به‌عنوان پاسخ الگوریتم برمی‌گردانیم. توجه کنید که بازه‌ی i ممکن است با چند بازه‌ی T هم‌پوشانی داشته باشد که یک جواب آن به‌دست می‌آید.

به‌عنوان مثالی که پاسخ آن **null** است، بازه‌ی $i = [۱۱, ۱۴]$ را در نظر بگیرید. در این‌صورت ابتدا به فرزند چپ ریشه (y) و سپس به فرزند راست y می‌رویم و سپس به گره z می‌رسیم که بازه‌ی $[۶, ۱۰]$ را دارد. چون این بازه با i هم‌پوشانی ندارد و فرزند راست آن **null** است، پاسخ همان **null** است که درست می‌باشد.

قضیه‌ی ۷-۲ در هر تکرار حلقه‌ی **while** در الگوریتم INTERVAL-SEARCH(T, i)،

۱. اگر سطر ۴ اجرا شود و جست‌وجو به فرزند چپ x برود، یا زیردرخت چپ x شامل بازه‌ای است که با i هم‌پوشان است یا هیچ بازه‌ای در زیردرخت راست x وجود ندارد که با i هم‌پوشانی داشته باشد،

۲. اگر سطر ۵ اجرا شود و جست‌وجو به فرزند راست x برود، هیچ بازه‌ای در زیر درخت چپ x وجود ندارد که با i هم‌پوشانی داشته باشد.

اثبات: ابتدا بند ۲ را ثابت می‌کنیم. سطر ۵ هنگامی اجرا می‌شود که یا $left[x] =$ **null** و یا $max[left[x]] < low[i]$. در هر دو حالت امکان ندارد بازه‌ای در زیردرخت چپ x باشد که

با i هم‌پوشانی داشته باشد. بنابراین، در این صورت، بازه‌ی هم‌پوشان با i تنها ممکن است در زیردرخت راست x باشد.

در مورد بند ۱، می‌دانیم که $max[left[x]] \geq low[i]$. بازه‌ی j در زیردرخت چپ x را در نظر بگیرید که $high[j] = max[left[x]]$. بازه‌ی i یا با j هم‌پوشانی دارد یا خیر. اگر داشته باشد، بند ۱ درست است و گرنه حتماً $high[i] < low[j]$. پس امکان ندارد i با بازه‌ای در زیردرخت راست x هم‌پوشانی داشته باشد، و حکم ثابت می‌شود. □

تمرین‌های زیربخش ۷-۳-۳

۷-۳-۱.۳ الگوریتم INTERVAL-SEARCH را برای بازه‌های باز بنویسید.

۷-۳-۲.۳ در یک درخت بازه می‌خواهیم کلیه‌ی بازه‌هایی را که با بازه‌ی ورودی i هم‌پوشانی دارند پیدا کنیم. برای این‌کار الگوریتمی از مرتبه‌ی $O(min(n, k \lg n))$ ارائه دهید که در آن k تعداد بازه‌های جواب است.

۷-۳-۳.۳ برای درخت بازه‌ی معرفی شده، رویه‌ی INTERVAL-SEARCH2(T, i) را به‌گونه‌ای بنویسید که بازه‌های دقیقاً برابر با بازه i را بیابد و این کار را در زمان $O(\lg n)$ انجام دهد.

۷-۳-۴.۳ عدد جالب یک مجموعه برابر با کمینه‌ی اختلاف دوبه‌دوی عناصر می‌باشد. برای مثال، عدد جالب مجموعه‌ی $A = \{۱, ۲۲, ۵, ۹, ۱۸\}$ برابر با ۴ است.

الف) الگوریتمی با حافظه‌ی $O(n \lg n)$ و زمان $O(\lg n)$ برای یافتن عدد جالب یک مجموعه‌ی n عضوی طراحی کنید.

ب) الگوریتمی با حافظه‌ی $O(n \lg n)$ و زمان $O(۱)$ برای یافتن عدد جالب یک مجموعه‌ی n عضوی طراحی کنید.

۷-۳-۵.۳ الگوریتمی طراحی کنید که در یک درخت بازه و بازه‌ی ورودی a، از میان بازه‌هایی که با i هم‌پوشانی دارند، بازه‌ی با کم‌ترین نقطه‌ی شروع (یا نقطه‌ی پایان) را به‌دست آورد.

۷-۳-۶.۳ n بازه با مختصات صحیح روی محور اعداد $۱ \ldots m$ داده شده‌اند. می‌خواهیم در صورت وجود، کمینه‌ی تعداد بازه‌ها را انتخاب کنیم به‌طوری‌که هر نقطه با مختصات صحیح روی این محور دست‌کم در k بازه آمده باشد $(۱ \leq k \leq n)$. برای حل این مسئله، الگوریتمی از $O(n \lg m \lg n)$ ارائه دهید. توجه کنید که مقایسه‌ی دو عدد صحیح بین ۱ و m به‌اندازه‌ی $O(\lg m)$ طول می‌کشد.

۷-۳-۷.۳ می‌خواهیم داده‌ساختاری برای نگه‌داری نقطه‌ی با بیش‌ترین هم‌پوشانی در مجموعه‌ای از بازه‌های بسته ایجاد کنیم. طبیعتاً چنین نقطه‌ای همواره وجود دارد و همیشه در یک انتهای بازه قرار می‌گیرد. (ثابت کنید!) یک داده‌ساختار مشابه درخت بازه ایجاد کنید که اعمال INTERVAL-INSERT، INTERVAL-DELETE و FIND-MAXIMUM-OVERLAP را در زمان مناسبی انجام دهد.

۷-۳-۴ درخت اِی.وی.اِل

درخت اِی.وی.اِل[۲۸] اولین درخت دودویی جست‌وجو است که مانند درخت قرمز-سیاه خود را متوازن نگه می‌دارد و ارتفاع آن همیشه لگاریتمی است. این درخت در سال ۱۹۶۲ توسط «جی.ام. اندرسن-ولسکی[۲۹]» و «ای.ام. لندیس[۳۰]» پیشنهاد شد و نام خود را از حروف اول نام این طراحان گرفت.

تعریف ۷-۱ درخت اِی.وی.اِل درخت دودویی جست‌وجویی است که اختلاف ارتفاع‌های دو زیردرخت هر گره آن حداکثر یک واحد است.

در این تعریف ارتفاع یک زیردرخت تهی ۱-، و ارتفاع یک گره تک، هم‌چون گذشته ۰ است.

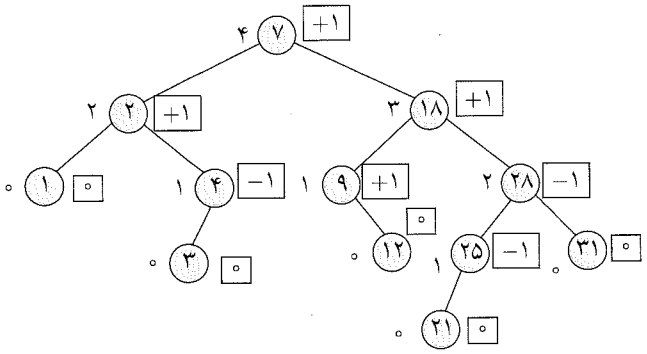
شکل ۷-۲۳ یک درخت اِی.وی.اِل با ۱۲ گره را نشان می‌دهد. در این شکل ارتفاع هر گره در کنار آن نوشته شده است. کلید گره‌ها خاصیت د.د.ج دارند. برای پیاده‌سازی این درخت یک مؤلفه‌ی «توازن» به هر گره اضافه می‌کنیم که برابر است با اختلاف ارتفاع زیردرخت راست با زیردرخت چپ. بنابراین مؤلفه‌ی توازن مقادیر ۰، ۱+ یا ۱- را خواهد گرفت و در پیاده‌سازی به دو بیت اضافی در هر گره نیاز داریم. در شکل ۷-۲۳ مقدار توازن برای هر گره در یک جعبه در کنار آن نشان داده شده است.

شکل ۷-۲۴ یک درخت غیرِ اِی.وی.اِل را نشان می‌دهد که در آن یک گره که با پیکان مشخص شده است، این خاصیت را نقض می‌کند، چون عدد توازن آن ۲+ است. مانند این شکل و در ادامه، درخت‌های اِی.وی.اِل را فقط با عدد توازن در داخل هر گره نشان می‌دهیم.

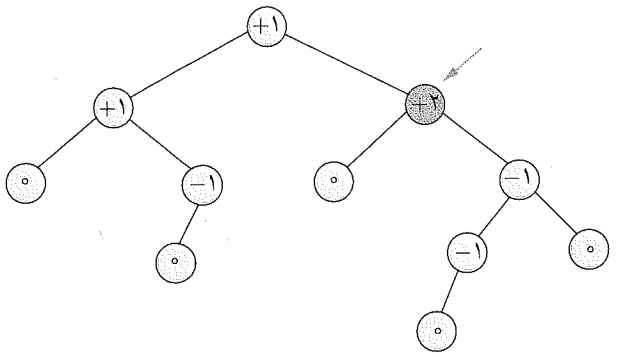
ارتفاع درخت اِی.وی.اِل

برای محاسبه‌ی ارتفاع بیشینه‌ی یک درخت اِی.وی.اِل با n گره، ابتدا $M(h)$، یعنی کمینه‌ی تعداد گره‌های یک درخت اِی.وی.اِل با ارتفاع h را محاسبه می‌کنیم. شکل ۷-۲۵ چهار درخت اِی.وی.اِل با ارتفاع‌های ۰ تا ۴ و با کم‌ترین تعداد گره‌ها را نشان می‌دهد. به این

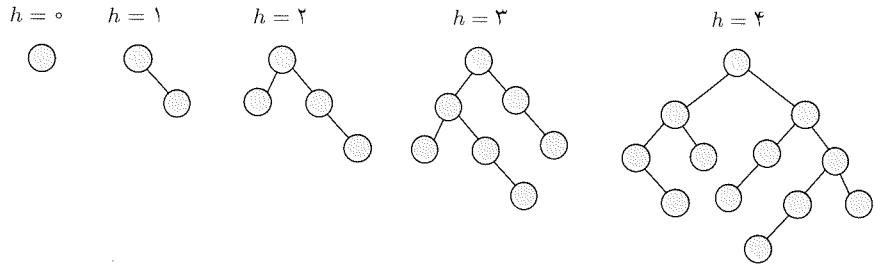
[۲۸] AVL-tree
[۲۹] G.M. Anderson-Velsky
[۳۰] E.M. Landis



شکل ۷-۲۳ مثالی از یک درخت اِی.وی.اِل. کلیدهای عناصر در داخل گره‌ها، ارتفاع هر گره در سمت چپ آن و «توازن» هر گره در سمت راست آن در داخل جعبه نشان داده شده است.



شکل ۷-۲۴ مثالی از یک درختِ غیر اِی.وی.اِل. گره نشان‌داده شده خاصیت اِی.وی.اِل را نقض می‌کند.



شکل ۷-۲۵ درخت‌های اِی.وی.اِل «خلوت»؛ با کم‌ترین تعداد گره‌ها و با ارتفاع‌های ۰ تا ۴.

درخت‌ها «خلوت[۳۱]» می‌گوییم. بدیهی است که این درخت‌ها تک نیستند.

اگر ارتفاع درخت خلوت h باشد، حتماً ارتفاع یکی از زیردرخت‌های آن $h - ۱$ است و در آن‌صورت ارتفاع زیردرخت دوم نمی‌تواند از $h - ۲$ کم‌تر باشد. برای آن‌که تعداد گره‌ها کمینه شود، باید زیردرخت‌ها هم کم‌ترین ارتفاع را داشته باشند، و نیز هر یک خلوت باشد (کم‌ترین تعداد گره را داشته باشد). پس، رابطه‌ی بازگشتی زیر برقرار است:

$$\begin{aligned} M(۰) &= ۱ \\ M(۱) &= ۲ \\ M(h) &= M(h-۱) + M(h-۲) + ۱, \qquad h > ۱ \text{ برای} \end{aligned} \tag{۷-۷}$$

دنباله‌ی اعدادی که رابطه‌ی بازگشتی ۷-۷ تولید می‌کند ($M(h)$) و دنباله‌ی فیبوناچی ($F(h)$) برای h های مختلف در زیر نشان داده شده است: VSNSP

h	۰	۱	۲	۳	۴	۵	۶
$M(h)$	۱	۲	۴	۷	۱۲	۲۰	۳۳
$F(h)$	۱	۱	۲	۳	۵	۸	۱۳

روشن است که $M(h) = F(h+۲) - ۱$. چنان‌چه در بخش ۳-۶ دیدیم، براساس رابطه‌ی ۳-۳۵ داریم

$$F(h) = \frac{۱}{\sqrt{۵}} \left[\left(\frac{۱ + \sqrt{۵}}{۲} \right)^h - \left(\frac{۱ - \sqrt{۵}}{۲} \right)^h \right]. \tag{۸-۷}$$

---

sparse[۳۱]

$\phi = \frac{۱+\sqrt{۵}}{۲} \approx ۱/۶۱۸$ «نسبت طلایی[۳۲]» نامیده می‌شود. پس

$$F(r) > \frac{\phi^r}{\sqrt{۵}} - ۱.$$

اگر n تعداد گره‌های درخت اِی.وی.اِل به ارتفاع h باشد، داریم

$$\begin{aligned} n &\geq F(h+۲) - ۱ \\ &> \frac{\phi^{h+۲}}{\sqrt{۵}} - ۲. \end{aligned}$$

بنابراین،

$$\log_\phi(n+۲) > \log_\phi\left(\frac{\phi^{h+۲}}{\sqrt{۵}}\right) = h + ۲ - \log_\phi \sqrt{۵}$$

که از آن نتیجه می‌شود: $h < ۱/۴۴۰۴ \lg(n+۲) - ۰/۳۲۸$. از طرفی می‌دانیم که $h \geq \lg(n+۱) - ۱$ (حالت مساوی وقتی است که درخت کاملاً متوازن باشد). پس،

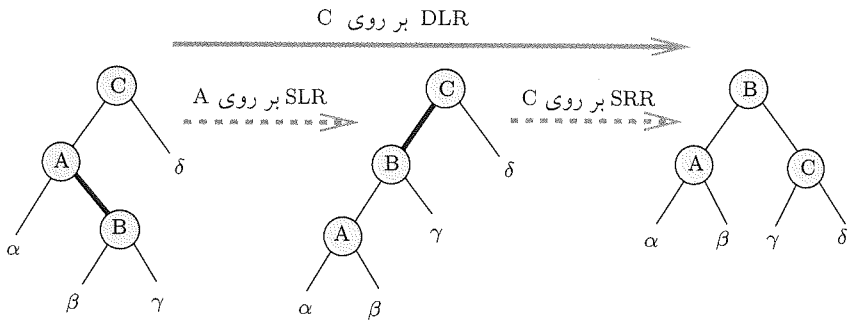
$$\lg(n+۱) - ۱ \leq h < ۱/۴۴ \lg(n+۲) - ۰/۳۲۸ \qquad (۹-۷)$$

دوران در درخت اِی.وی.اِل

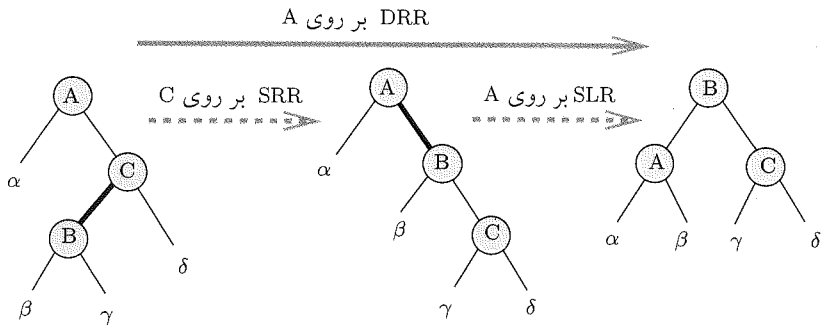
برای تنظیم ارتفاع زیردرخت‌ها در اعمال درج و حذف، علاوه بر دوران راست‌گرد و چپ‌گرد بیان‌شده در بخش ۷-۳-۱، دو «دوران دوتایی[۳۳]» نیز تعریف می‌شوند. این دوران‌های دوتایی چپ‌گرد (به‌اختصار DLR) و راست‌گرد (به‌اختصار DRR) به‌ترتیب در شکل‌های ۷-۲۶ و ۷-۲۷ به‌تصویر کشیده شده‌اند و هر کدام شامل دو «دوران تکی[۳۴]» هستند. دوران‌های تکی نیز یا چپ‌گرد هستند یا راست‌گرد و آن‌را به اختصار با SLR[۳۵] و SRR[۳۶] نام‌گذاری می‌کنیم. دوران‌های تکی برای درخت قرمز-سیاه را در شکل ۷-۱۳ گفته‌ایم.

از این شکل‌ها می‌توان دریافت که انجام این دوران‌ها برروی x توازن گره‌های بالاتر از x (یعنی از x تا ریشه‌ی درخت) را به‌هم نمی‌زند.

[۳۲] golden ratio
[۳۳] double rotation
[۳۴] single rotation
[۳۵] single left rotation
[۳۶] single right rotation



شکل ۷-۲۶ دوران دوتایی چپ‌گرد بر روی گره C.

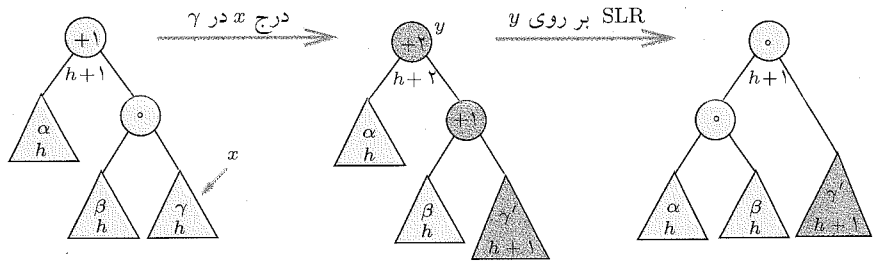


شکل ۷-۲۷ دوران دوتایی راست‌گرد بر روی گره A.

درج در درخت ای.وی.ال

برای درج یک عنصر x ابتدا آن‌را به‌صورت عادی در د.د.ج درج می‌کنیم و سپس با دنبال کردن اشاره‌گر «پدر»، عدد توازن گره‌های این مسیر را مورد بررسی قرار می‌دهیم و آن‌ها را اصلاح می‌کنیم. اولین گره‌ای را که عدد توازنش نادرست است، با دوران اصلاح می‌کنیم. برای این‌کار باید حالت‌های مختلفی را در نظر بگیریم:

حالت اول: وقتی است که پس از عمل درج، عدد توازن یک گره y برابر $+2$ و عدد توازن فرزند راستش $+1$ شود (شکل ۷–۲۸). این مشکل با انجام عمل SLR بر روی y حل می‌شود.

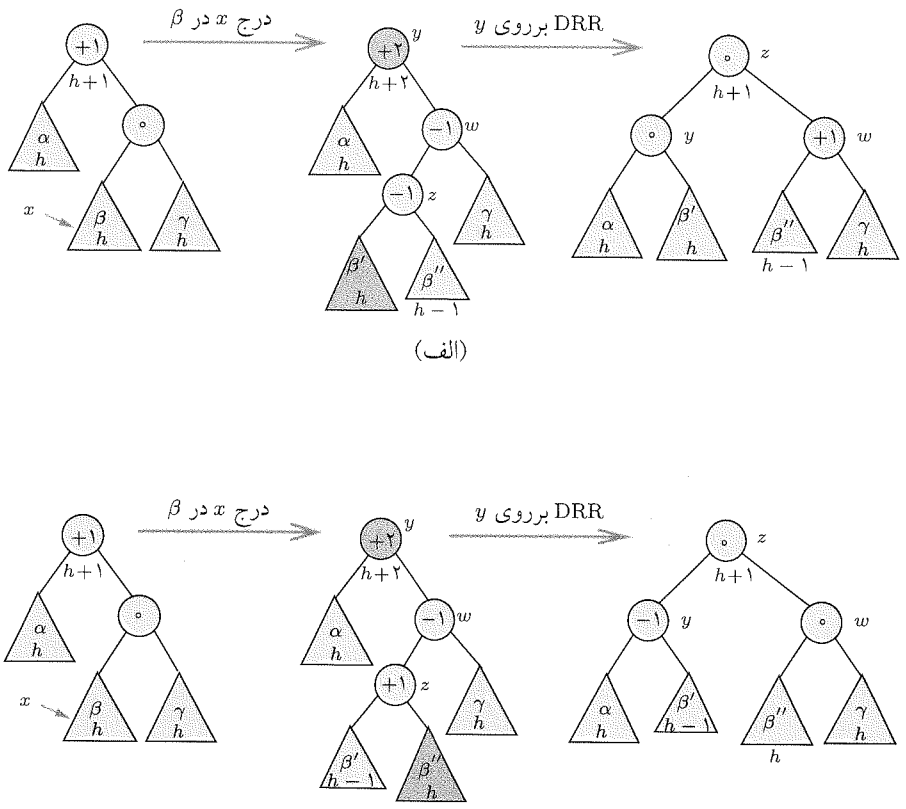


شکل ۷–۲۸ حالت اول درج x در درخت اِی.وی.اِل، اگر x در زیردرخت راست فرزند راست درج شود و توازن به‌هم بخورد.

حالت دوم: وقتی است که عددهای توازن یک گره y برابر $+2$ و فرزند راستش به‌نام w (با فرزند چپ z) برابر -1 شود (شکل ۷–۲۹). این خود دو حالت دارد: x در زیردرخت چپ (حالت (الف)) یا راست w (حالت (ب)) z درج شده باشد. این دو مشکل با انجام یک دوران دوتایی بر روی y حل می‌شود.

حالت‌های قرینه‌ی دیگر نیز مشابه هستند. توجه کنید که چون پس از یک یا دو دوران، ارتفاع گره‌ای که عدد توازنش به‌هم خورده است، به همان مقدار قبل از درج برمی‌گردد، اعداد توازن گره‌های بالاتر تا ریشه نیز تغییری نمی‌کند. بنابراین هزینه‌ی عمل اصلاح توازن درخت پس از درج، $\mathcal{O}(1)$ است. در نتیجه، کل عمل درج $\mathcal{O}(\lg n)$ است.

برحسب داده‌های تجربی، حدود ۵۳٪ از اعمال درج در درخت اِی.وی.اِل، نیازی به انجام دوران ندارد.



شکل ۷-۲۹ حالت دوم درج x در درخت اِی.وی.اِل، اگر x در زیردرخت چپ فرزند راست درج شود و توازن به‌هم بخورد.

حذف از درخت اِی.وی.اِل

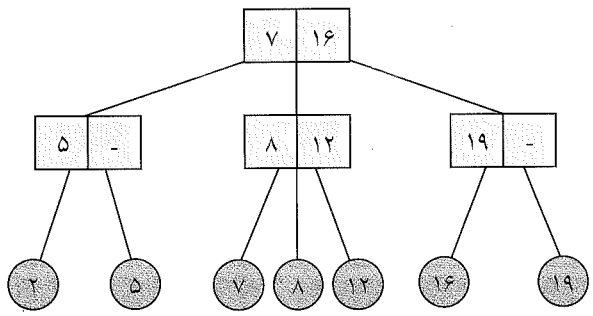
عمل حذف در درخت اِی.وی.اِل به‌سادگی درج که گفتیم نیست و حالت‌های مختلف زیادی دارد و ممکن است به‌تعداد $\mathcal{O}(\lg n)$ دوران نیاز داشته باشد. به‌همین دلیل، در این کتاب الگوریتم حذف را بیان نمی‌کنیم و علاقه‌مندان را به مراجع [۲]، [۸] و [۱۶] ارجاع می‌دهیم.

۷-۴ درخت ۳-۲

درخت ۳-۲ درختی است کاملاً متوازن که هر گره داخلی آن ۲ یا ۳ فرزند دارد. عناصری که این درخت ذخیره می‌کند، به‌ترتیب کلیدشان از چپ‌به‌راست فقط در برگ‌ها قرار دارند و فرض می‌شود که کلید هر عنصر تک است.

برای هدایت جست‌وجو، در هر گره داخلی x، دو مقدار ذخیره می‌شود، یکی کلید کوچک‌ترین عنصر موجود در زیردرخت دوم x و دیگری (در صورت وجود) کلید کوچک‌ترین عنصر در زیردرخت سوم x. یک درخت ۳-۲ با یک عنصر، تنها یک برگ است.

شکل ۷-۳۰ نمونه‌ای از یک درخت ۳-۲ را نشان می‌دهد.



شکل ۷-۳۰ مثالی از یک درخت ۳-۲.

بر اساس این تعریف، یک درخت ۳-۲ به‌ارتفاع h حداقل 2^h و حداکثر 3^h برگ دارد و همین تعداد عنصر را می‌تواند در خود ذخیره کند. به‌عبارت دیگر، اگر h ارتفاع یک درخت ۳-۲ با n عنصر باشد داریم

$$\lceil \log_3 n \rceil \leq h \leq \lfloor \log_2 n \rfloor .$$

بنابراین، اگر اعمال مختلف بر روی این درخت متناسب با ارتفاع آن باشد، این اعمال از $O(\lg n)$ خواهند بود.

درخت ۳-۲ ساده‌ترین گونه‌ی درخت «بی» است که جزئیات آن در بخش ۷-۵ ارائه خواهد شد.

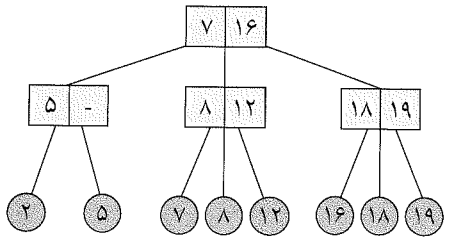
اعمال مختلف بر روی درخت ۳-۲

جست‌وجو

با استفاده از اطلاعات ذخیره‌شده در گره‌های داخلی می‌توان یک عنصر با کلید x را به‌سرعت جست‌وجو کرد و در صورت وجود به آن دست‌رسی داشت. فرض کنید در برگ‌های زیردرختی با ریشه‌ی یک گره‌ی داخلی a، که در آن اعداد x_1 و x_2 ذخیره شده‌اند، به‌دنبال x می‌گردیم. اگر $x < x_1$، باید به فرزند اول a برویم و همین‌کار را پی می‌گیریم. اگر $x_1 \le x$ و a فقط دو فرزند داشته باشد، باید کار جست‌وجو را از فرزند دوم a دنبال کنیم. در صورت داشتن سه فرزند، اگر $x_1 \le x < x_2$، کار از فرزند دوم و اگر $x_2 \le x$ از فرزند سوم دنبال می‌شود. اگر a برگ باشد کلیدش با x مقایسه می‌شود و در صورت برابری x پیدا می‌شود، و گرنه جست‌وجو ناموفق است.

درج در درخت ۳-۲

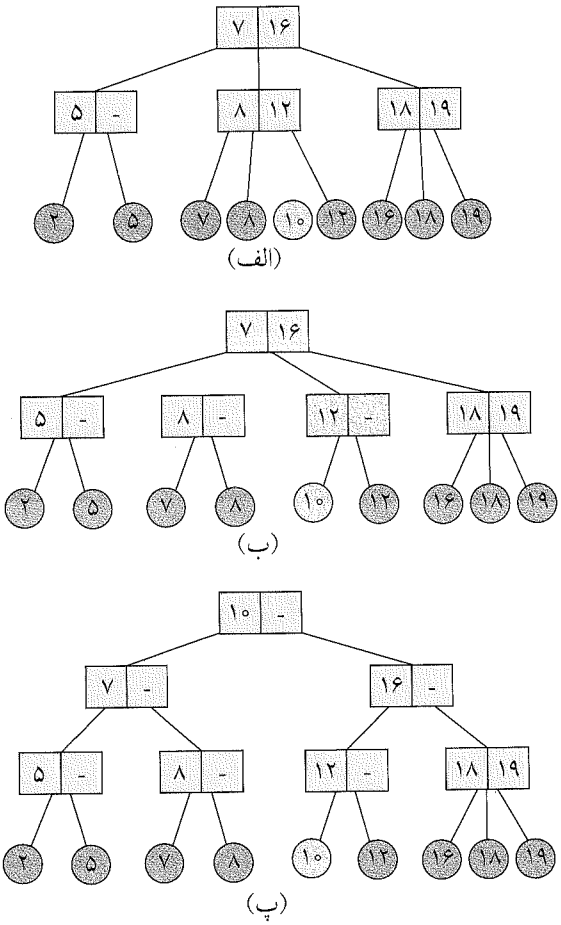
برای درج یک عنصر جدید با کلید x در درخت ۳-۲ ی T، ابتدا x را در T جست‌وجو می‌کنیم. در صورتی که در درخت نباشد، در یک سطح بالاتر به گره a رسیده‌ایم که x در وضع موجود باید فرزند آن باشد. اگر a فقط دو فرزند داشته باشد، x را می‌توان به‌سادگی به عنوان فرزند سوم آن درج کرد. اگر x اولین فرزند a باشد، باید در صورت لزوم اعداد ثبت‌شده در گره‌های داخلی را تغییر داد. این کار به‌صورت بازگشتی ممکن است تا ریشه ادامه یابد. شکل ۷-۳۱ درخت حاصل را پس از درج عنصری با کلید ۱۸ نشان می‌دهد.



شکل ۷-۳۱ درخت حاصل از درج $x = ۱۸$ در درخت شکل ۷-۳۰.

اگر a از قبل سه فرزند داشته باشد، مکان x را در بین فرزندانش به‌دست می‌آوریم. سمت چپ a، یک گره برادر برای a، به‌نام a_1 ایجاد می‌کنیم و دو فرزند اول (بین چهار

فرزند) را به a و دو فرزند بعدی را به a_1 می‌دهیم. سپس a_1 را به‌صورت بازگشتی در درخت T و در یک سطح بالاتر درج می‌کنیم. اگر a ریشه‌ی T باشد، در آن صورت یک ریشه‌ی جدید به‌نام r لازم است که a_1 و a فرزندان آن باشند (شکل ۷-۳۲ را ببینید.)



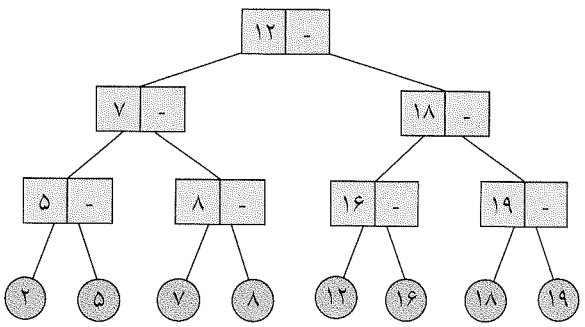
شکل ۷-۳۲ مراحل درج $x = ۱۰$ در درخت شکل ۷-۳۱. (الف) درج در جای درست. (ب) تقسیم گره با ۴ فرزند به دو گره با ۲ فرزند، و (پ) درج گره داخلی جدید و اصلاح درخت.

حذف از درخت ۳-۲

برای حذف نیز ابتدا عنصرِ مورد نظر، مثلاً x را پیدا می‌کنیم و آن‌را از درخت ۳-۲ حذف می‌کنیم. مشکل هنگامی پیش می‌آید که پدر عنصری که حذف می‌شود، مثلاً a، تنها برایش یک فرزند به‌نام c باقی بماند. اگر a ریشه باشد، آن‌را حذف و تنها فرزند a، یعنی c را به‌عنوان ریشه‌ی یک درخت ۳-۲ قرار می‌دهیم. اما حالت‌های دیگر را نیز باید در نظر بگیریم که به‌طور مختصر به‌آن اشاره می‌کنیم.

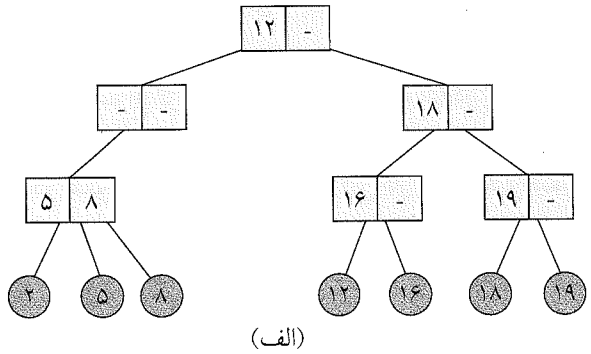
در حالت کلی باید به‌سراغ برادرهای a (یا عموهای x) در درخت برویم. اگر یکی از این عموها، مثلاً y، ۳ فرزند داشته باشد، فرزند مناسب y را به a می‌دهیم تا با c دو فرزند داشته باشد. اگر هر دو عمو ۲ فرزند داشته باشند، این‌کار امکان‌پذیر نیست. در این‌صورت، c را به‌عنوان فرزند به یکی از عموها می‌دهیم و a را به‌صورت بازگشتی از درخت حذف می‌کنیم. البته این بازگشت از ابتدا صورت می‌گیرد تا بتوانیم اعداد ثبت‌شده در گره‌های داخلی را نیز روزآمد کنیم.

شکل ۷-۳۳ تغییرات حاصل از حذف ۱۰ را نشان می‌دهد که در آن فرزند عموی ۱۰ (یعنی ۱۶) را به‌عنوان برادر ۱۲ قرار می‌دهیم. پس از این تغییر، لازم است اطلاعات گره‌های داخلی اصلاح شود و این ممکن است تا ریشه بالا برود.

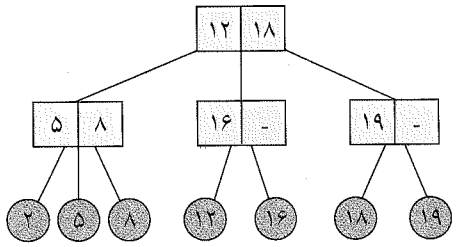


شکل ۷-۳۳ درخت حاصل از حذف ۱۰ درخت شکل ۷-۳۲(پ). پس از حذف ۱۰، گره ۱۶ را از عموی ۱۰ حذف و آن‌را برادر۱۲ قرار می‌دهیم و اطلاعات گره‌های داخلی را اصلاح می‌کنیم.

شکل ۷-۳۴ نیز مراحل مختلف حذف ۷ از درخت شکل ۷-۳۳ را نشان می‌دهد. در این مثال، عموی ۷ فقط دو فرزند دارد، بنابراین پس از حذف ۷، پدرش نیز باید به‌صورت بازگشتی از درخت حذف شود، که این در انتها موجب کاهش ارتفاع درخت می‌شود.



(الف)



(ب)

شکل ۷-۳۴ درخت حاصل از حذف ۷ از درخت شکل ۷-۳۳. (الف) تنها برادر ۷ به‌عنوان فرزند سوم به گره عمو داده می‌شود و گره پدر ۷ حذف می‌شود. (ب) با این تغییر گره جد ۷ یک فرزند خواهد داشت. این مشکل به‌صورت بازگشتی حل می‌شود، در نتیجه ارتفاع درخت یک واحد کم می‌شود.

۷-۵ درخت «بی»

درخت بی[۳۷] نوع خاصی از درخت m تایی متوازن است که به‌منظور بازیابی، درج و حذف رکوردها از یک فایل بزرگ طراحی شده است. مانند الگوریتم‌های مرتب‌سازی خارج که در بخش ۶-۶ گفته شد، فرض می‌شود اندازه‌ی فایل به‌حدی است که نمی‌توان همه‌ی آن‌را یک‌جا در حافظه‌ی اصلی آورد و تنها بخشی از آن در حافظه جای می‌گیرد. بنابراین فایل بر روی حافظه‌ی خارجی یا دیسک ذخیره می‌شود و با انجام اعمال DISK-READ و DISK-WRITE رکوردهای مختلف آن از روی دیسک خوانده و یا بر روی آن نوشته می‌شود. با توجه به زمان بسیار زیاد خواندن از و نوشتن بر روی دیسک، داده‌ساختاری که برای این کار طراحی می‌شود باید از نظر تعداد دست‌رسی‌ها کارا باشد.

درخت بی یک درخت کاملاً متوازن است که به‌صورت‌های مختلف طراحی می‌شود. یکی از این روش‌ها مبتنی بر تعمیم درخت ۳-۲ است و دیگری مبتنی بر درخت دودویی جست‌وجو است. در این بخش، روش اول را به‌اختصار توضیح می‌دهیم. برای روش دوم به کتاب [۵] مراجعه کنید.

پیاده‌سازی مبتنی بر درخت ۳-۲

به طور کلی یک درخت بی از مرتبه‌ی m یک درخت جست‌وجوی m تایی با ویژگی‌ها و پارامترهای زیر است:

۱. ریشه، یا برگ است یا دست‌کم دو فرزند دارد.

۲. هر گره، غیر از ریشه و برگ‌ها، حداقل $\lceil \frac{m}{2} \rceil$ و حداکثر m فرزند دارد.

۳. یک گره داخلی دارای $m-1$ کلید و m اشاره‌گر (آدرس کامل یک بلوک دیسک) از نوع $\langle p_1, k_2, p_2, k_3, p_4, \cdots, k_m, p_m \rangle$ است که (به‌شرط وجود) p_i اشاره‌گر به iامین زیردرخت آن گره و k_i کوچک‌ترین کلید زیردرخت iام است. همچنین داریم $k_2 < k_3 < \cdots < k_m$.

۴. برگ‌ها همه در یک سطح قرار دارند و رکوردهای ذخیره‌شده در هر برگ به‌ترتیب کلیدشان از چپ‌به‌راست قرار دارند.

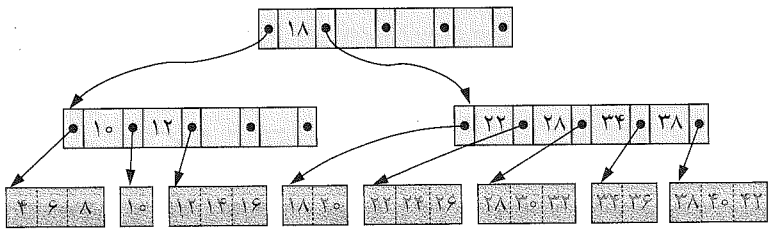
---

[۳۷] B-tree

۵. بسته به اندازه‌ی رکوردها، از یک تا حداکثر k رکورد در هر برگ قرار می‌گیرد.

توجه کنید که یک درخت ۳-۲ یک درخت بی از مرتبه‌ی ۳ است.

مقادیر m و k معمولاً طوری تعیین می‌شوند که اندازه‌ی هر گره داخلی و هر برگ برابر یک بلوک از دیسک باشد و بتوان با یک‌بار دست‌رسی آن را به حافظه‌ی اصلی خواند. هم‌چنین اشاره‌گرها در این درخت، آدرس بلوک بر روی دیسک هستند. شکل ۷-۳۵ یک درخت بی از مرتبه‌ی ۵ را نشان می‌دهد که در هر بلوک برگ حداکثر ۳ (k) رکورد جای می‌گیرد.

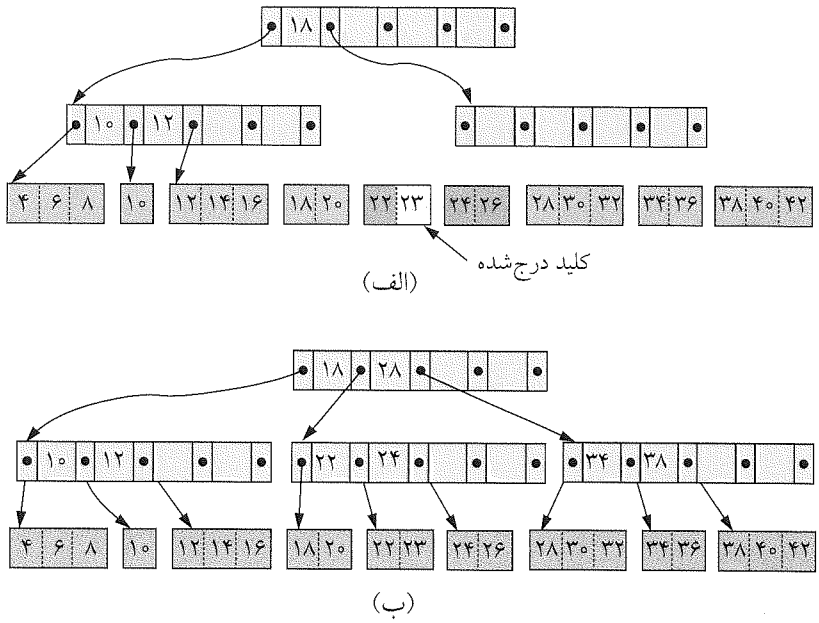


شکل ۷-۳۵ یک درخت بی از مرتبه‌ی ۵ و $k = ۳$.

اعمال مختلف

اعمال جست‌وجو، درج و حذف در درخت بی، گسترش‌یافته‌ی همان اعمال در درخت ۳-۲ است. عمل جست‌وجو با توجه به اطلاعات موجود در گره‌های داخلی به‌سادگی انجام می‌شود. هر بار دست‌رسی به عناصر داخلی و یا برگ‌ها معادل خواندن بلوک متناظر در دیسک است. با توجه به مرتب بودن کلیدها در هر گره داخلی می‌توان برای یافتن عنصر مورد نظر، از جست‌وجوی دودویی استفاده کرد.

برای درج یک عنصر x، ابتدا برگی که x باید در آن درج شود جست‌وجو و به حافظه خوانده می‌شود. در صورتی‌که آن برگ جایی برای درج یک عنصر جدید داشته باشد، x در آن درج و در صورت لزوم اطلاعات گره‌های داخلی به‌هنگام می‌شود. اگر برگ پر باشد، پس از اضافه‌شدن x یک برگ جدید با مجموع $\lceil (k+1)/2 \rceil$ عنصر ایجاد و مانند یک گره جدید در درخت بی (مثل درخت ۳-۲) درج می‌شود. شکل ۷-۳۶ درخت شکل ۷-۳۵ را پس از درج عنصر ۲۳ نشان می‌دهد. چنان‌چه مشخص است برگ با عناصر ۲۳، ۲۴ و ۲۶ به



شکل ۷-۳۶ مراحل مختلف درج ۲۳ در درخت ۷-۳۵.

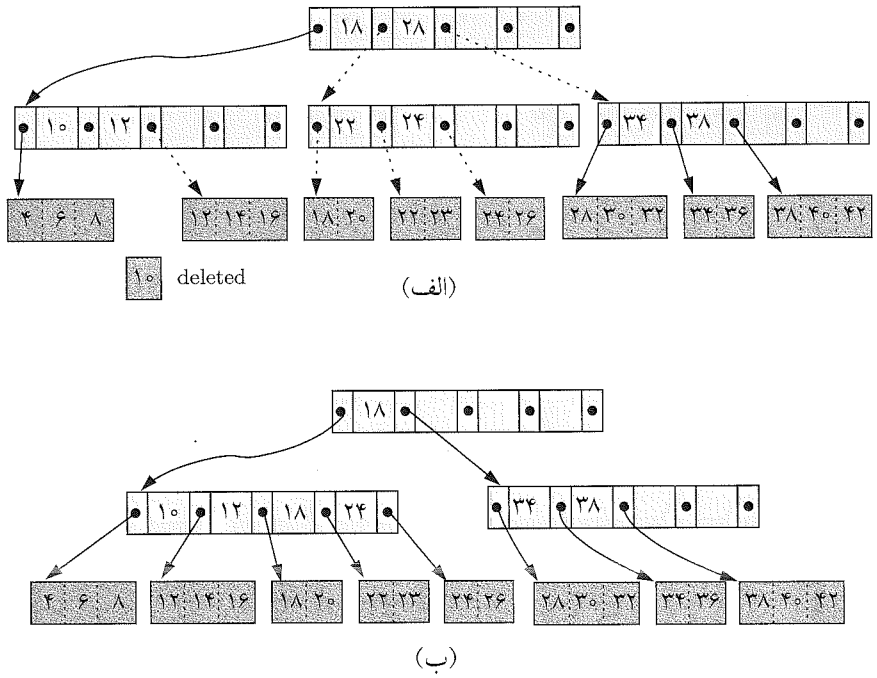
دو برگ، اولی با عناصر ۲۲ و ۲۳ و دومی با عناصر ۲۴ و ۲۶، تبدیل می‌شود و برگ جدید در درخت درج می‌شود.

حذف در درخت بی هم مشابه الگوریتم حذف در درخت ۳-۲ است. شکل ۷-۳۷ درخت شکل ۷-۳۶ را پس از حذف ۱۰ نشان می‌دهد.

تحلیل

اگر درخت بی از مرتبه‌ی m شامل n عنصر باشد و اگر هر برگ به‌طور میانگین b عنصر داشته باشد (تخمین $b = k/2$ برای توزیع یکنواخت درست است)، در آن‌صورت تعداد برگ‌ها برابر $\lceil n/b \rceil$ است. بیش‌ترین ارتفاع درخت وقتی است که گره‌های داخلی (بجز ریشه) $\lceil \frac{m}{2} \rceil$ فرزند و ریشه دو فرزند داشته باشد. در این‌صورت، برگ‌ها حدود $2\lceil n/b \rceil / m$ عدد پدر و $4\lceil n/b \rceil / m^2$ عدد جد و ... همین‌طور تا بالاتر دارد.

اگر j تعداد گره‌های موجود در مسیر بین ریشه و برگ‌ها باشد، داریم



شکل ۷-۳۷ مراحل مختلف حذف ۱۰ از درخت ۷-۳۶.

$۱ \geq ۲^{j-۱}\lceil n/b \rceil / m^{j-۱}$، چرا که در غیر این‌صورت تعداد فرزندان ریشه کم‌تر از ۱ خواهد بود. بنابراین $\lceil n/b \rceil \geq (m/۲)^{j-۱}$ و یا $j \leq ۱ + \log_{m/۲}\lceil n/b \rceil$. مثلاً اگر $n = ۱۰^۶$، $b = ۱۰$ و $m = ۱۰۰$ در آن‌صورت $j \leq ۳/۵$.

روشن است که اعمال جست‌وجو، و حذف حداکثر j بار و درج حداکثر $j + ۱$ بار نیاز دارد تا به دیسک دست‌رسی داشته باشد.

تمرین‌های فصل ۷

۱.۷ فرض کنید که یک جست‌وجو برای پیدا کردن یک عنصر خاص در یک درخت قرمز-سیاه پس از ۲۰ شاخه از ریشه‌ی درخت به‌صورت ناموفق متوقف می‌شود. حداکثر و حداقل تعداد شاخه‌هایی که در هر جست‌وجوی ناموفق در این درخت ملاقات می‌شوند چقدر است؟

۲.۷ مسئله‌های زیر را حل کنید:

الف) n نقطه بر روی محور xها داده شده‌اند. می‌خواهیم تعداد نقاط موجود در یک بازه‌ی $[a, b]$ را پیدا کنیم. الگوریتمی از $\mathcal{O}(\lg n)$ (مستقل از تعداد جواب) برای این مسئله ارائه دهید. داده‌ساختاری که استفاده می‌کنید و الگوریتم خود را توضیح دهید، و تحلیل کنید.

ب) n قطعه‌خط افقی یا عمودی داده شده‌اند. با استفاده از الگوریتم بند (الف) (و یا با فرض وجود آن) الگوریتمی از $\mathcal{O}(n \lg n)$ (مستقل از تعداد جواب) برای یافتن تعداد نقاط تلاقی این خطوط ارائه کنید. الگوریتم خود را توضیح دهید و آن را تحلیل نمایید.

۳.۷ عکس دسته‌جمعی

برای گرفتن عکس دسته‌جمعی، n نفر با قدهای h_1 تا h_n در کنار هم در یک ردیف ایستاده‌اند. این ایستادن را «k-ناجور» می‌نامیم اگر یک گروه k نفری کنار هم از این افراد باشند که اندازه‌ی قد بلندقدترین فرد در این گروه حداقل نیم‌متر از قد کوتاه‌قدترین فرد همان گروه بیش‌تر باشد ($k < n$ پارامتر ورودی است). می‌خواهیم به‌کمک یک داده‌ساختار مناسب، الگوریتمی کارا از $\mathcal{O}(n \lg k)$ طراحی کنیم تا با دریافت قدهای افراد (به‌ترتیب ایستادن) و عدد k مشخص کند که آیا ایستادن آن‌ها k-ناجور است یا خیر. با گسترش درخت قرمز-سیاه الگوریتم خود را دقیقاً بیان و تحلیل کنید و درستی آن‌را ثابت نمایید. (توجه: برای این‌کار الگوریتمی با هزینه‌ی سرشکنی $\mathcal{O}(n)$ وجود دارد.)

*** ۴.۷ خسارت طوفان در یک منطقه‌ی ساحلی**

یک منطقه‌ی ساحلی شامل n خانه است. در زمان طوفان، هر موج قوی به‌خانه‌های واقع در یک فاصله‌ی معین از دریا خسارت وارد می‌کند. می‌خواهیم داده‌ساختاری طراحی کنیم تا دو عمل زیر را در زمان $\mathcal{O}(\lg n)$ انجام دهد.

INCREASE-DAMAGE (*distance, amount*) : میزان خرابی هر خانه به فاصله *distance* از ساحل (یا کم‌تر) را به‌اندازه‌ی *amount* اضافه کن،

ASSESS-DAMAGE (*distance*) : میزان کل خرابی وارده به یک خانه در فاصله‌ی *distance* از ساحل را محاسبه کن.

فرض کنید n خانه و فاصله‌ی هر یک از ساحل داده شده‌اند. داده‌ساختاری طراحی کنید که در زمان

$\mathcal{O}(n \lg n)$ بتوان آن را ساخت، و نیز هر یک از اعمال بیان‌شده را بتوان در $\mathcal{O}(\lg n)$ انجام داد. نحوه‌ی ساخت داده‌ساختار اولیه را توضیح دهید و شبه‌کدهای دو عمل را بنویسید.

کاربرد مسئله آن است که در زمان طوفان، هر بار که موجی قوی می‌آید، فاصله‌ی خراب‌کاری آن موج محاسبه می‌شود و میزان خرابی خانه‌ها واقع در آن فاصله بیش‌تر می‌شود. پس از طوفان، ساکنان خانه که به‌جای امنی رفته‌اند می‌توانند با بررسی میزان خرابی تصمیم بگیرند که به منزل خود باز گردند یا این‌که سراغ بیمه‌ی خود بروند.

⋆ ۵.۷ مولد پویای اعداد تصادفی

می‌خواهیم یک مولد پویای اعداد تصادفی بسازیم. این مولد باید آرایه‌ی $W[1 \cdots n]$ از اعداد نامنفی (صحیح یا حقیقی) به نام «وزن» را با داده‌ساختار مناسب پیاده‌سازی و روزآمد کند. مولد باید اعمال زیر را فراهم بیاورد:

- MODIFY (i, w): گمارش $W[i] = w$ را انجام بده.
- GENERATE: یک عدد تصادفی بین $\{1, 2, \cdots, n\}$ را برگردان، به‌طوری‌که احتمال برگرداندن i برابر $\frac{W[i]}{\sum_{j=1}^{n} W[j]}$ باشد.

الف) نشان دهید که چگونه بر اساس یکی از گسترش‌های درخت قرمز-سیاه می‌توان داده‌ساختاری طراحی کرد که اعمال فوق را در زمان $\mathcal{O}(\lg n)$ انجام دهد. برای حل این مسئله، شما می‌توانید از تابع RAND(a, b) استفاده کنید که در زمان $\mathcal{O}(1)$ و با احتمال یک‌سان عددی حقیقی در بازه‌ی $[a, b]$ را انتخاب می‌کند. حالت اولیه‌ی داده‌ساختار باید برابر $W[1] = 1$ و $W[i] = 0$ برای $1 < i \leq n$ در نظر گرفته شود. شما می‌توانید زمان ایجاد داده‌ساختار در حالت اولیه را نادیده بگیرید. (راهنمایی: داده‌ساختار شما قاعدتاً باید در $\mathcal{O}(\lg n)$ عنصر x را پیدا کند که $\sum_{j=1}^{x-1} W[j] < z$ و $\sum_{j=1}^{x} W[j] \geq z$. از این عنصر استفاده کنید.)

ب) n قطعه‌خط افقی یا عمودی داده شده‌اند. با استفاده از الگوریتم بند بالا (یا با فرض وجود آن) الگوریتمی از $\mathcal{O}(n \lg n)$ (مستقل از تعداد جواب) برای پیدا کردن تعداد نقاط تلاقی این خطوط ارائه کنید. به‌طور دقیق الگوریتم خود را توضیح دهید و آن را تحلیل نمایید.

۶.۷ یک مجموعه‌ی پویای Q شامل عناصر x و کلید $key[x]$ است. این مجموعه اعمال زیر را انجام می‌دهد:

- INSERT(x, Q): درج عنصر x در Q
- $x \leftarrow$ EXTRACT-OLDEST (Q): عنصری از Q را که از بقیه زودتر درج‌شده است حذف می‌کند و آن‌را برمی‌گرداند،
- $x \leftarrow$ FIND-MAX (Q): عنصر x را پیدا می‌کند (ولی حذف نمی‌کند) که بیش‌ترین کلید $key[x]$ را در Q داشته باشد.

یک داده‌ساختار کارا برای این مجموعه‌ی پویا طراحی کنید و بنویسید که هر یک از اعمال بیان‌شده چگونه انجام می‌شود و آن‌را تحلیل نمایید.

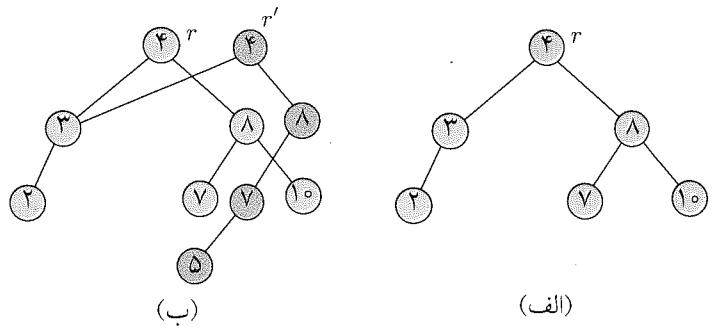
⋆ ۷.۷ د.د.ج ماندگار

در جریان توسعه‌ی یک الگوریتم گاهی متوجه می‌شویم که نیاز به نگه‌داری نسخه‌های قبلی یک مجموعه‌ی پویا در حین روزآمد کردن آن داریم. چنین مجموعه‌ای ماندگار[۳۸] نامیده می‌شود. یک راه برای پیاده‌سازی یک مجموعه‌ی ماندگار کپی‌کردن کل مجموعه هنگام تغییر آن است، اما این روش می‌تواند باعث کند شدن اجرای برنامه و اشغال مقدار زیادی حافظه شود. گاهی اوقات می‌توانیم بسیار بهتر عمل کنیم.

مجموعه‌ی ماندگار S با عملیات درج، حذف و جست‌وجو را آن‌چنان که در شکل ۷–۳۸(الف) نشان داده شده است در نظر بگیرید با استفاده از د.د.ج پیاده‌سازی شده است، . در این شکل، برای هر نسخه از مجموعه ریشه‌ای جداگانه نگه داشته‌ایم. برای درج کلید ۵ در مجموعه، گره جدید با کلید ۵ می‌سازیم. از آن‌جایی که نمی‌توانیم گره موجود با کلید ۷ را تغییر دهیم، این گره فرزند چپ گره جدید با کلید ۷ می‌شود. به‌طور مشابه، گره جدید با کلید ۷ فرزند چپ گره جدید با کلید ۸ می‌شود که فرزند سمت راستش گره موجود با کلید ۱۰ است. گره جدید با کلید ۸ نیز به‌نوبه‌ی خود فرزند راست ریشه‌ی جدید r' با کلید ۴ که فرزند چپش گره موجود با کلید ۳ است می‌شود. بنابراین آن‌چنان که در شکل ۷–۳۸(ب) نشان داده شده است، ما تنها قسمتی از درخت را کپی می‌کنیم و بعضی از گره‌ها را با درخت اصلی به اشتراک می‌گذاریم.

فرض کنید هر گره درخت مؤلفه‌های کلید، چپ و راست را دارد اما مؤلفه‌ای به‌عنوان پدر ندارد.

الف) برای حالت کلی یک د.د.ج ماندگار، گره‌هایی را که هنگام درج کلید k یا حذف گره y نیاز به تغییر دارند مشخص کنید.



شکل ۷–۳۸ (الف) یک د.د.ج. (ب) یک د.د.ج ماندگار پس از درج کلید ۵.

---

[۳۸] persistent

ب) رویه‌ی PERSISTENT-TREE-INSERT را بنویسید که با گرفتن یک درخت ماندگار T و کلید k برای درج، درخت ماندگار جدید T' را که نتیجه‌ی درج k در T است برگرداند.

پ) اگر ارتفاع د.د.ج ماندگارِ T برابر h باشد، زمان و حافظه‌ی مورد نیاز در پیاده‌سازی شما از PERSISTENT-TREE-INSERT چیست؟ (حافظه‌ی مورد نیاز متناسب با تعداد گره‌های جدید گرفته شده است.)

ت) فرض کنید مؤلفه‌ی پدر را به تمام گره‌ها اضافه کنیم. در این حالت PERSISTENT-TREE-INSERT نیاز به کپی‌کردن بیش‌تری دارد. ثابت کنید در این حالت PERSISTENT-TREE-INSERT نیاز به $\Omega(n)$ زمان و حافظه خواهد داشت که n تعداد گره‌های درخت است.

ث) روش استفاده از درخت قرمز-سیاه را برای تضمین این‌که زمان اجرا و حافظه‌ی مورد نیاز برای هر درج یا حذف در بدترین حالت $\mathcal{O}(\lg n)$ بماند، بیان کنید.

⋆ ۸.۷ پیوند دو مجموعه‌ی پویا

عمل پیوند، دو مجموعه‌ی پویای S_1 و S_2 و یک عنصر x را به‌طوری که برای هر $x_1 \in S_1$ و $x_2 \in S_2$ داشته باشیم $key[x_1] \leq key[x] \leq key[x_2]$ می‌گیرد و مجموعه‌ی $S = S_1 \bigcup \{x\} \bigcup S_2$ را برمی‌گرداند. در این مسئله به دنبال روشی برای پیاده‌سازی عمل پیوند روی درخت‌های قرمز-سیاه هستیم.

الف) با در دست داشتن درخت قرمز-سیاه T، سیاه-ارتفاع آن‌را در مؤلفه‌ی $bh[T]$ نگه می‌داریم. نشان دهید این مؤلفه می‌تواند بدون نیاز به فضای اضافه، در گره‌های درخت و بدون افزایش زمان مجانبی در رویه‌های RB-INSERT و RB-DELETE، نگه‌داری و روزآمد شود. همچنین ثابت کنید هنگام پایین رفتن در T می‌توانیم سیاه-ارتفاع گره‌های ملاقات‌شده را برای هر گره در زمان $\mathcal{O}(1)$ محاسبه کنیم.

می‌خواهیم عمل RB-JOIN(T_1, x, T_2) که T_1 و T_2 را خراب کرده و درخت قرمز-سیاه $T = T_1 \bigcup \{x\} \bigcup T_2$ را برمی‌گرداند، پیاده‌سازی کنیم. فرض کنید n جمع گره‌های T_1 و T_2 باشد.

ب) فرض کنید $bh[T_1] \geq bh[T_2]$. الگوریتمی با زمان اجرای $\mathcal{O}(\lg n)$ توصیف کنید که گره سیاه y را در T_1 با بزرگترین کلید در میان گره‌هایی که سیاه-ارتفاع آن‌ها برابر $bh[T_2]$ است پیدا کند.

پ) فرض کنید T_y نشان‌دهنده‌ی زیردرختی با ریشه‌ی y باشد. نشان دهید چگونه می‌توانیم T_y را بدون خراب کردن خاصیت د.د.ج، در $\mathcal{O}(1)$ با $T_y \bigcup \{x\} \bigcup T_2$ عوض کنیم.

ت) x را با چه رنگی رنگ کنیم تا همه‌ی خواص درخت قرمز-سیاه (بجز این‌که ریشه سیاه است و پدر گره قرمز حتماً سیاه است) باقی بماند؟ شرح دهید چگونه می‌توانیم دو خاصیت ذکر شده را در زمان $\mathcal{O}(\lg n)$ اعمال کنیم؟

ث) نشان دهید که با فرض قسمت (ب) چیزی از کلیت مسئله کم نمی‌شود. حالت متقارنی را

توصیف کنید که هنگامی که $bh[T_1] = bh[T_2]$ می‌شود، رخ می‌دهد.

ج) ثابت کنید که زمان اجرای RB-Join از $\mathcal{O}(\lg n)$ است.

۹.۷ طراحی مدارهای وی‌ال‌اس‌آی

در طراحی مدارهای وی‌ال‌اس‌آی[۳۹]، یک مدار مجموعه‌ای از چند مستطیل (آی‌سی) است که بر روی برد قرار دارد. الگوریتمی طراحی کنید که با گرفتن مختصات آی‌سی‌های روی برد، مشخص کند که آیا این برد قابل پیاده‌سازی هست یا خیر؟ (آیا هیچ‌کدام از مستطیل‌ها روی هم می‌افتند یا خیر؟) الگوریتمی از مرتبه‌ی $\mathcal{O}(n \lg n)$ برای این‌کار ارائه دهید. (راهنمایی: یک خط جاروب را از روی صفحه عبور دهید و بازه‌ها را بررسی کنید. به‌علاوه، ابتدا مستطیل‌ها را مرتب کنید.)

[۳۹] Very Large Scale Integrated Circuits

پروژه‌های برنامه‌نویسی فصل ۷

۱ د.د.ج از نوع ساده، متوازن و بهینه

دراین پروژه شما یک فرهنگ داده‌ای از کلمه‌های کلیدی زبان برنامه‌نویسی پاسکال را با استفاده از درخت دودویی جست‌وجو و به‌صورت‌های مختلف ساده، متوازن، و بهینه ایجاد می‌کنید و کارایی آن‌ها را از نظر میانگین زمان جست‌وجوی موفق و ناموفق مقایسه می‌کنید.

در این پروژه شما باید مراحل زیر را انجام دهید:

۱. فایلی به‌نام `create.in` را که حاوی تعداد زیادی دستور `INSERT name` و `DELETE name` است دریافت کند و اعمال آن‌را به‌ترتیب بر روی یک د.د.ج که در ابتدا تهی است انجام دهد. این دستورها برای درج کلمه‌ی کلیدی `name` به T یا حذف آن کلمه از آن است. علاوه بر ایجاد درخت، این قسمت از برنامه‌ی شما فایلی به‌نام `create.out` ایجاد می‌کند که در مقابل هر دستور تعداد مقایسه‌های انجام‌شده بین کلمه‌ی کلیدی ورودی و عناصر درخت و نتیجه‌ی آن دستور چاپ می‌شود. دستورهای موجود در فایل ورودی شامل درج کلمه‌های تکراری و حذف کلمه‌های ناموجود در درخت نیز هست.

۲. یک فایل دیگر به‌نام `member.in` حاوی تعداد زیادی دستور `MEMBER name` را دریافت کنید و برای هر دستور، عمل جست‌وجو در درخت ساخته‌شده‌ی بالا برای کلمه‌ی `name` را انجام دهید. این جست‌وجوها ممکن است موفق یا ناموفق باشند (یعنی عنصر مورد جست‌وجو پیدا شود یا خیر). خروجی این قسمت از برنامه فایلی است به‌نام `member1.out` که در آن در مقابل هر دستور ورودی، تعداد مقایسه‌های انجام شده بین `name` و کلمه‌های موجود در درخت و موفق یا ناموفق بودن جست‌وجو نوشته می‌شود. در انتها، میانگین تعداد مقایسه‌ها برای یک دستور دل‌خواه محاسبه و در این فایل نوشته می‌شود.

   برای استفاده در بند ۳ لازم است در این قسمت بسامد جست‌وجو را برای هر عنصر داخلی درخت (برای جست‌وجوهای موفق) و عناصر خارجی آن (برای جست‌وجوهای ناموفق) محاسبه کنید. (در فایل ورودی تعدادی از دستورها تکراری هستند.)

۳. د.د.ج بالا را بازسازی کنید به‌طوری‌که حاصل د.د.ج و متوازن باشد. سپس بند ۲ فوق را مجدداً بر روی این درخت انجام دهید و فایل `member2.out` را با همان محتوا بسازید.

۴. با استفاده از بسامدهای جست‌وجو برای عناصر داخلی و خارجی محاسبه‌شده در بند ۲، درخت فوق را مجدداً بازسازی کنید و به‌صورت د.د.ج بهینه درآورید. بند ۲ را بر روی این درخت انجام دهید و فایلی به‌نام `member3.out` را با همان محتوا ایجاد کنید.

۲ درخت کی‌دی

هدف از این پروژه‌ی برنامه‌نویسی، ایجاد یک داده‌ساختار مناسب برای ذخیره‌ی تعداد زیادی نقطه در فضای دو بعدی یا بیش‌تر است به‌طوری که بتوان با سرعت خوبی نقاطی را که داخل یا روی مرز یک پنجره‌ی دل‌خواه در ورودی مشخص می‌شود به‌دست آورد.

این برنامه می‌تواند مثلاً در سیستم پنجره‌ای شامل اشیاء نقطه‌ای‌شکل و یک پنجره‌ی متحرک کاربرد داشته باشد که در هر لحظه بخواهیم نقاط داخل آن پنجره مشخص شوند. همچنین اگر نقاط k بعدی و هر کدام، یک رکورد h مؤلفه‌ای از داده‌پایگاه باشند، پنجره‌ی d بعدی مثل یک جست‌وجو در داده‌پایگاه است.

درخت کی‌دی

نقاط با مختصات x و y داده شده‌اند. می‌توان فرض کرد که این نقاط در داخل یک جعبه‌ی بزرگ قرار دارند. ریشه‌ی درخت کی‌دی، r این ناحیه را نشان می‌دهد که حاوی همه‌ی نقاط است. یک خط عمودی به‌نام l_1 از نقطه‌ی میانه‌ی این نقاط، یا p رسم می‌کنیم. این خط ناحیه‌ی اصلی را به دو ناحیه‌ی (جعبه‌ی) کوچک‌تر تقسیم می‌کند. ناحیه‌ی سمت چپ را با گره $left[r]$ و ناحیه‌ی سمت راست را با گره $right[r]$ نشان می‌دهیم. فرض می‌کنیم که نقطه‌ی p متعلق به ناحیه‌ی متناظر با $left[r]$ است. حال هر یک از دو ناحیه‌ی متناظر با $left[r]$ و $right[r]$ را با یک خط افقی که از نقطه‌ی میانه‌ی نقاط این ناحیه رد می‌شود، به دو قسمت بالایی و پایینی تقسیم می‌کنیم و هر ناحیه را با دو گره (گره سمت چپ برای ناحیه‌ی پایینی و گره سمت راست برای ناحیه‌ی بالایی) نشان می‌دهیم (l_2 و l_3 این خطوط افقی هستند). فرض می‌کنیم که در این حالت، نقاط میانه به ناحیه‌های پایینی متعلق هستند. هر یک از ناحیه‌های جدید و کوچک‌تر را با خطوط عمودی مطابق قبل تقسیم می‌کنیم. این کار را ادامه می‌دهیم تا جایی که هر ناحیه فقط شامل یک نقطه باشد که یک برگ درخت کی‌دی را می‌سازد و می‌دانیم که در هر برگ اطلاعات آن نقطه ذخیره می‌شود. گره‌های داخلی هم حاوی مختصات خطوط افقی یا عمودی هستند.

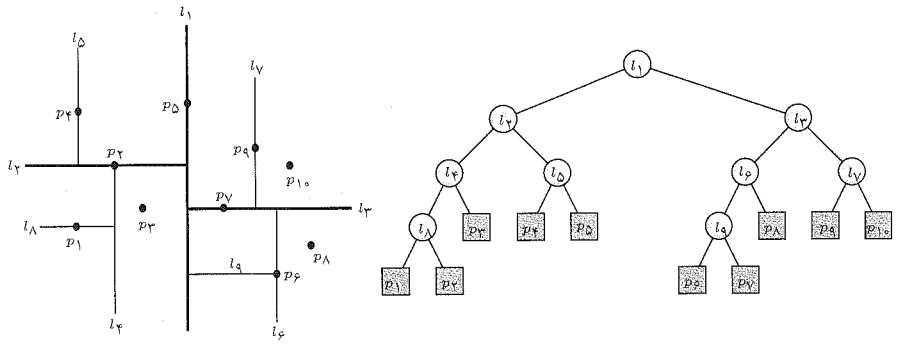
شکل ۷-۳۹ یک درخت کی‌دی را برای ۱۰ نقطه‌ی p_1 تا p_{10} نشان می‌دهد. روشن است که ارتفاع این درخت برای n نقطه $O(\lg n)$ است.

این درخت را می‌توان برای ابعاد بالاتر (k) هم تعمیم داد، در آن‌صورت برای ذخیره‌ی نقاط بُعد k (k-dimension) و پرس‌وجوهای در آن بُعد به‌کار می‌رود. نام k-d از این نکته گرفته شده است.

فرض‌ها: در این پروژه فرض می‌کنیم که:

۱. هیچ دو نقطه‌ای دارای مختصات x یا y یک‌سانی نیستند.

۲. تعداد نقاط حداکثر ۱۰۰,۰۰۰ و مختصات آن‌ها بین ۱۰۰۰۰۰− و ۱۰۰۰۰۰+ است.



شکل ۷-۳۹ یک درخت کی‌دی برای ۱۰ نقطه‌ی داده‌شده

ورودی و خروجی: ورودی دو فایل متنی است. فایل اولی حاوی نقاط و دومی حاوی تعداد زیادی دستورهای پرس‌وجو است. هر دستور پرس‌وجو یک پنجره را نشان می‌دهد. شما باید داده‌ساختار درخت کی‌دی را بر اساس اطلاعات فایل اول بسازید و اطلاعات درخت را بنویسید. سپس به‌ازای هر پرس‌وجو، نقاط داخل (و روی مرز) پنجره‌ی پرس‌وجو را به‌دست آورید و در فایل خروجی بنویسید. الگوریتم شما باید سریع باشد و نباید از $\mathcal{O}(\sqrt{n}+k)$ برای هر پرس‌وجو بیش‌تر باشد (k تعداد نقاط جواب برای آن پرس‌وجوست).

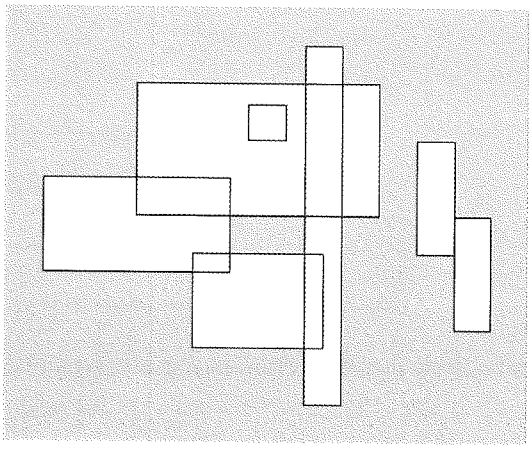
۳ پوسترها

n پوستر مستطیل‌شکل روی دیوار طوری نصب شده‌اند که اضلاع همه‌ی آن‌ها افقی و عمودی هستند. قسمتی از هر پوستر یا تمام آن می‌تواند با دیگر پوسترها پوشانده شده باشد. از اجتماع سطح‌های پوشیده شده‌ی پوسترها، یک ناحیه‌ی کلی به‌وجود می‌آید که لزوماً هم‌بند نیست. این ناحیه با سطح مکمل آن، مرز (نه لزوماً هم‌بند) دارد.

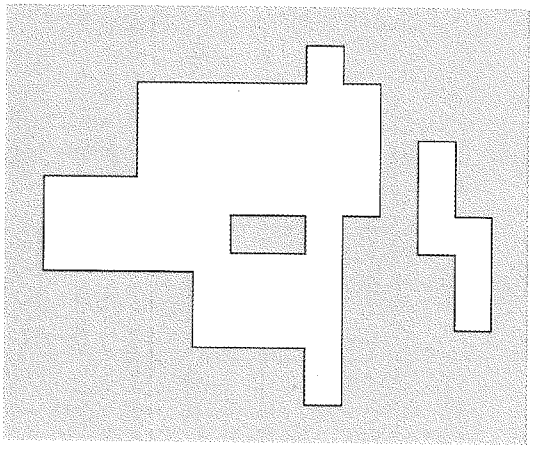
به‌عنوان مثال، شکل ۷-۴۰، حاوی ۷ پوستر است که ناحیه‌ی کلی و مرز آن‌ها در شکل ۷-۴۱ مشخص شده است.

برنامه‌ای بنویسید که با گرفتن اطلاعات مربوط به پوسترها، مساحت ناحیه‌ی کلی و طول مرز آن را به‌دست آورد.

ورودی: ورودی مسئله را ازورودی استاندارد بخوانید. در سطر اول ورودی، تنها n نوشته شده است. در هر یک از n سطر بعد، اطلاعات مربوط به یک پوستر آمده است. هر پوستر را با ۴ عدد صحیح که بین آن‌ها فاصله است نشان می‌دهیم. این اعداد به‌ترتیب مختصات نقطه‌ی پایین و سمت چپ و نیز



شکل ۷-۴۰ وضعیت اولیه‌ی ۷ پوستر روی دیوار.



شکل ۷-۴۱ ناحیه‌ی کلی پوسترهای روی دیوار.

بالای سمت راست آن پوستر است.

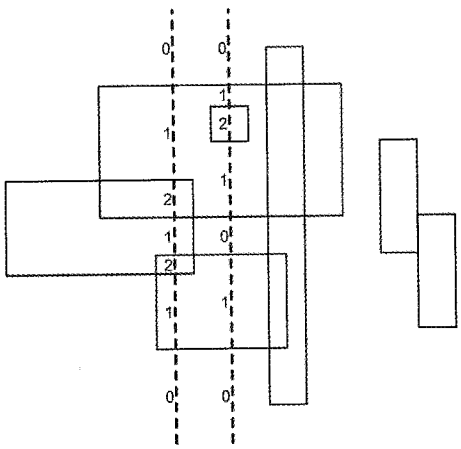
خروجی: خروجی خود را در خروجی استاندارد بنویسید. در سطر اول خروجی مساحت و در سطر دوم، طول مرز ناحیه‌ی کلی را بنویسید.

محدودیت:

- خروجی برنامه‌ها یک‌تا است.
- به برنامه‌ی شما ۵ ثانیه وقت برای اجرا داده می‌شود.
- $n \leq ۱۰۰۰۰۰$ و مختصات‌ها اعدادی صحیح و در محدوده‌ی $[-۱۰^۹, ۱۰^۹]$ هستند.
- تضمین شده که جواب‌ها هم در محدوده‌ی integer هستند.

راهنمایی: برنامه را می‌توان با زمان اجرای $O(n \lg n)$ نوشت. راه‌حل عموماً دارای محاسبه‌ی ۳ چیز است: مرزهای افقی، مرزهای عمودی و مساحت. به‌علت تشابه فقط به توضیح محاسبه‌ی مرزهای عمودی می‌پردازیم.

ایده‌ی حل، حرکت یک خط جاروب[۴۰] عمودی از سمت چپ صفحه به طرف راست است. با رسیدن خط به هر ضلع عمودی، طول قسمت‌هایی از آن‌را که در مرز است به‌دست می‌آوریم. برای این کار باید بدانیم که در وضعیت جاری، روی هر بازه از خط جاروب، چند پوستر وجود دارد. هرگاه این عدد برای بازه‌ای، از صفر به غیر صفر یا از غیرصفر به صفر تغییر کرد، به اندازه‌ی طول آن بازه به جواب اضافه می‌کنیم. دو وضعیت خط جاروب در شکل ۷-۴۲ دیده می‌شود.



شکل ۷-۴۲ دو وضعیت از خط جاروب.

[۴۰] sweep line

پیاده‌سازی باید به‌گونه‌ای انجام شود که اضافه شدن و کم شدن بازه‌ها در زمان $O(\lg n)$ انجام گیرد. برای این کار، باید از نوعی درخت دودویی استفاده کنید که هر گره آن نمایان‌گر یک بازه است. یک گره یا فرزند ندارد یا دارای دو فرزند است که در این صورت اگر نمایان‌گر بازه‌ی $[a, c]$ باشد، فرزند چپ آن نمایانگر $[a, b]$ و فرزند راست آن نمایان‌گر $[b, c]$ خواهد بود ($b \in [a, c]$).

برای اِعمال تغییرات مربوط به یک بازه (ناشی از شروع یا پایان یک پوستر)، کافی است تا عمقی از این درخت پایین بروید که بازه‌ی نمایان‌گر گره جاری، به‌وسیله‌ی آن بازه کاملاً پوشیده شود. برای گره متناظر با بازه‌ی $[a, b]$ نگه‌داشتن این مؤلفه‌ها منطقی است:

- اشاره‌گر به فرزندان چپ و راست،
- a و b،
- هم‌اکنون پوشش چند پوستر (متقاطع با خط جاروب) روی این گره اِعمال شده است و
- هم‌اکنون چه مقدار از بازه‌ی $[a, b]$ کلاً در این زیردرخت پوشانده شده است.

اگر دقت کنید متوجه می‌شوید که در هر مرحله تغییرات مؤلفه‌ی آخر در ریشه‌ی درخت است که به‌جواب اضافه می‌شود.

مثال: ورودی و خروجی مربوط به شکل ۷-۴۰ را در زیر می‌بینید.

input	output
7 -10 -1 0 4 -2 -5 5 0 4 -8 6 11 -5 2 8 9 1 6 3 8 10 0 12 6 12 -4 14 2	207 114

۴ بازیابی نقاط درون یک پنجره

n نقطه داده شده است. شما باید برنامه‌ای بنویسید که با مصرف حافظه و پیش‌پردازش کمی روی نقطه‌ها، بتواند پرسش‌هایی را در زمان کوتاه پاسخ دهد. هر پرسش به‌شکل یک پنجره‌ی مستطیل با اضلاع افقی و عمودی است و پاسخ آن مجموعه‌ای از n نقطه‌ی ورودی است که درون این پنجره قرار می‌گیرند. فقط ضلع‌های چپ و پایین مستطیل، داخل آن حساب می‌شوند.

ورودی: از ورودی استاندارد بخوانید. در سطر اول ورودی، n (تعداد نقطه‌ها) و q (تعداد پرسش‌ها) نوشته شده است. در سطر $i+1$ اُم ($1 \le i \le n$)، مختصات نقطه‌ی i ام آمده است.

در هر یک از q سطر بعد، یک پرسش به‌شکل «$cmd\ x_۱\ y_۱\ x_۲\ y_۲$» آمده است. $(x_۱, y_۱)$ مختصات نقطه‌ی پایین و سمت چپ پنجره، و $(x_۲, y_۲)$ مختصات نقطه‌ی بالا و سمت راست آن است. مقدار cmd، «0» یا «1» است و چگونگی پاسخ به پرسش را نشان می‌دهد. این مطلب در بخش خروجی توضیح داده می‌شود.

خروجی: در خروجی استاندارد بنویسید. خروجی باید شامل q سطر باشد. در سطر iاُم، پاسخ پرسشِ iام ورودی را بنویسید. اگر در این پرسش، مقدار cmd (که در بخش ورودی معرفی شد)، «0» بود، تنها تعداد نقطه‌های درون پنجره را در این سطر بنویسید. ولی اگر مقدار cmd، «1» بود، ابتدا تعداد نقطه‌های درون پنجره را بنویسید و سپس شماره‌ی این نقطه‌ها را با یک فاصله از هم، در خروجی بنویسید.

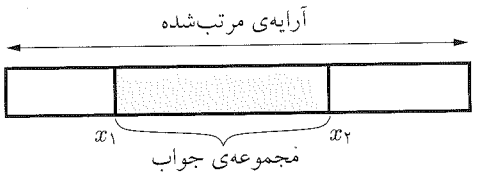
محدودیت:

- $۱ \le n, q \le ۱۰۰۰۰۰$.
- به برنامه‌ی شما ۵ ثانیه وقت برای اجرا داده می‌شود.

راهنمایی: می‌توان با صرف حافظه و پیش‌پردازش $\mathcal{O}(n \lg n)$، به پرسش‌های شمارشی $(cmd = 0)$ در زمان $\mathcal{O}(\lg n)$، و به بقیه‌ی پرسش‌ها $(cmd = 1)$ در زمان $\mathcal{O}(k + \lg n)$ پاسخ داد که k اندازه‌ی خروجی است.

ابتدا مسئله را در حالت یک بعدی حل می‌کنیم. در این حالت فقط محور xها را داریم و پرسش‌ها به‌شکل بازه‌های $[x_۱, x_۲]$ هستند.

ایده‌ی اول: نقطه‌ها را بر حسب x، به صورت صعودی مرتب می‌کنیم. برای پرسش $[x_۱, x_۲]$، ابتدا با جست‌وجوی دودویی و در زمان $\mathcal{O}(\log n)$، اندیس‌های $x_۱$ و $x_۲$ را در آرایه‌ی مرتب‌شده پیدا می‌کنیم. نقاط بین این دو اندیس در آرایه، جواب است. اگر بخواهیم فقط تعداد نقاط جواب را به‌دست آوریم، کافی است مقدار این دو اندیس را از هم کم کنیم و نتیجه را در خروجی بنویسیم که زمان پاسخ‌گویی در مجموع $\mathcal{O}(\lg n)$ می‌شود. اما اگر خود k نقطه‌ی جواب را هم بخواهیم در خروجی بنویسیم، زمان پاسخ‌گویی برابر $\mathcal{O}(k + \lg n)$ می‌شود.



ایده‌ی دوم: نقطه‌ها را با حساب x به عنوان کلید، در یک د.د.ج متوازن قرار می‌دهیم.[۴۱]

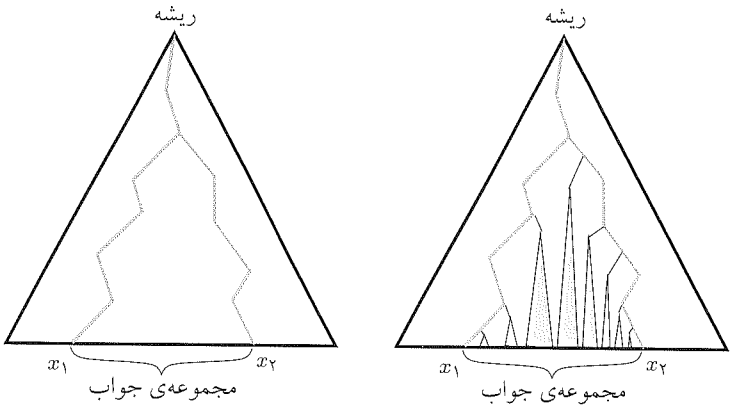
[۴۱] چون همه‌ی نقطه‌ها را از قبل می‌دانیم، هزینه‌ی پیاده‌سازی این کار خیلی کم‌تر از پیاده‌سازی کامل د.د.ج متوازن مثل قرمز-سیاه یا اِی.وی.اِل است.

برای راحتی پیاده‌سازی، بهتر است نقطه‌ها را تنها در برگ‌های درخت نگه داریم، و در هر گره داخلی درخت، تعداد برگ‌ها و همچنین مقدار بیش‌ترین و کم‌ترین x در زیردرخت آن گره را داشته باشیم. برای پرسش $[x_1, x_2]$، محل قرار گرفتن x_1 و x_2 را در این درخت به‌کمک خواص د.د.ج و در زمان $O(\lg n)$ پیدا می‌کنیم. این دو محل، دو مسیر از ریشه به برگ‌ها را نشان می‌دهند. نقاط جواب، در برگ‌های بین این دو مسیر قرار دارند.

اگر بخواهیم همه‌ی k نقطه‌ی جواب را خروجی دهیم، کافی است از محل x_1 روی درخت حرکت کنیم تا به محل x_2 برسیم. این کار در مجموع، هزینه‌ی $O(k + \lg n)$ را خواهد داشت. ولی اگر بخواهیم تنها تعداد نقطه‌های جواب را خروجی دهیم، لازم نیست روی همه‌ی گره‌های بین دو مسیر حرکت کنیم. اگر دقیق‌تر به این گره‌ها نگاه کنیم، می‌بینیم که این گره‌ها در قالب چند زیردرخت هستند که در شکل ۷–۴۳ نمایش داده شده‌اند. تعداد این زیردرخت‌ها $O(\lg n)$ است.

برای به‌دست آوردن تعداد نقطه‌های جواب، باید مجموع تعداد برگ‌های هر یک از این زیردرخت‌ها را به‌دست آوریم. با توجه به این‌که تعداد برگ‌های هر زیر درخت در ریشه‌ی آن ذخیره شده است، می‌توان در زمان $O(\lg n)$ جواب کلی را به‌دست آورد.

حال به مسئله‌ی دوبُعدی بازمی‌گردیم. راه‌حلی که ارائه می‌کنیم، از ایده‌ی دوم برای بُعدِ x، و از ایده‌ی اول برای بُعد y استفاده می‌کند. یعنی نقطه‌ها را بر اساس x در برگ‌های یک درخت دودویی متوازن قرار می‌دهیم و وقتی پرسشی به فرم $[x_1, x_2] \times [y_1, y_2]$ آمد، زیردرخت‌های رنگی بین مسیر ریشه تا x_1 و مسیر ریشه تا x_2 را پیدا می‌کنیم. ولی این زیردرخت‌ها را پیمایش نمی‌کنیم، چون تنها زیرمجموعه‌ای از نقطه (برگ)های این زیردرخت‌ها جواب ما هستند که yشان در بازه‌ی $[y_1, y_2]$ باشند. کافی است برای هر گره درخت، نقطه‌های درون زیردرخت آن گره را مرتب شده برحسب y در آرایه‌ای درون آن گره نگه داریم. در این صورت برای پاسخ به پرسش‌های دوبعدی، کافی است در



شکل ۷–۴۳ ایده‌ی دوم.

ریشه‌ی زیردرخت‌های رنگی، به سراغ آرایه‌ی مذکور برویم و زمان اجرا را تنها صرف نقاطی کنیم که از نظرِ y هم درون پنجره قرار می‌گیرند. اگر به این روش عمل کنیم زمان پاسخ‌گویی به پرسش‌ها $\mathcal{O}(\lg^{۲} n)$ (یا $\mathcal{O}(k + \lg^{۲} n)$) خواهد بود، چون در هر پرسش، $\mathcal{O}(\lg n)$ زیردرخت رنگی داریم، و در ریشه‌ی هر زیردرخت، زمان $\mathcal{O}(\lg n)$ صرف می‌کنیم تا محل $y_۱$ و $y_۲$ در آرایه‌ی آن گره پیدا شود.

میزان حافظه‌ی مصرفی چنین داده‌ساختاری $\mathcal{O}(n \lg n)$ است، چون هر نقطه حداکثر در آرایه‌ی $\mathcal{O}(\lg n)$ گره (تنها در مسیر از برگ تا ریشه) ظاهر می‌شود. ساخت درخت به همراه آرایه‌ها را نیز می‌توان در زمان $\mathcal{O}(n \lg n)$ انجام داد. آرایه‌ی نقطه‌های هر گره داخلی درخت (مشابه مرتب‌سازی ادغامی) از ادغام آرایه‌های دو فرزندش قابل محاسبه است.

نمونه

یک ورودی و خروجی ساده را در زیر می‌بینید.

Standard Input	Standard Output
6 4 4 2 2.5 1.5 5 2.5 1 1 2.5 2 4 1 1 2.5 1.5 5 3 0 2.5 1.5 5.5 3 1 1.5 -3 4 1.5 0 -10 -10 1000 1000.0001	3 2 5 1 4 0 6

۵ پیاده‌سازی فرهنگ‌نامه با داده‌ساختار تِرای

در این پروژه قرار است یک فرهنگ‌نامه را پیاده‌سازی و به‌کمک آن زبان انگلیسی را به زبان پنگلیسی[۴۲] و برعکس تبدیل کنید. زبان موسوم به پنگلیسی نگارش فارسی با نویسه‌ی انگلیسی است که در ایمیل‌ها و پیامک‌ها مرسوم شده است.

عملیاتی که در این فرهنگ‌نامه باید پشتیبانی شود عبارتند از:

- `declareEnglish` ⟨E⟩: دستور معرفی یک کلمه‌ی انگلیسی

با اجرای این دستور، کلمه‌ی ⟨E⟩ جزء کلمات انگلیسیِ شناخته‌شده قرار می‌گیرد.

[۴۲] Penglish

اگر این کلمه قبلاً شناخته‌شده بود، عبارت "already declared." و گرنه، عبارت "declare successfull." نوشته می‌شود.

مثال: declareEnglish milk

- ⟨P⟩ declarePenglish: دستور معرفی یک کلمه‌ی پنگلیسی

با اجرای این دستور، کلمه‌ی ⟨P⟩ جزء کلمات پنگلیسیِ شناخته‌شده قرار می‌گیرد. اگر این کلمه قبلاً شناخته‌شده بود، عبارت "already declared." و گرنه، عبارت "declare successfull." نوشته می‌شود.

مثال: declarePenglish shir

- ⟨P⟩ ⟨E⟩ addPair: دستور افزودن یک جفت معنی

با اجرای این دستور، کلمه‌ی ⟨E⟩، جزء کلمات انگلیسی شناخته‌شده، و کلمه‌ی ⟨P⟩، جزء کلمات پنگلیسی شناخته‌شده قرار می‌گیرند، اگر قبلاً شناخته‌شده نباشند. همچنین ⟨P⟩ و ⟨E⟩ هم‌معنی شناخته می‌شوند. اگر این دو کلمه قبلاً هم‌معنی معرفی شده باشند، عبارت "pair already exists." وگرنه، عبارت "addPair successfull." نوشته می‌شود.

مثال: addPair milk shir

- ⟨P⟩ ⟨E⟩ removePair: دستور حذف یک جفت معنی

با اجرای این دستور، کلمه‌ی انگلیسی ⟨E⟩، دیگر با کلمه‌ی پنگلیسی ⟨P⟩ هم‌معنی شناخته نمی‌شود. اگر این دو کلمه قبلاً نیز هم‌معنی نبودند، عبارت "no such pair exists." و گرنه، عبارت "removePair successfull." نوشته می‌شود. در هر صورت، ⟨E⟩ و ⟨P⟩ خود به‌تنهایی و در زبان خود، کلماتی شناخته‌شده باقی خواهند ماند (حتی اگر در زبان مقابل‌شان هیچ هم‌معنی‌ای نداشته باشند).

مثال: removePair milk shir

- ⟨E⟩ removeEnglish: دستور حذف یک کلمه‌ی انگلیسی

با اجرای این دستور، کلمه‌ی ⟨E⟩ دیگر جزء کلمات انگلیسیِ شناخته‌شده نمی‌باشد. هم‌چنین، این کلمه دیگر هم‌معنی هیچ کلمه‌ی پنگلیسی‌ای نیز نخواهد بود. اگر این کلمه از قبل شناخته‌شده نبود، عبارت "no such word exists." و گرنه، عبارت "removeEnglish successfull." نوشته می‌شود.

مثال: removeEnglish milk

- ⟨P⟩ removePenglish: دستور حذف یک کلمه‌ی پنگلیسی

با اجرای این دستور، کلمه‌ی ⟨P⟩ دیگر جزء کلمات پنگلیسی شناخته‌شده نمی‌باشد. همچنین، این کلمه دیگر هم‌معنی هیچ کلمه‌ی انگلیسی‌ای نیز نخواهد بود. اگر این کلمه از قبل شناخته‌شده نبود، عبارت "no such word exists." و گرنه، عبارت "removePenglish successfull." نوشته می‌شود.

مثال: removePenglish shir

- queryEnglish $\langle E \rangle$: پرسش یک کلمه‌ی انگلیسی
  پاسخ این پرسش، مجموعه‌ی کلمات پنگلیسی‌ای است که با $\langle E \rangle$ هم‌معنی هستند. اگر $\langle E \rangle$ یک کلمه‌ی انگلیسی شناخته‌شده نباشد، تنها علامت "?" و گرنه، کلمات هم‌معنی $\langle E \rangle$، به ترتیب الفبایی، با یک فاصله از هم نوشته می‌شوند.
  مثال: `queryEnglish milk`

- queryPenglish $\langle P \rangle$: پرسش یک کلمه‌ی پنگلیسی
  پاسخ این پرسش، مجموعه‌ی کلمات انگلیسی‌ای است که با $\langle P \rangle$ هم‌معنی هستند. اگر $\langle P \rangle$ یک کلمه‌ی پنگلیسی شناخته‌شده نباشد، تنها علامت "?" و گرنه، کلمات هم‌معنی $\langle P \rangle$، به ترتیب الفبایی، با یک فاصله از هم نوشته می‌شوند.
  مثال: `queryPenglish shir`

ورودی: از ورودی استاندارد بخوانید. در سطر اول ورودی، n نوشته شده که تعداد عمل‌های داده‌شده است. در هر یک از n سطر بعد، یکی از عمل‌هایی آمده است که معرفی شده‌اند. روند اجرای عملیات به‌همان ترتیبی می‌باشد که در ورودی آمده است.

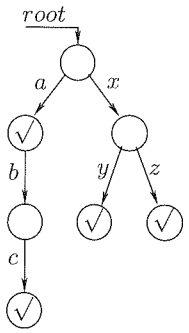
کلمات ورودی تنها شامل حروف کوچک انگلیسی (a تا z) و نویسه‌های «-» و «'» هستند. از نظر مقایسه در مرتب‌سازی کلمات، ترتیب این حروف به شکل $z < \cdots < a <$ «'» $<$ «-» است. در مرتب‌سازی کلمات از روش الفبایی استفاده کنید.

خروجی: در خروجی استاندارد بنویسید. خروجی باید شامل n سطر باشد. در سطر iاُم، نتیجه‌ی اجرای عملِ iاُم ورودی را به‌همان شکل که توضیح داده‌شده بنویسید.

محدودیت: ورودی به‌گونه‌ای است که اگر پیاده‌سازی خود را با داده‌ساختار تِرای انجام دهید، دچار مشکل زمان اجرا یا کمبود حافظه نخواهید شد.

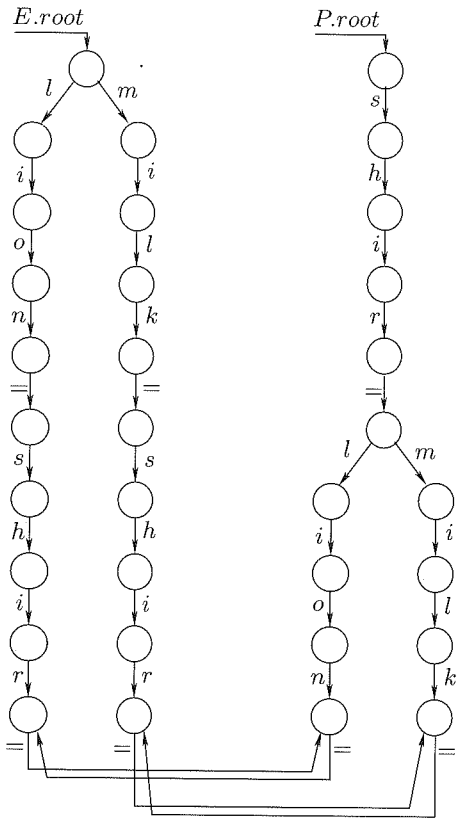
برای نگه‌داری اطلاعات کلمات، از داده‌ساختار تِرای استفاده کنید. ایده‌ی ساخت تِرای استفاده از یک درخت ریشه‌دار است که هر مسیر از ریشه‌ی آن به‌سمت پایین، متناظر با یک رشته‌ی خاص است. اگر رشته‌های ما تنها از حروف a، b، $\cdots$ تا z ساخته شده باشند، هر یک از گره‌های تِرای شامل یک مقدار منطقی به نام endFlag، ۲۶ اشاره‌گر به گره‌های فرزند با نام‌های $child['a']$، $child['b']$، $\cdots$ تا $child['z']$ خواهد بود. اشاره‌گرها به‌طور پیش‌فرض تهی و متغیر endFlag به‌طور پیش‌فرض false است. اگر برای هر گره، تنها اشاره‌گرهای با مقدار غیر تهی را نمایش دهیم، و حالت endFlag=true را با گذاشتن «√» روی آن گره نشان دهیم، از مجموعه کلمات $\{a, abc, xy, xz\}$ درختی به شکل ۷-۴۴ حاصل می‌شود.

برای نگه‌داری اطلاعات فرهنگ‌نامه، از دو تِرای استفاده کنید، یکی (E) برای انگلیسی به پنگلیسی و دیگری (P) برای پنگلیسی به انگلیسی. به‌غیر از نویسه‌های «-»، «'»، $a, b, \ldots, z$، نویسه‌ی «=» را هم به‌عنوان یک نویسه‌ی ممکن در رشته‌های تِرای در نظر بگیرید. روش استفاده از نویسه‌ی «=» برای نگه‌داری اطلاعات فرهنگ‌نامه را در شکل ۷-۴۵ مشاهده می‌کنید.



شکل ۷-۴۴ یک تِرای برای کلمات $\{a, abc, xy, xz\}$.

```
addPair milk shir
addPair lion shir
```



شکل ۷-۴۵ روش استفاده از نویسه‌ی «=» برای نگه‌داری اطلاعات فرهنگ‌نامه.

همان‌طور که مشاهده می‌کنید، می‌توان از اطلاعات $child['=']$ به جای endFlag نیز استفاده کرد. با اشاره‌گرهایی که در پایین درخت می‌بینید، می‌توان عملیات پاک کردن کامل یک کلمه در یک زبان را در زمان کوتاه‌تری انجام داد، که هزینه‌ی آن تنها با مجموع طول آن کلمه و طول کلمه‌های هم‌معنی آن کلمه متناسب است.

یک نکته‌ی مهم در عملیات حذف این است که اگر بخواهیم می‌توانیم در هنگام حذف رشته‌ها در تِرای، اندازه‌ی (تعداد گره‌های) این ساختار را کاهش دهیم. در یک تِرای ساده، گره‌ای که بچه‌ای ندارد و endFlag آن هم false است را می‌توان حذف کرد. پس از حذف این گره، پدر آن نیز ممکن است قابل حذف شود، و به‌همین روش، تا جای ممکن می‌توان بالا رفت. در طراحی خاصی از تِرای که برای فرهنگ‌نامه پیش‌نهاد شد، مفهوم endFlag نیز به‌شکل فرزند ظاهر می‌شود. پس در این روش، برای حذف یک گره، کافی است تنها بررسی کنیم که آیا فرزندی دارد یا خیر.

مثال: ورودی و خروجی مربوط به یک سناریوی ساده را در زیر ببینید.

Standard Input	Standard Output
18 queryEnglish let's addPair let's biaid queryEnglish let's declareEnglish let's queryEnglish biaid queryPenglish biaid removeEnglish biaid removePenglish biaid queryPenglish biaid queryEnglish let's addPair milk shir queryPenglish shir addPair lion shir queryPenglish shir queryEnglish milk removePenglish shir queryEnglish milk queryPenglish shir	? addPair successfull. biaid already declared. ? let's no such word exists. removePenglish successfull. ? addPair successfull. milk addPair successfull. lion milk shir removePenglish successfull. ?

پیوست‌ها

پیوست

۱ نمونه‌ای از برنامه‌ی جاوا

در این پیوست نمونه‌ای از پیاده‌سازی شیءها و راه‌حلی را می‌بینید که به زبان جاوا و برای مسئله‌ی زمان‌بندی چراغ راهنما که در فصل ۱ کتاب توضیح داده شد طراحی شده است.

```
package graph;

import java.util.*;

class Vertex {
  Vertex() {};
  Vertex(int num, boolean colored, int colorID) {
    this.num = num;
    this.colored = colored;
    this.colorID = colorID;
  }
  Vertex(int num) { this(num, false, 0); }

  private int num;
  public void setNum(int num) { this.num = num; }
  public int getNum() {return num; }

  private boolean colored = false;
  public void setColored(boolean colored) { this.colored = colored; }
  public boolean isColored() {return colored; }
```

```
  private int colorID;
  public void setColorID(int colorID) { this.colorID = colorID;
setColored(true); }
  public int getColorID() {return colorID;}

  public String toString() {
    return "node(" + num + "," + colored + "," + colorID + ")";
  }

  public boolean equals(Object o) {
    try {
      if ( (o.getClass() == Class.forName("graph.Vertex")) &&
          ( ( (Vertex) o).getNum() == num)) {
        return true;
      }
      else {
        return false;
      }
    } catch(Exception exp) {
      return false;
    }
  }
}

class Edge {
  Edge() {};
  Edge(Vertex v1, Vertex v2) {
    this.v1 = v1;
    this.v2 = v2;
  }

  private Vertex v1 = null;
  public void setV1(Vertex v1) { this.v1 = v1; }
  public Vertex getV1() {return v1; }

  private Vertex v2 = null;
  public void setV2(Vertex v2) { this.v2 = v2; }
  public Vertex getV2() {return v2; }

  public String toString() {
    return "Edge(" + v1.toString() + ", " + v2.toString() + ")";
  }

  public boolean equals(Object o) {
    try {
```

```
      if ( (o.getClass() == Class.forName("graph.Edge")) &&
        ((Edge) o).getV1().equals(v1) && ((Edge) o).getV2().equals(v2)){
        return true;
      }
      else {
        return false;
      }
    } catch(Exception exp) {
      return false;
    }
  }
}

class Graph {
  LinkedList nodes = new LinkedList();
  LinkedList edges = new LinkedList();

  Graph() {}



  public void addVertex(Vertex v) {
    if (!nodes.contains(v)) {
      nodes.add(v);
    }
  }

  public void removeVertex(Vertex ver) {
    for (Iterator i=nodes.iterator(); i.hasNext();) {
      Vertex v = (Vertex)i.next();
      if (v.equals(ver)) {
        nodes.remove(v);
        return;
      }
    }
  }

  public void addEdge(Edge e) {
    if (!edges.contains(e)) {
      e.setV1((Vertex)nodes.get(nodes.indexOf(e.getV1())));
      e.setV2((Vertex)nodes.get(nodes.indexOf(e.getV2())));
      edges.add(e);
    }
  }

  public void removeEdge(Edge edge) {
```

```
    for (Iterator i=edges.iterator(); i.hasNext();) {
      Edge e = (Edge)i.next();
      if (e.equals(edge)) {
        edges.remove(e);
        return;
      }
    }
  }

  public Iterator getNodes() { return nodes.iterator(); }

  public Iterator getAdjacent(Vertex v) {
    LinkedList adj = new LinkedList();
    for (Iterator i=edges.iterator(); i.hasNext();) {
      Edge e = (Edge) i.next();
      if (e.getV1().equals(v)) {
        adj.add(e.getV2());
      }
      if (e.getV2().equals(v)) {
        adj.add(e.getV1());
      }
    }
    return adj.iterator();
  }

  public String toString() {
    String temp = "graph with elements:(Vertices[";
    for (Iterator i=nodes.iterator(); i.hasNext();) {
      temp += ( (Vertex) i.next()).toString() + " ";
    }
    temp += "], Edges[";
    for (Iterator i=edges.iterator(); i.hasNext();) {
      temp += ( (Edge) i.next()).toString() + " ";
    }
    return temp + "])";
  }
}

public class GraphAlgorithms {
  public void greedyColoring(Graph g) {
    int colorID = 1;
    boolean finished;
    do {
      finished = true;
      for (Iterator i = g.getNodes(); i.hasNext(); ) {
        Vertex v = (Vertex) i.next();
```

```
        if (!v.isColored()) {
          finished = false;
          boolean isColorable = true;
          for (Iterator adj = g.getAdjacent(v); adj.hasNext(); ) {
            Vertex adjV = (Vertex) adj.next();
            if (adjV.isColored() && adjV.getColorID()==colorID) {
              isColorable = false;
            }
          }
          if (isColorable) {
            v.setColorID(colorID);
          }
        }
      }
      colorID++;
    } while (!finished);
  }

  public static void main(String[] args) {
    Graph g = new Graph();
    g.addVertex(new Vertex(1));
    g.addVertex(new Vertex(2));
    g.addVertex(new Vertex(3));
    g.addVertex(new Vertex(4));
    g.addVertex(new Vertex(5));

    g.addEdge(new Edge(new Vertex(1), new Vertex(2)));
    g.addEdge(new Edge(new Vertex(1), new Vertex(3)));
    g.addEdge(new Edge(new Vertex(3), new Vertex(5)));
    g.addEdge(new Edge(new Vertex(2), new Vertex(5)));
    g.addEdge(new Edge(new Vertex(4), new Vertex(1)));

    GraphAlgorithms graphAlgorithms = new GraphAlgorithms();
    graphAlgorithms.greedyColoring(g);
    System.out.println(g.toString());
  }

}
```

پیوست

۲

نمادها و تابع‌های مهم

در این بخش چند تابع را ارائه می‌دهیم که در این کتاب کاربرد زیادی دارند.

یک‌نوایی

یک تابع $f(n)$ اکیداً صعودی[۱] است اگر $m < n$ نتیجه دهد $f(m) < f(n)$ و اکیداً نزولی[۲] است اگر $m < n$ نتیجه دهد $f(m) > f(n)$. یک تابع $f(n)$ صعودی یک‌نوا[۳] است اگر $m \leq n$ نتیجه دهد $f(m) \leq f(n)$. هم‌چنین یک تابع $f(n)$ نزولی یک‌نوا[۴] است اگر $m \leq n$ نتیجه دهد $f(m) \geq f(n)$.

تابع‌های سقف و کف

برای هر عدد حقیقی x، بزرگ‌ترین عدد صحیح کوچک‌تر یا مساوی x را با $\lfloor x \rfloor$ نشان می‌دهیم و آن‌را تابع «کف[۵]» می‌نامیم، و کوچک‌ترین عدد صحیح بزرگ‌تر یا مساوی x را با $\lceil x \rceil$ نمایش می‌دهیم و

---

[۱] strictly increasing
[۲] strictly decreasing
[۳] monotically increasing
[۴] monotically decreasing
[۵] floor

آن‌را تابع «سقف[۶]» می‌نامیم و داریم

$$x - ۱ < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + ۱. \tag{۱-۲}$$

هم‌چنین برای هر عدد صحیح n، و برای هر عدد حقیقی $n \geq ۰$ و عددهای صحیح $a, b > ۰$، داریم

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil, \tag{۲-۲}$$
$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor, \tag{۳-۲}$$
$$\lceil a/b \rceil \leq (a + (b - ۱))/b, \tag{۴-۲}$$
$$\lfloor a/b \rfloor \geq (a - (b - ۱))/b. \tag{۵-۲}$$

تابع کف $f(x) = \lfloor x \rfloor$ و تابع سقف $f(x) = \lceil x \rceil$ صعودی یک‌نوا هستند.

حساب هم‌نهشتی

برای هر عدد صحیح a و هر عدد صحیح و مثبت n، مقدار $a \bmod n$ را «a در هم‌نهشتی n» یا به‌عبارت ساده و معمول‌تر «a به هنگ n» می‌گوییم و از آن با «حساب هم‌نهشتی[۷]» نام می‌بریم. این مقدار برابر با باقی‌مانده‌ی تقسیم صحیح a بر n است:

$$a \bmod n = a - \lfloor a/n \rfloor n. \tag{۶-۲}$$

با داشتن این نماد برای نمایش باقی‌مانده‌ی تقسیم صحیح دو عدد، می‌توان نماد خاصی برای بیان تساوی باقی‌مانده‌ها تعریف کرد:

> $(a \bmod n) = (b \bmod n)$ که به‌صورت $a \equiv b \pmod{n}$ نشان داده می‌شود، به معنی هم‌ارز[۸] بودن a و b در هم‌نهشتی n است. به‌عبارت دیگر، $a \equiv b \pmod{n}$ اگر a و b هنگام تقسیم‌شدن بر n دارای باقی‌مانده‌ی برابر باشند. هم‌چنین، $a \equiv b \pmod{n}$ اگر و تنها اگر n یک مقسوم‌علیه $b - a$ باشد. هرگاه a و b در هم‌نهشتی n هم‌ارز نباشند با $a \not\equiv b \pmod{n}$ نشان داده می‌شود.

تابع‌های چندجمله‌ای

برای عدد صحیح و غیر منفی d، یک چندجمله‌ای از n با درجه‌ی d تابع $p(n)$ به‌صورت

$$p(n) = \sum_{i=۰}^{d} a_i n^i \tag{۷-۲}$$

---

[۶] ceiling
[۷] modular arithmetic
[۸] equivalent

است که ثابت‌های a_0، a_1، ... تا a_d ضرایب چندجمله‌ای هستند و $a_d \neq 0$. یک چندجمله‌ای به‌صورت مجانبی[۹] مثبت است اگر و تنها اگر $a_d > 0$. برای هر تابع چندجمله‌ای $p(n)$ با درجه‌ی d که به‌صورت مجانبی مثبت است، داریم: $p(n) = \Theta(n^d)$. برای هر ثابت حقیقی $a > 0$ تابع n^a اکیداً صعودی و برای هر ثابت حقیقی $a < 0$ تابع n^a اکیداً نزولی است. تابع $f(n)$ «محدودشده‌ی چندجمله‌ای[۱۰]» است، هرگاه برای یک ثابت k رابطه‌ی

$$f(n) = O(n^k)$$

برقرار باشد.

فاکتوریل

نماد $n!$ که به صورت «n فاکتوریل» خوانده می‌شود، برای اعداد صحیح $n \geq 0$ به صورت زیر تعریف می‌شود:

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases} \qquad (۲-۸)$$

بنابراین: $n! = 1 \times 2 \times 3 \times \ldots \times n$. یک کران بالای ضعیف برای تابع فاکتوریل به‌صورت $n! \leq n^n$ است. تقریب استرلینگ[۱۱]،

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \qquad (۲-۹)$$

که e پایه‌ی لگاریتم طبیعی است، تقریب نزدیک‌تری[۱۲] برای کران بالا و پایین فاکتوریل به‌دست می‌آورد. هم‌چنین رابطه‌ی زیر برای $n \geq 1$ برقرار است:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \qquad (۲-۱۰)$$

که

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

تابع‌های نمایی

برای تمام اعداد حقیقی m و n و $a > 0$ تساوی‌های زیر برقرار است:

$$a^0 = 1,$$

---

[۹] asymptotically
[۱۰] polynomially bounded
[۱۱] Stirling
[۱۲] tighter

$$\begin{aligned} a^{۱} &= a, \\ a^{-۱} &= ۱/a, \\ (a^m)^n &= a^{mn}, \\ (a^m)^n &= (a^n)^m, \\ a^m a^n &= a^{m+n}. \end{aligned}$$

برای هر n و $a \geq ۱$ تابع a^n نسبت به n صعودی یک‌نوا است. در برخی موارد فرض می‌شود که $۰^۰ = ۱$.

رابطه‌ی بین رشد تابع‌های نمایی و چندجمله‌ای به این ترتیب است که برای تمام اعداد حقیقی $a, b > ۱$:

$$\lim_{n\to\infty} \frac{n^b}{a^n} = ۰ \tag{۱۱-۲}$$

که می‌توان از آن رابطه‌ی $n^b = o(a^n)$ را نتیجه گرفت. یعنی هر تابع نمایی با پایه‌ی بزرگ‌تر از ۱ سریع‌تر از هر تابع چندجمله‌ای رشد می‌کند.

با استفاده از e برای نمایش پایه‌ی لگاریتم طبیعی که برابر با ۲/۷۱۸۲۸ است، برای هر عدد حقیقی x داریم

$$e^x = ۱ + x + \frac{x^۲}{۲!} + \frac{x^۳}{۳!} + \cdots = \sum_{i=۰}^{\infty} \frac{x^i}{i!} \tag{۱۲-۲}$$

در ضمن برای هر عدد حقیقی x نامساوی زیر برقرار است:

$$e^x \geq ۱ + x \tag{۱۳-۲}$$

که در این رابطه، تساوی فقط برای $x = ۰$ وجود دارد. برای $|x| \leq ۱$ تقریب زیر برقرار است:

$$۱ + x \leq e^x \leq ۱ + x + x^۲. \tag{۱۴-۲}$$

و هرگاه $x \to ۰$، استفاده از $۱ + x$ به عنوان تقریب برای e^x کاملاً مناسب است:

$$e^x = ۱ + x + \Theta(x^۲). \tag{۱۵-۲}$$

برای هر x رابطه‌ی زیر نیز برقرار است:

$$\lim_{x\to\infty} (۱ + \frac{x}{n})^n = e^x. \tag{۱۶-۲}$$

تابع‌های لگاریتمی

از نمادهای زیر برای نمایش تابع‌های مربوط استفاده می‌شود:

$$\lg n = \log_۲ n,$$

$$\begin{aligned}\ln n &= \log_e n,\\ \lg^k n &= (\lg n)^k,\\ \lg\lg n &= \lg(\lg n).\end{aligned}$$

از این به‌بعد این‌طور قرارداد می‌کنیم که تابع‌های لگاریتمی فقط روی عبارت بعد از آن‌ها در فرمول اعمال می‌شوند. یعنی اولویت اجرای تابع lg از ضرب و تقسیم بیش‌تر و از توان کم‌تر است. بنابراین $\lg n + k$ به معنی $\lg(n) + k$ است. برای هر ثابت $b > 1$، $\log_b n$ برای $n > 0$ صعودی یک‌نوا است.

برای تمام اعداد حقیقی $a, b, c > 0$ و n رابطه‌های زیر برقرار است:

$$\begin{aligned}a &= b^{\log_b a},\\ \log_c(ab) &= \log_c a + \log_c b,\\ \log_b a^n &= n\log_b a,\\ \log_b a &= \frac{\log_c a}{\log_c b},\\ \log_b(1/a) &= -\log_b a,\\ \log_b a &= \frac{1}{\log_a b},\\ a^{\log_b c} &= c^{\log_b a}\end{aligned}$$

که در تمام رابطه‌های بالا، پایه‌ی لگاریتم مخالف ۱ فرض شده است.

از رابطه‌های بالا نتیجه می‌شود که تغییر پایه‌ی لگاریتم از یک ثابت به ثابت دیگر، فقط به‌صورت یک ضریب ثابت مقدار آن را تغییر می‌دهد. به‌همین دلیل هرجا که ضرایب ثابت مهم نباشند از lg و بدون ذکر پایه‌ی لگاریتم استفاده می‌شود. در رشته‌ی علم کامپیوتر عدد ۲ به عنوان پرکاربردترین پایه استفاده می‌شود زیرا در بسیاری از الگوریتم‌ها و داده‌ساختارها، مسئله به دو زیرمسئله شکسته می‌شود.

یک بسط مهم برای $\ln(1+x)$ به‌ازای $|x| < 1$ به‌صورت زیر است:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots \qquad (۲\text{-}۱۷)$$

هم‌چنین نامساوی زیر برای $x > -1$ وجود دارد:

$$\frac{x}{1+x} \le \ln(1+x) \le x \qquad (۲\text{-}۱۸)$$

که تساوی فقط برای $x = 0$ رخ می‌دهد.

تابع $f(n)$ «کران‌دار چندجمله‌ای لگاریتمی[۱۳]» است، هرگاه برای یک ثابت k، $f(n) = O(\lg^k n)$ باشد. رابطه‌ی بین رشد تابع‌های چندجمله‌ای و چندجمله‌ای لگاریتمی همانند رابطه‌ی بین توابع نمایی و چندجمله‌ای است و با جای‌گذاری $\lg n$ به‌جای n، و 2^a به‌جای a به‌دست می‌آید:

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

[۱۳] polylogarithmically bounded

و از آن می‌توان نتیجه گرفت که برای هر ثابت $a > 0$:

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

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

تکرار تابعی

نماد $f^{(i)}(n)$ برای نمایش تابع $f(n)$ که به‌طور متوالی i مرتبه روی مقدار اولیه‌ی n اعمال شده است استفاده می‌شود. مثلاً، $f(f(n)) = f^2(n)$. به‌طور رسمی، اگر $f(n)$ تابعی روی اعداد حقیقی باشد، برای $i \geq 0$ تابع $f^{(i)}(n)$ به صورت بازگشتی زیر تعریف می‌شود:

$$f^{(i)}(n) = \begin{cases} n, & i = 0 \\ f(f^{i-1}(n)), & i > 0 \end{cases} \qquad (۲-۱۹)$$

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

تابع لگاریتم تکراری

نماد $\lg^* n$ برای نمایش تابع‌های «لگاریتم تکراری[۱۴]» استفاده می‌شود. اگر $\lg^{(i)} n$ مطابق تعریف بالا به‌ازای $f(n) = \lg n$ باشد، از آن‌جا که لگاریتم برای اعداد منفی تعریف نشده است، $\lg^{(i)} n$ فقط در صورت $\lg^{(i-1)} n > 0$ تعریف شده است. برای حل این مشکل، تابع لگاریتم تکراری به‌صورت زیر تعریف می‌شود:

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\} \qquad (۲-۲۰)$$

تابع لگاریتم تکراری، یک تابع با سرعت رشد بسیار کم است:

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

قابل توجه است که تعداد اتم‌های دنیا تقریباً در حدود 10^{80} تخمین زده شده است که خیلی کوچک‌تر از 2^{65536} می‌باشد. به‌ندرت با ورودی‌هایی به‌اندازه‌ی n برخورد می‌کنیم که $\lg^* n > 5$ باشد.

[۱۴] iterated logarithm

اعداد فیبوناچی

اعداد فیبوناچی[۱۵] به‌وسیله‌ی رابطه‌ی بازگشتی زیر تعریف می‌شوند:

$$\begin{aligned} F_۰ &= ۰, \\ F_۱ &= ۱, \\ F_i &= F_{i-۱} + F_{i-۲} \quad : i \geq ۲ \end{aligned}$$

بنابراین هر عدد فیبوناچی برابر با مجموع دو عدد قبلی خود است که به دنباله‌ی زیر منجر می‌شود:

$$۰, ۱, ۱, ۲, ۳, ۵, ۸, ۱۳, \ldots$$

اعداد فیبوناچی مرتبط با نسبت طلایی ϕ و مزدوج آن $\hat{\phi}$ می‌باشند که به صورت زیر تعریف می‌شوند:

$$\begin{aligned} \phi &= \frac{۱ + \sqrt{۵}}{۲} = ۱/۶۱۸۰۳\ldots, \\ \hat{\phi} &= \frac{۱ - \sqrt{۵}}{۲} = -۰/۶۱۸۰۳\ldots, \\ F_i &= \frac{\phi^i - \hat{\phi}^i}{\sqrt{۵}} \end{aligned}$$

که تساوی آخری با استقرا قابل اثبات است. از آن‌جا که $|\hat{\phi}| < ۱$ نتیجه می‌شود که

$$|\hat{\phi}^i|/۵ < ۱/۵ < ۱/۲,$$

بنابراین iامین عدد فیبوناچی، F_i، برابر با $\phi^i/\sqrt{۵}$ است که به نزدیک‌ترین عدد صحیح گرد شده است. بنابراین اعداد فیبوناچی به‌صورت نمایی رشد می‌کنند.

---

[۱۵] Fibonacci numbers

پیوست

۳

واژه‌نامه‌ی فارسی به انگلیسی

ا

آخرین بازگشت ..... tail recursion
آخرین ورودی-اولین خروجی ..... last-in, first-out
آدرس‌دهی مستقیم ..... direct-addressing
آرایه ..... array
آرایه‌ی انجمنی ..... associative array
آرایه‌ی پُر ..... full array
آرایه‌ی پویا ..... dynamic array
آرایه‌ی سطلی ..... bucket array
اَبَرچاهک ..... supersink
اَبَرگراف ..... hypergraph
اَبَرمکعب ..... hypercube
اَبَرمنبع ..... supersource
اَبَریال ..... hyperedge
اثبات سازنده ..... constructive proof
اجتماع ..... union
اجتماع مجموعه ..... set union
احتمالی ..... probabilistic
ادغام k-راهه ..... k-way merge
ادغام چندراهه ..... multiway merge
ادغام چندمرحله‌ای ..... polyphase merge
ادغام خارجی ..... external merge
ادغام درجا ..... in-place merging
ادغام ..... merge
استقرا ..... induction
استقرای ضعیف ..... weak induction
استقرای قوی ..... strong induction
استلزام ..... implication
اشاره‌گر ..... pointer
اشاره‌گر اندیسی، مکان‌نما ..... cursor
اشتراک ..... intersection
اصل ردّ و شمول ... inclusion-exclusion principle
اصل صفر و یک ..... zero-one principle
اصل لانه‌کبوتری ..... pigeonhole principle
أعمال ..... operations
افراز ..... partition
افزایشی ..... incremental
افقی-عمودی ..... rectilinear
اقلیدس ..... Euclid
اقلیدسی ..... Euclidean
اکیداً صعودی ..... strictly increasing
اکیداً نزولی ..... strictly decreasing
الحاق ..... concatenation
الفبا ..... alphabet
الگوریتم ..... algorithm
الگوریتم ρ-تقریبی ... ρ-approximation algorithm
الگوریتم احتمالی ..... probabilistic algorithm
الگوریتم تقریبی ..... approximation algorithm
الگوریتم حریصانه ..... greedy algorithm

الگوریتم خارجی ..... external algorithm
الگوریتم دورگه (آمیخته) ..... hybrid algorithm
الگوریتم زمان چندجمله‌ای
polynomial-time algorithm
الگوریتم غیرقطعی ... nondeterministic algorithm
الگوریتم قطعی ..... deterministic algorithm
الگوریتم مرتب‌سازی ..... sorting algorithm
امضا (نشان) ..... signature
امکان‌پذیر ..... feasible
اُمگا ..... omega
اُمیکرون ..... omicron
ان‌پی-تمام ..... NP-complete
ان‌پی-سخت ..... NP-hard
انباشتگر ..... accumulator
انبوهه ..... aggregate
انتزاع ..... abstraction
انتقال پارامتر ..... parameter passing
اندیس (نمایه) آرایه ..... array index
انشعاب ..... branching
انشعاب خروجی ..... out-branching
انشعاب ورودی ..... in-branching
انشعاب و حد ..... branch and bound
اولویت ..... priority
اولین ورودی-اولین خروجی ..... first-in, first-out
ایستا ..... static

ب

ب‌م‌م دودویی ..... binary GCD
بازتابی ..... reflexive
بازگشت ..... recursion
بازگشت از یک فراخوانی ..... return
بازگشت تجمعی ..... collective recursion
بازگشتی‌شمارش‌پذیر ..... recursively enumerable
بازگشتی ..... recursive
بازمقیاس‌پذیر ..... rescalable
بازه ..... interval
بازه‌ی باز ..... open interval
بازه‌ی بسته ..... closed interval
بازه‌ی نیمه‌باز ..... semi-open interval
بخش‌بندی ..... partitioning
بخش ..... partition
بدترین حالت ..... worst-case
بدترین حالت سرشکن‌شده ... amortized worst case
برج‌های هانُوی ..... towers of Hanoi
برخط ..... on-line
برخورد ..... collision
بُرد ..... range

بردار بیتی ..... bit vector
بررسی مدل ..... model checking
بُرش ..... cut
برش رأسی کمینه ..... minimum vertex cut
برش کمینه ..... minimum cut
برگ ..... leaf
برنامه‌ریزی پویا ..... dynamic programming
برون‌خط ..... off-line
بزرگ‌ترین مخرج مشترک
greatest common denominator
بزرگ‌ترین مقسوم‌علیه مشترک (ب‌م‌م)
greatest common divisor
بسته‌بندی ..... bin packing
بسته‌بندی ..... encapsulation
بعدی ..... succesor
بقای شار ..... flow conservation
بلمن-فورد ..... Bellman-Ford
بلوک ..... block
بولی ..... boolean
بهترین حالت ..... best-case
به‌خاطرسپاری ..... memoization
بهین ..... optimal
بهینه‌سازی ..... optimization
بهینه ..... optimum
بهینه‌ی سراسری ..... global optimum
بیشین ..... maximal
بیشینه ..... maximum
بین‌ترتیب، میان‌ترتیب ..... inorder

پ

پادزنجیره ..... antichain
پادگانی، سلسله‌مراتبی ..... hierarchical
پادمتقارن ..... antisymmetric
پارامتر آدرسی ..... reference parameter
پارامتر ارزشی ..... value parameter
پایانه ..... terminal
پایدار ..... stable
پایه‌ی استقرا ..... induction basis
پتانسیل ..... potential
پدر-فرزندی ..... parent-child
پرسمان ..... query
پریش ..... derangement
پریم ..... Prim
پس‌ترتیب ..... postorder
پس‌گرد ..... backtracking
پس‌وندی ..... postfix
پُشته ..... stack

پشته‌ی کران‌دار ..... bounded stack
پل ..... bridge
پوشا ..... on-to
پوشش رأسی ..... vertex cover
پویا ..... dynamic
پیچیدگی ..... complexity
پیچیدگی الگوریتم ..... complexity of algorithm
پیچیدگی حافظه‌ی مجانبی
asymptotic space complexity
پیچیدگی زمان/حافظه ..... time/space complexity
پیچیدگی زمانی مجانبی
asymptotic time complexity
پیچیدگی مدار ..... circuit complexity
پیش ترتیب ..... preorder
پیش‌گو ..... oracle
پیش‌وندی ..... prefix
پیکربندی ..... configuration
پیمایش ..... traversal
پیوند ..... link

ت

تابع ..... function
تابع اَکِرمَن ..... Ackermann's function
تابع بازگشتی جزیی .... partial recursive function
تابع پتانسیل ..... potential function
تابع جزیی ..... partial function
تابع درهم‌ساز ..... hash function
تابع رشد ..... growth function
تابع گذار ..... transition function
تابع هدف ..... objective function
تابع یگانی ..... unary function
تام ..... total
تبدیل فوریه ..... Fourier transform
تبدیل فوریه‌ی سریع
Fast Fourier Transform (FFT)
تبدیل فوریه‌ی متناهی .... finite Fourier transform
تحلیل رقابتی ..... competitive analysis
تحلیل سرشکنی ..... amortized analysis
تحلیل پس‌سو ..... backward analysis
تِرای ..... trie
تِرای دودویی ..... binary trie
ترانهاده ..... transpose
ترایا (تراگذری) ..... transitive
ترتیب ..... order
ترتیب ..... permutation
ترتیب الفبایی ..... lexicographic Ordering
ترتیب توپولوژیکی ..... topological order
ترتیب جزیی ..... partial order

ترتیب خطی ..... linear order
ترتیب صعودی ..... ascending order
ترتیب تام ..... total order
ترتیب نزولی ..... descending order
ترسیم متعامد ..... orthogonal drawing
ترکیب ..... combination
ترکیب عطفی ..... conjunction
ترکیب فصلی ..... disjunction
ترکیبیات ..... combinatorics
ترمیم ..... relaxation
تشخیص‌دهنده (شناسنده) ..... recognizer
تصادفی‌سازی ..... randomization
تصادفی ..... randomized
تصمیم‌پذیر ..... decidable
تصمیم‌پذیر پاره‌ای ..... partially decidable
تصمیم چندراهه ..... multiway decision
تصمیم‌ناپذیر ..... undecidable
تطابق ..... matching
تطابق دوبخشی ..... bipartite matching
تطابق دوبخشی بیشینه
maximum bipartite matching
تطابق کامل ..... perfect matching
تفاضل ..... difference
تفاضل متقارن مجموعه ... symmetric set difference
تفاضل مجموعه ..... set difference
تقارن ..... symmetry
تقارن اریب ..... skew symmetry
تقاطع یالی ..... edge crossing
تقریب ..... approximation
تقریب استرلینگ ..... Stirling's approximation
تقسیم‌وحل ..... divide and conquer
تکرار ..... iteration
تکین ..... singular
تمام‌ماندگار ..... fully persistent
تناوب ..... alternation
تناوبی ..... alternating
تُنُک ..... sparse
تُنُک‌سازی ..... sparsification
توازن ..... balance
توپ و ظرف ..... balls and urns
تور ..... tour
تور اویلری ..... Euler tour
تورنمنت ..... tournament
توصیف ..... description

ث

ثبات ..... register

ج

جابه‌جایی ..... commutative
جاوا ..... Java
جای‌گذاری ..... layout
جبر پردازه‌ها ..... process algebra
جداسازی ..... separation
جدول درهم‌سازی ..... hash table
جدول نمادها ..... symbol table
جزء دو هم‌بند ..... biconnected component
جزء قویاً هم‌بند .. strongly connected component
جزء هم‌بند ..... connected component
جزیی‌مرتب ..... partially ordered
جست‌وجوی بهتر-اول ..... best first search
جست‌وجوی تجزیه‌پذیر .. decomposable searching
جست‌وجوی دودویی ..... binary search
جست‌وجوی سطح-اول ..... breadth first search
جست‌وجوی عمق-اول ..... depth first search
جست‌وجوی نزدیک‌ترین همسایه
nearest neighbor search
جنگل ..... forest
جنگل جست‌وجوی عمق-اول ..... DFS forest
جنگل مجموعه‌ی مجزا ..... disjoint-set forest
جهان (مجموعه‌ی مرجع) ..... universe

چ

چاهک ..... sink
چرخه ..... cycle
چرخه‌ی اویلری ..... Euler cycle
چرخه‌ی همیلتنی ..... Hamiltonian cycle
چندجمله‌ای ..... polynomial
چندضلعی ستاره‌ای ..... star-shaped polygon
چندلگاریتمی ..... polylogarithmic
چندوجهی ..... polyhedron

ح

حافظه‌ی اشتراکی ..... shared memory
حافظه‌ی خارجی ..... external memory
حالت ..... state
حالت پذیرفتنی ..... accepting state
حالت جامع ..... universal state
حالت شروع ..... start state
حالت میانگین ..... average-case
حذف ..... delete
حذف از پشته ..... pop
حریصانه ..... greedy
حساب‌داری ..... accounting
حساب ..... arithmetic
حل‌شدنی ..... solvable
حلقه‌ای ..... circular
حل‌نشدنی ..... unsolvable

خ

خاتمه‌ی بازگشت ..... recursion termination
خصیصه ..... attribute
خطی ..... linear
خودکاره ..... automaton
خودکاره‌ی پس‌وندی ..... suffix automaton
خودکاره‌ی پشته‌ای ..... pushdown automaton
خودکاره‌ی حالت متناهی ... finite state automaton
خودکاره‌ی درختی ..... tree automaton
خودکاره‌ی سلولی ..... cellular automaton
خودکاره‌ی متناهی ..... finite automaton
خوشه‌بندی ..... clustering
خوشه ..... clique

د

دادگان، داده‌پایگاه ..... database
داده‌ساختار ..... data structure
داده‌ساختار حافظه‌ی خارجی
external memory data structure
داده‌ساختار ساده ..... elementary data strucutre
داده‌ساختار ماندگار ..... persistent data structure
داده‌گونه ..... data type
داده‌گونه‌ی انتزاعی ..... abstract data type
داده‌گونه‌ی ساده ..... simple data type
داده‌گونه‌ی مرکب ..... compound data type
دامنه ..... domain
دومینو ..... domino
دایکسترا ..... Dijkstra
درایه ..... array entry
درج ..... insertion
درج در پشته ..... push
درج دودویی ..... binary insertion
درجا ..... in-place
درجه ..... degree
درجه‌ی خروجی ..... out-degree
درجه‌ی ۲ ..... quadratic
درجه‌ی ورودی ..... in-degree
درخت ۳-۲ ..... 2-3 tree

درخت آزاد ..... free tree
درخت اشتاینر ..... Steiner tree
درخت اِی.وی.اِل ..... AVL tree
درخت بازگشت ..... recursion tree
درخت بازی ..... game tree
درخت بی‌پلاس ..... B+-tree
درخت بی ..... B-tree
درخت تصمیم ..... decision tree
درخت جهت‌دار ..... directed tree
درخت دودویی پُر ..... full binary tree
درخت دودویی تقریباً کامل
nearly complete binary tree
درخت دودویی جست‌وجو ..... binary search tree
درخت دودویی کامل ..... complete binary tree
درخت دودویی معادل ..... equivalent binary tree
درخت اسپلی ..... splay tree
درخت عبارت ..... expression tree
درخت فراگیر، درخت پوشا ..... spanning tree
درخت فراگیر کمینه ..... minimum spanning tree
درخت قرمز-سیاه ..... red-black tree
درخت کامل ..... complete tree
درخت کلی ..... general tree
درخت کی‌دی ..... k-d tree
درخت مبنا ..... radix tree
درخت نیمه‌مرتب ..... partially ordered tree
درخت هافمن ..... Huffman tree
درستی‌سنجی ..... verification
درهم تنیدن ..... interleave
درهم‌سازی ..... hashing
درهم‌سازی باز ..... open hashing
درهم‌سازی پویا ..... dynamic hashing
درهم‌سازی دوگانه ..... double hashing
درهم‌سازی ساده و یک‌نواخت
simple uniform hashing
درهم‌سازی سراسری ..... universal hashing
درهم‌سازی کامل ..... perfect hashing
دست‌رسی ..... reachability
دستور ساده ..... simple statement
دُم ..... tail
دنباله ..... sequence
دنباله‌ی وارسی ..... probe sequence
دودویی ..... binary
دَوَران ..... rotation
دَوَران تکی ..... single rotation
دَوَران چپ‌گرد ..... left-rotate
دَوَران دوگانه ..... double rotation
دَوَران راست‌گرد ..... right-rotate
دوسویه ..... bidirectional
دو بُعدی ..... two-dimensional
دوگان ..... dual

ر

رابطه ..... relation
رابطه‌ی بازگشتی ..... recurrence relation
رابطه‌ی بازگشتی همگن
homogeneous recursive relation
رابطه‌ی بازگشتی ناهمگن
non-homogeneous recursive relation
رابطه‌ی مستقل از حلقه ..... loop invariant
رأس ..... vertex
رأس آزاد ..... free vertex
رأس اشتاینر ..... Steiner vertex
رأس بُرشی ..... cut vertex
رأس تطابق‌یافته ..... matched vertex
رأس‌ها ..... vertices
رام‌نشدنی ..... intractable
رایانش‌پذیر ..... computable
رایانش کوانتمی ..... quantum computation
رایانش‌ناپذیر ..... uncomputable
رد و شمول ..... inclusion-exclusoin
رد و شمول متقارن . symmetric inclusion-exclusoin
رده ..... class
رده‌ی پیچیدگی ..... complexity class
رده‌ی پیچیدگی متعارف (کانونی)
canonical complexity class
رسم گراف ..... graph drawing
رسمی ..... formal
رشته ..... string
رشته‌ی فشرده‌نشدنی ..... incompressible string
رقیب ..... adversary
رکورد ..... record
رنگ‌آمیزی رأسی ..... vertex coloring
رنگ‌آمیزی گراف ..... graph coloring
رنگ‌آمیزی مجاز ..... proper coloring
رنگ‌آمیزی یالی ..... edge coloring
روش انبوهه ..... aggregate method
روش پس‌گرد ..... backtracking
روش حساب‌داری ..... accounting method
روش رسمی ..... formal method
روش سطل‌بندی ..... bucketing method
رویه ..... procedure
ریشه ..... root

ز

زباله‌روبی ..... garbage collection

زبان ..... language
زبان رسمی (صوری) ..... formal language
زبَر مجموعه ..... superset
زمان‌بندی ..... scheduling
زمان اتمام آخرین کار ..... makespan
زمان اجرا ..... run time
زمان انتظار ..... waiting time
زمان پاسخ ..... response time
زمان چندجمله‌ای ..... polynomial time
زمان چندجمله‌ای غیرقطعی
nondeterministic polynomial time
زنجیره‌ای ..... chaining
زنجیره ..... chain
زنجیره‌ی مارکوف ..... Markov chain
زیردرخت ..... subtree
زیرگراف ..... subgraph
زیرمجموعه ..... subset
زیرمجموعه‌ی سره ..... proper subset

ژ

ژنرال‌های رومی ..... Byzantine generals

س

ساخت‌یافته ..... structured
ساختار رابطه‌ای ..... relational structure
ساده‌سازیِ ترایا ..... transitive reduction
سرآیند، سرلیست ..... header, sentinel
سرریز ..... overflow
سرشکن‌شده ..... amortized
سر ..... head
سره ..... proper
سطح‌ترتیب ..... level-order
سطل ..... bucket
سقف ..... ceiling
سلسله‌مراتبی، پایگانی ..... hierarchical
سلولی ..... cellular
سه‌تایی ..... trinary
سیاه‌ـارتفاع ..... black-height

ش

شار ..... flow
شار انسدادی ..... blocking flow
شار چندکالایی ..... multi-commodity flow
شبکه ..... lattice
شبکه‌ی شار ..... flow network
شبه‌تصادفی ..... pseudo-random
شبه چندجمله‌ای ..... psuedo poluynomial
شبه‌کد ..... pseudo-code
شبیه‌سازی ..... simulation
شرکت‌پذیری (انجمنی) ..... associative
شمارنده ..... counter
شمارنده‌ی برنامه ..... program counter
شمای تقریب چندجمله‌ای
polynomial approximation scheme
شمول و عدم شمول ..... inclusion exclusion
شیب ..... slope
شئ‌گرا ..... object oriented

ص

صعودی یکنواخت ..... monotonically increasing
صف ..... queue
صف اولویت ..... priority queue
صف اولویت دودویی ..... binary priority queue
صف حلقه‌ای ..... circular queue
صف دوسر ..... deque
صف دوسر ..... doubly-ended queue
صف کران‌دار ..... bounded queue
صفحه‌ی بُرشی ..... cutting plane
صفرتایی ..... nullary

ض

ضرب خطی ..... linear product
ضمانت کارایی ..... performance guarantee

ط

طوقه ..... self-loop

ظ

ظرفیت ..... capacity

ع

عبارت ..... expression
عدد رنگی ..... chromatic number
عدد فیبوناچی ..... Fibonacci number

عدد کاتالان ..... Catalan's number
عمق ..... depth
عمل‌گر ..... operator
عمل‌گر دودویی ..... binary operator
عمل‌گر یگانی ..... unary operator
عمل‌وند ..... operands
عملیات ..... operation

غ

غلط‌یاب ..... spell-checker
غیرقطعی ..... nondeterministic

ف

فازی، نادقیق ..... fuzzy
فاصله‌ی افقی-عمودی ..... rectilinear distance
فاصله‌ی اقلیدسی ..... Euclidean distance
فاصله‌ی منهتنی ..... Manhattan distance
فاکتوریل ..... factorial
فراخوانی ..... call
فرترن ..... Fortran
فرزند ..... child
فرض استقرا ..... induction hypothesis
فرکتال ..... fractal
فرمول استرلینگ ..... Stirling's formula
فروشنده‌ی دوره‌گرد ..... traveling salesman
فروشنده‌ی دوره‌گرد اقلیدسی
Euclidean traveling salesman
فرهنگ داده‌ای ..... dictionary
فشرده‌سازی ..... compaction
فشرده‌سازی مسیر ..... path compression
فشرده‌نشدنی ..... incompressible
فلاویوس ژوزفوس ..... Flavius Josephus
فلوید-وارشال ..... Floyd-Warshall
فورد-فالکرسن ..... Ford-Fulkerson
فیلسوف‌های گرسنه ..... dining philosophers

ق

قابل دست‌یابی ..... reachable
قابل دید ..... visible
قانون هورنر ..... Horner's rule
قبلی ..... predecessor
قضیه‌ی باقی‌مانده‌ی چینی
Chinese remainder theorem
قضیه‌ی بُرشی ..... cutting theorem
قضیه‌ی کوک ..... Cook's theorem
قطر ..... diameter
قطری کردن ..... diagonalization
قطعی ..... deterministic
قویاً هم‌بند ..... strongly connected
قید ..... constraint

ک

کارایی ..... efficiency
کارت منگنه ..... punched card
کامل ..... total
کانولوژن ..... convolution
کاهش، ساده‌سازی ..... reduction
کاهش چندبه‌یک ..... many-one reduction
کد گری ..... Gray code
کران ..... bound
کران بالای مجانبی ..... asymptotic upper bound
کران پایین ..... lower bound
کران پایین مجانبی ..... asymptotic lower bound
کران چندجمله‌ای ..... polynomially bounded
کران چندجمله‌ای لگاریتمی
polylogarithmically bounded
کران بسته‌ی مجانبی .. asymptotically tight bound
کران مجانبی ..... asymptotic bound
کروسکال ..... Kruskal
کف ..... floor
کلید ..... key
کمان ..... arc
کمکی ..... auxiliary
کمین ..... minimal
کمینه‌سازی ..... minimization
کمینه ..... minimum
کوتاه‌ترین مسیر ..... shortest path
کوتاه‌ترین مسیر از یک مبدأ
single-source shortest-path
کوتاه‌ترین مسیر به یک مقصد
single-destination shortest-path
کوتاه‌ترین مسیر بین هر دو رأس
all pairs shortest path
کوتاه‌ترین مسیر بین یک زوج‌رأس
single-pair shortest-path
کوچک‌ترین مستطیل دربرگیرنده
minimum bounding box
کوچک‌ترین مضرب مشترک least common multiple
کورکورانه ..... brute force
کو اِن‌پی ..... coNP

کوله‌پشتی پیوسته ..... continuous knapsack
کوله‌پشتی تقسیم‌پذیر ..... fractional knapsack
کوله‌پشتی صفر و یک ..... 0-1 knapsack

گ

گام استقرا ..... induction step
گذار حالت ..... state transition
گذار (تغییر وضعیت) ..... transition
گراف ..... graph
گراف اویلری ..... Eulerian graph
گراف برچسب‌دار ..... labeled graph
گراف تقاطع ..... conflict graph
گراف تُنُک ..... sparse graph
گراف جزیی پویا ..... partially dynamic graph
گراف جهت‌دار ..... digraph
گراف جهت‌دار ..... directed graph
گراف جهت‌دار ..... oriented graph
گراف چندگانه ..... multigraph
گراف دوبخشی ..... bipartite graph
گراف دوهم‌بند ..... biconnected graph
گراف جهت‌دار ناچرخه‌ای .. directed acyclic graph
گراف غیرجهت‌دار ..... undirected graph
گراف قویاً هم‌بند ..... strongly connected graph
گراف کامل ..... complete graph
گراف لایه‌شده ..... layered graph
گراف متراکم ..... dense graph
گراف مسطح ..... planar graph
گراف مسطح با خطوط مستقیم planar straight-line graph
گراف ناچرخه‌ای ..... acyclic graph
گراف وزن‌دار ..... weighted graph
گراف هم‌بند ..... connected graph
گردایه (مجموعه‌ی چندگانه) ..... bag
گردش ..... turn
گرد کردن تصادفی ..... randomized rounding
گره ..... node
گره داخلی ..... internal node
گره متوازن ..... balanced node
گسترش ..... expansion
گشت ..... walk
گلوگاه ..... bottleneck
گمارش ..... assignment
گواهی ..... certificate

ل

لانه‌کبوتری ..... pigeonhole
لگاریتم تکراری ..... iterated logarithm
لگاریتمی ..... logarithmic
لیست ..... list
لیست پیوندی ..... linked list
لیست پیوندی دوسویه ..... doubly linked list
لیست پیوندی دوسویه ..... two-way linked list
لیست پیوندی متقارن .... symmetrically linked list
لیست پیوندی مرتب ..... ordered linked list
لیست پیوندی یک‌سویه ..... singly linked list
لیست حلقه‌ای ..... circular list
لیست دوسویه ..... two-way list
لیست شرکت‌پذیری ..... association list
لیست کلی ..... general list
لیست نامرتب ..... unsorted list
لیست هم‌جواری ..... adjacency list

م

ماتریس ..... matrix
ماتریس اکیداً بالا مثلثی strictly upper triangular matrix
ماتریس اکیداً پایین مثلثی strictly lower triangular matrix
ماتریس بالامثلثی ..... upper triangular matrix
ماتریس پایین‌مثلثی ..... lower triangular matrix
ماتریس تُنُک ..... sparse matrix
ماتریس مربعی ..... square matrix
ماتریس مستطیلی ..... rectangular matrix
ماتریس ناهموار ..... ragged matrix
ماتریس هم‌جواری ..... adjacency matrix
ماتریس یک‌نواخت ..... uniform matrix
ماشین با دست‌رسی تصادفی random access machine
ماشین تورینگ ..... Turing machine
ماشین حالت ..... state machine
ماشین حالت متناهی ..... finite state machine
ماندگار ..... persistent
مرتبه ..... order
مبتنی بر درون ..... interior-based
مبدِل ..... transducer
متقارن ..... symmetric
متن ..... text
مثلث‌بندی ..... triangulation
مجانبی ..... asymptotically
مجزا ..... disjoint
مجموع ..... aggregate

مجموعه ..... set
مجموعه‌های مجزا ..... disjoint find-merge
مجموعه‌های مجزا ..... disjoint set
مجموعه‌ی جزیی‌مرتب ..... poset
مجموعه‌ی چندگانه ..... multi-set
مجموعه‌ی مستقل ..... independent set
مجموعه‌ی مستقل بیشینه maximum independent set
محاوره‌ای ..... interactive
محور ..... pivot
مدار ..... circuit
مُد ..... mode
مدل رایانش ..... model of computation
مدل مارکوف مخفی ..... hidden Markov model
مربع لاتین ..... latin square
مرتب‌سازی ادغامی ..... merge sort
مرتب‌سازی ادغامی kـراهه ..... k-way merge sort
مرتب‌سازی ادغامی چندفازه .. polyphase merge sort
مرتب‌سازی ادغامی خارجی .... external merge sort
مرتب‌سازی ادغامی دوراهه .... two-way merge sort
مرتب‌سازی ادغامی نوسانی .. oscillating merge sort
مرتب‌سازی انتخابی ..... selection sort
مرتب‌سازی بازه‌ای ..... range sort
مرتب‌سازی تطابقی ..... adaptive sort
مرتب‌سازی تعویضی ..... exchange sort
مرتب‌سازی توپولوژیکی ..... topological sort
مرتب‌سازی توزیعی ..... distribution sort
مرتب‌سازی حبابی ..... bubble sort
مرتب‌سازی حبابی دوسویه bidirectional bubble sort
مرتب‌سازی حبابی دوسویه
double-direction bubble sort
مرتب‌سازی خارجی ..... external sort
مرتب‌سازی خشتی ..... brick sort
مرتب‌سازی خودآزما ..... introspection sort
مرتب‌سازی داخلی ..... internal sort
مرتب‌سازی درجا ..... in-place sort
مرتب‌سازی درجی ..... insertion sort
مرتب‌سازی درجی خطی ..... linear insertion sort
مرتب‌سازی درجی دودویی ... binary insertion sort
مرتب‌سازی درجی مستقیم .. straight insertion sort
مرتب‌سازی درختی ..... tree sort
مرتب‌سازی ساده‌لوحانه ..... stooge sort
مرتب‌سازی سریع ..... quicksort
مرتب‌سازی سریع خارجی ..... external quicksort
مرتب‌سازی سریع متوازن ..... balanced quicksort
مرتب‌سازی سطلی ..... bin sort
مرتب‌سازی سطلی ..... bucket sort
مرتب‌سازی شمارشی ..... counting sort
مرتب‌سازی صدفی ..... shell sort
مرتب‌سازی غیر مقایسه‌ای .... non-comparison sort

مرتب‌سازی مبنایی ..... radix sort
مرتب‌سازی مقایسه‌ای ..... comparison sort
مرتب‌سازی وارسی خطی ..... linear probing sort
مرتب‌سازی هرمی ..... heapsort
مرتبه‌ی آماری ..... order statistics
مرکز ثقل ..... centroid
مسئله‌ی ارزش مدار ..... circuit value problem
مسئله‌ی پستچی چینی .. Chinese postman problem
مسئله‌ی تصمیم‌گیری ..... decision problem
مسئله‌ی تصمیم‌ناپذیر ..... undecidable problem
مسئله‌ی توقف ..... halting problem
مسئله‌ی گمارش ..... assignment problem
مسطح‌سازی ..... planarization
مسطح ..... planar
مسطح بودن ..... planarity
مسیر ..... path
مسیر افزایشی ..... augmenting path
مسیر اویلری ..... Eulerian path
مسیر بحرانی ..... critical path
مسیر تناوبی ..... alternating path
مسیر داخلی ..... internal path
مسیر حلقه‌ای ..... circular path
مسیر ساده ..... simple path
معادله‌ی سرشت‌نما ..... characteristic equation
معاهده‌ی رومی ..... Byzantine Agreement
معمای اسیران ..... prisoner's dilemma
مَفصَل ..... articulation vertex
مقعر ..... concave
مکاشفه‌ای ..... heuristic
مکاشفه‌ی حریصانه ..... greedy heuristic
مکان تسهیلات ..... facility location
مکان تسهیلات با ظرفیت محدود
capacitated facility location
مکمل ..... complement
مکمل مجموعه ..... set complement
منبع، مبدأ ..... source
منطق زمانی ..... temporal logic
منظم ..... regular
مولِّد اعداد تصادفی .... random number generator
مؤلفه‌ی بیشین‌هم‌بند
maximally connected component
مؤلفه‌ی دوهم‌بند ..... biconnected component
مؤلفه‌ی هم‌بند ..... connected component
مهارشدنی ..... tractable
میان‌گیر ..... buffer
میان‌وندی ..... infix
میان‌وندی با پرانتزی کامل
infix with complete paranthesis
میانا، واسط ..... interface

mean ..... میانگین
median ..... میانه

ن

irreflexive ..... نابازتاب
unstable ..... ناپایدار
acyclic ..... ناچرخه‌ای
unbalanced ..... نامتوازن
triangle inequality ..... نامساوی مثلثی
monotonically decreasing ..... نزولی یکنواخت
competitive ratio ..... نسبت رقابتی
golden ratio ..... نسبت طلایی
performance ratio ..... نسبت کارایی
visibility map ..... نقشه‌ی دید
Karnaugh map ..... نقشه‌ی کارنو
extreme point ..... نقطه‌ی کُنج
negation ..... نقیض (نفی)
mapping ..... نگاشت
O notation ..... نماد اُیِ بزرگ
big-O notation ..... نماد اُیِ بزرگ
little-o notation ..... نماد اُیِ کوچک
o notation ..... نماد اُیِ کوچک
postfix notation ..... نماد پس‌وندی
prefix notation ..... نماد پیش‌وندی
infix notation ..... نماد میان‌وندی
exponential ..... نمایی
moderately exponential ..... نمایی ملایم
Venn diagram ..... نمودار وِن
descendant ..... نواده
ancestor ..... نیا
bisector ..... نیم‌ساز

و

and ..... و
probing ..... وارسی
linear probing ..... وارسی خطی
quadratic probing ..... وارسی درجه‌ی ۲

ه

pruning ..... هَرَس
prune and search ..... هَرَس و جست‌وجو
max-heap ..... هرم بیشینه
binary heap ..... هرم دودویی
min-heap ..... هرم کمینه
cost ..... هزینه
amortized cost ..... هزینه‌ی سرشکن‌شده
equivalent ..... هم‌ارز
connectivity ..... هم‌بندی
vertex connectivity ..... هم‌بندی رأسی
edge connectivity ..... هم‌بندی یالی
overlap ..... هم‌پوشانی
adjacent ..... هم‌جوار
adjacency ..... هم‌جواری
homomorphic ..... هم‌ریخت
adjacent ..... هم‌سایه
modular ..... هم‌نهشتی
homeomorphic ..... همسان‌ریخت
computational geometry ..... هندسه‌ی محاسباتی
geometric ..... هندسی

ی

or ..... یا
edge ..... یال
free edge ..... یال آزاد
saturated edge ..... یال اشباع‌شده
feedback edge ..... یال بازخور (پس‌خورد)
matched edge ..... یال تطابق‌یافته
fuzzy edge ..... یال فازی
xor ..... یای انحصاری
exclusive or ..... یای انحصاری
inclusive or ..... یای منطقی
one-to-one ..... یک‌به‌یک
isomorphic ..... یک‌ریخت
graph isomorphism ..... یک‌ریختی گراف‌ها
one-dimensional ..... یک‌بُعدی
uniform ..... یک‌نواخت
monoticity ..... یک‌نوایی
subgraph isomorphism ..... یکریختی زیرگراف‌ها
unique ..... یگانه
element uniqueness ..... یگانی اعضا
unary ..... یگانی

سایر

k-dimensional ..... k-بُعدی
k-coloring ..... k-رنگ‌آمیزی
kth shortest path ..... k اُمین کوتاه‌ترین مسیر
discrete p-center ..... p-مرکز گسسته

پیوست

۴

واژه‌نامه‌ی انگلیسی به فارسی

A

abstract data type ..... داده‌گونه‌ی انتزاعی
abstraction ..... انتزاع
accepting state ..... حالت پذیرفتنی
accounting ..... حساب‌داری
accounting method ..... روش حساب‌داری
accumulator ..... انباشتگر
Ackermann's function ..... تابع اَکِرمَن
acyclic ..... ناچرخه‌ای
acyclic graph ..... گراف ناچرخه‌ای
adaptive sort ..... مرتب‌سازی تطابقی
adjacency ..... هم‌جواری
adjacency list ..... لیست هم‌جواری
adjacency matrix ..... ماتریس هم‌جواری
adjacent ..... هم‌جوار، هم‌سایه
adversary ..... رقیب
aggregate ..... انبوهه
aggregate ..... مجموع
aggregate method ..... روش انبوهه
algorithm ..... الگوریتم
all pairs shortest path
کوتاه‌ترین مسیر بین هر دو رأس
alphabet ..... الفبا
alternating ..... تناوبی
alternating path ..... مسیر تناوبی
alternation ..... تناوب
amortized ..... سرشکن‌شده
amortized analysis ..... تحلیل سرشکنی
amortized cost ..... هزینه‌ی سرشکن‌شده
amortized worst case ... بدترین حالت سرشکن‌شده
ancestor ..... نیا
and ..... و
antichain ..... پادزنجیره
antisymmetric ..... پادمتقارن
approximation ..... تقریب
approximation algorithm ..... الگوریتم تقریبی
arc ..... کمان
arithmetic ..... حساب
array ..... آرایه
array entry ..... درایه
array index ..... اندیس (نمایه) آرایه
articulation vertex ..... مَفصَل
ascending order ..... ترتیب صعودی
assignment ..... گمارش
assignment problem ..... مسئله‌ی گمارش
association list ..... لیست شرکت‌پذیری
associative ..... شرکت‌پذیری (انجمنی)
associative array ..... آرایه‌ی انجمنی

asymptotic bound ..... کران مجانبی
asymptotic lower bound ..... کران پایین مجانبی
asymptotic space complexity
پیچیدگی حافظه‌ی مجانبی
asymptotic time complexity
پیچیدگی زمانی مجانبی
asymptotic upper bound ..... کران بالای مجانبی
asymptotically ..... مجانبی
asymptotically tight bound .. کران بسته‌ی مجانبی
attribute ..... خصیصه
augmenting path ..... مسیر افزایشی
automaton ..... خودکاره
auxiliary ..... کمکی
average-case ..... حالت میانگین
AVL tree ..... درخت اِی.وی.اِل

B

B+-tree ..... درخت بی‌پلاس
B-tree ..... درخت بی
backtracking ..... پس‌گرد
backtracking ..... روش پس‌گرد
backward analysis ..... تحلیل پس‌سو
bag ..... گردایه (مجموعه‌ی چندگانه)
balance ..... توازن
balanced node ..... گره متوازن
balanced quicksort ..... مرتب‌سازی سریع متوازن
balls and urns ..... توپ و ظرف
Bellman-Ford ..... بلمن-فورد
best first search ..... جست‌وجوی بهتر-اول
best-case ..... بهترین حالت
biconnected component ..... جزء دو هم‌بند
biconnected component ..... مؤلفه‌ی دوهم‌بند
biconnected graph ..... گراف دوهم‌بند
bidirectional ..... دوسویه
bidirectional bubble sort مرتب‌سازی حبابی دوسویه
big-O notation ..... نماد اُیِ بزرگ
bin packing ..... بسته‌بندی
bin sort ..... مرتب‌سازی سطلی
binary ..... دودویی
binary GCD ..... ب.م.م دودویی
binary heap ..... هرم دودویی
binary insertion ..... درج دودویی
binary insertion sort ... مرتب‌سازی درجی دودویی
binary operator ..... عمل‌گر دودویی
binary priority queue ..... صف اولویت دودویی
binary search ..... جست‌وجوی دودویی
binary search tree ..... درخت دودویی جست‌وجو
binary trie ..... تِرای دودویی
bipartite graph ..... گراف دوبخشی
bipartite matching ..... تطابق دوبخشی
bisector ..... نیم‌ساز
bit vector ..... بردار بیتی
black-height ..... سیاه-ارتفاع
block ..... بلوک
blocking flow ..... شار انسدادی
boolean ..... بولی
bottleneck ..... گلوگاه
bound ..... کران
bounded queue ..... صف کران‌دار
bounded stack ..... پشته‌ی کران‌دار
branch and bound ..... انشعاب و حد
branching ..... انشعاب
breadth first search ..... جست‌وجوی سطح-اول
brick sort ..... مرتب‌سازی خشتی
bridge ..... پل
brute force ..... کورکورانه
bubble sort ..... مرتب‌سازی حبابی
bucket ..... سطل
bucket array ..... آرایه‌ی سطلی
bucket sort ..... مرتب‌سازی سطلی
bucketing method ..... روش سطل‌بندی
buffer ..... میان‌گیر
Byzantine Agreement ..... معاهده‌ی رومی
Byzantine generals ..... ژنرال‌های رومی

C

call ..... فراخوانی
canonical complexity class
رده‌ی پیچیدگی متعارف (کانونی)
capacitated facility location
مکان تسهیلات با ظرفیت محدود
capacity ..... ظرفیت
Catalan's number ..... عدد کاتالان
ceiling ..... سقف
cellular ..... سلولی
cellular automaton ..... خودکاره‌ی سلولی
centroid ..... مرکز ثقل
certificate ..... گواهی
chain ..... زنجیره
chaining ..... زنجیره‌ای
characteristic equation ..... معادله‌ی سرشت‌نما
child ..... فرزند

Chinese postman problem ... مسئله‌ی پستچی چینی
Chinese remainder theorem
قضیه‌ی باقی‌مانده‌ی چینی
chromatic number ... عدد رنگی
circuit ... مدار
circuit complexity ... پیچیدگی مدار
circuit value problem ... مسئله‌ی ارزش مدار
circular ... حلقه‌ای
circular list ... لیست حلقه‌ای
circular path ... مسیر حلقه‌ای
circular queue ... صف حلقه‌ای
class ... رده
clique ... خوشه
closed interval ... بازه‌ی بسته
clustering ... خوشه‌بندی
collective recursion ... بازگشت تجمعی
collision ... برخورد
combination ... ترکیب
combinatorics ... ترکیبیات
commutative ... جابه‌جایی
compaction ... فشرده‌سازی
comparison sort ... مرتب‌سازی مقایسه‌ای
competitive analysis ... تحلیل رقابتی
competitive ratio ... نسبت رقابتی
complement ... مکمل
complete binary tree ... درخت دودویی کامل
complete graph ... گراف کامل
complete tree ... درخت کامل
complexity ... پیچیدگی
complexity class ... رده‌ی پیچیدگی
complexity of algorithm ... پیچیدگی الگوریتم
compound data type ... داده‌گونه‌ی مرکب
computable ... رایانش‌پذیر
computational geometry ... هندسه‌ی محاسباتی
concatenation ... الحاق
concave ... مقعر
configuration ... پیکربندی
conflict graph ... گراف تقاطع
conjunction ... ترکیب عطفی
connected component ... جزء هم‌بند
connected component ... مؤلفه‌ی هم‌بند
connected graph ... گراف هم‌بند
connectivity ... هم‌بندی
coNP ... کو اِن‌پی
constraint ... قید
constructive proof ... اثبات سازنده
continuous knapsack ... کوله‌پشتی پیوسته
convolution ... کانولوژن

Cook's theorem ... قضیه‌ی کوک
cost ... هزینه
counter ... شمارنده
counting sort ... مرتب‌سازی شمارشی
critical path ... مسیر بحرانی
cursor ... اشاره‌گر اندیسی، مکان‌نما
cut ... بُرش
cut vertex ... رأس بُرشی
cutting plane ... صفحه‌ی بُرشی
cutting theorem ... قضیه‌ی بُرشی
cycle ... چرخه

D

data structure ... داده‌ساختار
data type ... داده‌گونه
database ... داده‌گان، داده‌پایگاه
decidable ... تصمیم‌پذیر
decision problem ... مسئله‌ی تصمیم‌گیری
decision tree ... درخت تصمیم
decomposable searching ... جست‌وجوی تجزیه‌پذیر
degree ... درجه
delete ... حذف
dense graph ... گراف متراکم
depth ... عمق
depth first search ... جست‌وجوی عمق‌ـاول
deque ... صف دوسر
derangement ... پریش
descendant ... نواده
descending order ... ترتیب نزولی
description ... توصیف
deterministic ... قطعی
deterministic algorithm ... الگوریتم قطعی
DFS forest ... جنگل جست‌وجوی عمق‌ـاول
diagonalization ... قطری کردن
diameter ... قطر
dictionary ... فرهنگ داده‌ای
difference ... تفاضل
digraph ... گراف جهت‌دار
Dijkstra ... دایکسترا
dining philosophers ... فیلسوف‌های گرسنه
direct-addressing ... آدرس‌دهی مستقیم
directed acyclic graph ... گراف جهت‌دار ناچرخه‌ای
directed graph ... گراف جهت‌دار
directed tree ... درخت جهت‌دار
discrete p-center ... pـمرکز گسسته
disjoint ... مجزا

disjoint find-merge ..... مجموعه‌های مجزا
disjoint set ..... مجموعه‌های مجزا
disjoint-set forest ..... جنگل مجموعه‌ی مجزا
disjunction ..... ترکیب فصلی
distribution sort ..... مرتب‌سازی توزیعی
divide and conquer ..... تقسیم‌وحل
domain ..... دامنه
domino ..... دومینو
double hashing ..... درهم‌سازی دوگانه
double rotation ..... دَوَران دوگانه
double-direction bubble sort
مرتب‌سازی حبابی دوسویه
doubly linked list ..... لیست پیوندی دوسویه
doubly-ended queue ..... صفِ دوسر
dual ..... دوگان
dynamic ..... پویا
dynamic array ..... آرایه‌ی پویا
dynamic hashing ..... درهم‌سازی پویا
dynamic programming ..... برنامه‌ریزی پویا

E

edge ..... یال
edge coloring ..... رنگ‌آمیزی یالی
edge connectivity ..... هم‌بندی یالی
edge crossing ..... تقاطع یالی
efficiency ..... کارایی
element uniqueness ..... یگانی اعضا
elementary data strucutre ..... داده‌ساختار ساده
encapsulation ..... بسته‌بندی
equivalent ..... هم‌ارز
equivalent binary tree ..... درخت دودویی معادل
Euclid ..... اقلیدس
Euclidean ..... اقلیدسی
Euclidean distance ..... فاصله‌ی اقلیدسی
Euclidean traveling salesman
فروشنده‌ی دوره‌گرد اقلیدسی
Euler cycle ..... چرخه‌ی اویلری
Euler tour ..... تور اویلری
Eulerian graph ..... گراف اویلری
Eulerian path ..... مسیر اویلری
exchange sort ..... مرتب‌سازی تعویضی
exclusive or ..... یای انحصاری
expansion ..... گسترش
exponential ..... نمایی
expression ..... عبارت
expression tree ..... درخت عبارت

external algorithm ..... الگوریتم خارجی
external memory ..... حافظه‌ی خارجی
external memory data structure
داده‌ساختار حافظه‌ی خارجی
external merge ..... ادغام خارجی
external merge sort ..... مرتب‌سازی ادغامی خارجی
external quicksort ..... مرتب‌سازی سریع خارجی
external sort ..... مرتب‌سازی خارجی
extreme point ..... نقطه‌ی کُنج

F

facility location ..... مکان تسهیلات
factorial ..... فاکتوریل
Fast Fourier Transform (FFT)
تبدیل فوریه‌ی سریع
feasible ..... امکان‌پذیر
feasible ..... امکان‌پذیر
feedback edge ..... یال بازخور (پس‌خورد)
Fibonacci number ..... عدد فیبوناچی
finite automaton ..... خودکاره‌ی متناهی
finite Fourier transform ..... تبدیل فوریه‌ی متناهی
finite state automaton ..... خودکاره‌ی حالت متناهی
finite state machine ..... ماشین حالت متناهی
first-in, first-out ..... اولین ورودی-اولین خروجی
Flavius Josephus ..... فلاویوس ژوزفوس
floor ..... کف
flow ..... شار
flow conservation ..... بقای شار
flow network ..... شبکه‌ی شار
Floyd-Warshall ..... فلوید-وارشال
Ford-Fulkerson ..... فورد-فالکرسن
forest ..... جنگل
formal ..... رسمی
formal language ..... زبان رسمی (صوری)
formal method ..... روش رسمی
Fortran ..... فرترن
Fourier transform ..... تبدیل فوریه
fractal ..... فرکتال
fractional knapsack ..... کوله‌پشتی تقسیم‌پذیر
free edge ..... یال آزاد
free tree ..... درخت آزاد
free vertex ..... رأس آزاد
full array ..... آرایه‌ی پُر
full binary tree ..... درخت دودویی پُر
fully persistent ..... تمام‌ماندگار
function ..... تابع

fuzzy ..... فازی، نادقیق
fuzzy edge ..... یال فازی

G

game tree ..... درخت بازی
garbage collection ..... زباله‌روبی
general list ..... لیست کلی
general tree ..... درخت کلی
geometric ..... هندسی
global optimum ..... بهینه‌ی سراسری
golden ratio ..... نسبت طلایی
graph ..... گراف
graph coloring ..... رنگ‌آمیزی گراف
graph drawing ..... رسم گراف
graph isomorphism ..... یک‌ریختی گراف‌ها
Gray code ..... کد گری
greatest common denominator
بزرگ‌ترین مخرج مشترک
greatest common divisor
بزرگ‌ترین مقسوم‌علیه مشترک (ب‌م‌م)
greedy ..... حریصانه
greedy algorithm ..... الگوریتم حریصانه
greedy heuristic ..... مکاشفه‌ی حریصانه
growth function ..... تابع رشد

H

halting problem ..... مسئله‌ی توقف
Hamiltonian cycle ..... چرخه‌ی همیلتنی
hash function ..... تابع درهم‌ساز
hash table ..... جدول درهم‌سازی
hashing ..... درهم‌سازی
head ..... سر
header, sentinel ..... سرآیند، سرلیست
heapsort ..... مرتب‌سازی هرمی
heuristic ..... مکاشفه‌ای
hidden Markov model ..... مدل مارکوف مخفی
hierarchical ..... پادگانی، سلسله‌مراتبی
hierarchical ..... سلسله‌مراتبی، پایگانی
homeomorphic ..... همسان‌ریخت
homogeneous recursive relation
رابطه‌ی بازگشتی همگن
homomorphic ..... هم‌ریخت
Horner's rule ..... قانون هورنر
Huffman tree ..... درخت هافمن
hybrid algorithm ..... الگوریتم دورگه (آمیخته)
hypercube ..... اَبَرمکعب
hyperedge ..... اَبَریال
hypergraph ..... اَبَرگراف

I

implication ..... استلزام
in-branching ..... انشعاب ورودی
in-degree ..... درجه‌ی ورودی
in-place ..... درجا
in-place merging ..... ادغام درجا
in-place sort ..... مرتب‌سازی درجا
inclusion exclusion ..... شمول و عدم شمول
inclusion-exclusion principle ... اصل ردّ و شمول
inclusion-exclusoin ..... رد و شمول
inclusive or ..... یای منطقی
incompressible ..... فشرده‌نشدنی
incompressible string ..... رشته‌ی فشرده‌نشدنی
incremental ..... افزایشی
independent set ..... مجموعه‌ی مستقل
induction ..... استقرا
induction basis ..... پایه‌ی استقرا
induction hypothesis ..... فرض استقرا
induction step ..... گام استقرا
infix ..... میان‌وندی
infix notation ..... نماد میان‌وندی
infix with complete paranthesis
میان‌وندی با پرانتزی کامل
inorder ..... بین‌ترتیب، میان‌ترتیب
insertion ..... درج
insertion sort ..... مرتب‌سازی درجی
interactive ..... محاوره‌ای
interface ..... میانا، واسط
interleave ..... درهم تنیدن
interior-based ..... مبتنی بر درون
internal node ..... گره داخلی
internal path ..... مسیر داخلی
internal sort ..... مرتب‌سازی داخلی
intersection ..... اشتراک
interval ..... بازه
intractable ..... رام‌نشدنی
introspection sort ..... مرتب‌سازی خودآزما
irreflexive ..... نابازتاب
isomorphic ..... یک‌ریخت
iterated logarithm ..... لگاریتم تکراری
iteration ..... تکرار

J

Java ..... جاوا

K

k-coloring ..... kـرنگ‌آمیزی
k-d tree ..... درخت کی‌دی
k-dimensional ..... kـبُعدی
k-way merge ..... ادغام kـراهه
k-way merge sort ..... مرتب‌سازی ادغامی kـراهه
kth shortest path ..... kاُمین کوتاه‌ترین مسیر
Karnaugh map ..... نقشه‌ی کارنو
key ..... کلید
Kruskal ..... کروسکال

L

labeled graph ..... گراف برچسب‌دار
language ..... زبان
last-in, first-out ..... آخرین ورودی‌ـاولین خروجی
latin square ..... مربع لاتین
lattice ..... شبکه
layered graph ..... گراف لایه‌شده
layout ..... جای‌گذاری
leaf ..... برگ
least common multiple ..... کوچک‌ترین مضرب مشترک
left-rotate ..... دَوَران چپ‌گرد
level-order ..... سطح‌ترتیب
lexicographic Ordering ..... ترتیب الفبایی
linear ..... خطی
linear insertion sort ..... مرتب‌سازی درجی خطی
linear order ..... ترتیب خطی
linear probing ..... وارسی خطی
linear probing sort ..... مرتب‌سازی وارسی خطی
linear product ..... ضرب خطی
link ..... پیوند
linked list ..... لیست پیوندی
list ..... لیست
little-o notation ..... نماد اُیِ کوچک
logarithmic ..... لگاریتمی
loop invariant ..... رابطه‌ی مستقل از حلقه
lower bound ..... کران پایین
lower triangular matrix ..... ماتریس پایین‌مثلثی

M

makespan ..... زمان اتمام آخرین کار
Manhattan distance ..... فاصله‌ی منهتنی
many-one reduction ..... کاهش چندبه‌یک
mapping ..... نگاشت
Markov chain ..... زنجیره‌ی مارکوف
matched edge ..... یال تطابق‌یافته
matched vertex ..... رأس تطابق‌یافته
matching ..... تطابق
matrix ..... ماتریس
max-heap ..... هرم بیشینه
maximal ..... بیشین
maximally connected component ..... مؤلفه‌ی بیشین‌هم‌بند
maximum ..... بیشینه
maximum bipartite matching ..... تطابق دوبخشی بیشینه
maximum independent set ..... مجموعه‌ی مستقل بیشینه
mean ..... میانگین
median ..... میانه
memoization ..... به‌خاطرسپاری
merge ..... ادغام
merge sort ..... مرتب‌سازی ادغامی
min-heap ..... هرم کمینه
minimal ..... کمین
minimization ..... کمینه‌سازی
minimum ..... کمینه
minimum bounding box ..... کوچک‌ترین مستطیل دربرگیرنده
minimum cut ..... برش کمینه
minimum spanning tree ..... درخت فراگیر کمینه
minimum vertex cut ..... برش رأسی کمینه
mode ..... مُد
model checking ..... بررسی مدل
model of computation ..... مدل رایانش
moderately exponential ..... نمایی ملایم
modular ..... هم‌نهشتی
monoticity ..... یک‌نوایی
monotonically decreasing ..... نزولی یکنواخت
monotonically increasing ..... صعودی یکنواخت
multi-commodity flow ..... شار چندکالایی
multi-set ..... مجموعه‌ی چندگانه
multigraph ..... گراف چندگانه
multiway decision ..... تصمیم چندراهه
multiway merge ..... ادغام چندراهه

N

nearest neighbor search
جست‌وجوی نزدیک‌ترین همسایه
nearly complete binary tree
درخت دودویی تقریباً کامل
negation ..... نقیض (نفی)
node ..... گره
non-comparison sort .... مرتب‌سازی غیر مقایسه‌ای
non-homogeneous recursive relation
رابطه‌ی بازگشتی ناهمگن
nondeterministic ..... غیرقطعی
nondeterministic algorithm ... الگوریتم غیرقطعی
nondeterministic polynomial time
زمان چندجمله‌ای غیرقطعی
NP-complete ..... ان‌پی-تمام
NP-hard ..... ان‌پی-سخت
nullary ..... صفرتایی

O

O notation ..... نماد اُیِ بزرگ
o notation ..... نماد اُیِ کوچک
object oriented ..... شئ‌گرا
objective function ..... تابع هدف
off-line ..... برون‌خط
omega ..... اُمگا
omicron ..... اُمیکرون
on-line ..... برخط
on-to ..... پوشا
one-dimensional ..... یک-بُعدی
one-to-one ..... یک‌به‌یک
open interval ..... بازه‌ی باز
open hashing ..... درهم‌سازی باز
operands ..... عمل‌وند
operation ..... عملیات
operations ..... اَعمال
operator ..... عمل‌گر
optimal ..... بهین
optimization ..... بهینه‌سازی
optimum ..... بهینه
or ..... یا
oracle ..... پیش‌گو
order ..... ترتیب، مرتبه
order statistics ..... مرتبه‌ی آماری
ordered linked list ..... لیست پیوندی مرتب
oriented graph ..... گراف جهت‌دار
orthogonal drawing ..... ترسیم متعامد
oscillating merge sort .. مرتب‌سازی ادغامی نوسانی
out-branching ..... انشعاب خروجی
out-degree ..... درجه‌ی خروجی
overflow ..... سرریز
overlap ..... هم‌پوشانی

P

parameter passing ..... انتقال پارامتر
parent-child ..... پدر-فرزندی
partial function ..... تابع جزیی
partial order ..... ترتیب جزیی
partial recursive function .... تابع بازگشتی جزیی
partially decidable ..... تصمیم‌پذیر پاره‌ای
partially dynamic graph ..... گراف جزیی پویا
partially ordered ..... جزیی‌مرتب
partially ordered tree ..... درخت نیمه‌مرتب
partition ..... افراز
partition ..... بخش
partitioning ..... بخش‌بندی
path ..... مسیر
path compression ..... فشرده‌سازی مسیر
perfect hashing ..... درهم‌سازی کامل
perfect matching ..... تطابق کامل
performance guarantee ..... ضمانت کارایی
performance ratio ..... نسبت کارایی
permutation ..... ترتیب
persistent ..... ماندگار
persistent data structure ..... داده‌ساختار ماندگار
pigeonhole ..... لانه‌کبوتری
pigeonhole principle ..... اصل لانه‌کبوتری
pivot ..... محور
planar ..... مسطح
planar graph ..... گراف مسطح
planar straight-line graph
گراف مسطح با خطوط مستقیم
planarity ..... مسطح بودن
planarization ..... مسطح‌سازی
pointer ..... اشاره‌گر
polyhedron ..... چندوجهی
polylogarithmic ..... چندلگاریتمی
polylogarithmically bounded
کران چندجمله‌ای لگاریتمی
polynomial ..... چندجمله‌ای
polynomial approximation scheme
شمای تقریب چندجمله‌ای

polynomial time ..... زمان چندجمله‌ای
polynomial-time algorithm
الگوریتم زمان چندجمله‌ای
polynomially bounded ..... کران چندجمله‌ای
polyphase merge ..... ادغام چندمرحله‌ای
polyphase merge sort .. مرتب‌سازی ادغامی چندفازه
pop ..... حذف از پشته
poset ..... مجموعه‌ی جزیی‌مرتب
postfix ..... پس‌وندی
postfix notation ..... نماد پس‌وندی
postorder ..... پس‌ترتیب
potential ..... پتانسیل
potential function ..... تابع پتانسیل
predecessor ..... قبلی
prefix ..... پیش‌وندی
prefix notation ..... نماد پیش‌وندی
preorder ..... پیش ترتیب
Prim ..... پریم
priority ..... اولویت
priority queue ..... صف اولویت
prisoner's dilemma ..... معمای اسیران
probabilistic ..... احتمالی
probabilistic algorithm ..... الگوریتم احتمالی
probe sequence ..... دنباله‌ی وارسی
probing ..... وارسی
procedure ..... رویه
process algebra ..... جبر پردازه‌ها
program counter ..... شمارنده‌ی برنامه
proper ..... سره
proper coloring ..... رنگ‌آمیزی مجاز
proper subset ..... زیرمجموعه‌ی سره
prune and search ..... هَرَس و جست‌وجو
pruning ..... هَرَس
pseudo-code ..... شبه‌کد
pseudo-random ..... شبه‌تصادفی
psuedo poluynomial ..... شبه چندجمله‌ای
punched card ..... کارت منگنه
push ..... درج در پشته
pushdown automaton ..... خودکاره‌ی پشته‌ای

Q

quadratic ..... درجه‌ی ۲
quadratic probing ..... وارسی درجه‌ی ۲
quantum computation ..... رایانش کوانتمی
query ..... پرسمان
queue ..... صف
quicksort ..... مرتب‌سازی سریع

R

radix sort ..... مرتب‌سازی مبنایی
radix tree ..... درخت مبنا
ragged matrix ..... ماتریس ناهموار
random access machine ماشین با دست‌رسی تصادفی
random number generator .... مولّد اعداد تصادفی
randomization ..... تصادفی‌سازی
randomized ..... تصادفی
randomized rounding ..... گرد کردن تصادفی
range ..... بُرد
range sort ..... مرتب‌سازی بازه‌ای
reachability ..... دست‌رسی
reachable ..... قابل دست‌یابی
recognizer ..... تشخیص‌دهنده (شناسنده)
record ..... رکورد
rectangular matrix ..... ماتریس مستطیلی
rectilinear ..... افقی-عمودی
rectilinear distance ..... فاصله‌ی افقی-عمودی
recurrence relation ..... رابطه‌ی بازگشتی
recursion ..... بازگشت
recursion termination ..... خاتمه‌ی بازگشت
recursion tree ..... درخت بازگشت
recursive ..... بازگشتی
recursively enumerable ..... بازگشتی‌شمارش‌پذیر
red-black tree ..... درخت قرمز-سیاه
reduction ..... کاهش، ساده‌سازی
reference parameter ..... پارامتر آدرسی
reflexive ..... بازتابی
register ..... ثبات
regular ..... منظم
relation ..... رابطه
relational structure ..... ساختار رابطه‌ای
relaxation ..... ترمیم
rescalable ..... بازمقیاس‌پذیر
response time ..... زمان پاسخ
return ..... بازگشت از یک فراخوانی
right-rotate ..... دَوَران راست‌گرد
root ..... ریشه
rotation ..... دَوَران
run time ..... زمان اجرا

S

saturated edge ..... یال اشباع‌شده
scheduling ..... زمان‌بندی
selection sort ..... مرتب‌سازی انتخابی
self-loop ..... طوقه
semi-open interval ..... بازه‌ی نیمه‌باز
separation ..... جداسازی
sequence ..... دنباله
set ..... مجموعه
set complement ..... مکمل مجموعه
set difference ..... تفاضل مجموعه
set union ..... اجتماع مجموعه
shared memory ..... حافظه‌ی اشتراکی
shell sort ..... مرتب‌سازی صدفی
shortest path ..... کوتاه‌ترین مسیر
signature ..... امضا (نشان)
simple data type ..... داده‌گونه‌ی ساده
simple path ..... مسیر ساده
simple statement ..... دستور ساده
simple uniform hashing
درهم‌سازی ساده و یک‌نواخت
simulation ..... شبیه‌سازی
single rotation ..... دَوَران تکی
single-destination shortest-path
کوتاه‌ترین مسیر به یک مقصد
single-pair shortest-path
کوتاه‌ترین مسیر بین یک زوج‌رأس
single-source shortest-path
کوتاه‌ترین مسیر از یک مبدأ
singly linked list ..... لیست پیوندی یک‌سویه
singular ..... تکین
sink ..... چاهک
skew symmetry ..... تقارن اریب
slope ..... شیب
solvable ..... حل‌شدنی
sorting algorithm ..... الگوریتم مرتب‌سازی
source ..... منبع، مبدأ
spanning tree ..... درخت فراگیر، درخت پوشا
sparse ..... تُنُک
sparse graph ..... گراف تُنُک
sparse matrix ..... ماتریس تُنُک
sparsification ..... تُنُک‌سازی
spell-checker ..... غلط‌یاب
splay tree ..... درخت اسپِلی
square matrix ..... ماتریس مربعی
stable ..... پایدار
stack ..... پُشته
star-shaped polygon ..... چندضلعی ستاره‌ای
start state ..... حالت شروع
state ..... حالت
state machine ..... ماشین حالت
state transition ..... گذار حالت
static ..... ایستا
Steiner tree ..... درخت اشتاینر
Steiner vertex ..... رأس اشتاینر
Stirling's approximation ..... تقریب استرلینگ
Stirling's formula ..... فرمول استرلینگ
stooge sort ..... مرتب‌سازی ساده‌لوحانه
straight insertion sort .. مرتب‌سازی درجی مستقیم
strictly decreasing ..... اکیداً نزولی
strictly increasing ..... اکیداً صعودی
strictly lower triangular matrix
ماتریس اکیداً پایین مثلثی
strictly upper triangular matrix
ماتریس اکیداً بالا مثلثی
string ..... رشته
strong induction ..... استقرای قوی
strongly connected ..... قویاً هم‌بند
strongly connected component .. جزء قویاً هم‌بند
strongly connected graph ..... گراف قویاً هم‌بند
structured ..... ساخت‌یافته
subgraph ..... زیرگراف
subgraph isomorphism ..... یکریختی زیرگراف‌ها
subset ..... زیرمجموعه
subtree ..... زیردرخت
succesor ..... بعدی
suffix automaton ..... خودکاره‌ی پس‌وندی
superset ..... زبَر مجموعه
supersink ..... اَبَرچاهک
supersource ..... اَبَرمنبع
symbol table ..... جدول نمادها
symmetric ..... متقارن
symmetric inclusion-exclusoin . رد و شمول متقارن
symmetric set difference ... تفاضل متقارن مجموعه
symmetrically linked list .... لیست پیوندی متقارن
symmetry ..... تقارن

T

tail ..... دُم
tail recursion ..... آخرین بازگشت
temporal logic ..... منطق زمانی
terminal ..... پایانه
text ..... متن

time/space complexity ..... پیچیدگی زمان/حافظه
topological order ..... ترتیب توپولوژیکی
topological sort ..... مرتب‌سازی توپولوژیکی
total ..... کامل، تام
total order ..... ترتیب تام
tour ..... تور
tournament ..... تورنمنت
towers of Hanoi ..... برج‌های هانُوی
tractable ..... مهارشدنی
transducer ..... مبدِل
transition ..... گذار (تغییر وضعیت)
transition function ..... تابع گذار
transitive ..... ترایا (تراگذری)
transitive reduction ..... ساده‌سازیِ ترایا
transpose ..... ترانهاده
traveling salesman ..... فروشنده‌ی دوره‌گرد
traversal ..... پیمایش
tree automaton ..... خودکاره‌ی درختی
tree sort ..... مرتب‌سازی درختی
triangle inequality ..... نامساوی مثلثی
triangulation ..... مثلث‌بندی
trie ..... تِرای
trinary ..... سه‌تایی
Turing machine ..... ماشین تورینگ
turn ..... گردش
two-dimensional ..... دو بُعدی
two-way linked list ..... لیست پیوندی دوسویه
two-way list ..... لیست دوسویه
two-way merge sort ..... مرتب‌سازی ادغامی دوراهه

U

unary ..... یگانی
unary function ..... تابع یگانی
unary operator ..... عمل‌گر یگانی
unbalanced ..... نامتوازن
uncomputable ..... رایانش‌ناپذیر
undecidable ..... تصمیم‌ناپذیر
undecidable problem ..... مسئله‌ی تصمیم‌ناپذیر
undirected graph ..... گراف غیرجهت‌دار
uniform ..... یک‌نواخت
uniform matrix ..... ماتریس یک‌نواخت
union ..... اجتماع
unique ..... یگانه
universal hashing ..... درهم‌سازی سراسری
universal state ..... حالت جامع
universe ..... جهان (مجموعه‌ی مرجع)
unsolvable ..... حل‌نشدنی
unsorted list ..... لیست نامرتب
unstable ..... ناپایدار
upper triangular matrix ..... ماتریس بالامثلثی

V

value parameter ..... پارامتر ارزشی
Venn diagram ..... نمودار وِن
verification ..... درستی‌سنجی
vertex ..... رأس
vertex coloring ..... رنگ‌آمیزی رأسی
vertex connectivity ..... هم‌بندی رأسی
vertex cover ..... پوشش رأسی
vertices ..... رأس‌ها
visibility map ..... نقشه‌ی دید
visible ..... قابل دید

W

waiting time ..... زمان انتظار
walk ..... گشت
weak induction ..... استقرای ضعیف
weighted graph ..... گراف وزن‌دار
worst-case ..... بدترین حالت
worst-case ..... بدترین حالت

X

xor ..... یای انحصاری

Z

zero-one principle ..... اصل صفر و یک

others

0-1 knapsack ..... کوله‌پشتی صفر و یک
2-3 tree ..... درخت ۳-۲
ρ-approximation algorithm ... الگوریتم ρ-تقریبی

کتاب‌نامه

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[3] Alfred V. Aho and Jeffry D. Ullman. *Foundations of Computer Science.* W. H. Freeman, 1992.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 2001.

[6] Lester R. Ford and Selmer M. Johnson. A tournament problem. *American Mathematical Monthly*, 66(5):391–395, 1959.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[8] Michael T. Goodrich, Roberto Tamassia, and David M. Mount. *Data Structures and Algorithms in C++.* Wiley, 2004.

[9] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, 1989.

[10] Tardos J. Kleinberg, E. *Algorithm Design.* Addison-Wesley, 2005.

[11] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley, 1968. Second edition, 1973.

[12] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, third edition, 1997.

[13] Udi Manber. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, 1989.

[14] Stephen B. Maurer and Anthony Ralston. *Discrete Algorithmic Mathematics.* A K Peters, Ltd., third edition, 2004.

[15] V. Vazirani. *Approximation Algorithms.* Springer, 2002.

[16] Mark Allen Weiss. *Algorithms, Data Structures, and Problem Solving with C++.* Addison-Wesley, 1996.

[17] J. W.J̃. Williams. Algorithm 232 - Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[۱۸] مرتضی محمدآبادی. الفبای المپیاد کامپیوتر. ویژه‌نامه‌ی شماره‌ی ۱۵، فصل‌نامه‌ی دانش‌پژوه، نشریه‌ی باشگاه دانش‌پژوهان جوان، پاییز ۱۳۷۹.

فهرست الفبایی

☐ درباره‌ی مؤلف

دکتر محمد قدسی در سال ۱۳۳۱ در شهر ملایر متولد شد. دیپلم خود را از دبیرستان علوی در تهران گرفت و لیسانس خود را در سال ۱۳۵۴ در رشته‌ی مهندسی برق از دانشگاه صنعتی شریف اخذ نمود. سپس برای ادامه‌ی تحصیل به دانشگاه کالیفرنیا، برکلی در آمریکا رفت و در سال ۱۳۵۶ فوق‌لیسانس خود را در رشته‌ی مهندسی برق و علم کامپیوتر گرفت. در همان سال به ایران بازگشت و عضو هیئت علمی دانشگاه صنعتی شریف و مربی دانشکده‌ی ریاضی و علوم کامپیوتر آن دانشگاه شد. در سال ۱۳۶۳ جهت ادامه تحصیل مجدداً به آمریکا رفت و در سال ۱۳۶۸ دکتری خود را در علم کامپیوتر از دانشگاه ایالتی پنسیلوانیا گرفت. از آن سال تاکنون عضو هیئت علمی دانشکده‌ی مهندسی کامپیوتر دانشگاه صنعتی شریف است و از سال ۱۳۸۴، استاد تمام این رشته است. علاوه بر سمت‌های علمی و اجرایی فراوان، او از سال ۱۳۷۱ رییس کمیته‌ی ملی المپیاد کامپیوتر در کشور است، و از سال ۱۳۷۸ مسابقه‌ی برنامه‌نویسی دانش‌جویی ای‌سی‌ام را در ایران آغاز کرد و سرپرست مسابقه‌ی منطقه‌ای ای‌سی‌ام در تهران است.

☐ درباره‌ی کتاب

این کتاب با نگاهی الگوریتمی مطالب مربوط به داده ساختارهای کامپیوتری را، هم در سطح پایه و هم پیشرفته، ارائه می‌کند. از این‌رو، از همان ابتدا به مبانی طراحی الگوریتم‌ها می‌پردازد و ترکیب مناسبی از داده‌ساختارها و الگوریتم‌هاست. این کتاب که بخشی از آن سال‌ها به‌عنوان جزوه‌ی درسی در دانشگاه صنعتی شریف تدریس شده است، می‌تواند به‌عنوان کتاب اصلی در اولین درسی که دانش‌جویان رشته‌های مهندسی و علوم کامپیوتر در این زمینه می‌گیرند، و در برنامه‌ی مصوب به‌نام «ساختمان داده و الگوریتم‌ها» یا «ساختمان داده‌ها» آمده است، استفاده شود. این کتاب حاوی ۱۲۸ شبه کد، ۱۶۵ شکل، بیش از ۳۳۰ تمرین و ۱۵ پروژه‌ی برنامه‌نویسی است و حاصل سال‌ها تجربه‌ی تدریس مؤلف است. استفاده از این کتاب علاوه بر دانش‌جویان، برای دانش‌آموزانی که خود را برای ورود به دوره‌های المپیادکامپیوتر آماده می‌کنند مفید خواهد بود.

☐ تصویر بال پروانه در طرح روی جلد، برگرفته از فتوبلاگ به آدرس: http://bsurprised.aminus3.com است.

ISBN 978964318549-7
شابک ۹۷۸-۹۶۴-۳۱۸-۵۴۹-۷

انتشارات فاطمی
www.fatemi.ir