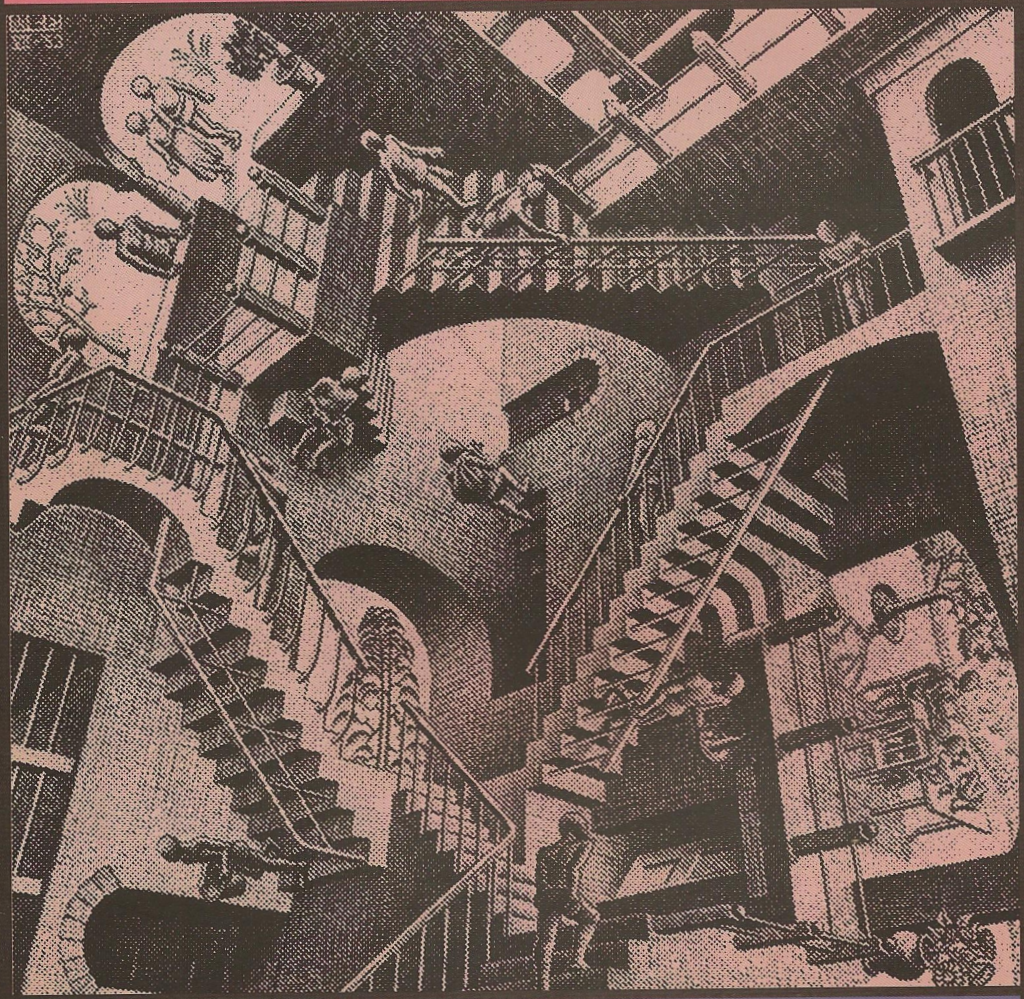
جلد نخست

طراحی الگوریتم

با رویکردی خلاقانه

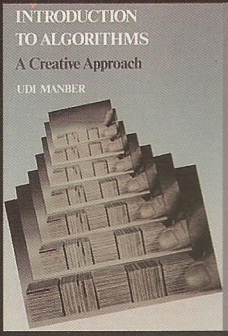


یودی منبر

ترجمه‌ی احمد صادقی صفت و سید علی حسینی

Introduction to Algorithms

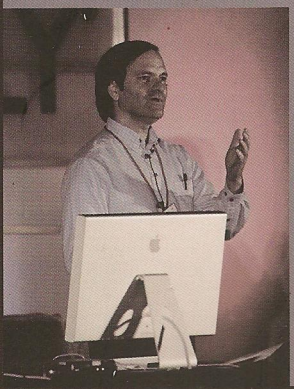
A Creative Approach



اگرچه این کتاب بیش‌تر مباحث درس طراحی الگوریتم از دوره ی کارشناسی رشته ی رایانه را پوشش می دهد ، اما مخاطبان اصلی آن کسانی هستند که می خواهند خلاقیت خویش را پرورش دهند. از این رو، خواندن آن به کسانی که قصد شرکت در المپیاد دانش‌آموزی یا دانش‌جویی رایانه و یا مسابقات برنامه نویسی ACM را دارند، سفارش می شود.

درباره‌ی نویسنده

- دارنده ی ph.D. در رشته علوم رایانه از دانشگاه Washington (1982)
- برنده ی جایزه ی Presidential Young Investigator (1985)
- نگارنده ی همین کتاب پر فروش که تا سال 1999 به چاپ شانزدهم رسید (1989)
- برنده ی سه جایزه برای بهترین مقاله و یک جایزه برای تدریس
- محقق ارشد Yahoo! (1998)
- برنده ی جایزه ی USENIX Software Tools User Group (1999)
- سرپرست بخش الگوریتم ها در Amazon (2002)
- مدیر اجرایی A9.COM در Amazon
- معاون ارشد Google در امور مهندسی (2006)
- بنیان‌گذار پروژه ی Knol در Google (2007)



Udi Manber

Translated by Ahmad Sadeghi Sefat، Sayed Ali Hosseini

طراحی الگوریتم

با رویکردی خلاقانه

یودی منبر

ترجمه‌ی

احمد صادقی صفت

سید علی حسینی

جلد نخست

Introduction to Algorithms: A Creative Approach
1989
Udi Manber
Addison-Wesley Publishing Company
Translated by Ahmad Sadeghi Sefat & Sayed Ali Hosseini
E-mail: Creative.Algorithms@Gmail.com

طراحی الگوریتم با رویکردی خلاقانه
نویسنده: یودی منبر
مترجمان: احمد صادقی صفت و سید علی حسینی
جلد نخست
طراح جلد: مهدی دوایی
نقاش تصویر روی جلد: M.C. Escher(۱۸۹۸-۱۹۷۲)
لیتوگرافی: صبا
چاپ: شریف
صحافی: البرز
نوبت چاپ: نخست
شمارگان: ۲۰۰۰ نسخه
شماره‌ی شابک: 978-964-04-2278-6
شماره‌ی شابک دوره: 978-964-04-2279-3

قیمت: ۵۰۰۰ تومان

سرشناسه :منبر، یودی
Manber, Udi
عنوان و نام پدیدآور : طراحی الگوریتم با رویکردی خلاقانه/ یودی منبر؛ ترجمه‌ی احمد صادقی صفت، علی حسینی.
مشخصات نشر : ملایر: احمد صادقی صفت، ۱۳۸۷.
مشخصات ظاهری : ۲ج.: مصور، جدول.
شابک :ج.۱: **978-964-04-2278-6**؛ دوره: **978-964-04-2279-3**
وضعیت فهرست نویسی : فیپا
یادداشت : عنوان اصلی: introduction to algorithms: a creative, c1989
موضوع : ساختار داده‌ها
موضوع : الگوریتمها
شناسه افزوده :صادقی صفت، احمد، ۱۳۵۵-
شناسه افزوده : حسینی، علی، ۱۳۵۷-
رده‌بندی کنگره :۱۳۸۷ ۷۵م۲س/QA۷۶/۹
رده‌بندی دیویی : ۰۰۵/۷۳
شماره کتابشناسی ملی : ۱۳۰۴۲۵۳

فهرست مطالب

پیش‌گفتار مترجمان ۱
پیش‌گفتار ۵
تقدیر و تشکر ۸
فصل ۱: آشنایی ۱۱
قراردادهای کتاب در توصیف الگوریتم‌ها ۱۵
تمرین‌ها ۱۵
فصل ۲: استقرای ریاضی ۲۱
۲–۱ آشنایی ۲۱
۲–۲ سه مثال ساده ۲۳
۲–۳ شمارش ناحیه‌های یک صفحه ۲۵
۲–۴ یک مسأله‌ی رنگ‌آمیزی ساده ۲۷
۲–۵ مسأله‌ای پیچیده‌تر درباره‌ی حاصل‌جمع ۲۸
۲–۶ یک نامساوی ساده ۲۹
۲–۷ رابطه‌ی اویلر ۳۰
۲–۸ مسأله‌ای از نظریه‌ی گراف ۳۲
۲–۹ کدهای Gray ۳۳
۲–۱۰ یافتن مسیرهایی با یال‌های «دو به دو جداازهم» در یک گراف ۳۷
۲–۱۱ رابطه‌ی بین میانگین‌های حسابی و هندسی ۳۹
۲–۱۲ مثالی از قانون ثابت حلقه در تبدیل عددی ده‌دهی به عددی دودویی ۴۱
۲–۱۳ لغزش‌های رایج در استقرا ۴۳
۲–۱۴ خلاصه ۴۵
مراجعی برای مطالعه‌ی بیش‌تر ۴۶
تمرین‌ها ۴۷
فصل ۳: تحلیل الگوریتم‌ها ۵۵
۳–۱ آشنایی ۵۵
۳–۲ نماد O ۵۷

نماد OO ۶۰
۳-۳ پیچیدگی فضایی (حافظه‌ی مورد نیاز) و پیچیدگی زمانی ۶۱
۳-۴ محاسبه‌ی حاصل‌جمع‌ها ۶۲
۳-۵ رابطه‌های بازگشتی ۶۶
۳-۵-۱ حدس‌های هوشمندانه ۶۶
۳-۵-۲ روابط «تقسیم‌وحل» ۷۰
۳-۵-۳ روابط بازگشتی با حافظه‌ی کامل ۷۲
۳-۶ چند رابطه‌ی سودمند ۷۴
۳-۷ خلاصه ۷۵
مراجعی برای مطالعه‌ی بیش‌تر ۷۶
تمرین‌های آموزشی ۷۶
تمرین‌های خلاقانه ۷۸
فصل ۴: نگاهی کوتاه به ساختمان‌های داده‌ای ۸۱
۴-۱ آشنایی ۸۱
۴-۲ ساختمان‌های داده‌ای پایه ۸۲
۴-۲-۱ عناصر ۸۲
۴-۲-۲ آرایه‌ها ۸۳
۴-۲-۳ رکوردها ۸۴
۴-۲-۴ لیست‌های پیوندی ۸۵
۴-۳ درخت‌ها ۸۶
۴-۳-۱ شیوه‌ی ذخیره‌ی درخت ۸۸
۴-۳-۲ درخت‌های هرمی ۸۹
۴-۳-۳ درخت‌های دودویی جست‌وجو ۹۲
درخت جست‌وجو ۹۳
عمل درج ۹۴
عمل حذف ۹۵
۴-۳-۴ درخت‌های AVL ۹۷
۴-۴ درهم‌سازی ۱۰۰

توابع درهم‌ساز ۱۰۲
چاره‌جویی برای حل مشکل برخورد ۱۰۲
۴-۵ مسأله‌ی union-find ۱۰۴
۴-۶ گراف‌ها ۱۰۷
۴-۷ خلاصه ۱۰۹
مراجعی برای مطالعه‌ی بیش‌تر ۱۱۰
تمرین‌های آموزشی ۱۱۱
تمرین‌های خلاقانه ۱۱۲
فصل ۵: طراحی استقرایی الگوریتم‌ها ۱۱۷
۵-۱ آشنایی ۱۱۷
۵-۲ محاسبه‌ی مقدار چندجمله‌ای‌ها ۱۱۸
۵-۳ بزرگ‌ترین زیرگراف القایی ۱۲۰
۵-۴ یافتن نگاشت‌های یک‌به‌یک ۱۲۳
۵-۵ مسأله‌ی ستاره‌ی مشهور ۱۲۵
۵-۶ نمونه‌ای از یک الگوریتم تقسیم‌وحل: مسأله‌ی نمای افقی ۱۲۹
۵-۷ محاسبه‌ی عامل‌های توازن در درخت‌های دودویی ۱۳۲
۵-۸ یافتن بزرگ‌ترین زیردنباله‌ی متوالی یا به‌هم‌پیوسته ۱۳۳
۵-۹ تقویت فرض استقرا ۱۳۵
۵-۱۰ نمونه‌ای از برنامه‌نویسی پویا: مسأله‌ی کوله‌پشتی ۱۳۶
۵-۱۱ اشتباهات رایج ۱۳۹
۵-۱۲ خلاصه ۱۴۱
مراجعی برای مطالعه‌ی بیش‌تر ۱۴۲
تمرین‌های آموزشی ۱۴۳
تمرین‌های خلاقانه ۱۴۴
فصل ۶: الگوریتم‌های دنباله‌ها و مجموعه‌ها ۱۴۹
۶-۱ آشنایی ۱۴۹
۶-۲ جست‌وجوی دودویی و گونه‌هایی از آن ۱۵۰
جست‌وجوی دودویی محض ۱۵۰

جست‌وجوی دودویی در یک دنباله‌ی چرخشی ۱۵۱
جست‌وجوی دودویی به دنبال یک اندیس ویژه ۱۵۲
جست‌وجوی دودویی در دنباله‌هایی با اندازه‌های نامشخص ۱۵۳
مسأله‌ی زیردنباله‌ی ناپایدار ۱۵۴
حل معادله‌ها ۱۵۵
۶-۳ جست‌وجو با درون‌یابی ۱۵۶
۶-۴ مرتب‌سازی ۱۵۸
۶-۴-۱ مرتب‌سازی سطلی و مرتب‌سازی بر اساس مرتبه ۱۵۸
۶-۴-۲ مرتب‌سازی درجی و مرتب‌سازی با انتخاب ۱۶۲
۶-۴-۳ مرتب‌سازی ادغامی ۱۶۳
۶-۴-۴ مرتب‌سازی سریع ۱۶۶
۶-۴-۵ مرتب‌سازی هرمی ۱۷۱
روش ساخت heap یا هرم ۱۷۲
۶-۴-۶ حد پایین مرتب‌سازی ۱۷۵
۶-۵ مرتبه‌ی آماری ۱۷۸
۶-۵-۱ بیشینه و کمینه‌ی عناصر ۱۷۸
۶-۵-۲ یافتن آماری kام یک دنباله ۱۷۹
۶-۶ فشرده‌سازی داده‌ها ۱۸۱
۶-۷ تطابق رشته‌ای ۱۸۵
۶-۸ مقایسه‌ی دنباله‌ها ۱۹۲
۶-۹ الگوریتم‌های مبتنی بر احتمال ۱۹۶
۶-۹-۱ اعداد تصادفی ۱۹۸
۶-۹-۲ یک مسأله‌ی رنگ‌آمیزی ۱۹۹
۶-۹-۳ روشی برای تبدیل الگوریتم‌های مبتنی بر احتمال (یا احتمال‌گرا)
به الگوریتم‌های قطعی ۲۰۰
۶-۱۰ یافتن اکثریت ۲۰۴
۶-۱۱ سه نمونه از روش‌های جالب اثبات ۲۰۷
۶-۱۱-۱ بلندترین زیردنباله‌ی صعودی (یا افزایشی) ۲۰۷

۶-۱۱-۲ یافتن بزرگ‌ترین دو عنصر یک مجموعه ۲۱۰
۶-۱۱-۳ پیدا کردن مد در یک مجموعه‌ی چندگانه ۲۱۲
۶-۱۲ خلاصه ۲۱۵
مراجعی برای مطالعه‌ی بیش‌تر ۲۱۵
تمرین‌های آموزشی ۲۱۸
تمرین‌های خلاقانه ۲۲۰
فصل ۷: الگوریتم‌های گراف ۲۳۱
۷-۱ آشنایی ۲۳۱
۷-۲ گراف‌های اویلری ۲۳۴
۷-۳ روش‌های پیمایش گراف ۲۳۶
۷-۳-۱ جست‌وجوی نخست-ژرفا ۲۳۷
گراف‌های بدون‌جهت ۲۳۷
ساخت درخت DFS ۲۴۰
گراف‌های جهت‌دار ۲۴۴
۷-۳-۲ جست‌وجوی نخست-پهنا ۲۴۶
۷-۴ ترتیب توپولوژیک ۲۴۸
۷-۵ کوتاه‌ترین مسیر از یک رأس به رأس‌های دیگر ۲۵۱
حالت بدون‌دور ۲۵۱
حالت کلی ۲۵۴
۷-۶ درخت پوشای کمینه ۲۵۹
۷-۷ کوتاه‌ترین مسیرها بین تمام زوج‌رأس‌های گراف ۲۶۴
۷-۸ بسط ترایا ۲۶۷
۷-۹ شیوه‌های تجزیه‌ی گراف ۲۶۹
۷-۹-۱ مؤلفه‌های دوهمبند ۲۷۰
۷-۹-۲ مؤلفه‌های قویاً همبند ۲۸۰
۷-۹-۳ نمونه‌هایی از کاربرد تجزیه‌ی گراف ۲۸۷
۷-۱۰ تطابق ۲۸۸
۷-۱۰-۱ یافتن تطبیق‌های کامل در گراف‌های بسیار چگال ۲۸۹

۷-۱۰-۲ تطابق دوبخشی ۲۹۰
بهبود الگوریتم ۲۹۳
۷-۱۱ شارهای شبکه ۲۹۴
۷-۱۲ دورهای هامیلتونی ۳۰۰
۷-۱۲-۱ استقرای معکوس ۳۰۰
۷-۱۲-۲ یافتن دورهای هامیلتونی در گراف‌های بسیار چگال ۳۰۱
۷-۱۳ خلاصه ۳۰۳
مراجعی برای مطالعه‌ی بیش‌تر ۳۰۴
تمرین‌های آموزشی ۳۰۶
تمرین‌های خلاقانه ۳۰۹
واژه‌نامه‌ی پارسی به انگلیسی ۳۲۹

پیش‌گفتار مترجمان

سپاس آفریدگار یکتا را که ما را یاری کرد تا سرانجام جلد نخست ترجمه‌ی خویش از کتاب ارزشمند و زیبای Introduction to Algorithms: A Creative Approach را عرضه کنیم؛ کتابی که بی‌گمان خواندنش وقت‌گیر و فهمیدنش لذت‌بخش است. نویسنده در پیش‌گفتار، کتاب خود را به خوبی معرفی کرده است؛ از این رو، تنها لازم است به خوانندگان گرامی گوش‌زد کنیم که نویسنده‌ی کتاب یکی از طراحان زبردست الگوریتم‌های رایانه‌ای است. کافی است نام او را در یکی از جست‌وجوگرهای اینترنتی وارد کنید تا با کارهای فراوان او در این زمینه آشنا شوید. وی پیش‌تر استاد علوم رایانه در دانشگاه Arizona بوده است، اما اینک یکی از معاونان ارشد Google در امور مهندسی است.

این کتاب یکی از مراجع اصلی درس طراحی الگوریتم بوده، در بسیاری از دانشگاه‌های جهان به عنوان نخستین مرجع این درس معرفی می‌شود. کتاب چنان تألیف شده است که علاوه بر دانش‌جویان شاخه‌های گوناگون رشته‌ی رایانه (در هر مقطعی!) برای شرکت‌کنندگان در المپیادهای دانش‌آموزی یا دانش‌جویی رایانه نیز بسیار سودمند است. برخی نیز این کتاب را بهترین مرجع آمادگی برای شرکت در مسابقات معتبر و جهانی برنامه‌نویسی ACM می‌دانند.

مترجمان کوشیده‌اند ترجمه‌ای یک‌دست، روان و خوانا عرضه کنند و از آنجایی که خود را ملزم به وفاداری به متن اصلی می‌دانسته‌اند، هرگاه ناگزیر از گفتن مطلبی بوده‌اند، به صراحت مشخص کرده‌اند که آن مطلب را مترجمان به کتاب افزوده‌اند. شیوه‌ای که مترجمان در ترجمه به آن توجه داشته‌اند، پرهیز از ترجمه‌ی واژه به واژه در عین امانت‌داری بوده است. به علاوه، تلاش کرده‌اند بیش‌تر از واژگان پارسی برای بیان مطلب یاری گیرند به گونه‌ای که برای نمونه، با وجود معادل رسایی مانند «دست‌کم» تا جایی که توانسته‌اند از به کار بردن واژه‌ی «حداقل» دوری کرده‌اند. به کار نبردن واژگانی مانند «فقط» و «می‌باشد» از دیگر ویژگی‌های این ترجمه است که به دانسته‌ها و سلیقه‌ی ویرایشی مترجمان برمی‌گردد.

همیشه ترجمه‌ی کتاب‌های فنی در معرض انتقاد شدید مترجمان و ویراستاران زبردست آثار ادبی بوده است. با آن که بخش عمده‌ی این اشکال‌ها بیراه نیست، اما بسیاری از افراد که از نزدیک با مفاهیم فنی سر و کار ندارند، با دشواری‌های ترجمه‌ی یک اثر فنی آشنا نیستند. برای نمونه، صاحب‌نظران زبان پارسی همواره گفته‌اند از به کار بردن فعل‌های مجهول باید تا جایی که می‌توانیم، بپرهیزیم؛ اما در این‌گونه کتاب‌ها بیش‌تر جمله‌ها به صورت مجهول بیان شده‌اند. هرچند مترجمان این کتاب کوشش کرده‌اند بیش‌تر، افعال معلوم را به کار برند، اما در پاره‌ای موارد خود را ناگزیر به بهره‌گیری از افعال مجهول دیده‌اند. در همین کتاب، بارها مترجمان برای پرهیز از ترکیب بدآهنگ «استقرا را» فعل‌های مجهول را به کار برده‌اند تا ناچار به استفاده از «را»ی نشانه‌ی مفعول نباشند.

نکته‌ی مهمی که در ترجمه‌ی متن‌های فنی با آن روبه‌رو هستیم، واژگان ویژه‌ی متن‌های فنی است که برای اهالی آن فن آشنا و بامعناست، در حالی که دیگران - و به ویژه اهل ادب و هنر - آن واژگان را در معنای رایج خود به کار می‌برند. این پدیده نه در ایران - که به نوعی، مصرف‌کننده‌ی محصولات فنی غرب است - بلکه در خاستگاه این دانش‌ها نیز امری عادی و پذیرفتنی است. بارها دیده‌ایم کسانی که در مغرب‌زمین در رشته‌ای جز رایانه دانش‌آموخته شده و به ایران برگشته‌اند، هنگام نشستن پشت رایانه کاربرد بسیاری از واژه‌های رایانه‌ای را در زبان انگلیسی، بسیار دور از معنای رایج آن‌ها یافته‌اند. برای نمونه، برخی صاحب‌نظران زبان پارسی کاربرد واژه‌ی «پیاده‌سازی» را که از قضا در رشته‌ی رایانه کاملاً رایج است، نادرست می‌دانند و مثلاً پیش‌نهاد می‌کنند که به جای آن واژه‌ی «اجرا» به کار گرفته شود، در حالی که اجرا در علوم رایانه مفهومی کاملاً متفاوت از پیاده‌سازی است.

چندین نکته‌ی دیگر را فهرست‌وار روشن می‌سازیم تا خوانندگان از روش ترجمه و نگارش متن دیدگاه دقیق‌تری داشته باشند:

- در نگارش واژه‌ها بر جدانویسی «بیش‌تر واژه‌ها» تأکید داشته‌ایم، هرچند شاید مورد پسند برخی نباشد. مترجمان باور دارند این شیوه خوانایی و یک‌دستی بیش‌تری به متن می‌بخشد.
- برخی محدودیت‌های ناخواسته ما را واداشت تا کتاب را در دو جلد عرضه کنیم. بخشی را که نویسنده برای ارائه‌ی «خطوط کلی راه‌حل گزیده‌ای از تمرین‌ها» آورده بود و نیز فهرست کامل مراجع را به پایان جلد دوم واگذار می‌کنیم، اما فهرست واژگان تخصصی به‌کاررفته در این جلد را در پایان همین جلد می‌آوریم.
- همان‌گونه که در پیش‌گفتار اصلی آمده، نویسنده این کتاب را با همراهی همسرش نوشته است، به همین دلیل در متن کتاب گاهی فعل جمع و گاهی فعل مفرد را برای نویسنده به کار برده است. مترجمان در همه‌ی این موارد به متن اصلی کتاب وفادار بوده‌اند، هرچند که شاید زیباتر می‌نمود همواره برای نگارنده‌ی کتاب، فعل مفرد را به کار ببرند.
- در کاربرد «را»ی نشانه‌ی مفعول، بیش‌تر از ساختار نگارشی به مفهومی که باید دانسته شود، توجه کرده‌ایم.
- chapter را فصل و section را بخش ترجمه کرده‌ایم؛ پس توجه کنید که هر فصل دربرگیرنده‌ی چندین بخش است.
- برخلاف بسیاری از کتاب‌ها که هنگام بیان اعداد ترتیبی با حروف الفبای انگلیسی، پسوند «اُم» را با خط تیره از خود حرف جدا می‌کنند و مثلاً می‌نویسند «n-ام»، ما این خط تیره را به کارنبرده‌ایم؛ پس ممکن است در متن ترجمه با «n/2امین» هم روبه‌رو شوید.

- نویسنده‌ی این کتاب گاهی واژه‌های maximal و maximum را به صورت معادل به کار برده است. مترجمان هرجا که maximal را در معنای واقعی خویش یافته‌اند، آن را «گسترش‌ناپذیر» ترجمه کرده‌اند نه «بیشینه».
- با این‌که در عبارتی مانند «همگی اعداد» باید یک حرف «ی» دیگر به کار برد تا نوشتار متناسب با گفتار گردد، یعنی باید نوشت «همگی‌ی اعداد»؛ به سبب ناآشنایی بسیاری از افراد با این روش نگارش، این شیوه را به کار نبرده‌ایم.
- اگر ترجمه‌ی کتاب پر از پرانتز است، به سبب آن است که متن اصلی کتاب نیز چنین است. به نظر می‌رسد در بسیاری از موارد، نویسنده خواسته است اصل و فرع مفاهیم را از هم جدا کند تا توجه خواننده از مفهوم اصلی منحرف نشود.
- هرچند واژگان method، way، approach، trick، technique و ... معناهای مشابهی دارند، اما دقیقاً یکسان نیستند. مترجمان بیش‌تر، واژگانی مانند شیوه، روش، رویکرد و ترفند را به جای آن‌ها به کار برده‌اند ولی هر واژه را تنها به یک معادل محدود نکرده‌اند.

برای آن‌که خوانندگان علاقه‌مند بتوانند از آخرین اصلاحات احتمالی در ترجمه آگاه شوند، یا مترجمان را در این کار یاری کنند، می‌توانند پیام‌های خود را به نشانی پست الکترونیکی «Creative.Algorithms@gmail.com» بفرستند. در پایان مترجمان بر خود لازم می‌دانند از «وحید صادقی صفت» که مدیریت بیش‌تر مراحل آماده‌سازی کتاب را بر عهده گرفت، بسیار سپاس‌گزاری کنند. از «مهری توکلی» و «فاطمه و رقیه مقدم» که کتاب را تایپ کردند و از «حمید علی‌محمدی» که نظارت بر چاپ را انجام داد و نیز از «مهدی دوایی» که طراحی جلد کتاب را بر عهده گرفت، قدردانی می‌کنیم.

پیش‌گفتار

این کتاب را نوشتم تا ناکامی‌هایم در بیان روشن الگوریتم‌ها را جبران کنم. همچون بسیاری از استادان دریافته بودم، برای برخی دانش‌جویان نه تنها حل مسائل ساده (ساده در نظر من) دشوار است، بلکه آنان از درک راه‌حل‌های عرضه‌شده نیز عاجزند. معتقدم ایجاد و شرح یک راه‌حل، به هم وابسته‌اند و نباید از یکدیگر جدا گردند؛ یعنی برای درک کامل یک راه‌حل، لازم است مراحل منتهی به آن را دنبال کنیم و تنها دقت در راه‌حل نهایی کافی نیست.

این کتاب بر نقش خلاقیت در طراحی الگوریتم تأکید می‌کند و هدف اصلی آن نشان دادن روش طراحی الگوریتم‌های نو به خواننده است. الگوریتم‌های این کتاب به صورت «مسأله‌ی X، الگوریتم A، الگوریتم 'A، برنامه‌ی P، برنامه‌ی 'P، ...» بیان نمی‌شوند، بلکه بیش‌تر به این صورتند: «مسأله‌ی X، الگوریتم سرراست آن، عیب‌هایش، دشواری‌های برطرف کردن این عیب‌ها، نخستین تلاش‌ها برای عرضه‌ی الگوریتمی بهتر (حتا تلاش‌های دارای اشتباه)، بهبودهایی در الگوریتم، تحلیل الگوریتم، رابطه‌ی الگوریتم با دیگر روش‌ها و دیگر الگوریتم‌ها، ... .» هدفمان این نیست که الگوریتم را به شیوه‌ای عرضه کنیم تا برنامه‌نویس بتواند به آسانی آن را به یک برنامه تبدیل کند، بلکه می‌خواهیم کار را چنان انجام دهیم که فهم اصول و مبانی الگوریتم‌ها ساده‌تر گردد. بنابراین الگوریتم‌ها را نه در قالب محصولات نهایی بلکه طی فرایند خلاقانه‌ی ایجاد آن‌ها توضیح می‌دهیم. هدفمان از آموزش الگوریتم‌ها تنها ارائه‌ی راه‌حل چند مسأله‌ی خاص نیست، بلکه می‌خواهیم خوانندگان بتوانند هنگام رویارویی با مسائل تازه آن‌ها را حل کنند. آموزش تفکر، حین طراحی یک الگوریتم به اندازه‌ی آموزش جزئیات آن الگوریتم، مهم است.

در این کتاب برای کمک به فرایند تفکر در طراحی الگوریتم‌ها، از یک شیوه‌ی «قدیم-جدید» استفاده می‌شود. این شیوه‌ی طراحی الگوریتم‌ها بسیاری از روش‌های آشنا را در بر می‌گیرد و چارچوبی دقیق و شهودی برای بیان ژرف‌تر آن‌ها عرضه می‌کند، ولی همه‌ی راه‌های طراحی الگوریتم را شامل نمی‌شود؛ پس ما نیز خود را به آن محدود نمی‌کنیم. اساس این شیوه، الگوبرداری از فرآیند فکریِ اثبات قضایای ریاضی به کمک استقرا، برای طراحی الگوریتم‌های ترکیبیاتی است. اگرچه بین استقرا و طراحی الگوریتم تفاوت وجود دارد، اما این دو، بیش از آنچه به نظر می‌رسد، به یکدیگر شبیه‌اند. افراد زیادی این دو را در کنار یکدیگر بررسی کرده‌اند، اما نه به ژرفای این کتاب. این شیوه (که در فصل ۱ اشاره‌ی مختصری به آن شده، اما صورت رسمی آن در فصل ۵ آمده است) بسیاری از روش‌های آشنای طراحی الگوریتم را در بر می‌گیرد و به فرایند ایجاد الگوریتم‌ها نیز یاری بسیاری می‌رساند.

مثلا در نظر بگیرید به شهری ناآشنا رسیده‌اید؛ خودرویی را کرایه کرده‌اید و در جست‌وجوی مسیرهایی به سمت هتلتان هستید. هنگام پرسیدن مسیرها از مردم، اگر درباره‌ی تاریخچه‌ی شهر،

مناظر عمومی، وسایل حمل‌ونقل و ... به شما پاسخ دهند؛ حوصله‌ی‌تان سرمی‌رود. شما منتظر پاسخ‌هایی در این قالب هستید: «دو مجتمع ساختمانی به جلو بروید، به سمت راست بپیچید، مستقیماً سه مایل به جلو بروید». اما اگر بخواهید مدتی طولانی در آن شهر زندگی کنید، دیدگاه‌تان عوض می‌شود. احتمالاً مدتی را به گشت و گذار می‌پردازید تا مسیرهای دیگری را نیز بیابید (البته اکر کسی را پیدا کنید که چنین مسیرهایی را به شما یاد دهد!) ولی سرانجام لازم می‌شود تا چیزهای بیش‌تری درباره‌ی آن شهر بدانید.

مسیرهایی که در این کتاب عرضه می‌شوند، چندان ساده نیستند. اگرچه این کتاب روش حل بسیاری از مسائل خاص را نشان می‌دهد، اما تکیه‌اش بر روی اصول و روش‌های کلی است. در نتیجه، کتابی است چالش‌برانگیز که شما را درگیر و وادار به اندیشیدن می‌کند. ایمان دارم که می‌ارزد در این راه بیش‌تر تلاش کنید.

طراحی الگوریتم‌های کارآمد در زمینه‌های گوناگونی مانند ریاضیات، آمار، مهندسی و زیست‌شناسی مولکولی بسیار مهم شده است. این کتاب به بررسی کلی الگوریتم‌ها می‌پردازد. افراد حرفه‌ایِ بسیاری از رشته‌ها و حتا دانشمندانی که عمیقاً با رایانه درگیر نیستند، برنامه‌نویسی را کاری غیرفکری و خسته‌کننده می‌دانند. گاهی چنین است، اما این باور سبب عرضه‌ی راه‌حل‌های دم‌دستی، پیش‌پاافتاده و ناکارآمد می‌شود، درحالی‌که راه‌حل‌های دقیق‌تر و کارآمدتری نیز وجود دارند. یکی از اهداف کتاب این است که خوانندگان خود را قانع کند تفکر الگوریتمی، دیدگاهی دقیق، ظریف و بااهمیت در برخورد با مسائل گوناگون به آنان می‌دهد.

کتاب خودکفاست و شیوه‌ی ارائه‌ی آن بیش‌تر شهودی است. نکات فنی هر بحث یا در کم‌ترین حد ممکن بیان شده، یا از بحث اصلی مجزا گشته است؛ به ویژه، جزئیات پیاده سازی تا جای ممکن از طراحی الگوریتم جدا شده است. برای تشریح اصولی که کتاب بر آن‌ها تأکید دارد، مثال‌های ویژه‌ی زیادی طراحی شده‌اند. مطالب کتاب به گونه‌ای نیست که بتوان بر آن‌ها مسلط شد یا آن‌ها را به خاطر سپرد. این مطالب به صورت مجموعه‌ای از ایده‌های اولیه، مثال‌ها، مثال‌های نقض، تغییر و بهبود در راه‌حل‌ها و ... ارائه شده‌اند. پس از توصیف اکثر الگوریتم‌ها، شبه‌کدهای متناظر با آن‌ها هم آورده شده است. هر فصل با بخشی برای مطالعه‌ی بیش‌تر به همراه مراجع وابسته به آن و تعدادی تمرین پایان می‌پذیرد. در بیش‌تر فصل‌ها، تمرین‌ها به دو دسته تقسیم می‌شوند: تمرین‌های آموزشی و تمرین‌های خلاقانه. تمرین‌های آموزشی برای ارزیابی استنباط خواننده از مثال‌ها و الگوریتم‌های آن فصل و تمرین‌های خلاقانه برای ارزیابی توانایی خواننده است. تمرین‌های خلاقانه به خواننده نشان می‌دهند که آیا با روش‌های آن فصل می‌تواند مسائل تازه را هم حل کند یا نه. خطوط کلی راه‌حل گزیده‌ای از تمرین‌ها (آن‌هایی که زیر شماره‌ی‌شان خط کشیده شده) در پایان کتاب آورده شده و خلاصه‌ای از مطالب هر فصل در پایان همان فصل عرضه گردیده است.

سازمان‌دهی کتاب چنین است: فصل‌های ۱ تا ۴ موضوعات مقدماتی را ارائه می‌کنند. فصل ۲ مقدمه‌ای بر استقرای ریاضی است. چنان که خواهیم دید، استقرای ریاضی در طراحی الگوریتم بسیار مهم است. به همین دلیل، تجربه‌ی اثبات‌های استقرایی در این راه مفیدند. بدبختانه شمار اندکی از دانش‌جویان علوم رایانه تجربه‌ی کافی در این کار دارند. ممکن است فصل ۲ برای برخی دانش‌جویان دشوار باشد. پیش‌نهاد می‌کنم که بار نخست از مثال‌های مشکل‌تر بگذرید و آن‌ها را بعداً بخوانید. فصل ۳ برای آشنایی با تحلیل الگوریتم‌هاست. این فصل، فرایند تحلیل الگوریتم‌ها را شرح می‌دهد و ابزارهای اساسی لازم را برای تحلیل الگوریتم‌های کتاب فراهم می‌آورد. فصل ۴ نگاهی مختصر به ساختمان‌های داده‌ای است. خوانندگانی که با ساختمان‌های داده‌ای پایه آشناترند و دانش ریاضی بیش‌تری دارند، می‌توانند خواندن کتاب را مستقیماً از این فصل شروع کنند (اگرچه خوب است دست‌کم بخش «آشنایی» تمام فصل‌ها را بخوانید.) فصل ۵ ایده‌ی اصلی طراحی الگوریتم‌ها را به کمک مقایسه‌ی آن‌ها با اثبات‌های استقرایی نشان می‌دهد. این فصل چندین مثال از الگوریتم‌های ساده ارائه می‌کند و شیوه‌ی ساخت آن‌ها را توضیح می‌دهد. اگر قصد خواندن تنها یک فصل از کتاب را دارید، همین فصل را بخوانید.

دو روش کلی برای سازمان‌دهی هر کتاب الگوریتم وجود دارد. یک روش بر پایه‌ی موضوع الگوریتم‌هاست؛ مثلاً فصلی برای الگوریتم‌های گراف یا فصلی برای الگوریتم‌های هندسی. راه دیگر بر پایه‌ی روش‌های طراحی است. درست است که این کتاب بر روش‌های طراحی تکیه می‌کند، اما من همان روش نخست را برگزیده‌ام و به نظرم این روش روشن‌تر و آسان‌تر است. فصل‌های ۶ تا ۹ به ترتیب الگوریتم‌هایی را در چهار مبحث زیر ارائه می‌کنند: الگوریتم‌های دنباله‌ها و مجموعه‌ها (مانند مرتب‌سازی، مقایسه‌ی دنباله‌ها، فشرده‌سازی داده‌ها)، الگوریتم‌های گراف (مانند درخت پوشای کمینه، کوتاه‌ترین مسیرها، تطبیق)، الگوریتم‌های هندسی (مانند پوسته‌ی کوژ، یافتن اشتراک دو چند ضلعی) و الگوریتم‌های عددی و جبری (مانند ضرب ماتریس‌ها، محاسبه‌ی سریع تبدیل فوریه).

فصل ۱۰ به reduction یا کاهش اختصاص دارد. اگرچه در فصل‌های پیش از آن نمونه‌هایی از کاهش آورده شده است، اما موضوع به قدری مهم و جالب‌توجه است که می‌ارزد یک فصل جداگانه را به آن اختصاص دهیم. این فصل مقدمه‌ای بر NP–تمام (موضوع فصل ۱۱) نیز هست. NP–تمام (از مباحث نظریه‌ی پیچیدگی) به بخشی جدانشدنی از نظریه‌ی الگوریتم‌ها تبدیل شده است. هرکس که دست به طراحی الگوریتم می‌زند، باید چیزهایی درباره‌ی روش‌های اثبات NP–تمام بودن مسأله‌ها بداند. فصل ۱۲ برای آشنایی با الگوریتم‌های موازی است و چندین الگوریتم جالب را از مدل‌های گوناگون پردازش موازی به خواننده‌ی خود نشان می‌دهد.

حجم مطالب کتاب بیش از یک نیم‌سال آموزشی است و دست استاد را در انتخاب، باز می‌گذارد. دوره‌ی مقدماتی طراحی الگوریتم باید فصل‌های ۳، ۵، ۶، ۷ و ۸ را شامل شود، ولی لازم نیست همه‌ی مطالب آن‌ها را در بر گیرد. در این دوره‌ی مقدماتی، قسمت‌های پیشرفته‌تر این فصل‌ها همراه با

فصل‌های ۹، ۱۰، ۱۱ و ۱۲ اختیاری تلقی می‌شوند و می‌توان آن‌ها را به یک دوره‌ی پیشرفته‌تر واگذار کرد.

تقدیر و تشکر

پیش و بیش از همه از همسرم راشل سپاس‌گزاری می‌کنم که کمک‌هایش به من در زمینه‌های مختلف، فهرست‌شدنی نیست. او در طراحی شیوه‌ای که کتاب بر آن بنا شده است، بسیار مؤثر بود. وی با پیش‌نهادها، اصلاحات و از همه مهم‌تر راه‌نمایی‌های به‌جایش مرا در نوشتن کتاب یاری داد.

به خاطر بررسی کامل و دقیق بخش زیادی از کتاب سپاس ویژه‌ام را به Jan van Leeuwen تقدیم می‌کنم. راه‌نمایی‌های ریزبینانه، پیش‌نهادهای پرشمار و اصلاحات فراوان او کتاب را بسیار بهبود بخشید. همچنین از Eric Bach، Darah Chavey، Kirk Pruhs و Sun Wu برای خواندن بخش‌هایی از نسخه‌ی دست‌نویس و ارائه‌ی راه‌نمایی‌های ارزنده‌ی‌شان و از آنان که کتاب را بررسی کرده‌اند، یعنی Guy T. Almes (دانشگاه Rice)، Agnes H. Chan (دانشگاه Northeastern)، Dan Gusfield (دانشگاه California، Davis)، David Harel (مؤسسه‌ی Weizmann، اسرائیل)، Daniel Herschberg (دانشگاه California، Irvine)، Jefferey H. Kingston (دانشگاه Iowa)، Victor Klee (دانشگاه Washington)، Charles Martel (دانشگاه California، Davis)، Michael J. Quinn (دانشگاه New Hampshire) و Diane M. Spresser (دانشگاه James Madison) بسیار سپاس‌گزارم.

از کارکنان Addison-Wesley نیز سپاس فراوان دارم، حتا آنان که نتوانستند چیزهای عجیب و غریبی را که من مشتاق گفتنشان بودم، تهیه کنند. آنان بسیار صبور، فهیم و یاری‌رسان بودند. از ناظر تولید کتابم Bette Aaronson، ویراستار آن Jim DeWolf و نمونه‌خوان کتاب Lyn Dupré تشکر فراوان می‌کنم که نه تنها مرا راه‌نمایی کردند، بلکه با وجود آن که گاهی روش‌های دیگری را بهتر می‌دانستند، گذاشتند تا کارها را مطابق سلیقه‌ام انجام دهم. همچنین از بنیاد ملی علوم (National Science Foundation) برای کمک‌های مالی‌اش در قالب پاداش رئیس‌جمهور برای محققین جوان (Presidential Young Investingator Award) و از شرکت‌های AT&T، Digital Equipment، Hewlett Packard و Tektronix به خاطر سرمایه‌گذاری‌هایشان قدردانی می‌کنم.

طراحی و ماشین‌نویسی کتاب را خودم انجام داده‌ام. قالب‌بندی آن در troff و چاپش با یک چاپگر Linotronic 300 در دانشکده‌ی علوم رایانه‌ی دانشگاه Arizona انجام شده است. از Ralph Griswold برای راه‌نمایی‌هایش، از John Luiten، Allen Peckham و Andrey Yeatts نیز برای یاری فنی‌شان در ماشین‌نویسی کتاب سپاس‌گزارم. تمامی شکل‌ها با نرم‌افزار gremlin – تهیه‌شده در دانشگاه California، Berkeley – ترسیم شده‌اند؛ به جز شکل ۱۲-۲۲ که Gregg Townsend آن

را کشیده است. نمایه‌ی کتاب به کمک نرم‌افزاری از Bentley و Kernighan [۱۹۸۸] آماده شده است. از Brian Kernighan ممنونم، چراکه کدهای برنامه را دقایقی پس از درخواست من، در اختیارم قرار داد. جلد کتاب را Marshall Henrichs براساس ایده‌ی خودم طراحی کرده است.

باید این را هم بگویم که نسخه‌ی نهایی کتاب را خودم آماده ساخته‌ام و بسیاری از راه‌نمایی‌ها و پیش‌نهادهای ارائه‌شده را نادیده گرفته‌ام. پس، تبعات این کار تنها برعهده‌ی من است.

Arizona ،Tuscan
Udi Manber
(نشانی الکترونیکی: udi@arizona.edu)

فصل ۱

آشنایی

«ترکیب اجزاء» فرایندی بسیار مهم است، چنان که برخی آن را شرط لازم‌وکافی برای پیش‌رفت علم می‌دانند. بی‌تردید، شرط لازم هست، اما شرط کافی نیست! برای آن‌که ترکیبی از اجزاء سودمند باشد و تلاش فکری‌مان را به هدر ندهد و برای آن‌که سکوی پرتابی برای رسیدن به چیزهای برتر شود؛ پیش از همه باید از وحدتی برخوردار باشد که ما آن را تنها چند «عنصرِ کنار یکدیگر» نبینیم.

Henri Poincaré، ۱۹۰۲

نهمین ویرایش از واژه نامه‌ی Webster (واژه‌نامه‌ی دانشگاهی جدیدش) الگوریتم را چنین تعریف می‌کند: «روالی برای حل یک مسأله‌ی ریاضی (مانند یافتن بزرگ‌ترین مقسوم علیه مشترک) در مراحلی متناهی که غالباً شامل تکرار یک عمل می‌شود» و یا در تعریفی کلی: «روالی گام‌به‌گام برای حل یک مسأله یا دست‌یابی به اهدافی چند». ما از تعریف کلی الگوریتم استفاده خواهیم کرد. طراحی الگوریتم، موضوعی قدیمی و ریشه‌دار است. مردم همواره می‌خواسته‌اند روشی بهتر برای دست‌یابی به اهدافشان داشته باشند؛ چه آنانی که می‌خواسته‌اند آتش روشن کنند، چه آنانی که اهرام را می‌ساخته‌اند و چه آنانی که نامه‌ها را مرتب می‌کرده‌اند. البته مطالعه روی الگوریتم‌های رایانه‌ای موضوعی نوین است. برخی الگوریتم‌های رایانه‌ای از روش‌هایی بهره می‌برند که پیش از اختراع رایانه هم شناخته شده بودند، اما حل بیش‌تر این مسأله‌ها با رایانه، به شیوه‌های نوینی نیاز دارد؛ برای مثال، این‌که به رایانه بگوییم «به آن سوی کوهستان بنگر و چنان‌چه دیدی ارتشی پیش می‌آید، فریاد خطر برآور!» کافی نیست. رایانه باید دقیقاً بداند معنای «نگریستن» چیست، چگونه «ارتش» را تشخیص دهد و چگونه فریاد خطر برآورد. (البته بنا به دلایلی همواره فریاد خطر برآوردن کاری ساده است!) رایانه تنها می‌تواند دستورالعمل‌هایی خوش تعریف، محدود به اعمال اولیه دریافت کند. ترجمه‌ی دستورالعمل‌های معمولی به زبانی که یک رایانه آن‌ها را بفهمد، دشوار است. برنامه‌نویسی، همین فرایند ضروری و دشواری است که امروزه میلیون‌ها نفر را در سطوح مختلف به خود مشغول کرده است.

برنامه‌نویسی رایانه، فراتر از ترجمه‌ی دستورالعمل‌ها به زبان قابل‌فهم برای رایانه است. اغلب لازم می‌شود روش‌های کاملاً تازه‌ای را برای حل مسأله ابداع کنیم. دشواری برنامه‌نویسی تنها به دلیل نیاز به زبانی عجیب و غریب برای حرف زدن با رایانه نیست؛ بلکه دانستن آنچه باید به رایانه بگوییم هم، مشکل است. رایانه‌ها نه تنها قادر به اجرای اعمالی هستند که پیش‌تر، انسان آن‌ها را انجام می‌داد؛ بلکه با سرعت شگفت‌آورشان می‌توانند بسیار بیش‌تر از گذشته، کار انجام دهند. در گذشته، الگوریتم‌ها با ده‌ها، یا شاید صدها و یا نهایتاً با هزاران دستورالعمل سر و کار داشتند، در حالی که رایانه‌ها می‌توانند با میلیاردها یا حتا تریلیون‌ها بیت از اطلاعات کار کنند. آن‌ها قادرند میلیون‌ها دستورالعمل اولیه را در یک ثانیه انجام دهند. طراحی الگوریتم‌هایی به این بزرگی موضوعی تازه است و چون ما عادت داریم به چیزهایی فکر کنیم که آن‌ها را می‌بینیم و احساس می‌کنیم؛ پس این کار از بسیاری جهات ناملموس است. در نتیجه، هنگام طراحی الگوریتم راحت‌تریم روش سرراست را به کار ببریم، چراکه این روش به خوبی از پس مسائل کوچک برمی‌آید. بدبختانه، الگوریتم‌هایی که برای مسائل کوچک به خوبی کار می‌کنند، ممکن است هنگام رویارویی با مسأله‌های بزرگ، بسیار ناتوان باشند. پیچیدگی و کارآمدی یک الگوریتم، در محاسبات بزرگ و حجیم، روشن می‌شود.

از سوی دیگر، الگوریتم‌هایی که ما در زندگی روزانه با آن‌ها سر و کار داریم، چندان پیچیده نیستند و خیلی هم تکرار نمی‌شوند. پس نمی‌ارزد برای ایجاد یک الگوریتم خوب تلاش زیادی کنیم؛ چراکه بازدهی‌اش اندک است. مثلاً مسأله‌ی خالی کردن کیسه‌های خواربار را در نظر بگیرید. قطعاً برای انجام این کار با توجه به محتویات کیسه‌ها و چیدمان وسیله‌های درون آشپزخانه، راه‌هایی با درجات کارآمدی گوناگون وجود دارد. افراد کمی هستند که حتا به این مسأله فکر کنند و از بین آن‌ها هم، کم‌تر افرادی هستند که الگوریتمی برای این کار بسازند. به عبارت دیگر، کسانی مجبورند روش‌های بهتری ایجاد کنند که در مقیاس وسیع تجاری، کار پر و خالی کردن کیسه‌ها را انجام می‌دهند. مثال دیگر زدن چمن‌هاست. ما می‌توانیم با کم‌تر کردن تعداد تغییر مسیر هنگام چمن‌زنی یا با کم‌تر کردن زمان چمن‌زنی و یا با کم‌تر کردن مسافت طی‌شده تا سطل زباله، کار را بهتر انجام دهیم. مگر آن‌که کسی واقعاً از چمن‌زنی متنفر باشد وگرنه کسی نیست که یک ساعت وقت بگذارد تا بفهمد چطور می‌شود از زمان چمن‌زنی یک دقیقه کم کرد! رایانه‌ها می‌توانند با کارهای پیچیده دست و پنجه نرم کنند و آن‌ها را بارها و بارها انجام دهند. اینجاست که می‌ارزد برای ایجاد روشی بهتر زمان زیادی را صرف کنیم، حتا اگر با این کار، الگوریتم‌های پیچیده‌تری به دست آید که فهمشان دشوارتر باشد. (البته در بهینه‌سازی نباید افراط کرد و به خاطر بهبود چند ثانیه‌ای در کار رایانه، ساعت‌ها وقت برنامه‌نویسان را به هدر داد.)

نیاز به روش‌های ناملموس در الگوریتم‌های حجیم و پیچیدگی احتمالی این الگوریتم‌ها، نشانگر دشوار بودن یادگیری آن‌هاست. نخست، باید درک کنیم که شیوه‌های شهودی و سرراست، بهترین شیوه‌های ممکن نیستند و لازم است به دنبال شیوه‌های بهتری هم بگردیم. مسلماً برای انجام این کار باید شیوه‌های نو و تازه‌ای را یاد بگیریم. کتاب، روش‌های فراوانی را برای طراحی الگوریتم‌ها بررسی و

تشریح می‌کند؛ اما یادگیری روش‌های گوناگون به تنهایی کارساز نیست و مانند این است که بخواهیم با حفظ و از بر کردن تعداد زیادی بازی شطرنج، بازیگر خوبی شویم. برای این کار باید اصول نهفته در پس شیوه‌های گوناگون را درک کرد و دانست که چگونه و مهم‌تر از آن چه وقت، باید این اصول را به کار گرفت.

می‌توان طراحی و پیاده‌سازی الگوریتم را با طراحی و ساخت خانه مقایسه کرد.[۱] ابتدا ساخت خانه را در نظر می‌گیریم: کار معمار تهیه‌ی نقشه‌ای است که نیازها را برآورده کند. مهندس عمران مطمئن می‌شود که نقشه، درست و عملی است. (تا پس از مدت کوتاهی فرو نریزد!) پس از آن بنّا بر پایه‌ی این نقشه، خانه را می‌سازد؛ البته هزینه‌ی تمام این مرحله‌ها باید بررسی و منظور شود. کارهای مراحل مختلف با هم فرق دارند، ولی به یکدیگر وابسته‌اند و در هم تنیده شده‌اند. طراحی الگوریتم نیز با ایده‌ها و شیوه‌های اساسی آغاز می‌شود. سپس طرح یا نقشه آماده می‌گردد. پس از آن باید درستی طرح را ثابت کنیم و مطمئن شویم که هزینه‌هایش معقول است. آخرین مرحله، پیاده‌سازی الگوریتم در یک رایانه‌ی مشخص است. با چشم‌پوشی از اشکالات ساده‌سازی زیاده از حد، می‌توانیم این فرایند را به چهار مرحله تقسیم کنیم: طراحی، اثبات درستی، تحلیل و پیاده‌سازی. تمام این مراحل با هم فرق دارند، اما به یکدیگر وابسته‌اند. هیچ یک از از آن‌ها را نمی‌توان بدون توجه به بقیه انجام داد. از آنجا که مشکلاتی در همه‌ی مراحل کار وجود دارد، بعید است کسی بتواند تمام این مراحل را یک‌باره و پشت‌سرهم انجام دهد. معمولاً تغییر طراحی لازم می‌شود و به دنبال این تغییر، باید اثبات امکان‌سنجی، تنظیم هزینه‌ها و پیاده‌سازی را دوباره انجام داد.

این کتاب روی مرحله‌ی اول، یعنی طراحی الگوریتم‌ها تمرکز می‌کند. نظر به مقایسه‌ای که پیش‌تر انجام شد، می‌توان نام کتاب را به «معماری الگوریتم‌ها» تغییر داد؛ اما معماری رایانه معنایی دیگر دارد و به کار بردن این واژه گیج‌کننده خواهد بود. به هرحال، کتاب از دیگر مراحل نیز کاملاً چشم‌پوشی نمی‌کند. پس از توصیف اکثر الگوریتم‌ها، بحثی هم درباره‌ی درستی، تحلیل و پیاده‌سازی آن‌ها - گاهی به طور خلاصه و گاهی به طور مشروح - می‌آید؛ اما تأکید و تکیه‌ی اصلی کتاب بر شیوه‌های طراحی است.

تنها یادگیری تعداد زیادی الگوریتم کافی نیست و نمی‌توان بر مبنای آن معمار خوبی برای طراحی الگوریتم‌های تازه شد. باید اصول کلی طراحی را درک کرد. ما در این کتاب برای تشریح الگوریتم‌ها روش متفاوتی را به کار می‌بریم. نخست می‌کوشیم تا خواننده، راه حل خودش را پیدا کند؛ چراکه قویاً باور داریم تلاش برای ساخت یک چیز، بهترین راه برای درک شیوه‌ی ساخت آن است. پس از آن و مهم‌تر از آن، از شیوه‌ای در طراحی الگوریتم‌ها پیروی می‌کنیم که به این فرایند خلاقانه کمک

۱- الهام‌بخش این مقایسه، دو کتاب خوب از Tracy Kidder به نام‌های The Soul of a New Machine (Little Brown، ۱۹۸۱) و House (Houghton Miffin، ۱۹۸۵) است.

می‌کند. این شیوه که در Manber [۱۹۸۸] معرفی شده است، برای توضیح ژرف‌تر طراحی الگوریتم‌ها، چارچوبی شهودی فراهم می‌کند که در عین سادگی، بسیار هوشمندانه است. این شیوه، راهی کلی برای رسیدن به طراحی نیز به دست می‌دهد. در برخورد با مسأله‌های جورواجور از روش‌های مختلفی بهره می‌بریم که همگی نمونه‌های گوناگونی از یک رویکرد اصلی هستند. به این ترتیب، گزینش از میان شمار زیاد شیوه‌های ممکن و به کار بستن آن‌ها، قاعده‌مندتر می‌گردد. این شیوه، همه‌ی راه‌های ممکن را برای طراحی الگوریتم، شامل نمی‌شود، اما می‌توان برای بیش‌تر الگوریتم‌های کتاب آن را به کار برد.

این رویکرد، بر مبنای استقرای ریاضی است و اساس آن الگوبرداری از فرایند فکری اثبات قضایای ریاضی به کمک استقرا برای طراحی الگوریتم‌های ترکیبیاتی است. ایده‌ی اولیه‌ی اصل استقرای ریاضی این است که برای اثبات یک گزاره، لازم نیست آن را با دست خالی ثابت کنیم، بلکه اگر نشان دهیم گزاره برای یک نمونه‌ی کوچک مبنا درست است؛ آنگاه کافی است ثابت کنیم درستی‌اش از درستی گزاره برای نمونه‌های کوچک‌تر نتیجه می‌شود. معنای این اصل در طراحی الگوریتم، حل مسائل بزرگ از روی مسائل کوچک است؛ یعنی اگر بدانیم چگونه می‌توان مسأله را برای ورودی‌های کوچک حل کرد، مسأله، به طور کامل حل شده است. ایده‌ی اصلی این روش، تمرکز بر گسترش راه‌حل به جای ساخت آن است. چنان‌که در فصل‌های آینده نشان خواهیم داد، راه‌های زیادی برای انجام این کار وجود دارد که طبعاً منجر به روش‌های فراوانی در طراحی الگوریتم می‌شود.

ما از استقرای ریاضی برای طراحی و تشریح الگوریتم‌های سطح بالا استفاده می‌کنیم و تا حدودی می‌کوشیم این شیوه را رسمیت ببخشیم و آن را به صورت یک اصل درآوریم. افراد زیادی کوشیده‌اند این کار را انجام دهند، مانند Dijkstra [۱۹۷۶]، Manna [۱۹۸۰]، Gries [۱۹۸۱]، Dershowitz [۱۹۸۳]، Paul [۱۹۸۸] و ... . کتاب، تلاش این افراد را تکمیل می‌کند. بیش‌تر به آموزش این شیوه توجه داشته‌ایم؛ اما اگر بتوان موضوعی را روشن‌تر از کتاب توضیح داد، قطعاً بهتر خواهد بود، چراکه فهمش آسان‌تر می‌شود. از جمله روش‌های اثبات که درباره‌ی آن‌ها بحث خواهیم کرد، عبارتند از تقویت فرض استقرا، انتخاب هوشمندانه‌ی دنباله‌ی استقرا، استقرای دوگانه و استقرای معکوس. شیوه‌ای که از آن استفاده می‌کنیم، از دو جهت اهمیت دارد: نخست، روش‌های ظاهراً مختلفی را که برای طراحی الگوریتم وجود دارد، زیر یک چتر گرد آورده‌ایم. دیگر این‌که، روش‌های آشنایی را که برای اثبات ریاضی طراحی الگوریتم‌ها وجود دارد، به کار بسته‌ایم. قسمت اخیر مهم‌تر است، چراکه راه را برای بهره‌گیری از روش‌های قدرتمندی که سالیان متمادی به گونه‌ای دیگر سامان یافته بودند، باز می‌کند.

ضعف اصلی روش ما، سراسری نبودن آن است؛ یعنی نمی‌توان همه‌ی الگوریتم‌ها را با استقرای فکری طراحی کرد و حتا اگر هم بتوانیم، نباید این کار را بکنیم. به هرحال، به‌کارگیری مبانی استقرایی چنان در طراحی الگوریتم‌ها رایج است که می‌ارزد روی استقرا متمرکز شویم. در این کتاب از سایر اصول ریاضی نیز چشم‌پوشی نمی‌شود. ایراد تقریباً تمام شیوه‌های جدید این است که هرچند روش جالبی برای توضیح پدیده‌های موجود دارند، اما هیچ گونه کمکی به ایجاد آن‌ها نمی‌کنند. این انتقاد کاملاً به‌جاست،

چون «آینده» نشان خواهد داد که یک شیوه‌ی مشخص چقدر کارآمد و پرکاربرد است. کاملاً مطمئنم که استقرا صرفاً ابزاری برای توضیح الگوریتم‌ها نیست، بلکه برای درک آن‌ها ضرورت دارد. حتا اگر شخصاً هیچ تجربه‌ای در ایجاد الگوریتم‌ها با این شیوه نداشتم، باز هم آن را مفید می‌یافتم؛ چراکه دست‌کم در دو مورد، منجر به ساخت الگوریتم‌های بسیار سریع‌تری شد (Manber و McVoy [۱۹۸۸]؛ Manber و Mayers [۱۹۸۹]).

قراردادهای کتاب در توصیف الگوریتم‌ها

در طی فرایند خلاقانه‌ی ایجاد الگوریتم‌ها، برای بسیاری از آن‌ها، علاوه بر توصیفشان از شبه‌کدها نیز استفاده کرده‌ایم. هدف از این شبه‌کدها، توصیف بهتر است؛ یعنی به بهینه‌سازی آن‌ها توجه چندانی نداشته‌ایم و توصیه هم نمی‌کنیم که از آن‌ها عیناً نسخه‌برداری کنید. در برخی موارد آگاهانه تصمیم گرفته‌ایم نسخه‌ی بهینه‌ی برنامه را به کار نبریم، چراکه اگر این کار را می‌کردیم، پیچیدگی اضافه‌ای در برنامه به وجود می‌آمد که ما را از ایده‌ی اصلی الگوریتم دور می‌کرد. گاهی روش تبدیل الگوریتم به برنامه را به طور دقیق توضیح نداده‌ایم. این تبدیلات، گاه بسیار روشن هستند، اگرچه گاهی هم چنین نیستند. تکیه و تأکید کتاب – چنان‌که پیش‌تر نیز گفته شد – بر اصول و مبانی طراحی الگوریتم است.

بیش‌تر، زبانی شبیه پاسکال (وگاهی خود پاسکال) را به کار برده‌ایم. در موارد زیادی از توصیف سطح بالا (مانند «افزودن به جدول» یا «بررسی تهی بودن مجموعه») درون کد پاسکال استفاده کرده‌ایم تا خوانایی بیش‌تر شود. آنچه برخلاف قوانین پاسکال انجام داده‌ایم، کاربرد begin و end برای محصور ساختن بخشی از کد است. این دستورها را تنها برای شروع و پایان برنامه به کار برده‌ایم و بخش‌های مختلف برنامه را با تورفتگی از یکدیگر جدا کرده‌ایم. این قرارداد بدون آن‌که سبب ابهام شود، باعث صرفه‌جویی در مصرف کاغذ است. معمولاً در مواردی که نیاز به اعلان نبوده است، اعلان دقیق متغیرها و انواع داده‌ای را انجام نداده‌ایم (برای مثال از G برای گراف و از T برای درخت سود جسته‌ایم).

تمرین‌ها

راه حل تمرین‌هایی که زیر شماره‌ی‌شان خط کشیده شده، در انتهای کتاب آورده شده است. تمرین‌هایی که با ستاره علامت‌گذاری شده‌اند، از نظر نویسنده دشوارترند.

برای حل تمرین‌های این فصل هیچ نیازی به دانش الگوریتم نیست. این تمرین‌ها، مسائل نسبتاً ساده‌ای را با ورودی‌های مشخص نشان داده‌اند. خواننده باید پاسخ‌ها را به روش دستی بیابد. هدف عمده‌ی تمرین‌ها این است که نشان دهد کار با تعداد حالات زیاد تا چه حد دشوار است. به عبارت دیگر،

می‌خواهند ناتوانی شیوه‌های دم‌دستی را نشان دهند. درباره‌ی مسأله‌هایی که در اینجا آورده شده‌اند، در فصل‌های آینده بحث خواهد شد.

۱-۱ اعداد ۱ تا ۱۰۰ را روی برگه‌های جداگانه بنویسید. سپس برگه‌ها را به هم بریزید و بکوشید آن‌ها را مجدداً مرتب کنید.

۱-۲ این ۱۰۰ عدد را روی برگه‌هایی مجزا بنویسید و آن‌ها را مرتب کنید. درباره‌ی تفاوت این تمرین با تمرین پیش بیندیشید.

۳۲۹۱۸	۲۱۱۹۲	۱۱۹۲۳	۴۲۳۳	۸۸۲۳۱	۸۳۱۲	۱۱	۷۲	۹۷۱	۸۲۳۴
۲۲۲۳۸	۴۹۲۸۳	۳۲۹۵	۲۹۳۴۷	۳۱۰۲	۳۲۸۸۳	۲۰۹۳۸	۲۹۳۰	۱۶	۸۲۳
۹۲۳۴	۹۲۳۶	۲۹۳۷۲	۲۲۱۸	۲۱۲۰۲	۸۳۷۲۱	۹۲۳۸	۸۲۲۱	۳۰۲۳۴	۹۳۹۲۰
۸۱۱۰۲	۱۰۱۱	۱۸۱۵۲	۲۸۳۱	۲۹۱۳۳	۹۲۲۹	۱۰۰۳۹	۹۲۳۵	۴۸۳۹۵	۲۸۳۲
۳۷۹۲۷	۷۳۴۹۲	۸۴۰۲	۴۸۲۰۱	۳۸۰۲۴	۲۸۰۰	۳۲۱۵۵	۲۷۷۳	۸۲۹۳۰	۲۲۲۱
۳۸۴۱	۳۱۱	۳۰۲۲	۳۸۰۹۹	۲۹۹۲۰	۲۸۳۴۹	۷۴۲۱۲	۷۰۱۱	۱۸۲۳	۹۰۳
۲۹۲۱	۹۳۳۵	۲۹۱۲۳	۲۸۹۱۰	۲۹۲۸۱	۳۷۷۲	۲۰۰۱۲	۷۰۴۵۸	۳۰۵۷۲	۳۸۰۱۳
۷۲۰۳۲	۲۸۰۰۱	۸۳۸۳۵	۳۰۱۷	۹۲۶۲۶	۸۳۸۲۵	۲۹۲۶۳	۲۰۱۷	۲۶۲	۸۳۶۲
۷۷۳۰۲	۸۵۹۳	۳۸۲۶	۹۳۷۴	۲۰۰۱	۸۳۲۶۱	۴۸۴۰۲	۴۸۴۵	۷۹۷۹۴	۲۷۲۷۱
۳۹۹۹۲	۲۲۹۳۶	۴۴۴	۲۹۳۷	۳۷۲۰۱	۳۷۳۲۲	۴۹۴۷۲	۱۱۳۲۹	۲۲۵۳	

۱-۳ فهرست اعداد زیر را در نظر بگیرید. کار شما این است که تا جای ممکن تعداد کم‌تری از آن‌ها را پاک کنید طوری که اعداد باقی‌مانده به ترتیب صعودی باشند. برای مثال پاک کردن همه‌ی آن‌ها به جز دو تای اول، دنباله‌ای صعودی ایجاد می‌کند؛ پاک کردن همه به جز اولین، سومین، ششمین و هشتمین نیز همین‌کار را انجام می‌دهد (والبته تعداد کم‌تری عدد پاک خواهد شد).

۹	۴۴	۳۲	۱۲	۷	۴۲	۳۴	۹۲	۳۵	۳۷	۴۱	۸	۲۰	۲۷
۸۳	۶۴	۶۱	۲۸	۳۹	۹۳	۲۹	۱۷	۱۳	۱۴	۵۵	۲۱	۶۶	۷۲
۲۳	۷۳	۹۹	۱	۲	۸۸	۷۷	۳	۶۵	۸۳	۸۴	۶۲	۵	۱۱
۷۴	۶۸	۷۶	۷۸	۶۷	۷۵	۶۹	۷۰	۲۲	۷۱	۲۴	۲۵	۲۶	

۱-۴ تمرین قبل را چنان حل کنید که اعداد باقی‌مانده به صورت نزولی باشند.

۱-۵ گیریم که در کشوری عجیب، پنج نوع سکه وجود داشته باشد: ۱۵، ۲۳، ۲۹، ۴۱ و ۶۷ سنتی. ترکیبی از این سکه‌ها بیابید که با آن بشود مبلغ ۱۸ دلار و ۸ سنت (۱۸۰۸ سنت) را پرداخت کرد. (از همه‌ی سکه‌ها به تعداد کافی در جیبتان وجود دارد.)

۱-۶ ورودی این مسأله زوج‌هایی از اعداد صحیح است به گونه‌ای که زوج (x,y) یعنی x منتظر پاسخی از y است. وقتی x منتظر پاسخ است، هیچ کار دیگری نمی‌تواند انجام دهد؛ مثلاً

نمی‌تواند به عدد دیگری پاسخ دهد. مسأله، یافتن دنباله‌ای از زوج‌ها به صورت $(x_1\ x_2)$، $(x_2\ x_3)$، ...، $(x_{k-1}\ x_k)$ و $(x_k\ x_1)$ برای هر $k>1$ است. اگر چنین دنباله‌ای موجود باشد، یک بن‌بست رخ داده است؛ یعنی هیچ کدام از اعداد، پیش نمی‌روند چون منتظر پاسخ عدد دیگری هستند.

می‌توانید از قلم و کاغذ استفاده کنید و هر نوع محاسبه‌ای که می‌خواهید (مانند مقایسه کردن و ایجاد جدول) با اعداد انجام دهید، اما نباید شکل بکشید (البته می‌توانید برای ورودی‌های دیگری شکل رسم کنید و به کمک آن روشی کلی برای حل چنین مسأله‌ای بیابید).

(۱۶ ۱)، (۲۱ ۲)، (۲۵ ۲)، (۲۲ ۲)، (۵۰ ۲۳)، (۴۷ ۲۳)، (۱ ۲۴)، (۱۰ ۲۵)، (۷ ۳۵)، (۴۵ ۳۶)، (۳۷ ۳۶)، (۴۲ ۳۸)، (۴۱ ۳۹)، (۳۷ ۱۲)، (۲۳ ۱۲)، (۳ ۱۲)، (۲۰ ۱۲)، (۲۵ ۱۴)، (۹ ۴۱)، (۳ ۴۲)، (۵ ۴۳)، (۲۲ ۴۳)، (۲ ۲۹)، (۴۸ ۳۰)، (۱۵ ۳۱)، (۱۷ ۳۲)، (۴۵ ۶)، (۱ ۶)، (۳۵ ۵)، (۲۰ ۵)، (۲۸ ۵)، (۱۱ ۵)، (۴ ۴۸)، (۱۰ ۴۸)، (۳۲ ۴۹)، (۳۱ ۷)، (۴ ۷)، (۳۳ ۵)، (۲۹ ۶)، (۱۲ ۶)، (۱۱ ۶)، (۳ ۶)، (۱۷ ۶)، (۲۷ ۴۵)، (۳۴ ۴۷)، (۲۰ ۴۸)، (۴۰ ۷)، (۳۴ ۷)، (۱۱ ۸)، (۱۹ ۹)، (۳۰ ۱۱)، (۴ ۱۱)، (۲۲ ۱۱)، (۲۵ ۱۱)، (۲۴ ۲۰)، (۲۳ ۲۱)، (۴۶ ۲۱)، (۴۷ ۲۲)، (۴۹ ۲۳)، (۳۹ ۳)، (۳۴ ۳)، (۱۴ ۴)، (۳۷ ۴)، (۴۲ ۵)، (۸ ۵)، (۲ ۱۵)، (۵۰ ۱۵)، (۴ ۱۵)، (۳۷ ۱۵)، (۱۶ ۱۳)، (۳۸ ۱۷)، (۲۸ ۱۸)، (۸ ۱۹)، (۱۵ ۲۶)، (۴۲ ۲۶)، (۱۸ ۲۷)، (۳۵ ۲۸)، (۳۶ ۱۳)، (۵۰ ۱۳)، (۳۴ ۱۳)، (۲۲ ۱۳)، (۳۴ ۲۹)، (۳۸ ۲۹)، (۳۰ ۲۹)، (۱۶ ۲۹)، (۳۳ ۴۴)، (۳۶ ۴۴)، (۷ ۴۴)، (۴ ۴۴)، (۳۲ ۴۴)، (۲۱ ۴۴)، (۹ ۳۳)، (۲۱ ۳۳)، (۳۵ ۳۳)، (۱۹ ۳۳)، (۴۱ ۳۳)، (۱۰ ۲۶)، (۴۴ ۲۶)، (۱۶ ۲۶)، (۳۹ ۲۶)، (۱۷ ۲۶)

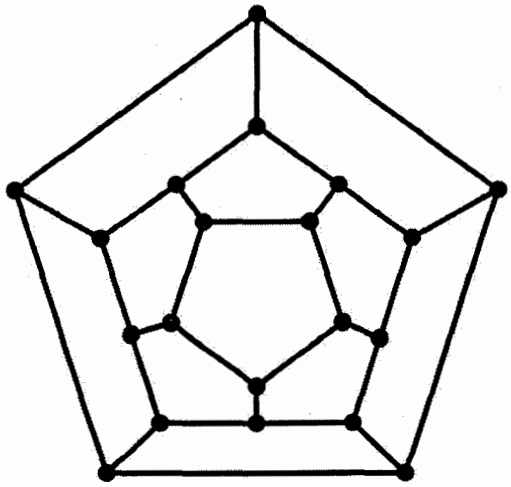
۱-۷ ورودی مسأله، جدول ۱۵×۱۵ شکل ۱-۱ است. به ازای هر i، شماره‌ی سطر i و شماره‌ی ستون i، بیانگر مکان یکسانی هستند. هر خانه‌ی جدول نشان‌دهنده‌ی مسافت بین یک سطر با یک ستون است؛ علامت «-» نیز مشخص می‌کند که بین دو مکان هیچ مسیر مستقیمی وجود ندارد. مسافت مستقیم دو مکان ممکن است کوتاه‌ترین مسافت نباشد و بتوان با عبور از یک (یا چند) مکان دیگر، بین دو مکان اصلی مسیری کوتاه‌تر یافت؛ برای مثال، کوتاه‌ترین مسیر از ۱ به ۶، از راه ۵ و ۱۲ است. کوتاه‌ترین مسیر از ۱ به ۱۵، از ۴ به ۳ و از ۱۵ به ۸ را بیابید.

	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱	۱۲	۱۳	۱۴	۱۵
۱	۰	۲	۳	–	۱	۹	۶	۲	۱	۷	۴	۲	۸	۳	–
۲	۷	۰	۲	–	–	–	–	۲	۱	۶	۹	۱	۷	۲	۸
۳	۸	–	۰	۸	۹	۳	۶	۸	۵	۷	–	۸	–	۳	–
۲	–	۸	–	۰	–	۵	۴	–	–	۱	۱	۹	–	۸	–
۵	۹	–	۸	–	۰	۳	۲	۷	۵	۸	–	۱	–	۴	۲
۶	۳	۲	–	۳	۶	۰	۵	۳	۲	–	۸	۷	۲	–	۸
۷	۲	–	–	۲	۸	–	۰	۶	۲	–	۸	۸	۲	–	۴
۸	۱	۱	–	–	۲	۳	۸	۰	–	۱	۱	–	۲	۷	–
۹	۴	–	۹	–	۲	۹	–	۲	۰	۴	۹	۳	–	–	–
۱۰	–	–	–	–	۱	۸	–	۷	۱	۰	۳	–	–	–	۲
۱۱	۳	۸	۷	۱	–	–	۳	۸	–	–	۰	۲	۹	۲	۱
۱۲	۳	–	۱	۲	۸	۱	۱	–	۵	۱	۹	۰	۲	–	۹
۱۳	۷	–	۳	۱	۶	–	–	۲	–	۳	–	۹	۰	۲	–
۱۴	۲	۹	۶	–	۷	–	۹	–	۳	–	۱	۱	۹	۰	–
۱۵	۲	۹	۲	۱	–	–	۱	–	۴	۳	۶	۵	۱	–	۰

شکل ۱-۱ جدول تمرین‌های ۱-۷ و ۱-۸

۱-۸ جدول شکل ۱-۱ را در نظر بگیرید و کوتاه‌ترین مسیرهای ممکن از ۵ به دیگر مکان‌ها را بیابید.

۱-۹ گراف شکل ۱-۲ را در نظر بگیرید. مسیری از یال‌های گراف بیابید که از هر رأس دقیقاً یک بار بگذرد. (این گراف متناظر با یال‌های یک دوازده‌وجهی منتظم است. نخستین بار ریاضی‌دان ایرلندی، Sir William R. Hamilton، این معما را بیان کرد. درباره‌ی این معما در بخش ۷-۱۲ بیش‌تر بحث خواهد شد.) (هر وجه دوازده‌وجهی منتظم، یک پنج‌ضلعی منتظم است - مترجمان)



شکل ۱-۲ معمای هامیلتون

۱-۱۰ در این مسأله با یک «هزارتو» سر و کار داریم، اما آن را به جای تصویر با عدد نمایش داده‌ایم. این هزارتو در مربعی شامل ۱۱ سطر و ۱۱ ستون قرار گرفته است که هر یک از سطرها یا ستون‌ها با اعداد ۰ تا ۱۰ مشخص شده‌اند. هزارتو از روی خط‌های سطرها و ستون‌ها عبور می‌کند (در یکی از چهار جهت بالا، پایین، چپ و راست). نقطه‌ی آغاز (۰,۰) و (۱۰,۱۰) نقطه‌ی پایان است. این فهرست، نقاطی را نشان می‌دهد که نمی‌توانید از آن‌ها بگذرید:

(۳,۲) (۶,۶) (۷,۰) (۲,۸) (۵,۹) (۸,۴) (۲,۴) (۰,۸) (۱,۳) (۶,۳) (۹,۳) (۱,۹) (۳,۰)
(۳,۷) (۴,۲) (۷,۸) (۲,۲) (۴,۵) (۵,۶) (۱۰,۵) (۶,۲) (۶,۱۰) (۴,۰) (۷,۵) (۷,۹) (۸,۱)
(۵,۷) (۴,۴) (۸,۷) (۹,۲) (۱۰,۹) (۲,۶)

الف- مسیری از نقطه‌ی آغاز به نقطه‌ی پایان بیابید که شامل هیچ یک از این نقاط مسدود نباشد.

ب- کوتاه‌ترین مسیر ممکن از نقطه‌ی آغاز به نقطه‌ی پایان را چنان بیابید که هیچ یک از نقاط مسدود را در بر نگیرد.

۱-۱۱ بزرگ‌ترین مقسوم‌علیه مشترک دو عدد ۲۲۵۲۷۷ و ۱۷۸۷۹۴ را بیابید. (بزرگ‌ترین مقسوم‌علیه مشترک دو عدد صحیح، بزرگ‌ترین عددی است که هر دو عدد را بشمارد؛ یا به عبارت دیگر، هر دو عدد بر آن بخش‌پذیر باشند.)

۱-۱۲ مقدار $۲^{۶۴}$ را حساب کنید. بکوشید این کار را با کم‌ترین تعداد عمل ضرب انجام دهید.

۱-۱۳ شکل ۱-۳، تعداد آرای انتخاباتی برای هر ایالت آمریکا در انتخابات ریاست جمهوری سال ۱۹۸۸ را نشان می‌دهد. (نامزدی که اکثریت آرای یک ایالت را به دست آورد، تمامی آرای انتخاباتی آن ایالت را به خود اختصاص می‌دهد.) در کل ۵۳۸ رأی انتخاباتی وجود دارد. مشخص کنید آیا از نظر ریاضی امکان دارد انتخابات به نتیجه‌ی مساوی ختم شود. (این مسأله با نام «مسأله‌ی بخش‌بندی» شناخته می‌شود و حالت ویژه‌ای از مسأله‌ی کوله‌پشتی است که در بخش ۵-۱۰ درباره‌اش بحث خواهد شد.)

Alabama	۹	Alaska	۳	Arizona	۷
Arkansas	۶	California	۴۷	Colorado	۸
Connecticut	۸	Delaware	۳	Florida	۲۱
Georgia	۱۲	Hawaii	۴	Idaho	۴
Illinois	۲۴	Indiana	۱۲	Iowa	۸
Kansas	۷	Kentucky	۹	Louisiana	۱۰
Maine	۴	Maryland	۱۰	Massachusetts	۱۳
Michigan	۲۰	Minnesota	۱۰	Mississippi	۷
Missouri	۱۱	Montana	۴	Nebraska	۵
Nevada	۴	New Hampshire	۴	New Jersey	۱۶
New Mexico	۵	New York	۳۶	North Carolina	۱۳
North Dakota	۳	Ohio	۲۳	Oklahoma	۸
Oregon	۷	Pennsylvania	۲۵	Rhode Island	۴
South Carolina	۸	South Dakota	۳	Tennessee	۱۱
Texas	۲۹	Utah	۵	Vermont	۳
Virginia	۱۲	Washington	۱۰	Washington, D.C.	۳
West Virginia	۶	Wisconsin	۱۱	Wyoming	۳

شکل ۱-۳ فهرست تعداد آرای انتخاباتی ایالت‌های آمریکا در سال ۱۹۸۸

فصل ۲

استقرای ریاضی

هیچ کس جز بنیان‌گذار یک فرضیه، آن را باور نـدارد،
اما جز آزمون‌گر یک تجربه، همه کس آن را باور دارد.
ناشناس

«واضح است» همواره دشمن «اثبات درست» است.
Bertrand Russell (۱۹۷۰-۱۸۷۲)

۲-۱ آشنایی

در فصل‌های آینده خواهیم دید که استقرا نقشی محوری در طراحی الگوریتم بازی می‌کند. در این فصل به کمک چند مثال - از آسان تا بسیار مشکل - و به طور خلاصه استقرای ریاضی را معرفی مـی‌کنـیم. خوانندگانی که اثبات‌های استقرایی زیادی ندیده‌اند، ممکن است این فصل را قدری دشوار بدانند. ما ادعا می‌کنیم که فرایند ساخت اثبات‌ها مشابه فرایند ساخت الگوریتم‌هاسـت. بنـابراین تجربـه‌ی اثبـات‌هـای استقرایی را برای ساخت الگوریتم‌ها بسیار مفید می‌دانیم.

استقرای ریاضی روش بسیار نیرومندی برای اثبات است و معمولاً به این صورت انجام مـی‌شـود: T را قضیه‌ای می‌گیریم که می‌خواهیم اثبات کنیم. فرض کنید T شامل پـارامتر n اسـت و مقـدار ایـن پارامتر می‌تواند هر عدد طبیعی باشد. (اعداد طبیعی همان اعداد صحیح مثبت هستند.) به جای آن‌که بـه طور مستقیم ثابت کنیم T برای تمام مقادیر n برقرار است، این دو مطلب را ثابت می‌کنیم:

۱- T برای n=1 برقرار است.

۲- برای هر n>1، اگر T برای n-1 برقرار باشد، آنگاه T برای n نیز برقرار است.

روشن است اثبات این دو مطلب برای اثبات قضیه کافی است. از ۱ و ۲ به طور مستقیم نتیجه می‌گیریم که T برای n=2 نیز برقرار است. اگر T برای n=2 برقرار باشد، از شرط ۲ نتیجه می‌شود T بـرای n=3 نیز برقرار است و ... . اصل استقرا چنان روشن است که معمولاً ثابت نمی‌شود؛ بلکه به صورت یک اصل موضوع در تعریف اعداد طبیعی بیان می‌گردد.

معمولاً اثبات ۱ ساده است. اثبات ۲ در بسیاری حالات آسان‌تر از اثبات مستقیم قضیه است، زیرا می‌توانیم از فرض درستی T برای n-1 استفاده کنیم. این فرض، فرض استقرا نام دارد. به عبارت دیگر، فرض استقرا بی‌هیچ زحمتی در اختیار ماست. به جای اثبات با دست خالی، می‌توانیم درستی قضیه را برای مقادیر کوچک‌تر n فرض بگیریم. پس ما بر اثبات قضیه از روی درستی آن برای مقادیر کوچک‌تر تمرکز می‌کنیم. بیایید کار را با یک مثال آغاز کنیم:

□ قضیه‌ی ۲-۱

برای تمام اعداد طبیعی x و n، x^n-1 بر x-1 بخش‌پذیر است.

برهان: اثبات با استقرا روی n انجام می‌شود. قضیه برای n=1 درست است. فرض می‌کنیم قضیه برای n-1 درست باشد؛ یعنی فرض می‌کنیم $x^{n-1}-1$ برای تمام اعداد طبیعی x، بر x-1 بخش‌پذیر باشد. حال، باید ثابت کنیم x^n-1 بر x-1 بخش‌پذیر است. ایده‌ی این کار تلاش برای نوشتن x^n-1 برحسب $x^{n-1}-1$ است: ($x^{n-1}-1$ طبق فرض استقرا بر x-1 بخش‌پذیر است.)

$$x^n-1=x(x^{n-1}-1)+(x+1)$$

قسمت نخست عبارت، یعنی $x(x^{n-1}-1)$ طبق فرض استقرا بر x-1 بخش‌پذیر است و قسمت دوم آن هم خود x-1 است.

□

اصل استقرا این‌گونه تعریف می‌شود:

اگر گزاره‌ی P که دارای پارامتر n است، برای n=1 درست باشد و اگر برای هر n>1 از درستی P برای n-1، بتوان درستی P برای n را نتیجه گرفت، آنگاه P برای تمام اعداد طبیعی درست است.

گاهی به جای n-1 و n، به ترتیب از n و n+1 استفاده می‌کنیم که با هم معادل هستند:

اگر گزاره‌ی P که دارای پارامتر n است، برای n=1 درست باشد و اگر برای هر $n \geq 1$ از درستی P برای n، بتوان درستی P برای n+1 را نتیجه گرفت، آنگاه P برای تمام اعداد طبیعی درست است.

اثبات قضیه‌ی ۲-۱ نمونه‌ای ساده از کاربرد استقراست. استقرا انواع گوناگونی دارد. برای مثال، این گونه از استقرا - که استقرای قوی نام دارد - بسیار رایج است:

اگر گزاره‌ی P که دارای پارامتر n است، برای n=1 درست باشد و برای هر n>1، از درستی P برای همه‌ی اعداد طبیعی کوچک‌تر از n بتوان درستی P را برای n نتیجه گرفت، آنگاه P برای تمام اعداد طبیعی درست است.

تفاوت صورت اخیر این است که در اثبات درستی P برای n می‌توانیم از فرض درستی P برای همه‌ی اعداد کوچک‌تر از n استفاده کنیم. این فرض قوی‌تر، خیلی جاها ممکن است بسیار مفید باشد. نوع ساده‌ی دیگری از استقرا چنین است:

اگر گزاره‌ای P که دارای پارامتر n است، برای n=1 و n=2 درست باشد و اگر برای هر n>2 از درستی P برای n-2، بتوان درستی P را برای n نتیجه گرفت، آنگاه P برای تمام اعداد طبیعی درست است.

این نوع از استقرا در دو مسیر موازی عمل می‌کند: از حالت پایه‌ی n=1 و گام استقرا، برقراری P برای تمام اعداد فرد نتیجه می‌شود. از حالت پایه‌ی n=2 و گام استقرا، برقراری P برای همه‌ی اعداد زوج نتیجه می‌شود. دیگر نوع رایج چنین است:

اگر گزاره‌ی P که دارای پارامتر n است، برای n=1 درست باشد و اگر برای هر n>1 به طوری که n توان صحیحی از ۲ باشد، از درستی P برای n/2 بتوان درستی P برای n را نتیجه گرفت، آنگاه P برای تمام توان‌های صحیح مثبت ۲ درست است.

(سودمندی این صورت از استقرا در اثبات قضیه‌ی ۲-۱۳ نشان داده شده است - مترجمان) این نوع از استقرا، از نوع اول آن با نوشتن پارامتر n به صورت 2^k و اعمال استقرا روی k (با شروع از k=0) به دست می‌آید.

می‌توان از اصل استقرا برای اثبات ویژگی‌های ساختارهای دیگری به غیر از اعداد نیز سود جست. در بیش‌تر این حالت‌ها استقرا روی اندازه‌ی مسأله است. پیدا کردن معیار مناسبی که استقرا روی آن انجام پذیرد، همیشه کار راحتی نیست (مثلاً ما می‌توانستیم در مثال پیش به جای n استقرا را روی x بنا کنیم که اثبات را بسیار پیچیده‌تر می‌کرد). گاهی اندازه‌ی نمونه، عددی طبیعی نیست و باید عددی دیگر، مناسب با هدف استقرا پیدا کنیم. لغزش معمول در این نوع اثبات‌ها، گسترش دادن ادعا از ساختارهای کوچک‌تر به ساختارهای بزرگ‌تر است.

۲-۲ سه مثال ساده

می‌خواهیم عبارتی برای جمع n عدد طبیعی نخست، یعنی $S(n)=1+2+\ldots+n$ بیابیم. برای این کار قضیه‌ی ۲-۲ را ثابت می‌کنیم.

□ قضیه‌ی ۲-۲

جمع n عدد طبیعی نخست، عبارت است از: $n(n+1)/2$.

برهان: اثبات با استقرا روی n است. اگر n=1، آنگاه ادعای گفته‌شده درست است، زیرا $S(1)=1=1.(1+1)/2$. حال فرض می‌کنیم جمع n عدد طبیعی نخست، یعنی S(n)، برابر با $n(n+1)/2$ باشد و از روی آن ثابت می‌کنیم، جمع n+1 عدد طبیعی نخست عبارت است از $S(n+1)=(n+1)(n+2)/2$. از تعریف S(n) داریم: $S(n+1)=S(n)+n+1$ و طبق فرض استقرا نیز داریم: $S(n)=n(n+1)/2$ و بنابراین: $S(n+1)=n(n+1)/2+n+1=(n+2)(n+1)/2$ که همان چیزی است که می‌خواستیم ثابت کنیم. □

حال، مثال پیچیده‌تری را بررسی می‌کنیم. فرض کنید می‌خواهیم حاصل این جمع را محاسبه کنیم: $S(n)=8+13+18+23+\ldots+(3+5n)$. $T(n)$ در مثال پیش برابر با $n^2/2+n/2$ بود. هر یک از اجزای مثال اخیر ۳ واحد از ۵ برابر اجزای مثال قبل بیش‌ترند؛ پس حدس می‌زنیم که $T(n)$ نیز عبارتی از درجه‌ی دوم است. بیایید حدسمان را بررسی کنیم: $G(n)=c_1n^2+c_2n+c_3$. اگر مقادیری مناسب برای پارامترهای c_1، c_2 و c_3 مشخص کنیم، کار انجام شده است. مثلاً می‌توان این ضرایب را با بررسی چند جمله‌ی نخست عبارت یافت. اگر $n=0$، حاصل‌جمع نیز 0 خواهد بود، پس c_3 هم باید 0 باشد. با بررسی $G(1)$ و $G(2)$ به دو معادله‌ی:

(1) $1.c_1+1.c_2=8$
(2) $4.c_1+2.c_2=13+8$

اگر (۱) را دو برابر کرده، سپس حاصل را از (۲) کم کنیم، خواهیم داشت: $2c_1=5$ و از آنجا $c_1=2.5$ و $c_2=5.5$. بنابراین حدس می‌زنیم: $G(n)=2.5n^2+5.5n$. حال می‌کوشیم با استقرا ثابت کنیم: $G(n)=T(n)$. درستی حالت پایه را کمی پیش نشان داده‌ایم. بنابراین فرض می‌کنیم: $G(n)=T(n)$ و می‌کوشیم تا از روی آن ثابت کنیم: $G(n+1)=T(n+1)$. به این صورت عمل می‌کنیم:

طبق فرض استقرا: $T(n+1)=T(n)+5(n+1)+3=G(n)+5(n+1)+3$

$=2.5\,n^2+5.5n+5n+8=2.5n^2+5n+2.5+5.5n+5.5$

$=2.5(n+1)^2+5.5(n+1)=G(n+1)$

که در واقع قضیه‌ی ۲-۳ را ثابت کرده‌ایم.

□ قضیه ۲-۳

جمع سری

$8+13+18+23+\ldots+(3+5n)$

برابر با $2.5n^2+5.5n$ است.

□

این بخش را با مثال ساده‌ی دیگری به پایان می‌رسانیم:

□ قضیه‌ی ۲-۴

اگر n عددی طبیعی باشد و $1+x>0$ آنگاه

$$(1+x)^n\geq 1+nx \qquad (۱-۲)$$

برهان: اثبات با استقرا روی n صورت می‌گیرد. اگر $n=1$، هر دو سمت نامساوی (۲-۱) نیز برابر با $1+x$ هستند. فرض می‌کنیم برای x هایی که نامساوی $1+x>0$ درست است، نامساوی $(1+x)^n\geq 1+nx$ نیز برقرار باشد. باید ثابت کنیم برای x هایی که نامساوی $1+x>0$ درست است، نامساوی $(1+x)^{n+1}\geq 1+(n+1)x$ نیز برقرار است؛ اما طبق فرض استقرا:

$(1+x)^{n+1}=(1+x)(1+x)^n\geq(1+x)(1+nx)=$: $1+x>0$

$1+(n+1)x+nx^2\geq 1+(n+1)x$: $nx^2\geq 0$

پس: $(1+x)^{n+1}\geq 1+(n+1)x$.

□

۲-۳ شمارش ناحیه‌های یک صفحه

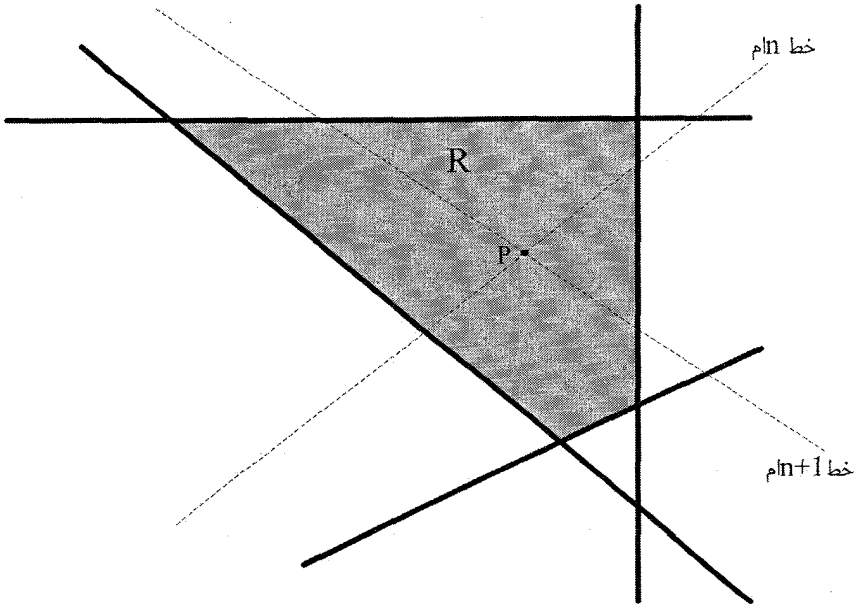
وضعیت یک مجموعه خط از صفحه، عمومی گفته می‌شود، اگر هیچ دو خطی موازی نباشند و هیچ سه خطی از یک نقطه‌ی مشترک نگذرند. مسأله، شمارش تعداد ناحیه‌هایی است که از وضعیت عمومی n خط در صفحه تشکیل می‌شوند. بررسی موارد کوچک کمک می‌کند برای پاسخ، حدس درستی بزنیم. برای n=1، ۲ ناحیه و برای ۲ خط متقاطع، ۴ ناحیه و برای ۳ خط با وضعیت عمومی، ۷ ناحیه در صفحه تشکیل می‌شود. دست‌کم برای $i\leq 3$ چنین به نظر می‌رسد که خط iام، i ناحیه به صفحه می‌افزاید. اگر این حدس برای همه‌ی iها درست باشد، تعداد ناحیه‌ها به آسانی از S(n) - که پیش‌تر حساب شد - به دست خواهد آمد. بنابراین هنگامی که خطی به صفحه افزوده می‌شود، ما به افزایش تعداد ناحیه‌ها توجه می‌کنیم و می‌کوشیم ثابت کنیم:

حدس: افزودن خطی به n-1 خط صفحه با حفظ وضعیت عمومی خطها، n ناحیه به تعداد ناحیه‌های صفحه می‌افزاید.

چنان‌که دیدیم، حدسمان برای $n\leq 3$ درست است. حال این حدس را فرض استقرا می‌گیریم و می‌کوشیم ثابت کنیم افزودن یک خط به n خط صفحه با حفظ وضعیت عمومی، n+1 ناحیه به تعداد ناحیه‌های صفحه می‌افزاید. دقت کنید که فرض استقرا به صورت مستقیم با تعداد ناحیه‌ها سر و کار ندارد؛ بلکه با افزایش تعداد آن‌ها هنگام افزودن یک خط، مربوط است. بنابراین حتا اگر حکم استقرا ثابت شود، ناچاریم تعداد کل ناحیه‌ها را محاسبه کنیم که البته کار راحتی است.

چگونه خط جدید تعداد ناحیه‌های صفحه را افزایش می‌دهد؟ به شکل ۲-۱ توجه کنید. چون همه‌ی خطها در وضعیت عمومی قرار دارند، پس هیچ خطی بر مرز یک ناحیه منطبق نخواهد شد؛ یعنی هر خط یا از یک ناحیه می‌گذرد و آن را به دو ناحیه تقسیم می‌کند (و در نتیجه، یک ناحیه به کل ناحیه‌های صفحه می‌افزاید) و یا اصلاً از آن ناحیه نخواهد گذشت. در نتیجه، تنها باید ثابت کنیم که خط n+1ام دقیقاً از n+1 ناحیه می‌گذرد. اینک با وجود این‌که می‌توانیم قضیه را به طور مستقیم ثابت کنیم، بر آنیم روش دیگری را برای اثبات گام استقرا شرح دهیم. بیایید خط nام را برداریم. بنا بر فرض استقرا، بدون حضور خط nام، خط n+1ام، n ناحیه‌ی جدید به صفحه می‌افزاید؛ پس تنها باید ثابت کنیم که حضور خط nام سبب می‌شود خط n+1ام یک ناحیه‌ی دیگر به صفحه بیفزاید. دوباره خط nام را سرجایش برمی‌گردانیم. از آنجا که همه‌ی خطها در وضعیت عمومی هستند، خط‌های nام و n+1ام در نقطه‌ای مانند P یکدیگر را قطع می‌کنند که باید درون ناحیه‌ای مانند R باشد. پس هر دو خط، ناحیه‌ی R را قطع کرده‌اند. هر یک از این دو خط به طور جداگانه، R را به دو ناحیه تقسیم می‌کنند، اما با همدیگر آن را به چهار ناحیه تقسیم خواهند کرد! بدین‌ترتیب، افزودن خط n+1ام در نبود خط nام، R را به دو ناحیه تقسیم می‌کند؛ اما افزودن خط n+1ام در حضور خط nام، به جای افزودن یک ناحیه، دو ناحیه به R می‌افزاید. (یعنی R از دو ناحیه به چهار ناحیه تقسیم می‌شود.) از طرفی R تنها ناحیه‌ای است

که از افزایش هر دو خط nام و n+1ام تاثیر پذیرفته است، زیرا این دو خط همدیگر را تنها در یک نقطه قطع می‌کنند. در نتیجه، خط n+1ام در نبود خط nام، n ناحیه و در بودنش، n+1 ناحیه به صفحه می‌افزاید؛ بدین ترتیب اثبات کامل می‌شود.



شکل ۲-۱ n+1 خط در وضعیت عمومی

□ قضیه‌ی ۲-۵

تعداد ناحیه‌های حاصل از n خط با وضعیت عمومی در صفحه برابر با n(n+1)/2+1 است.

برهان: ثابت کردیم خط nام، n ناحیه به صفحه می‌افزاید. اگر تنها یک خط در صفحه باشد، ۲ ناحیه در صفحه وجود خواهد داشت. پس تعداد کل ناحیه‌های صفحه (برای n>1) عبارت است از 2+2+3+4+5+...+n. پیش‌تر دیدیم که 1+2+3+...+n=n(n+1)/2؛ بنابراین تعداد کل ناحیه‌ها برابر با n(n+1)/2+1 خواهد شد.

□

توجه: در این اثبات دو نکته‌ی جالب وجود دارد: نخست این‌که به جای کار با خود تابع، با تغییراتش کار کردیم. در نتیجه اثبات استقرایی، با توجه به مقداری بود که با تغییر تابع، به آن افزوده می‌شد. پس لازم نیست فرض استقرا به گونه‌ای تعریف شود که قضیه به صورت مستقیم ثابت گردد، چون می‌توانیم اثبات را در چند مرحله انجام دهیم. تا وقتی که در این مراحل اثبات، چیزهای بیش‌تری از مسأله می‌فهمیم، حرکتمان رو به جلو و در جهت حل مسأله است. نباید شتاب کنیم که با سرعت بیش‌تر به پاسخ برسیم؛ معمولاً شکیبایی، چاره‌ساز است. نکته‌ی دیگر این بود که از فرض استقرا دو بار و در دو موقعیت مختلف

استفاده کردیم: یک بار برای خط l_n و بار دیگر برای خط l_{n+1} درجایگاه خط l_n. استفاده‌ی مضاعف از فرضیات، کار بدی نیست، بلکه به ما می‌آموزد از فرض‌های موجود بیش‌ترین بهره را ببریم.

۲-۴ یک مسأله‌ی رنگ‌آمیزی ساده

فرض کنید n خط مجزا در صفحه داریم که ممکن است نسبت به یکدیگر وضعیت عمومی نداشته باشند. می‌خواهیم ناحیه‌های بین این خطوط را به صورتی رنگ‌آمیزی کنیم که رنگ هر ناحیه با ناحیه‌های همسایه‌اش متفاوت باشد. (دو ناحیه همسایه‌اند، اگر و تنها اگر یالی مشترک داشته باشند.) اگر بتوانیم ناحیه‌ها را این‌گونه رنگ‌آمیزی کنیم، می‌گوییم ناحیه‌های صفحه «رنگ‌شدنی» هستند و رنگ‌آمیزی آن‌ها را یک «رنگ‌آمیزی معتبر» می‌نامیم. در حالت کلی هر نقشه‌ی رسم‌شده روی صفحه‌ی عادی را می‌توان با چهار رنگ، رنگ‌آمیزی کرد. (اثبات این موضوع حدود صد سال وقت ریاضی‌دانان را گرفت و سرانجام به تازگی اثبات شد.) ناحیه‌هایی که خطوط، در صفحه‌ی عادی تشکیل می‌دهند، ویژگی‌هایی دارد که قضیه‌ی ۲-۶ یکی از آن‌ها را بیان می‌کند: (توجه کنید که نقشه‌هایی که روی سطح چنبره یا صفحات نااقلیدسی ترسیم شوند، ممکن است این ویژگی را نداشته باشند - مترجمان)

□ قضیه ۲-۶

ناحیه‌های حاصل از هر تعداد خط در صفحه، تنها با دو رنگ «رنگ‌شدنی» هستند.

برهان: از فرض طبیعی استقرا استفاده می‌کنیم.

فرض استقرا: ناحیه‌های حاصل از خطوطی در صفحه که تعدادشان از n کم‌تر باشد، تنها با دو رنگ «رنگ‌شدنی» هستند.

روشن است که برای $n=1$ وجود دو رنگ لازم و کافی است. حال، به فرض استقرا توجه کنید. پرسش این است که چه باید کرد تا با افزوده شدن خط l_n، رنگ‌آمیزی، معتبر بماند. ناحیه‌های صفحه را با توجه به این‌که در کدام سمت خط l_n هستند، به دو گروه تقسیم کنید. رنگ ناحیه‌های یک سمت را تغییر ندهید، ولی رنگ ناحیه‌های سمت دیگر را وارونه کنید. برای این‌که ثابت کنیم «رنگ‌آمیزی معتبری» انجام داده‌ایم، دو ناحیه‌ی همسایه‌ی R1 و R2 را در نظر می‌گیریم. اگر هر دوی آن‌ها در یک سمت خط l_n باشند، حتماً پیش از افزوده شدن آن خط رنگ‌های متفاوتی داشته‌اند (بنا بر فرض استقرا). می‌توان رنگ آن‌ها را وارونه کرد، اما هنوز هم رنگشان متفاوت از یکدیگر است. اگر یال بین این دو ناحیه، بخشی از خط l_n باشد، آنگاه این دو ناحیه پیش از افزوده شدن خط l_n جزو یک ناحیه و بنابراین هم‌رنگ بوده‌اند. از آنجا که در این حالت، رنگ یکی از این دو ناحیه وارونه شده است، پس بازهم رنگشان با یکدیگر متفاوت است.

□

توجه: شیوه‌ی کلی این مثال، جست‌وجو برای انعطاف یا جست‌وجو برای درجه‌ی آزادی بیش‌تر است. ایده‌ی این شیوه، گسترش فرض استقرا تا جای ممکن برای استخراج چیزهای بیش‌تری از دل آن است. فکر اصلی این مثال چنین بود که با داشتن یک «رنگ‌آمیزی معتبر» می‌توان تمام رنگ‌ها را وارونه کرد و بازهم «رنگ‌آمیزی معتبری» داشت. پیش‌تر نیز برای شمارش ناحیه‌های جدید صفحه، با افزون خطی دیگر به صفحه، این ایده را به کار برده بودیم.

۲-۵ مسأله‌ای پیچیده‌تر درباره‌ی حاصل‌جمع

این مثال قدری پیچیده‌تر است. مثلث زیر را در نظر بگیرید:

۱	= ۱
۳ + ۵	= ۸
۷ + ۹ + ۱۱	= ۲۷
۱۳ + ۱۵ + ۱۷ + ۱۹	= ۶۴
۲۱ + ۲۳ + ۲۵ + ۲۷ + ۲۹	= ۱۲۵

مسأله، یافتن عبارتی برای حاصل‌جمع سطر iام و اثبات درستی آن است. به نظر می‌رسد که جمع سطرها الگویی منظم دارد و دنباله‌ای از مکعب‌های اعداد طبیعی است.

فرض استقرا: جمع سطر iام در مثلث پیش برابر i^3 است.

این مسأله با یاری یک شکل بیان شده است و تعریف دقیق فرض استقرا در آن، کار آسانی نیست. در عمل، عجیب نیست اگر برخی از مسأله‌ها تعریف مبهمی داشته باشند. بخش عمده‌ای از هر راه‌حل، فهم و استخراج درست مسأله است. برای حل این مسأله با توجه به مثلثی که نشان داده شد، فرضیاتی می‌سازیم و سپس با در نظر داشتن این فرضیات، مسأله را حل می‌کنیم. (در حالت کلی ممکن است بتوان فرضیات گوناگونی در نظر گرفت.) سطر iام شامل i عدد فرد متوالی است. حال، به اختلاف دو سطر پشت‌سرهم توجه می‌کنیم. برای این‌که ثابت کنیم جمع اعداد سطر iام، i^3 است، نشان می‌دهیم اختلاف بین سطر iام و سطر $i+1$ام برابر با $(i+1)^3-i^3$ است. (مثلث پیش نشان می‌دهد فرض استقرا برای $i \leq 4$ درست است.)

اختلاف بین نخستین عدد سطر $i+1$ام و نخستین عدد سطر iام چقدر است؟ از آنجا که در سطر i، i عدد فرد متوالی وجود دارد، پس اختلاف این دو عدد $2i$ خواهد بود. اختلاف بین عدد دوم سطر $i+1$ و عدد دوم سطر i، بین عدد سوم دو سطر، بین عدد چهارم دو سطر و ... نیز همین است. روی هم، i اختلاف وجود دارد که مقدار همه‌ی آن‌ها $2i$ است. سطر $i+1$ام نسبت به سطر iام یک عنصر اضافه نیز دارد. در نتیجه، اختلاف بین دو سطر $2i^2$ به اضافه‌ی مقدار آخرین عدد سطر $i+1$ است. از آنجایی که $(i+1)^3-i^3=3i^2+3i+1$، پس تنها کافی است ثابت کنیم مقدار آخرین عدد سطر $i+1$ برابر با

$3i^2+3i+1-2i^2=i^2+3i+1$ است. اینجاست که حدس حاصل‌جمع i^3 به کار آمد و توانستیم مسأله‌ی یافتن جمع را به مسأله‌ی یافتن یک عنصر کاهش دهیم. این گزاره را نیز با استقرا ثابت می‌کنیم:

فرض استقرای درونی (تودرتو): آخرین عدد سطر i+1، i^2+3i+1 است.

این ادعا برای i=1 درست است. باید ثابت کنیم که اختلاف بین آخرین عدد سطر i+1 و آخرین عدد سطر i برابر با

$$[i^2+3i+1]-[(i-1)^2+3(i-1)+1]=2i+2$$

است؛ اما کمی پیش‌تر دیدیم که اختلاف بین دو عنصر متناظر از سطر i+1 و سطر i برابر با 2i است. پس حدسمان ثابت شد.

توجه: این برهان دوباره نشان داد که نباید انتظار داشته باشیم همیشه در یک مرحله به اثبات کامل برسیم. افزودن مرحله‌ای دیگر به مراحل اثبات، به شرطی که در حل مسأله پیشرفت کنیم، خوب است. این برهان نمونه‌ای از شیوه‌ی «عقب‌رو» برای رسیدن به اثبات است. به جای شروع از مسأله‌ی ساده‌تر و تلاش برای رسیدن به مسأله‌ی نهایی، کارمان را از مسأله‌ی نهایی شروع کردیم، سپس آن را ساده و ساده‌تر کردیم. این شیوه نه تنها در ریاضیات، بلکه در زمینه‌های گوناگون دیگری نیز بسیار رایج است.

۲-۶ یک نامساوی ساده

در این بخش یک نامساوی را ثابت می‌کنیم.[۱]

□ قضیه‌ی ۲-۷

برای هر $n\geq 1$:

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots+\frac{1}{2^n}<1 \qquad \textbf{(۲-۲)}$$

برهان: می‌خواهیم قضیه را با استقرا ثابت کنیم. روشن است که قضیه برای n=1 درست است. فرض می‌کنیم نامساوی (۲-۲) برای n برقرار است و ثابت می‌کنیم برای n+1 نیز برقرار خواهد بود. تنها اطلاعی که از فرض استقرا به دست می‌آوریم، این است که جمع n جمله‌ی نخست از ۱ کوچک‌تر است. چگونه می‌توانیم این موضوع را گسترش دهیم تا جمله‌ی n+1ام را نیز شامل شود؟ چراکه افزودن $1/2^{n+1}$ به سمت چپ نامساوی (۲-۲)، ممکن است حاصل‌جمع آن را به بیش از ۱ افزایش دهد. چاره این است که از استقرا به ترتیبی دیگر استفاده کنیم. در جمع

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots+\frac{1}{2^n}+\frac{1}{2^{n+1}}$$

۱- این نامساوی معمولاً در بحث همگرایی سری‌های نامتناهی مطرح می‌شود؛ ولی ما فرض می‌کنیم که هیچ دانشی درباره‌ی سری‌ها نداریم و این رابطه کاملاً متناهی است.

به n جمله‌ی آخر توجه می‌کنیم:

$$\frac{1}{4}+\frac{1}{8}+\ldots+\frac{1}{2^n}+\frac{1}{2^{n+1}}=\frac{1}{2}\left[\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots+\frac{1}{2^n}\right]<\frac{1}{2}$$

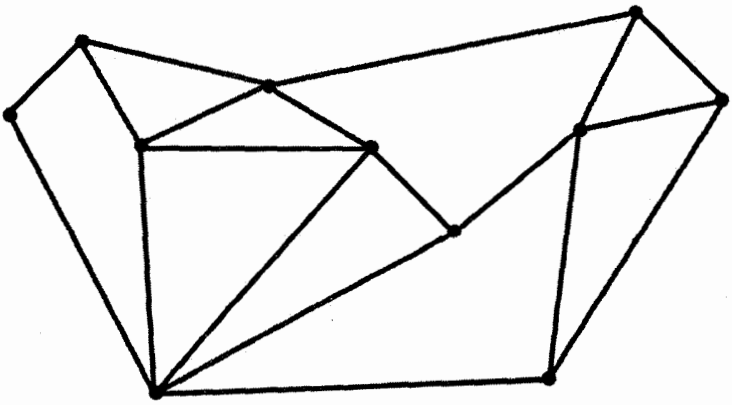
نامساوی آخر از فرض استقرا نتیجه شده است. حال با افزودن $\frac{1}{2}$ به دو سمت نامساوی اخیر، به نامساوی (۲-۲) برای n+1 می‌رسیم.

□

توجه: در استقرا لازم نیست آخرین عنصر را عنصر n+1ام در نظر بگیریم. گاهی اگر عنصر نخست را عنصر n+1ام استقرا بگیریم، اثبات آسان‌تر می‌شود. نمونه‌های دیگری نیز وجود دارد که در آن‌ها بهتر است عنصر n+1ام را عنصری بگیریم که ویژگی‌های خاصی دارد. چنان‌چه با مسأله‌ای روبه‌رو شدید، انعطاف داشته باشید و تا جایی که می‌توانید گزینه‌های گوناگون را در نظر بگیرید. به مثال‌های بعد دقت کنید.

۲-۷ رابطه‌ی اویلر

برهان بعدی برای قضیه‌ای است که به رابطه‌ی اویلر مشهور شده است. نقشه‌ای رسم‌شده روی یک صفحه‌ی معمولی را در نظر بگیرید که V رأس، E یال و F وجه داشته باشد. (وجه، ناحیه‌ای بسته است؛ ناحیه‌ی بیرونی نیز یک وجه محسوب می‌شود. برای مثال یک مربع، چهار رأس، چهار یال و دو وجه دارد.) نقشه‌ی شکل ۲-۲، ۱۱ رأس، ۱۹ یال و ۱۰ وجه دارد. دو رأس یک نقشه را متصل گوییم، اگر بتوان با گذر از یال‌های نقشه از یک رأس به رأس دیگر رسید. اگر هر دو رأس از نقشه‌ای متصل باشند، آن نقشه را همبند گویند. به صورت شهودی می‌توان گفت یک نقشه هنگامی همبند است که از یک بخش تشکیل شده باشد.



شکل ۲-۲ نقشه‌ای رسم‌شده روی یک صفحه‌ی معمولی با ۱۱ رأس، ۱۹ یال و ۱۰ وجه

□ قضیه‌ی ۲-۸

در یک نقشه‌ی همبند رسم‌شده روی یک صفحه‌ی معمولی بین تعداد رأس‌ها (V)، تعداد یال‌های (E) و تعداد وجه‌ها (F) رابطه‌ی V+F=E+2 برقرار است.

برهان: این قضیه را با نوعی از استقرا که استقرای دوگانه خوانده می‌شود، ثابت می‌کنیم. استقرا، نخست روی تعداد رأس‌ها و سپس روی تعداد وجه‌ها بنا می‌شود.

نخست نقشه‌ای با یک وجه در نظر بگیرید. این نقشه دارای دور نیست (چراکه درون هر یک از دورها یک وجه به وجود می‌آید؛ یک وجه بیرونی نیز وجود خواهد داشت). گراف همبندی که دور نداشته باشد، درخت نامیده می‌شود. ابتدا برای تمام درخت‌ها رابطه‌ی V+1=E+2 را ثابت می‌کنیم. (اگر تعریف همبندی یا دور را نمی‌دانید، بخش ۷-۱ را ببینید - مترجمان)

فرض استقرای فرعی: یک درخت با n رأس، n-1 یال دارد.

حالت پایه‌ی استقرا روشن است. با فرض این‌که درختی با n رأس، n-1 یال داشته باشد، درختی را با n+1 رأس در نظر می‌گیریم. دست‌کم باید یک رأس، مثلاً v، تنها به یک یال متصل باشد. چراکه اگر هر رأس دست‌کم به دو یال متصل باشد و ما با شروع از رأسی دل‌خواه از یال‌های درخت بگذریم، آنگاه به یقین به یکی از رأس‌هایی باز خواهیم گشت که پیش‌تر در آن بوده‌ایم که به معنای وجود دور است (بروز تناقض). حال می‌توانیم رأس v و یال متصل به آن را حذف کنیم. نقشه‌ی حاصل بازهم همبند و در نتیجه یک درخت است. البته یک رأس و یک یال از آن کم شده است که ادعای ما را ثابت می‌کند.

این اثبات، حالتی پایه برای استقرا روی تعداد وجه‌ها محسوب می‌شود.

فرض استقرای اصلی: در هر نقشه‌ی رسم‌شده روی صفحه‌ی معمولی با n وجه، E یال و V رأس، رابطه‌ی V+n=E+2 برقرار است.

نقشه‌ای با n+1 وجه در نظر بگیرید. باید وجهی مانند f همسایه‌ی وجه بیرونی در آن وجود داشته باشد. f یک وجه است، پس درون یک دور قرار دارد. برداشتن یک یال از این دور، نقشه را ناهمبند نمی‌کند. یکی از یال‌های مشترک بین f و وجه بیرونی را برمی‌داریم. پس از این کار، یک وجه و یک یال از نقشه کم خواهد شد. بدین‌ترتیب قضیه ثابت می‌شود.

□

توجه: این قضیه سه پارامتر داشت. اثبات روی یک پارامتر (تعداد وجه‌ها) انجام شد، اما حالت پایه با استقرایی دیگر روی پارامتری دیگر (تعداد رأس‌ها) صورت گرفت. این اثبات نشان داد که باید برای استقراها ترتیبی درست برگزینیم. گاهی استقرا، از یک پارامتر به پارامتر دیگر منتقل می‌شود؛ گاهی استقرا بر پایه‌ی مقداری مرکب از چندین پارامتر است و گاهی نیز به طور هم‌زمان روی دو پارامتر مختلف اعمال می‌شود. گزینش ترتیبی درست برای استقراها از دشواری اثبات می‌کاهد. چنان‌که در فصل‌های آینده خواهیم دید، چنین گزینش درستی روی کارایی الگوریتم‌ها نیز مؤثر است.

۲-۸ مسأله‌ای از نظریه‌ی گراف

ابتدا لازم است با برخی مفاهیم پایه در نظریه‌ی گراف آشنا شویم. (درباره‌ی این مفاهیم در فصل ۷ بیش‌تر بحث خواهد شد.) گراف G=(V,E) از مجموعه‌ی رأس‌های V و مجموعه‌ی یال‌های E تشکیل می‌شود. هر یال متناظر با دو رأس متفاوت است. گراف یا جهت‌دار است یا جهت‌دار نیست. یال‌های گراف جهت‌دار زوج مرتب هستند؛ در این زوج‌ها ترتیبی که یک یال، دو رأس را به هم وصل می‌کند، اهمیت دارد. در این حالت، یال را به صورت یک پیکان از یک رأس به رأس دیگر می‌کشیم. رأس شروع را، دم یا مبدأ و رأس پایان را، سر یا مقصد گویند. یال‌های گراف بدون‌جهت، زوج‌هایی نامرتب هستند. در این بخش، ما با گراف‌های جهت‌دار کار می‌کنیم. درجه‌ی رأس v، تعداد یال‌هایی است که از آن می‌گذرند. یک مسیر، دنباله‌ای از رأس‌های v_1، v_2، ... و v_k است که با یال‌های $(v_1\ v_2)$، $(v_2\ v_3)$، ... و $(v_{k-1}\ v_k)$ به یکدیگر متصل شده‌اند (معمولاً این یال‌ها را هم جزو مسیر به حساب می‌آوریم). رأس u را از رأس v دست‌رس‌پذیر می‌گوییم، اگر مسیری از v به u وجود داشته باشد. G=(V,E) را یک گراف و U را زیرمجموعه‌ای از رأس‌های آن در نظر بگیرید($U\subseteq V$). زیرگراف برآمده از U، زیرگراف H=(U,F) است که در آن F زیرمجموعه‌ای از یال‌های E است که دو سر آن‌ها به U تعلق دارند (معمولاً این زیرگراف را زیرگراف القایی با U می‌نامند). مجموعه‌ی مستقل S در گراف G=(V,E)، مجموعه‌ای از رأس‌های گراف است که هیچ دوتای آن‌ها همسایه‌ی یکدیگر نباشند.

□ قضیه ۲-۹

گراف G=(V,E) را یک گراف جهت‌دار بگیرید. مجموعه‌ی مستقل S(G) در گراف وجود دارد به گونه‌ای که هر رأس گراف G از رأسی در S(G) با مسیری حداکثر به طول ۲ دست‌رس‌پذیر است.

برهان: اثبات با استقرا روی تعداد رأس‌ها انجام می‌شود.

فرض استقرا: قضیه ۲-۹ برای تمام گراف‌های جهت‌داری که کم‌تر از n رأس دارند، برقرار است.

قضیه برای $n\leq 3$ روشن است. v را رأسی دل‌خواه در V بگیرید. فرض کنید: $N(v) = \{v\} \cup \{w \in V \mid (v,w) \in E\}$. در واقع N(v) مجموعه‌ی همسایه‌های v است. گراف H برآمده از (یا القایی با) مجموعه رأس‌های V-N(v)، از گراف G رأس‌های کم‌تری دارد. پس می‌توانیم از فرض استقرا برای H استفاده کنیم. S(H) را مجموعه‌ی مستقلی از گراف H بگیرید. روشن است که فرض استقرا برای آن برقرار است. ممکن است یکی از دو حالت صفحه‌ی بعد رخ دهد:

۱- $S(H) \cup \{v\}$ مستقل است. در این صورت می‌توانیم $S(G)$ را $S(H) \cup \{v\}$ در نظر بگیریم، زیرا هر رأس $N(v)$ با طی یک یال از رأس v در دسترس است. رأس‌هایی هم که در $N(v)$ نیستند، بنا بر فرض استقرا حداکثر با طی دو یال از رأسی در $S(H)$ دسترس‌پذیرند.

۲- $S(H) \cup \{v\}$ مستقل نیست. در این صورت، باید رأسی مانند w در $S(H)$ باشد که همسایه‌ی رأس v است. از $w \in S(H)$ نتیجه می‌شود: $w \in V\text{-}N(v)$؛ که بر اساس آن در گراف G یالی از رأس v به w وجود ندارد؛ اما از آنجایی که ما فرض کرده بودیم w همسایه‌ی v است، پس (w,v) باید یالی از گراف G باشد. بدین‌ترتیب هر رأس از $N(v)$ با طی حداکثر ۲ یال از w (با عبور از v) دسترس‌پذیر خواهد بود. حال $S(G)$ را $S(H) \cup \{w\}$ می‌گیریم که اثبات را کامل می‌کند.

□

توجه: مقدار کاهش در اینجا ثابت نبود؛ یعنی اندازه‌ی مسأله را از n با توجه به نمونه‌ی مسأله به عددی کوچک‌تر از آن کاهش دادیم. همچنین مسأله‌ی کوچک‌تر، مسأله‌ای با اندازه‌ای اختیاری و دل‌خواه نبود، بلکه اندازه‌اش کاملاً وابسته به شرایط مسأله‌ی بزرگ‌تر بود. ما به تعداد کافی از رأس‌های مسأله‌ی بزرگ‌تر حذف کردیم تا اثبات ممکن شود. در چنین اثبات‌هایی، موازنه‌ای دقیق بین حذف تعداد زیادی رأس و حذف تعداد کمی رأس برقرار است. در حالت نخست، فرض استقرا بسیار ضعیف است و در حالت دوم، فرض استقرا بسیار قوی است. در بیش‌تر موارد، محور اصلی اثبات، کشف این موازنه است. توجه کنید که ما معمولاً از اصل استقرای قوی استفاده می‌کنیم که فرض می‌کند قضیه برای تمام نمونه‌های کوچک‌تر مسأله برقرار است.

۲-۹ کدهای Gray

مجموعه‌ای از n شیء داریم و می‌خواهیم آن‌ها را نام‌گذاری کنیم. هر نام را با رشته‌ای یکتا از بیت‌ها نشان می‌دهیم. ممکن است برای دست‌یابی به روشی خوب در نام‌گذاری، به اهداف مختلفی توجه کنیم. در این مثال، تنها یک هدف وجود دارد. ما می‌خواهیم نام‌ها را در یک فهرست چرخشی به گونه‌ای بچینیم که بتوان هرکدام از آن‌ها را با تغییر دقیقاً یک بیت، از نام قبلی به دست آورد. چنین روشی یک کد Gray نامیده می‌شود.[۲] کدهای Gray کاربردهای گوناگونی دارند. برای مثال، با این روش یک حسگر، بهتر می‌تواند چند شیء را بررسی کند، زیرا تغییر سریع نمایش از یک شیء به شیء بعدی با تغییر یک بیت، شدنی است. این بخش بررسی می‌کند که آیا برای هر تعداد شیء، ساخت یک کد

۲- کدهای Gray، معمولاً هنگامی به کار می‌رود که تعداد اشیاء توانی از ۲ باشد، ولی ما آن را برای تمام مقدارهای n به کار می‌بریم.

Gray امکان‌پذیر است یا نه. روشن است که خود اشیاء نقشی در مسأله ندارند؛ پس تنها به شماره‌ی آن‌ها توجه می‌کنیم.

گراف، راهی مناسب برای نمایش رابطه‌ی میان نام‌ها بر اساس کد Gray است. این نام‌ها به رأس‌های یک گراف نسبت داده می‌شوند و دو رأس هنگامی به یکدیگر متصل می‌شوند که نام‌های نظیرشان تنها در یک بیت متفاوت باشند. «دوری» از این گراف که تمام رأس‌ها را در بر گیرد، یک کد Gray است.

کار را با مقادیر کوچک n آغاز می‌کنیم. حالت‌های n=1 و n=2 روشن هستند.اگر n=3، چطور؟ درک این موضوع چندان دشوار نیست که وجود یک کد Gray به طول ۳ غیرممکن است. اگر در رشته‌ای از بیت‌ها دو تغییر ایجاد کنیم، یا همان رشته حاصل می‌شود، یا رشته‌ای که با رشته‌ی آغازین در دو بیت متفاوت است. پس نمی‌توانیم با سه تغییر به همان رشته‌ی آغازین برسیم. در واقع روشن می‌شود که نمی‌توان یک کد Gray با طول فرد ساخت. اگر n=4، چطور؟ یک کد Gray به طول ۴ عبارت است از: 00، 01، 11، 10 و گراف متناظر با آن مانند یک مربع است. اینک آماده‌ایم تا مسأله را حل کنیم.

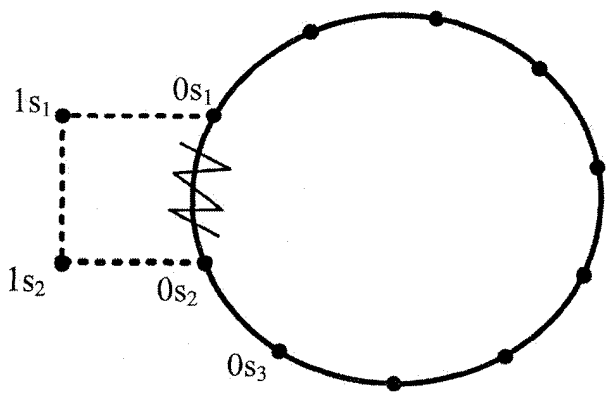
□ قضیه‌ی ۲-۱۰

برای هر عدد صحیح و مثبت k، یک کد Gray به طول 2k وجود دارد.

برهان: اثبات با استقرا روی k انجام می‌شود. حالت k=1 روشن است. فرض کنید یک کد Gray به طول 2k وجود دارد و s_1، s_2، ... و s_{2k} را متناظر با یک کد Gray به اندازه‌ی 2k بگیرید. روشن است که اگر ۱ یا ۰ را به ابتدای همه‌ی رشته‌ها بیفزاییم، نتیجه هنوز هم یک کد Gray است. پس یک کد Gray به اندازه‌ی 2k+2 می‌تواند چنین باشد (شکل ۲-۳ را ببینید):

$0s_1, 1s_1, 1s_2, 0s_2, 0s_3, 0s_4, \ldots, 0s_{2k}$

□



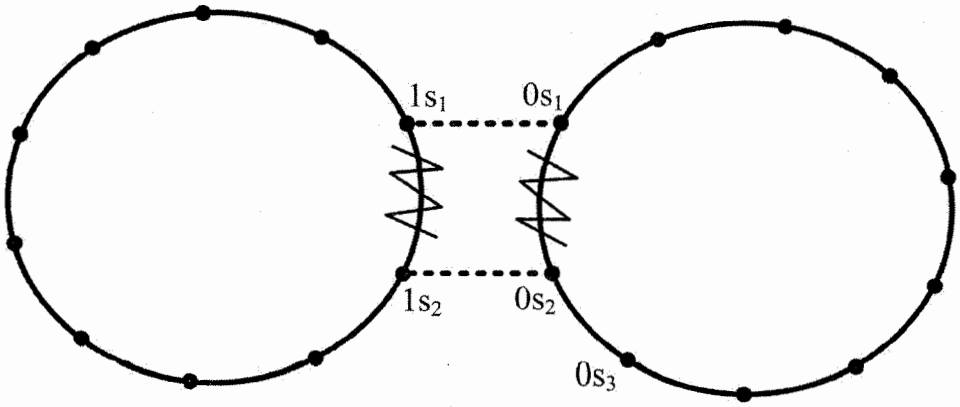
شکل ۲-۳ ساخت یک کد Gray به اندازه‌ی 2k+2

هرچند اثبات کامل است، اما فرایند ساخت چندان دل‌چسب نیست. طول هر رشته در کد Gray دست‌کم نصف تعداد اشیاست. در حالت کلی، نمایش n شیء با $\lceil \log_2'' \rceil$ بیت ممکن است. آیا می‌توانیم با کم‌تر از n/2 بیت، یک کد Gray به اندازه‌ی n بسازیم؟ لگاریتمی بودن تعداد بیت‌ها به این معناست که هرگاه تعداد اشیاء دو برابر شوند، باید یک بیت به کد Gray بیفزاییم. فرض کنید می‌دانیم چگونه برای هر عدد زوج 2k، یک کد Gray بسازیم به گونه‌ای که k<n. حال، هنگامی که 2n شیء داریم، می‌کوشیم از دو کد با اندازه‌ی n، یک کد Gray برای این اشیاء بسازیم.

در اینجا با یک مشکل روبه‌رو می‌شویم. اگر چه 2n زوج است و یک کد Gray به اندازه‌ی آن وجود دارد، اما ممکن است n فرد باشد که در آن صورت کدی از نوع Gray به اندازه‌ی n وجود نخواهد داشت. در نتیجه، ما نمی‌توانیم از فرض استقرا برای nهای فرد استفاده کنیم. بیایید خودمان را محدود به حالتی کنیم که مقدار n توانی از ۲ باشد. اگر بدانیم چگونه برای همه‌ی توان‌های دوی کم‌تر از n می‌توان کد Gray ساخت، می‌توانیم s_1، s_2، ... و $s_{n/2}$ را متناظر با یک کد Gray به اندازه‌ی n/2 بگیریم. اگر به ابتدای این دنباله، ۰ یا ۱ را بیفزاییم، دو دنباله‌ی $0s_1$، $0s_2$، ... و $0s_{n/2}$ و $1s_1$، $1s_2$، ... و $1s_{n/2}$ بازهم کد Gray هستند. حال، می‌توانیم این دو دنباله را به صورت

$1s_2, 0s_2, 0s_3, \ldots, 0s_{n/2}, 0s_1, 1s_1, 1s_{n/2}, 1s_{n/2-1}, \ldots, 1s_2$

در یکدیگر ادغام کنیم (شکل ۲-۴ را ببینید). برای مثال، نشان می‌دهیم چگونه با این روش می‌توان یک کد Gray برای n=4 را به یک کد Gray برای n=8 تبدیل کرد. دو دنباله عبارتند از: 000، 001، 011، 010 و 100، 101، 111، 110. دنباله‌ی ترکیبی چنین می‌شود: 101، 001، 011، 010، 000، 100، 110، 111. بدین‌ترتیب، تنها با استفاده از از یک بیت اضافی در یک کد Gray برای n/2، یک کد Gray برای n ساختیم. از اینجا طول هر رشته $\log_2''$ خواهد شد.



شکل ۲-۴ ساخت یک کد Gray از دو کد Gray کوچک‌تر

چگونه می‌توانیم این فرایند ساخت را به هر مقدار زوج n تعمیم دهیم؟ دشواری ساخت کد Gray به طول فرد را به یاد آورید: ما نمی‌توانستیم در گراف متناظر با آن، دور به وجود آوریم. دوباره به شکل ۲-۴ نگاه کنید. می‌بینیم داشتن دو «دور بسته» ضروری نیست، بلکه تنها کافی است دو «دنباله‌ی باز»

داشته باشیم. اگر بتوانیم یک کد Gray باز با طولی فرد بسازیم، آنگاه این کد می‌تواند برای فرایند کلی ساخت کافی باشد. (کد Gray باز، کدی است که دقیقاً دو نام آن در بیش از یک بیت اختلاف دارند.) (دنباله‌ای را در نظر بگیرید که همه‌ی عناصر متوالی‌اش تنها در یک بیت اختلاف دارند. حال، اگر عنصر اول و عنصر آخر هم در یک بیت اختلاف داشته باشند، آن دنباله را کد Gray بسته می‌گوییم و اگر این دو عنصر در بیش از یک بیت اختلاف داشته باشند، آن را کد Gray باز می‌گوییم - مترجمان) قضیه‌ی ۲-۱۱، مسأله را در هر دو حالت ممکن حل می‌کند.

□ قضیه‌ی ۲-۱۱

برای هر عدد صحیح و مثبت k، یک کد Gray به طول $\lceil \log_2^n \rceil$ وجود دارد که اگر k زوج باشد، این کد Gray، بسته و اگر k فرد باشد، این کد، باز است.

برهان: با یک فرض استقرای قوی‌تر هر دو حالت را ثابت می‌کنیم.

فرض استقرا: برای هر مقدار k که k<n، یک کد Gray به طول $\lceil \log_2^n \rceil$ وجود دارد. اگر k زوج باشد، آنگاه کد، بسته است؛ اما اگر k فرد باشد، کد، باز است.

حالت پایه‌ی استقرا روشن است. حال، یک کد Gray به طول n می‌سازیم. دو حالت ممکن است:

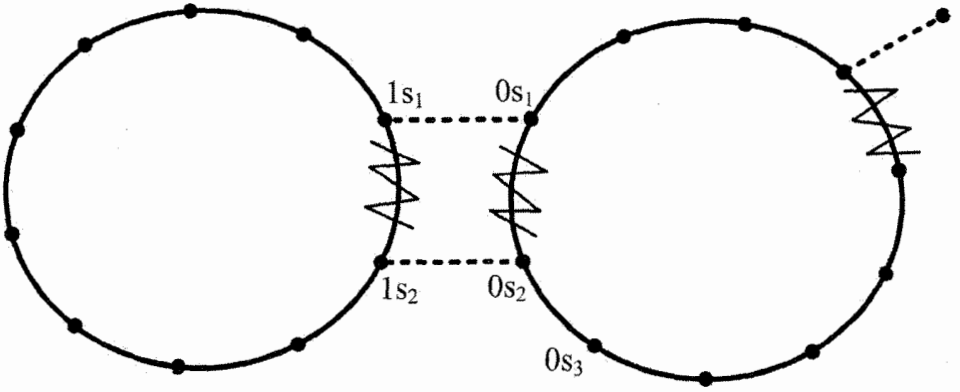
۱- n زوج است. در این حالت، مانند حالتی که n توانی از ۲ بود، عمل می‌کنیم. بنا بر فرض استقرا یک کد Gray به طول n/2 وجود دارد (خواه این کد، بسته باشد یا باز). دو نسخه از این کد را در نظر بگیرید که در یکی پیش از تمام نام‌ها ۰ و در دیگری پیش از تمام نام‌ها ۱ قرار داده‌ایم. این دو کد را همانند شکل ۲-۴ به هم متصل می‌کنیم. بنا بر فرض استقرا تعداد بیت‌های این دو کد $\lceil \log_2^{n/2} \rceil$ است. یک بیت به نام‌ها افزوده‌ایم و تعداد اشیاء دو برابر شده است. بنابراین تعداد بیت‌ها برای کد جدید چنین است: $\lceil \log_2^{n/2} \rceil + 1 = \lceil \log_2^n \rceil$.

۲- n فرد است. فرض می‌کنیم: n=2k+1. دو کد Gray به اندازه‌ی k می‌سازیم و آن‌ها را مانند حالت پیش به یکدیگر متصل می‌کنیم. اگر 2k توانی از ۲ نباشد، آنگاه از چند رشته‌ی به طول $\lceil \log_2^{2k} \rceil$ برای نام‌گذاری استفاده نشده است و می‌توان یکی از این رشته‌ها را به یکی از رشته‌های مورد استفاده متصل ساخت. اینک، دور به طول 2k را با افزودن این رشته‌ی تازه می‌شکنیم که نتیجه‌ی آن مسیری باز به طول 2k+1 خواهد بود (شکل ۲-۵). تعداد بیت‌ها شرط را برآورده می‌کند. اگر 2k توانی از ۲ باشد، هیچ رشته‌ای بدون استفاده نیست و لازم است یک بیت دیگر به کد بیفزاییم. بنابراین تعداد کل بیت‌ها $\lceil \log_2^{2k} \rceil + 1$ است و چون 2k توانی از ۲ است: $\lceil \log_2^{2k} \rceil = \log_2^{2k}$ و در نتیجه $\lceil \log_2^{2k} \rceil + 1 = \lceil \log_2^{2k+1} \rceil$ ($\log_2^{2k} + 1$).

□

توجه: در این مثال، قضیه‌ای با دو حالت جداگانه داشتیم. شاید اگر برای هر حالت، قضیه‌ای جداگانه در نظر می‌گرفتیم، طبیعی‌تر به نظر می‌رسید؛ اما این کار همیشه بهترین روش نیست. با این‌که دو حالت با

هم فرق داشتند، اما با هم در نظر گرفتن آن‌ها آسان‌تر بود. پس هر دوی آن‌ها را در یک فرض استقرا قرار دادیم. با این روش از فرض استقرا برای یک حالت، در حل حالت دیگر بهره‌مند شدیم. این روش، بسیار مانند بالا رفتن از کوه با هر دو پاست. ما گام‌های هر پا را جدای از دیگری طرح‌ریزی نمی‌کنیم، بلکه هر پا از گام‌هایی که پای دیگر برداشته است، بهره‌مند می‌شود. گاهی بهتر است فرض استقرا به گونه‌ای تعریف شود که مسأله‌ای کلی‌تر را در بر گیرد. در این مثال، مسأله‌ی کلی تنها دو حالت داشت. در بخش بعد مثالی عرضه می‌کنیم که نشان می‌دهد گاهی برای حل یک مسأله، بهتر است مسأله‌ی گسترش‌یافته‌ی آن را حل کنیم. فایده‌ی کار کردن با مسأله‌ی کلی‌تر این است که فرض استقرا قوی‌تر می‌شود و می‌توان به گونه‌ی کارآمدتری از آن بهره‌مند شد. در اینجا باید حساب سود و زیان را کرد. ما باید با فرض درستی گزاره‌ی استقرا برای n، آن را برای n+1 ثابت کنیم. پس اگر گزاره‌ی استقرا برای n، قوی‌تر باشد، استفاده از آن در اثبات، آسان‌تر می‌شود و البته چیزهای بیش‌تری هم باید ثابت گردند. درباره‌ی این موضوع در بخش بعد و در بخش ۵-۱۰ بیش‌تر بحث خواهیم کرد. همچنین به جای آن‌که در فرض استقرا تنها به 2n-2 توجه کنیم، تمام مقدارهای کم‌تر از 2n را در نظر گرفتیم.



شکل ۲-۵ ساخت یک کد Gray باز

۲-۱۰ یافتن مسیرهایی با یال‌های «دو به دو جداازهم» در یک گراف

فرض کنید G=(V,E) گرافی همبند و بدون‌جهت باشد. یال‌های دو مسیر در G را «جداازهم» گوییم، اگر این دو مسیر هیچ یال مشترکی نداشته باشند. O را مجموعه‌ی رأس‌هایی از V بگیرید که درجه‌ی‌شان فرد است. ادعا می‌کنیم تعداد رأس‌های O زوج است. برای اثبات این ادعا، درجه‌ی همه‌ی رأس‌ها را با هم جمع می‌کنیم و می‌بینیم حاصل‌جمع دقیقاً دو برابر تعداد یال‌هاست (چون هر یال دو بار شمرده می‌شود) اما از آنجا که همه‌ی رأس‌های دارای درجه‌ی زوج، عددی زوج را به این حاصل‌جمع می‌افزایند، پس باید تعداد رأس‌های با درجه‌ی فرد، زوج باشد. اینک به قضیه‌ی ۲-۱۲ توجه کنید:

☐ قضیه‌ی ۲-۱۲

$G=(V,E)$ را گرافی همبند و بدون‌جهت بگیرید و O را مجموعه‌ی رأس‌هایی از G فرض کنید که درجه‌ی‌شان فرد است. می‌توان رأس‌های O را به زوج‌رأس‌هایی تقسیم کرد و مسیرهایی با یال‌های جداازهم در بین رأس‌های هر زوج یافت.

برهان: اثبات با استقرا روی تعداد یال‌هاست. روشن است که قضیه برای $m=1$ درست است.

فرض استقرا: قضیه برای تمام گراف‌های همبند و بدون‌جهتی که کم‌تر از m یال داشته باشند، برقرار است.

G را گرافی همبند و بدون‌جهت فرض کنید که m یال دارد و O را مجموعه‌ی رأس‌هایی از G بگیرید که درجه‌ی فرد دارند. اگر O تهی باشد، درستی قضیه روشن است. در غیر این صورت، دو رأس دل‌خواه از O را در نظر بگیرید. چون G همبند است، پس دست‌کم یک مسیر وجود دارد که این دو رأس را به هم متصل کند. این مسیر را به طور کامل از G حذف کنید. گراف باقی‌مانده یال‌های کم‌تری دارد؛ بنابراین می‌توانیم از فرض استقرا استفاده کنیم تا مسیرهایی را برای بقیه‌ی رأس‌های فرد پیدا کنیم و بدین‌ترتیب اثبات تکمیل می‌شود؛ اما مشکل این است که با حذف این مسیر، ممکن است گراف ناهمبند شده باشد، در حالی که فرض استقرا تنها برای گراف‌های همبند برقرار است. اینجاست که ما باید در استفاده‌ی درست از فرض استقرا بسیار دقیق باشیم. در این مورد برای پرهیز از مشکل می‌توانیم از روشی هوشمندانه بهره ببریم؛ یعنی با تغییر فرض استقرا، آن را با نیازهای‌مان سازگار می‌کنیم!

مشکل ما لزوم همبندی گراف بود. بیایید این شرط را برداریم. فرض استقرا چنین می‌شود:

فرض اصلاح‌شده‌ی استقرا: قضیه‌ی ۲-۱۲ برای همه‌ی گراف‌های بدون‌جهتی که کم‌تر از m یال دارند، برقرار است.

روشن است که این قضیه قوی‌تر است و جالب اینجاست که اثباتش نیز ساده‌تراست. حال، گرافی بدون‌جهت با m یال در نظر بگیرید و فرض کنید O مجموعه‌ی رأس‌های با درجه‌ی فرد آن باشد. ممکن است این گراف همبند نباشد. در آن صورت، گراف را به چندین مؤلفه‌ی همبند افراز می‌کنیم. از هرمؤلفه، دو رأس با درجه‌ی فرد در نظر می‌گیریم. از آنجا که هر مؤلفه، گرافی همبند است، پس باید تعداد رأس‌های فردش، زوج باشد. از این رو، اگر در یک مؤلفه، رأسی فرد وجود داشته باشد، باید دست‌کم یک رأس فرد دیگر نیز در آن مؤلفه موجود باشد. به این ترتیب، اساس کار را انجام داده‌ایم. به دلیل این‌که دو رأس فرد برگزیده‌شده، در یک مؤلفه قرار دارند، می‌توانیم آن‌ها را با یک مسیر به یکدیگر متصل کنیم. سپس آن مسیر را حذف می‌کنیم. حال، گراف، کم‌تر از m یال دارد و چون دیگر لازم نیست گراف، همبند باشد، می‌توانیم از فرض استقرا استفاده کنیم. بنابراین، در گراف باقی‌مانده می‌توان برای هر رأس با درجه‌ی فرد، یک رأس فرد دیگر در «مسیرهای با یال‌های دوبه‌دو جداازهم» یافت. سپس می‌توانیم مسیر حذف‌شده را سرجایش برگردانیم و اثبات را کامل کنیم.

در واقع قضیه‌ای را ثابت کردیم، قوی‌تر از آنچه در جست‌وجویش بودیم! ما ثابت کردیم که همبندی شرطی غیرضروری است و اثبات قضیه آسان‌تر هم شد.

□

توجه: این برهان نمونه‌ای از روشی بسیار قدرتمند است که ما آن را «تقویت فرض استقرا» می‌نامیم. این روش از برخی جهات مانند روشی است که در بخش پیش به کار بردیم. ترفند اصلی، تغییر فرض استقرا برای تطبیق با نیازهاست. پس اثبات، ممکن است حتا با قوی‌تر کردن قضیه، آسان‌تر هم شود که Polya این موضوع را «تضاد ابداعی» نامیده است (Polya [۱۹۵۴]). این تضاد آشکار از آنجا پیدا شده است که هرچند کوشیده‌ایم چیزهای بیش‌تری را ثابت کنیم، اما چیزهای بیش‌تری هم برای بنا نهادن اثبات در دست داشته‌ایم، چراکه فرض استقرا نیز قوی‌تر شده است. بازهم نمونه‌های بیش‌تری از شیوه‌ی تقویت فرض استقرا در کتاب خواهیم دید؛ این شیوه بسیار با اهمیت است.

۲-۱۱ رابطه‌ی بین میانگین‌های حسابی و هندسی

مثال بعد اثباتی زیباست برای رابطه‌ی بین میانگین‌های حسابی و هندسی که به Cauchy نسبت داده شده است. این اثبات، روشی دقیق و نامعمول در به‌کارگیری استقراست که بعداً نیز از آن سود خواهیم جست.

□ قضیه‌ی ۲-۱۳

اگر x_1، x_2، ... و x_n همگی اعدادی مثبت باشند، آنگاه

$$(x_1 x_2 \ldots x_n)^{\frac{1}{n}} \le \frac{x_1 + x_2 + \ldots + x_n}{n} \qquad (۲-۳)$$

برهان: اثبات با استقرا روی n انجام می‌شود. فرض استقرا همان رابطه‌ی (۲-۳) است. نکته‌ی جالب، این‌که اثبات رو به عقب پیش می‌رود. به جای اثباتِ یک حالت پایه و سپس گسترش فرض استقرا از مقادیر کوچک‌تر n به مقادیر بزرگ‌تر، از اصل استقرای معکوس استفاده می‌کنیم:

اگر گزاره‌ی P برای زیرمجموعه‌ای نامتناهی از اعداد طبیعی درست باشد و اگر از درستی‌اش برای n، درستی‌اش برای n-1 نتیجه شود؛ آنگاه P برای تمامی اعداد طبیعی درست است.

این اصل درست است، زیرا برقراری گزاره‌ی P برای مجموعه‌ای نامتناهی تضمین می‌کند که برای هر عدد طبیعی مانند k، عدد بزرگ‌تری مانند m نیز در این مجموعه وجود دارد، سپس می‌توان گام استقرای معکوس را برای حرکت رو به عقب، از m به k، به کار برد.

این قضیه را در دو مرحله ثابت می‌کنیم: در مرحله‌ی نخست، قضیه را با استقرای معمولی برای مقادیری از n که توانی از ۲ هستند، ثابت می‌کنیم. توان‌های ۲، همان مجموعه‌ی نامتناهی مورد نیاز برای اثبات است. در مرحله‌ی بعد، با استقرای معکوس، قضیه را برای همه‌ی nها ثابت می‌کنیم.

نخست، مقادیری از n را در نظر بگیرید که توانی از ۲ هستند. قضیه برای n=1 روشن است و برای n=2 نیز چنین می‌شود:

$$\sqrt{x_1 x_2} \le \frac{x_1 + x_2}{2}$$

که درستی آن به راحتی با به توان ۲ رساندن دو طرف ثابت می‌شود. فرض کنید رابطه‌ی (۲-۳) برای $n=2^k$ درست باشد و $2n=2^{k+1}$ را در نظر بگیرید. سمت چپ رابطه‌ی (۲-۳) را چنین می‌نویسیم:

$$(x_1 x_2 \ldots x_n)^{\frac{1}{2n}} = \sqrt{(x_1 x_2 \ldots x_n)^{\frac{1}{n}} (x_{n+1} x_{n+2} \ldots x_{2n})^{\frac{1}{n}}} \qquad (۴-۲)$$

حال می‌توانیم با فرض $y_1=(x_1x_2\ldots x_n)^{1/n}$ و $y_2=(x_{n+1}x_{n+2}\ldots x_{2n})^{1/n}$ قضیه را برای n=2 به کار گیریم. در این صورت رابطه‌ی (۲-۴) چنین می‌شود:

$$(x_1 x_2 \ldots x_{2n})^{\frac{1}{2n}} = \sqrt{y_1 y_2} \le \frac{y_1 + y_2}{2}$$

اما بنا به فرض داریم:

$$\frac{y_1 + y_2}{2} \le \frac{\frac{x_1 + x_2 + \ldots + x_n}{n} + \frac{x_{n+1} + x_{n+2} + \ldots + x_{2n}}{n}}{2}$$

و ادعای ما ثابت می‌شود.

اینک می‌توانیم با به‌کارگیری استقرای معکوس قضیه را برای هر عدد طبیعی ثابت کنیم. فرض کنید رابطه‌ی (۲-۳) برای یک n دل‌خواه برقرار است و n-1 را در نظر بگیرید. z را چنین تعریف می‌کنیم:

$$z = \frac{x_1 + x_2 + \ldots + x_{n-1}}{n-1}$$

قضیه برای هر عدد مثبت برقرار است. پس به طور خاص برای x_1، x_2، ...، x_{n-1} و z نیز برقرار است. پس:

$$(x_1 x_2 \ldots x_{n-1} z)^{\frac{1}{n}} \le \frac{x_1 + x_2 + \ldots + x_{n-1} + z}{n} = \frac{(n-1)z + z}{n} = z$$

(آگاهانه z را به گونه‌ای برگزیدیم که سمت راست عبارت را ساده کند.) در نتیجه داریم:

$$(x_1 x_2 \ldots x_{n-1} z)^{\frac{1}{n}} \le z$$

که از آن نتیجه می‌شود:

$$x_1 x_2 \ldots x_{n-1} z \le z^n$$

و در نهایت:

$$(x_1 x_2 ... x_{n-1})^{\frac{1}{n-1}} \le z = \frac{x_1 + x_2 + ... + x_{n-1}}{n-1}$$

که دقیقاً همان رابطه‌ی (۲-۳) برای n-1 است.

□

۲-۱۲ مثالی از قانون ثابت حلقه در تبدیل عددی ده‌دهی به عددی دودویی

استقرا برای اثبات درستی الگوریتم‌ها نیز بسیار مفید است. برنامه‌ای را در نظر بگیرید که شامل یک حلقه برای محاسبه‌ی مقداری خاص است. می‌خواهیم ثابت کنیم نتیجه‌ی اجرای حلقه، دقیقاً همان نتیجه‌ی مطلوب است. برای این کار از استقرا روی تعداد دفعات اجرای حلقه سود می‌جوییم. فرض استقرا باید روابط بین متغیرها را در هنگام اجرای حلقه نشان دهد. چنین فرضی در استقرا «قانون ثابت حلقه» نامیده می‌شود. روش به کار بردن «قانون ثابت حلقه» را با الگوریتم شکل ۲-۶ توضیح می‌دهیم. این الگوریتم عدد ده‌دهی n را به عددی دودویی واقع در آرایه‌ی b تبدیل می‌کند. (در آغاز این آرایه خالی است.)

الگوریتم Convert_to_Binary شامل حلقه‌ای با سه دستور است. دستور نخست، k را افزایش می‌دهد. k اندیسی برای آرایه‌ی b است. دستور دوم t mod 2 را حساب می‌کند که باقی‌مانده‌ی تقسیم t بر ۲ است (یعنی حاصل آن، هنگام فرد بودن t، یک و در غیر این صورت صفر است). دستور سوم، t را با بهره‌گیری از تقسیم صحیح، بر ۲ تقسیم می‌کند (یعنی عمل تقسیم را با نادیده گرفتن بخش اعشاری انجام می‌دهد).

الگوریتم: Convert_to_Binary

ورودی: n (یک عدد صحیح مثبت)

خروجی: b (آرایه‌ای از بیت‌ها که بیانگر معادل دودویی n است.)

```
begin
        t := n;     {از متغیر t برای نگه‌داری مقدار n استفاده می‌کنیم.}
        k := 0;
        while t>0 do
                k := k+1;
                b[k] := t mod 2;
                t := t div 2;
end
```

شکل ۲-۶ الگوریتم Convert_to_Binary

□ قضیه‌ی ۲-۱۴

زمانی که الگوریتم Convert_to_Binary به پایان برسد، معادل دودویی n در آرایه‌ی b قرار خواهد داشت.

برهان: اثبات با استقرا روی k، یعنی تعداد دفعات اجرای حلقه انجام می‌شود. لازم نیست فرض استقرا عیناً گزاره‌ی قضیه باشد، بلکه می‌تواند تنها در بخشی از الگوریتم مطرح شود. در این مورد، بخش اصلی الگوریتم یک حلقه است و ما از فرض استقرا برای بررسی الگوی اجرای این حلقه استفاده می‌کنیم. در اینجا می‌توان به فرض استقرا به عنوان یک «قانون ثابت» نگاه کرد. در واقع فرض استقرا گزاره‌ای است درباره‌ی متغیرها بدون توجه به تعداد دفعات اجرای حلقه. مشکل‌ترین بخش کار، یافتن فرضی درست برای استقراست. این فرض را در نظر بگیرید:

فرض استقرا: در آرایه‌ی b[1..k] تعدادی ۰ و ۱ قرار دارند که اگر آن‌ها را کنار هم قرار دهیم، نشانگر عددی در مبنای ۲ می‌شود. اگر این عدد را m بنامیم، آنگاه $n=t.2^k+m$.

عبارت $t.2^k+m$ قلب قانون ثابت حلقه و حتا الگوریتم است. فرض استقرا می‌گوید مقدار این عبارت مستقل از تعداد دفعات اجرای حلقه است. این فرض، ایده‌ی پس‌زمینه‌ی الگوریتم را نیز در خود دارد. در تکرار kام حلقه، آرایه‌ی دو دویی، k تا از کم‌ارزش‌ترین بیت‌های n را در خود دارد و مقدار t نیز پس از k بار جابه‌جایی به سمت چپ، معادل بیت‌های دیگر n خواهدشد.

برای اثبات درستی این الگوریتم باید سه شرط را ثابت کنیم: (۱) فرض استقرا در آغاز حلقه، درست است. (۲) از درستی فرض استقرا در تکرار kام، درستی آن در تکرار k+1ام نتیجه می‌شود. (۳) هنگام پایان الگوریتم، فرض استقرا، درستی الگوریتم را نشان می‌دهد. در آغاز حلقه k=0 و m=0 (چون آرایه خالی است) و n=t. فرض کنید در آغاز تکرار kام حلقه داشته باشیم: $n=t.2^k+m$ و مقادیر متناظر را در پایان تکرار kام در نظر بگیرید. دو حالت ممکن است: یکی این‌که t در آغاز تکرار kام حلقه، زوج باشد. در این حالت t mod 2، صفر است. پس چیزی وارد آرایه نمی‌شود (یعنی m تغییر نمی‌کند) t نیز بر ۲ تقسیم می‌شود و k هم یک واحد افزایش می‌یابد. در نتیجه، فرض استقرا هنوزهم درست باقی می‌ماند. در حالت دوم t فرد است. در این حالت، مقدار b[k+1]، ۱ می‌شود که 2^k را به m می‌افزاید، مقدار t نیز به (t-1)/2 تغییر می‌یابد و به k هم یک واحد افزوده می‌شود. پس، در پایان تکرار kام حلقه، عبارت متناظر چنین خواهد بود: $n=t.2^k+m+2^k=(t-1).2^k+m+2^k=((t-1)/2).2^{k+1}+m$ که دقیقاً همان چیزی است که می‌خواستیم ثابت کنیم. سرانجام، حلقه، زمانی پایان خواهد یافت که t=0، پس بنا بر فرض استقرا: $n=0.2^k+m=m$.

□

۲-۱۳ لغزش‌های رایج در استقرا

در پایان این فصل، چند مثال هشداردهنده از دام‌هایی می‌آوریم که با به‌کارگیری شتاب‌زده‌ی استقرا ممکن است در آن‌ها گرفتار شوید. بسیاری از اثبات‌های نادرست نتیجه‌ی اطمینان بیش از حد است. چنان‌چه کسی قضیه‌ای را قویاً باور داشته باشد، همه چیز را درباره‌ی آن روشن، آشکار و بدیهی می‌بیند. این پدیده در اثبات‌های استقرایی، بیش‌تر چنین است: گاهی از آنجا که قضیه روشن است، به طور ضمنی چندین واقعیت آشکار دیگر را به فرض می‌افزاییم و اثبات از مرحله‌ی n به مرحله‌ی n+1 را با در نظر گرفتن این فرض‌ها انجام می‌دهیم. پس، با آن‌که فرض استقرا قوی‌تر شده است، اما فرض‌های قوی‌تر آن را ثابت نکرده‌ایم. برای مثال، ممکن است فردی گراف‌های قضیه‌ای را همبند فرض کند، اما همبندی گراف‌های کاهش‌یافته را بررسی نکند (بخش ۲-۱۰ را ببینید - مترجمان). هرچند چنین حذفی خیلی ظریف است، ولی ممکن است به اثباتی بسیار اشتباه منتهی گردد. بنابراین، بیان دقیق فرض استقرا بسیار مهم است.

اشتباه معمول دیگر چنین است: بخش اصلی یک اثبات استقرایی نشان دادن درستی قضیه برای n+1، از روی درستی آن برای n است. به این منظور ممکن است دو کار انجام دهیم: یا از نمونه‌ی n+1 آغاز کنیم و نشان دهیم درستی‌اش از درستی نمونه‌ی n به دست می‌آید و یا از نمونه‌ی n شروع و ثابت کنیم از درستی آن، درستی نمونه‌ی n+1 نتیجه می‌شود؛ هر دو رویکرد معتبرند. در هر دو حالت، نمونه‌ی n+1 باید نمونه‌ای دل‌خواه باشد! اشتباه است اگر از نمونه‌ی n شروع کنیم و آن را در حالی به نمونه‌ی n+1 گسترش دهیم که نمونه‌ی n+1 دارای برخی ویژگی‌های خاص باشد. برای نمونه به این اثبات اشتباه از قضیه‌ی ۲-۸ توجه کنید: با نقشه‌ای دل‌خواه، دارای n وجه کار را آغاز می‌کنیم و بنا به فرض استقرا، V+n را برابر E+2 می‌گیریم. وجهی دل‌خواه را در نظر می‌گیریم و یک یال با دو رأس تازه به آن می‌افزاییم، به گونه‌ای که آن وجه را به دو وجه تبدیل کند. افزودن دو رأس تازه (که هر کدام یک یال قدیمی را قطع کرده، به دو یال تبدیل می‌کند) موجب اضافه شدن دو یال می‌گردد. پس با افزودن یک وجه جدید، سه یال و دو رأس دیگر اضافه می‌شود؛ یعنی: V+2+n+1=E+3+2 که ادعای قضیه برای n+1 وجه نیز ثابت می‌شود. این اثبات نامعتبر است، زیرا افزایش یال، در حالتی خاص انجام شد. می‌توان به نقشه هم بین رأس‌های موجود و هم بین یک رأس تازه و یک رأس موجود، یال افزود. در واقع، گراف‌هایی که با این روش به دست می‌آیند، تنها رأس‌هایی از درجه‌ی ۳ یا کم‌تر خواهند داشت. پس، بی‌شک گراف‌هایی بسیار خاص هستند. در حالت کلی مطمئن‌تر این است که کار را با نمونه‌ای دل‌خواه آغاز کنیم و بکوشیم به جای هر کار دیگر تنها با به‌کارگیری فرض استقرا، قضیه را ثابت کنیم.

دام خطرناک دیگری که سر راه اثبات‌های استقرایی قرار می‌گیرد، استفاده از حالت‌های استثنایی قضیه است. حالت‌های استثنایی کوچک معمولاً به صورت‌هایی مانند $n \geq 3$ و یا «n عدد اولی کوچک‌تر

از ۳۰ نیست» رخ می‌دهند. اصل استقرا به قابلیت اثبات درستی فرض برای n=2 از n=1 و برای n=3 از n=2 و ... بستگی دارد؛ حتا اگر یکی از این گام‌ها نادرست باشد، کل اثبات نادرست خواهد شد. دو مثال از این مورد ارائه می‌کنیم: اولی، ساده و خنده‌دار، اما دومی، جدی‌تر است.

ادعایی خنده‌دار: n خط در صفحه مفروضند که هیچ زوجی از آن‌ها با هم موازی نیستند.

با این فرض‌ها همه‌ی این خط‌ها باید یک نقطه‌ی مشترک داشته باشند.

روشن است که این ادعا نادرست است، اما بیایید نگاهی به اثباتش بیندازیم. ادعای گفته‌شده برای یک خط درست است. بهتر است کمی دقت کنیم و ادعا را برای دو خط در نظر بگیریم که بازهم درست است. فرض می‌کنیم این ادعا برای n خط درست باشد. حال، آن را برای n+1 خط در نظر می‌گیریم. بنا به فرض استقرا، n خط نخست از این n+1 خط، یک نقطه‌ی مشترک دارند و باز هم بنا به فرض استقرا n خط آخر (که خط n+1ام هم جزو آن‌هاست) نیز یک نقطه‌ی مشترک دارند. نقطه‌ی مشترک «n خط نخست» و «n خط آخر» باید بین n+1 خط مشترک باشد؛ زیرا خط‌هایی که در دو نقطه‌ی متفاوت یکدیگر را قطع می‌کنند، یکسان هستند. در نتیجه خط n+1ام نیز از همان نقطه‌ی مشترک n خط می‌گذرد و ادعا ثابت می‌شود.

اشتباه این اثبات چیست؟ در واقع اشتباه این اثبات بسیار ناچیز است. تنها گامی که در این اثبات ناآگاهانه (در اینجا بسیار آگاهانه!) برداشته شد، چشم‌پوشی از این واقعیت است که n باید دست‌کم ۳ باشد تا استدلال، درست از کار درآید. در این مورد باید چنین استدلال می‌کردیم: این ادعا برای n=1 و n=2 درست است و اگر برای n=3, 4, ... درست باشد، آنگاه برای n+1=4, 5, ... نیز درست خواهد بود. تنها مشکل این استدلال گامی است که از n=2 به n=3 برداشته می‌شود. همین اشتباه ناچیز کافی است تا کل اثبات را بسیار نادرست کند. ممکن است خیال کنید این مثال آن قدر روشن است که کسی چنین اشتباهی نمی‌کند. بیایید نمونه‌ی دیگری را بررسی کنیم که آن قدرها هم روشن نیست.

این ادعا را در نظر بگیرید (عبارت تا بی‌نهایت ادامه دارد):

$$n = \sqrt{1 + (n-1)\sqrt{1 + n\sqrt{1 + (n+1)\sqrt{1 + (n+2)\ldots}}}} \qquad (۵-۲)$$

برای این رابطه، اثباتی بنا بر استقرا ارائه می‌کنیم: نخست باید نشان دهیم این عبارت برای تمام مقادیر n همگراست تا ادعای بالا معنا داشته باشد. از اثبات همگرایی می‌گذریم. (مطمئن باشید این عبارت همگراست!) رابطه‌ی (۲-۵) برای n=1 چنین خواهد شد: $\sqrt{1 + 0(\ldots)} = 1$ که درست است (چون عبارت درون پرانتز همگراست). فرض کنید رابطه‌ی (۲-۵) برای n برقرار باشد و آن را برای n+1 در نظر بگیرید. اگر دو سمت رابطه‌ی (۲-۵) را به توان ۲ برسانیم، داریم:

$$n^2 = 1 + (n-1)\sqrt{1 + n\sqrt{1 + (n+1)\sqrt{1 + (n+2)\ldots}}}$$

پس از مرتب کردن جملات داریم:

$$\frac{n^2-1}{n-1} = n+1 = \sqrt{1+n\sqrt{1+(n+1)\sqrt{1+(n+2)...}}}$$

که دقیقاً همان رابطه‌ی (۲-۵) برای n+1 است. آیا اثبات کامل شده است؟ تنها گام اشتباه، انجام تقسیم بر n-1 است، بی‌آن‌که مطمئن شویم n-1 صفر نیست. اگر n=1 آنگاه n-1=0، یعنی در گام نخست استقرا، بر صفر تقسیم کرده‌ایم! پس همه چیز درست است؛ مگر برای n=1 - یعنی گامی که باید از آن به n=2 برسیم - و همین لغزش برای نادرست شدن کل اثبات کافی است. در این مورد، ادعا درست است، اما اثباتش چندان ساده نیست.

۲-۱۴ خلاصه

استقرای ریاضی، روشی نیرومند و پرکاربرد است. در این فصل انواع گوناگونی از آن را دیدیم و برخی شیوه‌های کاربردش را بررسی کردیم. گام نخست، تعریف فرض استقراست. ما باید تصمیم بگیریم که استقرا روی کدام پارامتر بنا شود. در بیش‌تر موارد تنها یک پارامتر وجود دارد و انتخاب، روشن است. گاهی هم برای انتخاب پارامتر تا حدی دستمان باز است. گاهی هم ممکن است برای اثبات استقرایی پارامتر تازه‌ای تعریف کنیم. همان‌گونه که دیدیم فرض استقرا همیشه به طور مستقیم از گزاره‌ی قضیه پیروی نمی‌کند. گاهی استقرا در چند مرحله به کار گرفته می‌شود که هر یک از مراحل، ما را به اثبات پایانی نزدیک‌تر می‌کند. گاهی نیز در استقرا فرض را تقویت می‌کنیم تا قضیه‌ای قوی‌تر، از آن نتیجه شود.

در هر اثبات استقرایی دو بخش وجود دارد: حالت پایه و گام استقرا. معمولاً و نه همیشه، حالت پایه آسان است و به همین دلیل برخی آن را نادیده می‌گیرند. گام استقرا، قلب هر اثبات استقرایی است و روش‌های فراوانی برای اثبات آن وجود دارد. رایج‌ترین روش اثبات گام استقرا، رفتن به n از n-1 است. از n+1 نیز می‌توان به n رفت. استقرای قوی، ادعایی را برای n، به یک یا چند ادعا برای مقادیری کوچک‌تر از n کاهش می‌دهد (و البته هیچ ضرورتی ندارد که n-1 هم جزو این مقدارهای کوچک‌تر باشد). از دیگر انواع استقرا رفتن از 2n به n و استقرای معکوس است. در استقرای معکوس، از ادعا برای n+1، همان ادعا را برای n نتیجه می‌گیریم و حالت پایه شامل مجموعه‌ای نامتناهی است که ادعا برای آن‌ها ثابت شده است. نکته‌ی اصلی گام استقرا در روش‌های گوناگون این است که باید گزاره‌ی ادعا را بدون کوچک‌ترین تغییری ثابت کند. به ادعا برای مقادیر کوچک‌تر نمی‌توان هیچ فرضی را افزود، مگر آن‌که به طور مشخص در فرض استقرا آمده باشد.

به گام کاهش می‌توان به عنوان گام گسترش نیز نگریست. در این صورت، ادعا را از مقدارهای کوچک‌تر پارامتر، به مقادیر بزرگ‌تر آن گسترش می‌دهیم. باید مطمئن شویم که عمل گسترش، تمام مقادیر ممکن را برای پارامتر در بر می‌گیرد و نیز باید مطمئن شویم که ادعای گسترش‌یافته، ادعایی

کلی برای قضیه است؛ بی‌آن‌که فرض‌ها یا محدودیت‌های دیگری به آن بیفزاید. در فصل ۵ شباهتی مستقیم بین انواع گوناگون استقرا در این فصل و روش‌های مختلف طراحی الگوریتم خواهیم دید.

مراجعی برای مطالعه‌ی بیش‌تر

کشف استقرای ریاضی به ریاضی‌دان ایتالیایی، Franciscus Maurolycus (متولد ۱۴۹۴ میلادی) نسبت داده شده است. تاریخچه‌ی استقرای ریاضی در Bussey [۱۹۱۷] آمده است. (Vacca [۱۹۰۹] را نیز ببینید.) جالب است بدانیم از اصلی شبیه استقرای ریاضی برای تفسیر تلمود (یا تلموذ) در قرن دوازدهم میلادی استفاده شده است. (J. Gillis این موضوع را بررسی کرده است.) آن زمان، مشکل، تفسیر دستوری بود که زمان کاری را «سه روز پیش از یک روز تعطیل» تعیین می‌کرد. (در زمان نگارش تلمود عجیب نبود اگر کسی می‌گفت «x روز پیش از یک روز تعطیل» و خود روز تعطیل را هم جزو آن x روز قلمداد می‌کرد.) پرسش این بود که آیا باید خود روز تعطیل هم جزو آن سه روز در نظر گرفته شود یا خیر. تفسیر، چنین شد که آن سه روز، خود روز تعطیل را شامل نمی‌شوند، برای این‌که در آن صورت ابهام به وجود می‌آید. استدلالی استقرایی، منتهی به این نتیجه شد. حالت پایه ۱ روز بود. معنایی ندارد که بگوییم «۱ روز پیش از یک روز تعطیل» و منظورمان خود روز تعطیل باشد. بنابراین «۱ روز پیش از یک روز تعطیل» خود روز تعطیل را در بر نمی‌گیرد. پس «۲ روز پیش از یک روز تعطیل» باید خود روز تعطیل را مستثنا کند، زیرا در غیر این صورت با «۱ روز پیش از یک روز تعطیل» یکسان خواهد شد. بنابراین «۳ روز پیش از یک روز تعطیل» خود روز تعطیل را شامل نمی‌شود. روشن است که این استدلال، استقرایی است.

مسأله‌ی حاصل‌جمعی که در بخش ۲-۵ ارائه شده است، از Polya [۱۹۵۷] است. Lakatos [۱۹۷۶] درباره ی تعمیم رابطه‌ی اویلر به اشیای سه بعدی، مبحثی درخشان عرضه کرده است. سفارش می‌کنیم حتماً آن را بخوانید. مثال بخش ۲-۸ از Lovász [۱۹۷۹] است. Gray [۱۹۵۳] مبتکر کدهای Gray است. مطالب بیش‌تری درباره نظریه‌ی کدگذاری در Hamming [۱۹۸۶] نیز وجود دارد. اثبات رابطه‌ی میانگین‌های حسابی و هندسی از Cauchy است. (می‌توانید Polya و Szego [۱۹۷۲] یا Beckenbach و Bellman [۱۹۶۱] را نیز ببینید.) کتاب‌شناسی نظریه‌ی گراف در فصل ۷ آمده است. مطالب بیش‌تری درباره‌ی قانون ثابت حلقه در Gries [۱۹۸۱] یافت می‌شود. همچنین اثبات (۲-۵) را از Darrah Chavey گرفته‌ایم.

مطالب بیش‌تری درباره‌ی استقرای ریاضی در کتاب‌های درخشان Polya [۱۹۵۴؛ ۱۹۵۷؛ ۱۹۸۱] وجود دارد. مثال‌های بیش‌تری درباره‌ی استقرا در Sominskii [۱۹۶۳]، Golovina و Yaglom [۱۹۶۳] و البته در همین کتاب نیز یافت می‌شود.

تمرین‌ها

۱-۲ ثابت کنید برای هر سه عدد طبیعی x، y و n که $x \neq y$، x^n-y^n بر x-y بخش‌پذیر است.

۲-۲ راه حل ارائه‌شده در بخش ۲-۲ را برای جمع حسابی، تعمیم دهید؛ یعنی جمع $a_1+a_2+\ldots+a_n$ را بیابید که در آن $a_n=c_1n+c_2$ (c_1 و c_2 اعدادی ثابت هستند).

۳-۲ حاصل $1.2+2.3+\ldots+n(n+1)$ چیست؟ درستی ادعای خود را ثابت کنید.

۴-۲ حاصل $\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots+\frac{1}{2^n}$ چیست؟ درستی ادعای خود را ثابت کنید.

۵-۲ جمع مربع‌های نخستین n عدد طبیعی را بیابید و درستی ادعای خود را ثابت کنید.

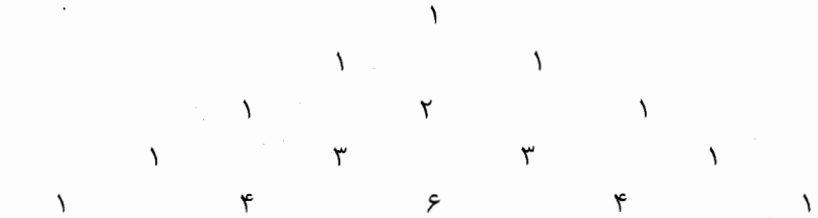
۶-۲ ثابت کنید: $1^2-2^2+3^2-4^2+\ldots+(-1)^{k-1}.k^2=(-1)^{k-1}.k(k+1)/2$

۷-۲ مجموعه‌ای از n+1 عدد طبیعی داریم که هر عضو آن از نخستین 2n عدد طبیعی، یعنی 1، 2، ... و 2n برگزیده شده است. ثابت کنید در این مجموعه دست‌کم دو عدد وجود دارد که یکی از آن‌ها دیگری را می‌شمارد (یا به عبارت دیگر، یکی از آن‌ها بر دیگری بخش‌پذیر است).

۸-۲ اگر a، b و n اعداد صحیح مثبتی باشند، ثابت کنید: $2^{n-1}(a^n+b^n) \geq (a+b)^n$

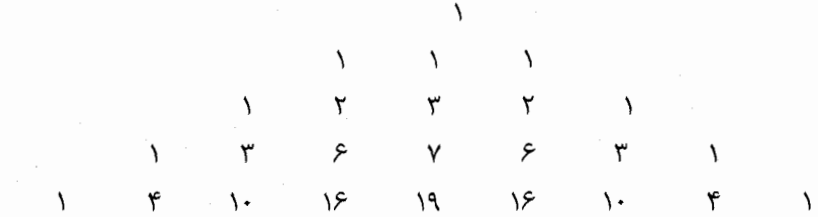
۹-۲ با استقرا ثابت کنید عددی که به صورت دهدهی نمایش داده شده است، بر ۳ بخش‌پذیر است، اگر و تنها اگر جمع ارقامش بر ۳ بخش‌پذیر باشد.

۱۰-۲



عبارتی برای مجموع اعداد سطر iام مثلث داده‌شده - که مثلث پاسکال نام دارد - بیابید و درستی ادعایتان را ثابت کنید. در این مثلث اعداد کناری ۱ و اعداد میانی مجموع دو عدد بالای خود هستند. (خیام، این مثلث و رابطه‌ی آن با ضرایب $(a+b)^n$ را بسیار پیش‌تر از پاسکال پیدا کرده بود - مترجمان)

۱۱-۲



عبارتی برای مجموع اعداد سطر iام مثلث داده‌شده بیابید و درستی ادعایتان را ثابت کنید. هر عدد این مثلث از جمع سه عدد بالای آن به دست می‌آید. (اگر عددی وجود نداشته باشد، به جایش ۰ در نظر گرفته می‌شود.)

۲-۱۲ ثابت کنید برای هر $n>1$ داریم:

$$\frac{1}{n+1}+\frac{1}{n+2}+\ldots+\frac{1}{2n}>\frac{13}{24}$$

۲-۱۳ ✯ ثابت کنید برای هر $n>1$ داریم:

$$1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}=\frac{k}{m}$$

که k عددی فرد و m عددی زوج است.

۲-۱۴ دنباله‌ی ۱، ۲، ۳، ۴، ۵، ۱۰، ۲۰، ۴۰ و ... را در نظر بگیرید که با تصاعدی حسابی آغاز می‌شود و پس از ۵ جمله به تصاعدی هندسی تبدیل می‌گردد. ثابت کنید هر عدد صحیح مثبت را می‌توان به صورت جمع اعداد متمایزی از این دنباله نوشت.

۲-۱۵ دنباله‌ی ۱، ۲، ۳، ۶، ۱۲، ۲۴، ۵۴، ۸۴، ۱۱۴ و ... را در نظر بگیرید که با تصاعدی حسابی آغاز می‌شود و پس از ۳ جمله به تصاعدی هندسی تبدیل می‌گردد و باز پس از ۳ جمله به تصاعدی حسابی تبدیل می‌شود. آیا اثبات تمرین ۲-۱۴ برای این مسأله نیز به کار می‌آید؟ اگر چنین است، اشتباه اثبات را بیابید، زیرا برای مثال ۸۱ را نمی‌توان به صورت جمع اعداد متمایزی از این دنباله نوشت. نکته‌ی هوشمندانه‌ی اثبات تمرین ۲-۱۴ چیست؟

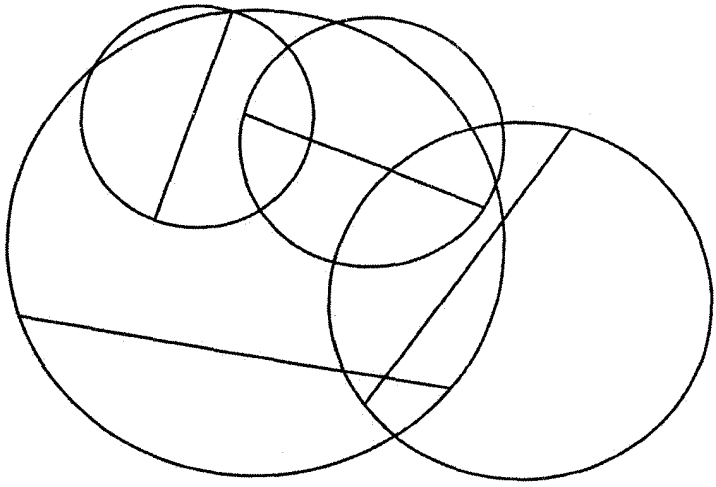
۲-۱۶ n خط را در صفحه و در وضعیت عمومی در نظر بگیرید به گونه‌ای که $n\geq 3$. ثابت کنید دست‌کم یکی از ناحیه‌های تشکیل‌شده از تقاطع این خطوط، مثلث است.

۲-۱۷ n خط را در صفحه و در وضعیت عمومی در نظر بگیرید به گونه‌ای که $n\geq 3$. ثابت کنید این خط‌ها دست‌کم n-2 مثلث تشکیل می‌دهند.

۲-۱۸ مجموعه‌ای از n نقطه در صفحه چنان مفروض است که هر سه تای آن‌ها درون یک دایره به شعاع واحد قرار دارند. ثابت کنید همه‌ی n نقطه درون دایره‌ای به شعاع واحد قرار دارند.

۲-۱۹ ثابت کنید ناحیه‌هایی را که از n دایره در صفحه تشکیل می‌شوند، می‌توان با دو رنگ چنان رنگ‌آمیزی کرد که هر دو ناحیه‌ی همسایه با رنگی مختلف رنگ‌آمیزی شده باشند.

۲-۲۰ n دایره را در نظر بگیرید که یک وتر از هر کدام رسم شده است (شکل ۲-۷). ثابت کنید ناحیه‌هایی را که از رسم این n دایره همراه با وترهایشان در صفحه تشکیل می‌شوند، می‌توان با سه رنگ چنان رنگ‌آمیزی کرد که هر دو ناحیه‌ی همسایه رنگ‌های متفاوتی داشته باشند.

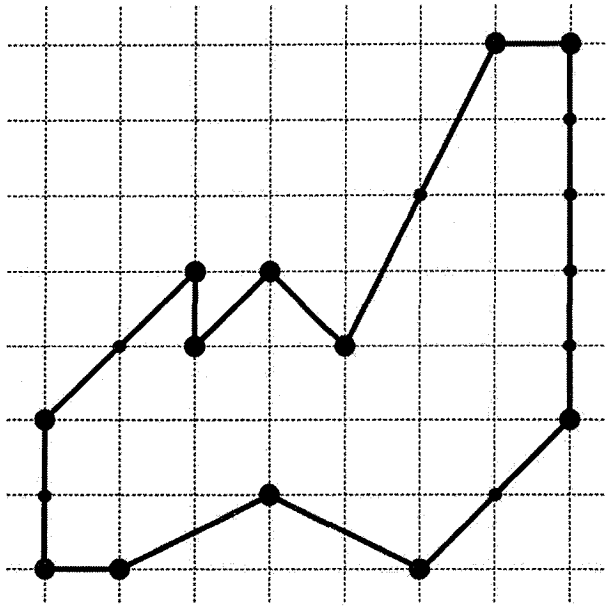


شکل ۲-۷ دایره‌هایی همراه با یک وتر از آن‌ها

۲-۲۱ اگر درجه‌ی همه‌ی رأس‌های ناحیه‌های تشکیل‌شده از یک نقشه‌ی رسم‌شده در صفحه‌ی معمولی زوج باشد؛ ثابت کنید این ناحیه‌ها را می‌توان با دو رنگ چنان رنگ‌آمیزی کرد که هیچ دو ناحیه‌ی همسایه‌ای، هم‌رنگ نباشند.

۲-۲۲ ثابت کنید یک نقشه‌ی رسم‌شده در صفحه‌ی معمولی را می‌توان با دو رنگ، چنان رنگ‌آمیزی کرد که هر دو ناحیه‌ی همسایه رنگ‌های مختلفی داشته باشند، اگر و تنها اگر تعداد همسایه‌های هر ناحیه، زوج باشد. (دو ناحیه، همسایه‌اند اگر دست‌کم یک یال مشترک داشته باشند.)

۲-۲۳ ✯ نقاط توری منظم (یا شبکه) در صفحه، نقاطی هستند که مختصات صحیح دارند. فرض کنید P یک چندضلعی است که خودش را قطع نمی‌کند (این نوع چندضلعی را ساده می‌گویند) و همه‌ی رأس‌های آن، نقاط توری منظم هستند. (شکل ۲-۸ را ببینید.) p را تعداد نقاطی از توری منظم بگیرید که روی محیط چندضلعی قرار دارند (رأس‌ها را نیز شامل می‌شود)؛ q را نیز تعداد نقاطی از توری منظم بگیرید که درون چندضلعی قرار دارند. ثابت کنید که مساحت چندضلعی برابر است با: $p/2+q-1$.



شکل ۲-۸ یک چند ضلعی ساده روی نقاط توری منظم

۲-۲۴ کدهای Gray را چنان تعریف کرده‌ایم که اختلاف بین دو رشته‌ی متوالی آن کمینه شود. حال، کدهای ضد Gray را به گونه‌ای تعریف می‌کنیم که اختلاف بین دو رشته‌ی متوالی آن بیشینه شود. آیا می‌توان یک شیوه‌ی کدگذاری برای هر تعداد زوج از اشیاء طراحی کرد که دو رشته‌ی متوالی، در k بیت اختلاف داشته باشند؟ (k تعداد بیت‌های هر رشته است.) در مورد k-1 بیت (یا k-2، یا k-3 و ...) چه می‌توان گفت؟ اگر این کار ممکن است، شیوه‌ای کارآمد برای انجام آن پیدا کنید.

۲-۲۵ درخت T و k زیردرخت از آن مفروضند، چنان‌که هر دو زیردرخت، دست‌کم یک رأس مشترک دارند. ثابت کنید دست‌کم یک رأس در تمام زیردرخت‌ها مشترک است.

۲-۲۶ d_1، d_2، ... و d_n را اعداد صحیح مثبتی بگیرید که $n\geq 2$. ثابت کنید اگر $d_1+d_2+\ldots+d_n=2n-2$ آنگاه درختی با n رأس وجود دارد که درجه‌ی رأس‌هایش دقیقاً d_1، d_2، ... و d_n باشد.

۲-۲۷ ✯ n نقطه را روی محیط یک دایره در نظر می‌گیریم و هر یک از آن‌ها را با خط مستقیمی به بقیه وصل می‌کنیم. فرض کنید هیچ یک از این سه خط یکدیگر را در نقطه‌ی مشترکی قطع نکنند (همرس نباشند). تعداد ناحیه‌های درون دایره را یافته، ادعایتان را ثابت کنید.

۲-۲۸ ✯ T=(V,E) را درختی بدون‌جهت در نظر بگیرید. f را هم تابعی بگیرید که رأس‌های این درخت را به یکدیگر نسبت می‌دهد (یا نگاشت می‌کند) به گونه‌ای که اگر (v,w) یالی در E باشد، آنگاه یا (f(v),f(w)) نیز یالی در E باشد یا f(v)=f(w). به عبارت دیگر، این تابع یا یک یال را به یالی دیگر می‌نگارد و یا این‌که هر دو رأس آن را به یک رأس نگاشت می‌کند. ثابت

کنید یا رأسی مانند v در V وجود دارد که f(v)=v و یا یالی مانند (v,w) در E هست که f(v)=w و f(w)=v. (به عبارت دیگر، یا یک رأس و یا یک یال وجود دارد که این تابع آن را به خودش نسبت می‌دهد.)

۲-۲۹ ساده‌ترین نوع اصل لانه‌ی کبوتر چنین است: اگر n+1 توپ (n≥1) درون n جعبه قرار داشته باشند، آنگاه دست‌کم یک جعبه، بیش از یک توپ دارد. این اصل را با استقرا ثابت کنید.

۲-۳۰ درخت دودویی کامل، به کمک استقرا چنین تعریف شده است: درخت دو دویی کامل به ارتفاع ۰، از یک گره تشکیل شده است که این گره همان ریشه است. درخت دودویی کامل به ارتفاع h+1 از دو درخت دودویی کامل به ارتفاع h تشکیل شده است که ریشه‌های این دو درخت به یک ریشه‌ی تازه متصل گردیده‌اند. فرض کنید T یک درخت دودویی کامل به ارتفاع h باشد. ارتفاع یک گره در T، h-d تعریف می‌شود که در آن d فاصله‌ی گره از ریشه است (برای مثال، ارتفاع ریشه، h و ارتفاع یک برگ، ۰ است). ثابت کنید جمع ارتفاع همه‌ی گره‌های T برابر است با: 2^{h+1}-h-2.

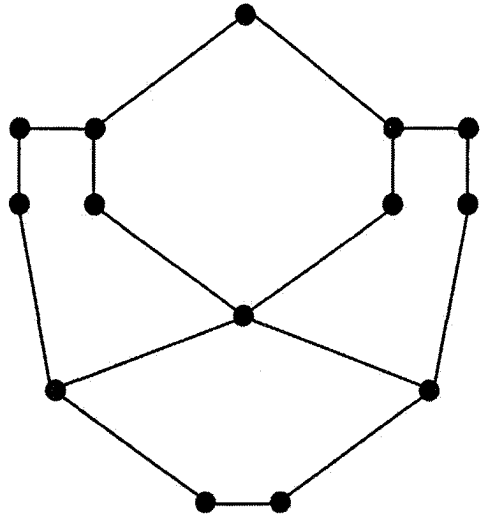
۲-۳۱ فرض کنید F(n)، nامین عدد فیبوناچی باشد که به صورت استقرایی چنین تعریف می‌شود: F(1)=F(2)=1 و برای n>2، F(n)=F(n-1)+F(n-2). ثابت کنید: $(F(n))^2+(F(n+1))^2=F(2n+1)$. (راه‌نمایی: در اینجا، فرض استقرا همانند کدهای Gray، با اثبات هم‌زمانِ دو قضیه‌ی ظاهراً مجزا قابل تقویت است.)

۲-۳۲ فرض کنید n و m اعداد صحیحی باشند که 1≤m≤n. به کمک استقرا ثابت کنید:

$n^2-m(n+1)+2n+m^2\leq n^2+n$

(راه‌نمایی: استقرای دو طرفه را روی m به کار ببرید. دو حالت پایه‌ی m=1 و m=n را ثابت کنید. سپس یا از حالت m=1 رو به جلو و یا از حالت m=n رو به عقب حرکت کنید.)

۲-۳۳ در یک گراف بدون‌جهت، یالی که حذفش گراف را ناهمبند کند، پل نام دارد. اگر G=(V,E) گرافی همبند، بدون‌جهت و بدون‌پل باشد؛ ثابت کنید G، صورت «تجزیه‌ی خوشه‌ای» دارد (شکل ۲-۹)؛ یعنی یال‌های G را می‌توان به مجموعه‌های جدااز هم E_1، E_2، ... و E_k تقسیم کرد، به گونه‌ای که E_1 یک دور باشد و برای هر i که 1<i≤k، E_i مسیری باشد که نقاط پایانی‌اش، رأس‌هایی باشند که در یک E_j پیشین ظاهر شده‌اند (j<i) و دیگر رأس‌های آن (اگر موجود باشند) در E_j های پیشین ظاهر نشده باشند. (این مسیر، می‌تواند مسیری بسته باشد که در این حالت تنها شامل یک رأس پیشین می‌شود.)



شکل ۲-۹ یک تجزیه‌ی خوشه‌ای

۲-۳۴ ✯ K_n را گرافی بدون‌جهت و کامل با n رأس در نظر بگیرید (یعنی هر دو رأس آن مستقیماً به یکدیگر متصل هستند) که n عددی زوج است. ثابت کنید یال‌های K_n را می‌توان دقیقاً به n/2 درخت پوشا تقسیم کرد. (درخت پوشا، زیرگرافی همبند است که همه‌ی رأس‌ها را در بر می‌گیرد و هیچ دوری ندارد.)

۲-۳۵ ✯ در گراف G=(V,E)، یک تطبیق، مجموعه‌ای از یال‌هاست که هیچ دوتایی از آن‌ها رأس مشترکی نداشته باشند. یک تطبیق کامل، تطابقی است که همه‌ی رأس‌های گراف را در بر گیرد. گراف G را با 2n رأس و n^2 یال به گونه‌ای بسازید که تنها یک تطبیق کامل داشته باشد.

۲-۳۶ a_1، a_2، ... و a_n را اعدادی حقیقی و مثبت بگیرید به گونه‌ای که $a_1a_2...a_n=1$. بدون استفاده از رابطه‌ی بین میانگین‌های حسابی و هندسی ثابت کنید: $(1+a_1)(1+a_2)...(1+a_n)\geq 2^n$
(راه‌نمایی: با معرفی متغیر دیگری که جای‌گزین دو مقدار ویژه از این دنباله می‌گردد، کاهش را انجام دهید.)

۲-۳۷ رابطه‌ی بازگشتی بین اعداد فیبوناچی را در نظر بگیرید: F(n)=F(n-1)+F(n-2). بدون حل این رابطه‌ی بازگشتی، F(n) را با G(n) مقایسه کنید که G(n) با این رابطه‌ی بازگشتی تعریف می‌شود: G(n)=G(n-1)+G(n-2)+1. چنین به نظر می‌رسد که همواره G(n)>F(n) (به دلیل ۱ اضافی در G(n)). حال به این اثبات به ظاهر معتبر (با استقرا) توجه کنید که ثابت می‌کند: G(n)=F(n)-1. به کمک استقرا فرض می‌کنیم برای هر k از بازه‌ی [1,n] G(k)=F(k)-1 و آنگاه G(n+1) را در نظر می‌گیریم:

G(n+1)=G(n)+G(n-1)+1=F(n)-1+F(n-1)-1+1=F(n+1)-1

اشتباه این اثبات چیست؟

۲-۳۸ به یک اثبات دیگر برای رابطه‌ی بین میانگین‌های حسابی و هندسی توجه کنید. ضعف عمده‌ای در این اثبات وجود دارد که آن را در حالت کلی، ناقص می‌کند. این ضعف را بیان کنید و محدودیت‌هایی روی قضیه اعمال کنید تا این اثبات درست شود.

فرض کنید $S=x_1+x_2+\ldots+x_n$. برای یافتن مثال نقض برای این قضیه، لازم است n عدد ارائه کنیم که جمعشان S باشد، ولی میانگین هندسی‌شان از S/n بزرگ‌تر شود. منطقی است اگر به دنبال مجموعه‌ای از اعداد بگردیم که جمعشان S باشد و حاصل‌ضربشان در بین چنین مجموعه‌هایی از اعداد بیشینه باشد. به عبارت دیگر، جمع (S) را ثابت می‌گیریم و می‌کوشیم حاصل‌ضرب را بیشینه سازیم. $\{x_1,x_2,\ldots,x_n\}$ را مجموعه‌ای بگیرید که حاصل‌ضرب را بیشینه می‌کند و حاصل‌جمع اعضای آن هم برابر S است. اگر $x_1\neq x_2$ می‌توانیم به جای x_1 و x_2، میانگین‌شان، یعنی $(x_1+x_2)/2$ را قرار دهیم. حاصل‌جمع تغییری نمی‌کند، اما حاصل‌ضرب بیش‌تر خواهد شد؛ زیرا:

$$x_1 x_2 \leq \left(\frac{x_1 + x_2}{2}\right)^2$$

این نامساوی تنها هنگامی به تساوی تبدیل می‌گردد که $x_1=x_2$. اگر همه‌ی این اعداد با هم مساوی باشند، قضیه برقرار است. در غیر این صورت، این مورد، یک مثال نقض برای فرض بیشینه بودن حاصل‌ضرب اعداد مجموعه است.

۲-۳۹ الگوریتمی برای تبدیل اعداد دودویی به اعداد ده‌دهی طراحی کنید. این الگوریتم باید برعکس الگوریتم Convert_to_Binary (شکل ۲-۶) باشد. ورودی الگوریتم، آرایه‌ای از بیت‌ها به نام b و به طول k است وخروجی الگوریتم عدد n خواهد بود. درستی الگوریتم را با به‌کارگیری قانون ثابت حلقه نشان دهید.

۲-۴۰ الگوریتم Convert_to_Binary (شکل ۲-۶) را چنان تغییر دهید که عددی در مبنای ۶ را به عددی دودویی تبدیل کند. ورودی الگوریتم، آرایه‌ای از ارقام در مبنای ۶ است و خروجی آن، آرایه‌ای از بیت‌هاست. درستی الگوریتم را با به‌کارگیری قانون ثابت حلقه نشان دهید.

فصل ۳

تحلیل الگوریتم‌ها

به بزرگی نیست وگرنه گاو از خرگوش جلو می‌زد.
ضرب‌المثل آلمانی‌های پنسیلوانیا

آن کس که به میوه‌های درختان سر به فلک کشیده می‌نگرد، اما بلندای آن درختان را اندازه نمی‌گیرد؛ احمقی بیش نیست.
Quintus Curtius Rufus

۳-۱ آشنایی

هدف از تحلیل یک الگوریتم پیش‌بینی رفتار آن، به ویژه زمان اجرای آن است؛ بدون آن که مجبور شویم آن الگوریتم را روی رایانه‌ای خاص پیاده‌سازی کنیم. فایده‌ی چنین کاری روشن است: داشتن معیارهای ساده‌ای برای کارایی یک الگوریتم بسیار راحت‌تر از آن است که مجبور باشیم با هر تغییری در پارامترهای یک رایانه، آن الگوریتم را از نو پیاده‌سازی کنیم تا کارایی‌اش آشکار شود. یک برنامه‌ی پیچیده معمولاً الگوریتم‌های کوچک فراوانی در خود دارد. بنابراین بررسی همه‌ی جای‌گزین‌ها برای این الگوریتم‌های کوچک، کاری بسیار وقت‌گیر و خسته‌کننده است.

متأسفانه پیش‌بینی دقیق رفتار یک الگوریتم معمولاً غیرممکن است، چون عوامل فراوانی بر آن تأثیر می‌گذارند. به جای این کار می‌کوشیم تا ویژگی‌های اصلی الگوریتم را به دست آوریم؛ یعنی پارامترها و معیارهای مشخصی را تعریف می‌کنیم که در تحلیل الگوریتم بیش‌ترین اهمیت را دارند. پس بیش‌ترِ جزئیات پیاده‌سازی را نادیده می‌گیریم. بنابراین تحلیل الگوریتم موضوعی تقریبی است و نباید آن را ابزار کاملی برای پیش‌بینی رفتار الگوریتم‌ها به شمار آورد؛ به عبارت دیگر، حتا تقریب نادقیق هم می‌تواند درباره‌ی الگوریتم، اطلاعات ارزشمندی به ما بدهد. مهم‌تر از همه، با این روش می‌توانیم الگوریتم‌های مختلف را با هم مقایسه کرده، دریابیم کدام یک از آن‌ها با اهدافمان سازگارتر است. این موضوع را می‌توان با ادعای مصرف سوخت در خودروها مقایسه کرد و گفت: «این معیارها تنها برای مقایسه‌اند و ممکن است زمان واقعی اجرای الگوریتم‌ها، کم‌تر یا بیش‌تر باشد».

در این فصل شیوه‌ای را برای برآورد زمان تقریبی اجرای الگوریتم‌ها و نیز برای مقایسه‌ی الگوریتم‌های مختلف با یکدیگر توضیح می‌دهیم. ویژگی اصلی این روش، نادیده گرفتن عوامل ثابت و تمرکز روی رفتار الگوریتم‌ها هنگام افزایش اندازه‌ی ورودی است. مثلاً اگر ورودی الگوریتم، آرایه‌ی به اندازه‌ی n باشد و الگوریتم از 100n مرحله تشکیل شده باشد، از ثابت ۱۰۰ چشم‌پوشی می‌کنیم و می‌گوییم زمان اجرای الگوریتم تقریباً n است، یا اگر تعداد مراحل الگوریتم $2n^2+50$ باشد، از ثابت‌های ۲ و ۵۰ چشم‌پوشی می‌کنیم و می‌گوییم زمان اجرای آن تقریباً n^2 است. (به زودی برای این کار، نمادگذاری دقیقی معرفی خواهیم کرد.) از آنجایی که n^2 از n بزرگ‌تر است، می‌گوییم الگوریتم دوم کندتر است؛ هرچند برای مثلاً n=5، الگوریتم نخست به ۵۰۰ مرحله و الگوریتم دوم به ۱۰۰ مرحله نیاز دارد. به هر حال، اگر n به اندازه‌ی کافی بزرگ باشد، چنین تقریبی، مناسب است. بی‌شک برای $n\geq 50$، الگوریتم دوم کندتر از الگوریتم نخست است. حال، فرض کنید زمان اجرای الگوریتم نخست $100n^{1.8}$ باشد. هنوز هم الگوریتم نخست بهتر است، چون $n^{1.8}$ از n^2 کوچک‌تر است؛ اما در این حالت، باید n تقریباً ۳۰۰,۰۰۰,۰۰۰ باشد تا $100n^{1.8}$ از $2n^2+50$ کوچک‌تر شود. خوشبختانه بیش‌تر الگوریتم‌ها، در بیان تقریبی زمان اجرا ثابت‌های کوچکی دارند. پس با این که گاهی ممکن است روش‌های مجانبی گمراه‌کننده باشند، اما در عمل کارآمد هستند. در بیش‌تر موارد، نگاهی به رفتار مجانبی برای برآورد کارایی، کافی است.

تحلیلی که روی الگوریتم انجام می‌دهیم باید نشان دهد که زمان اجرا برای یک ورودی مشخص چقدر پیش‌بینی می‌شود؛ اما قادر نخواهیم بود زمان اجرای تمام ورودی‌ها را فهرست کنیم، مگر آن که الگوریتم موردنظر واقعاً ساده باشد. ورودی، حالت‌های گوناگون بسیاری دارد و بیش‌تر الگوریتم‌ها در برابر ورودی‌های مختلف، رفتارهای متفاوتی از خود نشان می‌دهند. به همین دلیل، ورودی را با معیاری به نام اندازه‌ی ورودی می‌سنجیم و تحلیل را بر پایه‌ی آن انجام می‌دهیم. یک الگوریتم در برابر ورودی‌های هم‌اندازه رفتار دقیقاً یکسانی ندارد ولی ما امیدواریم که نوسانش، معقول و پذیرفتنی باشد. معمولاً اندازه‌ی ورودی چنین تعریف می‌شود: «فضای لازم برای ذخیره‌ی ورودی». تلاش نمی‌کنیم تا برای همه‌ی الگوریتم‌ها تعریفی کلی از اندازه‌ی ورودی ارائه دهیم، زیرا در اصل علاقه‌مندیم الگوریتم‌های مختلف یک مسأله را با هم مقایسه کنیم. بیش‌تر اوقات تعریف اندازه‌ی ورودی، کاری سرراست است که به زودی چند مثال از آن خواهیم دید. اندازه‌ی ورودی را با n نشان خواهیم داد، مگر آن که به صراحت چیز دیگری گفته باشیم.

در صورت داشتن یک مسأله و اندازه‌ی ورودی آن، به دنبال عبارتی هستیم که زمان اجرای الگوریتم را نشان دهد. (در بخش ۳-۳ تعریف دقیق زمان اجرا آمده است.) چنان که پیش‌تر نیز گفتیم، معمولاً زمان اجرای همه‌ی ورودی‌های هم‌اندازه یکسان نیست. در نتیجه از بین تمام ورودی‌های هم‌اندازه باید یکی را به عنوان شاخص برگزینیم. متداول‌ترین گزینه، انتخاب بدترین ورودی است.

ممکن است چنین گزینشی، عادی به نظر نرسد و بپرسیم: «چرا بهترین ورودی یا حالت میانگین برگزیده نشده است؟»

از آنجا که در بیش‌تر موارد، بهترین ورودی نماینده‌ی خوبی برای ورودی‌های الگوریتم نیست، پس معمولاً آن را برای تحلیل برنمی‌گزینیم. عموماً بهترین ورودی هر مسأله، آن را پیش‌پاافتاده کرده، مسأله را از حالت کلی خارج می‌سازد. هرچند، حالت میانگین ورودی می‌تواند گزینه‌ای مناسب باشد، اما تعیین کردن این حالت گاهی بسیار دشوار است. نخست این که در حالت کلی منظور از «میانگین ورودی» روشن و گویا نیست. ما می‌توانیم میانگین را بر مبنای پارامترهای گوناگون و به روش‌های مختلفی محاسبه کنیم. چنان‌چه در این کار دقت نکنیم، ممکن است میانگین به‌دست‌آمده شامل موارد فراوانی باشد که هرگز در عمل اتفاق نمی‌افتند. پس چنین میانگین‌هایی نامناسب هستند. دیگر مشکلی که «حالت میانگین برای ورودی الگوریتم‌ها» دارد، دشواری ریاضی تحلیل کارایی در این حالت است. ما هنوز به روش‌هایی جامع، با کاربرد آسان برای تحلیل حالت میانگین دست نیافته‌ایم. اگرچه برای شمار اندکی از مسأله‌ها تحلیل حالت میانگین را به کار خواهیم برد، ولی تحلیل اکثر الگوریتم‌ها را برای بدترین حالت آن‌ها انجام می‌دهیم. برگزیدن بدترین ورودی به عنوان شاخص، بسیار مناسب است. در برخی موارد، بدترین ورودی به حالت میانگین ورودی و مشاهدات تجربی بسیار نزدیک است. در دیگر موارد، حتا در آن‌هایی که بدترین ورودی با حالت میانگین بسیار متفاوت است؛ الگوریتمی که برای بدترین ورودی، کارایی بهتری از خود نشان می‌دهد، برای ورودی‌های دیگر نیز کار را بهتر انجام خواهد داد. در این کتاب بدترین حالت را تحلیل می‌کنیم، مگر آن که به صراحت چیز دیگری گفته باشیم.

خلاصه این که تحلیل مجانبی و تحلیل بدترین حالت، تنها برآوردهایی از زمان اجرای یک الگوریتم خاص برای یک ورودی با اندازه‌ی مشخص هستند. اگرچه این روش‌های تحلیل همه‌ی نیازها را برآورده نمی‌کنند، اما در بیش‌تر موارد کافی هستند.

۳-۲ نماد O

چنان‌که گفتیم، برای ارزیابی زمان اجرای یک الگوریتم مشخص، از عوامل ثابت چشم‌پوشی می‌کنیم. برای بهتر انجام دادن این کار به نمادگذاری ویژه‌ای نیاز داریم. می‌گوییم تابع $g(n)$ از $O(f(n))$ است، اگر ثابت‌های c و N وجود داشته باشند به گونه‌ای که برای $n \geq N$ داشته باشیم: $g(n) \leq cf(n)$. (O به صورت «اُ» یا گاهی به صورت «اُی بزرگ» تلفظ می‌شود.) به عبارت دیگر برای nهای به قدر کافی بزرگ، تابع $g(n)$ از چند برابر تابع $f(n)$ بزرگ‌تر نیست. (در اینجا «چند» ضریبی ثابت است.) ممکن است تابع $g(n)$ از $cf(n)$ کم‌تر و حتا خیلی کم‌تر باشد، اما نماد O آن را تنها از بالا محدود می‌کند. برای مثال $5n^2+15$ از $O(n^2)$ است زیرا برای $n \geq 4$ داریم: $5n^2+15 \leq 6n^2$. همچنین $5n^2+15$ از $O(n^3)$ است، چراکه برای $n \geq 6$ داریم: $5n^2+15 \leq n^3$.

نماد O به ما امکان می‌دهد به راحتی از ثابت‌ها صرف‌نظر کنیم. هرچند در نماد O می‌توان ثابت‌ها را نیز در نظر گرفت، اما هیچ دلیلی برای انجام این کار وجود ندارد و همیشه به جای عبارتی مانند O(5n+4) از O(n) استفاده می‌کنیم. به طور مشابه O(log n) را نیز بدون مبنای لگاریتم به کار می‌بریم، زیرا با تغییر مبنای لگاریتم، مقدار آن با ضریبی ثابت تغییر می‌کند. برای بیان مقدار ثابت O(1) را به کار می‌بریم. از نماد O در بخش‌هایی از یک عبارت نیز استفاده می‌کنیم تا وجود ثابت‌هایی در آن بخش‌ها روشن شود؛ مثلاً $T(n) = 3n^2 + O(n)$ و $S(n) = 2n\log_2^n + 5n + O(1)$.

در حالت کلی تشخیص این‌که تابع خاصی مانند g(n) از O(f(n)) هست یا نه، ممکن است چندان آسان نباشد. بیش‌تر تابع‌های به‌کاررفته در تحلیل الگوریتم‌های کتاب، نسبتاً ساده‌اند و با بهره‌گیری از چند قانون ساده می‌توانیم کار تحلیل بیش‌تر الگوریتم‌های کتاب (ولی نه همه‌ی آن‌ها) را انجام دهیم. مفیدترین این قانون‌ها قضیه‌ی زیر است: (توجه کنید که اگر از $n_1 \geq n_2$ نتیجه گرفته شود که $f(n_1) \geq f(n_2)$ ؛ تابع f(n) را صعودی گوییم.)

□ قضیه‌ی ۳-۱

برای هر دو ثابت c>0 و a>1 و برای هر تابع صعودی f(n): $(f(n))^c = O(a^{f(n)})$. به عبارت دیگر، یک تابع نمایی سریع‌تر از یک تابع چندجمله‌ای رشد می‌کند.

□

این قاعده را می‌توان برای مقایسه‌ی بسیاری از توابع به کار برد. برای مثال اگر در قضیه‌ی ۳-۱، n را به جای f(n) قرار دهیم؛ درمی‌یابیم که برای هر ثابت c>0 و هر ثابت a>1 :

$$n^c = O(a^n) \qquad (۱-۳)$$

مثال دیگر، جای‌گزینی $\log_a^n$ به جای f(n) است. برای هر ثابت c>0 و هر ثابت a>1 داریم:

$$(\log_a^n)^c = O(a^{\log_a^n}) = O(n) \qquad (۲-۳)$$

می‌توانیم بنا بر لم ۳-۲ روی نماد O جمع و ضرب هم انجام دهیم:

□ لم ۳-۲

۱- اگر $f(n)=O(s(n))$ و $g(n)=O(r(n))$ آنگاه $f(n)+g(n)=O(s(n)+r(n))$.

۲- اگر $f(n)=O(s(n))$ و $g(n)=O(r(n))$ آنگاه $f(n).g(n)=O(s(n).r(n))$.

برهان: بنا به تعریف نماد O، ثابت‌هایی مانند c_1، N_1، c_2 و N_2 وجود دارند به گونه‌ای که برای $n \geq N_1$: $f(n) \leq c_1 s(n)$ و برای $n \geq N_2$: $g(n) \leq c_2 r(n)$. با به کار بردن بزرگ‌ترین عدد بین c_1 و c_2 ($\max\{c_1,c_2\}$) و بزرگ‌ترین عدد بین N_1 و N_2 ($\max\{N_1,N_2\}$) می‌توان هر دو ادعای گفته‌شده را ثابت کرد.

□

از آنجا که نماد O بیانگر رابطه‌ی «≤» است، پس نمی‌توان تفریق یا تقسیم را برای آن به کار برد؛ یعنی در حالت کلی نمی‌توان از $f(n)=O(s(n))$ و $g(n)=O(r(n))$ نتیجه گرفت $f(n)-g(n)=O(s(n)-r(n))$ یا $f(n)/g(n)=O(s(n)/r(n))$. (تمرین‌های ۳-۱۵ و ۳-۱۶ را ببینید.)

اهمیت توجه به رفتار مجانبی در جدول ۳-۱ نشان داده شده است. این جدول به ازای چند زمان اجرای پارامتری، «زمان اجرای واقعی» را برای حل مسأله‌هایی با اندازه‌ی ورودی ۱۰۰۰، در رایانه‌هایی با سرعت‌های گوناگون در خود دارد. سرعت رایانه در این جدول از هر ستون به ستون بعدی دو برابر شده؛ یعنی از ۱۰۰۰ گام بر ثانیه در ستون نخست به ۸۰۰۰ گام بر ثانیه در ستون آخر رسیده است. در این جدول به راحتی می‌توان «بهبود ناشی از به کار بردن الگوریتمی با رفتار مجانبی بهتر» را با «بهبود ناشی از افزایش سرعت رایانه به اندازه‌ی ضریبی ثابت» مقایسه کرد. یک الگوریتم نمایی برای حل مسأله‌هایی با اندازه‌ی ورودی ۱۰۰۰ به زمانی نجومی (میلیاردها میلیارد سال) نیاز دارد (مگر آن که پایه‌ی آن به ۱ بسیار نزدیک باشد).

زمان اجرای پارامتری	زمان واقعی اجرا در رایانه‌ای با سرعت ۱۰۰۰ گام بر ثانیه	زمان واقعی اجرا در رایانه‌ای با سرعت ۲۰۰۰ گام بر ثانیه	زمان واقعی اجرا در رایانه‌ای با سرعت ۴۰۰۰ گام بر ثانیه	زمان واقعی اجرا در رایانه‌ای با سرعت ۸۰۰۰ گام بر ثانیه
$\log_2^n$	۰/۰۱۰	۰/۰۰۵	۰/۰۰۳	۰/۰۰۱
n	۱	۰/۵	۰/۲۵	۰/۱۲۵
$n\log_2^n$	۱۰	۵	۲/۵	۱/۲۵
$n^{1.5}$	۳۲	۱۶	۸	۴
n^2	۱۰۰۰	۵۰۰	۲۵۰	۱۲۵
n^3	۱,۰۰۰,۰۰۰	۵۰۰,۰۰۰	۲۵۰,۰۰۰	۱۲۵,۰۰۰
1.1^n	$۱۰^{۳۹}$	$۱۰^{۳۹}$	$۱۰^{۳۸}$	$۱۰^{۳۸}$

جدول ۳-۱ زمان اجرا (به ثانیه) در شرایط گوناگون (n=۱۰۰۰)

نماد O برای نشان دادن حد بالای زمان اجرای الگوریتم‌ها به کار می‌رود؛ اما استفاده از حد بالا به تنهایی کافی نیست، زیرا برای مثال، زمان اجرای همه‌ی الگوریتم‌های این کتاب از $O(2^n)$ است؛ یعنی به زمانی، بیش از زمان نمایی نیاز ندارند. به هر حال $O(2^n)$ برای بیش‌تر این الگوریتم‌ها، حد بالای خام و ناشیانه‌ای است، چون بیش‌تر آن‌ها بسیار سریع‌تر هستند. علاوه بر حد بالا، به «عبارتی» که تا جای ممکن به زمان واقعی اجرا نزدیک باشد، نیز توجه داریم. در مواردی که پیدا کردن عبارت دقیق، بسیار دشوار باشد، علاقه‌مندیم تا دست‌کم حد بالا و حد پایین را برای زمان اجرا به دست آوریم. به دست آوردن حد پایین، بسیار دشوارتر از به دست آوردن حد بالاست. حد بالای زمان اجرای الگوریتم، تنها نشان می‌دهد که زمان اجرای آن الگوریتم از مقداری مشخص بیش‌تر نیست، اما حد پایین باید

مشخص کند که حد بهتری (کم‌تری) برای الگوریتم وجود ندارد. یقیناً نمی‌توان همه‌ی الگوریتم‌های ممکن برای حل یک مسأله را بررسی کرد تا حد پایین زمان اجرای آن مسأله به دست آید. ما به روشی برای مدل کردن مسأله‌ها و الگوریتم‌ها نیاز داریم به گونه‌ای که بتوانیم حد پایین آن‌ها را پیدا کنیم. درباره‌ی حد پایین در بخش ۶-۴-۶ بحث خواهد شد. برای حد پایین نیز در صورت نادیده گرفتن ضریب‌های ثابت، نمادی مشابه حد بالا وجود دارد. اگر ثابت‌های c و N موجود باشند، به گونه‌ای که برای $n \geq N$ (n اندازه‌ی ورودی الگوریتم است)؛ T(n) یا تعداد گام‌های لازم برای حل مسأله دست‌کم cg(n) باشد، آنگاه گوییم: $T(n) = \Omega(g(n))$. بنابراین برای مثال $n^2 = \Omega(n^2 - 100)$ یا $n = \Omega(n^{0.9})$ پس نماد Ω متناظر با رابطه‌ی «$\geq$» است.

اگر برای دو تابع f(n) و g(n)، هر دو رابطه‌ی f(n)=O(g(n)) و $f(n) = \Omega(g(n))$ درست باشند، آنگاه گوییم: $f(n) = \theta(g(n))$؛ برای مثال $5n\log_2^n - 10 = \theta(n\log n)$. (در رابطه‌ی $\theta(n\log n)$، مبنای لگاریتم را حذف کردیم، چون تغییر مبنا، مقدار لگاریتم را با ضریبی ثابت تغییر می‌دهد.) در اثبات θ، لازم نیست ثابت‌های به‌کاررفته برای اثبات O و Ω یکسان باشند.

نمادهای O، Ω و θ به ترتیب شبیه «$\leq$»، «$\geq$» و «=» هستند. گاهی هم لازم است نمادی متناظر با «<» یا «>» را به کار ببریم. گوییم: f(n)=o(g(n)) (این‌گونه تلفظ می‌شود: f(n) از «اُی کوچک g(n)» است) اگر:

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

برای مثال $n\log_2^n = o(n)$ اما n/10≠o(n). همچنین گوییم f(n)=ω(g(n)) اگر g(n)=o(f(n)).

می‌توانیم قضیه‌ی ۳-۱ را با جای‌گزین کردن o (کوچک) به جای O (بزرگ) قوی‌تر کنیم:

□ قضیه‌ی ۳-۳

برای هر ثابت c>0 و هر ثابت a>1 و هر تابع صعودی f(n) داریم: $(f(n))^c = o(a^{f(n)})$. به عبارت دیگر، تابع نمایی سریع‌تر از تابع چندجمله‌ای رشد می‌کند.

□

نماد OO

با گذشت زمان ایرادهای فراوانی در نماد O پیدا شده است. مهم‌تر از همه این که ضریب‌های ثابت عملاً تأثیرگذار هستند، در حالی که کاربرد گسترده‌ی این نماد موجب شده است تا به راحتی آن‌ها را نادیده بگیریم. لازم است همواره به یاد داشته باشیم که به کار بردن این نماد، تنها به عنوان یک برآورد اولیه، سودمند است. در واقع، رسیدن به O یا مرتبه‌ی بهتری برای زمان اجرا ایده‌ی ساخت بسیاری از

الگوریتم‌های کاربردی بوده است. این نماد در رشد نظریه‌ی پیچیدگی نیز تأثیر داشته است؛ نظریه‌ای که جنبه‌های بسیاری از بهترین کارایی ممکن الگوریتم‌ها را آشکار می‌سازد.

در برخی حالت‌ها، ثابت‌هایی که نماد O از آن‌ها چشم می‌پوشد، به گونه‌ای سرسام‌آور بزرگند ولی در برخی حالات دیگر این ثابت‌ها بسیار کوچک هستند. ما باید این دو وضعیت را از هم تفکیک کنیم، چراکه هنوز الگوریتم‌های وضعیت دوم در عمل کارآمد هستند. نمادی جدید در کتاب می‌آوریم تا به کمک آن بتوانیم این دو وضعیت را از هم تشخیص دهیم. این نماد تعریف دقیق ریاضی ندارد و تنها کاربردش، قرار دادن آن به جای بخش‌هایی از عبارت مشخص‌کننده‌ی زمان اجرای الگوریتم است که زمان اجرای «آن بخش‌ها» طبق نماد O، تنها در مباحث نظری سودمند است؛ یعنی ضریب‌های ثابت آن‌ها بسیار بزرگ هستند. ما OO(f(n)) را پیش‌نهاد می‌کنیم که همان O(f(n)) است؛ با این تفاوت که ثابت‌هایش در بیش‌تر کاربردهای عملی بسیار بسیار بزرگ هستند. (نماد OO(f(n)) را «اُی اُیِ» f(n) بخوانید. این نماد به آسانی به خاطر سپرده می‌شود، چراکه مانند ∞ است.)

چون دقیقاً روشن نیست که یک ثابت معین در «عبارت مشخص‌کننده‌ی زمان اجرای یک الگوریتم» باید چقدر باشد تا آن الگوریتم را در عمل غیرقابل‌استفاده کند؛ پس استفاده از OO به قضاوت شخصی نویسنده برمی‌گردد. قصد نداریم از این نماد تعریفی دقیق‌تر ارائه کنیم، چون هدف اصلی‌مان تنها عرضه‌ی دیدگاهمان به صورتی ساده به خوانندگان است. هدف دیگر از معرفی این نماد تأکید بر این نکته است که نماد O، نمادی کامل و جامع نیست.

۳-۳ پیچیدگی فضایی (حافظه‌ی مورد نیاز) و پیچیدگی زمانی

چگونه می‌توان زمان مورد نیاز الگوریتم را بدون اجرای آن تحلیل کرد؟ برای این کار لازم است تعداد گام‌هایی را که الگوریتم برمی‌دارد، بشماریم. مشکل این است که گام‌های گوناگونی، ممکن است و برای برداشتن هر یک از این گام‌ها به مقدار زمانی متفاوت نیازمندیم. برای نمونه، زمان محاسبه‌ی یک تقسیم مکن است از زمان محاسبه‌ی یک جمع طولانی‌تر باشد. یک راه برای تحلیل الگوریتم، شمارش جداگانه‌ی تعداد گام‌های متفاوت است، اما این کار در بیش‌تر موارد بسیار دست‌وپاگیر است. علاوه بر آن، زمان واقعی اجرای این گام‌ها، هم وابسته به کامپایلر و هم وابسته به سخت افزار مورد استفاده است. تلاش می‌کنیم تا از این وابستگی‌ها دوری کنیم.

به جای شمارش همه‌ی انواع گام‌ها، تنها روی یک نوع محوری آن متمرکز می‌شویم. برای مثال، در تحلیل یک الگوریتم مرتب‌سازی، عمل مقایسه را به عنوان گام محوری برمی‌گزینیم چون به طور شهودی روشن است که عمل محوری مرتب‌سازی، مقایسه‌ی عناصر است و اعمال دیگر را می‌توان سربار این عمل دانست. البته هنوز هم ناگزیر باید مطمئن شویم که مقایسه‌ها، بخش محوری الگوریتم هستند. از آنجا که ضریب‌های ثابت را نادیده می‌گیریم، پس تنها کافی است مطمئن شویم تعداد کل

اعمال دیگر از c برابرِ تعداد مقایسه‌ها کم‌تر است (c ضریبی ثابت است). چنان‌چه این مطلب درست باشد و چنان‌چه O(f(n)) حد بالایی برای تعداد مقایسه‌ها باشد، آنگاه O(f(n)) حد بالایی برای تعداد کل گام‌هاست. در این صورت، گوییم پیچیدگی زمانی الگوریتم (یا همان زمان اجرا) از O(f(n)) است. با این روش می‌توان بر مشکل تفاوت گام‌ها (یعنی نیاز هر گام به زمان اجرایی متفاوت) چیره شد؛ البته به شرط آن که تفاوت‌ها مقداری ثابت باشد.

پیچیدگی فضایی یک الگوریتم، بیانگر مقدار حافظه‌ی ضروری برای اجرای آن الگوریتم است. در بیش‌تر موارد حافظه‌ی لازم برای ورودی یا خروجی را جزو پیچیدگی فضایی به حساب نمی‌آوریم. علت این است که پیچیدگی فضایی، برای مقایسه‌ی الگوریتم‌های متفاوت، روی مسأله‌ای یکسان (با ورودی و خروجی مشخص) است. در ضمن، حل مسأله بدون بهره‌گیری از ورودی و خروجی، شدنی نیست و ما تنها می‌خواهیم حافظه‌ی قابل صرفه‌جویی را حساب کنیم. حافظه‌ی لازم برای نگه‌داری کدهای برنامه را نیز حساب نمی‌کنیم، زیرا مقدار این حافظه مستقل از ورودی است. برای پیچیدگی فضایی نیز همانند پیچیدگی زمانی، بدترین حالت را در نظر می‌گیریم و معمولاً آن را به صورت عبارتی مجانبی از اندازه‌ی ورودی نشان می‌دهیم. پس حافظه‌ی مورد نیاز الگوریتمی با پیچیدگی فضایی O(1)، مستقل از اندازه‌ی ورودی است و حافظه‌ی لازم برای الگوریتمی با پیچیدگی فضایی O(n)، حداکثر ضریبی است ثابت از اندازه‌ی ورودی.

شمارش تعداد گام‌های محوری، همیشه آسان نیست. در بخش بعد خلاصه‌ی چند روش ریاضی را برای محاسبه‌ی زمان اجرا یا پیچیدگی زمانی معرفی می‌کنیم؛ اما معمولاً برآورد پیچیدگی فضایی برای یک الگوریتم مشخص، ساده است؛ پس درباره‌ی آن کم‌تر بحث می‌کنیم.

۳-۴ محاسبه‌ی حاصل‌جمع‌ها

اگر الگوریتمی از چند بخش تشکیل شده باشد، پیچیدگی آن، از جمع پیچیدگی تک تک بخش‌هایش به دست می‌آید. این موضوع همیشه آن قدر که به نظر می‌رسد، ساده نیست. ممکن است الگوریتم، حلقه‌ای داشته باشد که دفعات زیادی اجرا شود و هر بار، پیچیدگی آن با دیگر بارها تفاوت داشته باشد. برای تحلیل این الگوریتم‌ها، به روش‌هایی ویژه برای جمع عبارات نیاز داریم. احتمالاً ساده‌ترین حالت، حلقه‌ای به اندازه‌ی n است که در iامین گام آن (i≤n)، i عمل انجام می‌شود. در این حالت، تعداد کل اعمال، 1+2+...+n است. برای نمایش جمع، نماد $\sum$ را به کار می‌بریم؛ برای مثال 1+2+...+n را به صورت $\sum_{i=1}^{n} i$ می‌نویسیم که معنای آن «جمع جمله‌ی i، برای i از ۱ تا n» است. چنان که در بخش ۲-۲ دیدیم، حاصل این جمع، n(n+1)/2 می‌شود. می‌توانیم این حالت را با حالتی مقایسه کنیم که در

هر گام حلقه، دقیقاً n عمل انجام می‌شود؛ می‌بینیم که با کاهش زمان اجرای گام iام، از n به i، زمان اجرای حلقه تقریباً نصف خواهد شد.

□ مثال ۳-۱

اینک حلقه‌ای را در نظر بگیرید که در گام iام آن، i^2 عمل انجام می‌شود. به عبارت دیگر، می‌خواهیم این حاصل‌جمع را حساب کنیم:

$$S_2(n)=\sum_{i=1}^{n} i^2$$

چون زمان اجرای حلقه برای n^2 عمل در هر گام برابر با n^3 است، پس روشن است که $S_2(n)\le n^3$. به این ترتیب می‌توانیم حدس بزنیم که $S_2(n)$ عبارتی از درجه‌ی سه است. به یاری استقرا، می‌توانیم این حدس را اثبات کنیم و ثابت‌ها را نیز بیابیم. حدس ما، $S_2(n)=P(n)=an^3+bn^2+cn+d$ است. باید $P(1)=1$ و $P(n)$، شرط گام استقرا یعنی $P(n+1)=P(n)+(n+1)^2$ را نیز برآورده سازد. از گام استقرا نتیجه می‌شود: $a(n+1)^3+b(n+1)^2+c(n+1)+d-(an^3+bn^2+cn+d)=n^2+2n+1$

چون ضرایب توان‌های یکسان n، باید برابر باشند، داریم:

ضریب n^2 $3a+b-b=1$

ضریب n^1 $3a+2b+c-c=2$

ضریب n^0 یا مقدار ثابت $a+b+c+d-d=1$

از این معادله‌ها نتیجه می‌شود که $a=1.3$، $b=1.2$ و $c=1.6$. مقدار d از روی حالت اولیه $P(1)=1$ به دست می‌آید: $a+b+c+d=1$. پس باید $d=0$. با ترکیب تمام جمله‌های پیش داریم:

$$S_2(n)=\frac{n^3}{3}+\frac{n^2}{2}+\frac{n}{6}=\frac{n(n+1)(2n+1)}{6} \qquad (۳-۳)$$

جالب است که شبیه حالت پیش با کاهش اندازه‌ی گام iام از n^2 به i^2، زمان اجرا تقریباً 1/3 می‌شود. □

راه دیگری برای رسیدن به عبارت (۳-۳) وجود دارد. این روش ترفندی کلی است که بارها آن را به کار خواهیم برد: اگر حدس زده باشیم $S_2(n)$ یک چندجمله‌ای از درجه‌ی سه است، آنگاه می‌توانیم تلاش کنیم $S_2(n)$ را به صورت یک چندجمله‌ای از درجه‌ی سه نمایش دهیم و سپس با حل معادله‌هایی، ضرایب $S_2(n)$ را پیدا کنیم. این حاصل‌جمع را در نظر بگیرید:

$$S_3(n)=\sum_{i=1}^{n} i^3 \qquad (۴-۳)$$

نخست، (۳-۴) را به صورتی دیگر می‌نویسیم:

$$S_3(n)=\sum_{i=1}^{n} i^3=\sum_{i=1}^{n}(i-1+1)^3=\sum_{i=0}^{n-1}(i+1)^3$$
$$=\sum_{i=0}^{n-1}(i^3+3i^2+3i+1) \qquad (۵-۳)$$

به عبارت دیگر، محدوده‌ی جمع را جابه‌جا کردیم، چنان‌که به جای ۱ تا n از ۰ تا n-1 باشد. این جابه‌جایی در شکل ۳-۱ توضیح داده شده است. حال می‌توانیم (۳-۵) را چنین بنویسیم:

$$\sum_{i=1}^{n} i^3=\sum_{i=0}^{n-1}(i^3+3i^2+3i+1) \qquad (۶-۳)$$

محدوده‌ی جمله‌ای i^3 از ۱ تا n-1، در هر دو سمت (۳-۶) مشترک است؛ پس می‌توان آن را از هر دو سمت کنار گذاشت:

$$n^3=0^3+\sum_{i=0}^{n-1}(3i^2+3i+1)$$

می‌دانیم $\sum_{i=0}^{n-1} i=n(n-1)/2$ و همچنین روشن است $\sum_{i=0}^{n-1} i^2=S_2(n)-n^2$ (چون تنها تفاوت در جمله‌ی n ام است). در نتیجه داریم:

$n^3=3(S_2(n)-n^2)+3n(n-1)/2+n$

حال می‌توانیم $S_2(n)$ را حساب کنیم:

$n^3-3n(n-1)/2-n=3(S_2(n)-n^2)$

$$S_2(n)=\frac{n^3-3n(n-1)/2-n}{3}+n^2=\frac{n^3}{3}+\frac{3n^2}{6}+\frac{n}{6}=\frac{n(n+1)(2n+1)}{6}$$

که دقیقاً همان عبارت (۳-۳) است.

ترفند اصلی، شمردن حاصل‌جمعی مشخص (در اینجا $S_3(n)$) از دو راه گوناگون بود؛ به گونه‌ای که این دو راه بخش عمده‌ای از یکدیگر را حذف کنند. مجموع‌های فراوان دیگری نیز وجود دارند که چنین رفتاری از خود بروز می‌دهند. اگر اختلاف بین جمع $f_1+f_2+\ldots+f_n$ و جمع جابه‌جاشده‌ی $f_2+f_3+\ldots+f_{n+1}$ را در نظر بگیریم، می‌بینیم که بیش‌تر ضرایب یکدیگر را حذف می‌کنند. تنها نکته‌ی مبهمی که باقی می‌ماند، محدوده‌ی جملات است. برای برطرف کردن این نکته‌ی مبهم، سه مثال دیگر از این روش ارائه می‌کنیم.

$$1^3 + 2^3 + \ldots + (n-1)^3 + n^3$$

$$(0+1)^3 + (1+1)^3 + (2+1)^3 + \ldots + (n-1+1)^3$$

شکل ۳-۱ محاسبه‌ی یک جمع با جابه‌جایی محدوده‌ی آن

☐ مثال ۳-۲

این جمع را در نظر بگیرید:

$$F(n)=\sum_{i=0}^{n}2^i=1+2+4+\ldots+2^n$$

می‌خواهیم با جابه‌جایی جملات، F(n) را به گونه‌ای دیگر بنویسیم و سپس از مقایسه‌ی این حالت با خود F(n)، بخش عمده‌ای از آن را حذف کنیم. اختلاف بین دو جمله‌ی متوالی در F(n)، یک ضریب ۲ است. پس بیایید کل عبارت را در ۲ ضرب کنیم تا بتوانیم محدوده‌ها را جابه‌جا کنیم:

$2F(n)=2+4+8+\ldots+2^n+2^{n+1}$

حال به عبارتی شامل F(n) می‌رسیم:

$2F(n)-F(n)=2^{n+1}-1$

و این عبارت به سادگی نشان می‌دهد: $F(n)=2^{n+1}-1$.

☐

☐ مثال ۳-۳

حالا یک جمع دشوارتر را در نظر بگیرید:

$$G(n)=\sum_{i=1}^{n}i2^2=1.2^1+2.2^2+3.2^3+\ldots+n.2^n$$

باز هم می‌توانیم همان روش را به کار گیریم:

$$2G(n)=1.2^2+2.2^3+3.2^4+\ldots+n.2^{n+1}$$

می‌بینیم که همه‌ی توان‌ها یک واحد افزایش یافته‌اند. با کم کردن G(n) از این عبارت، ضریب i را از جملات حذف می‌کنیم:

$G(n)=2G(n)-G(n)=n.2^{n+1}-(1.2^1+1.2^2+\ldots+1.2^n)$
$=n.2^{n+1}-(2^{n+1}-2)=(n-1)2^{n+1}+2$

(توضیح مترجمان: حاصل $1.2^1+1.2^2+\ldots+1.2^n$ در مثال ۳-۲ حساب شد.)

☐

☐ مثال ۳-۴

سرانجام به این جمع دقت کنید که در بخش ۶-۴-۵ برای تحلیل مرتب‌سازی هرمی به کار خواهد آمد:

$$G(n)=\sum_{i=1}^{n}i2^{n-i}=1.2^{n-1}+2.2^{n-2}+3.2^{n-3}+\ldots+n.2^0$$

همان روش را به کار می‌بریم:

$$2G(n)=1.2^n+2.2^{n-1}+3.2^{n-2}+\ldots+n.2^1$$

در اینجا نیز با تفاضل دو عبارت، تأثیر عامل i را از بین می‌بریم:

$$G(n)=2G(n)-G(n)=2^n+1.2^{n-1}+1.2^{n-2}+\ldots+1.2^1-n.2^0$$
$=2^{n+1}-2-n$

☐

۳-۵ رابطه‌های بازگشتی

رابطه‌ی بازگشتی راهی برای بیان یک تابع با بهره‌گیری از خود آن تابع است. احتمالاً مشهورترین رابطه‌ی بازگشتی، رابطه‌ی است که اعداد فیبوناچی را تعریف می‌کند:

$$F(n)=F(n-1)+F(n-2),\ F(1)=1,\ F(2)=1 \qquad (۷-۳)$$

این رابطه تابع را به گونه‌ای یکتا تعریف می‌کند. از روی این رابطه می‌توانیم مقدار تابع را به ازای هر عدد طبیعی حساب کنیم؛ برای نمونه: $F(3)=F(2)+F(1)=2$، $F(4)=F(3)+F(2)=3$ و... . به هر حال اگر بخواهیم با این تعریف، مقدار تابع را حساب کنیم، برای محاسبه‌ی $F(k)$ نیازمند $k-2$ گام هستیم. کار با عبارت صریح (یا بسته‌ی) $F(n)$ خیلی راحت‌تر از این رابطه‌ی بازگشتی است، زیرا با این عبارت می‌توانیم $F(n)$ را به سرعت، حساب و در صورت نیاز آن را با دیگر توابع آشنا مقایسه کنیم. این کار را «حل رابطه‌ی بازگشتی» گویند. (توجه کنید که در برخی منابع انگلیسی زبان، گاهی recurrence relation را به طور ساده recurrence نیز می‌گویند.)

رابطه‌های بازگشتی، در تحلیل الگوریتم‌ها کاربرد فراوانی دارند. در اینجا درباره‌ی راهی سودمند برای حل رابطه‌های بازگشتی، کمی بحث کرده، راه‌حل‌هایی کلی برای دو دسته‌ی ویژه از رابطه‌های بازگشتی ارائه می‌کنیم. این دو دسته، از رایج‌ترین رابطه‌های بازگشتی در تحلیل الگوریتم‌ها هستند. این رابطه‌ها بعداً در کتاب به کار خواهند رفت.

۳-۵-۱ حدس‌های هوشمندانه

حدس زدن راه‌حل، کاری غیرعلمی به نظر می‌رسد، اما باید غرور علمی خودمان را کنار بگذاریم و بپذیریم که این روش برای دسته‌ی گسترده‌ای از رابطه‌های بازگشتی بسیار مناسب است. هنگامی که دنبال راه‌حل دقیق نمی‌گردیم؛ یعنی تنها حد بالایی برای پاسخ می‌خواهیم، این روش بهتر هم هست. از آنجا که یافتن حدی مشخص برای پاسخ از اثبات درستی آن حد دشوارتر است، پس این شیوه سودمند خواهد بود. رابطه‌ی بازگشتی (۳-۸) را که تنها برای توان‌های ۲ تعریف شده است، در نظر بگیرید:

$$T(2n)\leq 2T(n)+2n-1\ ,\ T(2)=1 \qquad (۸-۳)$$

این رابطه را به جای تساوی، با نامساوی نشان داده‌ایم تا صرفاً برای یافتن حد بالا (یعنی همان هدف نماد O) مناسب شود. از آنجا که در پی بدترین حالت هستیم، یاری گرفتن از نامساوی بهتر است (بدترین حالت را در سمت راست نامساوی می‌گذاریم). می‌خواهیم تابعی مانند $f(n)$ را چنان بیابیم که $T(n)$ از $O(f(n))$ باشد، اما $f(n)$ از مقدار واقعی $T(n)$ خیلی دور نشده باشد.

در صورت داشتن حدسی برای f(n)، مثلاً $f(n)=n^2$، با استقرا روی n ثابت می‌کنیم: $T(n)=O(f(n))$. نخست، حالت پایه را بررسی می‌کنیم: $T(2)=1\le f(2)=4$. سپس ثابت می‌کنیم از $T(n)\le f(n)$ می‌توان $T(2n)\le f(2n)$ را به دست آورد. پس باید ثابت کنیم از درستی $T(n)\le n^2$، درستی $T(2n)\le (2n)^2$ نتیجه می‌شود. اثبات چنین است:

$T(2n)\le 2T(n)+2n-1$ (بنا به تعریف رابطه‌ی بازگشتی)

$\le 2n^2+2n-1$ (بنا به فرض استقرا)

$< (2n)^2$

که همان چیزی است که می‌خواستیم ثابت کنیم. پس داریم: $T(n)=O(n^2)$. اما آیا n^2 برآوردی مناسب برای T(n) است؟ در آخرین مرحله‌ی این اثبات، به جای $2n^2+2n-1$، مقدار بزرگ‌ترِ $4n^2$ را قرار دادیم. اما فاصله‌ی قابل توجهی (تقریباً $2n^2$) بین این دو عبارت وجود دارد؛ پس شاید n^2 برآورد بسیار بالایی برای T(n) باشد.

بیایید برآورد کوچک‌تری را آزمایش کنیم؛ مثلاً f(n)=cn که در آن c مقداری ثابت است. روشن است که cn کندتر از T(n) رشد می‌کند، زیرا $2n=2cn$ c و دیگر 2n-1 اضافه نشده است. (توضیح مترجمان: به رابطه‌ی ۳-۸ دقت کنید.) از این رو، مقدار واقعی T(n) بین cn و n^2 است. اینک $T(n)\le n\log_2^n$ را می‌آزماییم. روشن است که $T(2)\le 2\log_2^2$. فرض کنید $T(n)\le n\log_2^n$ و T(2n) را در نظر بگیرید:

$T(2n)\le 2T(n)+2n-1$ (بنا به تعریف رابطه‌ی بازگشتی)

$\le 2n\log_2^n+2n-1$ (بنا به فرض استقرا)

$< 2n(\log_2^{2n})$

که همان چیزی است که می‌خواستیم. از آنجا که تفاوت دو عبارت اخیر تنها ۱ است، پس به مقدار واقعی T(n) بسیار نزدیک شده‌ایم. بعدها برهانی به کمک یک مقدار ثابت ارائه خواهیم کرد که نشان می‌دهد پاسخ دقیق واقعاً همین $n\log_2^n$ است.

رابطه‌ی بازگشتی (۳-۸) تنها برای توان‌های ۲ تعریف می‌شود. می‌توانیم رابطه‌ای شبیه آن، اما برای همه‌ی مقادیر صحیح n تعریف کنیم:

$$T(n)\le 2T(\lfloor n/2\rfloor)+n-1,\ T(2)=1 \qquad (۹-۳)$$

(توجه کنید که علامت کف - یعنی $\lfloor\ \rfloor$ - ضروری است، چون T(n) تنها برای اعداد صحیح تعریف می‌شود.) رابطه‌ی بازگشتی (۳-۹) از رابطه‌ی (۳-۸) کلی‌تر است، زیرا با آن که برای تمام nها، تعریف شده، اما برای توان‌های ۲، دقیقاً به همان رابطه‌ی (۳-۸) تبدیل می‌شود. پس، برای nهایی که توان ۲ باشند: $T(n)=O(n\log n)$. حال نشان می‌دهیم که این حد بالا، برای تمام nها معتبر است. روشن است که T(n) تابعی صعودی است. پس اگر n، توانی از ۲ نباشد و 2^k را نخستین توانی از ۲ بگیریم که از n بزرگ‌تر است، T(n) نیز نباید از $T(2^k)$ بیش‌تر باشد؛ یعنی برای چنین kیی داریم: $2^{k-1}<n<2^k$؛ در

نتیجه: $T(2^{k-1}) \le T(n) \le T(2^k)$. پیش‌تر ثابت کردیم که مقدار ثابت c وجود دارد که $T(2^k) \le c2^k \log_2^{2^k}$. از این رو، برای یک ثابت دیگر مانند c_1 داریم:

$$T(n) \le c2^k \log_2^{2^k} \le c(2n)\log_2^{(2n)} \le c_1 n \log_2^n$$

که به ازای هر n رابطه‌ی T(n)=O(nlogn) را برقرار می‌کند. معمولاً هنگام جست‌وجوی یک عبارت مجانبی، کافی است n را توانی از ۲ در نظر بگیریم.

بیایید مراحلی را که در اثبات استقرایی به کار می‌روند، برای رابطه‌ی بازگشتی نیز به کار گیریم. برای مثال رابطه‌ی بازگشتی (۳-۱۰) را در نظر بگیرید:

$$T(g(n))=E(T,n) \qquad (۱۰-۳)$$

g(n) تابعی از n است (که رشد رابطه‌ی بازگشتی را تعریف می‌کند) و E، عبارتی شامل T(n) و n است. مثلاً در (۳-۸)، g(n)=2n و E(T,n)=2T(n)+2n-1. همچنین فرض کنید که برای تابعی مانند f(n)، حدس زده‌ایم که T(n)≤f(n). برای اثبات این حدس لازم است که در f(n)، g(n) را به جای n قرار دهیم و سپس در E، f(n) را جای‌گزین T(n) کنیم. پس از آن باید نشان دهیم f(g(n)) از مقداری که جای‌گزین E(T,n) شده است، کوچک‌تر نیست. باید ثابت کنیم:

$$f(g(n)) \ge E(f,n) \qquad (۱۱-۳)$$

برای مثال در رابطه‌ی (۳-۸) حدس زدیم که $f(n) = n\log_2^n$؛ پس بایست نشان می‌دادیم:

$$(2n)(\log_2^{(2n)}) \ge 2(n\log_2^n) + 2n - 1$$

اشتباهی رایج، اثبات عکس این مطلب است - یعنی قرار دادن «کوچک‌تر از» به جای «بزرگ‌تر از». توضیحی شهودی برای این مطلب که می‌توان آن را به آسانی به خاطر سپرد، چنین است: ما می‌خواهیم ثابت کنیم f(n) بسیار سریع‌تر از T(n) رشد می‌کند. بنابراین اگر در f(n) و T(n)، g(n) را به جای n جای‌گزین کنیم، مقدار f(n) از T(n) بزرگ‌تر خواهد شد. از طرفی T(g(n))=E(T,n) (همان رابطه‌ی بازگشتی)؛ پس می‌توانیم به جای T(g(n))، E(f,n) را قرار دهیم. ممکن است این فرایند، چندین بار با توابع (حدس‌های) گوناگون تکرار شود تا سرانجام نامساوی مورد نظر ثابت گردد.

اشتباه رایج دیگر، به کار بردن نماد O به هنگام حدس زدن است؛ به این صورت که پاسخ را از O(f(n)) حدس می‌زنیم و به جای n، O(f(n)) قرار می‌دهیم، غافل از آن که در اینجا نمی‌توان نماد O را به کار برد؛ چون حتا اگر در پایان به ثابت‌ها کاری نداشته باشیم، نمی‌توانیم طی اثبات، آن‌ها را نادیده بگیریم. برای مثال اگر بخواهیم با جای‌گزین کردن O(n) به جای n، ثابت کنیم پاسخ (۳-۸) از O(n) است، چنین چیزی به دست می‌آوریم (حالت پایه روشن است):

$T(2n) \le 2T(n)+2n-1$ (بنا به تعریف رابطه‌ی بازگشتی)

$\le O(n)+2n-1$ (بنا به فرض استقرا)

$= O(n)$

اما چنان که پیش‌تر دیدیم، این نتیجه نادرست است. خطای این روش از این واقعیت ناشی می‌شود که ثابت‌های متفاوتی را در مراحل مختلف برهان به کار بردیم (یا به عبارت دیگر، از ثابتهای متفاوتی در مراحل گوناگون اثبات چشم‌پوشی کردیم). روش درست، در نظر گرفتن صریح این ثابت‌هاست. هنگامی که حدس می‌زنیم جواب از $O(f(n))$ است، در واقع چنین حدس زده‌ایم که پاسخ، $cf(n)$ است که مقدار ثابت c را بعداً مشخص می‌کنیم.

حال می‌کوشیم تا با حدس زدن، رابطه‌ی فیبوناچی را حل کنیم؛ یعنی چنین رابطه‌ای به ما داده شده است:

$$F(n)=F(n-1)+F(n-2),\ F(1)=1,\ F(2)=1 \qquad (۱۲-۳)$$

از آنجا که مقدار $F(n)$، مجموع دو مقدار پیشین آن است، یک حدس معقول این است که بگوییم هر بار $F(n)$، دو برابر می‌شود؛ یعنی، پاسخ تقریباً 2^n است. پس $F(n)=c2^n$ را بررسی می‌کنیم. با جای‌گزینی $c2^n$ در (۳-۱۲) داریم:

$$c2^n=c2^{n-1}+c2^{n-2}$$

روشن است که این تساوی ناممکن است، چراکه می‌توان c را از دو طرف تساوی کنار گذاشت و آنگاه سمت چپ تساوی از سمت راست آن بزرگ‌تر می‌گردد. بنابراین، می‌فهمیم که $c2^n$ زیادی بزرگ است و ضریب ثابت c هم نقشی در گام استقرا ندارد.

دوباره می‌کوشیم یک تابع نمایی (اما با پایه‌ای کوچک‌تر) را بیازماییم. به جای حدس زدن پایه‌های مختلف، آسان‌تر است که پایه را یک پارامتر در نظر بگیریم و سپس در هنگام بررسیِ درستی این حدس، مقدار پارامتر را محاسبه کنیم. $F(n)=a^n$ را که در آن a یک ثابت است، بررسی می‌کنیم. با جای‌گزین کردن آن در (۳-۱۲) داریم:

$$a^n=a^{n-1}+a^{n-2}$$

که نتیجه می‌شود:

$$a^2=a+1 \qquad (۱۳-۳)$$

دو جواب (۳-۱۳) عبارتند از $a_1=(1+\sqrt{5})/2$ و $a_2=(1-\sqrt{5})/2$. چون در حال حاضر داریم: $F(n)=O((a_1)^n)$، پس $(a_1)^n$ باید در رابطه‌ی بازگشتی صدق کند. از اینجا به آسانی می‌توانیم ثابت c را به گونه‌ای بیابیم که $c(a_1)^n$ از مقادیر داده‌شده برای $n=1$ و $n=2$ بزرگ‌تر باشد.

اگر بخواهیم مقدار دقیق $F(n)$ را بیابیم، باید مقادیر اولیه‌ی آن را به دقت در نظر بگیریم. از آنجا که $(a_1)^n$ و $(a_2)^n$، هر دو رابطه‌ی بازگشتی را حل می‌کنند، پس هر ترکیب خطی از آن دو نیز رابطه‌ی بازگشتی را حل خواهد کرد. پس حل کلی رابطه‌ی بازگشتی چنین است:

$$c_1(a_1)^n+c_2(a_2)^n$$

حال لازم است که مقادیر c_1 و c_2 را به گونه‌ای محاسبه کنیم که عبارت حاصل برای مقادیر F(1) و F(2) نیز برقرار باشد. به سادگی روشن می‌شود که $c_1 = 1/\sqrt{5}$ و $c_2 = -1/\sqrt{5}$. در نتیجه حل دقیق رابطه‌ی فیبوناچی چنین است:

$$F(n) = \frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^n - \frac{1}{\sqrt{5}}\left[\frac{1-\sqrt{5}}{2}\right]^n$$

معادله‌ی $a^2=a+1$ را که هنگام جست‌وجو برای حل رابطه‌ی بازگشتی (۳-۱۲) با آن روبه‌رو شدیم، «معادله‌ی مشخصه‌ی» رابطه‌ی بازگشتی می‌نامیم. همین روش را برای حل هر رابطه‌ی بازگشتی به صورت:

$$F(n)=b_1F(n-1)+b_2F(n-2)+\ldots+b_kF(n-k)$$

به کار می‌بریم (k یک ثابت است).

۳-۵-۲ روابط «تقسیم‌وحل»

در یک الگوریتم «تقسیم‌وحل»، مسأله، به زیرمسأله‌هایی کوچک‌تر تقسیم می‌شود؛ هر زیرمسأله را به صورت بازگشتی حل می‌کنیم و سپس حل زیرمسأله‌ها را برای حل مسأله‌ی اصلی ترکیب می‌کنیم. فرض کنید مسأله‌ی اصلی را به a زیرمسأله (هر یک به اندازه‌ی 1/b از مسأله‌ی اصلی) تبدیل کرده‌ایم و الگوریتم ترکیب‌کننده‌ی حل زیرمسأله‌ها در زمان cn^k اجرا شود (a، b، c و k ثابت هستند). در نتیجه، زمان اجرای الگوریتم یعنی T(n) در این رابطه صدق می‌کند:

$$T(n)=aT(n/b)+cn^k \qquad \textbf{(۱۴-۳)}$$

برای سادگی، n را b^m در نظر می‌گیریم، تا n/b عددی صحیح شود (b عدد صحیحی بزرگ‌تر از یک است). نخست می‌کوشیم رابطه‌ی (۳-۱۴) را چندین و چندبار گسترش دهیم تا معنای آن را متوجه شویم:

$$T(n)=a(aT(n/b^2)+c(n/b)^k)+cn^k=a(a(aT(n/b^3)+c(n/b^2)^k)+c(n/b)^k)+cn^k$$

در حالت کلی اگر عبارت پیش را آن قدر گسترش دهیم تا n/b^m برابر یک شود، خواهیم داشت:

$$T(n)=a(a(\ldots T(n/b^m)+c(n/b^{m-1})^k)+\ldots)+cn^k$$

بیایید T(1) را برابر با ثابت c بگیریم (هر مقدار دیگری، نتیجه‌ی نهایی را تنها به اندازه‌ی یک ضریب ثابت تغییر می‌دهد). پس داریم:

$$T(n)=ca^m+ca^{m-1}b^k+ca^{m-2}b^{2k}+\ldots+cb^{mk}$$

که از آن نتیجه می‌شود:

$$T(n) = c\sum_{i=0}^{m} a^{m-i}b^{ik} = ca^m\sum_{i=0}^{m}\left(\frac{b^k}{a}\right)^i$$

این مجموع، یک تصاعد هندسی ساده است. (b^k/a) یا کوچک‌تر از یک است، یا برابر یک و یا بزرگ‌تر از آن.

حالت نخست: $a>b^k$

در این حالت ضریب تصاعد هندسی از یک کوچک‌تر است، پس این مجموع، حتا اگر تا بی‌نهایت ادامه یابد، به یک مقدار ثابت، همگراست. بنابراین: $T(n)=O(a^m)$. از آنجا که $m = \log_b^n$ پس داریم: $a^m = a^{\log_b^a} = n^{\log_b^a}$ (اگر از دو سمت تساوی دوم در مبنای b لگاریتم بگیریم، درستی آن ثابت می‌گردد). پس:

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

حالت دوم: $a=b^k$

در این حالت، ضریب تصاعد هندسی، یک است؛ بنابراین: $T(n)=O(a^m m)$. توجه کنید که از $a=b^k$ نتیجه گرفته می‌شود که $\log_b^a = k$ و $m = O(\log n)$. در نتیجه:

$$T(n) = O(n^k \log n)$$

حالت سوم: $a<b^k$

در این حالت، ضریب تصاعد هندسی بزرگ‌تر از یک است؛ پس روش معمولی برای جمع جملات یک تصاعد هندسی را به کار می‌گیریم. b^k/a را با F (یک ثابت) نشان می‌دهیم. از آنجا که جمله‌ی نخست، a^m است؛ پس داریم:

$$T(n) = a^m \frac{F^{m+1}-1}{F-1} = O(a^m F^m = O((b^k)^m) = O((b^m)^k) = O(n^k)$$

این سه حالت را در قضیه‌ی ۳-۴ خلاصه کرده‌ایم:

□ قضیه‌ی ۳-۴

حل رابطه‌ی بازگشتی $T(n)=aT(n/b)+cn^k$ که در آن a، b، c و k ثابت‌هایی صحیح هستند که $a\geq 1$، $b\geq 2$، $c>0$ و $k>0$؛ چنین است:

$$T(n) = \begin{cases} O(n^{\log_b^a}) & : a > b^k \\ O(n^k \log n) & : a = b^k \\ O(n^k) & : a < b^k \end{cases}$$

□

قضیه‌ی ۳-۴ برای برآورد زمان اجرای بسیاری از الگوریتم‌های «تقسیم‌وحل» به کار می‌رود، بنابراین باید آن را به خاطر سپرد. این قضیه در مرحله‌ی طراحی الگوریتم‌ها نیز می‌تواند بسیار سودمند باشد، زیرا برای برآورد زمان اجرا هم قابل‌استفاده است. در تمرین‌ها چند تعمیم از این رابطه آورده شده است.

۳-۵-۳ روابط بازگشتی با حافظه‌ی کامل

برای محاسبه‌ی هر جمله‌ی یک «رابطه‌ی بازگشتی با حافظه‌ی کامل»، به تمام (ونه برخی از) مقادیر پیشین آن نیازمندیم. یکی از ساده‌ترین این نوع روابط چنین است:

$$T(n) = c + \sum_{i=1}^{n-1} T(i) \qquad (۱۵-۳)$$

که در آن c مقداری ثابت است و T(1) را نیز داریم. می‌توانیم این مجموع را همانند حاصل‌جمع‌های پیشین حساب کنیم. برای این کار می‌کوشیم تا رابطه را به صورتی بنویسیم که بیش‌تر جملات آن حذف شوند. (گاهی به این روش، «حذف حافظه» گفته می‌شود.) در این مورد، T(n+1) را با T(n) مقایسه می‌کنیم. می‌دانیم:

$$T(n+1) = c + \sum_{i=1}^{n} T(i) \qquad (۱۶-۳)$$

اگر (۳-۱۵) را از (۳-۱۶) کم کنیم، داریم: T(n+1)-T(n)=T(n). پس T(n+1)=2T(n)؛ که از آن به سادگی می‌توان نتیجه گرفت: $T(n+1)=T(1)2^n$. (این ادعا برای T(1) درست است و چون هر بار مقدار جمله دو برابر می‌شود؛ پس اگر این ادعا برای T(n) درست باشد، برای T(n+1) نیز درست خواهد بود.)

هرچند این استدلال، درست به نظر می‌رسد، اما کاملاً نادرست است! مثلاً می‌توانیم T(1) را ۱ و c را ۵ بگیریم. در آن صورت می‌بینیم: T(2)=6≠2T(1). این استدلال نادرست، نمونه‌ی دیگری از اثبات استقرایی نادرست به دلیل نادیده گرفتن حالت پایه است. این خطا، ناشی از آن است که شاید T(1) با c حذف نشود؛ پس ادعای گفته‌شده برای T(2) برقرار نیست. چنان‌چه پارامتری (در اینجا c) در عبارت رابطه‌ی بازگشتی وجود داشته باشد، اما در حل نهایی اثری از آن دیده نشود، باید در درستی راه‌حل خود شک کنیم. حال، به حل درست مسأله توجه کنید: T(2)=T(1)+c (بنا به تعریف) و از آنجا می‌بینیم که اثبات گفته‌شده برای هر n>2 درست است. در نتیجه داریم: $T(n+1)=(T(1)+c)2^{n-1}$.

این رابطه‌ی بازگشتی بسیار ساده بود. به رابطه‌ی مهم دیگری توجه کنید که چندان ساده نیست. این رابطه در تحلیل حالت میانگین مرتب‌سازی سریع - که در بخش ۶-۴-۴ درباره‌ی آن بحث خواهیم کرد - مطرح می‌شود. رابطه‌ی بازگشتی این مسأله، رابطه‌ی (۳-۱۷) است:

$$T(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} T(i) \quad (\text{برای هر } n\geq 2)\ ,\ T(1)=0 \qquad (۱۷-۳)$$

روش جابه‌جایی و حذف جملات را به کار می‌گیریم. هدف ما حذف بیش‌تر جملات T(i) است. بیایید نگاهی به عبارت T(n+1) بیندازیم:

$$T(n+1)=(n+1)-1+\frac{2}{(n+1)}\sum_{i=1}^{n}T(i)\ \ (n\geq 2) \qquad (۱۸-۳)$$

برای راحتی کار، دو سمت (۳-۱۷) را در n و دو سمت (۳-۱۸) را در n+1 ضرب می‌کنیم:

$$nT(n)=n(n-1)+2\sum_{i=1}^{n-1}T(i)\,(n\geq 2) \qquad (۱۹-۳)$$

$$(n+1)T(n+1)=(n+1)n+2\sum_{i=1}^{n}T(i)\,(n\geq 2) \qquad (۲۰-۳)$$

حال (۳-۱۹) را از (۳-۲۰) کم می‌کنیم:

$$(n+1)T(n+1)-nT(n)=(n+1)n-n(n-1)+2T(n)$$
$$=2n+2T(n)\,(n\geq 2)$$

پس می‌توان نتیجه گرفت:

$$T(n+1)=\frac{n+2}{n+1}T(n)+\frac{2n}{n+1}\ \ (n\geq 2)$$

حل این رابطه‌ی بازگشتی، آسان‌تر از رابطه‌ی اصلی است. نخست به جای $\frac{2n}{n+1}$، ۲ قرار می‌دهیم زیرا این دو به هم بسیار نزدیکند:

$$T(n+1)\leq\frac{n+2}{n+1}T(n)+2\,(n\geq 2) \qquad (۲۱-۳)$$

با گسترش (۳-۲۱) داریم:

$$T(n)\leq 2+\frac{n+1}{n}\left[2+\frac{n}{n-1}\left[2+\frac{n-1}{n-2}\left[\ldots\frac{4}{3}\right]\right]\right]$$
$$=2\left[1+\frac{n+1}{n}+\frac{n+1}{n}.\frac{n}{n-1}+\frac{n+1}{n}.\frac{n}{n-1}.\frac{n-1}{n-2}+\ldots+\frac{n+1}{n}.\frac{n}{n-1}.\frac{n-1}{n-2}\ldots\frac{4}{3}\right]$$
$$=2\left[1+\frac{n+1}{n}+\frac{n+1}{n-1}+\frac{n+1}{n-2}+\ldots+\frac{n+1}{3}\right]$$
$$=2(n+1)\left[\frac{1}{n+1}+\frac{1}{n}+\frac{1}{n-1}+\ldots+\frac{1}{3}\right]$$
$$=2(n+1)(H(n+1)-1.5)$$

که در آن $H(n)=1+1.2+1.3+\ldots+1/n$، همان رشته‌ی توافقی است. این رشته، تقریبی ساده دارد که ما آن را ثابت نخواهیم کرد: $H(n)=\ln n+\gamma+O(1/n)$ ($\gamma=0.577..$، ثابت اویلر نام دارد) و با کمک آن به پاسخ T(n) می‌رسیم:

$$T(n)\leq 2(n+1)(\ln n+\gamma-1.5)+O(1)=O(n\log n)$$

۳-۶ چند رابطه‌ی سودمند

در این بخش، چندین رابطه را بدون اثبات ارائه می‌کنیم که در تحلیل الگوریتم‌ها به کار می‌آیند.

تصاعد حسابی

$$1+2+3+...+n=\frac{n(n+1)}{2} \qquad (۲۲-۳)$$

در حالت کلی‌تر اگر $a_n = a_{n-1} + c$ و c یک ثابت باشد؛ داریم:

$$a_1 + a_2 + a_3 + ... + a_n = \frac{n(a_n + a_1)}{2} \qquad (۲۳-۳)$$

تصاعد هندسی

$$1+2+4+...+2^n = 2^{n+1}-1 \qquad (۲۴-۳)$$

در حالت کلی‌تر اگر $a_n=ca_{n-1}$ و c ثابتی مخالف یک باشد؛ داریم:

$$a_1 + a_2 + a_3 + ... + a_n = a_1 \frac{c^n - 1}{c-1} \qquad (۲۵-۳)$$

اما اگر $0<c<1$، آنگاه جمع تصاعد هندسی نامتناهی عبارت است از:

$$\sum_{i=1}^{\infty} a_i = \frac{a_1}{1-c} \qquad (۲۶-۳)$$

جمع مربعات

$$\sum_{i=1}^{m} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad (۲۷-۳)$$

رشته‌ی توافقی

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + O(1/n) \qquad (۲۸-۳)$$

که در آن $\gamma = 0.577..$، ثابت اویلر است.

رابطه‌هایی در مورد لگاریتم

$$\log_b^a = \frac{1}{\log_a^b} \qquad (۲۹-۳)$$

$$\log_a^x = \frac{\log_b^x}{\log_b^a} \qquad (۳۰-۳)$$

$$b^{\log_b^x} = x \qquad (۳۱-۳)$$

$$b^{\log_a^x} = x^{\log_a^b} \qquad (۳۲-۳)$$

جمع لگاریتم‌ها

$$\sum_{i=1}^{n}\lfloor\log_2^i\rfloor=(n+1)\lfloor\log_2^n\rfloor-2^{\lfloor\log_2^n\rfloor+1}+2=\theta(n\log n) \qquad (۳۳-۳)$$

محاسبه‌ی حد بالای یک جمع با انتگرال

اگر f(x) تابعی صعودی و پیوسته باشد؛ داریم:

$$\sum_{i=1}^{n}f(i)\le\int_{x=1}^{x=n+1}f(x)dx \qquad (۳۴-۳)$$

تقریب استرلینگ

$$n!=\sqrt{2\pi n}\left[\frac{n}{e}\right]^n(1+O(1/n)) \qquad (۳۵-۳)$$

از این تقریب می‌توان نتیجه گرفت: $\log_2(n!)=\theta(n\log n)$.

۳-۷ خلاصه

Neils Bohr یک بار گفت: «پیش‌بینی کار سختی است؛ به ویژه آن که بخواهیم این کار را درباره‌ی آینده انجام دهیم». اگرچه پیش‌بینی رفتار یک الگوریتم به این سختی نیست، اما چندان آسان هم نیست. «تقریب و تخمین»، شیوه‌ی اصلی ما برای پیش‌بینی رفتار یک الگوریتم است؛ یعنی جزئیات الگوریتم را نادیده می‌گیریم و تنها مهم‌ترین ویژگی‌های آن را بررسی می‌کنیم. از این نظر، نماد O مفید است، اما هرگز نباید فراموش کرد که این نماد تنها برآوردی اولیه است. از طرفی، تحلیل الگوریتم نباید آن قدر دشوار باشد که طراح الگوریتم از انجام تحلیل منصرف شود. برای تعیین کارایی یک الگوریتم، معیارهایی لازم است.

هنگام تحلیل الگوریتم‌ها به رابطه‌های بازگشتی، بسیار زیاد برخورد می‌کنیم؛ به ویژه اگر الگوریتم به روش بازگشتی طراحی شده باشد. نخستین کاری که باید با این دسته از رابطه‌ها انجام داد، نگاهی به چند جمله‌ی آغازین است. بدین ترتیب، دید بهتری نسبت به رفتار رابطه‌ی بازگشتی پیدا می‌کنیم که البته به هیچ وجه کافی نیست. بررسی چند جمله‌ی آغازین به حدس زدن پاسخ کمک می‌کند. روش دیگر، چنان که در بخش ۳-۵-۲ دیدیم، این است که چندین و چند بار رابطه‌ی بازگشتی را باز کنیم. یک گام اولیه‌ی خوب برای حل رابطه‌های بازگشتی، حدس جواب و سپس بررسی درستی آن است. باید هشیار باشیم که مبادا حدی بالاتر را حدس بزنیم، که اگرچه درست است، اما از پاسخ واقعی بدبینانه‌تر است. شیوه‌های دیگری هم برای حل رابطه‌های بازگشتی وجود دارد. خوشبختانه الگوریتم‌هایی که در عمل با آن‌ها روبه‌رو می‌شویم، بیش‌تر به یکی از چند دسته‌ی محدودی که پیش‌تر در این فصل بررسی

کرده‌ایم، منتهی می‌شوند. معمولاً در گام‌های نخستِ حل رابطه‌ی بازگشتی می‌توانیم برای n مقداری ویژه در نظر بگیریم؛ مثلاً آن را توانی از ۲ فرض کنیم.

مراجعی برای مطالعه‌ی بیش‌تر

ایده‌ی تحلیل مجانبی در سال‌های ابتدایی دهه‌ی ۱۹۷۰ مطرح، ولی با مقاومت‌هایی روبه‌رو شد. امروزه این ایده، معیار اصلی برای سنجش کارایی الگوریتم‌هاست. چندین کتاب، با موضوع ریاضیات گسسته و ترکیبیات وجود دارند که روش‌های مورد استفاده برای محاسبه‌ی جمع‌ها، حل روابط بازگشتی و دیگر عبارت‌های سودمند برای تحلیل الگوریتم‌ها را بررسی می‌کنند. Brualdi [۱۹۷۷]، Bavel [۱۹۸۲]، Roberts [۱۹۸۴]، Graham، Knuth و Patashnik [۱۹۸۹] چند کتاب از این دست هستند. کتاب‌های اندکی تماماً به تحلیل الگوریتم‌ها اختصاص یافته‌اند. Knuth [۱۹۷۳a]، موضوعات و منابع فراوانی را گرد آورده است. چند بررسی جامع و کتاب دیگر عبارتند از: Greene و Knuth [۱۹۸۲]، Lueker [۱۹۸۰]، Purdom و Brown [۱۹۸۵a]، Flajolet و Vitter [۱۹۸۷] و Hofri [۱۹۸۷]. (توضیح مترجمان: منظور از بررسی جامع، مقاله‌ای است که ارائه‌دهنده‌ی مجموعه‌ی کارهای انجام‌شده در یک زمینه باشد.)

Knuth [۱۹۷۶]، درباره‌ی نمادهای شبیه O است. می‌توان روش‌های دیگری را نیز برای حل روابط بازگشتی در Lueker [۱۹۸۰] و Bentley، Haken و Saxe [۱۹۸۰]یافت. (حل تمرین ۳-۲۳ و ۳-۲۴ در مرجع دوم آمده است.) Tarjan [۱۹۸۵] پیچیدگی را در حالت سرشکن‌شده بررسی می‌کند که شیوه‌ای دقیق و زیبا برای تحلیل زمان اجرای الگوریتم‌هایی خاص است؛ یعنی اگر بخش خاصی از الگوریتم چندین بار و هر بار با زمان اجرایی متفاوت اجرا شود، آنگاه به جای آن که هر بار بدترین حالت را در نظر بگیریم، بدترین حالت کل را در نظر می‌گیریم. (چون ممکن است در حالی که یک عمل در بدترین حالت است، عمل دیگر نتواند در بدترین حالت قرار گیرد - مترجمان) رابطه‌ی بازگشتی تمرین ۳-۱۹، از Manber [۱۹۸۶] و تمرین ۳-۲۱ از Purdom و Brown [۱۹۸۵a] گرفته شده است.

تمرین‌های آموزشی

۳-۱ ثابت کنید اگر P(n) یک چند جمله‌ای از n باشد، آنگاه O(log(P(n)))=O(logn).

۳-۲ ثابت کنید اگر f(n)=o(g(n)) آنگاه f(n)=O(g(n)). آیا از f(n)=O(g(n)) نیز می‌توان f(n)=o(g(n)) را نتیجه گرفت؟

۳-۳ به یاری قضیه‌ی ۳-۱ ثابت کنید:

$$n\left(\log_3^n\right)^5 = O(n^{1.2})$$

۴-۳ به کمک قضیه‌ی ۳-۱ ثابت کنید برای همه‌ی ثابت‌های a,b>0 داریم:

$$\left(\log_2^n\right)^a = O(n^b)$$

۵-۳ هر جفت از توابع داده‌شده را از نظر مرتبه‌ی بزرگی با هم مقایسه کنید. در هر مورد بگویید کدام یک از روابط f(n)=O(g(n))، f(n)=Ω(g(n)) و یا $f(n)=\theta(g(n))$ برقرار است؟

الف- $f(n)=100n+\log n$ ، $g(n)=n+(\log n)^2$

ب- $f(n)=\log n$ ، $g(n)=\log(n^2)$

پ- $f(n)=\frac{n^2}{\log n}$ ، $g(n)=n(\log n)^2$

ت- $f(n)=(\log n)^{\log n}$ ، $g(n)=\frac{n}{\log n}$

ث- $f(n)=n^{1/2}$ ، $g(n)=(\log n)^5$

ج- $f(n)=n2^n$ ، $g(n)=3^n$

۶-۳ این رابطه‌ی بازگشتی را حل کرده، پاسخ دقیق آن را ارائه کنید:

T(n)=T(n-1)+n/2, T(1)=1

۷-۳ این رابطه‌ی بازگشتی را حل کرده، پاسخ دقیق آن را بیابید:

T(n)=8T(n-1)-15T(n-2), T(1)=1, T(2)=4

۸-۳ ثابت کنید این رابطه‌ی بازگشتی در رابطه‌ی $T(n)=O(n\log^2 n)$ صدق می‌کند:

$$T(n)=2T(\lfloor n/2 \rfloor)+2n\log_2^n, T(2)=4$$

۹-۳ این رابطه‌ی بازگشتی، بیانگر زمان اجرای الگوریتمی بازگشتی برای ضرب ماتریس‌هاست (Pan [۱۹۷۸]). «زمان اجرای مجانبی» این الگوریتم چیست؟

T(n)=143640T(n/70)+O(n^2), T(1)=1

۱۰-۳ اشتباه این تحلیل را پیدا کنید: A را الگوریتمی بگیرید که روی درخت‌های دودویی کامل عمل می‌کند. (درخت دودویی کامل، درختی است دودویی که تمام برگ‌هایش در عمقی یکسان قرار دارند.) فرض کنید الگوریتم A، بر روی هر برگ درخت، کاری از O(k) انجام می‌دهد. K پارامتری وابسته به مقدار اطلاعات ذخیره‌شده در برگ (اما مستقل از خود درخت) است. c نیز زمان ثابتی است که الگوریتم روی هر گره داخلی صرف می‌کند. ادعا می‌کنیم که زمان اجرای کل الگوریتم از O(k) است.

برهان اشتباه: اثبات با استقرا روی n (تعداد گره‌های درخت) انجام می‌شود. اگر n=1، آنگاه روشن است که تعداد کل مراحل از O(k) است. فرض کنید این ادعا برای تمام درخت‌های دودویی کاملی که کم‌تر از n گره دارند، درست باشد و درختی را با n گره در نظر بگیرید. چنین درختی از یک ریشه و دو زیردرخت، هر یک با (n-1)/2 گره تشکیل می‌شود. بنا به

فرض استقرا زمان اجرا برای هر یک از زیردرخت‌ها از O(k) است. از این رو، زمان اجرای کل O(k)+O(k)+c و در نتیجه از O(k) خواهد بود و برهان، کامل می‌گردد.

۱۱-۳ این «رابطه‌ی بازگشتی با حافظه‌ی کامل» را حل کنید:

$$T(n) = \max_i \{T(i)\}, T(1) = 1$$

۱۲-۳ این «رابطه‌ی بازگشتی با حافظه‌ی کامل» را حل کنید:

$$T(n) = n + \sum_{i=1}^{n-1} T(i), T(1) = 1$$

۱۳-۳ به یاری رابطه‌ی (۳-۳۴) ثابت کنید برای هر عدد صحیح مثبت k داریم:

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

۱۴-۳ با کمک رابطه‌ی (۳-۳۴) ثابت کنید برای هر عدد صحیح مثبت k داریم:

$$\sum_{i=1}^{n} i^k \log_2^i = O(n^{k+1} \log n)$$

تمرین‌های خلاقانه

۱۵-۳ یک مثال نقض برای این ادعا بیابید: اگر f(n)=O(s(n)) و g(n)=O(r(n))، آنگاه f(n)-g(n)=O(s(n)-r(n)).

۱۶-۳ یک مثال نقضی برای این ادعا بیابید: اگر f(n)=O(s(n)) و g(n)=O(r(n))، آنگاه f(n)/g(n)=O(s(n)/r(n)).

۱۷-۳ ✯ دو تابع f(n) و g(n) را چنان بیابید که هر دو صعودی باشند و f(n)≠O(g(n)) و g(n)≠O(f(n)).

۱۸-۳ این رابطه‌ی بازگشتی را در نظر بگیرید:

T(n)=2T(n/2)+1, T(2)=1

می‌کوشیم ثابت کنیم T(n)=O(n) (تنها به توان‌های ۲ توجه می‌کنیم). حدس می‌زنیم که T(n)≤cn برای یک ثابت c (که مقدارش را نمی‌دانیم) برقرار باشد و سپس cn را در رابطه‌ی بازگشتی قرار می‌دهیم (بخش ۳-۵-۱ را ببینید). باید نشان دهیم که cn≥2c(n/2)+1؛ اما روشن است که چنین ادعایی درست نیست. راه‌حل درست را پیدا کنید (می‌توانید n را توانی از ۲ در نظر بگیرید) و توضیح دهید چرا روش گفته‌شده به شکست انجامید.

۱۹-۳ S(n) در این رابطه‌ی بازگشتی صدق می‌کند؛ رفتار مجانبی آن را بیابید:

$$S(mn) \le cm \log_2^m S(n) + O(mn), S(2) = 1$$

در این رابطه‌ی بازگشتی، m و c پارامترهایی ثابت هستند. (پاسخ باید تابعی از n، m و c باشد.)

۳-۲۰ ثابت کنید حل مجانبی این رابطه‌ی بازگشتی، $T(n)=O(d^n)$ به ازای یک مقدار ثابت مانند d است:

$$T(n)=2T(n-c)+k$$

دراین رابطه‌ی بازگشتی، c و k هر دو ثابت‌هایی صحیح هستند.

۳-۲۱ این رابطه‌ی بازگشتی در آن دسته از الگوریتم‌های «تقسیم‌وحل» به کار می‌رود که مسأله، به دو بخش با اندازه‌های نابرابر تقسیم شود:

$$T(n)=\sum_{i=1}^{k} a_i T(n/b_i)+cn$$

همه‌ی a_iها و b_iها ثابتند و رابطه‌ی $1-\sum_{i=1}^{k} a_i/b_i > 0$ برای آن‌ها برقرار است. رفتار مجانبی این رابطه‌ی بازگشتی را (به روش حدس زدن و بررسی درستی آن حدس) پیدا کنید.

۳-۲۲ دو رابطه‌ی بازگشتی داده‌شده را حل کنید. حتا اگر رفتار مجانبی T(n) را هم بیابید، کافی است.

الف- $T(n)=4T(\lceil\sqrt{n}\rceil)+1, T(2)=1$

ب- $T(n)=2T(\lceil\sqrt{n}\rceil)+2n, T(2)=1$

(راه‌نمایی: متغیر دیگری را جای‌گزین n کنید.)

۳-۲۳ ثابت کنید حل این رابطه‌ی بازگشتی

$$T(n)=kT(n/2)+f(n),\ T(1)=c$$

عبارت است از:

$$T(n)\sim n^{\log k}(c+g(2)+g(4)+\ldots+g(n))$$

که در آن $g(m)=f(m)/m^{\log k}$. می‌توانید n را توانی از ۲ بگیرید. (این راه‌حل از راه‌حل ارائه‌شده در بخش ۳-۵-۲ کلی‌تر است، چون برای هر تابع f(n) معتبر است.)

۳-۲۴ ثابت کنید حل این رابطه‌ی بازگشتی

$$T(n)=kT(n/d)+f(n),\ T(1)=c$$

عبارت است از:

$$T(n)=n^{\log_d^k}(c+g(d)+g(d^2)+\ldots+g(n))$$

که در آن $g(m)=f(m)/m^{\log k}$. بازهم می‌توانید n را توانی از ۲ بگیرید.

۳-۲۵ رفتار مجانبی تابع T(n) را پیدا کنید:

$$T(n)=T(n/2)+T(\lfloor\sqrt{n}\rfloor+n, T(1)=1, T(2)=2$$

می‌توانید n را توانی از ۲ بگیرید.

۳-۲۶ رفتار مجانبی تابع T(n) را بیابید:

$$T(n)=T(n/2)+\sqrt{n}, T(1)=1$$

۳-۲۷ رفتار مجانبی تابع T(n) را پیدا کنید:

$$T(n)=T(n/2)+T\left(\left\lfloor \frac{n}{\log_2^n} \right\rfloor\right)+n\,(n>2), T(1)=1, T(2)=2$$

می‌توانید n را توانی از ۲ بگیرید.

۳-۲۸ حل این رابطه‌ی بازگشتی را بیابید. کافی است رفتار مجانبی T(n) را پیدا کنید و دلیلی قانع‌کننده ارائه دهید که تابع پاسخ یا f(n) (که آن را یافته‌اید) در $f(n)=\theta(T(n))$ صدق می‌کند:

$$T(n)=2T\left(\left\lfloor \frac{n}{\log_2^n} \right\rfloor\right)+3n\,(n>2), T(1)=1, T(2)=2$$

۳-۲۹ اگرچه معمولاً ارزیابی روابط بازگشتی، با توان‌های ۲ کافی است، اما همیشه این‌گونه نیست. این رابطه‌ی بازگشتی را در نظر بگیرید:

$$T(n)=\begin{cases} T(n/2)+1 & \text{اگر n زوج باشد:} \\ 2T((n-1)/2) & \text{اگر n فرد باشد:} \end{cases}$$

T(1)=1

الف- ثابت کنید حل این رابطه‌ی بازگشتی برای توان‌های ۲ عبارت است از $T(2^k)=k+1$ (یعنی برای توان‌های ۲ داریم: T(n)=O(log n)).

ب- ثابت کنید برای تعداد نامتناهی n داریم: T(n)=Ω(n). چرا در این مورد، فرض رایجِ یکسانی رفتار برای توان‌های ۲ و عددهای دیگر، درست نیست؟ در این مورد بحث کنید.

۳-۳۰ با به کار بردن رابطه‌ی (۳-۳۴) ثابت کنید $\sum_{i=1}^{n}\left\lceil \log_2^{(n/i)} \right\rceil$ از O(n) است.

۳-۳۱ حاصل دقیق این عبارت را بیابید:

$$\sum_{i=1}^{n}\left\lceil \log_2^{(n/i)} \right\rceil$$

می‌توانید n را توانی از ۲ بگیرید.

۳-۳۲ تعریف عددهای فیبوناچی، یعنی F(0)=0، F(1)=1 و F(n+2)=F(n+1)+F(n) را به مقدارهای منفی هم تعمیم دهید (مثلاً F(-1)=1، F(-2)=-1 و...).

فرض کنید: G(n)=F(-n). رابطه‌ای بازگشتی برای G(n) بنویسید و راهی برای حل آن پیش‌نهاد کنید.

۳-۳۳ اگر F(n) و G(n) را مانند تمرین پیش تعریف کنیم؛ ثابت کنید: $G(n)=(-1)^{n+1}F(n)$.

فصل ۴

نگاهی کوتاه به ساختمان‌های داده‌ای

علم همان بدیهیات است؛ پس از دسته‌بندی و آموزش.
T.H. Huxley، ۱۸۷۸

از روشن‌فکران بیزارم. آنان از بالا به پایین می‌رسند؛ من از پایین به بالا.
Frank Lloyd Wright (۱۹۵۹-۱۸۶۹)

۴-۱ آشنایی

ساختمان‌های داده‌ای، اجزای سازنده‌ی الگوریتم‌های رایانه‌ای هستند. طراحی یک الگوریتم همانند طراحی یک ساختمان است. اتاق‌های یک ساختمان باید به گونه‌ای در کنار هم قرار گیرند که برای کاربرد مورد نظر بیش‌ترین کارایی را داشته باشند. برای این کار دانستن عمل‌کرد، بهره‌وری، شکل و زیبایی، به تنهایی کافی نیست، بلکه به دانش کامل روش‌های ساختمان‌سازی نیازمندیم. ممکن است کارایی دل‌خواه ما قرار دادن اتاق در هوا، بین زمین و آسمان باشد، اما چنین کاری شدنی نیست یا ممکن است برخی ایده‌های شدنی، بسیار گران تمام شوند. به همین ترتیب، طراحی یک الگوریتم باید بر مبنای درک کاملی از روش‌های ساختمان‌داده و هزینه‌ی هر یک از این روش‌ها باشد.

در این فصل تنها ساختمان‌های داده‌ای پایه را که در کتاب به کار رفته‌اند، بررسی می‌کنیم. نمی‌خواهیم درباره‌ی تمام ساختمان‌های داده‌ای توضیح جامعی دهیم؛ چرا که برای این کار (دست‌کم) به یک کتاب کامل نیازمندیم و بدون شک کتاب‌های خوبی در این زمینه وجود دارند. انتظار داریم که بیش‌تر خوانندگان کم و بیش با ساختمان‌های داده‌ای آشنا باشند. هدف عمده‌ی این فصل، نگاهی سریع به موضوع است.

یک روش مناسب در مطالعه‌ی ساختمان‌های داده‌ای، بررسی «داده‌گونه‌های مجرد» است. عموماً هنگام نوشتن برنامه، باید نوع داده‌های به‌کاررفته (صحیح، اعشاری، کاراکتری و ...) را نیز مشخص کنیم؛ اما گاهی تعیین نوع داده‌ها در طراحی الگوریتم، موضوع مهمی نیست. مثلاً ممکن است بخواهیم یک صف ترتیبی از عناصر را نگه‌داری کنیم. اعمالی که در اینجا مورد نیازند، درج یک عنصر در صف و

حذف آن از صف هستند. هنگام برداشتن عنصرها از صف باید آن‌ها را به همان ترتیبی که به صف افزوده شده‌اند، حذف کنیم. طراحی الگوریتم‌های این اعمال (یعنی درج یا حذف یک عنصر) بدون تعیین نوع داده‌ی عنصر، راحت‌تر و کلی‌تر است. بهتر است تنها، اعمال مورد نیاز را مشخص کنیم؛ مثلاً در این مورد خاص، به داده‌گونه‌ی مجردی که از اعمال گفته‌شده پشتیبانی می‌کند، صف ترتیبی می‌گوییم. مهم‌ترین بخش هر داده‌گونه‌ی مجرد، فهرست اعمال مورد نیاز آن است. مثال دیگری از داده‌گونه‌ی مجرد، صفی است که عناصر آن، اولویت هم دارند؛ یعنی حذف آن‌ها بنا به ترتیب درجشان نیست، بلکه برحسب اولویت آن‌هاست. به عبارت دیگر، نخستین عنصری که در هر گام از عمل حذف، کنار گذاشته می‌شود، عنصری است که در میان عناصر صف، بیش‌ترین اولویت را داشته باشد. این داده‌گونه‌ی مجرد، صف اولویت نام دارد. در اینجا هم، نوع داده‌ای عناصر را مشخص نمی‌کنیم. (حتا نیازی نیست که نوع داده‌ای اولویت‌ها را تعیین کنیم؛ تنها لازم است آن اولویت‌ها ترتیب کلی داشته باشند و ما بتوانیم ترتیب آن‌ها را تعین کنیم.)

با تمرکز بر عمل‌کرد یک ساختمان‌داده و چشم‌پوشی از جزئیات پیاده‌سازی آن برای یک مسأله‌ی خاص، به طراحی الگوریتم عمومیت بیش‌تری خواهیم بخشید. برای نمونه، روش‌های پیاده‌سازی یک صف اولویت را، تا جای ممکن، مستقل از نوع دقیق داده در نظر می‌گیریم. اگر در جایی ببینیم که یک ساختمان‌داده‌ی مجرد، متناسب با نیازهای ماست، آن را به کار می‌بریم. با تعریف داده‌گونه‌های مجرد می‌توانیم پیمانه‌ای‌تر، الگوریتم‌ها را طراحی کنیم.

۴-۲ ساختمان‌های داده‌ای پایه

۴-۲-۱ عناصر

از «عنصر» در تمام کتاب، به عنوان نامی کلی برای هر نوع نامشخص داده سود خواهیم جست. یک عنصر، ممکن است عددی صحیح، مجموعه‌ای از اعداد صحیح، پرونده‌ای متنی یا ساختمان‌داده‌ی دیگری باشد. این واژه را هنگامی که مبحث مورد نظر، مستقل از نوع داده باشد، به کار می‌بریم. برای نمونه، الگوریتم‌های مرتب‌سازی را در نظر بگیرید. اگر گام‌هایی که الگوریتم برمی‌دارد، تنها مقایسه‌ی عناصر یا جابه‌جا کردن آن‌ها باشد؛ آنگاه می‌توان از آن الگوریتم، هم برای مرتب‌سازی اعداد صحیح و هم برای مرتب‌سازی نام‌ها (یعنی رشته‌های کاراکتری) سود جست. پیاده‌سازی این دو الگوریتم (یعنی برنامه‌ی آن‌ها) احتمالاً قدری متفاوت از یکدیگر است، اما هر دو از ایده‌های یکسانی بهره می‌برند. از آنجایی که ما بر ایده‌های طراحی الگوریتم (و نه پیاده‌سازی آن‌ها) تمرکز می‌کنیم، پس چشم‌پوشی از نوع عنصر عاقلانه است.

تنها سه پیش‌فرض برای عناصر در نظر می‌گیریم:

۱- می‌توان عناصر را با هم مقایسه کرد.

۲- همه‌ی عناصر به مجموعه‌هایی با ترتیب کلی تعلق دارند و از هر دو عنصر نابرابر، یکی «کوچک‌تر» از دیگری است. (با این تعبیر، اعضای برخی مجموعه‌ها مانند مجموعه‌ی اعداد مختلط، ترتیب‌پذیر نیستند - مترجمان) تا هنگامی که با مجموعه‌های ترتیب‌ناپذیر (مانند اعداد مختلط) سر و کار نداریم، به تعریف دقیق رابطه‌ی «کوچک‌تری» نخواهیم پرداخت.

۳- از یک عنصر می‌توان نسخه‌برداری کرد (یعنی کپی گرفت).

فرض می‌کنیم انجام هر یک از اعمال گفته‌شده نیازمند یک واحد از زمان است. هرچند که در عمل، این واحد زمانی، متناسب با اندازه‌ی واقعی عنصر است، اما نگاه ما به این اعمال به گونه‌ای است که زمان انجام آن‌ها را مقداری ثابت می‌گیریم. با این‌که الگوریتم‌های مورد نظر می‌توانند برای ساختمان‌های داده‌ای پیچیده هم به کار روند، اما اغلب ساده‌تر است که عناصر را اعداد صحیح در نظر بگیریم.

۴-۲-۲ آرایه‌ها

آرایه را می‌توان سطری از عناصر هم‌نوع در نظر گرفت. اندازه‌ی یک آرایه، تعداد عناصر آن است. اندازه‌ی آرایه باید ثابت باشد. از آنجا که اندازه‌ی آرایه ثابت است و عناصرش نیز هم‌نوع هستند، پس مقدار حافظه‌ی لازم برای ذخیره‌ی آن از پیش آشکار است. برای مثال اگر عناصر آرایه نام‌هایی ۸ کاراکتری باشند و هر کاراکتر نیازمند یک بایت حافظه بوده، طول آرایه هم ۱۰۰ باشد؛ برای ذخیره‌ی آن آرایه به ۸۰۰ بایت حافظه نیاز داریم. عناصر آرایه همواره پشت‌سرهم ذخیره می‌شوند. اگر اولین بایت یک آرایه در محل x از حافظه ذخیره شود، آنگاه محل ذخیره‌ی kامین بایت آرایه خانه‌ی x+k-1 از حافظه خواهد بود. پس به آسانی می‌توان محل هر عنصر آرایه در حافظه را محاسبه کرد. در مثال پیش، اگر محل شروع آرایه ۱۰۰۰۰ باشد، پنجاه و پنجمین نام از بایت چهارصد و سی و سوم آغاز می‌شود، یعنی محل ۱۰۴۳۲ از حافظه. (در این مثال، فرض بر این است که محل‌های حافظه را بایت به بایت می‌شماریم، اما اگر شمارش به صورت دیگری هم انجام شود، به آسانی می‌توان محاسبات را تغییر داد و محل مورد نظر را به درستی پیدا کرد.)

آرایه‌ها، ساختمان‌داده‌هایی بسیار کارآمد و رایج هستند. به هر یک از عناصر آرایه می‌توان در زمانی ثابت دست‌رسی داشت. آن دسته از طراحان الگوریتم که زبان‌های سطح بالا را به کار می‌برند، به ندرت نگران محاسبه‌ی محل عناصر هستند - چرا که کامپایلر این کار را انجام می‌دهد. به عنوان یک قانون سرانگشتی، اگر آرایه قابل استفاده است، ساختمان‌داده‌ی دیگری به جای آن به کار نبرید. ضعف اصلی آرایه‌ها، محدودیت‌های آن‌هاست. نمی‌توان یک آرایه را برای نگه‌داری عنصرهایی به کار برد که

نوع داده‌ای آن‌ها متفاوت است. اندازه‌ی آرایه‌ها نیز پس از تعریف، قابل‌تغییر نیست. دو بخش بعدی برای برخورد با این محدودیت‌هاست.

۴-۲-۳ رکوردها

رکورد نیز شبیه آرایه است، با این تفاوت که عناصر آن را هم‌نوع در نظر نمی‌گیریم. پس یک رکورد مجموعه‌ای از چند عنصر گوناگون است که به شیوه‌ی مشخصی کنار یکدیگر قرار گرفته‌اند. اندازه‌ی حافظه‌ی مورد نیاز برای ذخیره‌ی یک رکورد (همانند آرایه) روشن است. می‌توان به هر یک از عناصر رکورد در زمانی ثابت دست‌رسی پیدا کرد. این کار با نگه‌داری رکورد به صورت یک آرایه انجام می‌شود، به گونه‌ای که هر عنصر این آرایه از محل ویژه‌ی خود آغاز گردد. نگه‌داری یک رکورد به صورت آرایه، برای دست‌رسی به عناصر آن در زمانی ثابت ضروری است. دست‌رسی، با مراجعه به محل عنصر در آرایه‌ی گفته‌شده انجام می‌شود. کامپایلر، خود، برنامه‌ای را که برای مدیریت آرایه لازم است، ایجاد خواهد کرد. مثلاً ممکن است رکوردی شامل ۲ عدد صحیح، ۳ آرایه‌ی ۲۰تایی از اعداد صحیح، ۴ عدد صحیح دیگر و ۲ نام ۱۲ کاراکتری باشد. (توجه کنید که هر دو نوع آرایه‌ی مختلفِ موجود در این رکورد، خود، عنصرهایی از رکورد هستند.) این رکورد در شکل ۴-۱ تعریف شده است. در آرایه‌ی ذخیره‌کننده‌ی این رکورد، مکان شروع عنصرها با توجه به ترتیب نسبی آن‌ها و محل شروع آرایه در حافظه، قابل محاسبه است. بنابراین اگر اعداد صحیح ۴ بایت جا بگیرند، Int6 که نهمین عنصر رکورد است، از بایت ۲۶۱ (یعنی 4×2+4×20×3+4×3+1) آغاز می‌شود. از آنجایی که اندازه‌ی همه‌ی عنصرهای رکورد مشخص است، پس می‌توان محل هر عنصر را در زمانی ثابت حساب کرد. مانند آرایه، فضای ذخیره‌سازی رکورد نیز همواره ترتیبی است و باز مانند آرایه‌ها نمی‌توان پس از تعریف، عنصری را به رکورد افزود.

```
record example1
begin
        Int1: integer;
        Int2: integer;
        Ar1: array [1..20] of integer;
        Ar2: array [1..20] of integer;
        Ar3: array [1..20] of integer;
        Int3: integer;
        Int4: integer;
        Int5: integer;
        Int6: integer;
        Name1: array [1..12] of character;
        Name2: array [1..12] of character;
end
```

شکل ۴-۱ تعریف یک رکورد

۴-۲-۴ لیست‌های پیوندی

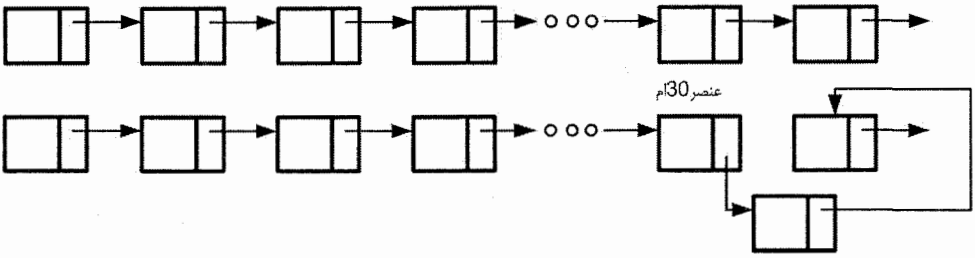
در بسیاری از الگوریتم‌ها ممکن است تعداد عناصر، هنگام اجرا تغییر کند. برای اطمینان از کافی بودن فضای ذخیره‌سازی می‌توان همه‌ی عناصر را به صورت آرایه‌هایی (یا رکوردهایی) بسیار بزرگ تعریف کرد. اگرچه با این کار، مسأله در صورت وجود حافظه‌ی کافی حل می‌شود، اما بسیار محتمل است که با کم‌بود حافظه مواجه شویم. (زمان بروز این مشکل هم مشخص نیست.) گاهی هم لازم است عمل حذف یا درج در وسط عناصر انجام شود که اگر آرایه به کار برده باشیم، ناچاریم بسیاری از عناصر را جابه‌جا کنیم. این ناکارآمدی، ناشی از ذات ترتیبی آرایه‌هاست؛ پس حذف و درج آزادانه در آرایه‌های بزرگ، بسیار پرهزینه است. در این مواقع به ساختمان‌های داده‌ای پویا نیازمندیم. چون این ساختارها در کتاب، بسیار به کار خواهند رفت، پس لازم است با آن‌ها آشنا شویم.

لیست‌های پیوندی ساده‌ترین نوع ساختمان‌های داده‌ای پویا هستند. فرض کنید فهرستی از عناصر داریم و می‌خواهیم عمل درج عنصر تازه یا حذف عنصر موجود را به گونه‌ای کارآمد انجام دهیم. ایده‌ی اصلیِ انجام این کار، نمایش جداگانه‌ی عنصرها و پیوند دادن آن‌ها با بهره‌گیری از اشاره‌گرها، به جای نمایش ترتیبی آرایه است. اشاره‌گر، متغیری است که نشانی یک عنصر را در خود نگه‌داری می‌کند. لیست پیوندی، لیستی از دوتایی‌هاست که هر یک از آن‌ها از یک عنصر و یک اشاره‌گر تشکیل شده‌اند، بدین‌ترتیب که هر اشاره گر، آدرس دوتایی بعدی را در خود ذخیره کرده است. هر یک از این دوتایی‌ها با یک رکورد نمایش داده می‌شوند. می‌توان یک لیست پیوندی را با پی‌گیری نشانی موجود در اشاره‌گرهای آن پویش کرد. چنین پویشی حتماً خطی است؛ یعنی نمی‌توان مستقیماً به یک عنصر دست‌رسی داشت، بلکه باید عناصر لیست را به ترتیب بپیماییم تا به عنصر مورد نظر برسیم.

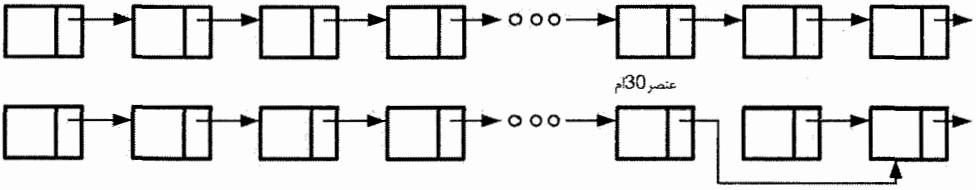
لیست‌های پیوندی دو ضعف دارند: نیاز به حافظه‌ی بیش‌تر (همراه با هر عنصر، یک اشاره‌گر اضافی هم وجود دارد) و ترتیبی بودن آن‌ها (مثلاً اگر بخواهیم عنصر ۳۰ام را ببینیم، باید از ابتدای لیست پیوندی شروع کنیم و ۲۹ عنصر را یکی یکی بررسی کنیم تا به عنصر ۳۰ام برسیم، در حالی که در آرایه‌ها می‌توانیم با محاسبه‌ای ساده، یک‌راست به عنصر ۳۰ام برسیم). از سوی دیگر، لیست‌های پیوندی یک برتری عمده هم دارند. فرض کنید عنصر ۳۰ام را یافته‌ایم و حال قصد داریم عنصر ۳۱ام را به لیست بیفزاییم[۱]. تنها کار لازم، قرار دادن نشانی عنصر ۳۱امِ پیشین در اشاره‌گر عنصر ۳۱ام فعلی (این نشانی در اشاره‌گر عنصر ۳۰ام ذخیره شده است) و به دنبال آن تنظیم اشاره‌گر عنصر ۳۰ام است (به گونه‌ای که به عنصر ۳۱ام تازه اشاره کند) (شکل ۴-۲ را ببینید). پس تنها دو عمل لازم است، در

۱- هنگام شمردن، اعداد ترتیبی را به کار می‌بریم. پس صحبت درباره‌ی عنصر ۳۱ام پیشین و فعلی، گیج‌کننده است. غالباً از a_{30} برای نمایش عنصری که پس از (after) عنصر ۳۰ام افزوده می‌شود، بهره می‌گیریم. این نمادگذاری سبب مشکلات بسیاری می‌شود. اگر دوباره پس از عنصر ۳۰ام، عنصری دیگر اضافه کنیم، نمادگذاری آن چه خواهد شد؟ a_{30} (!). این موضوع، مثال خوبی از نیاز به ساختمان‌های داده‌ای پویاست.

صورتی که برای انجام چنین کاری در آرایه‌ها، جابه‌جایی همه‌ی عناصرِ پس از عنصر ۳۰ام لازم بود. در لیست‌های پیوندی عمل حذف هم ساده است؛ مثلاً اگر بخواهیم عنصر ۳۱ام را حذف کنیم، به سادگی با قرار دادن نشانیِ درون اشاره‌گر عنصر ۳۱ام در اشاره‌گر عنصر ۳۰ام، ترتیبی می‌دهیم که عنصر ۳۰ام به عنصر ۳۲ اشاره کند (شکل ۴-۳ را ببینید). باز هم تنها به دو عمل نیازمندیم.



شکل ۴-۲ افزودن عنصری تازه به لیست پیوندی



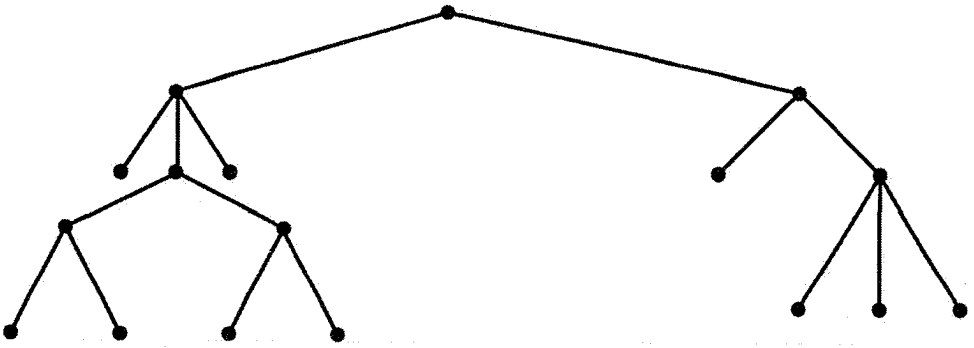
شکل ۴-۳ برداشتن یک عنصر از لیست پیوندی

در بحث حذف و اضافه کردن عناصر، در لیست‌های پیوندی از چندین موضوع مهم چشم‌پوشی کردیم. این موضوعات سبب می‌شوند که پیاده‌سازی لیست‌های پیوندی اندکی پیچیده‌تر گردد. مسأله‌ی اصلی، چگونگی شناسایی انتهای لیست است. معمولاً برای برطرف کردن این مشکل، نشانی ویژه‌ای به نام nil به کار برده می‌شود. nil اشاره‌گری به هیچ است؛ پس می‌توان انتهای لیست را با آن مشخص کرد. راه‌حل دیگر، بهره‌گیری از رکوردی معمولی اما با یک کلید خاص است. این کلید خاص تضمین می‌کند که عمل جست‌وجوی عناصر در این رکورد متوقف خواهد شد. گاهی به این رکورد اضافی، رکورد پوچ یا قلابی می‌گویند. این رکورد، برنامه را ساده‌تر می‌کند، زیرا تعداد حالت‌های خاص کم‌تر می‌شود. رکوردهای پوچ در بسیاری از ساختمان‌داده‌های دیگر نیز سودمند واقع می‌شوند.

۴-۳ درخت‌ها

آرایه‌ها و لیست‌های پیوندی تنها برای مشخص کردن ترتیبی از عناصر (که در خود ذخیره کرده‌اند) به کار می‌آیند. در کاربردهای فراوانی نیازمند ساختارهای دیگری (جز ساختارهای ترتیبی ساده) هستیم. درخت‌ها، بیانگر ساختارهایی سلسله مراتبی هستند. می‌توان درخت‌ها را در ساختارهایی کارآمدتر برای

انجام عمل‌هایی خاص روی ساختارهای خطی نیز به کار گرفت. فعلاً تنها به درخت‌های سلسله مراتبی می‌پردازیم. این نوع درخت را هم درخت ریشه‌دار و هم arborescence می‌نامند. یک درخت ریشه‌دار، مجموعه‌ای است از عنصرها به نام گره (یا رأس) همراه با مجموعه‌ای از یال‌ها که عناصر را به روشی ویژه به یکدیگر پیوند می‌زنند (شکل ۴-۴ را ببینید). یکی از گره‌ها، ریشه‌ی درخت (و در بالای سلسله مراتب) است. ریشه به گره‌هایی متصل است که در سطح ۱ از سلسله مراتب قرار دارند. هر یک از آن‌ها نیز به گره‌های سطح ۲ متصلند و به همین ترتیب گره‌های درخت به یکدیگر متصل شده‌اند. پس هر پیوند، بین یک گره و تنها «بالادست» (supervisor) مستقیم آن برقرار شده است. (معمولاً به تقلید از درخت‌های تبارشناسی به گره بالادست مستقیم، «والد» گفته می‌شود.) ریشه، تنها گره بدون والد است. ویژگی اصلی درخت‌ها این است که «دور» ندارند. پس بین هر دو گره درخت، یک و تنها یک مسیر وجود دارد.



شکل ۴-۴ یک درخت ریشه‌دار

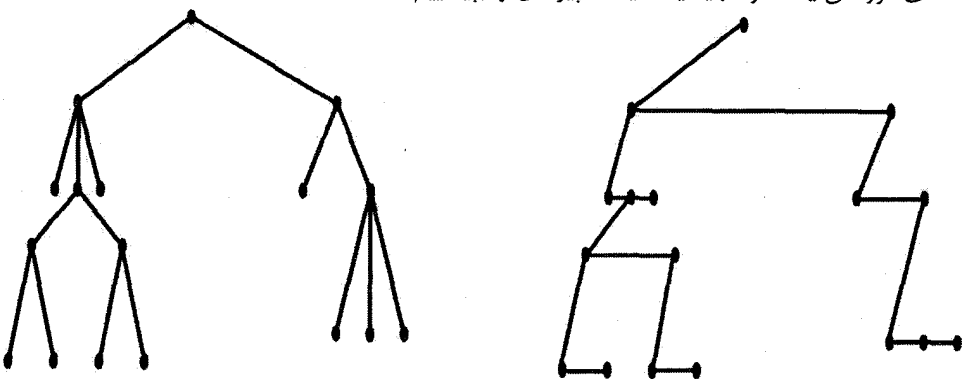
یک گره به والد خود و چندین گره زیردست (underling) مستقیم متصل است. (باز هم به پیروی از واژه‌های تبارشناسی به گره زیردست مستقیم، فرزند می‌گوییم.) بیش‌ترین تعداد ممکن برای فرزندان یک گره درخت، درجه‌ی آن درخت نامیده می‌شود. فرزندان هر گره را معمولاً به ترتیب قرار می‌دهیم؛ سپس هر گره را به کمک شماره‌ی ترتیبی خودش (یعنی فرزند اول، فرزند دوم و ...) از دیگر فرزندان متمایز می‌کنیم. در مورد درخت‌های درجه‌ی دو که درخت‌های دودویی نام دارند، فرزندان یک گره را به صورت فرزند چپ (اول) و فرزند راست (دوم) از هم متمایز می‌سازیم. گره بدون فرزند را برگ می‌گوییم (این اصطلاح از درختان واقعی گرفته شده است). گرهی که برگ نباشد، گره داخلی نامیده می‌شود. ارتفاع یا بلندی یک درخت، بیش‌ترین تعداد سطح‌های درخت است؛ یعنی، بیش‌ترین مقدار بین فاصله‌ی برگ‌های درخت تا ریشه‌ی آن. هر گره، یک کلید دارد که از مجموعه‌ای با ترتیب کلی (مانند اعداد حقیقی یا صحیح) گرفته شده است. اگر ابهامی رخ ندهد، می‌توانیم کلید گره و خود گره را به جای یکدیگر به کار ببریم. برای راحتی فرض می‌کنیم کلیدها یکتا هستند. اگر هم کلیدهای درخت یکتا نباشند، می‌توانیم همه‌ی عناصری را که کلید یکسان دارند، در یک لیست پیوندی قرار دهیم و به جای همه‌ی این عناصر، تنها یک گره در درخت بگذاریم که آن گره، اشاره‌گری به این لیست پیوندی داشته

باشد. معمولاً هر گره درخت یک میدان داده‌ای دارد که در برگیرنده‌ی داده‌های آن گره (یا اشاره‌گری به داده‌های آن گره) است. میدان داده‌ای گره، به کاربرد درخت بستگی دارد که اغلب کاری با آن نداریم.

حال به دو کاربرد درخت توجه می‌کنیم: درخت‌های جست‌وجو و درخت‌های هرمی. در هر دو مورد درخت‌های دودویی را به کار می‌بریم. بیایید بحث را با بررسی شیوه‌ی ذخیره‌ی درخت در حافظه آغاز کنیم.

۴-۳-۱ شیوه‌ی ذخیره‌ی درخت

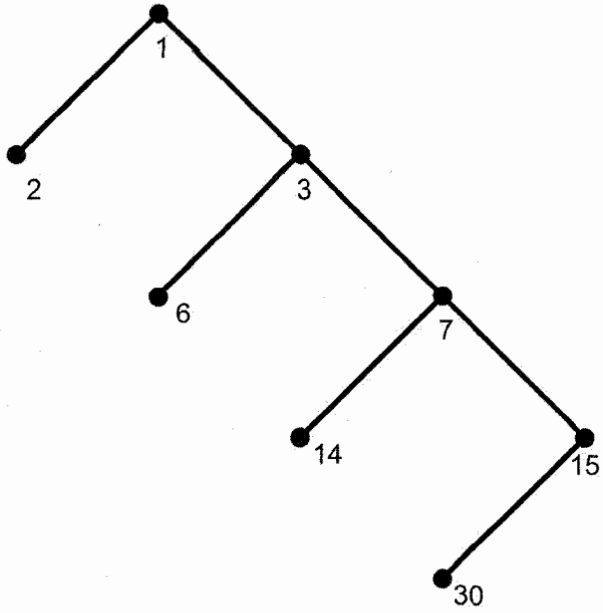
دو شیوه‌ی اصلی برای ذخیره‌ی درخت در حافظه وجود دارد: شیوه‌ی صریح و شیوه‌ی ضمنی. در روش صریح، پیوند بین گره‌های درخت را با اشاره‌گر برقرار می‌کنیم. گرهی با k فرزند، رکوردی است که آرایه‌ای از k اشاره‌گر دارد. (گاهی اشاره‌گری به گره والد را نیز در این رکورد قرار می‌دهند.) معمولاً راحت‌تر است که همه‌ی گره‌های درخت، هم‌نوع باشند. از این رو، اگر m، درجه‌ی درخت باشد. همه‌ی گره‌ها باید m اشاره‌گر داشته باشند. به جای این روش می‌توان در هر گره دو اشاره‌گر داشت که اشاره‌گر نخست به فرزند نخست و اشاره‌گر بعدی به گره بعدی در همین سطح از درخت اشاره کند. (گره‌هایی را که والد یکسان دارند، sibling یکدیگر گویند که می‌توان آن را «هم‌شیر» ترجمه کرد - مترجمان) در شکل ۴-۵ نمونه‌ای از این دو روش را برای ذخیره‌ی یک درخت نشان داده‌ایم. (توجه کنید که این دو روش، هر دو، جزو شیوه‌ی صریح هستند - مترجمان) عیب روش دوم در این است که برای بررسی همه‌ی فرزندان یک گره، باید یک لیست پیوندی را بپیماییم.



شکل ۴-۵ نمایش دودویی (الف) از یک درخت غیردودویی (ب)

درشیوه‌ی ضمنی از اشاره‌گر استفاده نمی‌شود، بلکه یک آرایه را برای نگه‌داری همه‌ی گره‌های درخت به کار می‌برند؛ پیوند گره‌ها با یکدیگر نیز با موقعیت گره در آرایه مشخص می‌شود. رایج‌ترین روش ذخیره‌ی ضمنی، هنگام پیاده‌سازی درخت چنین است: درخت دودویی T را در نظر بگیرید.

ریشه‌ی T در A[1] و فرزندان چپ و راست آن به ترتیب در A[2] و A[3] نگه‌داری می‌شوند. فرزندان فرزند چپ ریشه نیز به ترتیب در A[4] و A[5] نگه‌داری می‌گردند و ... . چنین آرایه‌ای، نمایش پیمایش درخت ازچپ به راست و از یک سطح به سطح بعدی است. می‌توانیم این نمایش را با استقرا چنین تعریف کنیم: (۱) ریشه در A[1] نگه‌داری می‌شود (حالت پایه). (۲) محل نگه‌داری فرزندان چپ و راست گره v که خودش در A[i] ذخیره شده است، به ترتیب در A[2i] و A[2i+1] خواهد بود. در این روش نیازی به اشاره‌گر نیست، پس در مصرف حافظه صرفه‌جویی خواهد شد. از سویی، اگر درخت نامتوازن باشد؛ یعنی فاصله‌ی برگ‌ها از ریشه یکسان نباشد، آنگاه وجود نداشتن برخی گره‌ها مشکل به وجود می‌آورد. یک درخت نامتوازن در شکل ۴-۶ نشان داده شده است. در این شکل، شماره‌ی زیر هر گره، مکان آن گره را در آرایه مشخص می‌کند. می‌بینیم که برای نمایش ۸ گره، به آرایه‌ای به اندازه‌ی ۳۰ نیاز است. ذخیره‌ی ضمنی، بسته به متوازن بودن یا متوازن نبودن درخت، ممکن است سبب صرفه‌جویی در حافظه بشود یا نشود. در این شیوه آرایه به کار برده می‌شود، پس انجام اعمال پویا (درج و حذف) در میانه‌ی درخت هزینه‌بر است. از سوی دیگر، اگر این اعمال، تنها به گره‌های پایانی آرایه محدود شوند، پشتیبانی از این اعمال، معقول و منطقی است.



شکل ۴-۶ ذخیره‌ی یک درخت نامتوازن به روش ضمنی

۴-۳-۲ درخت‌های هرمی

درخت هرمی، درختی است دودویی با این ویژگی:

«کلید هر گره، بزرگ‌تر یا مساوی کلید هر یک از فرزندانش است.»

بنا به قانون ترایایی، از ویژگی هرم نتیجه می‌شود که کلید هیچ گرهی از کلید گره‌های پایین‌دستش کوچک‌تر نیست. درخت هرمی در پیاده‌سازی صف اولویت به کار می‌آید. صف اولویت داده‌گونه‌ای است مجرد با این دو عمل:

Insert(x): عنصری با کلید x، به ساختمان‌داده می‌افزاید.

Remove(): عنصری را که دارای بزرگ‌ترین کلید است، از ساختمان‌داده کنار می‌گذارد.

پیاده‌سازی درختان هرمی هم با روش صریح و هم با روش ضمنی ممکن است. از آنجا که درخت هرمی متوازن است، آن را به شیوه‌ی ضمنی پیاده‌سازی می‌کنیم. آرایه‌ی A[1..k] را در نظر می‌گیریم که k حد بالایی برای تعداد عناصر هرم است. (اگر حد بالا معلوم نباشد، آنگاه به کار بردن لیست پیوندی ضروری خواهد بود.) n را تعداد فعلی عناصر هرم در نظر بگیرید؛ یعنی تنها بخش A[1..n] از آرایه مورد توجه است. اینک شیوه‌ی کارآمدی برای پیاده‌سازی اعمال درج و حذف در هرم بیان می‌کنیم.

بیایید نخست عمل Remove را بررسی کنیم. بنا به ویژگی هرم، گرهی که بزرگ‌ترین کلید را دارد، گره ریشه، یعنی A[1] است. پس عمل Remove همواره کلید را از ریشه برمی‌دارد. مهم آن است که پس از این کار، خاصیت هرم را به درخت بازگردانیم. در این حالت، آرایه‌ی A[2..n] شامل دو درخت هرمی جداگانه است. ابتدا برگ A[n] را به جای ریشه‌ی درخت قرار می‌دهیم و خودش را حذف می‌کنیم؛ یعنی عمل A[1]:=A[n] را انجام می‌دهیم و یک واحد از n کم می‌کنیم. مقدار تازه‌ی A[1] را x می‌گیریم. هنوز هم دو درخت هرمیِ جداگانه همراه با مقداری در رأس هر یک از آن‌ها داریم، اما این مقدارها ممکن است خاصیت هرم را برآورده کنند یا برآورده نکنند. (تنها هنگامی که مقدار گره‌های واقع در مسیر ریشه تا محل پیشین x، همگی برابر x باشند، ویژگی هرم بازهم برقرار است.) برای بازگرداندن ویژگی هرم به درخت، x را آن قدر به سمت پایین حرکت می‌دهیم تا به زیردرختی برسیم که x در آن زیردرخت، بیشینه باشد. این کار با مقایسه‌ی x با مقدار دو فرزند فعلی‌اش (A[2] و A[3]) آغاز می‌شود. اگر x از این دو بیش‌تر نبود، A[1] را با بیشینه‌ی آن دو جابه‌جا می‌کنیم. فرض کنید A[2] از A[3] بیش‌تر باشد. در این صورت، به روشنی پیداست که A[2]، کلید بیشینه در تمام هرم است و می‌تواند جای‌گزین ریشه گردد. به علاوه، زیردرختی که ریشه‌اش A[3] است، تغییری نمی‌کند و طبعاً از ویژگی هرم برخوردار است. ما تنها نگران زیردرختی هستیم که ریشه‌اش A[2] است (چراکه هم‌اکنون مقدار A[2]، x شده است). حال می‌توانیم به همان روش، کار را به صورت استقرایی ادامه دهیم. فرض کنید i گام از کار را انجام داده‌ایم و هم‌اکنون کلید x در A[j] قرار دارد. تنها یک زیردرخت با ریشه‌ی A[j]، ممکن است از ویژگی هرم برخوردار نباشد. مانند قبل، x را با دو فرزندش (در صورت وجود)، یعنی A[2j] و A[2j+1] مقایسه کرده، x را با بیشینه‌ی این دو جابه‌جا می‌کنیم. الگوریتم، هنگامی به پایان می‌رسد که یا x بیشینه‌ی یک زیردرخت شود، یا به برگ برسیم. بیش‌ترین تعداد

مقایسه‌های مورد نیاز برای عمل حذف، $2\lceil \log_2'' \rceil$ (یعنی دو برابر ارتفاع درخت) است. الگوریتم حذف عنصر بیشینه از درخت هرمی، در شکل ۴-۷ ارائه شده است.

الگوریتم: Remove_Max_from_Heap(A,n)

ورودی: A (آرایه‌ای به اندازه‌ی n که بیانگر یک هرم است.)

خروجی: Top_of_the_Heap (عنصر بیشینه‌ی هرم)، A (هرم تازه) و n (اندازه‌ی تازه‌ی هرم که اگر هرم، تهی باشد، صفر است.)

```
begin
    if n = 0 then print "هرم تهی است."
    else
        Top_of_the_Heap := A[1];
        A[1] := A[n];
        n := n-1;
        parent := 1;
        child := 2;
        while child ≤ n-1 do
            if A[child] < A[child+1] then
              child := child + 1;
            if A[child] > A[parent] then
              swap(A[parent],A[child]);
              parent := child;
              child := 2 * child;
            else child := n    {برای این که حلقه متوقف شود.}
end
```

شکل ۴-۷ الگوریتم Remove_Max_from_Heap

عمل Insert نیز به شیوه‌ای مشابه انجام‌پذیر است. نخست n را یک واحد می‌افزاییم و کلید تازه را برگ A[n] از درخت قرار می‌دهیم. سپس برگ تازه را با والدش مقایسه می‌کنیم و چنان‌چه برگ، از والدش بزرگ‌تر بود، آن دو را با یکدیگر جابه‌جا می‌کنیم. پس از این کار، کلید تازه، بیشینه‌ی زیردرخت خود است (زیرا پیش از آن، والد، بیشینه‌ی زیردرخت بوده و کلید تازه از آن هم کوچک‌تر نبوده است). به طور استقرایی فرض می‌کنیم هم درختی که ریشه‌اش A[j] (در آغاز A[n]) است، شرط هرم را برآورده کند و هم پس از برداشتن این درخت، باقی‌مانده‌ی درخت نیز از خاصیت هرم برخوردار باشد. سپس این فرایند را با حرکت کلید تازه به سمت بالای درخت ادامه می‌دهیم تا آن که یا به ریشه برسیم، یا کلید تازه، دیگر از والدش بزرگ‌تر نباشد. بدین‌ترتیب، کل درخت، یک درخت هرمی معتبر خواهد شد. بیش‌ترین تعداد مقایسه‌ی لازم برای افزایش یک گره به درخت $\lceil \log_2'' \rceil$، یعنی ارتفاع درخت است. الگوریتم اضافه کردن عنصری تازه به هرم در شکل ۴-۸ نشان داده شده است.

می‌توانیم هر یک از اعمال دنباله‌ای از Insert و Revmoveها را در زمانی از O(logn) انجام دهیم. از سویی، انجام کارآمد عمل‌های دیگر در یک درخت هرمی ممکن نیست. برای مثال، اگر بخواهیم کلیدی خاص را جست‌وجو کنیم، سلسله مراتب درخت هرمی هیچ کمکی به این کار نمی‌کند. هرم، نمونه‌ی خوبی از پیاده‌سازی یک داده‌گونه‌ی مجرد است. درخت هرمی، تنها شمار اندکی از اعمال مشخص را به صورتی کارآمد، مورد پشتیبانی قرار می‌دهد. هرگاه به هر یک از این اعمال نیاز پیدا کردیم، می‌توانیم داده‌ها را از هر نوعی هم که باشند، در یک هرم بچینیم.

الگوریتم: Insert_to_Heap(A,n,x)

ورودی: A (آرایه‌ای به اندازه‌ی n که بیانگر یک هرم است) و x (یک عدد)

خروجی: A (هرم تازه) و n (اندازه‌ی تازه‌ی هرم)

```
begin
    n := n + 1; {فرض می‌کنیم آرایه سرریز نخواهد شد.}
    A[n] := x;
    child := n;
    parent := n div 2;
    while parent ≥ 1 do
        if A[parent] < A[child] then
          swap(A[parent],A[child]);  { تمرین ۴-۶ را نیز ببینید.}
          child := parent;
          parent := parent div 2;
        else parent := 0    {برای این که حلقه متوقف شود.}
end
```

شکل ۴-۸ الگوریتم Insert_to_Heap

۴-۳-۳ درخت‌های دودویی جست‌وجو

درخت‌های دودویی جست‌وجو، این اعمال را به صورتی کارآمد پیاده‌سازی می‌کنند:

search(x): یا کلید x را در ساختمان‌داده می‌یابد و یا اعلام می‌کند که در ساختمان‌داده عنصری با کلید x وجود ندارد (برای سادگی فرض می‌کنیم که هر کلید، حداکثر یک بار پیدا می‌شود).

insert(x): کلید x را به ساختمان‌داده اضافه می‌کند (مگر این که در ساختمان‌داده، عناصری با کلید x وجود داشته باشد).

delete(x): اگر عنصری با کلید x در ساختمان‌داده موجود باشد، آن را حذف می‌کند.

به داده‌گونه‌ی مجردی که این اعمال را انجام دهد، فرهنگ داده‌ای گویند. درخت دودویی جست‌وجو توانایی پیاده‌سازی کارآمد فرهنگ داده‌ای و دیگر اعمال پیچیده را دارد. در این بخش، فرض می‌کنیم

درخت به روش صریح ذخیره شده باشد، چون پویا بودن اعمال درج و حذف در درخت‌های دودویی جست‌وجو مهم است و ما نمی‌خواهیم خودمان را به حد بالایی معین برای تعداد عناصر محدود کنیم. فرض می‌کنیم هر گره درخت، رکوردی است شامل دست‌کم سه میدان: key، left و right. key کلید متناظر با گره را نگه‌داری می‌کند. left و right اشاره‌گرهایی به گره‌های دیگر (یا nil) هستند. درخت‌های دودویی جست‌وجو از درخت‌های هرمی پیچیده‌ترند، زیرا در هرم تنها برگ‌ها، حذف یا اضافه می‌شدند و چیزی، جز کلید، جابه‌جا نمی‌گردید، ولی در درخت‌های دودویی جست‌وجو هر گرهی ممکن است حذف گردد و اشاره‌گرها هم می‌توانند به روش‌های مختلفی دست‌کاری شوند. برای سادگی، کلیدها را متمایز در نظر می‌گیریم.

درخت جست‌وجو

درخت جست‌وجو، چنان که از نامش پیداست، ساختاری برای آسان کردن عمل جست‌وجوست. اگر روال جست‌وجو را بفهمیم، درک این ساختار نیز آسان می‌گردد. فرض کنید کلیدی مانند x داریم و می‌خواهیم بدانیم آیا این کلید در درخت وجود دارد یا نه و در صورت وجود، می‌خواهیم گره دربردارنده‌ی این کلید را بیابیم. این کار جست‌وجو نام دارد. نخست x را با کلید ریشه‌ی درخت (که مقدارش را r می‌نامیم) مقایسه می‌کنیم. اگر x=r، کار جست‌وجو تمام می‌شود. اگر x<r، آنگاه جست‌وجو را با فرزند چپ، وگرنه با فرزند راست ادامه می‌دهیم. هر کلید درخت جست‌وجو، محدوده‌ی کلیدهای پایین‌تر از خود را به دو دسته تقسیم می‌کند: کلیدهای زیردرخت چپ که همگی کوچک‌تر از آن کلید و کلیدهای زیردرخت راست که همگی بزرگ‌تر از آن کلید هستند. این قاعده درخت‌های جست‌وجو را تعریف می‌کند. اگر همه‌ی کلیدها، شرایط گفته‌شده را برآورده سازند، درخت را سازگار گوییم. یک برنامه‌ی بازگشتی ساده برای جست‌وجو در این‌گونه درخت‌ها در شکل ۴-۹ ارائه شده است.

الگوریتم: BST_Search(root,x)

ورودی: root (اشاره‌گری به ریشه‌ی یک درخت دودویی جست‌وجو) و x (یک عدد)

خروجی: node (یا یک اشاره‌گر به گرهی که شامل کلید x است و یا nil، در صورتی که چنین گرهی وجود ندارد.)

```
begin
    if root = nil or root^.key = x then node := root
    {root^، رکوردی است که اشاره‌گر ریشه به آن اشاره می‌کند.}
    else
        if x < root^.key then BST_Search(root^.left,x)
        else BST_Search(root^.right,x)
end
```

شکل ۴-۹ الگوریتم BST_Search

عمل درج

عمل درج نیز، در درخت دودویی جست‌وجو بسیار ساده است. برای درج کلید x در درخت، نخست جست‌وجو برای x انجام می‌شود. چنان‌چه x در درخت وجود داشته باشد، دیگر عمل درج انجام نمی‌گیرد (فرض بر این است که نمی‌خواهیم چندین گره با کلید یکسان داشته باشیم)؛ اما اگر عمل جست‌وجو (بدون یافتن کلید x) به یک برگ ختم شود، گرهی با کلید تازه به زیر آن برگ می‌افزاییم (بسته به مقدار x، یا به عنوان فرزند راست، یا به عنوان فرزند چپ). پس از درج، خاصیت درخت، باز هم برقرار می‌ماند، چراکه جست‌وجوهای بعدی به همان برگ خواهند رسید و از آنجا به گره تازه خواهند رفت. برنامه‌ی جست‌وجو باید اندکی تغییر کند، به گونه‌ای که برگ نو را هم در نظر بگیرد. در شکل ۴-۱۰ یک برنامه‌ی جست‌وجوی غیر بازگشتی برای این کار نوشته‌ایم.

الگوریتم: BST_Insert(root,x)

ورودی: root (اشاره‌گری به ریشه‌ی یک درخت دودویی جست‌وجو) و x (یک عدد)

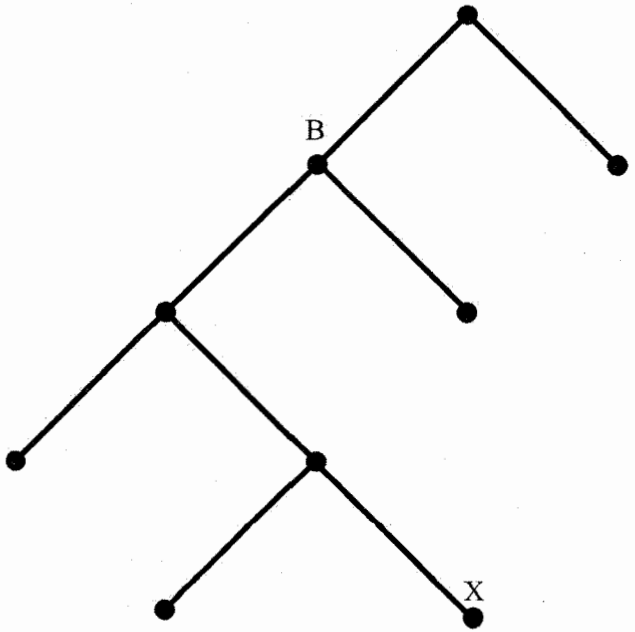
خروجی: اگر درخت با افزوده شدن گرهی که کلید آن x است، تغییر کند، اشاره‌گر child نیز به همین گره اشاره خواهد کرد؛ اما هنگامی که درخت از پیش گرهی با کلید x دارد، child، nil خواهد شد.

```
begin
    if root = nil then
      گره تازه‌ای ایجاد کن که child به آن اشاره کند;
      root := child;
      root^.key := x
    else
        node := root;
        child := root;   {برای مقداردهی اولیه تا مطمئن شویم که nil نیست.}
        while node ≠ nil and child ≠ nil do
              if node^.key = x then child := nil
              else
                  parent := node;
                  if x < node^.key then node := node^.left
                  else node := node^.right;
        if child ≠ nil then
          گره تازه‌ای ایجاد کن که child به آن اشاره کند;
          child^.key := x;
          child^.left := nil; child^.right := nil;
          if x < parent^.key then parent^.left := child
          else parent^.right := child
end
```

شکل ۴-۱۰ الگوریتم BST_Insert

عمل حذف

معمولاً عمل حذف پیچیده‌تر است. حذف یک برگ کار آسانی است؛ تنها کافی است اشاره‌گری را که به آن اشاره می‌کند، nil کنیم. حذف گره‌هایی که یک فرزند دارند، کار سختی نیست: اشاره‌گری را که به چنین گرهی اشاره می‌کند، به گونه‌ای تغییر می‌دهیم که به فرزند این گره اشاره کند؛ اما اگر بخواهیم گرهی با دو فرزند را حذف کنیم، ناچاریم جایی برای هر دو اشاره‌گر آن بیابیم. فرض کنید B گرهی است دارای دو فرزند که می‌خواهیم آن را حذف کنیم (شکل ۴-۱۱ را ببینید). در نخستین گام کلید گره B را با کلید گره X جابه‌جا می‌کنیم. گره X چنان است که: (۱) حداکثر یک فرزند دارد؛ (۲) حذف X (پس از جابه‌جایی کلید X و B) خاصیت درخت را به هم نمی‌زند. در گام بعدی X را حذف می‌کنیم، زیرا حالا شامل کلید B است و ما می‌خواستیم کلید B را حذف کنیم. حذف X آسان است، چون حداکثر یک فرزند دارد. برای سازگار ماندن درخت (حفظ ویژگی آن)، کلید X باید بزرگ‌تر یا مساوی بزرگ‌ترین کلید زیردرخت چپ B باشد و از همه‌ی کلیدهای زیردرخت راست B نیز کوچک‌تر باشد. توجه کنید که در شکل ۴-۱۱ کلید X این نیازها را برآورده می‌کند، یعنی کلید آن از همه‌ی کلیدهای زیردرخت چپ B بزرگ‌تر است. X، سلف B در درخت خوانده می‌شود. X فرزند راست نخواهد داشت، چراکه در آن صورت، دیگر بزرگ‌ترین کلید زیردرخت نیست. الگوریتم حذف در شکل ۴-۱۲ نشان داده شده است.



شکل ۴-۱۱ حذف گرهی با دو فرزند

الگوریتم: BST_Delete(root,x)

ورودی: root (اشاره‌گری به ریشه‌ی یک درخت دودویی جست‌وجو) و x (یک عدد)

خروجی: همان درخت، بدون کلید x

```
{ فرض می‌کنیم همه‌ی کلیدها متمایزند و ریشه نیز هیچ‌گاه حذف نمی‌گردد.}
begin
    node := root;
    while node≠nil and node^.key ≠ x do
        parent := node;
        if x < node^.key then node := node^.left
        else node := node^.right
    if node = nil then print ("x در درخت نیست."); halt;      {halt برای خروج از
    if node ≠ root then                                      الگوریتم است - مترجمان}
      if node^.left = nil then
        if x ≤ parent^.key then
          parent^.left := node^.right
        else parent^.right := node^.right
      else if node^.right = nil then
              if x ≤ parent^.key then
                parent^.left := node^.left
              else parent^.right := node^.left
           else    {حالتی که گره دو فرزند دارد.}
              nodel := node^.left;
              parentl := node;
              while nodel^.right ≠ nil do
                    parentl := nodel;
                    nodel := nodel^.right;
              {عمل حذف واقعی اینجا انجام می‌شود.}
              parentl^.right := nodel^.left;
              node^.key := nodel^.key
end
```

شکل ۴-۱۲ الگوریتم BST_Delete

پیچیدگی: زمان اجرای هر یک از اعمال جست‌وجو، درج و حذف، هم به شکل درخت و هم به محل گره مورد نظر بستگی دارد. در بدترین حالت، مسیر جست‌وجو تا پایین درخت ادامه می‌یابد. گام‌های دیگر این الگوریتم‌ها (برای مثال، خود عمل درج یا جابه‌جایی کلیدها در عمل حذف) به زمان ثابتی نیاز دارند. بنابراین بدترین حالت زمان اجرا، متناسب با طولانی‌ترین مسیر بین ریشه و برگ‌ها، یعنی ارتفاع درخت است. اگر درخت نسبتاً متوازن باشد (توازن را به زودی تعریف خواهیم کرد)، آنگاه ارتفاع آن تقریباً $\log_2^n$ خواهد بود (n تعداد گره‌های درخت است). انجام همه‌ی عمل‌ها در این حالت، کارآمد خواهد بود، اما اگر درخت نامتوازن باشد، انجام آن‌ها بسیار ناکارآمد خواهد شد.

چنان‌چه کلیدها به ترتیبی تصادفی به درخت افزوده شوند، ارتفاع مورد انتظار درخت از O(logn) – دقیقاً 2lnn – خواهد بود که در این حالت، اعمال جست‌وجو و افزودن یک گره، کارآمد هستند؛ اما در بدترین حالت، ارتفاع درخت n خواهد شد (هنگامی که درخت شبیه یک لیست پیوندی شده باشد). درخت‌هایی که مسیرهای بلندی دارند، در اثر افزودن گره‌ها به صورت مرتب یا تقریباً مرتب به وجود آمده‌اند. عمل حذف هم (حتا اگر به صورت تصادفی انجام پذیرد) ممکن است سبب بروز مشکل شود؛ چراکه هنگام حذف یک گره ممکن است ارتفاع زیردرخت‌های چپ و راستِ گرهی دیگر از درخت بسیار متفاوت شود. (نویسنده‌ی کتاب از این مشکل با عنوان «عدم تقارن» یاد کرده است – مترجمان) چنان‌چه به دنبال اعمال حذف، اعمال درج نیز انجام گردند – حتا اگر این حذف و درج‌ها به ترتیبی تصادفی صورت گیرند – ممکن است ارتفاع درخت از $O(\sqrt{n})$ شود. برای جلوگیری از بروز عدم تقارن می‌توانیم به جای آن که همیشه سلف یک گره را برگزینیم، گاهی نیز خلف آن را به کار ببریم (که کوچک‌ترین کلید در زیردرخت راست است). خوش‌بختانه روش‌هایی برای پیش‌گیری از ایجاد مسیرهای طولانی در درخت‌های دودویی جست‌وجو وجود دارد. یکی از این شیوه‌ها را در بخش بعدی بررسی خواهیم کرد.

۴-۳-۴ درخت‌های AVL

درخت AVL (که نامش از Adel'son-Vel'skii و Landis [۱۹۶۲] گرفته شده است) نخستین ساختمان‌داده‌ای است که زمان اجرای O(logn) را برای جست‌وجو، حذف و اضافه، در بدترین حالت تضمین می‌کند (n تعداد عناصر است). ایده‌ی اصلی درخت‌های AVL (و بیش‌تر ساختارهای درختی دیگری که حد بالای لگاریتمی دارند) این است که زمانی اضافی، صرف عمل درج و حذف شود تا درخت همواره متوازن، یعنی با ارتفاعی از O(logn) باقی بماند. زمان متوازن کردن نیز نباید از O(logn) فراتر رود، چراکه دو عمل درج و حذف، زیادی هزینه‌بر خواهند شد. روشی مناسب، تعریف معیاری برای توازن است که کار با آن آسان باشد.

تعریف: درخت AVL، یک درخت دودویی جست‌وجو است که در هر یک از گره‌های آن، اختلاف بین ارتفاع زیردرخت‌های چپ و راست بیش از یک نیست (ارتفاع درخت تهی را صفر تعریف می‌کنیم).

این تعریف بنا به قضیه‌ی ۴-۱ تضمین می‌کند که ارتفاع درخت از O(logn) بیش‌تر نیست.

قضیه‌ی ۴-۱

اگر یک درخت AVL با ارتفاع h، n گره داخلی داشته باشد، این نامساوی برقرار است:

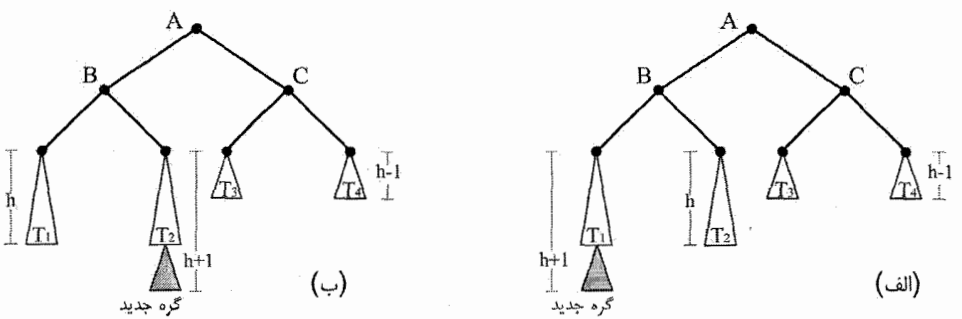
$$h < 1.4404 \log_2^{(n+2)} - 0.328$$

اثبات به عنوان تمرین به خواننده واگذار می‌شود. □

از این قضیه می‌توان نتیجه گرفت تعداد مقایسه‌های عمل جست‌وجو در درخت‌های AVL از O(logn) است. مشکل، یافتن روشی برای انجام عمل‌های درج و حذف است که خاصیت AVL را از بین نبرد. از عمل درج شروع می‌کنیم؛ با این فرض که همه‌ی کلیدها متمایزند.

x را کلید تازه‌ای بگیرید که می‌خواهیم آن را به درخت AVL بیفزاییم. نخست به روش معمول، x را به پایین درخت اضافه می‌کنیم. اگر پس از این کار، درخت، دیگر AVL نبود، ناچاریم درخت را از نو متوازن سازیم. چهار حالت گوناگون ممکن است که دوتای آن را در شکل ۴-۱۳ نشان داده‌ایم. (دو حالت دیگر نیز شبیه دو حالت گفته‌شده هستند، اما در آن‌ها، ارتفاع زیردرخت راست بیش‌تر از ارتفاع زیردرخت چپ است.)

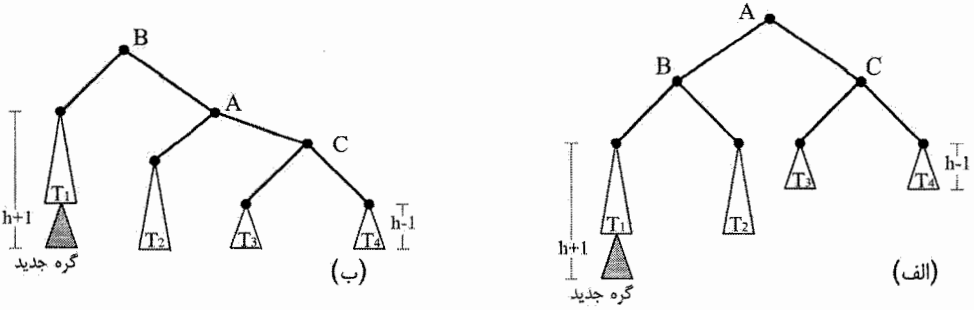
در بخش (الف) از شکل ۴-۱۳، گره تازه به زیردرخت چپ اضافه شده و ارتفاع B را h+2 کرده است، در حالی که ارتفاع C، h است. برای برطرف کردن مشکل، یک چرخش انجام می‌دهیم: B را به بالا می‌بریم و بقیه‌ی درخت را بنا به ویژگی درخت دودویی جست‌وجو تغییر می‌دهیم (شکل ۴-۱۴). با این کار ارتفاع زیردرخت تازه‌ای که ریشه‌اش B است، h+2 می‌شود که با ارتفاع زیردرخت، پیش از عمل درج، برابر است. (در شکل ۴-۱۴، بخش (الف)، گره A را ریشه فرض کرده‌ایم، اما در حالت کلی، خود این گره ممکن است فرزند گره دیگری باشد. به همین دلیل به جای واژه‌ی درخت، زیردرخت را به کار برده‌ایم - مترجمان) پس خاصیت درخت AVL بازهم برقرار است و نیاز به کار دیگری نیست. این چرخش، چرخش منفرد نامیده می‌شود؛ اما با این روش نمی‌توان مشکل شکل ۴-۱۳ (ب) را حل کرد و برای این حالت به چرخشی دوگانه (یعنی دو چرخش) نیاز است (شکل ۴-۱۵). ارتفاع زیردرختِ تازه، پس از انجام چرخش دوگانه با ارتفاع زیردرخت اولیه (پیش از افزودن عنصر تازه - مترجمان) برابر است و نیاز به کار دیگری نیست. ویژگی مهم درخت‌های AVL این است که همواره یک چرخش (منفرد یا دوگانه) پس از عمل درج، کافی است. از اثبات این مطلب چشم‌پوشی می‌کنیم.



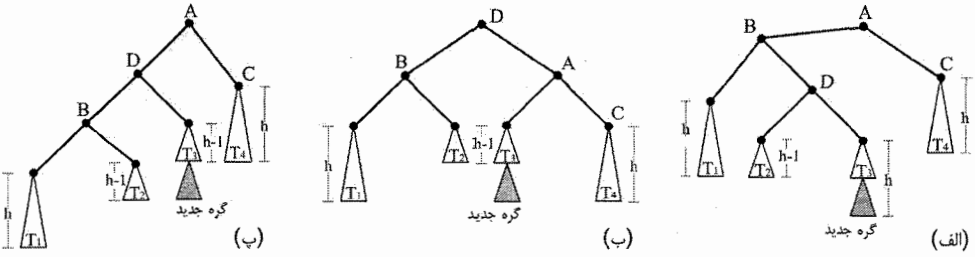
شکل ۴-۱۳ اعمال درجی که ویژگی درخت AVL را به هم می‌زنند.

در هر دو مورد به گره A «گره بحرانی» می‌گوییم. این گره، ریشه‌ی کوچک‌ترین زیردرختی است که پس از عمل درج، خاصیت AVL در آن برقرار نخواهد بود. برای انجام عمل درج، باید گره بحرانی را بیابیم و مشخص کنیم که کدام حالت رخ داده است. در هر گره، به اختلاف ارتفاع دو زیردرخت چپ و

راست توجه می‌کنیم و آن را عامل توازن گره می‌نامیم. عامل توازن در هر گره از یک درخت AVL، ۱، ۱- یا ۰ است. عمل درج در یک زیردرخت هنگامی نیاز به توازن دوباره دارد که عامل توازن ریشه‌ی زیردرخت ۱ یا ۱- باشد و عمل درج، ارتفاع زیردرخت را در جهتی «نادرست» افزایش دهد. از این موضوع نتیجه می‌شود که عامل توازن گره بحرانی صفر نیست. در ضمن، اگر عامل توازن یک گره پایین‌تر صفر نباشد، پس از توازن دوباره، ارتفاع گره‌های زیردرخت آن گره، نسبت به پیش از عمل درج تغییر نمی‌کند. (به یاد بیاورید که متوازن ساختن، ارتفاع پیشینِ گره‌های زیردرختِ گره بحرانی را به هم نمی‌زد.) از این رو، گره بحرانی، در بین بالادست‌های گره تازه‌ای که عامل توازن آن صفر نیست، کوچک‌ترین کلید را دارد. به سمت پایین درخت، حرکت و به عامل‌های توازن نگاه می‌کنیم و آخرین عامل توازن غیرصفر را در نظر می‌گیریم. هنگامی که به برگ برسیم، می‌توانیم به آسانی مشخص کنیم که آیا جهت درج گره تازه، «درست» بوده است یا «نادرست». سپس گذر دیگری (یا از پایین به بالا یا از بالا به پایین که اولی بهتر است، چون معمولاً تعداد کم‌تری گره دارد) انجام می‌دهیم و تنها، عامل‌های توازن را بررسی می‌کنیم و در صورت نیاز عمل چرخش را انجام می‌دهیم. دیگر به بررسی جزئیات نمی‌پردازیم.



شکل ۴-۱۴ درخت AVL؛ (الف) پیش از چرخش منفرد (ب) پس از چرخش منفرد



شکل ۴-۱۵ درخت AVL: (الف) پیش از چرخش دوگانه (ب) پس از چرخش دوگانه (پ) حالت درخت پس از انجام نخستین بخش از چرخش دوگانه (بند «پ» را مترجمان افزوده‌اند.)

عمل حذف، طبق معمول پیچیده‌تر است. پس از عمل حذف یک گره، دیگر، تنها با چرخش منفرد یا دوگانه نمی‌توان توازن را دوباره برقرار کرد. در برخی موارد، تعداد چرخش‌های مورد نیاز از $O(\log n)$ است (در اینجا n تعداد گره‌های درخت است). خوش‌بختانه تعداد گام‌های هر چرخش، ثابت

است؛ پس بدترین زمان اجرای عمل حذف یک گره هنوز هم از O(logn) است. در اینجا نیز وارد جزئیات نمی‌شویم.

توجه: درخت AVL، ساختمان‌داده‌ی کارآمدی است و عمل‌کرد بدترین حالت آن، خوب است و نسبت به درخت‌های بهینه حداکثر به ۴۵ درصد مقایسه‌ی بیش‌تر نیاز دارد. در حالت میانگین، حتا از این هم بهتر عمل می‌کند. مطالعات تجربی نشان داده است که برای جست‌وجو به طور متوسط تقریباً به $\log_2^n + 0.25$ مقایسه نیاز است (صفحات ۴۶۰ به بعد از Knuth [۱۹۷۳] را ببینید). عیب اصلی درخت‌های AVL نیاز به حافظه‌ی اضافی برای نگه‌داری عامل‌های توازن و پیچیدگی نسبی برنامه‌ی پیاده‌ساز آن‌هاست. روش‌های فراوان دیگری نیز برای درخت‌های متوازن‌کننده‌ی جست‌وجو پیش‌نهاد شده‌اند؛ مانند درخت‌های ۲-۳، B، تناسب وزنی و قرمز-سیاه.

۴-۴ درهم‌سازی

درهم‌سازی (اگر نگوییم سودمندترین) یکی از سودمندترین ساختمان‌های داده‌ای در الگوریتم‌های رایانه‌ای است. درهم‌سازی را بیش‌تر برای درج و جست‌وجو به کار می‌برند، ولی می‌توان از برخی گونه‌های آن برای حذف نیز بهره برد. ایده‌ی اصلی درهم‌سازی بسیار ساده است. طراحی یک ساختمان‌داده برای ذخیره‌ی داده‌هایی که کلیدهایشان از ۱ تا n است، کار سختی نیست: داده‌ها در آرایه‌ای به اندازه‌ی n نگه‌داری شوند، به این ترتیب که کلید i در محل i از آرایه ذخیره گردد؛ بدین ترتیب می‌توان به هر کلید بلافاصله دست‌رسی داشت. برای مثال اگر n کلید متمایز در محدوده‌ی ۱ تا 2n وجود داشته باشند، بهتر است آن‌ها را در آرایه‌ای به اندازه‌ی 2n نگه‌داری کنیم، هرچند که بهره‌وری حافظه به ۵۰ درصد کاهش می‌یابد. در این حالت، سرعت دست‌رسی چنان است که می‌ارزد فضای بیش‌تری را تخصیص دهیم؛ اما اگر کلیدها اعدادی صحیح در محدوده‌ی ۱ تا M باشند که M بزرگ‌ترین عدد صحیح قابل ارائه در رایانه‌ی ماست، تخصیص فضایی به اندازه‌ی M، هزینه‌ای زیاده از حد است. به عنوان مثال، اگر ۲۵۰ دانش‌جو وجود داشته باشند که با شماره‌ی بیمه‌ی‌شان شناسایی شوند، نمی‌توانیم برای نگه‌داری اطلاعات مربوط به آن‌ها آرایه‌ای با اندازه‌ی یک میلیارد تخصیص دهیم. (توجه کنید که یک میلیارد شماره‌ی بیمه‌ی متفاوت وجود دارد). در عوض، می‌توانیم سه رقم پایانی شماره‌ی بیمه را به کار ببریم که در این حالت، تنها به آرایه‌ای با اندازه‌ی ۱۰۰۰ نیازمندیم. (توجه کنید که این مثال برای کشور آمریکاست که در آنجا افراد، شماره‌ی بیمه‌ی یکتایی دارند. می‌توانید آن را مانند شماره‌ی ملی در کشور خودمان در نظر بگیرید - مترجمان) این شیوه بدون اشکال نیست. ممکن است سه رقم پایانی شماره‌ی بیمه‌ی برخی دانش‌جویان یکسان باشد (واقعیت آن است که در بین ۲۵۰ دانش‌جو احتمال بروز چنین اشکالی که آن را تکرار کلید گوییم، بسیار بالاست). به زودی چگونگی حل مشکل تکرار کلید را نشان خواهیم داد. یک راه هیگر، به کار بردن چهار رقم پایانی یا سه رقم پایانی به

همراه نخستین حرف نام دانش‌جوست تا تکرار کلیدها را کم‌تر کنیم؛ اما برای به‌کارگیری رقم‌های بیش‌تر به آرایه‌ای بزرگ‌تر نیاز داریم که سبب کاهش بهره‌وری حافظه می‌شود.

فرض کنید مجموعه‌ای از n کلید به ما داده شده است که اعضای آن از مجموعه‌ی U با اندازه‌ی M گرفته شده‌اند و M از n بسیار بزرگ‌تر است. می‌خواهیم این کلیدها را در جدولی با اندازه‌ی s که خیلی از n بیش‌تر نیست، نگه‌داری کنیم. شیوه‌ی مناسب انجام این کار، بهره‌گیری از یک تابع به نام «تابع درهم‌ساز» است که کلیدهای فاصله‌ی ۱ تا M را به کلیدهای تازه‌ای در محدوده‌ی ۱ تا s بنگارد. به این ترتیب، می‌توانیم همه‌ی کلیدها را در آرایه‌ای با اندازه‌ی s نگه‌داری کنیم. به‌کارگیری سه رقم پایانیِ یک عدد صحیح بزرگ می‌تواند چنین تابعی باشد؛ یعنی تابعی که مجموعه‌ی بزرگ U با اندازه‌ی یک میلیارد را به مجموعه‌ای با اندازه‌ی ۱۰۰۰ می‌نگارد. پس هر کلید، محلی (یا اندیسی) در جدولِ با اندازه‌ی s خواهد داشت که ما تلاش می‌کنیم آن کلید را در همان محل ویژه‌ی خودش ذخیره سازیم. چنان‌چه محاسبه‌ی تابع آسان باشد، دست‌رسی به کلید هم آسان خواهد بود. از آنجایی که مجموعه‌ی U، بزرگ و جدول، کوچک است، بدون توجه به تابعی که برای درهم‌سازی به کار برده‌ایم، کلیدهای بسیاری به موقعیت‌های یکسان از جدول نگاشته خواهند شد. هرگاه دو کلید متمایز به موقعیتی یکسان از جدول نگاشته شوند، می‌گوییم یک «برخورد» رخ داده است. پس با دو مشکل روبه‌رو هستیم: (۱) پیدا کردن تابعی برای درهم‌سازی که احتمال برخورد را کمینه کند و (۲) چاره‌جویی برای حل مشکل برخورد.

حتا بیش‌تر وقت‌هایی که مجموعه‌ی U بسیار بزرگ‌تر از اندازه‌ی جدول است، مجموعه‌ی کلیدهایی که مدیریت می‌کنیم، چندان بزرگ نیست. تابعی برای درهم‌سازی خوب است که کلیدها را به طور یک‌نواخت به جدول نگاشت کند. روشن است که یک تابع درهم‌ساز نمی‌تواند بدون برخورد، هر مجموعه‌ای از کلیدها را نگاشت کند. اگر اندازه‌ی U، M و اندازه‌ی جدول، s باشد، آنگاه باید دست‌کم M/s کلید به یک مکان از جدول نگاشته شوند. اگر درهم‌سازی، یک‌نواخت باشد، آنگاه به هر محل جدول تقریباً M/s کلید نگاشته می‌گردد. در اینجا، تابع درهم‌ساز باید به طور یک‌نواخت مجموعه‌ای از کلیدها را به مجموعه‌ای از «محل‌های تصادفی در محدوده‌ی ۱ تا s» تبدیل کند. «یک‌نواختی» و «تصادفی بودن» پایه‌ی اصلی درهم‌سازی است. در مثال پیش، به جای بهره‌گیری از سه رقم پایانی شماره‌ی بیمه، می‌توانستیم سه رقم پایانی سال تولد دانش‌جویان را به کار ببریم. روشن است که چنین تابعی بسیار نامناسب است، چراکه احتمال تولد چند دانش‌جو در یک سال، بسیار بیش‌تر است تا یکسان بودن سه رقم پایانی شماره‌ی بیمه‌ی آن‌ها.

توابع درهم‌ساز

فرض می‌کنیم کلیدها، اعدادی صحیح و اندازه‌ی جدول درهم، s باشد. یک تابع درهم‌ساز ساده $h(x)=x \bmod s$ است که اگر s، عددی اول باشد، بسیار کارآمد خواهد بود. اگر تنظیم اندازه‌ی جدول به یک عدد اول، آسان نباشد (برای مثال، گاهی اگر اندازه‌ی جدول، توانی از ۲ باشد، راحت‌تریم) آنگاه می‌توانیم تابع درهم‌ساز $h(x)=(x \bmod p) \bmod s$ را به کار ببریم که در آن p عددی اول و بزرگ‌تر از s است. (p باید به اندازه‌ی کافی از s بزرگ‌تر باشد تا تابع درهم‌ساز کارآمد گردد، اما باید از |U| یعنی تعداد اعضای مجموعه‌ی U نیز به قدر کافی کوچک‌تر باشد.)

چنان‌که اندکی پیش نیز اشاره شد، تابع خوبی برای درهم‌سازی همه‌ی ورودی‌ها وجود ندارد. بنا به آنچه توصیف شد، به‌کارگیری اعداد اول، روشی نسبتاً مطمئن است، زیرا در عمل، ساختار بیش‌تر داده‌ها به اعداد اول وابسته نیست. از سویی دیگر، امکان دارد (هرچند با احتمال کم) که در کاربردی خاص، کسی بخواهد نتیجه‌ی برخی آزمون‌های روی اعداد صحیح را ذخیره کند و این عددهای صحیح همگی به صورت r+kp باشند (که در آن r یک ثابت است). اگر p را به روش گفته‌شده برگزینیم، بدبختانه مقدار تابع درهم‌سازی برای همه‌ی این عددها یکسان می‌شود. می‌توانیم از ایده‌ی به هم ریختن داده‌ها، به یاری مرحله‌ی دیگری از درهم‌سازی کمک بگیریم و از یک روال تصادفی برای گزینش تابع درهم‌ساز بهره‌مند شویم! برای نمونه، می‌توان از فهرستی از عددهای اول که در محدوده‌ای مناسب قرار دارند، عدد اولی مانند p را برگزید، اما یافتن فهرستی بزرگ از اعداد اول کار آسانی نیست. راه دیگر چنین است: به طور تصادفی دو عدد a و b را برگزینید، به گونه‌ای که $a,b<p$ و $a\neq 0$ و $h(x)$ را برابر با $[ax+b \bmod p] \bmod s$ قرار دهید. محاسبه‌ی این تابع، از تابع پیش پیچیده‌تر است، اما این مزیت را دارد که به طور میانگین برای همه‌ی ورودی‌ها خوب است. قطعاً در هر دست‌رسی به یک جدول باید تابع درهم‌ساز همان جدول را به کار برد. هنگامی که به جدول‌های مستقل بسیاری نیازمندیم، یا جدول‌هایی را به کار می‌بریم که بارها و بارها ایجاد و حاذف می‌شوند؛ می‌توانیم هر بار که جدول متفاوتی ایجاد می‌گردد، از تابع درهم‌ساز متفاوتی نیز سود بجوییم. توابع درهم‌ساز تصادفیِ مورد اشاره، ویژگی‌های مطلوب دیگری نیز دارند (که در اینجا، این ویژگی‌ها مورد بحث قرار نگرفته است – مترجمان).

چاره‌جویی برای حل مشکل برخورد

ساده‌ترین روش برای حل مشکل برخورد، شیوه‌ای است که «زنجیره‌بندی مجزا» نام دارد. در این روش، از روی هر خانه‌ی جدول درهم، می‌توان آغاز یک لیست پیوندی را یافت. این لیست پیوندی شامل کلیدهایی است که تابع درهم‌ساز به آن خانه نسبت می‌دهد. برای دست‌رسی به یک کلید، پس از

درهم‌سازی آن، جست‌وجویی خطی روی لیست پیوندی یافته‌شده انجام می‌دهیم. می‌توان کلید تازه را به ابتدای لیست افزود (البته باید لیست را جست‌وجو کنیم تا مطمئن شویم که آن کلید، تکراری نیست). اگر برخی از این لیست‌ها طولانی باشند، آنگاه جست‌وجو ناکارآمد خواهد بود. اگر اندازه‌ی جدول در مقایسه با تعداد واقعی کلیدها کوچک باشد، یا اگر تابع درهم‌ساز بد عمل کند، لیست‌ها طولانی می‌شوند. بنابراین، درهم‌سازی، ساختار پویای مناسبی نیست و برآورد تقریبی تعداد کلیدها مهم است. مشکل اصلی زنجیره‌بندی مجزا، نیاز به تخصیص پویای حافظه و نیاز به فضای اضافی برای اشاره‌گرهاست (حتا اگر تعداد کلیدها بسیار زیاد نباشد و از اشاره‌گرها استفاده نشود). از طرفی، حتا اگر بنا به دلیلی برآورد اندازه‌ی جدول اشتباه باشد، زنجیره‌بندی مجزا به کار خود ادامه خواهد داد، درحالی‌که دیگر شیوه‌های ایستا به شکست می‌انجامند.

روش ساده‌ی دیگر «بررسی خطی» است. در این شیوه، اندازه‌ی جدول ثابت است و اشاره‌گری هم وجود ندارد. تابع درهم‌ساز محل کلید را در جدول مشخص می‌کند. چنان‌چه آن محل، از پیش پر شده باشد، یعنی اگر برخورد روی دهد، به جای آن محل، از نخستین جای خالیِ پس از آن استفاده می‌شود. جست‌وجوی کلید با همان روال گذشته صورت می‌گیرد. (ترتیب جدول، چرخشی در نظر گرفته می‌شود؛ یعنی اگر به آخرین محل برسیم و آن محل پر شده باشد، آنگاه محل بعدی، نخستین خانه‌ی جدول خواهد بود.) پس، رسیدن به نخستین محل خالی نشانگر نبودن عنصر در جدول است. زمانی که جدول نسبتاً خالی باشد، این شیوه‌ی ساده عمل‌کرد خوبی خواهد داشت؛ اما اگر جدول نسبتاً پر باشد، «برخوردهای ثانویه‌ی» بسیاری روی خواهند داد. (برخوردهای ثانویه به برخوردهایی گفته می‌شود که از برخورد کلیدهایی با مقدارهای درهم‌سازی گوناگون به وجود می‌آیند.) ما نمی‌توانیم از برخورد کلیدهایی که مقدار تابع درهم‌ساز برای آن‌ها برابر می‌شود، جلوگیری کنیم، زیرا این کلیدها به محلی یکسان نگاشته می‌شوند؛ اما باید تلاش کنیم تا برخوردهای ثانویه کمینه گردند. بیایید مثالی را با هم بررسی کنیم: فرض کنید مکان i‌ام پر و مکان i+1‌ام خالی است. کلید تازه‌ای که به i نگاشته شود، سبب برخورد خواهد شد و در محل i+1‌ام درج می‌گردد. در این مورد، چون با اندکی تلاش، مشکل برطرف شد، پس روش، کارآمد باقی ماند. حال، اگر کلید تازه‌ای به محل i+1 نگاشته شود، برخوردی ثانویه روی می‌دهد و محل i+2 پر می‌شود (البته اگر از پیش پر نشده باشد). هر کلید تازه‌ای که به i، i+1 یا i+2 نگاشته شود، نه تنها با برخورد ثانویه روبه‌رو می‌گردد، بلکه اندازه‌ی این بخش پرشده را نیز افزایش می‌دهد و به دنبال آن در آینده برخوردهای ثانویه‌ی بیش‌تری روی خواهد داد. این پدیده را «خوشه‌ی پرشده» گویند. در این روش هنگامی که جدول تقریباً پر است، تعداد برخوردهای ثانویه آن قدر زیاد خواهد شد که جست‌وجو تقریباً به کندی یک جست‌وجوی خطی خواهد گردید.

پیاده‌سازی عمل حذف با روش «بررسی خطی» کارآمد نیست. اگر برای درج یک عنصر از روی یک کلید عبور کنیم تا به یک محل خالی برسیم و بعداً آن کلید حذف شود، آنگاه در آینده جست‌وجو

ناموفق خواهد بود، زیرا این عمل در محل «کلید تازه حذف‌شده» متوقف خواهد شد که اگر انجام عمل حذف ضرورت داشته باشد، باید برای حل این مشکل چاره‌ای بیندیشیم.

می‌توان اثر خوشه‌ی پرشده را با درهم‌سازی دوگانه کاهش داد. در این روش هنگام برخورد، تابع دومی (مانند $h_2(x)$) را برای درهم‌سازی به کار می‌بریم و به جای جست‌وجوی خطی،یعنی i+1، i+2، ... به ترتیب مکان‌های $i+h_2(x)$، $i+2h_2(x)$ ... را جست‌وجو می‌کنیم (باز هم به ترتیب چرخشی). اگر کلید دیگری مانند y به $i+h_2(x)$ نگاشته شود، محل بعدی $i+h_2(x)+h_2(y)$ خواهد بود، نه $i+2h_2(x)$. اگر $h_2(x)$ از $h_2(y)$ مستقل باشد، با پدیده‌ی خوشه‌ی پرشده روبه‌رو نخواهیم شد. باید در گزینش تابع دوم درهم‌ساز دقت کنیم تا دنباله‌ی $i+h_2(x)$، $i+2h_2(x)$ ...، $i+nh_2(x)$ تمام جدول را در بر گیرد (که اگر اعداد $h_2(x)$ و n نسبت به هم اول باشند، این‌گونه خواهد شد).

اشکال عمده‌ی درهم‌سازی دوگانه نیاز به محاسبات اضافی، برای محاسبه‌ی مقدار دوم، هنگام عمل جست‌وجوست. یک روش برای کم کردن محاسبات، برگزیدن تابع دوم درهم‌ساز به گونه‌ای است که از تابع نخست کاملاً مستقل نباشد، اما سبب کاهش خوشه‌ی پرشده گردد. یک نمونه از این روش برگزیدن تابع $h_2(x)$ به صورت

$$h_2(x)=\begin{cases}1 & ,h_1(x)=0\\ m-h_1(x) & ,h_1(x)\neq 0\end{cases}$$

است (با این فرض که m عددی اول است و $h_1(x)=x \bmod m$).

۴-۵ مسأله‌ی union-find

مسأله‌ی union-find (که آن را مسأله‌ی هم ارزی نیز می‌گویند) نمونه‌ای خوب از کاربرد ساختمان‌های داده‌ای عجیب و غریب برای بهبود کارایی الگوریتم‌هاست. مسأله چنین است: n عنصر x_1، x_2، ... و x_n موجودند که به گروه‌هایی دسته‌بندی شده‌اند. در آغاز، هر عنصر به تنهایی یک گروه تشکیل می‌دهد. دو نوع عمل روی عناصر و گروه‌ها به ترتیب دل‌خواه انجام می‌گردد:

find(i): نام گروهی را برمی‌گرداند که در برگیرنده‌ی x_i است.

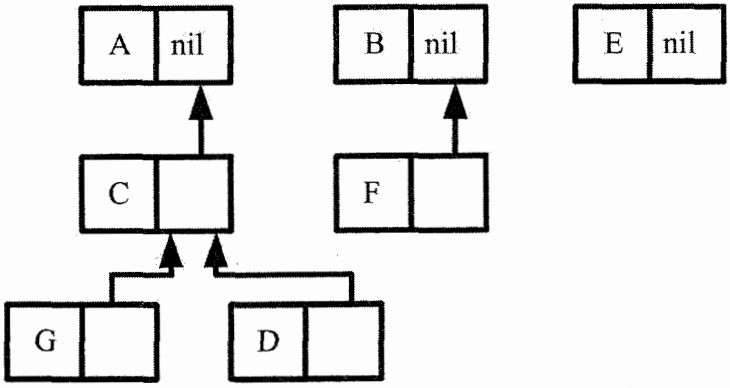
union(A,B): گروه A و B را با هم ترکیب می‌کند تا یک گروه تازه با نامی یکتا به وجود آورد (نام‌ها نباید تکراری باشند).

هدف، طراحی ساختاری است که هر دنباله‌ای از این دو عمل را به کارآمدترین شیوه‌ی ممکن پشتیبانی کند.

از آنجا که همه‌ی عناصر، پیشاپیش شناخته‌شده (و از ۱ تا n اندیس‌گذاری‌شده) هستند، می‌توان آرایه‌ی X[1..n] را به آن‌ها تخصیص داد. راه سرراست حل مسأله، ذخیره‌ی نام گروه دربرگیرنده‌ی عنصر iام در X[i] است. روشن است که عمل find به سادگی با نظر به آرایه انجام می‌شود، اما عمل

union نیاز به زمان بیش‌تری دارد. فرض کنیم نتیجه‌ی union(A,B) گروهی ترکیبی با نام A شود؛ در این صورت، لازم است هرجا که نام گروهی B است، آن را به A تغییر دهیم.

اینک، روشی متفاوت برای حل این مسأله ارائه می‌کنیم. به جای ساده‌سازی عمل find، عمل union را با یاری نشانی غیرمستقیم، ساده می‌کنیم. هر خانه‌ی آرایه، رکوردی است شامل نام عنصر و اشاره‌گری به یک رکورد دیگر. در آغاز، همه‌ی اشاره‌گرها nil هستند. عمل union(A,B) اشاره‌گر درون رکورد B را چنان تغییر می‌دهد که به رکورد شامل A اشاره کند و یا برعکس (به زودی در این مورد بحث خواهیم کرد). پس از چند عمل union، ساختمان‌داده به مجموعه‌ای از درخت‌ها مانند شکل ۴-۱۶ تبدیل خواهد شد. هر درخت، متناظر با یک گروه و هر گره، متناظر با یک عنصر است. نام هر گروه، از ریشه‌ی درخت متناظر با آن گرفته می‌شود. برای یافتن گروهی که شامل عنصر G است، از اشاره‌گر G شروع می‌کنیم تا به ریشه برسیم. (ریشه، گرهی است که اشاره‌گر آن nil است.) این فرایند، شبیه تغییر نشانی پستی است که در آن، به جای اعلام نشانی تازه به همه، نامه‌های نشانی پیشین به نشانی تازه فرستاده می‌شوند؛ البته یافتن نشانی درست دشوارتر می‌شود، یعنی کارایی عمل find کم‌تر می‌گردد. این ناکارآمدی هنگامی بیش‌تر می‌شود که عمل union، سبب ساخت درخت‌هایی بلند گردد.



شکل ۴-۱۶ نمایش مسأله‌ی union-find

ایده‌ی اصلی برای کارآمد کردن این ساختمان‌داده، متوازن ساختن و هرس کردن درخت‌هاست. به تازگی دیدیم که می‌ارزد، زمانی بیش‌تر را صرف توازن ساختمان‌داده کنیم. عمل union(A,B) را در شکل ۴-۱۶ در نظر بگیرید. دو حالت ممکن است: یا اشاره‌گر B را به گونه‌ای تنظیم می‌کنیم که به A اشاره کند، یا اشاره‌گر A را به گونه‌ای تنظیم می‌کنیم که به B اشاره کند. روشن است که گزینه‌ی نخست منجر به تشکیل درختی متوازن‌تر می‌گردد. پس در رکورد متناظر با ریشه، علاوه بر نام گروه، تعداد عناصر آن را نیز نگه‌داری می‌کنیم تا به سرعت تشخیص دهیم کدام درخت متوازن‌تر است.

تعریف توازن: هنگام انجام عمل union، اشاره‌گر گروه کوچک‌تر به گونه‌ای تنظیم می‌شود که به گروه بزرگ‌تر اشاره کند (هنگام هم‌اندازه بودنِ هر دو گروه، یکی را به

دل‌خواه برمی‌گزینیم). اندازه‌ی گروه ترکیبی حاصل نیز محاسبه شده، در میدان مناسبی از رکورد ریشه قرار می‌گیرد.

اگر عمل union، بنا به تعریف توازن (چنان که گفته شد) رفتار کند، ارتفاع درخت‌ها هرگز از $\log_2^n$ بیش‌تر نخواهد شد. این موضوع در قضیه‌ی ۴-۲ نشان داده شده است.

□ قضیه‌ی ۴-۲

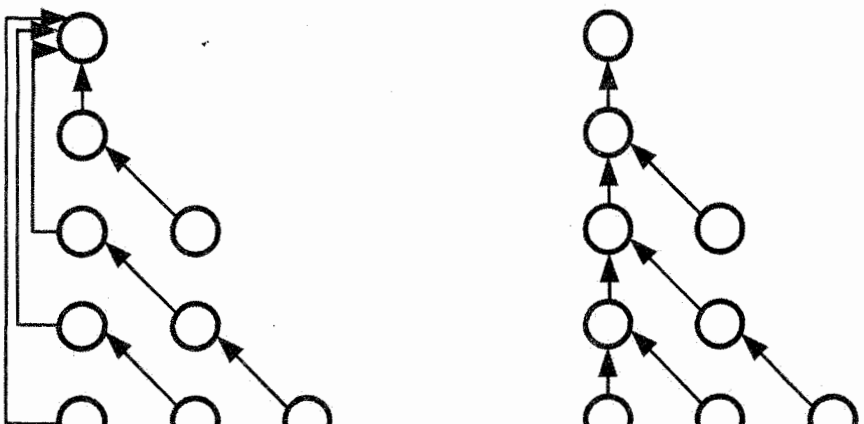
هر درخت متوازن به ارتفاع h، دست‌کم 2^h عنصر دارد.

برهان: اثبات با استقرا روی تعداد اعمال union است. روشن است که قضیه برای نخستین union که سبب ایجاد درختی با دو عنصر و به ارتفاع یک می‌شود، درست است. عمل union(A,B) را در نظر بگیرید و فرض کنید A، گروه بزرگ‌تر باشد؛ یعنی B باید به A اشاره کند. ارتفاع درخت‌های متناظر با گروه‌های A و B را به ترتیب با h(A) و h(B) نشان می‌دهیم. ارتفاع درخت ترکیبی حاصل، بیشینه‌ی h(A) و h(B)+1 است. اگر h(A) بزرگ‌تر باشد، آنگاه درخت ترکیبی حاصل، هم‌ارتفاع با درخت A ولی با تعداد عناصر بیش‌تری است؛ در این حالت، به روشنی قضیه برقرار است. در حالت دیگر، تعداد عناصر درخت ترکیبی حاصل، دست‌کم دو برابر تعداد عناصر درخت B (زیرا B کوچک‌تر از A فرض شده بود) و ارتفاعش یکی بیش از ارتفاع اولیه‌ی B است. بازهم می‌بینیم که قضیه برقرار است.

□

از قضیه‌ی ۴-۲ نتیجه می‌شود که عمل find، حداکثر از $\log_2^n$ اشاره‌گر عبور می‌کند. عمل union، همواره زمان ثابتی می‌گیرد. در نتیجه، حداکثر گام‌های هر دنباله‌ای از m عمل (چه union، چه find) هنگامی که $m \geq n$، از O(mlogn) خواهد بود.

می‌توان کارایی ساختمان‌داده‌ی union-find را بهبود بخشید. دوباره مثال فرستادن نامه‌های یک نشانی پستی به یک نشانی دیگر را در نظر بگیرید. اگر چندین تغییر نشانی روی دهد، نامه از یک نشانی به نشانی دیگر می‌رود تا سرانجام به مقصد برسد. خوب است به همه‌ی ایستگاه‌های پستی که عمل ارسال نامه را به نشانی بعدی انجام می‌دهند، اعلام کنیم که مقصد نهایی کجاست. در این صورت، این ایستگاه‌ها می‌توانند نامه‌ها را یک‌راست به مقصد نهایی بفرستند. در مورد این ساختمان‌داده، ما می‌توانیم پس از پیمایش اشاره‌گرها از یک رکورد به ریشه، اشاره‌گرهای درون مسیر را به گونه‌ای تغییر دهیم که مستقیماً به ریشه اشاره کنند (شکل ۴-۱۷ را ببینید). به این عمل، فشرده‌سازی مسیر گفته می‌شود. پیمایش دوباره‌ی مسیر برای انجام این کار، تعداد گام‌ها را دو برابر می‌کند؛ بنابراین پیچیدگی مجانبی زمان عمل find همان مقدار پیش خواهد بود. می‌توانیم هر بار که عمل find را انجام دادیم، فشرده‌سازی مسیر را نیز انجام دهیم. قضیه‌ی بعد که اثباتش نخواهیم کرد، حد بالای خوبی برای بدترین حالت ارائه می‌دهد.



(الف) (ب)

شکل ۴-۱۷ (الف) پیش از فشرده‌سازی مسیر؛ (ب) پس از فشرده‌سازی مسیر (در این شکل، تنها یک مسیر فشرده شده است - مترجمان)

قضیه‌ی ۴-۳

اگر هر دو عمل توازن و فشرده‌سازی با هم به کار گرفته شوند، آنگاه تعداد گام‌ها در بدترین حالت، برای هر دنباله‌ای از m عمل (خواه find، خواه union) که m≥n، از $O(m \log^* n)$ خواهد بود که $\log^* n$ تابع لگاریتم پی‌درپی است و این‌گونه تعریف می‌شود: $\log^* n = 1 + \log^*(\lceil \log_2^n \rceil)$ و برای هر n>2، log*1=log*2=1.

□

برای مثال، log*2=2، log*4=1+log*2=2، log*14=1+log*4=3 و log*16=1+log*4=4. log*60000=1+log*16=4. برای هر $n \le 2^{65536}$ (که تمام کاربردهای عملی را دربرمی‌گیرد) داریم: log*n≤5. بنابراین پیچیدگی هر دنباله‌ای از اعمال union و find تقریباً خطی است (و در عمل هم واقعاً خطی است). توجه کنید اگرچه هنوز هم یک عمل find خاص ممکن است به O(logn) گام نیاز داشته باشد، اما تعداد کل گام‌های O(n) عمل find از O(nlog*n) خواهد بود. این مورد، نمونه‌ای خوب از تحلیل سرشکن‌شده است. در این روش به جای محاسبه‌ی جداگانه‌ی تک‌تک گام‌ها، همه‌ی آن‌ها را با هم می‌شماریم. هنوز هم طراحی الگوریتمی با زمان خطی برای این مسأله، یک مسأله‌ی باز و حل‌نشده است.

۴-۶ گراف‌ها

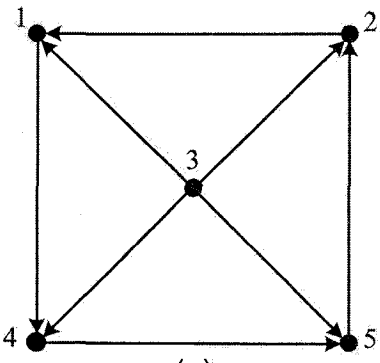
یک فصل کامل (فصل ۷) را به الگوریتم‌های گراف اختصاص خواهیم داد. در این بخش، درباره‌ی ساختمان‌داده‌هایی بحث می‌کنیم که برای ذخیره‌ی گراف‌ها به کار می‌روند. گراف G=(V,E) از

مجموعه‌ی رأس‌های (یا گره‌های) V و مجموعه‌ی یال‌های E تشکیل می‌شود و در آن، هر یال متناظر با یک جفت از رأس‌هاست. یال‌ها بیانگر رابطه‌ی بین رأس‌ها هستند. برای مثال، ممکن است گراف، مجموعه‌ای از افراد و یال‌ها، رابطه‌ی آشنایی آن‌ها با یکدیگر باشد. یک گراف، ممکن است جهت‌دار یا بدون‌جهت باشد. یال‌های گراف جهت‌دار، زوج‌هایی مرتب هستند؛ یعنی ترتیب اتصال رأس‌های یال مهم است. در این حالت، یک یال را با پیکانی از یک رأس (دم یا مبدأ) به رأس دیگر (سر یا مقصد) مشخص می‌کنیم. یال‌های یک گراف بدون‌جهت، زوج‌هایی نامرتب هستند. درخت‌ها، نمونه‌هایی ساده از گراف‌ها هستند. چنان‌چه بخواهیم وجود یک سلسله مراتب را در درخت نشان دهیم، همه‌ی یال‌ها را به گونه‌ای تنظیم می‌کنیم که از ریشه سرچشمه گرفته باشند. گاهی به چنین درخت‌هایی، درخت‌های ریشه‌دار می‌گویند، زیرا برای تعریف جهت یال‌ها کافی است ریشه را مشخص کنیم. درخت‌های بدون‌جهت را نیز (که گاهی درخت‌های آزاد خوانده می‌شوند) می‌توان به کار برد. (این درخت‌ها، سلسله مراتب ندارند.)

در این کتاب، معمولاً یکی از دو شیوه‌ی ماتریس همسایگی یا لیست را برای ذخیره‌ی گراف به کار برده‌ایم. نخستین شیوه‌ی ذخیره‌ی گراف‌ها به‌کارگیری ماتریس همسایگی یا مجاورت است. اگر در گراف G=(V,E)، |V|=n، یعنی گراف، n رأس داشته باشد، ماتریس همسایگی گراف G، ماتریسی n×n است به گونه‌ای که $a_{ij}=1$ اگر و تنها اگر $(v_i, v_j) \in E$. پس سطر iام ماتریس، آرایه‌ای به اندازه‌ی n است که مقدار درایه‌ی jام، اگر یالی از v_i به v_j وجود داشته باشد، ۱ و گرنه ۰ خواهد بود. بدی ماتریس همسایگی، نیاز آن به فضایی به اندازه‌ی n^2 است (بدون توجه به تعداد یال‌های گراف). برای مثال، تعداد یال‌های یک درخت (با n رأس) n-1 است و می‌توان این یال‌ها را با یک یا دو اشاره‌گر به ازای هر رأس ذخیره کرد (بسته به آن‌که بخواهیم به سمت بالای درخت، یا به سمت پایین آن و یا به هر دو سمت حرکت کنیم). هر رأس گراف، آرایه‌ای به اندازه‌ی n در ماتریس همسایگی گراف دارد؛ به عبارت دیگر، اگر تعداد یال‌ها کم باشد، بیش‌تر درایه‌های ماتریس همسایگی ۰ خواهد بود.

به جای ذخیره‌ی تمامی درایه‌های ماتریس همسایگی می‌توانیم یک‌ها را (که بیانگر یال‌ها هستند) در یک لیست پیوندی نگه‌داری کنیم. در این حالت، به ازای هر یال یک اشاره‌گر وجود خواهد داشت. به این شیوه‌ی ذخیره، لیست همسایگی گفته می‌شود. در این شیوه‌ی ذخیره‌ی گراف، هر رأس با یک لیست پیوندی متناظر است که لیست پیوندی رأسی مانند a، در برگیرنده‌ی همه‌ی رأس‌های همسایه‌ی a است. معمولاًمجموعه‌ی این لیست‌ها را طبق برچسب رأس آغازین مرتب می‌کنیم. کل گراف با آرایه‌ای از لیست‌ها ذخیره می‌شود. هر خانه‌ی آرایه، برچسب (یا اندیس) یک رأس را مشخص می‌کند و اشاره‌گری به آغاز لیست رأس‌های همسایه‌ی آن رأس دارد. اگر بر روی گراف ثابتی کار کنیم (یعنی پس از ایجاد گراف، دیگر، اعمال درج یا حذف ممکن نباشد). می‌توان لیست‌ها را با آرایه‌هایی به این روش ذخیره کرد: یک آرایه به اندازه‌ی |V|+|E| در نظر می‌گیریم. در ابتدا، رأس‌ها را به ترتیب در |V| خانه‌ی نخست قرار می‌دهیم. هر یک از این خانه‌ها اندیسی از آرایه را در خود دارد. اگر مقدار

خانه‌ای، i باشد، همسایه‌های رأس متناظر با این خانه در اندیس‌های i، i+1، ... از همین آرایه قرار دارند. برای نمونه، اگر ۲۰ رأس و ۵۰ یال وجود داشته باشد و ۴ یال از رأس ۱ آغاز شده باشند، آنگاه اولین خانه، ۲۱ (همیشه به صورت |V|+1) و دومین خانه، ۲۵ خواهد بود. در خانه‌های متناظر با این یال‌ها، شماره‌ی رأس‌های پایانی یال‌ها قرار دارد. در این مثال، اگر دومین یال از دومین رأس به پنجمین رأس اشاره کند، آنگاه مقدار خانه‌ی ۲۶ برابر با ۵ خواهد بود. معمولاً یال‌ها به صورت مرتب نگه‌داری می‌شوند، هرچند همیشه لازم نیست که چنین باشد. این سه شیوه‌ی نمایش در شکل ۴-۱۸ توضیح داده شده‌اند. معمولاً کار با ماتریس همسایگی آسان‌تر و برنامه‌ی مربوط به آن نیز ساده‌تر است؛ اما هنگام کم بودن تعداد یال‌های گراف، کارایی لیست‌های همسایگی بسیار بهتر می‌شود. در عمل، تعداد یال‌های اکثر گراف‌ها بسیار کم‌تر از مقدار بیشینه، یعنی n(n-1)/2 برای گراف‌های بدون‌جهت و n(n-1) برای گراف‌های جهت‌دار است؛ بنابراین، لیست‌های همسایگی رایج‌تر هستند.



(ب)

	1	2	3	4	5
1	0	0	0	1	0
2	1	0	0	0	0
3	1	1	0	1	1
4	0	0	0	0	1
5	0	1	0	0	0

(الف)

6	7	8	12	13	4	1	1	2	4	5	5	2
1	2	3	4	5	6	7	8	9	10	11	12	13

(پ)

شکل ۴-۱۸ شیوه‌های ذخیره‌ی یک گراف

۴-۷ خلاصه

می‌توان ساختمان‌های داده‌ای را به دو دسته‌ی ایستا و پویا تقسیم‌بندی کرد. آرایه‌ها، ساختمان‌هایی ایستا هستند. باید اندازه‌ی لازم برای آرایه (و یا دست‌کم حد بالای مناسبی از آن) را داشته باشیم. پس از آن، دیگر نمی‌توانیم اندازه‌ی آرایه را افزایش دهیم. دسترسی به آرایه بسیار کارآمد است، اما لیست‌های پیوندی پویا هستند و می‌توان به آسانی اندازه‌ی آن‌ها را کم و زیاد کرد. لیست‌های پیوندی می‌توانند هر اندازه‌ای داشته باشند و تنها محدودیت آن‌ها مقدار حافظه‌ی در دسترس است.

ساختمان‌های داده‌ای به تک‌بعدی و چندبعدی نیز دسته‌بندی می‌شوند. آرایه‌ها و لیست‌های پیوندی تک‌بعدی هستند و تنها می‌توانند ترتیب بین عناصر را نمایش دهند. اطلاعات درخت‌ها اندکی بیش از یک ساختمان‌داده‌ی تک‌بعدی است (یعنی سلسله مراتب دارند). گراف‌ها می‌توانند ساختارهای پیچیده‌تری را نیز نمایش دهند؛ البته خودمان هم می‌توانیم آرایه‌ها و لیست‌های پیوندی چندبعدی بسازیم.

مفهوم داده‌گونه‌ی مجرد بسیار سودمند است. این مفهوم به ما امکان می‌دهد تا روی اعمال مورد نیاز ساختمان‌داده تمرکز کنیم؛ بدون آن‌که درگیر جزئیاتی از پیاده‌سازی شویم که وابسته به نوع داده است. چگونگی پیاده‌سازی فرهنگ‌های داده‌ای، صف‌های اولویت و ساختمان‌داده‌ی union-find را نیز توصیف کردیم.

اگر می‌خواهیم تنها خود داده‌ها را ذخیره کنیم و نیازمند توجه به ساختار آن‌ها نیستیم، درهم‌سازی بهترین گزینه است؛ اما اگر دست‌رسی، وابسته به چیزی فراتر از کلید صریح عناصر باشد، دیگر نمی‌توان از درهم‌سازی بهره گرفت؛ مثلاً اگر بخواهیم کوچک‌ترین کلید را در بین داده‌های یک جدول درهم بیابیم، باید سرتاسر جدول را بگردیم.

مراجعی برای مطالعه‌ی بیش‌تر

امروزه مطالعه بر روی ساختمان‌های داده‌ای، بخشی پایه‌ای در آموزش علوم رایانه شده است. در نتیجه، کتاب‌های فراوانی درباره‌ی ساختمان‌های داده‌ای وجود دارد. Knuth [۱۹۷۳a] و Knuth [۱۹۷۳b] شامل گنجینه‌ای از اطلاعات درباره‌ی ساختمان‌های داده‌ای هستند. کتاب‌های دیگری نیز در این زمینه وجود دارند؛ مانند Standish [۱۹۸۰]، Aho، Hopcroft و Ullman [۱۹۸۳]، Reingold و Hansen [۱۹۸۳]، Gonnet [۱۹۸۴] و Wirth [۱۹۸۶]. Tarjan [۱۹۸۳] مقاله‌ی مفصل و پیش‌رفته‌تری درباره‌ی ساختمان‌های داده‌ای و الگوریتم‌ها نگاشته است.

Jones [۱۹۸۶] مطالعه‌ای تطبیقی درباره‌ی ساختار بسیاری از صف‌های اولویت انجام داده است. در مقاله‌ی وی فهرست مفصلی از منابع مربوط به صف‌های اولویت وجود دارد. در کنار کارهای دیگران، Hibbard [۱۹۶۲]، الگوریتم‌های درج و حذف در درخت‌های دودویی جست‌وجو را توصیف کرده و در این مقاله ثابت شده است که پس از n درج تصادفی، میانگین طول مسیر، 2lnn خواهد بود. برای کسب اطلاع بیش‌تر، Knuth [۱۹۷۳b] را نیز ببینید. Eppinger [۱۹۸۳]، مطالعه‌ای تجربی روی آثار درج و حذف تصادفی در درخت‌های دودویی جست‌وجو انجام داده و حدس زده است طول مسیر میانگین، ممکن است از $O(\log^3 n)$ باشد. Culberson [۱۹۸۵] ثابت کرده است که در شرایط مشخصی، درج و حذف تصادفی سبب می‌شود که طول مسیر میانگین از $O(\sqrt{n})$ باشد. در Baer و Schwab [۱۹۷۷] مقایسه‌ای بین طرح‌ها و شیوه‌های مختلف توازن ارائه شده است. درخت‌های متوازن در Knuth

[۱۹۷۳b] و Tarjan [۱۹۸۳] نیز بررسی شده‌اند. Sleator و Tarjan [۱۹۸۵]، چندین شیوه‌ی تازه برای کار با درخت‌های خودتنظیم ارائه کرده‌اند. ایده‌ی این کار، انتقال گره‌هایی که به تازگی مورد دسترسی قرار گرفته‌اند، به بالای درخت است. هرچند این درخت‌ها همواره متوازن نیستند، اما کارایی خوبی در تحلیل سرشکن‌شده دارند؛ یعنی هرچند ممکن است گاهی انجام یک عمل وقت‌گیر باشد، اما در یک دوره‌ی زمانی بلند، میانگین زمان‌های انجام آن کوچک خواهد شد.

در Knuth [۱۹۷۳b] و Gonnet [۱۹۸۴] اطلاعات بیش‌تری درباره‌ی درهم‌سازی یافت می‌شود. در کتابی از Viter و Chen [۱۹۸۷] جزئیات دقیقی از راه‌بردی به نام درهم‌سازی یک‌پارچه بیان شده است. Carter و Wegman [۱۹۷۹] دسته‌هایی از توابع درهم‌ساز تصادفی را توصیف کرده‌اند که توابع درهم‌ساز عمومی نامیده می‌شوند. چندین کاربرد جالب از این مفهوم در Carter و Wegman [۱۹۷۹] و Karlin و Upfal [۱۹۸۶] و نیز در Krutz و Manber [۱۹۸۷] پیدا می‌شود. برخی روش‌های درهم‌سازی قابل گسترش نیز برای پشتیبانی از رشد پویای جدول‌های درهم وجود دارد؛ برای نمونه، Fagin، Nievergett، Pippenger و Strong [۱۹۷۹] و Litwin [۱۹۸۰] را ببینید.

نخستین بار Galler و Fischer [۱۹۶۴]، Fischer [۱۹۷۲]، Hopcroft و Ullman [۱۹۷۳] (Ullman) کسی است که نتیجه‌ی مورد اشاره در قضیه‌ی ۴-۳ را به دست آورده است) همراه با دیگران ساختمان‌داده‌ی union-find را بررسی کرده‌اند. Tarjan [۱۹۷۵] زمان اجرای union-find را تا $O(m\alpha(m,n))$ بهبود داده است که در آن $\alpha(n)$ وارون تابع Ackerman است و رشد آن حتا کندتر از $\log^* n$ است. Tarjan و van Leeuwen [۱۹۸۴] چندین گونه‌ی ساده‌تر از روش فشرده‌سازی مسیر را مطالعه کرده‌اند که برای اجرا به همان زمان نیاز دارند. برای کسب اطلاع بیش‌تر درباره‌ی گراف‌ها، فصل ۷ و کتاب‌شناسی آن را ببینید.

تمرین‌های آموزشی

۴-۱ برنامه‌ای برای حذف یک عنصر از لیست پیوندی بنویسید.

۴-۲ برنامه‌ای برای وارونه کردن جهت یک لیست پیوندی بنویسید. به عبارت دیگر، جهت تمامی اشاره‌گرها باید وارونه شود.

۴-۳ روال جست‌وجوی بازگشتی درخت‌های دودویی جست‌وجو را به روالی غیربازگشتی تبدیل کنید.

۴-۴ الگوریتمی طراحی کنید که همه‌ی کلیدهای یک درخت دودویی جست‌وجو را به ترتیب نمایش دهد.

۴-۵ یک هرم (با روش ضمنی) در آرایه‌ی A[1..16] ذخیره شده است. کوچک‌ترین هرمی که بتواند آرایه‌ای به اندازه‌ی ۱۶ را اشغال کند، چند عنصر دارد؟

۴-۶ الگوریتم Insert_to_Heap ممکن است چندین و چندبار عناصر را به بالای هرم حرکت دهد. الگوریتم را به گونه‌ای تغییر دهید که حداکثر یک جابه‌جایی انجام دهد (البته هنوز هم تعداد مقایسه‌ها می‌تواند از O(logn) باشد).

۴-۷ فرض کنید می‌خواهیم صف اولویت را به یاری درخت‌های AVL پیاده‌سازی کنیم. پیچیدگی همه‌ی اعمال را در این حالت حساب کنید.

۴-۸ اعداد ۱ تا ۲۰ را به ترتیب به یک درخت خالی AVL می‌افزاییم. درخت حاصل را نمایش دهید.

۴-۹ یک درخت AVL پیدا کنید که حذف یک گره‌اش، آن را به درختی غیر AVL تبدیل کند؛ به گونه‌ای که نتوان تنها با یک چرخش (خواه منفرد، خواه دوگانه) آن را به درخت AVL تبدیل کرد. این درخت را رسم کنید و گره مورد نظر را نشان دهید. سپس توضیح دهید که چرا نمی‌توان تنها با یک چرخش، درخت حاصل را متوازن ساخت.

تمرین‌های خلاقانه

۴-۱۰ یک داده‌گونه‌ی مجرد طراحی کنید که از این اعمال پشتیبانی کند:

Insert(x): حتا در صورت وجود عنصر x در ساختمان‌داده، درج باید انجام شود. به عبارت دیگر، این ساختار باید تکرار را بپذیرد.

Remove(y): عنصری را از ساختمان‌داده حذف کند و در y قرار دهد. هر عنصر را می‌توان حذف کرد. اگر چندین عنصر یکسان وجود داشته باشد، باید تنها یکی از آن‌ها حذف گردد.

چنین داده‌گونه‌ی مجردی، pool (یا bag) نامیده می‌شود و مثلاً برای ذخیره کردن مشاغل سودمند است. شغل‌های تازه پس از ایجاد به pool افزوده می‌شوند و هنگام وجود یک جویای کار، شغل به او داده می‌شود (یعنی شغل از ساختمان‌داده حذف می‌گردد). زمان هر عمل باید از O(1) باشد.

۴-۱۱ ساختار داده‌ای تمرین پیش را به گونه‌ای تغییر دهید که هر عنصر تنها یک بار بتواند در ساختمان‌داده ظاهر شود. پس، پیش از عمل درج باید وجود عنصر بررسی گردد. دیگر اعمال را مانند پیش پیاده‌سازی کنید، اما همگی باید تکراری بودن عناصر را بررسی کنند. پیچیدگی هر عمل در بدترین حالت چقدر است؟ رفتار چه ساختمان‌داده‌ای برای حالت میانگین خوب و مناسب است؟

۴-۱۲ نوع دیگری از ساختمان‌داده‌ی pool (تمرین ۴-۱۰ و ۴-۱۱) چنین است: فرض کنید همه‌ی عناصر با اعداد صحیح ۱ تا n مشخص شده‌اند و n آن قدر کوچک است که می‌توان حافظه‌ای به اندازه‌ی O(n) به ساختمان‌داده اختصاص داد. هر عنصر نیز حداکثر یک بار ظاهر می‌شود.

الگوریتم‌هایی برای Insert و Remove (همانند تمرین ۴-۱۰) طراحی کنید که زمان اجرای آن‌ها از O(1) باشد.

۴-۱۳ الگوریتمی برای ساخت یک هرم، از دو هرم موجود به اندازه‌های m و n طراحی کنید که همه‌ی عناصر دو هرم اولیه را در بر گیرد. دو هرم موجود به صورت لیست پیوندی ذخیره شده‌اند (هر گره اشاره‌گرهایی به دو فرزندش نیز دارد). زمان اجرای الگوریتم در بدترین حالت باید از O(log(m+n)) باشد.

۴-۱۴ الگوریتمی برای ساخت یک هرم طراحی کنید که همه‌ی عناصر k هرم موجود را در بر گیرد. پیچیدگی الگوریتم را محاسبه کنید.

۴-۱۵ داده‌گونه‌ی مجردی طراحی کنید که از این اعمال پشتیبانی کند:

Insert(x): اگر کلید x از پیش در ساختمان‌داده وجود نداشته باشد، آن را به این ساختار بیفزاید.

Delete(x): کلید x را (در صورت وجود) از ساختمان‌داده حذف می‌کند.

Find_Next(x): کوچک‌ترین کلیدی از ساختمان‌داده را پیدا می‌کند که از x بزرگ‌تر باشد.

زمان لازم برای همه‌ی این اعمال، در بدترین حالت باید از O(logn) باشد که در آن، n تعداد عناصر ساختمان‌داده است.

۴-۱۶ داده‌گونه‌ی مجردی طراحی کنید که از این اعمال پشتیبانی کند:

Insert(x): تنها در صورتی کلید x به ساختمان‌داده افزوده می‌شود که از پیش در آن وجود نداشته باشد.

Delete(x): کلید x را (در صورت وجود) از ساختمان‌داده حذف می‌کند.

Find-Smallest(k): kامین کلید را از نظر کوچکی، در ساختمان‌داده می‌یابد.

زمان لازم برای همه‌ی این عمل‌ها باید در بدترین حالت از O(logn) باشد که در آن، n تعداد عناصر ساختمان‌داده است.

۴-۱۷ داده‌گونه‌ی مجردی طراحی کنید که از این اعمال پشتیبانی کند:

Insert(x): تنها در صورتی کلید x به ساختمان‌داده افزوده می‌شود که از پیش در آن وجود نداشته باشد.

Delete(x): کلید x را (در صورت وجود) از ساختمان‌داده حذف می‌کند.

Find_Next(x,k): در بین کلیدهایی از ساختمان‌داده که از x بزرگ‌تر هستند، kامین آن‌ها را از نظر کوچکی می‌یابد.

در بدترین حالت، زمان همه‌ی این اعمال باید از O(logn) باشد که در آن، n تعداد عناصر ساختمان‌داده است.

۱۸-۴ ✯ آن دسته از الگوریتم‌های AVL که در بخش ۴-۳-۴ ارائه شده‌اند، به عامل‌های توازنی با سه مقدار ممکن نیاز دارند: ۱، ۰ و یا ۱-. برای ذخیره‌ی این سه مقدار به ۲ بیت اضافی در هر گره نیازمندیم. با تغییراتی اندک، شیوه‌ای برای پیاده‌سازی این الگوریتم‌ها پیش‌نهاد کنید که برای هیچ گرهی بیش از یک بیت اضافی به کار نرود.

۱۹-۴ عمل الحاق، دو مجموعه را می‌گیرد و آن‌ها را به هم می‌چسباند. کلیدهای یکی از مجموعه‌ها از کلیدهای مجموعه‌ی دیگر کوچک‌ترند. الگوریتمی برای الحاق دو درخت دودویی جست‌وجو طراحی کنید که زمان اجرای آن، در بدترین حالت از O(h) باشد. (h، بیشینه‌ی ارتفاع دو درخت است.)

۲۰-۴ الگوریتمی برای الحاق دو درخت AVL (مانند آنچه در تمرین ۴-۱۹ تعریف شد) طراحی کنید. زمان اجرای الگوریتم، در بدترین حالت باید از O(h) باشد (بازهم h، بیشینه‌ی ارتفاع دو درخت است).

۲۱-۴ یک درخت AVL را در نظر بگیرید که با دنباله‌ای نسبتاً تصادفی از درج‌ها و حذف‌ها پدید آمده است. فرض کنید احتمال ظهور همه‌ی عامل‌های توازن با هم برابر (یعنی ۱/۳) است و آنگاه ثابت کنید میانگین طول مسیر از گره بحرانی تا محل درج، مقداری ثابت و مستقل از اندازه‌ی درخت است.

۲۲-۴ ساختار کلی یک درخت AVL را که از درج مرتب اعداد ۱ تا n به وجود می‌آید، مشخص کنید. ارتفاع این درخت چقدر است؟

۲۳-۴ «بدترین درخت AVL» را بیابید؛ یعنی یک درخت AVL به ارتفاع h با کم‌ترین تعداد گره ممکن بسازید و از این درخت، با توجه به بیش‌ترین ارتفاع یک درخت AVL با n گره، برای اثبات قضیه‌ی ۴-۱ (بخش ۴-۳-۴) یاری بگیرید. (راه‌نمایی: روش بازگشتی را برای ساخت به کار گیرید.)

۲۴-۴ T_1 و T_2 را درخت‌هایی دل‌خواه با n گره در نظر بگیرید. ثابت کنید برای تبدیل T_1 به T_2، حداکثر به 2n چرخش نیاز است.

۲۵-۴ پیوند دو گراف بدون‌جهت G=(V,E) و H=(U,F) بنا به تعریف، گراف تازه‌ی J=(W,D) است که در آن $W = V \cup U$ (یعنی رأس‌های گراف تازه، رأس‌های هر دو گراف را در بر می‌گیرد) و $D = E \cup F \cup V \times U$ (یعنی یال‌های گراف تازه، یال‌های هر دو گراف را به همراه یالی از هر رأس V به هر رأس U در بر می‌گیرد). روش مناسبی برای ذخیره‌ی گراف‌ها پیش‌نهاد کنید که انجام کارآمد عمل پیوند را ممکن سازد.

۲۶-۴ $S=\{s_1, s_2, \ldots, s_m\}$ را مجموعه‌ای بسیار بزرگ بگیرید که به k بخش افراز شده است و فرض کنید روالی به نام which_block دارید که با گرفتن عنصر s_i، شماره‌ی بخش دربرگیرنده‌ی آن را در زمانی ثابت برمی‌گرداند (برای نمونه S ممکن است تمام نشانی‌ها در

ایالات متحده برحسب نام خیابان‌ها و هر بخش، یک کدپستی باشد). قرار است زیرمجموعه‌ای کوچک از S، یعنی T را نگه‌داری کنید و این اعمال را روی آن انجام دهید:

Insert(s_i)
Delete(s_i)
Delete_block(j): تمام عناصر متعلق به بخش j را از T حذف می‌کند.

در آغاز، T تهی است. زمان هر عمل در بدترین حالت باید از O(logn) باشد که در آن، n تعداد عناصر موجود در T است. Delete_block تنها، عناصر را از ساختمان‌داده جدا می‌کند و نیازی نیست که عملاً آن‌ها را حذف کند. m و k هر دو آن قدر بزرگند که نمی‌توانید از جدولی به اندازه‌ی m یا k بهره ببرید؛ اما n کوچک‌تر است و شما می‌توانید فضایی از O(n) را به کار گیرید.

۴-۲۷ ✯ A[1..n] را آرایه‌ای از اعداد حقیقی بگیرید. الگوریتم‌هایی برای انجام دنباله‌هایی از این اعمال طراحی کنید:

Add(i,y): مقدار y را به iامین عدد می‌افزاید.

Partial_Sum(i): جمع i عدد نخست، یعنی $\sum_{l}^{i} A[i]$ را برمی‌گرداند.

توجه کنید که تعداد عناصر ثابت است (یعنی هیچ عمل درج یا حذفی صورت نمی‌گیرد) و تنها، مقدار عنصرها تغییر می‌کند. تعداد گام‌های هر عمل باید از O(logn) باشد. برای فضای کاری از یک آرایه‌ی اضافی به اندازه‌ی n نیز می‌توانید بهره بگیرید.

۴-۲۸ ✯ ساختمان‌داده‌ی تمرین ۴-۲۷ را چنان گسترش دهید که از حذف و درج هم پشتیبانی کند. در این حالت، هر عنصر، یک کلید و یک مقدار دارد. دست‌رسی به عناصر با کلیدشان، اما عمل جمع بر روی مقدارشان صورت می‌گیرد. عمل Partial_Sum کمی متفاوت با تمرین پیش است:

Partial_Sam(y): حاصل‌جمع تمام عناصری از مجموعه را برمی‌گرداند که مقداری کم‌تر از y دارند (یعنی $\sum_{x_i<y} x_i$ را).

زمان اجرای بدترین حالت هنوز هم باید برای هر دنباله‌ای از O(n) عمل، از O(nlogn) باشد.

۴-۲۹ ساختمان‌داده‌ای برای نگه‌داری مجموعه‌ای از عناصر طراحی کنید. در اینجا، هر عنصر یک کلید و یک مقدار دارد. از این اعمال باید پشتیبانی شود:

Find_value(x): مقدار عنصر x را می‌یابد (اگر هم x در مجموعه نباشد، مقدار nil را برمی‌گرداند).
Insert(x,y)
Delete(x)

Add(x,y): مقدار y را به مقدار فعلی عنصری با کلید x می‌افزاید.

Add_all(y): مقدار y را به مقدار همه‌ی عناصر مجموعه می‌افزاید.

در بدترین حالت، زمان اجرای هر یک از این اعمال باید از O(logn) باشد.

۴-۳۰ (یک ماجرای واقعی) یک بار برنامه‌نویسی از کامپایلری تازه پیام خطایی دریافت کرد که نشان می‌داد کامپایلر هنگام ترجمه‌ی برنامه به فضایی بیش از حافظه‌ی موجود نیازمند است. برنامه‌نویس گیج شد، زیرا برنامه‌اش به حافظه‌ی زیادی نیاز نداشت. او توانست محل بروز مشکل را در یک دستور case پیدا کند (این دستور نشان داده شده است). بدون این دستور، برنامه بدون خطا ترجمه می‌شد، اما پس از افزودن آن، مشکل کمبود حافظه پیش می‌آمد. مشخص کنید کامپایلر چه ساختمان‌داده‌ای به کار می‌برد که دچار چنین مشکلی شده بود. (دستور case درست و معتبر است؛ اشکال در کامپایلر است که نتوانسته دستور case را ترجمه کند.)

```
case i of
1:      statement(1);
2:      statement(2);
4:      statement(3);
256:    statement(4);
65535: statement(5);
```

فصل ۵

طراحی استقرایی الگوریتم‌ها

ریشه‌یابی اختراعات آن قدر اهمیت دارد که به نظر من
از خود اختراعات هم جالب‌تر است.
G. W. Leibniz (۱۷۱۶-۱۶۴۶)

هر اختراعی موجب اختراعات دیگر می‌شود.
R. W. Emerson (۱۸۸۲-۱۸۰۳)

۵-۱ آشنایی

در این فصل با الگوبرداری از استقرای ریاضی، شیوه‌ی خودمان در طراحی الگوریتم را نشان می‌دهیم. به این منظور از مثال‌های نسبتاً ساده سود می‌جوییم تا اصول و روش‌های پایه‌ی این شیوه را معرفی کنیم. آنچه باید در مورد استقرا بدانیم، در فصل ۲ بیان شده است، اما به هنگام نیاز، مطلب را تکرار می‌کنیم تا این فصل خودکفا باشد.

استقرای ریاضی مانند دومینو است. تعدادی مهره‌ی دومینو را در نظر بگیرید که پشت‌سرهم چیده شده‌اند. می‌خواهیم تنها با انداختن نخستین مهره، تمام آن‌ها را واژگون سازیم. برای اطمینان از این رویداد کافی است مطمئن شویم که هر مهره پس از افتادن، مهره‌ی پس از خود را واژگون خواهد کرد. هر بار که مهره‌ی تازه‌ای به جمع مهره‌ها می‌افزاییم، دیگر لازم نیست تمام آن‌ها را بیندازیم تا از درستی عمل‌کرد مطمئن شویم. می‌توان از این اصل در طراحی الگوریتم هم بهره گرفت:

> لازم نیست گام‌های مورد نیاز برای حل مسأله را با دست خالی بپیماییم. کافی است تضمین کنیم که (۱) نمونه‌ی کوچکی از مسأله حل‌شدنی است (حالت پایه) و (۲) می‌توان راه‌حل نمونه‌های بزرگ‌تر مسأله را از روی حل نمونه‌های کوچک‌تر یافت (گام استقرا).

هنگامی که می‌خواهیم از این اصل بهره‌برداری کنیم، باید روی کاهش مسأله به مسأله‌ای کوچک‌تر (یا مجموعه‌ای از مسأله‌های کوچک‌تر) تمرکز کنیم؛ اما عموماً یافتن راهی برای کاهش اندازه‌ی مسأله چندان آسان نیست. در این فصل چندین روش را برای آسان کردن این فرایند نشان می‌دهیم. مثال‌های این فصل، نه به دلیل زیادی اهمیتشان در عمل (چراکه برخی از آن‌ها کاربرد اندکی دارند) بلکه به این

دلیل بررسی شده‌اند که ساده هستند و اصول مورد تأکید را به خوبی تشریح می‌کنند. در کتاب نمونه‌های بسیاری از این رویکرد ارائه خواهیم کرد.

۵-۲ محاسبه‌ی مقدار چندجمله‌ای‌ها

کار را با یک مسأله‌ی ساده‌ی جبری آغاز می‌کنیم: محاسبه‌ی مقدار یک چندجمله‌ای در یک نقطه.

مسأله: دنباله‌ای از اعداد حقیقی a_0، a_1، ...، a_{n-1} و a_n به همراه عدد حقیقی x داده شده است. مقدار چندجمله‌ای $P_n(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x+a_0$ را محاسبه کنید.

ممکن است این مسأله، نامزدی مناسب برای رویکرد استقرایی به نظر نرسد. با وجود این، نشان خواهیم داد که با استقرا می‌توان راه‌حل بسیار خوبی برای این مسأله یافت. کار را با ساده‌ترین (و تقریباً بدیهی‌ترین) روش آغاز می‌کنیم. سپس به دنبال شیوه‌هایی می‌گردیم که به راه‌حل بهتری می‌انجامند.

این مسأله، n+2 عدد دارد، ولی ما می‌خواهیم آن را براساس نمونه‌های کوچک‌تر خودش حل کنیم؛ یا به عبارت دیگر می‌کوشیم تا مسأله را به مسأله‌ای با اندازه‌ی کوچک‌تر کاهش دهیم. سپس به طور بازگشتی، آن را حل می‌کنیم؛ ما این شیوه را استقرایی می‌نامیم. نخستین ایده‌ای که به ذهن می‌رسد، کاهش مسأله به کمک حذف a_n است. در این صورت، مسأله‌ای که باقی می‌ماند، محاسبه‌ی این چندجمله‌ای است:

$$P_{n-1}(x)=a_{n-1}x^{n-1}+a_{n-2}x^{n-2}+\ldots+a_1x+a_0$$

این همان مسأله، با یک پارامتر کم‌تر است. بنابراین می‌توانیم مسأله را با استقرا حل کنیم:

فرض استقرا: می‌دانیم چگونه چندجمله‌ایِ متناظر با ورودی a_0، a_1، ...، a_{n-1} را در نقطه‌ی x محاسبه کنیم (یعنی چگونگی محاسبه‌ی $P_{n-1}(x)$ را می‌دانیم).

اینک از فرض استقرا برای حل این مسأله بهره می‌گیریم، اما نخست باید حالت پایه، یعنی محاسبه‌ی a_0 را انجام دهیم که روشن و بدیهی است. سپس باید نشان دهیم می‌توان مسأله‌ی اصلی (محاسبه‌ی $P_n(x)$) را به کمک حل مسأله‌ی کوچک‌تر (یعنی مقدار $P_{n-1}(x)$) حل کرد. در این مورد، این گام از حل مسأله سرراست است: محاسبه‌ی x^n، ضرب a_n در آن و افزودن نتیجه به $P_{n-1}(x)$:

$$P_n(x)=P_{n-1}(x)+a_nx^n$$

در اینجا ممکن است به کار بردن استقرا برای حل این مسأله، سبک‌سرانه به نظر برسد، زیرا حل مسأله‌ای بسیار ساده را پیچیده‌تر کرده‌ایم. الگوریتم این کار، صرفاً محاسبه‌ی چندجمله‌ای، از راست به چپ و به همان ترتیبی است که نوشته می‌شود، اما بعداً قدرت این روش بر ما آشکار خواهد شد.

هرچند این الگوریتم درست است اما کارآمد نیست. انجام این محاسبه به $n(n+1)/2$ (یعنی 1+2+...+n) ضرب و n جمع نیاز دارد. حال، به گونه‌ای دیگر از استقرا بهره می‌گیریم تا به راه‌حل بهتری برسیم.

نخستین تلاش برای بهبود راه‌حل با توجه به تعداد زیاد محاسبات تکراری انجام می‌شود. می‌توانیم هنگام محاسبه‌ی x^n با بهره‌گیری از مقدار محاسبه‌شده‌ی x^{n-1} در انجام عمل ضرب صرفه‌جویی کنیم. با وارد کردن محاسبه‌ی x^{n-1} در فرض استقرا آن را تغییر می‌دهیم:

فرض قوی‌تر استقرا: چگونگی محاسبه‌ی مقدار چندجمله‌ای $P_{n-1}(x)$ و x^{n-1} را می‌دانیم.

در این فرض، دانستن مقدار x^{n-1} آمده و آن را قوی‌تر کرده است. حالا گسترش فرض آسان‌تر است (زیرا محاسبه‌ی x^n آسان‌تر شده است). اینک برای محاسبه‌ی x^n، تنها به یک ضرب نیاز داریم و سپس ضربی دیگر انجام می‌دهیم تا a_nx^n به دست آید؛ آنگاه با یک عمل جمع، محاسبه را کامل می‌کنیم. (فرض استقرا خیلی قوی‌تر نیست، چراکه هنوز ناچاریم x^{n-1} را محاسبه کنیم.) در کل باید 2n ضرب همراه با n عمل جمع انجام شود. جالب است که با وجود آن که فرض استقرا به محاسبات بیش‌تری نیاز دارد، اما درکل به محاسبات کم‌تری نیاز خواهیم داشت. بعداً به این نکته بازمی‌گردیم و دوباره آن را به کار می‌بریم. این الگوریتم براساس تمام معیارها مناسب به نظر می‌رسد، زیرا کارآمد و ساده بوده، محاسبه‌اش هم آسان است. به هر حال، الگوریتمی بهتر از این هم وجود دارد. با بهره‌گیری از استقرا به روشی دیگر، این مطلب روشن خواهد شد.

یک گام سرراست برای کاهش مسأله، حذف آخرین ضریب، یعنی a_n است، اما برای کاهش اندازه‌ی مسأله راه‌های دیگری هم وجود دارد. ما می‌توانیم نخستین ضریب، یعنی a_0 را حذف کنیم؛ به این ترتیب، مسأله‌ی کوچک‌تر، محاسبه‌ی یک چندجمله‌ای با ضرایب a_n، a_{n-1}، ... و a_1 است؛ یعنی:

$$P'_{n-1}(x)=a_nx^{n-1}+a_{n-1}x^{n-2}+\ldots+a_1$$

(توجه کنید که حالا a_n، ضریب x^{n-1}، a_{n-1}، ضریب x^{n-2} و ... هستند.) بدین‌ترتیب، فرض تازه‌ی استقرا چنین خواهد شد:

فرض استقرا (با ترتیب وارونه): می‌دانیم چگونه مقدار چندجمله‌ایِ مشخص‌شده با ضرایب a_1، a_2، ...، a_{n-1} و a_n را در نقطه‌ی x محاسبه کنیم (یعنی چگونگی محاسبه‌ی $P'_{n-1}(x)$ را می‌دانیم).

این فرض برای دست‌یابی به هدف مورد نظرمان مناسب‌تر است، چراکه راحت‌تر گسترش می‌یابد (روشن است که $P_n(x)=x.P'_{n-1}(x)+a_0$). بنابراین برای محاسبه‌ی $P_n(x)$ از روی $P'_{n-1}(x)$، تنها به یک عمل ضرب و یک عمل جمع نیازمندیم. می‌توان الگوریتم کامل را با این عبارت توصیف کرد:

$$a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x+a_0=((\ldots((a_nx+a_{n-1})x+a_{n-2})\ldots)x+a_1)x+a_0$$

این الگوریتم به افتخار ریاضی‌دان انگلیسی، W.G. Horner، روش Horner نامیده می‌شود. (این الگوریتم را روش Newton هم می‌گویند. به Knuth [۱۹۸۱] صفحه‌ی ۴۶۷ مراجعه کنید.) این الگوریتم در شکل ۵-۱ نشان داده شده است.

الگوریتم: Polynomial_Evaluation(ā,x)

ورودی: $\bar{a}=a_0,a_1,a_2,\ldots,a_n$ (ضرایب چندجمله‌ای) و x (یک عددحقیقی)

خروجی: P (مقدار چندجمله‌ای به ازای x)

```
begin
    P := a_n;
    for i := 1 to n do
    P := x*P + a_{n-i}
end
```

شکل ۵-۱ الگوریتم Polynomial_Evaluation

پیچیدگی: این الگوریتم، تنها به n عمل ضرب و n عمل جمع و یک محل اضافی در حافظه نیاز دارد. پس با این که راه‌حل پیش ساده و کارآمد بود، اما دیدیم می‌ارزید در پی الگوریتم بهتری باشیم. این الگوریتم هم سریع‌تر است و هم برنامه‌ی ساده‌تری دارد.

توجه: استقرا به ما امکان می‌دهد تا بر گسترش راه‌حل‌های زیرمسأله‌های کوچک‌تر برای حل مسأله‌های بزرگ‌تر تمرکز کنیم. فرض کنید می‌خواهیم مسأله‌ی P(n) را حل کنیم. توجه کنید که مسأله‌ی P به پارامتر n (معمولاً اندازه‌ی مسأله) وابسته است. کار را با نمونه‌ای دل‌خواه از P(n) آغاز می‌کنیم و می‌کوشیم تا با فرض این که هم‌اینک P(n-1) حل شده است، آن نمونه را حل کنیم. راه‌های بسیاری، هم برای تعریف فرض استقرا و هم برای بهره‌گیری از این تعریف‌ها وجود دارد. بسیاری از این شیوه‌ها را بررسی خواهیم کرد و توانایی آن‌ها را در طراحی الگوریتم نشان خواهیم داد.

همین مثال ساده، لزوم نرمش و انعطاف‌پذیری ما را در بهره‌گیری از استقرا مشخص می‌کند. ترفندی که به روش Horner منجر شد، بررسی ورودی ازچپ به راست، به جای بررسی شهودی و رایج آن از راست به چپ بود. شیوه‌ی رایج دیگر، بررسی بالا به پایین به جای بررسی پایین به بالاست (هنگامی که ساختاری درختی مورد نظر باشد). راه دیگر، افزایش گام‌ها به صورت دوتایی (یا بیش‌تر) به جای افزایش تک واحدی گام‌های استقراست. از حالت‌های فراوان دیگری نیز می‌توان بهره گرفت. به علاوه، گاهی بهترین دنباله‌ی گام‌های استقرا، برای همه‌ی ورودی‌ها یکسان نیست. گاهی می‌ارزد برای یافتن بهترین روش کاهش مسأله نیز یک الگوریتم طراحی کنیم. در آینده، مثال‌هایی را از یکایک این روش‌ها خواهیم دید.

۵-۳ بزرگ‌ترین زیرگراف القایی

فرض کنید می‌خواهید برای دانشمندانی با علائق گوناگون یک همایش برگزار کنید و نام افرادی را که باید دعوت کنید، در یک فهرست قرار داده‌اید. هر یک از این افراد در صورتی حاضر به شرکت در این همایش خواهند شد که بتوانند به اندازه‌ی کافی با دیگران تبادل نظر کنند. همراه با نام هر دانشمند،

لیستی نیز از نام دانشمندان دیگری را که این دانشمند دوست دارد با آن‌ها گفت‌وگو کند، نگه‌داری می‌کنید. شما می‌خواهید تا جای ممکن، دانشمندان بیش‌تری را به همایش دعوت کنید، اما باید تضمین کنید که هر یک از آن‌ها می‌تواند دست‌کم با k نفر دیگر بحث و گفت‌وگو کند (k عددی ثابت و مستقل از تعداد دعوت‌شدگان است). شما کاری به بحث‌ها ندارید؛ مثلاً لازم نیست از وجود زمان کافی برای انجام تبادل‌نظرها مطمئن شوید. هدف شما کشاندن همه‌ی افراد به همایش است. چگونه تعیین می‌کنید که چه کسی را باید دعوت کنید؟ این مسأله، مسأله‌ای در نظریه‌ی گراف است: G=(V,E) را گرافی بدون‌جهت در نظر بگیرید. زیرگراف القایی در G، گراف H=(U,F) است به گونه‌ای که $U \subseteq V$ و F هر یالی از E را که رأس‌هایش در U باشد، در بر گیرد. درجه‌ی هر رأس، تعداد رأس‌های همسایه‌ی آن است. رأس‌های گراف، متناظر با دانشمندان و اتصال بین دو رأس، بیانگر وجود امکان تبادل نظر دو دانشمند و یک زیرگراف القایی، متناظر با زیرمجموعه‌ای از دانشمندان است.

مسأله: گراف بدون‌جهت G=(V,E) و عدد صحیح k داده شده‌اند. یا در G، زیرگراف القایی H=(U,F) را با بزرگ‌ترین اندازه‌ی ممکن به گونه‌ای بیابید که همه‌ی رأس‌های H درجه‌ای بزرگ‌تر یا مساوی k (البته در گراف H) داشته باشند و یا ثابت کنید چنین زیرگرافی وجود ندارد.

رویکردی مستقیم برای حل این مسأله، حذف رأس‌هایی است که درجه‌ی آن‌ها از k کم‌تر است. با حذف هر رأس و با توجه به یال‌های گذرنده از آن، ممکن است درجه‌ی دیگر رأس‌ها نیز کاهش یابد. هرگاه درجه‌ی رأسی از k کم‌تر شد، آن را حذف می‌کنیم. ترتیب این حذف‌ها روشن نیست؛ چراکه ممکن است نخست همه‌ی رأس‌هایی را که درجه‌ی آن‌ها از k کم‌تر است، حذف کنیم و سپس رأس‌هایی را که درجه‌ی آن‌ها کم شده است، بررسی کنیم و یا ممکن است نخستین رأس با درجه‌ی کم‌تر از k را حذف کنیم و سپس بررسی کنیم حذف این رأس روی کدام رأس‌ها تاثیر می‌گذارد. (این دو رویکرد متناظر با جست‌وجوی نخست-پهنا و جست‌وجوی نخست-ژرفا هستند که در بخش ۷-۳ آن‌ها را بیش‌تر بررسی خواهیم کرد.) آیا این دو رویکرد به نتیجه‌ای یکسان منتهی می‌شوند؟ آیا گراف حاصل، بزرگ‌ترین اندازه‌ی ممکن را دارد؟ پاسخ دادن به این پرسش‌ها آسان است. با رویکردی که در اینجا نشان می‌دهیم، پاسخ به این پرسش‌ها آسان‌تر هم خواهد شد.

به جای فکر کردن به جزئیات الگوریتمی برای حل این مسأله، به قضیه‌ای فکر کنید که نشان دهد چنین الگوریتمی وجود دارد. پیش‌نهاد می‌کنیم به دنبال اثبات رسمی نباشید (دست‌کم در مرحله‌ی نخست). ایده‌ی کار سرمشق قرار دادن گام‌هایی است که برای اثبات قضیه برمی‌داریم، تا به بینشی برای حل مسأله دست یابیم. باید به دنبال یک زیرگراف القایی یا بزرگ‌ترین درجه‌ی ممکن باشیم که شرایط گفته‌شده را برآورده سازد. یک «اثبات» استقرایی برای این مسأله چنین است:

فرض استقرا: می‌دانیم چگونه در گرافی که کم‌تر از n رأس دارد، بزرگ‌ترین زیرگراف القایی‌ای را بیابیم که درجه‌ی هر رأس آن بزرگ‌تر یا مساوی k باشد.

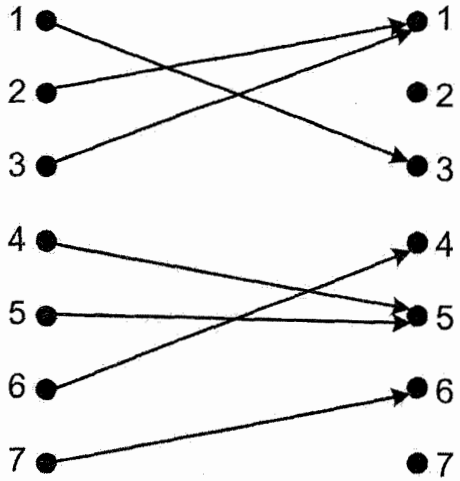
باید ثابت کنیم این «قضیه» برای یک «حالت پایه» درست است و از درستی‌اش برای n-1، درستی آن برای n نتیجه می‌شود. نخستین حالت پایه‌ی نه چندان روشن، هنگامی رخ می‌دهد که n=k+1، زیرا اگر n≤k، آنگاه درجه‌ی همه‌ی رأس‌ها کم‌تر از k خواهد بود. اگر n=k+1، آنگاه تنها راه ممکن برای k بودن درجه‌ی همه‌ی رأس‌ها این است که گراف، کامل باشد (یعنی همه رأس‌ها به هم متصل باشند) که کامل بودن گراف قابل بررسی است. پس فرض کنید G=(V,E)، گرافی با n رأس است که n>k+1. اگر درجه‌ی همه‌ی رأس‌ها بزرگ‌تر یا مساوی k باشد، کل گراف شرایط مورد نظر را دارد و انجام کار، بسیار ساده است وگرنه رأسی مانند v با درجه‌ی کم‌تر از k در گراف وجود دارد. روشن است که درجه‌ی رأس v در هر زیرگراف القایی، بازهم کم‌تر از k است. از این رو، v به هیچ زیرگرافی که شرایط مسأله را برآورده کند، تعلق ندارد. بنابراین می‌توانیم v و یال‌های گذرنده از آن را حذف کنیم، چون تأثیری در شرایط قضیه ندارند. پس از حذف v، گراف، n-1 رأس خواهد داشت - و بنا به فرض استقرا می‌توانیم مسأله را حل کنیم.

ما کارمان را انجام دادیم. اینک، پاسخ پرسش‌هایی که مطرح شد، روشن می‌شود و می‌توان الگوریتم حل مسأله را یافت. هر رأس با درجه‌ی کم‌تر از k را می‌توان حذف کرد. ترتیب حذف‌ها اهمیتی ندارد. گرافی که پس از این حذف‌ها بر جای می‌ماند، باید گرافی با بزرگ‌ترین اندازه‌ی ممکن باشد، زیرا انجام همه‌ی این حذف‌ها ضروری و لازم بود. روشن است که الگوریتم نیز درست است، چراکه طراحی آن با اثبات درستی‌اش همراه بود!

توجه: بهترین راه کاهش اندازه‌ی یک مسأله، حذف برخی از عناصر آن است. در این مثال، شیوه‌ی بهره‌گیری از استقرا، سرراست و مشخص بود، زیرا هم رأسی که باید حذف می‌شد و هم شیوه‌ی حذف آن روشن بود. در اینجا، کاهش اندازه‌ی مسأله به سادگی انجام شد؛ اما در حالت کلی ممکن است فرایند حذف چندان سرراست نباشد. در آینده نمونه‌هایی خواهیم دید که در آن‌ها با ترکیب (ادغام) دو عنصر در یکدیگر، تعداد عناصر را کاهش خواهیم داد (بخش ۶-۶). برای دیدن نمونه‌ای از حذف محدودیت‌های مسأله به جای حذف بخش‌هایی از ورودی آن، بخش ۷-۷ و برای دیدن نمونه‌ای از طراحی الگوریتمی ویژه برای یافتن عناصری که می‌توان حذف کرد، بخش ۵-۵ را ببینید. مثال دیگری از حذف درست عناصر در بخش بعد ارائه می‌شود. جالب است بدانیم که اگر در مسأله به جای عبارت «بزرگ‌تر یا مساوی»، عبارت «کوچک‌تر یا مساوی» قرار دهیم (یعنی اگر به دنبال بزرگ‌ترین گراف ممکن بگردیم که درجه‌ی همه‌ی رأس‌هایش حداکثر k باشد)، مسأله بسیار دشوارتر خواهد شد (تمرین ۱۱-۱۲ را ببینید).

۵-۴ یافتن نگاشت‌های یک‌به‌یک

اگر A مجموعه‌ای متناهی باشد، f را تابعی از A به A بگیرید (یعنی f هر عنصر A را به عنصر دیگری از A می‌نگارد). برای سادگی، عناصر A را با اعداد صحیح ۱ تا n نشان می‌دهیم. فرض کنید تابع f با آرایه‌ی f[1..n] نمایش داده شود، به گونه‌ای که f[i] مقدار f(i) را در خود داشته باشد (این مقدار عددی صحیح بین ۱ تا n است). f را نگاشتی یک‌به‌یک می‌نامیم، اگر برای هر عنصر j، حداکثر یک عنصر i به j نگاشته شده باشد. تابع f را می‌توان با یک نمودار، همانند شکل ۵-۲ نمایش داد که در آن، هر دو طرف، متناظر با مجموعه‌ای یکسان هستند و یال‌ها، نگاشت را نشان می‌دهند. روشن است که تابع شکل ۵-۲ یک‌به‌یک نیست.



شکل ۵-۲ نگاشتی از یک مجموعه به خودش (هر دو طرف به مجموعه‌ای یکسان تعلق دارند).

مسأله: مجموعه‌ی متناهی A و نگاشت f از A به A داده شده‌اند. زیرمجموعه‌ای از A، مانند S با بیش‌ترین تعداد عنصر ممکن بیابید، به گونه‌ای که (۱) f هر عنصر S را به عنصر دیگری از آن بنگارد (یعنی f نگاشتی از S به S باشد) و (۲) هیچ دو عنصر متمایزی از S به عنصر یکسانی نگاشته نشوند (یعنی f، در هنگام محدود شدن به S یک‌به‌یک شود). (در برخی کتاب‌ها، محدود شدن f به S را تحدید f به S می‌گویند - مترجمان)

اگر f از همان آغاز یک‌به‌یک باشد، کل مجموعه‌ی A شرایط مورد نظر مسأله را دارد و بی‌شک خود A، بزرگ‌ترین زیرمجموعه‌ی ممکن است؛ اما اگر دو عنصر متمایز i و j وجود داشته باشند که f(i)=f(j)، آنگاه S نمی‌تواند هر دو عنصر i و j را در بر گیرد. برای نمونه، چون در شکل ۵-۲ داریم: f(2)=f(3)=1، بنابراین در این مثال، مجموعه‌ی S نمی‌تواند هر دو عنصر ۲ و ۳ را در بر گیرد. برگزیدن یکی از این دو برای حذف نیز باید از روی حساب و کتاب باشد؛ مثلاً اگر ۳ را حذف کنیم، چون ۱ به ۳

نگاشته شده است، پس باید ۱ را هم حذف کنیم (چراکه ۳ دیگر در S نیست) اما اگر ۱ را حذف کنیم، به همان دلیل پیشین باید ۲ را نیز حذف کنیم؛ ولی زیرمجموعه‌ی حاصل بیشینه نیست. (روشن است که می‌توانستیم تنها ۲ را حذف کنیم.) پاسخ مسأله برای شکل ۵-۲، زیرمجموعه‌ی {1,3,5} است؛ اما چگونه می‌توان شیوه‌ای کلی برای گزینش عناصری یافت که باید در زیرمجموعه‌ی پاسخ قرار گیرند؟

خوش‌بختانه در تصمیم‌گیری برای چگونگی کوچک کردن مسأله، آزادی عمل داریم. ما می‌توانیم اندازه‌ی مسأله را، هم با یافتن عنصری که قطعاً به S تعلق دارد و هم با حذف عنصری که قطعاً به S تعلق ندارد، کاهش دهیم. در اینجا، از روش دوم یاری جسته‌ایم:

فرض استقرا: چگونگی حل مسأله را برای مجموعه‌هایی با n-1 عنصر می‌دانیم.

حالت پایه روشن است: اگر تنها یک عنصر در مجموعه وجود داشته باشد، باید به خودش نگاشته شود تا نگاشت، یک‌به‌یک باشد. حال، فرض کنید مجموعه‌ای با n عنصر داریم و می‌خواهیم زیرمجموعه‌ی S را در آن به گونه‌ای بیابیم که شرایط مسأله را برآورده کند. ادعا می‌کنیم اگر هیچ عنصری از A به عنصری مانند i نگاشته نشده باشد، آنگاه i به S تعلق نخواهد داشت (به عبارت دیگر، هر عنصر سمت راست نمودار، مانند i که یالی به آن متصل نیست، متعلق به S نخواهد بود) چراکه در غیر این صورت، یعنی اگر $i \in S$ و S، k عنصر داشته باشد، آنگاه حداکثر به k-1 عنصر نگاشت شده است. پس نگاشت، یک‌به‌یک نیست. بنابراین، اگر چنین عنصری وجود داشته باشد، آن را حذف می‌کنیم. اینک، به مجموعه‌ی n-1 عنصری A' (A'=A-{i}) رسیده‌ایم که f، آن را به خودش می‌نگارد. بنا به فرض استقرا، چگونگی حل مسأله را برای A' می‌دانیم. اگر هم چنین iیی وجود نداشته باشد، نگاشت یک‌به‌یک است؛ یعنی کار، انجام شده است.

این راه‌حل بر حذف i استوار است. ثابت کردیم i متعلق به S نیست. قدرت استقرا همین جاست: همین‌که عنصری را حذف کنیم و اندازه‌ی مسأله را کاهش دهیم، کار، انجام شده است. باید احتیاط کنیم که مسأله‌ی کاهش‌یافته تنها از نظر اندازه با مسأله‌ی اصلی تفاوت داشته باشد و بس. تنها شرط حاکم بر مجموعه‌ی A و تابع f این بود که f نگاشتی از A به A است. این شرط، پس از کاهش نیز برای مجموعه‌ی A-{i} برقرار است، چراکه عنصری وجود نداشت که به i نگاشته شده باشد. این الگوریتم، هنگامی که نتوان هیچ عنصری را حذف کرد، به پایان می‌رسد.

پیاده‌سازی: الگوریتم را به صورت بازگشتی توصیف کردیم. در هر گام، عنصری را که هیچ نگاشتی به آن صورت نگرفته است، می‌یابیم و آن را حذف می‌کنیم. کار را به طور بازگشتی ادامه می‌دهیم؛ هرچند لازم نیست پیاده‌سازی هم بازگشتی باشد. می‌توانیم برای هر عنصر i، یک شمارنده‌ی c[i] نگه‌داری کنیم. در آغاز، c[i] باید برابر با تعداد عناصری باشد که به عنصر i نگاشته شده‌اند. محاسبه‌ی c[i]ها در n گام و با بررسی تمام عناصر آرایه و افزایش شمارنده‌ی متناظر با آن‌ها انجام می‌شود. سپس همه‌ی عناصری را که شمارنده‌ی آن‌ها صفر است، بیرون می‌کشیم و در یک صف قرار می‌دهیم. در هر گام، عنصر ابتدای صف را بیرون می‌کشیم (این عنصر را j نامیده‌ایم) و c[f(j)] را یک واحد کم می‌کنیم؛ اگر

c[f(j)] برابر صفر شد، f(j) را هم در ته صف قرار می‌دهیم. الگوریتم، زمانی به پایان می‌رسد که صف، خالی شود. الگوریتم، در شکل ۵-۳ ارائه شده است.

الگوریتم: Mapping(f,n)

ورودی: f (آرایه‌ای از اعداد صحیح که مقدار آن‌ها بین ۱ تا n است.)

خروجی: S (زیرمجموعه‌ای از اعداد صحیح ۱ تا n به گونه‌ای که تحدید f به S یک‌به‌یک است.)

```
begin
    S := A; {A مجموعه‌ای از اعداد ۱ تا n است.}
    for j := 1 to n do c[j] := 0;
    for j := 1 to n do  c[f[j]] را یک واحد افزایش بده;
    for j := 1 to n do
        if c[j] = 0 then  j را در ته صف قرار بده;
    while صف خالی نیست do
            i را از ابتدا یا سر صف برداشته، حذف کن;
            S := S - {i};
            c[f[i]] را یک واحد کاهش بده;
            if c[f[i]] = 0 then  f[i] را در ته صف قرار بده
end
```

شکل ۵-۳ الگوریتم Mapping

پیچیدگی: بخش مقداردهی اولیه‌ی الگوریتم به O(n) عمل نیاز دارد. هر عنصر حداکثر یک بار در صف قرار می‌گیرد و حذف یک عنصر از صف نیز در زمانی ثابت انجام می‌شود. پس، تعداد کل گام‌ها از O(n) است.

توجه: در این مثال، کاهش اندازه‌ی مسأله را با حذف عناصری از یک مجموعه انجام دادیم. کوشیدیم تا آسان‌ترین روش را برای حذف عناصر بیابیم، بدون آن که شرایط مسأله را به هم بزنیم. از آنجا که تنها لازم بود تابع f نگاشتی از A به A باشد، برگزیدن عنصری که هیچ عنصر دیگری به آن نگاشته نشده باشد، انتخابی طبیعی بود.

۵-۵ مسأله‌ی ستاره‌ی مشهور

مثال بعد، یک تمرین رایج در طراحی الگوریتم‌هاست و نمونه‌ی خوبی از مسأله‌هایی است که برای حل آن‌ها نه تنها لازم نیست تمام داده‌ها بررسی شوند، بلکه حتا به بررسی بخش قابل توجهی از آن‌ها هم نیاز نداریم. ستاره‌ی «مشهور» در بین n نفر کسی است که همه او را می شناسند، اما او هیچ کس را نمی‌شناسد. هدف مسأله این است که اگر ستاره‌ی مشهوری در بین n نفر وجود دارد، با پرسش‌هایی در

قالب «بر من ببخشایید! آیا شما کسی را که آنجاست، می‌شناسید؟» ستاره‌ی مشهور را بیابیم. (فرض بر این است که همگی پاسخ‌ها درست هستند و همه، حتا فرد مشهور، به ما پاسخ خواهند داد.) می‌خواهیم پاسخ را با کم‌ترین تعداد پرسش ممکن بیابیم. از آنجا که در بین n عنصر می‌توان n(n-1)/2 زوج برگزید؛ اگر پرسش‌ها را بدون هیچ طرح و برنامه‌ای بپرسیم، در بدترین حالت، ممکن است n(n-1) پرسش لازم باشد. روشن نیست آیا می‌توان الگوریتمی یافت که در بدترین حالت ممکن، عملکرد بهتری داشته باشد یا نه.

می‌توانیم این مسأله را در قالب نظریه‌ی گراف بیان کنیم؛ یعنی گرافی جهت‌دار بسازیم که رأس‌هایش متناظر با افراد باشد و اگر فرد A، فرد B را بشناسد، یالی از A به B وجود داشته باشد. فرد مشهور چاهک گراف است. (در گراف جهت‌دار، رأسی با درجه‌ی ورودی n-1 و درجه‌ی خروجی ۰ را چاهک می‌گویند.) دقت کنید که هیچ گرافی نمی‌تواند بیش از یک چاهک داشته باشد. ورودی مسأله، یک ماتریس همسایگی n×n است (که در آن اگر شخص i، شخص j را بشناسد، درایه‌ی ij، ۱ و گرنه ۰ خواهد بود).

> **مسأله:** یک ماتریس همسایگی n×n داده شده است. مشخص کنید آیا یک i وجود دارد که همه‌ی درایه‌های ستون i، به جز درایه‌ی ii، ۱ و همه‌ی درایه‌های سطر i، به جز درایه‌ی ii، ۰ باشد.

حالت پایه، یعنی n=2 ساده است. طبق معمول به اختلاف مسأله برای n و n-1 نفر توجه می‌کنیم. فرض می‌کنیم بتوانیم با استقرا در بین n-1 نفر نخست، ستاره‌ی مشهور را بیابیم. چون حداکثر یک ستاره‌ی مشهور وجود دارد، پس سه حالت ممکن است: (۱) ستاره‌ی مشهور در بین n-1 نفر نخست است، (۲) فرد nام ستاره‌ی مشهور است و (۳) اصلاً ستاره‌ی مشهوری وجود ندارد. بررسی حالت نخست آسان است؛ تنها کافی است بررسی کنیم که شخص nام ستاره‌ی مشهور را بشناسد، اما ستاره‌ی مشهور او را نشناسد. دو حالت دیگر بسیار دشوارترند، زیرا بررسی ستاره‌ی مشهور بودن فرد nام، حتا ممکن است به 2(n-1) پرسش نیاز داشته باشد. اگر در گام nام، 2(n-1) پرسش بپرسیم، تعداد کل پرسش‌ها n(n-1) پرسش خواهد بود (که ما می‌کوشیم از انجام آن پرهیز کنیم). بنابراین به شیوه‌ی دیگری نیازمندیم.

ترفندی که اینجا به کار می‌گیریم، حل مسأله به صورت «معکوس» است. تعیین ستاره‌ی مشهور ممکن است دشوار باشد، اما احتمالاً تحقیق ستاره‌ی مشهور نبودنِ یک فرد آسان‌تر است. به علاوه، بی‌شک شمار ستارگان مشهور از شمار افراد عادی بیش‌تر نیست. اگر یکی از افراد را کنار بگذاریم، اندازه‌ی مسأله از n به n-1 کاهش می‌یابد. لازم هم نیست فردی مشخص را کنار بگذاریم و هر فردی را می‌توان کنار گذاشت. فرض کنید از A بپرسیم: «آیا B را می‌شناسی؟» اگر بشناسد، دیگر A ستاره‌ی مشهور نیست؛ اگر هم نشناسد، B ستاره‌ی مشهور نیست. پس تنها با یک پرسش می‌توانیم یکی از آن‌ها را کنار بگذاریم.

حال، دوباره سه حالت گفته‌شده را در نظر بگیرید. فردی دل‌خواه را نفر nام نمی‌گیریم، بلکه با به‌کارگیری ایده‌ی مطرح‌شده، یکی از A و B را کنار می‌گذاریم؛ سپس مسأله را برای n-1 نفر دیگر حل می‌کنیم. بدین‌ترتیب، خیالمان راحت است که حالت ۲ پیش نخواهد آمد، زیرا فرد کنار گذاشته‌شده، ستاره‌ی مشهور نیست. اگر هم حالت ۳ رخ دهد - یعنی در بین n-1 نفر اصلاً ستاره‌ی مشهوری وجود نداشته باشد - آنگاه در بین n نفر نیز ستاره‌ی مشهوری وجود نخواهد داشت. تنها، حالت ۱ باقی می‌ماند که آن هم آسان است. اگر در بین n-1 نفر، ستاره‌ی مشهوری وجود داشته باشد، تنها با دو پرسش دیگر می‌توان تعیین کرد که آیا آن فرد در کل مجموعه هم مشهور است یا نه. اگر او ستاره‌ای مشهور نباشد، پس در کل، هیچ ستاره‌ی مشهوری وجود ندارد.

الگوریتم حل مسأله چنین است: از A می‌پرسیم که آیا B را می‌شناسد و بنا به پاسخ او یکی از آن دو را کنار می‌گذاریم. فرض کنید A را کنار بگذاریم. سپس (به کمک استقرا) ستاره‌ی مشهور را در بین n-1 نفر باقی‌مانده می‌یابیم. چنان‌چه چنین فردی وجود نداشته باشد، الگوریتم به پایان می‌رسد؛ وگرنه بررسی می‌کنیم که A ستاره‌ی مشهور را بشناسد و ستاره‌ی مشهور او را نشناسد (که در این صورت، ستاره‌ی مشهور در بین n-1 نفر، در بین n نفر نیز ستاره‌ی مشهور است - مترجمان).

پیاده‌سازی: همانند الگوریتم بخش پیش، پیاده‌سازی تکراری الگوریتم ستاره‌ی مشهور از پیاده‌سازی بازگشتی آن کارآمدتر است. این الگوریتم دو مرحله دارد: در مرحله‌ی نخست، همه‌ی نامزدهای مشهور بودن را به جز یکی کنار می‌گذاریم. در مرحله‌ی بعد بررسی می‌کنیم آیا این نامزد، ستاره‌ی مشهور است یا نه. با n نامزد آغاز می‌کنیم و برای دست‌یابی آسان‌تر به هدفمان، فرض می‌کنیم که این نامزدها در یک پشته ذخیره شده باشند. در بین هر دو نامزد دل‌خواه، می‌توانیم تنها با پرسیدن یک پرسش، یعنی این که آیا یکی از آن‌ها دیگری را می‌شناسد یا نه، یکی را حذف کنیم. این کار را با دو نامزد ابتدای پشته آغاز می‌کنیم، به این صورت که یکی از آن‌ها را از پشته برمی‌داریم و در یک محل حافظه قرار می‌دهیم و آنگاه هر بار، عنصرِ این محل حافظه را با عنصر سرپشته در نظر می‌گیریم و با یک پرسش، یکی از آن دو را حذف می‌کنیم. در صورت حذف شدنِ عنصر این محل حافظه، عنصر سرپشته را به جای آن قرار می‌دهیم. پس در هر گام یک نامزد حذف‌شده، یکی دیگر باقی می‌ماند. تا هنگامی که پشته خالی نشده است، به این کار ادامه می‌دهیم. هنگامی که پشته خالی شود، یک نامزد باقی می‌ماند. حال، بررسی می‌کنیم که آیا نامزد باقی‌مانده، واقعاً ستاره‌ی مشهور است یا نه. این الگوریتم در شکل ۵-۴ ارائه گردیده است (دقت کنید که الگوریتم، کمی متفاوت پیاده‌سازی شده است و در آن، پشته به طور صریح با بهره‌گیری از اندیس‌های i، j و next پیاده‌سازی گردیده است).

الگوریتم: Celebrity(Know)

ورودی: Know (یک ماتریس Boolean و n×n)

خروجی: celebrity

```
begin
    i := 1;
    j := 2;
    next := 3;
    {در مرحله‌ی نخست همه‌ی نامزدها را به جز یکی حذف می‌کنیم.}
    while next ≤ n+1 do
        if Know[i,j] then i := next
        else j := next;
        next := next + 1;
        {از بین i و j یکی حذف می‌شود.}
    if i = n+1 then
      candidate := j
    else
       candidate := i;
    {حال باید بررسی کنیم آیا این نامزد، واقعاً ستاره‌ی مشهور است یا نه.}
    wrong := false;
    k := 1;
    Know[candidate,candidate] := false;
       {یک متغیر ساختگی برای انجام آزمایش}
    while not wrong and k ≤ n do
        if Know[candidate,k] then wrong := true;
        if not Know[k,candidate] then
          if candidate ≠ k then wrong := true;
        k := k + 1;
    if not wrong then celebrity := candidate
    else celebrity := 0  {وجود نداشتن ستاره‌ی مشهور}
end
```

شکل ۵-۴ الگوریتم Celebrity

پیچیدگی: حداکثر 3(n-1) پرسش پرسیده خواهد شد: n-1 پرسش در مرحله‌ی اول برای حذف n-1 نفر و سپس حداکثر 2(n+1) پرسش برای اطمینان از این که نامزد مورد نظر واقعاً ستاره‌ی مشهور باشد. توجه کنید که اندازه‌ی ورودی مسأله، n نیست، بلکه n(n-1) (یعنی به تعداد درایه‌های ماتریس همسایگی) است. این راه‌حل نشان می‌دهد در حالی که هر یک از n(n-1) درایه‌ی ماتریس می‌توانستند پاسخ مسأله باشند، اما با بررسی تنها O(n) درایه از ماتریس همسایگی، می‌توان ستاره‌ی مشهور را شناسایی کرد.

توجه: ایده‌ی اصلی این راه‌حل زیبا کاهش اندازه‌ی مسأله از n به n-1، با روشی هوشمندانه است. این مثال نشان می‌دهد گاهی می‌ارزد تلاشمان را (در این مورد، یک پرسش) صرف انجام کاهشی کارآمدتر کنیم. مسأله را ساده‌انگارانه با ورودی «دل‌خواهی» به اندازه‌ی n-1 آغاز نکنید که مجبور باشید حل آن را به n گسترش دهید؛ بلکه گاهی هم می‌توان ورودی «مشخصی» به اندازه‌ی n-1 را برگزید. نمونه‌های دیگری نیز خواهیم دید که در آن‌ها زمان قابل توجهی را برای گزینش ترتیب درست استقرا صرف می‌کنیم – و صرف این زمان، کار درستی است.

۵-۶ نمونه‌ای از یک الگوریتم تقسیم‌وحل: مسأله‌ی نمای افقی

تا اینجا، مسأله‌هایی از نظریه‌ی گراف و محاسبات عددی را بررسی کردیم، اما این مسأله با رسم شکل سر و کار دارد.

> **مسأله:** با داشتن محل و شکل دقیق چندین ساختمانِ «مستطیل‌شکل» در یک شهر، نمای افقیِ دوبعدی این ساختمان‌ها را با حذف خطوط پنهان رسم کنید.

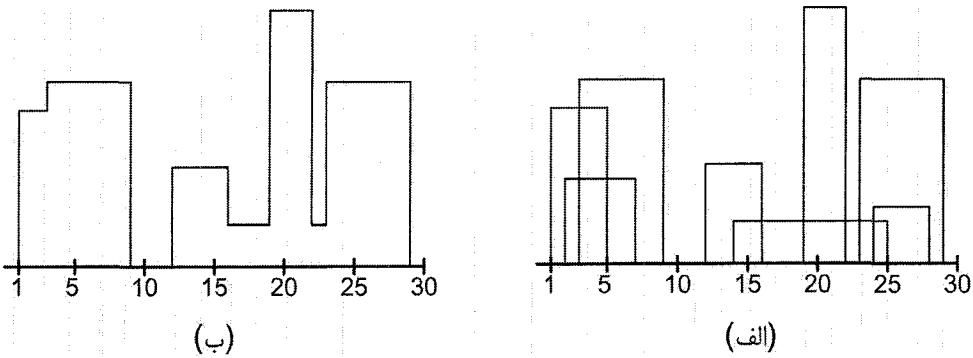
نمونه‌ی ورودی این مسأله در شکل ۵-۵ (الف) و خروجی آن در شکل ۵-۵ (ب) نشان داده شده‌اند. ما تنها به شکل دوبعدی علاقه‌مندیم. فرض می‌کنیم کف تمام ساختمان‌ها روی یک خط ثابت قرار گرفته باشد (یعنی افق یکسانی داشته باشند). ساختمان B_i با سه‌تایی (L_i,H_i,R_i) نشان داده می شود. L_i و R_i به ترتیب مؤلفه‌ی x از نقطه‌های انتهاییِ چپ و راست ساختمان و H_i، ارتفاع ساختمان را مشخص می‌کنند. یک «نمای افقی» فهرستی از مؤلفه‌های x به همراه ارتفاع‌هایی است که آن‌ها را از چپ به راست به هم متصل می‌کند. برای مثال، ورودی ساختمان شکل ۵-۵ (الف) چنین است:

(1,**11**,5), (2,**6**,7), (3,**13**,9), (12,**7**,16), (14,**3**,25), (19,**18**,22), (23,**13**,29), (24,**4**,28)

(اعداد تیره‌تر، نشانگر ارتفاع هستند.) نمای افقی حاصل از این ورودی، یعنی شکل ۵-۵ (ب) در خروجی باید این‌گونه نمایش داده شود:

(1,**11**,3,**13**,9,**0**,12,**7**,16,**3**,19,**18**,22,**3**,23,**13**,29,**0**)

(اینجا هم اعداد تیره‌تر بیانگر ارتفاع هستند.)



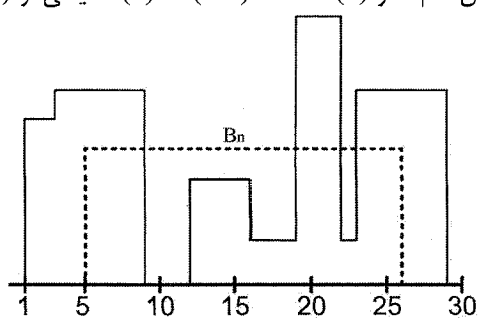
شکل ۵-۵ مسأله‌ی نمای افقی: (الف) ورودی (ب) خروجی

الگوریتم ساده‌ی این مسأله بر این مبناست که هر بار، یک ساختمان را به نمای افقی بیفزاییم. فرض استقرا ساده است: فرض می‌کنیم چگونگی حل مسأله را برای n-1 ساختمان می‌دانیم و سپس ساختمان nام را اضافه می‌کنیم. حل مسأله برای یک ساختمان روشن است. برای افزودن ساختمان B_n به نمای افقی، لازم است اشتراک B_n و نمای افقی را بیابیم (شکل ۵-۶ را ببینید). گیریم B_n، (5,**9**,26) باشد. نخست، نمای افقی را از چپ به راست می‌پیماییم تا به نقطه‌ی چپ B_n (۵، در این مثال) برسیم. در این مورد، یک خط افقی از ۳ به ۹ و به ارتفاع ۱۳، ۵ را می‌پوشاند. حال، می‌توانیم به نمای افقی نگاه کنیم و خطوط افقی آن را یکی پس از دیگری پویش کنیم و هنگام انجام این کار، در جایی که ارتفاع B_n از ارتفاع نمای افقی بیش‌تر است، ارتفاع نمای افقی را به ارتفاع B_n تغییر دهیم. هنگام پویش، پس از آن‌که مؤلفه‌ی x، بزرگ‌تر از مقدار متناظر با سمت راست B_n شد، کار به پایان می‌رسد. در اینجا، از ۳ تا ۹ ارتفاع را تغییر نمی‌دهیم، بلکه این کار را از ۹ تا ۱۹ انجام می‌دهیم و سپس از ۲۲ تا ۲۳ دوباره این کار را تکرار می‌کنیم. نمای افقی تازه با

(1,**11**,3,**13**,9,**9**,19,**18**,22,**9**,23,**13**,29,**0**)

نشان داده می‌شود.

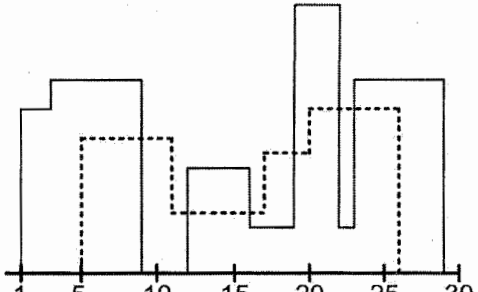
روشن است که الگوریتم درست است، اما لزوماً کارآمد نیست. در بدترین حالت، بررسی B_n به O(n) گام نیاز دارد. از این رو، تعداد کل گام‌ها از O(1)+...+O(n-1)+O(n)، یعنی از $O(n^2)$ است.



شکل ۵-۶ افزودن یک ساختمان (خط‌های نقطه‌چین) به نمای افقی شکل ۵-۵ (ب) (خط‌های توپر)

این الگوریتم را با روش شناخته‌شده‌ی «تقسیم‌وحل» بهبود می‌دهیم؛ یعنی به جای بهره‌گیری از صورت ساده‌ی استقرا برای گسترش حل مسأله از اندازه‌ی n-1 به اندازه‌ی n، مسأله‌ی با اندازه‌ی n را از روی مسأله‌ی با اندازه‌ی n/2 حل می‌کنیم. بازهم درستی حالتی که تنها «یک ساختمان» داریم، روشن است (حالت پایه). الگوریتم‌های تقسیم‌وحل، ورودی‌ها را به زیرمجموعه‌هایی کوچک‌تر تقسیم و هر زیرمجموعه را به صورت بازگشتی حل می‌کنند و سپس با ترکیب این راه‌حل‌ها، راه‌حل مسأله‌ی اصلی را می‌یابند. معمولاً با تقسیم مسأله به زیرمسأله‌هایی با اندازه‌ی تقریباً یکسان به کارایی بهتری دست خواهیم یافت. چنان که در فصل ۳ دیدیم، پاسخ رابطه‌ی بازگشتی $T(n)=T(n-1)+O(n)$ از $O(n^2)$، اما پاسخ $T(n)=2T(n/2)+O(n)$ از $O(nlogn)$ است. بنابراین، اگر مسأله را به دو زیرمسأله با اندازه‌ی مساوی تقسیم و سپس راه‌حل زیرمسأله‌ها را در زمانی خطی با هم ترکیب کنیم، زمان اجرای الگوریتم از $O(nlogn)$ خواهد شد. روش تقسیم‌وحل بسیار سودمند است و نمونه‌های فراوانی از آن را خواهیم دید.

ایده‌ی اصلیِ برگزیدن روش تقسیم‌وحل برای یافتن الگوریتم این مسأله، توجه به این نکته بود که در بدترین حالت، زمان ادغام مختصات یک ساختمان با یک نمای افقی، مانند زمان ادغام دو نمای افقیِ متفاوت با یکدیگر، خطی است. پس با بهره‌گیری از رویکرد دوم (یعنی ادغام دو نمای افقی) تقریباً در همان زمانِ رویکرد نخست، کار بیش‌تری انجام می‌شود. می‌توان با همان الگوریتم ادغامِ یک ساختمان و یک نمای افقی، دو نمای افقی را با یکدیگر ترکیب کرد (شکل ۵-۷). دو نمای افقی را از چپ به راست با یکدیگر مقایسه می‌کنیم، در آن‌ها مؤلفه‌های x را با یکدیگر تطبیق داده، در صورت نیاز، ارتفاع را تغییر می‌دهیم. می‌توان ادغام را در زمانی خطی انجام داد. پس، زمان اجرای کامل الگوریتم در بدترین حالت از $O(nlogn)$ است. از آنجا که این الگوریتم، شبیه مرتب‌سازی ادغامی است و این نوع از مرتب‌سازی در بخش ۶-۴-۳ به تفصیل مورد بررسی قرار خواهد گرفت؛ دیگر در اینجا، الگوریتم دقیق رسم نمای افقی را نمی‌آوریم.



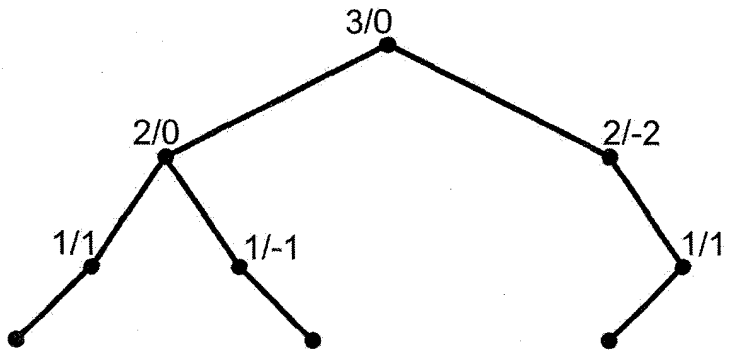
شکل ۵-۷ ادغام دو نمای افقی

توجه: همواره بکوشید تا با پولی که دارید، چیزهای بیش‌تری بخرید! این اصل نه مبهم است و نه فنی. اگر کاری که یک زیربخش الگوریتم انجام می‌دهد، فراتر از نیاز کل الگوریتم باشد، بکوشید آن زیربخش

را به قسمت پیچیده‌تری از الگوریتم اعمال کنید. رویکرد تقسیم‌وحل موفق است، چراکه در این روش می‌توان از تمام ظرفیت گام ترکیب سود جست. زمان اجرای اغلب الگوریتم‌های رایج تقسیم‌وحل را می‌توان با رابطه‌های بازگشتی بخش ۳-۵-۲ به دست آورد. باید این رابطه‌ها را به خاطر بسپارید.

۵-۷ محاسبه‌ی عامل‌های توازن در درخت‌های دودویی

T را یک درخت دودویی با ریشه‌ی r بگیرید. ارتفاع گره v، فاصله‌ی بین v و دورترین برگ در پایین درخت است. عامل توازن گره v را، اختلاف بین ارتفاع زیردرخت‌های چپ و راست آن گره تعریف می‌کنیم (فرض می‌کنیم که فرزندان یک گره را فرزند چپ و فرزند راست نامیده‌ایم). در فصل ۴ درباره‌ی درخت‌های AVL بحث کردیم که در آنجا، عامل توازن هر گره ۱-، ۰ یا ۱ بود؛ اما در این بخش، حالت کلیِ درخت‌های دودویی را در نظر می‌گیریم. شکل ۵-۸ درختی را نشان می‌دهد که هر گره‌اش با اعدادی به صورت h/b (h، ارتفاع گره و b، عامل توازن گره است) برچسب‌گذاری شده‌اند.



شکل ۵-۸ یک درخت دودویی. اعداد کنار هر گره به صورت h/b است که در آن h، ارتفاع گره و b، عامل توازن گره است.

مسأله: درخت دودویی T با n گره داده شده است. عامل توازن تمام گره‌های آن را حساب کنید.

از استقرایی معمولی کمک می‌گیریم:

فرض استقرا: شیوه‌ی محاسبه‌ی عامل توازن را برای همه‌ی گره‌های درختی که تعداد گره‌هایش کم‌تر از n است، می‌دانیم.

حالت پایه‌ی n=1 روشن است. اگر تعداد گره‌های درخت از ۱ بیش‌تر باشد (n>1)، ریشه را کنار گذاشته، سپس مسأله را (با استقرا) برای دو زیردرخت باقی‌مانده حل می‌کنیم. ریشه را، کنار گذاشتیم؛ چراکه عامل توازن هر گره، تنها به گره‌های پایین آن وابسته است و پس از کنار گذاشتن ریشه، عامل توازن همه‌ی گره‌های دیگر را می‌دانیم؛ اما عامل توازن ریشه وابسته به عامل توازن فرزندانش نیست،

بلکه به ارتفاع آن‌ها بستگی دارد. از این رو، در این مورد صورت ساده‌ی استقرا بـه کـار نمـی‌آیـد و لازم است ارتفاع فرزندان ریشه را نیز بدانیم. ایده‌ی حل مشکل چنین است که یافتن ارتفاع گره‌ها را نیـز بـه مسأله‌ی اصلی بیفزاییم.

فرض قوی‌تر استقرا: می‌دانیم چگونه عامل توازن و ارتفاع هـر گـره را در درختـی کـه کم‌تر از n گره دارد، محاسبه کنیم.

بازهم، حالت پایه روشن است. حالا هنگامی که عامل توازن ریشه را بخواهیم، می‌تـوانیم بـه آسـانی بـا محاسبه‌ی اختلاف بین ارتفاع فرزندانش آن را مشخص کنیم. ارتفاع ریشه را نیز - که برابر با یک واحد بیش از بیشینه‌ی ارتفاع دو فرزندش است - می‌توانیم تعیین کنیم.

نکته‌ی این الگوریتم در آن است که مسأله‌ای را که اندکی گسترده‌تـر است، حل می‌کند. به جای آن که تنها، عامل‌های توازن را محاسبه کنیم، ارتفاع گره‌ها را نیز همراه با آن‌ها محاسبه می‌کنـیم. حـل مسأله‌ی گسترش‌یافته آسان‌تر است، زیرا محاسبه‌ی ارتفاع گره‌ها دشوار نیست. در بسیاری مـوارد، حـل مسأله‌ی قوی‌تر آسان‌تر هم هست. با بهره‌گیری از استقرا، تنها لازم است حل یک مـسأله‌ی کوچـک را به حل مسأله‌ای بزرگ‌تر گسترش دهیم؛ هرچند به ظاهر باید کار بیش‌تری انجـام دهـیم (زیـرا مـسأله گسترش یافته است) اما از آنجا که چیزهای بیش‌تری نیز برای کار کردن در اختیـار داریـم، اثبـات گـام استقرا آسان‌تر می‌شود. نادیده گرفتن وجود پارامترهای مختلـف در ایـن مـسأله و ضـرورت محاسـبه‌ی جداگانه‌ی این پارامترها اشتباهی رایج است. در آینده نمونه‌های گوناگونی از این اشتباه نشان داده خواهد شد.

۵-۸ یافتن بزرگ‌ترین زیردنباله‌ی متوالی یا به‌هم‌پیوسته

حال به مسأله‌ای از Bentley [۱۹۸۶] (که در Bates و Constable [۱۹۸۵] نیز ارائـه گردیـده اسـت) توجه کنید.

> **مسأله:** دنباله‌ی x_1، x_2، ... و x_n از اعداد حقیقی داده شده است (ممکـن اسـت ایـن اعـداد مثبـت نباشـند). زیردنبالـه‌ای ماننـد x_i، x_{i+1}، ... و x_j (از عناصـر پـشت‌سـرهم) بیابیـد کـه حاصل‌جمع جمله‌های آن در بین تمام زیردنباله‌های به‌هم‌پیوسته، بیشینه باشد.

چنین زیردنباله‌ای را زیردنباله‌ی بیشینه گوییم. برای مثال، در دنبالـه‌ی ۲، ۳-، ۱/۵، ۱-، ۳، ۲-، ۳- و ۳ زیردنباله‌ی موردنظر ۱/۵، ۱- و ۳ است که حاصل جمع جمله‌هـای آن ۳/۵ مـی‌شـود. ممکـن اسـت در دنباله‌ی داده‌شده، چندین زیردنباله از این نـوع وجـود داشـته باشـد. اگـر اعـداد، همگـی منفـی باشـند، زیردنباله‌ی بیشینه تهی خواهد بود (بنـا بـه تعریـف، جمـع زیردنبالـه‌ی تهـی ۰ اسـت). دوسـت داریـم الگوریتمی داشته باشیم که تنها با یک بار خواندن اعداد، آن هم به ترتیب، مسأله را حل کند.

فرض استقرا: می‌دانیم چگونه زیردنباله‌ی بیشینه را در دنباله‌ای با اندازه‌ی کم‌تر از n بیابیم.

اگر دنباله تنها یک جمله داشته باشد و این جمله منفی نباشد، زیردنباله‌ی بیشینه همان یک عدد است، اما اگر n=1 و تنها جمله‌ی دنباله منفی باشد، زیردنباله‌ی بیشینه، زیردنباله‌ای تهی خواهد بود. دنباله‌ی S یعنی x_1، x_2، ... و x_n را در نظر بگیرید که بیش از یک جمله دارد (n>1). می‌دانیم چگونه به کمک استقرا زیردنباله‌ی بیشینه را در $S'=(x_1,x_2,\ldots,x_{n-1})$ بیابیم. اگر زیردنباله‌ی بیشینه تهی باشد، پس تمام اعداد 'S منفی هستند و کافی است تنها به x_n توجه کنیم. فرض کنید زیردنباله‌ی بیشینه‌ای که به کمک استقرا در 'S پیدا شده است، $S'_M=(x_i,x_{i+1},\ldots,x_j)$ به ازای دو مقدار خاص i و j باشد که $1\le i\le j\le n-1$. اگر j=n-1 (یعنی زیردنباله‌ی بیشینه پسوندی از دنباله‌ی اصلی باشد) آنگاه گسترش راه‌حل به S آسان است: اگر x_n مثبت باشد، به S'_M اضافه می‌شود، وگرنه خود S'_M بیشینه خواهد بود؛ اما اگر j<n-1، آنگاه دو حالت ممکن است: یا خود S'_M بیشینه است، یا زیردنباله‌ی دیگری وجود دارد که در 'S بیشینه نیست، اما پس از افزودن x_n به آن، به زیردنباله‌ی بیشینه تبدیل می‌شود.

ایده‌ی اصلی در اینجا، تقویت فرض استقراست. نخست با حل مسأله‌ی زیردنباله‌ی بیشینه، این روش را توضیح می‌دهیم؛ اما در بخش بعد بحث کلی‌تری در این مورد ارائه خواهیم کرد. در اینجا مشکل ما با فرض سرراست استقرا این بود که شاید زیردنباله‌ی بیشینه از افزوده شدن x_n به زیردنباله‌ای که در 'S بیشینه نیست، به وجود آید. پس، تنها شناسایی زیردنباله‌ی بیشینه‌ی 'S کافی نیست. به هر حال، x_n تنها می‌تواند به زیردنباله‌ای افزوده شود که در n-1 خاتمه یابد (یعنی زیردنباله‌ای که پسوندی از 'S باشد). پسوند بیشینه را با $S'_E=(x_k,x_{k+1},\ldots,x_{n-1})$ نشان می‌دهیم. تقویت فرض استقرا به یاری S'_E انجام می‌شود:

فرض قوی‌تر استقرا: در دنباله‌هایی که اندازه‌ی آن‌ها از n کوچک‌تر است، هم روش یافتن زیردنباله‌ی بیشینه در خود دنباله را می‌دانیم و هم شیوه‌ی یافتن زیردنباله‌ی بیشینه در بین تمام پسوندهای دنباله را.

الگوریتم حل مسأله، هنگامی که هر دو زیردنباله را می‌شناسیم، بسیار روشن است. x_n را به پسوند بیشینه می‌افزاییم. اگر حاصل‌جمع از زیردنباله‌ی بیشینه‌ی خود دنباله، بیش‌تر شد، یک زیردنباله‌ی بیشینه‌ی تازه (و یک پسوند بیشینه‌ی تازه) داریم. در غیر این صورت، زیردنباله‌ی بیشینه‌ی پیشین را نگه می‌داریم. البته هنوز کارمان تمام نشده است؛ چراکه لازم است پسوند بیشینه‌ی تازه را نیز بیابیم. همواره نمی‌توان x_n را به پسوند بیشینه‌ی پیشین افزود، چراکه ممکن است حاصل جمع منفی گردد. در چنین حالتی، بهتر است پسوند بیشینه را مجموعه‌ی تهی بگیریم (تا بعداً خود x_{n+1} به تنهایی به عنوان پسوند در نظر گرفته شود). الگوریتم یافتن مجموع زیردنباله‌ی بیشینه در شکل ۵-۹ ارائه شده است.

الگوریتم: Maximum_Consecutive_ Subsequence(X,n)

ورودی: X (آرایه‌ای به اندازه‌ی n)

خروجی: Global_Max (حاصل‌جمع زیردنباله‌ی بیشینه)

```
begin
    Global_Max := 0;
    Suffix_Max := 0;
    for i := 1 to n do
        if X[i] + Suffix_Max > Global_Max then
          Suffix_Max := Suffix_Max + X[i];
          Global_Max := Suffix_Max
        else if X[i] + Suffix_Max > 0 then
              Sufix_Max := X[i] + Suffix_Max
            else Suffix_Max := 0
end
```

شکل ۵-۹ الگوریتم Max_Consecutive_Subsequence

۵-۹ تقویت فرض استقرا

تقویت فرض استقرا یکی از مهم‌ترین شیوه‌های اثبات قضایای ریاضی به کمک استقراست. هنگام کوشش برای ارائه‌ی برهانی استقرایی، بسیار پیش می‌آید که با چنین ماجرایی روبه‌رو شویم: اگر قضیه با P و فرض استقرا با P(<n) نشان داده شوند، برهانی که می‌آوریم باید نشان دهد که $P(< n) \Rightarrow P(n)$. در بیش‌تر موارد، می‌توانیم فرض دیگری مانند Q را نیز به فرض استقرا بیفزاییم تا اثبات آسان‌تر گردد (اثبات $[P, Q](< n) \Rightarrow P(n)$ از اثبات $P(< n) \Rightarrow P(n)$ آسان‌تر است). در این موارد، با آن که فرض استقرا، درست به نظر می‌رسد، اما روشن نیست که چگونه می‌توانیم آن را ثابت کنیم. ترفند به کار برده‌شده این است که Q را نیز به فرض استقرا بیفزاییم، اما باید ثابت کنیم: $[P, Q](< n) \Rightarrow [P, Q](n)$. P و Q با یکدیگر، قضیه‌ای قوی‌تر از P هستند و اغلب اثبات آن‌ها با هم آسان‌تر است. این فرایند را می‌توان تکرار کرد و با افزودن فرضیات درست و مناسب، اثبات را ساده‌تر ساخت. مسأله‌ی زیردنباله‌ی بیشینه، نمونه‌ی خوبی از به‌کارگیری این اصل برای بهبود الگوریتم‌هاست.

نمونه‌ی خوبی از این اصل، یک موضوع شناخته‌شده است: افزایش ۱ میلیون دلار سود به فروشی ۱۰۰ میلیون دلاری، از افزایش ۱۰۰۰ دلار سود به فروشی ۱۰ دلاری آسان‌تر است.

رایج‌ترین اشتباهی که افراد هنگام به‌کارگیریِ این روش مرتکب می‌شوند، نادیده گرفتن لزوم سازگار کردن اثبات با فرضیات افزوده‌شده است. به عبارت دیگر، بدون توجه به افزودن شدن Q ثابت می‌کنند: $[P, Q](< n) \Rightarrow P(n)$. چنین غفلتی در مثال زیردنباله‌ی بیشینه، فراموش کردن

محاسبه‌ی پسوند بیشینه‌ی تازه و در مثال عامل توازن گره‌ها، فراموش کردن محاسبه‌ی جداگانه‌ی ارتفاع‌هاست – که بدبختانه اشتباهی رایج است. دیگر بیش از این نمی‌توانیم بر این واقعیت پافشاری کنیم که:

«پیروی دقیق از فرض استقرا، حیاتی و سرنوشت ساز است.»

نمونه‌های پیچیده‌تری از تقویت فرض استقرا در بخش‌های ۶-۱۱-۳، ۶-۱۳-۱، ۷-۵، ۸-۳، ۱۲-۳-۱ (و چند بخش دیگر) ارائه خواهیم کرد.

۵-۱۰ نمونه‌ای از برنامه‌نویسی پویا: مسأله‌ی کوله‌پشتی

فرض کنید یک کوله‌پشتی به ما داده‌اند و قرار است آن را با اشیایی کاملاً پر کنیم. ممکن است اشیای فراوانی با شکل‌ها و رنگ‌های گوناگون وجود داشته باشند و هدف، این باشد که کوله‌پشتی را تا جای ممکن، بیش‌تر پر کنیم. منظور از کوله‌پشتی می‌تواند کامیون، کشتی یا حتا تراشه‌ای از جنس سیلیکون و مسأله، پرکردن آن‌ها باشد. این مسأله گونه‌های مختلفی دارد، ولی در ابتدا نوع ساده‌ای از آن را در نظر می‌گیریم. دیگر گونه‌های این مسأله یا در تمرین‌ها و یا در فصل ۱۱ ارائه خواهند شد.

مسأله: عدد صحیح K و n عنصر با اندازه‌های مختلف مفروضند به گونه‌ای که اندازه‌ی عنصر iام k_i است. یا زیرمجموعه‌ای از این عناصر بیابید که جمع اندازه‌ی آن‌ها دقیقاً K شود و یا اعلام کنید که چنین زیرمجموعه‌ای وجود ندارد.

مسأله را با P(n,K) نشان می‌دهیم که در آن n، تعداد عناصر و K، ظرفیت کوله‌پشتی است. این n عنصر را ورودی ضمنی مسأله فرض کرده‌ایم، پس اندازه‌ی آن‌ها را در نمادگذاری مسأله نیاورده‌ایم. بنابراین، P(i,k)، مسأله را برای i عنصر نخست و یک کوله‌پشتی به ظرفیت k نشان می‌دهد. برای سادگی، در آغاز تنها روی «مسأله‌ی تصمیم‌گیری» متمرکز می‌شویم؛ یعنی این که آیا پاسخی برای مسأله وجود دارد یا نه. کار را با رویکرد استقرایی سرراست آغاز می‌کنیم. در نخستین تلاش، چنین فرضی برای استقرا در نظر می‌گیریم:

فرض استقرا (نخستین تلاش): چگونگی حل مسأله‌ی P(n-1,K) را می‌دانیم.

حالت پایه آسان است: تنها در صورتی که اندازه‌ی تک عنصر موجود K باشد، مسأله پاسخ دارد. اگر پاسخی برای P(n-1,K) وجود داشته باشد – یعنی راهی برای پر کردن کوله پشتی با همه، یا برخی از n-1 عنصر وجود داشته باشد – مسأله حل‌شدنی است؛ زیرا برای حل P(n,K) تنها کافی است عنصر nام را به کار نبریم؛ اما اگر پاسخی برای P(n-1,K) وجود نداشته باشد، آیا می‌توانیم از وجود نداشتن پاسخ P(n-1,K)، برای حل P(n,K) بهره‌برداری کنیم؟ بله؛ به این صورت که عنصر nام حتماً باید درون کوله‌پشتی قرار گیرد. در این حالت، عناصر دیگر باید بتوانند یک کوله‌پشتی با اندازه‌ی $K-k_n$ را پر

کنند. به این ترتیب، مسأله را به دو زیرمسأله‌ی کوچک‌تر کاهش داده‌ایم: P(n-1,K) و P(n-1,K-k_n). برای تکمیل راه‌حل، باید فرض استقرا تقویت شود. لازم است مسأله را نه تنها برای کوله‌پشتی‌هایی با اندازه‌ی K، بلکه برای کوله‌پشتی‌هایی که ظرفیت آن‌ها کوچک‌تر یا مساوی K است، حل کنیم.

فرض استقرا (دومین تلاش): چگونگی حل مسأله P(n-1,k) را برای همه‌ی kهایی که در بازه‌ی [0,K] هستند، می‌دانیم.

در نخستین تلاش برای حل مسأله، کاهش اندازه‌ی مسأله به K وابسته نبود. ما این فرض را در حل P(n,k) برای همه‌ی kهایی که در فاصله‌ی [0,K] هستند، به کار می‌بریم. می‌توان حالت پایه (یعنی P(1,k)) را به آسانی حل کرد: اگر k=0، آنگاه همیشه پاسخی (بدیهی) وجود خواهد داشت. در غیر این صورت، تنها در حالتی که اندازه‌ی نخستین عنصر، k باشد، پاسخ وجود خواهد داشت. بدین ترتیب، P(n,k) را به دو مسأله‌ی P(n-1,k) و P(n-1,k-k_n) تبدیل کرده‌ایم. (اگر مسأله‌ی B از روی حل مسأله‌ی A حل شود، گوییم B به A کاهش می‌یابد. دقت کنید که این «کاهش» با کاهش اندازه‌ی مسأله متفاوت است. در فصل‌های ۱۰ و ۱۱ به تفضیل در این مورد بحث خواهد شد - مترجمان) اگر $k-k_n<0$، مسأله‌ی دوم را کنار می‌گذاریم و بررسی نمی‌کنیم، وگرنه هر دو مسأله را می‌توان با کاهش حل کرد. چنین کاهشی درست است و بر این مبنا یک الگوریتم به دست می‌آید، هرچند که این الگوریتم ممکن است ناکارآمد باشد. مسأله‌ای با اندازه‌ی n را به دو زیرمسأله با اندازه‌ی n-1 کاهش دادیم! (حتا مقدار k را نیز در یکی از زیرمسأله‌ها کم کرده‌ایم.) می‌توان هر یک از این دو زیرمسأله را نیز به دو زیرمسأله‌ی دیگر کاهش داد که با چنین روشی زمان اجرای الگوریتم، نمایی خواهد شد.

خوش‌بختانه، در بسیاری موارد می‌توان زمان اجرای الگوریتم حل چنین مسأله‌هایی را بهبود بخشید. نکته‌ی اصلی این است که شاید تعداد زیرمسأله‌های ممکن خیلی زیاد نباشد. در حقیقت، نمادگذاری P(i,k) را به همین دلیل برگزیدیم؛ چراکه موجب درک بهتر این نکته می‌گردد. در این نمادگذاری برای پارامتر نخست، n حالت گوناگون و برای پارامتر بعدی، K حالت مختلف وجود دارد. پس مسأله، تنها nK حالت مختلف دارد! منشأ زمان اجرای نمایی، دو برابر شدن شمار زیرمسأله‌ها هنگام کاهش اندازه‌ی مسأله است. پس تعدادی از این nK زیرمسأله‌ی مختلف را بارها و بارها حل کرده‌ایم. چاره در این است که همه‌ی پاسخ‌ها را نگه داریم تا مجبور نشویم یک زیرمسأله را چند بار حل کنیم. این رویکرد، ترکیبی از تقویت فرض استقرا و بهره‌گیری از استقرای قوی است. (در استقرای قوی، برای اثبات درستی گزاره‌ی P(n) از درستی تمام گزاره‌های P(1)، P(2)، ... و P(n-1) استفاده می‌شود.) حال ببینیم چگونه می‌توان با این رویکرد، الگوریتم را پیاده‌سازی کرد.

پاسخ همه‌ی زیرمسأله‌ها را در ماتریسی به اندازه‌ی n×K ذخیره می‌کنیم. درایه‌ی (i,k) در ماتریس، اطلاعات مربوط به حل مسأله‌ی P(i,k) را در خود دارد. در واقع به کمک کاهش انجام‌شده در دومین تلاش برای فرض استقرا سطر nام این ماتریس را محاسبه کرده‌ایم. هر درایه‌ی سطر nام را می‌توان از روی دو درایه‌ی سطر n-1ام به دست آورد. اگر به یافتن زیرمجموعه‌ی پاسخ نیز علاقه‌مند

باشیم، می‌توانیم به هر درایه‌ی ماتریس، یک پرچم بیفزاییم تا آشکار شود که آیا عنصر متناظر با آن درایه، در آن گام از الگوریتم برگزیده شده است یا نه. بعداً می‌توانیم با ردگیری این پرچم‌ها از درایه‌ی (n,K)، زیرمجموعه‌ی پاسخ را بازیابی کنیم. این الگوریتم در شکل ۵-۱۰ آورده شده است و شکل ۵-۱۱ نیز ماتریس کامل نتایج را برای یک ورودی خاص نشان می‌دهد.

الگوریتم: Knapsack(S,K)

ورودی: S (آرایه‌ای به اندازه‌ی n برای نگه‌داری اندازه‌ی عناصر) و K (اندازه‌ی کوله‌پشتی)

خروجی: P (آرایه‌ی دوبعدی به گونه‌ای که اگر برای i عنصر نخست و یک کوله‌پشتی به ظرفیت k، پاسخی وجود داشته باشد، خواهیم داشت: P[i,k].exist=true و اگر iامین عنصر نیز متعلق به راه‌حل باشد، P[i,k].belong، true خواهد بود.)

{برای دیدن پیش‌نهادهایی درباره‌ی بهبود این برنامه، تمرین ۵-۱۵ را ببینید.}

```
begin
    P[0,0].exist := true;
    for k:=1 to K do
        P[0,k].exist := false;
    {لازم نیست که برای iهای بزرگ‌تر یا مساوی ۱، P[i,0]ها را مقداردهی اولیه کنیم،
    for i := 1 to n do                    زیرا از روی P[0,0] حساب می‌شوند.}
        for k := 0 to K do
            P[i,k].exist := false; {مقدار پیش فرض}
            if P[i-1,k].exist then
              P[i,k].exist := true;
              P[i,k].belong := false
            else if k-S[i] ≥ 0 then
                    if P[i-1,k-S[i]].exist then
                      P[i,k].exist := true;
                      P[i,k].belong := true
end
```

شکل ۵-۱۰ الگوریتم Knapsack

	۰	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱	۱۲	۱۳	۱۴	۱۵	۱۶
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I	-	I	-	-	-	-	-	-
k_4=6	O	-	O	O	-	O	I	O	O	I	O	I	-	I	I	-	I

شکل ۵-۱۱ نمونه‌ای از جدول ساخته‌شده برای یک مسأله‌ی کوله‌پشتی. ورودی، شامل چهار عنصر به اندازه‌های ۲، ۳، ۵ و ۶ است. منظور از نمادهای درون جدول چنین است: «I» یعنی پاسخی پیدا شده است که این عنصر را نیز در بر می‌گیرد؛ «O» یعنی پاسخی بدون این عنصر یافته شده است و «-» نیز یعنی تاکنون پاسخی پیدا نشده است. (اگر در خانه‌ی آخر یک ستون، نماد «-» وجود داشته باشد، یعنی

یک کوله‌پشتیِ با اندازه‌ی نشان داده‌شده در سطر نخستِ همین ستون را نمی‌توان با k_1، k_2، k_3 و k_4 پر کرد.)

روشی که در اینجا به کار گرفتیم، نمونه‌ای از یک شیوه‌ی کلی بود که «برنامه‌نویسی پویا» نام دارد. اساس این شیوه، ساختن جدول‌هایی بزرگ از تمام نتایج مشخص‌شده‌ی قبلی است. جدول‌ها به صورت تکراری ساخته می‌شوند؛ یعنی هر خانه‌ی جدول از روی برخی از خانه‌های بالا یا خانه‌های سمت چپش محاسبه می‌گردد. مشکل اصلی، سازمان‌دهی روشی برای ساخت ماتریس، با کارآمدترین شیوه‌ی ممکن است. نمونه‌ی دیگری از برنامه‌نویسی پویا در بخش۶-۸ ارائه خواهد شد.

پیچیدگی: nK خانه در جدول وجود دارد که هر یک از آن‌ها از روی دو خانه‌ی دیگر جدول در زمانی ثابت محاسبه می‌شود. از این‌رو، زمان کل اجرا از O(nK) است. اگر اندازه‌ی عناصر، بسیار بزرگ نباشد، آنگاه K نیز چندان بزرگ نخواهد بود و در نتیجه nK، از عبارتی نمایی برحسب n بسیار بهتر خواهد شد. (اگر K بسیار بزرگ باشد، یا اگر اندازه‌ی عناصر، اعدادی حقیقی باشند، این روش به کار نمی‌آید؛ درباره‌ی این موضوع در فصل ۱۱ بحث خواهیم کرد.) اگر تنها بخواهیم وجود پاسخ را بررسی کنیم، آنگاه پاسخ در P[n,K] قرار دارد، اما اگر دوست داشته باشیم که زیرمجموعه‌ی عناصر تشکیل‌دهنده‌ی پاسخ را نیز بیابیم، در مسأله‌ی کوله‌پشتی می‌توانیم مثلاً با کمک پرچم belong و با شروع از خانه‌ی (n,K)، این زیرمجموعه را پس از ردگیری، در زمانی از O(n) به دست آوریم. (منظور از پرچم belong، پرچمی است که در الگوریتم شکل ۵-۱۰ به کار رفته است - مترجمان)

توجه: هنگامی که بتوان مسأله را به چندین زیرمسأله‌ی نه چندان کوچک‌تر کاهش داد، روش برنامه‌نویسی پویا کارآمد خواهد بود. همه‌ی این زیرمسأله‌ها با نگه‌داری یک ماتریس بزرگ محاسبه و حل می‌شوند. پس برنامه‌نویسی پویا، در حالتی که تعداد کل زیرمسأله‌های مختلف خیلی بزرگ نباشد، به کار می‌آید. حتا در این صورت هم لازم است ماتریس‌های بزرگی ساخته شوند که این کار طبعاً نیازمند حافظه‌ی بسیاری است. (در برخی موارد، مانند برنامه‌ی شکل ۵-۱۰، با نگه‌داری تنها بخش کوچکی از ماتریس در هر لحظه، می‌توان در حافظه صرفه‌جویی کرد.) معمولاً زمان اجرای الگوریتم‌های برنامه‌نویسی پویا بهتر از $O(n^2)$ نخواهد شد.

۵-۱۱ اشتباهات رایج

در این بخش، برخی اشتباهات رایج هنگام کاربرد استقرا برای طراحی الگوریتم‌ها را به اختصار گوش‌زد می‌کنیم. پیش‌تر، در بخش ۲-۱۳ درباره‌ی اشتباهات رایج در اثبات‌های استقرایی بحث کردیم. ممکن است شبیه تمام آن اشتباهات در اینجا نیز رخ دهد؛ مثلاً فراموش کردن حالت پایه، اشتباهی رایج است. دقت کنید که برای روال‌های بازگشتی وجود یک حالت پایه برای پایان بازگشت‌ها ضروری است. اشتباه

رایج دیگر، گسترش راه‌حل حالت n، به راه‌حل گونه‌ی خاصی از مسأله برای n+1 است؛ در صورتی که این حالت باید دل‌خواه و کلی باشد.

تغییر دادن ناآگاهانه‌ی فرض استقرا نیز اشتباه رایجی است. در اینجا، نمونه‌ای از این اشتباه را می‌آوریم: گراف G=(V,E) دوبخشی نامیده می‌شود، اگر بتوان مجموعه‌ی رأس‌هایش را به دو زیرمجموعه افراز کرد، به گونه‌ای که هیچ یالی، دو رأس از یک زیرمجموعه را به یکدیگر متصل نکند. (به عبارت دیگر، هر یال گراف، رأسی از یک زیرمجموعه را به رأسی از زیرمجموعه‌ی دیگر متصل می‌کند - مترجمان) اگر گراف همبند و دوبخشی باشد، آنگاه این افراز یکتاست (از اثبات چشم‌پوشی می‌کنیم).

مسأله: گراف همبند و بدون‌جهت G=(V,E) داده شده است. مشخص کنید آیا گراف، دوبخشی است یا نه و اگر هست، رأس‌های آن را به دو بخش افراز کنید.

یک راه‌حل نادرست: می‌کوشیم رأسی مانند v را با استقرا حذف و باقی‌مانده‌ی گراف را به دو زیرمجموعه تقسیم کنیم. زیرمجموعه‌ی نخست را «قرمز» و زیرمجموعه‌ی دوم را «آبی» می‌نامیم. اگر v تنها به رأس‌های «قرمز» متصل باشد، آن را به زیرمجموعه‌ی «آبی» و اگر تنها به رأس‌های «آبی» متصل باشد، آن را به زیرمجموعه‌ی «قرمز» می‌افزاییم؛ اما اگر v به رأس‌هایی از هر دو زیرمجموعه متصل باشد، گراف دوبخشی نیست (چراکه این تقسیم‌بندی باید یکتا باشد).

اشتباه اصلی راه‌حل بیان‌شده، این است که پس از حذف یک رأس، ممکن است گراف، دیگر همبند نباشد. از این رو، نمونه‌ی کوچک‌تر مسأله، دقیقاً مانند نمونه‌ی اصلی آن نیست؛ پس به‌کارگیری استقرا درست انجام نشده است، اما اگر رأسی را حذف می‌کردیم که گراف را ناهمبند نمی‌کرد، این راه‌حل می‌توانست درست باشد. این مسأله، راه‌حل بهتری هم دارد که به همبندی گراف وابسته نیست. حل مسأله را به خواننده واگذار می‌کنیم (تمرین ۷-۳۲). برای دیدن مثالی مشابه و بحث بیش‌تر درباره‌ی این اشتباه رایج به بخش ۷-۵ مراجعه کنید. نتیجه‌ای مرتبط با این الگوریتم نادرست در تمرین ۵-۲۴ آمده است.

گاهی وسوسه می‌شویم در استقرا، فرض را تغییر دهیم. اگر فرض استقرا عبارتی به صورت «می‌دانیم چگونه فلان و بهمان را بیابیم» باشد، آنگاه وسوسه می‌شویم که شاید بتوانیم با همین میزان تلاش، چیزهای ساده‌ی دیگری را نیز بیابیم؛ اما نباید بدون تغیر صریح فرض استقرا، چنین کاری کنیم. راهی برای پرهیز از تغییر ناآگاهانه‌ی فرض استقرا چنین است که به آن به صورت یک جعبه‌ی سیاه نگاه کنیم. هرگز جعبه‌ی سیاه را تغییر ندهید، مگر آن که آمادگی باز کردنش را داشته باشید (یعنی بتوانید آن را به طور صریح، دوباره تعریف کنید.)

۵-۱۲ خلاصه

در این فصل، چندین روش برای طراحی الگوریتم معرفی شد که همه‌ی آن‌ها گونه‌های مختلفی از یک رویکرد هستند. این روش‌ها به هیچ وجه همه‌ی شیوه‌های شناخته‌شده‌ی طراحی الگوریتم را در بر نمی‌گیرند. روش‌ها و مثال‌های پرشمار دیگری نیز در فصل‌های بعدی ارائه خواهند شد. بهترین راه برای یادگیری این روش‌ها به‌کارگیری آن‌ها در حل مسأله است. بقیه‌ی کتاب نیز دقیقاً به همین هدف می‌پردازد. اصول معرفی‌شده در این فصل چنین بود:

- می‌توانیم از اصل استقرا در طراحی الگوریتم سود جوییم، به این صورت که مسأله را به دو یا بیش از دو زیرمسأله‌ی کوچک‌تر کاهش دهیم. اگر بتوان کاهش را پی‌درپی انجام داد و امکان حل حالت پایه نیز فراهم باشد، آنگاه می‌توان الگوریتم را با استقرا پیش برد. ایده‌ی اصلی، تمرکز روی کاهش مسأله به جای حل مستقیم آن است.
- یکی از آسان‌ترین راه‌ها برای کاهش اندازه‌ی یک مسأله، حذف برخی از عناصر آن است. برای حل مسأله، نخست این روش را بیازمایید و ببینید آیا مسأله حل می‌شود یا نه. انجام چنین حذفی به شیوه‌های گوناگون ممکن است. جدای از حذف عناصری که به روشنی درمی‌یابیم دخالتی در مسأله ندارند (مانند بخش ۵-۳) ممکن است بتوان دو عنصر را در یکدیگر ادغام کرد، ممکن است بتوان عناصری را یافت که مدیریت آن‌ها با حالات ویژه‌ی آسانی شدنی باشد، یا این که بتوان عنصر تازه‌ای را وارد کرد که نقش دو یا بیش از دو عنصر اولیه را بر عهده گیرد (بخش ۶-۶).
- می‌توان از راه‌های گوناگونی اندازه‌ی مسأله را کاهش داد، اما همه‌ی این راه‌ها کارایی یکسانی ندارند. در نتیجه، باید همه‌ی کاهش‌های ممکن را در نظر گرفت. به ویژه، می‌ارزد که ترتیب‌های گوناگون دنباله‌ی استقرا نیز بررسی شود. نمونه‌هایی را دیدیم که در آن‌ها بهتر بود، نخست بزرگ‌ترین عنصر را مورد توجه قرار دهیم و گاهی هم بهتر است نخست عنصر کوچک‌تر را در نظر بگیریم. نمونه‌هایی را هم خواهیم دید که در آن‌ها بهتر است کار را از وسط آغاز کنیم (بخش ۶-۲). در درخت‌ها، هم مثال‌هایی را خواهیم دید که در آن‌ها نخست ریشه را حذف می‌کنیم (روش بالا به پایین) و هم مثال‌هایی را که در آن‌ها نخست برگ‌ها حذف می‌شوند (روش پایین به بالا) (بخش ۶-۴-۴).
- یکی از کارآمدترین روش‌ها برای کاهش اندازه‌ی مسأله، تقسیم آن به دو (یا بیش از دو) بخش برابر است. این روش، یعنی «تقسیم‌وحل»، اگر بتوان مسأله را به گونه‌ای تقسیم کرد که خروجیِ (یا همان راه‌حلِ - مترجمان) زیرمسأله‌ها به آسانی، خروجی کل مسأله را به وجود

آورند، روشی مؤثر و سودمند است. الگوریتم‌های تقسیم‌وحل در بخش‌های ۶-۴، ۶-۵، ۸-۲، ۸-۴، ۹-۴ و ۹-۵ به کار گرفته خواهند شد.

- از آنجا که یک کاهش، تنها می‌تواند اندازه‌ی مسأله، و نه خود مسأله را تغییر دهد، باید به دنبال زیرمسأله‌هایی بگردیم که تا حد ممکن مستقل باشند. برای مثال، ممکن است مسأله‌ی یافتن ترتیب‌هایی در بین چند عنصر، به مسأله‌ی یافتن (یا حذف کردن) جزیی که نخستین عنصرِ آن ترتیب است، کاهش یابد؛ چراکه ترتیب نسبی باقی‌مانده‌ی عناصر، مستقل از عنصر نخست است (بخش‌های ۶-۴ و ۷-۵ را ببینید).
- لزوم یکسان بودن مسأله‌ی کاهش‌یافته با مسأله‌ی اصلی، خود، محدودیتی است، اما راهی هم برای غلبه بر آن وجود دارد: گزاره‌ی مسأله را تغییر دهید. این شیوه خیلی مهم است و از آن بسیار سود خواهیم جست. گاهی نیز بهتر است در استقرا فرض را تضعیف کنیم تا به الگوریتمی ضعیف‌تر برسیم و بعداً با بهره‌گیری از این فرض تضعیف‌شده، در جایگاه بخشی از الگوریتم اصلی، حل مسأله را کامل کنیم (بخش ۶-۱۰ را ببینید).
- سرانجام می‌توان از همه‌ی این روش‌ها یا ترکیب‌های گوناگونی از آن‌ها سود جست. برای مثال، می‌توانیم روش تقسیم‌وحل را همراه با تقویت فرض استقرا به کار ببریم تا بتوانیم زیرمسأله‌های گوناگون را آسان‌تر با هم ترکیب کنیم (بخش ۸-۴ را ببینید).

مراجعی برای مطالعه‌ی بیش‌تر

شیوه‌ی ارائه‌شده در این فصل را نویسنده‌ی کتاب به وجود آورده است (Manber [۱۹۸۸]). البته روشن است که این شیوه، روش تازه‌ای نیست. سود جستن از استقرا و در حالت کلی، سود جستن از روش‌های اثبات ریاضی در طراحی الگوریتم‌ها ریشه در «روندنماهای» Goldstine و von Neuman دارد (von Neuman [۱۹۶۳]) اما نخستین بار Floyd [۱۹۶۷] آن را تکمیل کرد. Dijkstra [۱۹۶۷]، Manna [۱۹۸۰]، Gries [۱۹۸۱] و Dershowitz [۱۹۸۳] شیوه‌هایی را مانند روش ما، برای ساخت برنامه‌ها همراه با اثبات درستی آن‌ها ارائه کرده‌اند. رویکرد آنان بسیار موشکافانه‌تر از روش این فصل در طراحی برنامه‌هاست و همه‌ی جزئیات را به دقت بررسی کرده‌اند. می‌توان کاربرد «قانون ثابت حلقه» را - که در بخش ۲-۱۲ معرفی شد - از برخی جهات، معادل بهره‌گیری از استقرا در این فصل در نظر گرفت. بی‌شک در طراحی الگوریتم روش‌های بازگشتی نیز بسیار زیاد به کار می‌روند (برای نمونه Burge [۱۹۷۵] و Paull [۱۹۸۸] را ببینید).

مسأله‌ی ستاره‌ی مشهور نخستین بار از سوی Aanderaa مطرح شد (Rosenberg [۱۹۷۳] را ببینید). می‌توان با پرهیز از پرسش‌های تکراری در مرحله‌ی حذف الگوریتم، تعداد $\lfloor \log_2^n \rfloor$ پرسش در مرحله‌ی تأیید صرفه‌جویی کرد (King و Smith-Thomas [۱۹۸۲]). احتمالاً تقویت فرض استقرا

ترفندی کهنه است. Polya [۱۹۷۵] این روش را «تضاد ابداعی» می‌نامد (چراکه ایجاد یا اثبات قضیه‌ای قوی‌تر آسان‌تر هم می‌شود). گاهی نیز این ترفند را «تعمیم» می‌نامند. Bellman [۱۹۷۵] شیوه‌ی برنامه‌نویسی پویا را ارائه کرد و رسمیت بخشید. این روش، کاربردها و گونه‌های فراوانی دارد. برای آشنایی دقیق‌تر با آن، برای نمونه Dreyfus و Law [۱۹۷۷] یا Denardo [۱۹۸۲] را ببینید. دقت نظر Tom Trotterموجب طرح تمرین ۵-۲۴ شد.

تمرین‌های آموزشی

۵-۱ یک الگوریتم تقسیم‌وحل برای ارزیابی چندجمله‌ای‌ها طراحی کنید. الگوریتمتان به چند جمع و چند ضرب نیاز دارد؟ آیا این الگوریتم مزیتی هم بر روش Horner دارد؟

۵-۲ بکوشید با بهره‌گیری از گام‌های استدلال استقرایی (به‌کارگرفته‌شده در بخش ۵-۳) این نوع از مسأله‌ی بزرگ‌ترین زیرگراف القایی را حل کنید: گراف G=(V,E) داده شده است، می‌خواهیم بزرگ‌ترین زیرگراف القایی 'G را به گونه‌ای بیابیم که درجه‌ی همه‌ی رأس‌های 'G حداکثر k باشد (برخلاف مسأله‌ی بخش ۵-۳ که درجه‌ی همه‌ی رأس‌های 'G «دست‌کم» k بود). این حالت مسأله، از حالت اصلی بسیار دشوارتر است، به گونه‌ای که روش حل مسأله‌ی اصلی، دیگر در اینجا کارساز نیست (برای دیدن بحثی درباره‌ی مسأله در حالت ساده‌ی k=0، فصل ۱۱ را ببینید).

۵-۳ الگوریتم Mapping را در نظر بگیرید (شکل ۵-۳). آیا ممکن است در پایان اجرای الگوریتم، مجموعه‌ی S تهی باشد؟ اگر چنین چیزی ممکن است، مثالی برای آن بیاورید، وگرنه ثابت کنید هرگز مجموعه‌ی S در پایان اجرای الگوریتم تهی نخواهد بود.

۵-۴ برای نخستین while در الگوریتم Celebrity (شکل ۵-۴) قانون ثابت حلقه‌ی مناسبی بیابید.

۵-۵ درخت دودویی T به شما داده شده است. اگر عامل توازن در همه‌ی گره‌های درختی ۰، ۱ یا ۱- باشند، آن درخت را AVL می‌نامیم T درخت AVL نامیده می‌شود، اگر عامل توازن همه‌ی گره‌هایش ۰، ۱ یا ۱- باشد (بخش ۴-۳-۴ را ببینید). فرض کنید گره‌ها فضای کافی برای ذخیره‌ی عامل توازن ندارند. الگوریتمی کارآمد برای حل این «مسأله‌ی تصمیم‌گیری» طراحی کنید: درخت T داده شده است و الگوریتم باید مشخص کند که درخت، AVL هست یا نه. الگوریتم از نوع تصمیم‌گیری است؛ یعنی پاسخ باید تنها به صورت «بله» یا «خیر» باشد.

۵-۶ الگوریتم Maximum_Consecutive_Subsequence (شکل ۵-۹) را به گونه‌ای اصلاح کنید که علاوه بر حاصل‌جمع، زیردنباله‌ی پاسخ را نیز بیابد.

۵-۷ با به‌کارگیری پرچم belong در مسأله‌ی کوله‌پشتی، برنامه‌ای برای یافتن اشیایی که باید در کوله‌پشتی قرار گیرند، بنویسید.

۸-۵ در الگوریتم Knapsack، نخست (با بررسی P[i-1,j]) مشخص کردیم که آیا iامین عنصر غیرضروری است یا نه. چنان‌چه با i-1 عنصر، پاسخی وجود داشت، همین پاسخ را برگزیدیم. می‌توانیم روش دیگری پیش گیریم و آن، بهره‌گیری از پاسخ iامین عنصر - در صورت وجود - است (یعنی باید نخست $P[i,j-k_i]$ را بررسی کنیم). به نظر شما کدام روش کاراتر است؟ شکل ۵-۱۱ را برای این روش تازه از نو رسم کنید.

۹-۵ ممکن است یک مسأله‌ی کوله‌پشتی پاسخ‌های گوناگونی داشته باشد. ویژگی‌های خاص پاسخ الگوریتم Knapsack (شکل ۵-۱۰) چیست؟ چه چیزی پاسخ این الگوریتم را از دیگر پاسخ‌ها متمایز می‌کند؟ اگر روش گزینش عنصر، طبق تمرین ۵-۷ باشد، در پاسخ چه تغییری به وجود خواهد آمد؟

تمرین‌های خلاقانه

۱۰-۵ این حالت گسترش‌یافته از مسأله‌ی نمای افقی را حل کنید: ساختمان‌های نمای افقی بام هم دارند. ساختمان‌ها مستطیل‌شکل هستند و بام‌های آن‌ها به صورت مثلثی در بالای آن‌ها قرار گرفته است. (می‌توانید برای سادگی، زاویه‌ی هر ساختمان با بام خود را ۴۵ درجه در نظر بگیرید.) اینجا نیز افق همه‌ی ساختمان‌ها یکسان است. برای رسم نمای افقی در این حالت، یک الگوریتم طراحی کنید.

۱۱-۵ فرض کنید دو نمای افقی متفاوت داده شده باشد. یکی از این دو با نور آبی و دیگری با نور قرمز روی یک پرده تابانده شده‌اند. الگوریتمی کارآمد طراحی کنید که شکل ارغوانی‌رنگ را به دست آورد. به عبارت دیگر، اشتراک دو نمای افقی را حساب کنید.

۱۲-۵ x_1، x_2، ... و x_n را دنباله‌ای از اعداد حقیقی در نظر بگیرید (همه‌ی این اعداد، لزوماً مثبت نیستند). الگوریتمی از O(n) طراحی کنید که زیردنباله‌ی متوالی (یا به‌هم‌پیوسته‌ی) x_i، x_{i+1}، ... و x_j را بیابد، به گونه‌ای که حاصل‌ضرب اعداد آن در بین همه‌ی زیردنباله‌های به‌هم‌پیوسته بیشینه باشد. حاصل‌ضرب زیردنباله‌ی تهی را ۱ تعریف می‌کنیم.

۱۳-۵ فرض کنید درختی به ما داده‌اند که AVL نیست. هرگره این درخت را که عامل توازنی برابر با ۰، ۱ یا ۱- داشته باشد، یک گره AVL می‌نامیم. الگوریتمی طراحی کنید که گره‌هایی از درخت را که خود، AVL نیستند اما همه‌ی گره‌های پایین‌دست آن‌ها AVL هستند، علامت بزند.

۱۴-۵ G=(V,E) را درختی دودویی با n رأس بگیرید. می‌خواهیم ماتریسی n×n بسازیم که مقدار درایه‌ی v_{ij}ام آن برابر با فاصله‌ی بین v_i و v_j باشد. (از آنجا که درخت بدون‌جهت است، این

ماتریس متقارن خواهد بود). الگوریتمی از $O(n^2)$ طراحی کنید که این ماتریس را از روی لیست همسایگی یک درخت بسازد.

۱۵-۵ $G=(V,E)$ را درختی دودویی بگیرید. فاصله‌ی بین دو رأس G، طول مسیری است که این دو رأس را به یکدیگر متصل می‌کند (فاصله‌ی همسایه‌ها ۱ است). قطر G بیش‌ترین فاصله، در بین تمام زوج‌رأس‌هاست. الگوریتمی با زمان خطی طراحی کنید که قطر درخت داده‌شده را بیابد.

۱۶-۵ بهره‌وری از حافظه را در الگوریتم Knapsack (بخش ۵-۱۰) بهبود دهید. آیا تخصیص یک ماتریس کامل $n\times K$ ضروری است؟ پیچیدگی فضایی الگوریتم بهبودیافته چقدر است؟

۱۷-۵ این حالت از مسأله‌ی کوله‌پشتی را حل کنید: مفروضات مانند بخش ۵-۱۰ است، اما تعداد نامحدودی از هر عنصر وجود دارد. به عبارت دیگر، مسأله، بسته‌بندی عناصری با اندازه‌های داده‌شده در یک کوله‌پشتی با ظرفیتی مشخص است، با این تفاوت که می‌توان هر عنصر را چندین بار به کار برد.

۱۸-۵ یک نوع دیگر از مسأله‌ی کوله‌پشتی: مفروضات مسأله همانند تمرین ۵-۱۷ است (n عنصر، تعداد نامحدودی از هر یک از آن‌ها و ظرفیت ثابتی برای کوله‌پشتی) اما هر عنصر، قیمت یا ارزش متناظری نیز دارد. الگوریتمی طراحی کنید که کوله‌پشتی را به گونه‌ای کاملاً پر کند که جمع ارزش‌ها (یا قیمت‌های) عناصر درون کوله‌پشتی بیش‌ترین مقدار ممکن باشد.

۱۹-۵ رایج‌ترین نوع مسأله‌ی کوله‌پشتی چنین است: مفروضات همانند تمرین ۵-۱۷ (n عنصر با اندازه و ارزش مشخص، تعداد نامحدودی از هر عنصر، ظرفیت ثابتی برای کوله‌پشتی و هدف بیشینه کردن جمع ارزش‌های عناصر) اما دیگر مجبور نیستیم کوله‌پشتی را دقیقاً به اندازه‌ی ظرفیتش پر کنیم. در این مورد، علاقه‌مندیم که با در نظر گرفتن محدودیت ظرفیت کوله‌پشتی، جمع کل ارزش‌ها را بیشینه سازیم.

۲۰-۵ x_1، x_2، ... و x_n را مجموعه‌ای از اعداد صحیح و S را حاصل‌جمع آن‌ها در نظر بگیرید($S=\sum_{i=1}^{n} x_i$). الگوریتمی طراحی کنید که یا این مجموعه را به دو زیرمجموعه با جمع مساوی افراز کند، یا مشخص کند که چنین کاری شدنی نیست. زمان اجرای این الگوریتم باید از $O(nS)$ باشد.

۲۱-۵ فرض کنید یک الگوریتم به صورت جعبه‌ای سیاه به شما داده‌اند؛ یعنی چگونگی طراحی این الگوریتم بر شما پوشیده است. الگوریتم دارای این ویژگی است: اگر دنباله‌ای از اعداد حقیقی و عدد صحیح k را به الگوریتم بدهید، پاسخش «بله» یا «خیر» است. این پاسخ نشان می‌دهد که آیا زیرمجموعه‌ای از این اعداد وجود دارد که جمع عناصرش دقیقاً k باشد یا نه. مشخص کنید چگونه با کمک این جعبه‌ی سیاه می‌توان زیرمجموعه‌ای را یافت که جمع عناصرش

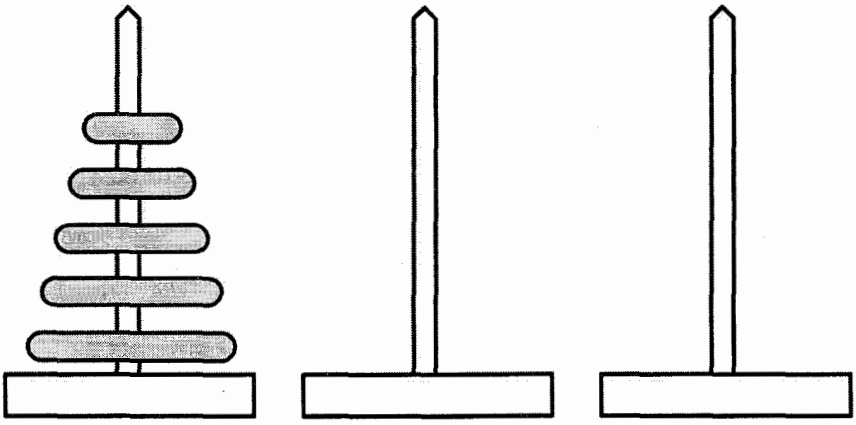
دقیقاً k باشد (البته اگر چنین زیرمجموعه‌ای وجود داشته باشد). باید تعداد دفعات به‌کارگیری این جعبه‌ی سیاه از O(n) باشد (n اندازه‌ی دنباله است).

۲۲-۵ معمای برج‌های هانوی نمونه‌ای شناخته‌شده از یک مسأله‌ی نه چندان ساده است که راه‌حل بازگشتی ساده‌ای دارد. n صفحه با اندازه‌های گوناگون و به ترتیب نزولی (یعنی صفحات بزرگ‌تر، پایین‌تر قرار می‌گیرند - مترجمان) روی یک میله قرار گرفته‌اند. دو میله‌ی خالی دیگر نیز وجود دارد (شکل ۵-۱۲ را ببینید). هدف معما پیدا کردن چگونگی جابه‌جایی همه‌ی صفحات میله‌ی اول به یک میله‌ی دیگر است. صفحاتی که در بالای یک میله قرار دارند، به بالای میله‌ی دیگر جابه‌جا می شوند. تنها هنگامی می‌توان یک صفحه را در بالای میله‌ای دیگر قرار داد که آن صفحه از تمامی صفحات این میله کوچک‌تر باشد. به عبارت دیگر، ترتیب نزولی اندازه‌ی صفحات هر میله همواره باید رعایت شود. هدف، جابه‌جایی همه‌ی صفحات از میله‌ی ۱ به میله‌ی ۳، با کمک میله‌ی ۲ و با کم‌ترین تعداد حرکات ممکن است.

الف- الگوریتمی طراحی کنید (با استقرا) که کوتاه‌ترین دنباله‌ی حرکات را برای n صفحه بیابد.

ب- در الگوریتم‌تان چند حرکت انجام می‌شود؟ یک رابطه‌ی بازگشتی برای تعداد حرکات بسازید و آن را حل کنید.

پ- نشان دهید تعداد حرکات بند «ب» بهینه است؛ یعنی ثابت کنید هیچ الگوریتم دیگری وجود ندارد که تعداد حرکاتش کم‌تر باشد.



شکل ۵-۱۲ معمای برج‌های هانوی

۲۳-۵ برنامه‌ای غیربازگشتی برای حل مسأله‌ی برج‌های هانوی بنویسید (این مسأله در تمرین ۵-۲۲ تعریف شده است).

۲۴-۵ نوع دیگری از مسأله‌ی برج‌های هانوی (تمرین ۵-۲۲) چنین است: دیگر فرض نمی‌کنیم که از آغاز، تمامی صفحات، روی یک میله قرار دارند. ممکن است این صفحات به صورت دل‌خواه

روی میله‌های مختلف باشند، اما بازهم صفحات هر میله ترتیب نزولی دارند. هدف معما، همان هدف مسأله‌ی اصلی با همان محدودیت‌هاست. الگوریتمی طراحی کنید که کوتاه‌ترین دنباله‌ی ممکن از حرکات را برای n صفحه بیابد.

۵-۲۵ این تمرین با الگوریتم نادرست تشخیص دوبخشی بودن گراف در بخش ۵-۱۱ مرتبط است. این تمرین، هم نشان می‌دهد الگوریتم بیان‌شده در بخش ۵-۱۱ نادرست است و هم نشان می‌دهد نمی‌توان رویکردی ساده و آسان برای حل این مسأله به کار گرفت. گونه‌ی کلی‌تر مسأله‌ی رنگ‌آمیزی گراف را در نظر بگیرید: اگر گراف بدون‌جهت G=(V,E) داده شده باشد، یک رنگ‌آمیزی درست و معتبر G، نسبت دادن رنگ‌هایی به گره‌های گراف است که در آن، دو سر هیچ یالی هم‌رنگ نباشد. مسأله، یافتن یک رنگ‌آمیزی درست و معتبر برای گراف داده‌شده با کم‌ترین تعداد رنگ ممکن است. (در حالت کلی، حل این مسأله بسیار بسیار دشوار است و در فصل ۱۱ درباره‌ی آن بحث خواهیم کرد.) یک گراف، دوبخشی است اگر بتوان آن را با دو رنگ رنگ‌آمیزی کرد.

الف- با استقرا ثابت کنید همه‌ی درخت‌ها دوبخشی هستند.

ب- فرض کنید گراف داده‌شده درخت است (که در نتیجه دوبخشی است). می‌خواهیم رأس‌ها را به دو زیرمجموعه افراز کنیم چنان که دو سر هیچ یالی، متعلق به زیرمجموعه‌ی یکسانی نباشند. دوباره به الگوریتم نادرست تشخیص دوبخشی بودن گراف در بخش ۵-۱۱ توجه کنید: یک رأس دل‌خواه برمی‌گزینیم، آن را حذف می‌کنیم. باقی گراف را (به صورت استقرایی) رنگ‌آمیزی می‌کنیم و سپس رأس باقی‌مانده را به بهترین صورت ممکن رنگ می‌زنیم؛ یعنی رأس را با قدیمی‌ترین رنگ ممکن رنگ‌آمیزی می‌کنیم و تنها در حالتی که رأس به رأس‌هایی با تمام رنگ‌های از پیش استفاده‌شده متصل شده باشد، یک رنگ نو برای آن به کار می‌بریم. ثابت کنید اگر رأس‌ها را یکی‌یکی و بدون توجه به ارتباط کلی گره‌ها در گراف رنگ‌آمیزی کنیم، در بدترین حالت حتا ممکن است به $1+\log_2^n$ رنگ نیاز داشته باشیم. ساختاری طراحی کنید که تعداد رنگ‌های لازم را در ترتیب دل‌خواه گره‌ها بیشینه کند (منظور از ترتیب دل‌خواه، ترتیب تصادفی و بدون حساب و کتاب است - مترجمان). ساختار می‌تواند به ترتیب برگزیدن گره‌ها به این صورت، وابسته باشد:

الگوریتم هر بار یک رأس را به عنوان رأس بعدی برمی‌گزیند و سپس یال‌های متصل به آن را بررسی می‌کند. در اینجا شما می‌توانید به دل‌خواه خود به گره - به شرط آن که گراف درخت باقی بماند - یال‌هایی را بیفزایید تا در پایان، به حداکثر تعداد رنگ ممکن نیاز باشد. پس از افزودن یک یال، دیگر نمی‌توان آن را حذف کرد (با این کار، الگوریتمی که از پیش، یال را مشاهده کرده است، دچار اشتباه می‌شود). بهترین راه دست‌یابی به چنین ساختاری یاری گرفتن از استقراست. فرض کنید می‌دانید چگونه باید ساختاری را بسازید که برای

رنگ‌آمیزی، حداکثر به k رنگ نیاز دارد و رأس‌هایش هم تا حد امکان کم است. حال، بدون افزودن تعداد زیادی رأس به آن ساختار، ساختار تازه‌ای بسازید که رنگ‌آمیزی معتبر آن دست‌کم نیازمند k+1 رنگ باشد.

فصل ۶

الگوریتم‌های دنباله‌ها و مجموعه‌ها

«نظم» را من دوست دارم بس زیاد
بال خـود بر روی «بی‌نظـمی» نهاد
«سـادگی» را نغـمه‌خوانـی داد یـاد
(۱۹۳۷-۱۸۷۵) Anna Hempstead Branch

۶-۱ آشنایی

در این فصل با ورودی‌هایی سر و کار داریم که دنباله‌ها یا مجموعه‌هایی متناهی هستند. دنباله و مجموعه با یکدیگر متفاوتند. در دنباله برخلاف مجموعه، ترتیب عناصر اهمیت دارد. به علاوه، می‌دانیم که در یک مجموعه، عنصر تکراری وجود ندارد، در صورتی که یک دنباله ممکن است عنصر تکراری هم داشته باشد. از آنجایی که معمولاً ورودی‌ها ترتیب مشخصی دارند، می‌توان آن‌ها را «دنباله» در نظر گرفت. با این حال، هنگامی که ترتیب ورودی برای ما مهم نباشد، می‌توانیم ورودی را یک مجموعه در نظر بگیریم. در سرتاسر این فصل، ورودی را با آرایه نمایش می‌دهیم، مگر آن که به صراحت خلاف آن را گفته باشیم و فرض می‌کنیم اندازه‌ی آرایه را از پیش می‌دانیم. فرض بر این است که عناصر مجموعه یا دنباله، از یک مجموعه با ترتیب کلی (مانند اعداد صحیح یا اعداد حقیقی) گرفته می‌شوند و می‌توان آن‌ها را با یکدیگر مقایسه کرد. در این فصل به مسأله‌هایی که عناصر آن‌ها هم‌نوع هستند، پرداخته‌ایم. موضوعاتی مانند بیشینگی، ترتیب، زیردنباله‌های ویژه، فشرده‌سازی داده‌ها و تشابه دنباله‌ها را در این فصل بررسی می‌کنیم.

در این فصل، الگوریتم‌های بسیاری را با کاربردهای گوناگون مطرح می‌کنیم. هدف ما این بوده است که از شیوه‌ی طراحی گفته‌شده در فصل ۵، مثال‌های بیش‌تری بیاوریم و همراه با آن برخی الگوریتم‌های مهم را نیز تشریح کنیم. برخی از الگوریتم‌های آورده‌شده بسیار مهم و کاربردی هستند (مانند مرتب‌سازی و جست‌وجوی دودویی)، برخی با آن که اهمیت چندانی ندارند، اما روش‌های جالبی را تشریح می‌کنند (مانند یافتن دو عنصر بزرگ‌تر در یک مجموعه و مسأله‌ی زیردنباله‌ی ناپایدار) و برخی نیز با آن که مهم هستند، اما تنها در زمینه‌های خاصی به کار می‌آیند (مانند فشرده‌سازی پرونده‌ها و مقایسه‌ی دنباله‌ها).

نخستین مثال این فصل، جست‌وجوی دودویی است - الگوریتمی پایه‌ای و اساسی که گونه‌های بسیاری دارد و در موارد گوناگونی ظاهر می‌شود. پس از آن، مرتب‌سازی را مورد بحث و بررسی قرار می‌دهیم که درباره‌ی آن مطالعات گسترده‌ای انجام شده است. سپس شما را با مرتبه‌ی آماری، فشرده‌سازی داده‌ها، الگوریتم‌های مبتنی بر احتمال و دو مسأله از دست‌کاری متن آشنا می‌کنیم. سرانجام، فصل را با چند مثال از الگوریتم‌هایی زیبا و عالی به پایان می‌رسانیم که روش‌های جالبی از طراحی الگوریتم را تشریح می‌کنند.

۶-۲ جست‌وجوی دودویی و گونه‌هایی از آن

نقش جست‌وجوی دودویی در الگوریتم‌ها شبیه نقش «چرخ» در «مکانیک» است، چراکه ساده، زیبا و بسیار بااهمیت است و بارها و بارها با آن روبه‌رو می‌شویم. ایده‌ی اصلی جست‌وجوی دودویی این است که تنها با پرسیدن یک پرسش، فضای جست‌وجو را نصف (یا تقریباً نصف) کنیم. در این بخش، چندین گونه از جست‌وجوی دودویی را شرح داده، نشان می‌دهیم که جست‌وجویی چندمنظوره و پرکاربرد است.

جست‌وجوی دودویی محض

مسأله: x_1، x_2، ... و x_n را دنباله‌ای از اعداد حقیقی با رابطه‌ی $x_1 \le x_2 \le ... \le x_n$ بگیرید. با داشتن یک عدد حقیقی مانند z، می‌خواهیم ببینیم آیا این عدد در دنباله وجود دارد یا نه، و اگر وجود دارد می‌خواهیم اندیس i را چنان بیابیم که $x_i = z$.

برای سادگی، تنها به دنبال یک اندیس i می‌گردیم که در $x_i = z$ صدق کند. در حالت کلی، ممکن است علاقه‌مند باشیم همه‌ی این اندیس‌ها یا کوچک‌ترین آن‌ها، یا بزرگ‌ترین آن‌ها، یا بزرگ‌ترین آن‌ها و یا چیزهایی از این دست را بیابیم. ایده‌ی کار، نصف کردن فضای جست‌وجو با بررسی عنصر میانی است. می‌توانید برای سادگی n را زوج بگیرید. اگر z از $x_{n/2+1}$ کوچک‌تر باشد، روشن است که باید در نیمه‌ی نخست دنباله در جست‌وجوی z باشیم، وگرنه در نیمه‌ی دوم به دنبال z می‌گردیم. یافتن z در هر یک از این دو نیمه، مسأله‌ای با اندازه‌ی n/2 است که می‌توان با استقرا آن را حل کرد. حالت پایه‌ی n=1 را مستقیماً با مقایسه‌ی z با تک عنصر دنباله انجام می‌دهیم. در شکل ۶-۱ الگوریتم جست‌وجوی دودویی محض آورده شده است.

الگوریتم: Binary_Search(X,n,z)

ورودی: X (آرایه‌ی مرتبی با اندیس‌هایی از ۱ تا n) و z (کلید جست‌وجو)

خروجی: Position (یا یک اندیس i به گونه‌ای که $X[i]=z$ و یا صفر، اگر چنین اندیسی وجود ندارد.)

```
begin
      Position := Find(z,1,n);
end

function Find (z,Left,Right): integer;
begin
      if Left = Right then
        if X[Left] = z then Find := Left
        else Find := 0
      else
          Middle := ⌈1/2 (Left + Right)⌉;
          if z < X[Middle] then
            Find := Find(z,Left,Middle-1)
          else
              Find := Find(z,Middle,Right)
end
```

شکل ۶-۱ الگوریتم Binary_Search

پیچیدگی: پس از هر مقایسه، محدوده‌ی جست‌وجو نصف می‌گردد. بنابراین، تعداد مقایسه‌های لازم برای یافتن عددی مشخص در دنباله‌ای به اندازه‌ی n از O(logn) خواهد بود. این نوع از جست‌وجوی دودویی، آزمودن برابری را تا پایان کار به تعویق می‌اندازد. راه دیگر، این است که در هر گام، شرط تساوی با z نیز بررسی شود. با این کار هرچند در هر گام به جای یک مقایسه با عملگر کوچک‌تری دو مقایسه با دو عملگر تساوی و کوچک‌تری انجام می‌شود، اما ممکن است جست‌وجو سریع‌تر به پایان برسد. هرچند نوشتن برنامه به صورت بازگشتی راحت‌تر است، اما به آسانی می‌توانیم آن را به یک برنامه‌ی غیربازگشتی تبدیل کنیم. جست‌وجوی دودویی هنگام بزرگ بودن n، کارآمدتر است، اما هنگام کوچک بودن n، بهتر است خیلی ساده، دنباله را به صورت خطی جست‌وجو کنیم.

جست‌وجوی دودویی در یک دنباله‌ی چرخشی

دنباله‌ی x_1، x_2، ... و x_n هنگامی مرتب‌شده‌ی چرخشی نامیده می‌شود که اگر اندیس کوچک‌ترین جمله‌ی دنباله را i بنامیم، دنباله‌ی x_i، x_{i+1}، ...، x_n، x_1، ... و x_{i-1} صعودی باشد.

مسأله: یک فهرست از اعداد مرتب‌شده‌ی چرخشی به شما داده شده است. موقعیت کوچک‌ترین عدد فهرست را بیابید (برای سادگی فرض می‌کنیم کوچک‌ترین عنصر یکتاست).

برای یافتن کوچک‌ترین عنصر دنباله، از ایده‌ی جست‌وجوی دودویی یاری می‌گیریم تا با یک مقایسه، نیمی از عناصر دنباله کنار گذاشته شوند. دو عدد دل‌خواه x_k و x_m را به گونه‌ای برمی‌گزینیم که $k<m$. اگر $x_k < x_m$، آنگاه i در بازه‌ی [k,m] قرار نمی‌گیرد، چراکه x_i کوچک‌ترین عنصر دنباله است. (دقت کنید که نمی توانیم x_k را کنار بگذاریم.) از سوی دیگر، چنان‌چه $x_k > x_m$، آنگاه i باید در بازه‌ی [k,m] قرار داشته باشد، چون ترتیب عناصر (از نظر صعودی یا نزولی بودن - مترجمان) در جایی از همین بازه تغییر کرده است. پس، با یک مقایسه می‌توانیم عناصر بسیاری را کنار بگذاریم. با گزینش درست k و m، می‌توانیم i را با O(logn) مقایسه بیابیم. این الگوریتم در شکل ۶-۲ ارائه شده است.

الگوریتم: Cyclic_Binary_Search(X,n,z)

ورودی: X (یک آرایه‌ی مرتب‌شده‌ی چرخشی که از عناصر متمایزی با اندیس‌های ۱ تا n ساخته شده است.)

خروجی: Position (اندیس کوچک‌ترین عنصر در X)

```
begin
      Position := Cyclic_Find(1,n);
end

function Cyclic_Find(Left,Right): integer;
begin
    if Left = Right then Cyclic_Find := Left
    else
        Middle := ⌊1/2(Left + Right)⌋;
        if X[Middle] < X[Right] then
          Cyclic_Find := Cyclic_Find(Left,Middle)
        else
            Cyclic_Find := Cyclic_Find(Middle+1,Right)
end
```

شکل ۶-۲ الگوریتم Cyclic_Binary_Search

جست‌وجوی دودویی به دنبال یک اندیس ویژه

در این مسأله‌ی جست‌وجو کلیدی در دست نیست؛ به جای آن، می‌دانیم که باید به دنبال اندیسی بگردیم که شرطی ویژه را برآورده سازد.

مسأله: دنباله‌ای مرتب از اعداد صحیح متمایز به صورت a_1، a_2، ... و a_n داده شده است. مشخص کنید آیا اندیسی مانند i وجود دارد که $a_i = i$.

از جست‌وجوی دودویی محض، نمی‌توان در اینجا سود جست، زیرا مقدار عنصر مورد جست‌وجو در دست نیست؛ اما می‌توانیم ویژگی مورد نظر را با جست‌وجوی دودویی سازگار کنیم. مقدار $a_{n/2}$ را در نظر بگیرید (بازهم n را زوج فرض کنید). اگر این مقدار دقیقاً n/2 باشد، کار، تمام است. اگر از n/2 کم‌تر باشد، با توجه به آن که همه‌ی اعداد متمایزند، آنگاه مقدار $a_{n/2-1}$ از n/2-1 کم‌تر است و تا عنصر نخست به همین ترتیب ادامه دارد. پس، هیچ یک از اعداد نیمه‌ی نخست دنباله ویژگی مورد نظر را برآورده نمی‌سازند و باید در نیمه‌ی دوم دنباله در جست‌وجوی عنصر مورد نظر باشیم. اگر $a_{n/2}$ از n/2 «بزرگ‌تر» باشد، بازهم همین استدلال برقرار خواهد بود. الگوریتم این کار در شکل ۶-۳ آمده است.

الگوریتم: Special_Binary_Search(A,n)

ورودی: A (آرایه‌ای مرتب از اعداد صحیح متمایز با اندیس‌هایی از ۱ تا n)

خروجی: Position (یا اندیس Position که برای آن A[Position]=Position و یا عدد ۰، اگر چنین اندیسی وجود نداشته باشد.)

```
begin
    Position := Special_Find(1,n);
end

function Special_Find(Left,Right): integer;
begin
    if Left = Right then
      if A[Left] = Left then Special_Find := Left
      else Special_Find := 0   {جست‌وجوی ناموفق}
    else
        Middle := ⌈1/2(Left + Right)⌉;
        if A[Middle] < Middle then
          Special_Find := Special_Find(Middle+1,Right)
        else
          Special_Find := Special_Find(Left,Middle)
end
```

شکل ۶-۳ الگوریتم Special_Binary_Search

جست‌وجوی دودویی در دنباله‌هایی با اندازه‌های نامشخص

گاهی روالی را به کار می‌بریم که بسیار شبیه جست‌وجوی دودویی است، اما به جای نصف کردن فضای جست‌وجو، آن را دو برابر می‌کند. مسأله‌ی جست‌وجوی معمولی را در نظر بگیرد، اما فرض کنید که

اندازه‌ی دنباله نامشخص است. نمی‌توانیم فضای این جست‌وجو را نصف کنیم، چون محدوده‌اش را نمی‌دانیم. به جای این کار، به دنبال عنصری مانند x_i می‌گردیم که بزرگ‌تر یا مساوی z باشد. اگر چنین عنصری را یافتیم، آنگاه می‌توانیم جست‌وجوی دودویی را در محدوده‌ی ۱ تا i انجام دهیم. نخست z را با x_1 مقایسه می‌کنیم. اگر $z \le x_1$، آنگاه تنها ممکن است x_1 با z برابر باشد. بنا به استقرا، فرض کنید یک j می‌شناسیم که $j \ge 1$ و $z > x_j$. اگر z را با x_{2j} مقایسه کنیم، فضای جست‌وجو را با انجام یک مقایسه دو برابر کرده‌ایم. اگر $z \le x_{2j}$، روشن می‌شود که $x_j < z \le x_{2j}$. در این صورت، می‌توانیم با چند مقایسه‌ی دیگر از O(logj)، z را بیابیم. روی‌هم‌رفته، اگر i کوچک‌ترین اندیسی باشد که در آن $z \le x_i$، آنگاه تعداد مقایسه‌ها برای یافتن x_j چنان که $z \le x_j$، از O(logi) خواهد بود و برای یافتن i، به O(logi) مقایسه‌ی دیگر نیاز داریم.

به کار بستن این الگوریتم، هنگامی که اندازه‌ی دنباله را می‌دانیم اما حدس می‌زنیم مقدار i بسیار کوچک است، بازهم مناسب خواهد بود. در چنین مواردی الگوریتم گفته‌شده، جست‌وجوی دودویی معمولی را بهبود می‌بخشد، چراکه زمان اجرا، به جای O(logn)، از O(logi) خواهد شد. از سوی دیگر، در اینجا دو روال از نوع جست‌وجوی دودویی وجود دارد، پس در زمان اجرای این الگوریتم، ضریبی اضافی برابر با ۲ پیدا خواهد شد. بنابراین، این الگوریتم تنها در صورتی بهتر از جست‌وجوی دودویی عادی است که $i = O(\sqrt{n})$ (یعنی i از $O(\sqrt{n})$ باشد – مترجمان).

مسأله‌ی زیردنباله‌ی ناپایدار

گاهی در برخی مسأله‌هایی که به نظر نمی‌رسد نیاز به جست‌وجو داشته باشند، بازهم ایده‌ی اصلی جست‌وجوی دودویی پدیدار می‌شود. A و B را دو رشته (دنباله‌ای از کاراکترها) با الفبایی متناهی در نظر بگیرید که $A = a_1 a_2 ... a_n$ و $B = b_1 b_2 ... b_m$ و $m \le n$. B را زیردنباله‌ای از A گوییم، اگر اندیس‌های $i_1 < i_2 < ... < i_m$ وجود داشته باشند که به ازای همه‌ی ژهای موجود در فاصله‌ی [1,m]: $b_j = a_{i_j}$. به عبارت دیگر، B زیردنباله‌ای از A است، اگر بتوانیم با برداشتن برخی عناصر A، به B برسیم. می‌توان به سادگی تعیین کرد که آیا B، زیردنباله‌ای از A هست یا نه؛ به این ترتیب که A را می‌پیماییم تا آن که به نخستین رخداد b_1 (به شرط وجود) برسیم، از آنجا کار را ادامه می‌دهیم تا به b_2 برسیم و ... . با استقرا اثبات درستی این الگوریتم آسان است و آن را به عنوان تمرین به خواننده واگذار می‌کنیم. از آنجا که الگوریتم روی A و B پویش خطی انجام می‌دهد، پس روشن است که زمان اجرا از O(m+n) خواهد بود. در اینجا، برای دنباله‌ی B، B^i را دنباله‌ای تعریف می‌کنیم که هر کاراکترش i بار پشت‌سرهم تکرار شود؛ مثلاً اگر $B = xyzzx$، آنگاه B^3 برابر xxxyyyzzzzzzxxx خواهد بود.

مسأله: دو دنباله‌ی A و B داده شده‌اند. بیش‌ترین مقدار i را چنان بیابید که B^i زیردنباله‌ای از A شود.

این مسأله را مسأله‌ی زیردنباله‌ی ناپایدار می‌نامند. هرچند در نگاه نخست، مسأله‌ی دشواری به نظر می‌رسد، اما به کمک جست‌وجوی دودویی می‌توان آن را به آسانی حل کرد.

برای هر مقدار i می‌توان دنباله‌ی B^i را به آسانی ساخت. از این رو، برای هر مقدار مشخص i می‌توانیم روشن سازیم که آیا B^i زیردنباله‌ای از A هست یا نه. به علاوه، اگر B^j زیردنباله‌ای از A باشد، برای iهای بازه‌ی [1,j]، B^i نیز زیردنباله‌ای از A خواهد بود. مقدار بیشینه‌ی i نمی‌تواند از n/m فراتر رود، زیرا در آن صورت، دنباله‌ی B^i از دنباله‌ی A بلندتر خواهد شد. پس، می‌توان برای حل این مسأله، از جست‌وجوی دودویی کمک گرفت. نخست، **i** را برابر با $\lceil n/m \rceil / 2$ قرار می‌دهیم و بررسی می‌کنیم که آیا B^i زیردنباله‌ای از A هست یا نه. سپس اگر پاسخ مثبت بود، با کنار گذاشتن حد پایین و اگر منفی بود، با کنار گذاشتن حد بالا، به جست‌وجوی دودویی ادامه می‌دهیم. تعداد بررسی‌ها برای تعیین مقدار بیشینه‌ی i، $\lceil \log_2(n/m) \rceil$ خواهد بود؛ بنابراین، زمان اجرای کل از $O((n/m)\log(n/m))$ و در نتیجه از $O(n\log(n/m))$ خواهد شد. دردسرها و مشکلات مقایسه‌ی دنباله‌ها در بخش ۶-۸ مورد بحث و بررسی قرار خواهد گرفت.

این راه‌حل به ما یک ایده‌ی کلی می‌دهد: اگر در جست‌وجوی مقدار بیشینه‌ای برای i باشیم که بتواند یک ویژگی را برآورده سازد، احتمالاً یافتن الگوریتمی که بتواند به ما بگوید آیا مقدار داده‌شده‌ی i، ویژگی مورد نظر را دارد یا نه، کافی خواهد بود. اگر حد بالایی برای i داشته باشیم و ویژگی مورد نظر نیز به گونه‌ای باشد که هرگاه i، آن را برآورده سازد، همه‌ی زهای فاصله‌ی [1,i] نیز آن ویژگی را برآورده کنند؛ آنگاه می‌توانیم بقیه‌ی کار را با جست‌وجوی دودویی به پایان برسانیم. چنان‌چه حد بالای i را نشناسیم، می‌توانیم روش دو برابر کردن i را به کار ببریم؛ یعنی از i=1 آغاز و هر بار مقدار i را دو برابر کنیم تا به محدوده‌ی مناسب برسیم. این جست‌وجو، زمان بیش‌تری می‌گیرد، اما بازهم کارآمد است؛ مگر آن که i مورد نظر بسیار بزرگ باشد. الگوریتمِ حاصل از چنین روشی ممکن است بهینه نباشد؛ اما در موارد بسیاری، همچون مسأله‌ی زیردنباله‌ی ناپایدار، می‌توان ضریب اضافی $O(\log n)$ را نادیده گرفت.

حل معادله‌ها

هرچند حل معادله‌ها با موضوع این فصل تناسب چندانی ندارد، اما شایسته است که نگاهی کوتاه به این بحث بیندازیم. فرض کنید می‌خواهیم پاسخی برای معادله‌ی $f(x)=0$ بیابیم که در آن، تابع

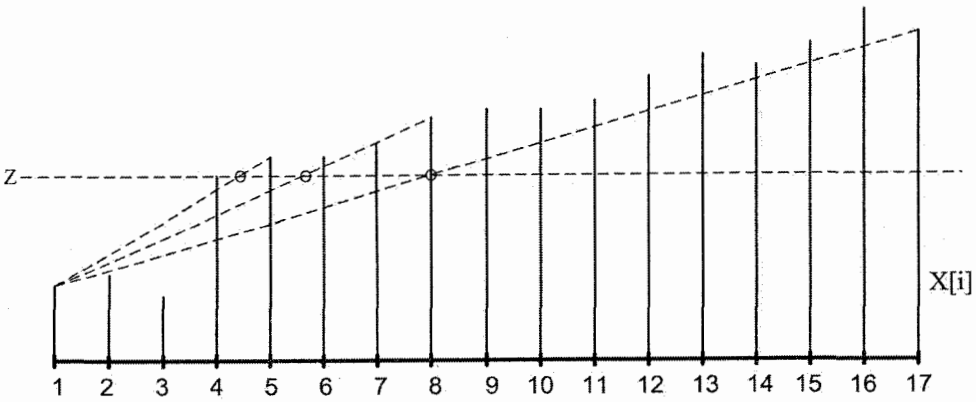
محاسبه‌پذیر f در بازه‌ی [a,b] پیوسته است. اگر بدانیم $f(a).f(b) < 0$ (یعنی f(a) و f(b) هم‌علامت نیستند) می‌خواهیم پاسخ معادله را با دقتی مشخص بیابیم.

از آنجا که تابع پیوسته است، باید در بازه‌ی [a,b] ریشه‌ای داشته باشد. می‌توانیم نوع خاصی از جست‌وجوی دودویی را که به آن روش تنصیف یا Bolzano گویند، به کار ببریم. در این روش، نخست تابع f در $x_1 = (a+b)/2$ ارزیابی می‌گردد. اگر $f(x_1) = 0$ (با دقت مورد نظر)، پاسخ را یافته‌ایم؛ وگرنه، درمی‌یابیم پاسخ در کدام یک از دو زیربازه‌ی [a,x_1] و [x_1,b] قرار دارد (که اندازه‌ی هر یک، نصف اندازه‌ی بازه‌ی اصلی است). گزینش بازه‌ی مورد نظر چنان انجام می‌شود که تابع f در یک سر بازه، مثبت و در سر دیگر، منفی باشد. به همین ترتیب، کار را ادامه می‌دهیم تا به دقت مورد نظر دست پیدا کنیم. پس از k گام، اندازه‌ی ناحیه‌ای که پاسخ در آن است، $(b-a)/2^k$ خواهد بود.

۶-۳ جست‌وجو با درون‌یابی

از آنجا که در جست‌وجوی دودویی، فضای جست‌وجو نصف می‌شود، زمان اجرای آن لگاریتمی خواهد بود. اگر هنگام جست‌وجو با مقداری روبه‌رو شویم که به عدد مورد جست‌وجوی z نزدیک باشد، منطقی‌تر است که به جای جست‌وجوی کورکورانه در نیمه‌ی دیگر، جست‌وجو را در «همسایگی» آن مقدار ادامه دهیم. به ویژه، اگر z بسیار کوچک باشد، خوب است به جای نقطه‌ی میانی، جست‌وجو را از جایی نزدیک به آغاز دنباله شروع کنیم.

فرض کنید می‌خواهیم به دنبال صفحه‌ای مشخص از یک کتاب بگردیم؛ مثلاً در کتابی ۸۰۰ صفحه‌ای به دنبال صفحه‌ی ۲۰۰ باشیم. اگر از حدود یک چهارم از کتاب بگذریم، به این صفحه می‌رسیم؛ با دانستن این موضوع، می‌کوشیم کتاب را از جای مناسبی باز کنیم. احتمالاً با نخستین تلاش به این صفحه نخواهیم رسید؛ مثلاً فرض کنید صفحه‌ی ۲۵۰ را باز کرده باشیم. در این صورت، محدوده‌ی جست‌وجو، ۲۵۰ صفحه خواهد شد و اگر تقریباً یک پنجم از پایان این محدوده را کنار بگذاریم، به صفحه‌ی مورد نظر می‌رسیم. پس به همین اندازه به عقب باز می‌گردیم. می‌توانیم این فرایند را ادامه دهیم تا آن که به اندازه‌ی کافی به صفحه‌ی ۲۰۰ نزدیک شویم، سپس صفحات را یکی‌یکی ورق می‌زنیم تا به صفحه‌ی ۲۰۰ برسیم. ایده‌ی جست‌وجو با درون‌یابی همین است؛ یعنی به جای تقسیم فضای جست‌وجو به دو نیمه‌ی ثابت، با کمک درون‌یابی، این فضا را به گونه‌ای تقسیم می‌کنیم که احتمال موفقیت بیش‌تر شود. این روش در شکل ۶-۴ نشان داده شده است. نخستین تلاش، یعنی $x[8]$، از z بیش‌تر است. با یک درون‌یابی دیگر به $x[5]$ و سرانجام با یکی دیگر به $x[4]$ می‌رسیم. (در اینجا دنباله‌ی $x[8]$، $x[5]$ و $x[4]$ را برای رسیدن به پاسخ پیموده‌ایم – مترجمان) الگوریتم شکل ۶-۵ این روش را بهتر نشان می‌دهد.



شکل ۶-۴ جست‌وجو با درون‌یابی

الگوریتم: Interpolation_Search(X,n,z)

ورودی: X (آرایه‌ای مرتب‌شده با اندیس‌های ۱ تا n) و z (کلید جست‌وجو)

خروجی: Position (یا اندیس i به گونه‌ای که X[i]=z، یا عدد ۰، اگر چنین اندیسی وجود نداشته باشد.)

```
begin
    if z < X[1] or z > X[n] then Position := 0
      {جست‌وجوی نا موفق}
    else Position := Int_Find(z,1,n)
end

function Int_Find(z,Left,Right): integer;
begin
    if X[Left] = z then Int_Find := Left
    else  if Left = Right or X[Left] = X[Right] then
            Int_Find := 0
          else
              Next_Guess := ⌈ Left + (z - X[Left]) (Right - Left) / (X[Right] - X[Left]) ⌉;
              if z < X[Next_Guess] then
                Int_Find := Int_Find(z,Left ,Next_Guess-1)
              else
                  Int_Find := Int_Find(z,Next_Guess,Right)
end
```

شکل ۶-۵ الگوریتم Interpolation_Search

پیچیدگی: کارایی جست‌وجو با درون‌یابی، هم به اندازه‌ی دنباله و هم به خود ورودی بستگی دارد. ممکن است حالت‌هایی رخ دهد که دنباله‌ی رسیدن به پاسخ، تمام اعداد ورودی را در بر گیرد (تمرین

۶-۴ را ببینید). «جست‌وجو با درون‌یابی» برای ورودی‌هایی که عناصرشان (مانند شماره‌ی صفحات یک کتاب) توزیع نسبتاً یک‌نواختی داشته باشند، بسیار کارآمد است. می‌توان نشان داد که میانگین مقایسه‌های جست‌وجو با درون‌یابی از O(loglogn) است (این مقدار، میانگینی از تمام دنباله‌های ممکن است). هرچند ظاهراً کارایی جست‌وجوی دودویی بسیار بهبود یافته است (با توجه به یک بار لگاریتم‌گیری بیش‌تر) اما به دو دلیل عمده، در عمل، جست‌وجو با درون‌یابی از جست‌وجوی دودویی چندان هم بهتر نیست: نخست آن که، مقدار $\log_2^n$ چنان کوچک است که لگاریتم‌گیری دوباره از آن، مقدارش را چندان کاهش نمی‌دهد، مگر آن که n بسیار بزرگ باشد. دیگر این که، جست‌وجو با درون‌یابی نسبت به جست‌وجوی دودویی به محاسبات پیچیده‌تری نیاز دارد.

۶-۴ مرتب‌سازی

مرتب‌سازی یکی از زمینه‌های علوم رایانه است که درباره‌ی آن، بیش‌ترین بررسی‌ها را انجام داده‌اند. مرتب‌سازی، پایه‌ی الگوریتم‌های فراوانی است و در برنامه‌های کاربردی بسیاری، بخش چشم‌گیری از زمان پردازش را به خود اختصاص می‌دهد. این مسأله، گونه‌های فراوانی دارد و برای آن ده‌ها الگوریتم نوشته‌اند. در اینجا، ما حتا نمی‌توانیم قسمت کوچکی از موضوع را پوشش دهیم، بلکه تنها به چندین روش رایج در مرتب‌سازی اشاره می‌کنیم. طبق معمول، بر اصول این الگوریتم‌ها تمرکز می‌کنیم؛ چراکه ممکن است همین اصول در حل مسأله‌های دیگر نیز سودمند باشند. در این بخش، بیش از دیگر قسمت‌ها وارد جزئیات خواهیم شد.

مسأله: اگر n عدد x_1، x_2، ... و x_n به شما داده شده باشند، آن‌ها را به ترتیب صعودی مرتب کنید. به عبارت دیگر، دنباله‌ای از اندیس‌های متمایز $1 \le i_1, i_2, ..., i_n \le n$ چنان بیابید که $x_{i_1} \le x_{i_2} \le ... \le x_{i_n}$.

فرض می‌کنیم این اعداد متمایز باشند، مگر آن که به صراحت خلاف آن را بگوییم. اگر برخی اعداد، یکسان باشند، بازهم روش‌های این بخش به درستی کار می‌کنند. اگر یک الگوریتم مرتب‌سازی به جز آرایه‌ی اولیه‌ی عناصر، به فضای کاری دیگری نیاز نداشته باشد، «درجا» خوانده می‌شود.

۶-۴-۱ مرتب‌سازی سطلی و مرتب‌سازی بر اساس مرتبه

شاید ساده‌ترین راه مرتب‌سازی به کار بردن روش پست‌خانه‌هاست: به اندازه‌ی کافی «جعبه» داریم - که به آن‌ها «سطل» می‌گوییم - و هر عنصر را در سطلی مناسب قرار می‌دهیم. چنین شیوه‌ای، مرتب‌سازی سطلی نامیده می‌شود. اگر عناصر مورد نظر، نامه‌هایی باشند که باید، مثلاً برحسب ایالت،

مرتب شوند، آنگاه تخصیص یک سطل برای هر ایالت کافی است و الگوریتم آن، بسیار کارآمد خواهد بود. از سوی دیگر، چنان‌چه لازم باشد نامه‌ها را از روی کد پستی (که در کشور آمریکا ۵ رقم دارد) مرتب کنیم، با این روش به ۱۰۰,۰۰۰ جعبه و یک پست‌خانه‌ی بسیار بزرگ نیازمندیم. پس، مرتب‌سازی سطلی، تنها برای عناصری با محدوده‌ی کوچک و ساده - که از پیش شناخته‌شده هم هست - خوب عمل می‌کند. حال، این روش را دقیق‌تر توصیف می‌کنیم:

فرض می‌کنیم n عنصر داریم که همگی اعدادی صحیح در بازه‌ی ۱ تا m هستند. m سطل تهیه می‌کنیم و سپس برای هر i (منظور نویسنده، اندیس عناصر موجود است - مترجمان) x_i را با توجه به مقدارش درون سطل مناسب قرار می‌دهیم. در پایان، همه‌ی سطل‌ها را به ترتیب نگاه می‌کنیم و عناصر درون هر سطل را مرتب کرده، کل عناصر را کنار هم گرد می‌آوریم. روشن است که پیچیدگی این الگوریتم ساده از $O(m+n)$ است. اگر m از $O(n)$ باشد، آنگاه برای مرتب‌سازی، الگوریتمی با زمان خطی داریم. از سوی دیگر، اگر m (مانند مثال کد پستی) نسبت به n خیلی بزرگ باشد، $O(m)$ نیز بسیار بزرگ خواهد شد. همچنین، این الگوریتم به حافظه‌ای از $O(m)$ نیاز دارد که اگر m بسیار بزرگ باشد، این مشکل از مشکل پیش هم جدی‌تر است.

یک گسترش طبیعی این ایده، مرتب‌سازی بر اساس مرتبه است. دوباره مثال کد پستی را در نظر بگیرید. بهره‌گیری از مرتب‌سازی سطلی برای کدهای پستی کارآمد نیست، چون محدوده‌ی کدهای پستی بزرگ‌تر از آن است که بتوان آن‌ها را مدیریت کرد. آیا می‌توانیم کاری انجام دهیم تا این محدوده کوچک‌تر شود؟ از استقرا روی محدوده، به این ترتیب سود می‌جوییم: کار را در چند مرحله انجام می‌دهیم. نخست، با کمک ۱۰ سطل، همه‌ی نامه‌ها را برحسب رقم نخست کد پستی مرتب می‌کنیم. بدین ترتیب، هر سطل ۱۰,۰۰۰ کد پستی مختلف را (با توجه به چهار رقم دیگر) می‌پوشاند. زمان اجرای این مرحله از $O(n)$ است. در پایان مرحله‌ی نخست، ۱۰ سطل داریم که عناصر هر کدام از آن‌ها محدوده‌ی کوچک‌تری دارد. اینک می‌توانیم به یاری استقرا مسأله را برای هر یک از سطل‌ها حل کنیم. از آنجا که در هر مرحله، محدوده‌ی کدهای پستی را به یک‌دهمِ مرحله پیش می‌رسانیم و نیز از آنجا که کدهای پستی تنها ۵ رقم دارند؛ تنها به ۵ مرحله نیازمندیم. پس از آن که عناصر داخل هر سطل را مرتب کردیم، می‌توانیم به آسانی همه‌ی عناصر را در فهرستی مرتب‌شده قرار دهیم. جزئیات این الگوریتم را به خواننده واگذار می‌کنیم (تمرین ۶-۵) چون می‌خواهیم به نوع دیگری از مرتب‌سازی با همین ایده بپردازیم. به این نکته دقت می‌کنیم که این محدوده به روش‌های مناسب دیگری نیز تقسیم‌پذیر است. در مثال کد پستی، محدوده، بنا به نمایش ده‌دهی کد پستی تقسیم گردید. اگر کلیدها (منظور نویسنده بخشی از عناصر است که مرتب‌سازی برحسب آن‌ها انجام می‌شود - مترجمان) رشته‌هایی کاراکتری باشند که باید به ترتیب الفبایی قرار گیرند، می‌توانیم با در نظر گرفتن یک حرف در هر مرحله، مرتب‌سازی الفبایی یا واژه‌نامه‌ای را روی آن‌ها انجام دهیم. این الگوریتم مانند الگوریتم

مرتب‌سازی بر اساس مرتبه است. به نسخه‌ای از مرتب‌سازی بر اساس مرتبه که در اینجا آورده شده است (یعنی بررسی رقم‌ها یا کاراکترها از چپ به راست) «مرتب‌سازی تعویض مرتبه» نیز می‌گویند.

هر روش سرراست برای پیاده‌سازی بازگشتی مرتب‌سازی تعویض مرتبه به سطل‌های موقتی هم نیاز دارد (در مثال کد پستی برای کشور آمریکا با ۵۰ ایالت، تعداد این سطل‌ها ۵۰تاست؛ تمرین ۶-۵ را نیز ببینید). روش دیگری برای انجام مرتب‌سازی بر اساس مرتبه، به‌کارگیری استقرا به صورت وارونه است؛ یعنی مرتب‌سازی بر اساس کم‌اهمیت‌ترین بخش کلید آغاز شده، آن قدر تکرار می‌شود تا سرانجام، مرتب‌سازی بر اساس پراهمیت‌ترین بخش صورت گیرد. فرض می‌کنیم عناصر مورد نظر اعداد صحیح بزرگی با k رقم هستند و هر رقم در بازه‌ی [0,d-1] قرار دارد. فرض استقرا، روشن و سرراست است:

فرض استقرا: روش مرتب کردن عناصری را که کم‌تر از k رقم دارند، می‌دانیم.

تفاوت این روش با روش پیش، یعنی مرتب‌سازی تعویض مرتبه، در شیوه‌ی گسترش فرض است. (ایده، به‌کارگیری استقرا با ترتیب وارونه مانند روش Horner در بخش ۵-۲ است.) با داشتن عناصری kرقمی، نخست، مهم‌ترین رقم آن‌ها را نادیده می‌گیریم و به کمک استقرا آن‌ها را برحسب بقیه‌ی رقم‌هایشان مرتب می‌کنیم. حال، فهرستی از عناصر داریم که برحسب k-1 رقم کم‌اهمیت‌ترشان مرتب شده‌اند. دوباره، با یاری d سطل همه‌ی عناصر را برحسب پراهمیت‌ترین رقمشان بررسی می‌کنیم. سپس عناصر همه‌ی سطل‌ها را به ترتیب کنار هم قرار می‌دهیم. به این الگوریتم، مرتب‌سازی گام به گام بر اساس مرتبه گفته می‌شود. حال، نشان می‌دهیم که همه‌ی عناصر برحسب k رقمشان مرتب هستند.

ادعا می‌کنیم هر دو عنصری که در گام آخر در دو سطل گوناگون قرار گرفته باشند، مرتب‌شده هستند. در این مورد، به فرض استقرا هم نیازی نداریم، زیرا بنا به ترتیب واژه‌نامه‌ای، مهم‌ترین رقم، رقمی است که بدون توجه به دیگر رقم‌ها ترتیب عناصر را مشخص کند. از سوی دیگر، اگر مهم‌ترین رقم دو عنصر یکسان باشد، بنا به فرض استقرا آن دو عنصر پیش از گام آخر مرتب شده‌اند. پس، کافی است مطمئن شویم در ترتیبی درست باقی خواهند ماند. نکته‌ی هوشمندانه و ظریف الگوریتم همین است. این مورد، نمونه‌ی خوبی از کاربرد رویکرد استقرایی برای اطمینان از درستی الگوریتم است. لازم است ترتیب عناصری که در یک سطل قرار داده می‌شوند، به هم نخورد. (مثلاً اگر دو عدد ۱۲ و ۱۰ قرار است بر اساس دهگانشان در یک سطل قرار گیرند و ۱۲ پیش از ۱۰ آمده باشد، پس از قرار دادن آن دو در سطل، ۱۲ نباید پس از ۱۰ قرار گیرد؛ یعنی ترتیبشان نباید به هم بخورد - مترجمان) می‌توان هر سطل را با یک صف پیاده‌سازی کرد و در پایان هر مرحله، با الحاق همه‌ی صف‌ها (که تعدادشان dتاست) یک صف از تمام عناصر تشکیل داد که برحسب iتا از کم‌اهمیت‌ترین رقم‌هایشان مرتب‌شده هستند. الگوریتم دقیق این کار در شکل ۶-۶ ارائه شده است.

الگوریتم: Straight_Radix(X,n,k)

ورودی: X (آرایه‌ای با اندیس‌های ۱ تا n از اعداد صحیح که هر کدام k رقم دارند.)

خروجی: X (آرایه‌ی مرتب‌شده)

```
begin
    فرض می‌کنیم در آغاز همه‌ی عناصر در صف GQ قرار دارند;
    {چون عبارت پیش، «فرض» است و نه «دستور»، به نظر مترجمان، نویسنده نباید در
    پایان آن «;» قرار می‌داد.}
{برای سادگی، GQ را به کار برده‌ایم، اما امکان پیاده‌سازی الگوریتم با همان X نیز وجود داشت.}
    for i := 1 to d do
    {d تعداد ارقام ممکن است؛ اگر اعداد ده‌دهی باشند، d برابر با ۱۰ خواهد بود.}
      Q[i]; {مقداردهی اولیه} را تهی کن
    for i := k downto 1 do
      while GQ ناتهی است do
            x; را از GQ، pop کن
            {منظور از pop کردن، برداشتن عنصر از ابتدای صف است – مترجمان}
            d := x امین رقم i;
            x; را به Q[d] اضافه کن
        for t := 1 to d do
            Q[t]; را به GQ بیفزا
    for i := 1 to n do
        X[i] را از GQ، pop کن
end
```

شکل ۶-۶ الگوریتم Straight_Radix

پیچیدگی: برای قرار دادن همه‌ی عناصر در صف GQ به n گام و برای مقداردهی صف Q[i] به d گام نیاز است. حلقه‌ی اصلی الگوریتم که k بار اجرا می‌شود، عناصر را از GQ بیرون می‌کشد و در یکی از Q[i]ها قرار می‌دهد (یا push می‌کند). سپس همه‌ی Q[i]ها به هم پیوند زده می‌شوند. زمان اجرای کل الگوریتم از $O(nk)$ است.

در ادامه‌ی این بخش، بدون توجه به ساختار عناصر، فرض می‌کنیم مقایسه بر اساس کلید عناصر انجام می‌شود. بدین ترتیب، الگوریتم‌ها کلی‌تر می‌شوند، چراکه تنها پیش‌فرض، امکان مقایسه‌ی دو عنصر خواهد بود.

۶-۴-۲ مرتب‌سازی درجی و مرتب‌سازی با انتخاب

هم برای مرتب‌سازی درجی و هم برای مرتب‌سازی با انتخاب، استقرای معمولی را به کار می‌گیریم. فرض کنید n عدد به ما داده شده است و می‌دانیم چگونه می‌توان n-1 عدد را مرتب کرد. پس از مرتب کردن n-1 عدد می‌توانیم با پویش n-1 عدد مرتب‌شده و یافتن جای درست عدد nام، کل اعداد را مرتب کنیم. این روال را «مرتب‌سازی درجی» نامیده‌اند که نامی مناسب است. این روش برای مقادیر کوچک n ساده و کارآمد است؛ اما برای مقادیر بزرگ n، الگوریتم کارآمدی نیست. در بدترین حالت، عدد nام با همه‌ی n-1 عدد پیش از خود مقایسه خواهد شد. تعداد کل مقایسه‌ها برای مرتب‌سازی n عدد ممکن است به n-1+...+2+1، یعنی $\frac{(n-1)n}{2}$ نیز برسد؛ پس، از $O(n^2)$ خواهد بود. به علاوه، برای درج عدد nام در جای درست خودش، احتمالاً به جابه‌جایی عناصر دیگر هم نیاز خواهیم داشت. در گام nام، در بدترین حالت، همه‌ی n-1 عنصر دیگر باید جابه‌جا شوند. از این رو، شمار جابه‌جایی عناصر نیز از $O(n^2)$ خواهد بود. با مرتب‌سازی این عناصر در آرایه و بهره‌گیری از جست‌وجوی دودویی برای یافتن جای درست عدد nام در بین n-1 عنصر مرتب‌شده، می‌توان مرتب‌سازی درجی را بهبود بخشید. با این روش، تعداد مقایسه‌های هر جست‌وجو برای درج هر عنصر، از O(logn) می‌شود و در نتیجه، تعداد کل مقایسه‌ها از O(nlogn) خواهد بود، اما تعداد جابه‌جایی‌ها تغییر نکرده است؛ پس زمان اجرای الگوریتم بازهم از مرتبه‌ی دوم است.

می‌توانیم با گزینش عددی ویژه به عنوان عدد nام، این استقرای سرراست را بهبود دهیم. برای مثال، می‌توانیم بیش‌ترین عدد را به عنوان عدد nام برگزینیم. از آنجا که می‌دانیم بزرگ‌ترین عنصر در انتهای آرایه قرار می‌گیرد، چنین گزینشی مناسب است. الگوریتم این‌گونه انجام می‌شود: نخست، انتخاب عنصر بیشینه، سپس قرار دادن این عنصر در جای درست خودش (با جابه‌جا کردن عنصر بیشینه با عنصری که جای آن را گرفته است) و سرانجام مرتب‌سازی بازگشتی بقیه‌ی عناصر. به این الگوریتم «مرتب‌سازی با انتخاب» می‌گویند. برتری «مرتب‌سازی با انتخاب» بر «مرتب‌سازی درجی» در این است که تنها به n-1 جابه‌جایی داده (آن هم از نوع عوض کردن جای دو عنصر با یکدیگر) نیاز دارد، در حالی که در مرتب‌سازی درجی، در بدترین حالت، تعداد جابه‌جایی‌های لازمِ داده‌ها از $O(n^2)$ است. از سوی دیگر، چون یافتن عنصر بیشینه به n-1 مقایسه نیاز دارد (درباره‌ی یافتن عنصر بیشینه در بخش ۶-۵ بحث خواهد شد) تعداد کل مقایسه‌ها، همواره از $O(n^2)$ خواهد بود، در حالی که مرتب‌سازی درجی به کمک جست‌وجوی دودویی، تنها به $O(n\log n)$ مقایسه نیازمند است.

همچنین می‌توان از درخت‌های متوازن برای درج و انتخاب کارآمد سود جست (فصل ۴ را ببینید). برای نمونه، با بهره‌گیری از درخت‌های AVL، زمان هر درج از $O(\log n)$ خواهد شد. زمان پویش، یا بررسی تمام عناصر درخت AVL برای به دست آوردن فهرست مرتب‌شده‌ی تمام اعداد آن از $O(n)$

است. اگر بنا به استقرا فرض کنیم چگونگی ساخت درخت AVL برای n-1 عدد را می‌دانیم، آنگاه تنها به یک عمل درج اضافی نیازمندیم که زمان آن از $O(\log n)$ است. در مجموع، زمان درج n عدد در یک درخت AVL تهی از $O(n \log n)$ است و زمان ارائه‌ی فهرست مرتب آن‌ها نیز از $O(n)$ خواهد شد. چنین روشی برای مقادیر بزرگ n بسیار بهتر از مرتب‌سازی درجی یا مرتب‌سازی با انتخاب است، اما برای نگه‌داری اشاره‌گره‌ها به فضای بیش‌تری نیاز دارد. روشن است که این روش «درجا» نیست و چون روشی پیچیده است، کارایی آن از الگوریتم‌هایی که بعداً خواهیم گفت، کم‌تر است. نوشتن برنامه‌ی مرتب‌سازی درجی و مرتب‌سازی با انتخاب ساده است و به عنوان تمرین به خواننده واگذار می‌شود.

۶-۴-۳ مرتب‌سازی ادغامی

برای بهبود کارایی مرتب‌سازی درجی، به مقدار زمانی توجه می‌کنیم که صرف بررسی اعداد مرتب‌شده می‌گردد تا جای درست یک عنصر مشخص شود. پیش‌تر، در بخش ۵-۶ نیز از این ایده سود جستیم. اگر دو مجموعه از اعداد مرتب داشته باشیم، می‌توانیم آن‌ها را با یک بار پویش با یکدیگر ادغام کنیم. برای انجام عمل ادغام، نخست، اعداد مجموعه‌ی دوم را به ترتیب در نظر می‌گیریم و سپس اعداد مجموعه‌ی نخست را به ترتیب، از کوچک به بزرگ در جای درست خود در مجموعه‌ی دوم قرار می‌دهیم. برای بیان دقیق‌تر مطلب، مجموعه نخست را با a_1، a_2، ... و a_n و مجموعه‌ی دوم را با b_1، b_2، ... و b_m نشان می‌دهیم و فرض می‌کنیم هر دو مجموعه به صورت صعودی مرتب شده باشند. مجموعه‌ی نخست را پویش می‌کنیم تا به جای درست b_1 برسیم و سپس b_1 را در آن مکان قرار می‌دهیم. کار را از همین جا ادامه می‌دهیم تا به جای درست b_2 برسیم و ... . از آنجا که bها مرتب هستند، هیچ‌گاه نیاز به عقب‌گرد نداریم. تعداد کل مقایسه‌ها در بدترین حالت، جمع اندازه‌های دو مجموعه است. درباره‌ی جابه‌جایی‌ها چه می‌توان گفت؟ از آنجا که هنگام عمل درج، عناصری جابه‌جا می‌شوند، کارایی الگوریتم کاهش می‌یابد؛ چراکه ممکن است یک عنصر چندین و چند بار جابه‌جا شود. در عوض، از آنجا که عمل ادغام، عناصر مرتب‌شده را یکی یکی ارائه می‌دهد، می‌توانیم آن‌ها را در یک آرایه‌ی موقت بچینیم؛ به این ترتیب، هر عنصر دقیقاً یک بار نسخه‌برداری می‌شود. در کل، مقایسه‌ها و جابه‌جایی‌های انجام‌شده در ادغام دو دنباله‌ی مرتب با اندازه‌های m و n (به شرط در دست‌رس بودن حافظه‌ی اضافی) از O(m+n) خواهد بود.

می‌توان این روال ادغام را پایه و مبنا قرار داد و آن را در مرتب‌سازی به روش تقسیم‌وحل به کار برد. این شیوه‌ی مرتب‌سازی را مرتب‌سازی ادغامی می‌نامند. روش کار الگوریتم چنین است: نخست، دنباله را به دو بخش برابر (یا تقریباً برابر، در حالتی که اندازه‌ی دنباله فرد باشد) تقسیم کرده، سپس به روش بازگشتی هر بخش را به صورت جداگانه مرتب می‌کند. دست آخر، به روشی که پیش‌تر گفته شد،

هر دو بخش مرتب‌شده با یکدیگر ادغام می‌شوند. الگوریتم دقیق این کار در شکل ۶-۷ آمده است. نمونه‌ای از مرتب‌سازی ادغامی (بدون عمل کپی یا نسخه‌برداری) در شکل ۶-۸ نشان داده شده است.

پیچیدگی: $T(n)$ را تعداد مقایسه‌های مرتب‌سازی ادغامی در بدترین حالت در نظر بگیرید. بیایید برای سادگی، فرض کنیم n توانی از ۲ باشد. برای محاسبه‌ی $T(n)$ لازم است این رابطه‌ی بازگشتی را حل کنیم:

$$T(2n) = 2T(n) + O(n), T(2) = 1$$

حل این رابطه‌ی بازگشتی عبارت است از: $T(n) = O(n \log n)$ (فصل ۳ را ببینید)؛ در نتیجه، زمان اجرای این الگوریتم از نظر مجانبی از زمان اجرای مرتب‌سازی درجی و مرتب‌سازی با انتخاب، یعنی از $O(n^2)$ بهتر است؛ اما تعداد جابه‌جایی از $O(n \log n)$ است که از جابه‌جایی‌های مرتب‌سازی با انتخاب، یعنی از $O(n)$، بیش‌تر است.

هرچند مرتب‌سازی ادغامی هنگام بزرگ بودن n از مرتب‌سازی درجی بهتر است، اما هنوز چند ضعف دارد: نخست این‌که پیاده‌سازی آن آسان نیست. دیگر این‌که مرحله‌ی ادغام برای کپی مجموعه‌ی ادغام‌شده، به حافظه‌ی اضافی نیاز دارد. پس مرتب‌سازی ادغامی، الگوریتمی «درجا» نیست. (برخی انواع پیچیده‌ترِ مرتب‌سازی ادغامی از مقدار ثابتی حافظه‌ی اضافی کمک می‌گیرند. بخش مراجع، در پایان فصل را ببینید.) هر بار که دو مجموعه‌ی کوچک‌تر با یکدیگر ادغام شوند، باید عمل کپی نیز انجام گیرد، که طبعاً این روال را کندتر می‌سازد.

الگوریتم: Mergesort(X,n)

ورودی: X (آرایه‌ای با اندیس‌های ۱ تا n)

خروجی: X (مرتب‌شده‌ی آرایه‌ی ورودی)

```
begin
    M_Sort(1,n)
end

procedure M_Sort(Left,Right);
begin
    if Right - Left = 1 then    {بدون بررسی این حالت هم برنامه به درستی کار می‌کند،
                                 اما بررسی آن کارایی برنامه را بیش‌تر می‌کند.}
      if X[Left ] > X[Right] then swap(X[Left ],X[Right])
    else if Left ≠ Right then
            Middle :=⌈1/2(Left + Right)⌉;
            M_Sort(Left,Middle-1);
            M_Sort(Middle,Right);
            {اینک، دو دنباله‌ی مرتب را با یکدیگر ادغام می‌کنیم.}
            i := Left;
            j := Middle;
            k := 0;
            while (i ≤ Middle-1) and (j ≤ Right) do
                  k := k+1;
                  if X[i] ≤ X[j] then
                    TEMP[k] := X[i];
                    i := i + 1
                  else
                      TEMP[k] := X[j];
                      j := j+1;
            if j > Right then
              {انتقال بقیه‌ی عناصر سمت چپ به انتهای آرایه}

              {اگر i ≥ Middle، عناصر سمت راست در جای درست خود قرار دارند.}
              for t := 0 to Middle-1-i do
                  X[Right-t] := X[Middle-1-t];
            {حال، مقدار TEMP را دوباره به X باز می‌گردانیم.}
            for t := 0 to k-1 do
                X[Left+t] := TEMP[t]
end
```

شکل ۶-۷ الگوریتم Mergesort (برای دریافتن مفهوم سمت‌های چپ و راست به شکل ۶-۸ دقت کنید - مترجمان)

۶	۲	۸	۵	۱۰	۹	۱۲	۱	۱۵	۷	۳	۱۳	۴	۱۱	۱۶	۱۴
(۲)	(۶)	۸	۵	۱۰	۹	۱۲	۱	۱۵	۷	۳	۱۳	۴	۱۱	۱۶	۱۴
۲	۶	(۵)	(۸)	۱۰	۹	۱۲	۱	۱۵	۷	۳	۱۳	۴	۱۱	۱۶	۱۴
(۲)	(۵)	(۶)	(۸)	۱۰	۹	۱۲	۱	۱۵	۷	۳	۱۳	۴	۱۱	۱۶	۱۴
۲	۵	۶	۸	(۹)	(۱۰)	۱۲	۱	۱۵	۷	۳	۱۳	۴	۱۱	۱۶	۱۴
۲	۵	۶	۸	۹	۱۰	(۱)	(۱۲)	۱۵	۷	۳	۱۳	۴	۱۱	۱۶	۱۴
۲	۵	۶	۸	(۱)	(۹)	(۱۰)	(۱۲)	۱۵	۷	۳	۱۳	۴	۱۱	۱۶	۱۴
(۱)	(۲)	(۵)	(۶)	(۸)	(۹)	(۱۰)	(۱۲)	۱۵	۷	۳	۱۳	۴	۱۱	۱۶	۱۴
۱	۲	۵	۶	۸	۹	۱۰	۱۲	(۷)	(۱۵)	۳	۱۳	۴	۱۱	۱۶	۱۴
۱	۲	۵	۶	۸	۹	۱۰	۱۲	۷	۱۵	(۳)	(۱۳)	۴	۱۱	۱۶	۱۴
۱	۲	۵	۶	۸	۹	۱۰	۱۲	۳	(۷)	(۱۳)	(۱۵)	۴	۱۱	۱۶	۱۴
۱	۲	۵	۶	۸	۹	۱۰	۱۲	۳	۷	۱۳	۱۵	(۴)	(۱۱)	۱۶	۱۴
۱	۲	۵	۶	۸	۹	۱۰	۱۲	۳	۷	۱۳	۱۵	۴	۱۱	(۱۴)	(۱۶)
۱	۲	۵	۶	۸	۹	۱۰	۱۲	۳	۷	۱۳	۱۵	(۴)	(۱۱)	(۱۴)	(۱۶)
۱	۲	۵	۶	۸	۹	۱۰	۱۲	(۳)	(۴)	(۷)	(۱۱)	(۱۳)	(۱۴)	(۱۵)	(۱۶)
(۱)	(۲)	(۳)	(۴)	(۵)	(۶)	(۷)	(۸)	(۹)	(۱۰)	(۱۱)	(۱۲)	(۱۳)	(۱۴)	(۱۵)	(۱۶)

شکل ۶-۸ نمونه‌ای از مرتب‌سازی ادغامی. سطر نخست ترتیب اولیه را نشان می‌دهد. در هر سطر یکی از دو عمل «تعویض جا» یا «ادغام» انجام شده است. دور اعدادی که در هر گام کاری روی آن‌ها انجام می‌شود، دایره کشیده‌ایم.

۶-۴-۴ مرتب‌سازی سریع

هنگام تحلیل مرتب‌سازی ادغامی، کارایی روش تقسیم‌وحل را به خوبی می‌بینیم. اگر بتوانیم مسأله را به دو زیرمسأله با اندازه‌های مساوی تقسیم کنیم و هر زیرمسأله را جداگانه حل کرده، راه‌حل‌ها را با هم ترکیب کنیم، به الگوریتمی از $O(n \log n)$ می‌رسیم، مشروط بر این که هم گام تقسیم و هم گام ترکیب از $O(n)$ باشند. مشکل مرتب‌سازی ادغامی نیاز به حافظه‌ی اضافی است، زیرا فرایند ادغام به گونه‌ای نیست که بتوانیم جای دقیق هر عنصر را در ترتیب نهایی پیش‌بینی کنیم؟ آیا می‌توانیم روش تقسیم‌وحل را چنان به کار گیریم که به یاری آن بتوان جای عناصر را از پیش تشخیص داد؟ ایده‌ی مرتب‌سازی سریع، انجام کارهای بیش‌تری در گام «تقسیم» است، به گونه‌ای که در گام «حل» کارهای کم‌تری لازم باشد.

فرض کنید عددی مانند x را می‌شناسیم که نیمی از عناصر، بزرگ‌تر یا مساوی آن هستند و نیم دیگر آن‌ها از این عدد کوچک‌ترند. می‌توانیم با مقایسه‌ی همه‌ی عناصر با x، دنباله را به دو بخش تقسیم کنیم. این بخش‌بندی به n-1 مقایسه نیاز دارد. از آنجا که اندازه‌ی دو بخش با هم برابر است،

می‌توان یک بخش را در نیمه‌ی نخست آرایه و بخش دیگر را در نیمه‌ی دوم آن قرار داد. این بخش‌بندی (یعنی گام تقسیم) چنان که بعداً نشان داده خواهد شد، بدون حافظه‌ی اضافی نیز ممکن است. حال می‌توانیم هر بخش را به صورت بازگشتی مرتب کنیم. گام حل ساده است، چون این دو بخش در جای درست خود در آرایه قرار گرفته‌اند. بنابراین، به حافظه‌ی اضافی هم نیازی نداریم.

تا اینجای کار فرض کرده بودیم مقدار x را می‌دانیم، اما معمولاً چنین نیست. چه مقدار x را بدانیم و چه ندانیم، به روشنی می‌توان دید که الگوریتم، بدون توجه به عددی که برای بخش‌بندی به کار رفته است، درست کار خواهد کرد. به عددی که برای بخش‌بندی به کار می‌رود، «محور» می‌گوییم. هدف ما تقسیم آرایه به دو بخش است: یک بخش، با اعدادی بزرگ‌تر از محور و بخش دیگر، با اعداد نابزرگ‌تر از آن. می‌توان کار را این‌گونه انجام داد: از دو اشاره‌گر L و R بهره می‌گیریم. در آغاز، L به سمت چپ و R به سمت راست آرایه اشاره می‌کند. اشاره‌گرها می‌توانند به سوی یکدیگر حرکت کنند. فرضی برای استقرا (یا همان قانون ثابت حلقه) که درستی بخش‌بندی را ضمانت می‌کند، چنین است:

فرض استقرا: در گام kام الگوریتم، برای هر i که L<i، محور، بزرگ‌تر یا مساوی x_iهاست و برای هر j که j<R، محور، کوچک‌تر از x_jهاست.

روشن است که فرض استقرا در آغاز درست است (زیرا هیچ نو زیی در محدوده‌ی شرط گفته‌شده قرار ندارند). هدف ما حرکت L به راست یا R به چپ در گام k+1ام است، بدون آن که فرض استقرا به هم بخورد.

اگر L=R، بخش‌بندی کامل است، به جز احتمالاً در X_L که بعداً روشن برخورد با آن را خواهیم دید؛ اما اگر L<R، دو حالت ممکن است: اگر $X_L \leq Pivot$ یا $X_R > Pivot$ آنگاه اشاره‌گر یا اشاره‌گرها می‌توانند حرکت کنند و فرض استقرا هم برقرار می‌ماند. اگر هم شرط پیش برقرار نباشد؛ یعنی $X_L > Pivot$ و $X_R \leq Pivot$ می‌توانیم X_L و X_R را با هم جابه‌جا کنیم و سپس L و R را به سمت یکدیگر حرکت دهیم. در هر دو حالت دست‌کم یکی از دو اشاره‌گر حرکت خواهد کرد. از این رو، سرانجام اشاره‌گرها به یکدیگر می‌رسند و الگوریتم پایان می‌پذیرد.

آنچه باقی می‌ماند، مشکل گزینش محوری مناسب برای بخش‌بندی و روش برخورد با آخرین گام الگوریتم (یعنی هنگام رسیدن دو اشاره‌گر به یکدیگر) است. الگوریتم‌های تقسیم‌وحل، هنگام برابری اندازه‌ی بخش‌ها بهتر عمل می‌کنند. پس هر چه محور به میانه‌ی اعداد نزدیک‌تر باشد، الگوریتم سریع‌تر اجرا می‌شود. درست است که می‌توان میانه‌ی یک دنباله را یافت (بخش بعد را ببینید) اما این کار به زحمتش نمی‌ارزد. چنان که بعداً تحلیل موضوع را خواهیم دید، خوب است عنصری تصادفی از دنباله را به عنوان محور برگزینیم. اگر ترتیب عناصر دنباله، تصادفی باشد، می‌توان عنصر نخست را به عنوان محور برگزید. از آنجا که این روش ساده است، در الگوریتم شکل ۶-۹ نیز همین روش را به کار برده‌ایم.

پس از برگزیدن نخستین عنصر به عنوان محور، عمل بخش‌بندی را انجام می‌دهیم و در آخرین گام این عمل، نخستین عنصر را با X_L جابه‌جا می‌کنیم (نویسنده در اینجا باید می‌گفت در آخرین گام بخش‌بندی، نخستین عنصر را با X_R جابه‌جا می‌کنیم - مترجمان). با این کار، عنصر نخست که محور هم بود، در جای درست خود قرار می‌گیرد. هنگام بحث درباره‌ی پیچیدگی الگوریتم، به دیگر روش‌های بخش‌بندی نیز اشاره خواهیم کرد. اگر روش دیگری را هم، برای برگزیدن محور از عناصر دنباله به کار بریم، با جابه‌جا کردن محور برگزیده شده با عنصر نخست دنباله، بازهم می‌توان الگوریتم ۶-۹ را به کار گرفت.

الگوریتم: Partition(X,Left,Right)

ورودی: X (یک آرایه)، Left (حد پایین آرایه) و Right (حد بالای آرایه)

خروجی: X و Middle به گونه‌ای که برای همه‌ی نهای کوچک‌تر یا مساوی Middle: X[i]≤X[Middle] و برای همه‌ی jهای بزرگ‌تر از Middle: X[j]>X[Middle].

```
begin
    Pivot := X[Left];
    L := Left; R := Right;
    while L < R do
        while X[L] ≤ pivot and L ≤ Right do L := L+1;
        while X[R] > pivot and R ≥ Left do R := R - 1;
        if L < R then
        X[L] را با X[R] جابه‌جا کن;
    Middle := R;
    X[Middle] را با X[Left] جابه‌جا کن
end
```

شکل ۶-۹ الگوریتم Partition

مثالی از الگوریتم Partition در شکل ۶-۱۰ ارائه شده است. محور همان عدد نخست (یعنی ۶) است و دور اعدادی که تازه جابه‌جا شده‌اند، دایره کشیده‌ایم. پس از انجام سه جابه‌جایی، P_h به X[6]=1 اشاره می‌کند و P_L به X[7]=12 (P_h و P_L به ترتیب همان R و L هستند که پیش‌تر به آن‌ها اشاره شد - مترجمان). آخرین جابه‌جایی بین عنصر وسط (۱) و محور (۶) انجام می‌گیرد. پس از این جابه‌جایی، همه‌ی عناصر سمت چپ محور، کوچک‌تر یا مساوی آن و همه‌ی عناصر سمت راست محور، بزرگ‌تر از آن هستند. این دو زیردنباله (یعنی اندیس‌های ۱ تا ۶ و اندیس‌های ۷ تا ۱۶) را می‌توان به صورت بازگشتی مرتب کرد. روشن است که مرتب‌سازی سریع، الگوریتمی درجاست و نیاز به حافظه‌ی اضافی ندارد. الگوریتم این روش مرتب‌سازی در شکل ۶-۱۱ و مثالی از آن در شکل ۶-۱۲ نشان داده شده است.

۶	۲	۸	۵	۱۰	۹	۱۲	۱	۱۵	۷	۳	۱۳	۴	۱۱	۱۶	۱۴
۶	۲	(۴)	۵	۱۰	۹	۱۲	۱	۱۵	۷	۳	۱۳	(۸)	۱۱	۱۶	۱۴
۶	۲	۴	۵	(۳)	۹	۱۲	۱	۱۵	۷	(۱۰)	۱۳	۸	۱۱	۱۶	۱۴
۶	۲	۴	۵	۳	(۱)	۱۲	(۹)	۱۵	۷	۱۰	۱۳	۸	۱۱	۱۶	۱۴
(۱)	۲	۴	۵	۳	(۶)	۱۲	۹	۱۵	۷	۱۰	۱۳	۸	۱۱	۱۶	۱۴

شکل ۶-۱۰ بخش‌بندی یک آرایه حول محور ۶

الگوریتم: Quicksort(X,n)

ورودی: X (آرایه‌ای با اندیس‌های 1 تا n)

خروجی: X (مرتب‌شده‌ی آرایه‌ی ورودی)

```
begin
      Q_Sort(1,n)
end

procedure Q_Sort(Left,Right)
begin
      if Left < Right then
        Partition(X,Left,Right);
        Q_Sort(Left,Middle-1);
        Q_Sort(Middle+1,Right)
end
```

شکل ۶-۱۱ الگوریتم Quicksort

۶	۲	۸	۵	۱۰	۹	۱۲	۱	۱۵	۷	۳	۱۳	۴	۱۱	۱۶	۱۴
۱	۲	۴	۵	۳	(۶)	۱۲	۹	۱۵	۷	۱۰	۱۳	۸	۱۱	۱۶	۱۴
(۱)	۲	۴	۵	۳	(۶)	۱۲	۹	۱۵	۷	۱۰	۱۳	۸	۱۱	۱۶	۱۴
(۱)	(۲)	۴	۵	۳	(۶)	۱۲	۹	۱۵	۷	۱۰	۱۳	۸	۱۱	۱۶	۱۴
(۱)	(۲)	۳	(۴)	۵	(۶)	۱۲	۹	۱۵	۷	۱۰	۱۳	۸	۱۱	۱۶	۱۴
(۱)	(۲)	۳	(۴)	۵	(۶)	۸	۹	۱۱	۷	۱۰	(۱۲)	۱۳	۱۵	۱۶	۱۴
(۱)	(۲)	۳	(۴)	۵	(۶)	۷	(۸)	۱۱	۹	۱۰	(۱۲)	۱۳	۱۵	۱۶	۱۴
(۱)	(۲)	۳	(۴)	۵	(۶)	۷	(۸)	۱۰	۹	(۱۱)	(۱۲)	۱۳	۱۵	۱۶	۱۴
(۱)	(۲)	۳	(۴)	۵	(۶)	۷	(۸)	۹	(۱۰)	(۱۱)	(۱۲)	۱۳	۱۵	۱۶	۱۴
(۱)	(۲)	۳	(۴)	۵	(۶)	۷	(۸)	۹	(۱۰)	(۱۱)	(۱۲)	(۱۳)	۱۵	۱۶	۱۴
(۱)	(۲)	۳	(۴)	۵	(۶)	۷	(۸)	۹	(۱۰)	(۱۱)	(۱۲)	(۱۳)	۱۴	(۱۵)	۱۶

شکل ۶-۱۲ نمونه‌ای از مرتب‌سازی سریع، سطر نخست، ورودی الگوریتم است. در هر سطر دور محور برگزیده‌شده دایره کشیده شده است. روشن است هنگامی که یک عدد بین دو محور قرار می‌گیرد، در جای درست خود واقع است.

پیچیدگی: زمان اجرای مرتب‌سازی سریع به ورودی و محور برگزیده‌شده‌ی آن بستگی دارد. اگر محور، دنباله را همواره به دو بخش برابر تقسیم کند، آنگاه رابطه‌ی بازگشتی زمان اجرا $T(n) = 2T(n/2) + O(n)$ و $T(2) = 1$ خواهد بود که از آن نتیجه می‌شود: $T(n) = O(n \log n)$. خواهیم دید که حتا در شرایط ضعیف‌تری هم، زمان اجرا از $O(n \log n)$ است. به هر حال، اگر محور، بسیار نزدیک به یکی از دو سوی دنباله باشد، زمان اجرا بسیار بیش‌تر می‌شود. برای مثال، اگر محور، کوچک‌ترین عنصر دنباله باشد، نخستین بخش‌بندی به n-1 مقایسه نیاز دارد و در پایان تنها محور در جای درست خود قرار می‌گیرد. بدین ترتیب، اگر دنباله از پیش به صورت صعودی مرتب باشد و همواره عنصر نخست را به عنوان محور برگزینیم، زمان اجرای الگوریتم از $O(n^2)$ خواهد شد. می‌توانیم با مقایسه سه عنصر نخست، وسط و آخر، و گزینش میانه‌ی آن‌ها (یعنی دومین عنصر از نظر بزرگی در بین آن‌ها) به عنوان محور، از بروز بدترین حالت پیش‌گیری کنیم. روش مطمئن‌تر، گزینش محور از بین عناصری است که خودِ آن عناصر به صورت تصادفی انتخاب شده‌اند. زمان اجرای مرتب‌سازی سریع هنوز هم در بدترین حالت از $O(n^2)$ است، چراکه بازهم ممکن است محور کوچک‌ترین عنصر دنباله باشد؛ هرچند احتمال چنین رخدادی بسیار اندک است. اینک زمان اجرای الگوریتم را در حالت میانگین به دست می‌آوریم:

فرض می‌کنیم احتمال محور شدن همه‌ی x_iها یکسان باشد. اگر iامین عنصر از نظر کوچکی محور شود، زمان اجرای T(n) برای مرتب‌سازی سریع عبارت است از:

$$T(n) = n - 1 + T(i-1) + T(n-i)$$

(n-1 مقایسه برای بخش‌بندی لازم است و ما باید دو دنباله‌ی کوچک‌تر با اندازه‌های i-1 و n-i را مرتب کنیم.) اگر احتمال گزینش همه‌ی عناصر یکسان باشد، آنگاه میانگین زمان اجرا برابر است با:

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i))$$

$$= n - 1 + \frac{1}{n}\sum_{i=1}^{n} T(i-1) + \frac{1}{n}\sum_{i=1}^{n} T(n-i)$$

$$= n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$$

این رابطه، یک رابطه‌ی بازگشتی با حافظه‌ی کامل است. درباره‌ی این رابطه در بخش ۳-۵-۳ بحث کردیم و نشان دادیم که $T(n)$ از $O(n \log n)$ است. از این رو، بی‌شک مرتب‌سازی سریع در حالت میانگین سریع است.

در عمل، سرعت «مرتب‌سازی سریع» بسیار بالاست و این الگوریتم، شایستگی نامی را که به آن داده‌اند، دارد. دلیل اصلیِ سرعت این روش مرتب‌سازی، جدای از شیوه‌ی تقسیم‌وحل زیبایی که دارد، مقایسه‌ی همه‌ی عناصر با محوری یکسان است. بنابراین با نگه‌داری محور در یک «ثبات» کم‌تر

نیازمند دست‌رسی به حافظه هستیم و این کار در بیش‌تر رایانه‌ها مقدار چشم‌گیری از زمان اجرا می‌کاهد.

یکی از راه‌های بهبود زمان اجرای مرتب‌سازی سریع به‌کارگیری روشی است که ما آن را «گزینش هوشمندانه‌ی پایه‌ی استقرا» می‌نامیم. در این روش، پایه‌ی استقرا از ۱ آغاز نمی‌شود. مرتب‌سازی سریع، چنان که در اینجا بیان شد، به صورت بازگشتی کار خود را انجام می‌دهد تا آن که به حالت پایه برسد (اندازه‌ی دنباله در حالت پایه ۱ است) اما روش‌های ساده‌ی مرتب‌سازی، مانند مرتب‌سازی درجی و مرتب‌سازی با انتخاب، کارایی خوبی در دنباله‌های کوچک دارند؛ در صورتی که کارایی مرتب‌سازی سریع تنها در دنباله‌های بزرگ نمایان می‌شود. بنابراین می‌توانیم اندازه‌ی حالت پایه‌ی مرتب‌سازی سریع را مقداری بیش از ۱ بگیریم (ظاهراً ۱۰ یا ۲۰ اندازه‌ی مناسبی است، اگرچه به روش پیاده‌سازی هم بستگی دارد) و کار حالت پایه را با مرتب‌سازی درجی انجام دهیم. (به عبارت دیگر، به جای شرط Left<Right از Left<Right-Threshold بهره می‌گیریم و یک بخش else نیز برای اجرای مرتب‌سازی درجی می‌افزاییم.) (منظور از Threshold اندازه‌ی تازه‌ای است که برای حالت پایه در نظر گرفته‌ایم - مترجمان) این روش، زمان اجرای مرتب‌سازی سریع را به اندازه‌ی یک مقدار ثابت کوچک بهبود می‌بخشد. در بخش ۶-۱۱-۳ خواهیم دید که چگونه با گزینش حالت پایه‌ی استقرا می‌توانیم زمان اجرای یک الگوریتم را به صورت مجانبی بهبود دهیم.

۶-۴-۵ مرتب‌سازی هرمی

مرتب‌سازی هرمی، الگوریتم سریع دیگری برای مرتب‌سازی است. سرعت این مرتب‌سازی عملاً برای مقادیر بزرگ n از سرعت مرتب‌سازی سریع کم‌تر است. از سوی دیگر، برخلاف مرتب‌سازی سریع کارایی این الگوریتم تضمین شده است. زمان اجرای بدترین حالت مرتب‌سازی هرمی، مانند مرتب‌سازی ادغامی از $O(n\log n)$ است، اما برخلاف مرتب‌سازی ادغامی، الگوریتمی درجاست. در این بخش، روی ساخت هرم تمرکز می‌کنیم. الگوریتم ساخت هرم نمونه‌ای از در هم آمیختن تحلیل و طراحی الگوریتم‌هاست.

از آنجا که پیاده‌سازی هرم را در فصل ۴ بررسی کردیم، دیگر وارد جزئیات نمی‌شویم؛ تنها فرض می‌کنیم عناصر در آرایه‌ای مانند A[1..n] قرار گرفته‌اند و این آرایه به ترتیبی که گفته خواهد شد، نشان‌دهنده‌ی یک درخت است: ریشه درخت در A[1] نگه‌داری می‌شود و فرزندان هر گره A[i] (اگر درخت، گرهی داشته باشد) نیز، در A[2i] و A[2i+1] ذخیره می‌گردند. چنین آرایه‌ای دارای خاصیت هرم است، اگر مقدار هر گره بزرگ‌تر یا مساوی مقدار فرزندانش باشد.

مرتب‌سازی هرمی این‌گونه کار می‌کند: ورودی، آرایه‌ی A[1..n] است. نخست، ترتیب عناصر آرایه را چنان تغییر می‌دهیم که آرایه، یک هرم شود. روش ساخت هرم را بعداً توضیح خواهیم داد. اگر

A، یک هرم باشد، آنگاه A[1] عنصر بیشینه‌ی آرایه است. A[1] را با A[n] جابه‌جا می‌کنیم؛ به این ترتیب در A[n] عنصر مناسبی قرار خواهد گرفت. سپس آرایه‌ی A[1..n-1] را در نظر می‌گیریم. بازهم، ترتیب عناصر آرایه را چنان تغییر می‌دهیم تا دوباره یک هرم بسازند (در اینجا، تنها باید نگران A[1] تازه باشیم). A[1] را با A[n-1] جابه‌جا می‌کنیم و کار را با A[1..n-2] ادامه می‌دهیم. پس از ساخته شدن هرم باید n-1 مرحله طی شود که هر یک از آن‌ها شامل جابه‌جایی عناصر و تغییر ترتیب آن‌ها برای برقراری دوباره‌ی خاصیت هرم است. تغییر ترتیب عناصر هرم، پس از عمل جابه‌جایی، در واقع همان الگوریتم Remove_Max_from_Heap است که در شکل ۴-۳-۲ ارائه گردید. چگونگی ساختن هرم، خود مسأله‌ی جالبی است که به دقت در اینجا توضیح داده خواهد شد. چون به ازای هر جابه‌جایی $O(\log n)$ عمل انجام می‌شود، پس زمان لازم برای مرتب‌سازی هرمی، بعد از ساخته شدن هرم، از $O(n \log n)$ است. روشن است که مرتب‌سازی هرمی الگوریتمی درجاست. این الگوریتم در شکل ۶-۱۳ آمده است.

الگوریتم: Heapsort(X,n)

ورودی: X (آرایه‌ای با اندیس‌های 1 تا n)

خروجی: X (مرتب‌شده‌ی آرایه‌ی ورودی)

```
begin
    Build_Heap(X); {برای آشنایی با این الگوریتم، ادامه‌ی مطلب را بخوانید.}
    for i := n downto 2 do
        swap(A[1],A[i]);
        Rearrange_Heap(i-1)
        {در واقع همان روال Remove_Max_from_Heap است که در شکل ۴-۷
        نشان داده شد.}
end
```

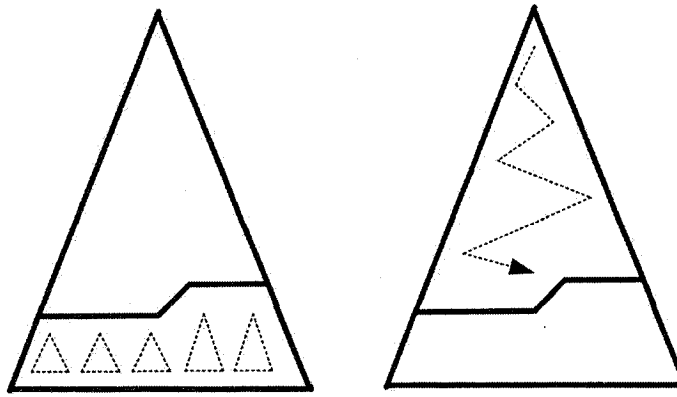
شکل ۶-۱۳ الگوریتم Heapsort

روش ساخت heap یا هرم

اینک، شیوه‌ی ساخت هرم از روی آرایه‌ای دل‌خواه را بررسی می‌کنیم.

> **مسأله:** ترتیب عناصر آرایه‌ی دل‌خواه A[1..n] را چنان تغییر دهید که خاصیت هرم در آن برقرار شود.

برای ساخت هرم دو روش سرراست وجود دارد: بالا به پایین و پایین به بالا. این دو روش، به ترتیب، متناظر با پویش آرایه‌ی نمایش‌دهنده‌ی هرم، از چپ به راست و از راست به چپ هستند. شکل ۶-۱۴ هر دو روش را نشان می‌دهد. نخست با استقرا هر دو روش را توصیف می‌کنیم و سپس نشان می‌دهیم بین کارایی آن‌ها اختلافی قابل توجه وجود دارد.



شکل ۶-۱۴ ساخت هرم به دو روش بالا به پایین و پایین به بالا

نخست، روش بالا به پایین را بررسی می‌کنیم (یعنی آرایه را از چپ به راست می‌پیماییم).

فرض استقرا (روش بالا به پایین): آرایه‌ی A[1..i] یک هرم است.

حالت پایه روشن است، زیرا A[1] خودش به تنهایی یک هرم است. بخش اصلی الگوریتم این است که A[i+1] را به هرم A[1..i] بیفزاییم؛ این کار دقیقاً مانند افزودن یک عنصر به هرم است (فصل ۴ را ببینید). A[i+1] با والدش مقایسه می‌شود و عناصر را آن قدر جابه‌جا می‌کنیم تا مقدار والد تازه از مقدار فرزند بیش‌تر باشد. تعداد مقایسه‌ها در بدترین حالت $\lfloor \log_2(i+1) \rfloor$ خواهد بود.

حال به روش پایین به بالا می‌پردازیم (یعنی آرایه از راست به چپ پیموده می‌شود). اگر آرایه‌ی A[i+1..n] یک هرم بود، بسیار خوب می‌شد! چراکه به راحتی می‌توانستیم عنصر تازه‌ی A[i] را به آن بیفزاییم، اما آرایه‌ی A[i+1..n] یک هرم نیست، بلکه مجموعه‌ای از هرم‌هاست. (دقت کنید که A[i+1..n] یک آرایه‌ی جداگانه نیست، بلکه آن را بخشی از درختی که با A[1..n] نشان داده شد، در نظر گرفته‌ایم. بنابراین، فرض استقرا اندکی پیچیده‌تر می‌شود.

فرض استقرا (روش پایین به بالا): همه‌ی درخت‌هایی که در آرایه‌ی A[i+1..n] قرار دارند، هرم هستند.

از آنجا که A[n] خود به تنهایی یک هرم است، پس حالت پایه برقرار می‌شود، اما روش بهتری هم وجود دارد (که با ساخت الگوریتم از روی آن به زمان اجرای بهتری دست خواهیم یافت. بخش پیچیدگی را ببینید - مترجمان). آرایه‌ی $A[\lfloor n/2 \rfloor + 1..n]$ برگ‌های درخت را نشان می‌دهد؛ یعنی درخت‌های متناظر با $A[\lfloor n/2 \rfloor + 1..n]$ همگی تک‌عنصری هستند، از این رو خاصیت هرم را نیز دارند. پس فرایند استقرا می‌تواند از $\lfloor n/2 \rfloor$ آغاز شود. همین نکته نشان می‌دهد که رویکرد پایین به بالا ممکن است بهتر باشد. به این ترتیب، نیمی از کار خود به خود انجام شده است. (این مثال نشان می‌دهد که باید در گزینش حالت پایه دقت کنیم.)

اینک A[i] را در نظر بگیرید. حداکثر دو فرزند (A[2i] و A[2i+1]) دارد که خود این دو بنا به فرض استقرا، ریشه‌های دو هرم معتبرند. گنجاندن A[i] در هرم کار سرراستی است. A[i] با بیشینه‌ی دو فرزندش مقایسه می‌گردد و در صورت نیاز، با فرزند بزرگ‌تر جابه‌جا می‌شود. این کار، شبیه عمل حذف در هرم است (فصل ۴ را ببینید). جابه‌جایی‌ها به سمت پایین درخت ادامه پیدا می‌کند تا آن که مقدار پیشین A[i] به جایی برسد که از هر دو فرزندش بزرگ‌تر باشد. نمونه‌ای از شیوه‌ی ساخت پایین به بالا در شکل ۶-۱۵ نشان داده شده است. از آنجا که ارتفاع A[i]، $\lfloor \log_2(n/i) \rfloor$ است، پس تعداد مقایسه‌ها در بدترین حالت $2\lfloor \log_2(n/i) \rfloor$ خواهد شد.

پیچیدگی روش بالا به پایین: i امین گام، حداکثر به $\lfloor \log_2 n \rfloor$ مقایسه نیاز دارد (زیرا $\lfloor \log_2 n \rfloor \le \lfloor \log_2 i \rfloor$). از این رو، زمان اجرا از $O(n \log n)$ خواهد بود. علاوه بر آن، $O(n \log n)$ برآورد بسیار بالایی از زمان اجرا نیست؛ زیرا:

$$\sum_{i=1}^{n} \lfloor \log \tfrac{i}{2} \rfloor \ge \sum_{i=n/2}^{n} \lfloor \log \tfrac{i}{2} \rfloor \ge n/2 \lfloor \log_2^{(n/2)} \rfloor = \Omega(n \log n)$$

۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱	۱۲	۱۳	۱۴	۱۵	۱۶
۶	۲	۸	۵	۱۰	۹	۱۲	۱	۱۵	۷	۳	۱۳	۴	۱۱	۱۶	۱۴
۲	۶	۸	۵	۱۰	۹	۱۲	(۱۴)	۱۵	۷	۳	۱۳	۴	۱۱	۱۶	(۱)
۲	۶	۸	۵	۱۰	۹	(۱۶)	۱۴	۱۵	۷	۳	۱۳	۴	۱۱	(۱۲)	۱
۲	۶	۸	۵	۱۰	(۱۳)	۱۶	۱۴	۱۵	۷	۳	(۹)	۴	۱۱	۱۲	۱
۲	۶	۸	۵	۱۰	۱۳	۱۶	۱۴	۱۵	۷	۳	۹	۴	۱۱	۱۲	۱
۲	۶	۸	(۱۵)	۱۰	۱۳	۱۶	۱۴	(۵)	۷	۳	۹	۴	۱۱	۱۲	۱
۲	۶	(۱۶)	۱۵	۱۰	۱۳	(۱۲)	۱۴	۵	۷	۳	۹	۴	۱۱	(۸)	۱
۲	(۱۵)	۱۶	(۱۴)	۱۰	۱۳	۱۲	(۶)	۵	۷	۳	۹	۴	۱۱	۸	۱
(۱۶)	۱۵	(۱۳)	۱۴	۱۰	(۹)	۱۲	۶	۵	۷	۳	(۲)	۴	۱۱	۸	۱

شکل ۶-۱۵ نمونه‌ای از ساخت هرم با روش پایین به بالا. اعداد بالای جدول، اندیس‌ها را نشان می‌دهند. در هر گام اعدادی را که دورشان دایره کشیده شده است، با یکدیگر جابه‌جا کرده‌ایم.

پیچیدگی روش پایین به بالا: تعداد مقایسه‌های هر گام، حداکثر دو برابر ارتفاع گره متناظر است (چون ممکن است گره مورد نظر با هر دو فرزندش مقایسه شده، سپس جابه‌جا گردد و همین روند تا پایین درخت ادامه یابد). بنابراین، پیچیدگی، حداکثر دو برابر مجموع ارتفاع همه‌ی گره‌های درخت است. می‌خواهیم این مجموع را محاسبه کنیم. بیایید نخست مسأله را درباره‌ی درخت‌های کامل حل کنیم. مجموع ارتفاع همه‌ی گره‌های درختی کامل به ارتفاع i را با H(i) نشان می‌دهیم. با توجه به این که درختی به ارتفاع i شامل دو درخت به ارتفاع i-1 و یک ریشه است، می‌توانیم برای H(i) رابطه‌ی بازگشتی H(i)=2H(i-1)+i و H(0)=0 را به دست آوریم. می‌توان (با استقرا) نشان داد که پاسخ این

رابطه عبارت است از: $H(i) = 2^{i+1} - (i+2)$. از آنجا که تعداد گره‌های یک درخت کامل دودویی به ارتفاع i، $2^{i+1} - 1$ است، نتیجه می‌شود که پیچیدگی ساخت هرم با روش پایین به بالا برای درخت‌های کامل از $O(n)$ است (در این حالت، هرم $2^k - 1$ گره دارد). پیچیدگی ساخت یک هرم با n گره به گونه‌ای که $2^k \le n < 2^{k+1} - 1$، از پیچیدگی ساخت یک هرم با $2^{k+1} - 1$ گره بیش‌تر نیست. پس پیچیدگی ساخت این هرم نیز از $O(n)$ است. (تحلیلی دقیق‌تر نشان می‌دهد که مقدار ثابت آن افزایش نمی‌یابد؛ تمرین ۶-۲۳را ببینید.) علت سریع‌تر بودنِ رویکرد پایین به بالا از رویکرد بالا به پایین این است که تعداد گره‌های پایین درخت از تعداد گره‌های بالای آن خیلی خیلی بیش‌تر است. پس بهتر است به جای کمینه‌سازی اعمال برای گره‌های بالایی، اعمال مربوط به گره‌های پایینی را کمینه کنیم.

این مورد، نمونه‌ای است که در آن به‌کارگیری ترتیب دیگری برای استقرا به الگوریتم بهتری منجر می‌گردد. اگرچه شیوه‌ی بالا به پایین، سرراست‌تر و شهودی‌تر است، اما کارایی شیوه‌ی پایین به بالا بیش‌تر است.

توجه: خلاصه کردن مرتب‌سازی، آن هم در چند سطر دشوار است. روش‌های اصلی بیان‌شده در این بخش، گونه‌هایی از روش تقسیم‌وحل هستند. دیدیم می‌ارزد که زمان بیش‌تری را صرف «تقسیم» کنیم تا «حل» آسان‌تر شود. این کار در استقرا، هم‌ارز به‌کارگیری ترتیب‌هایی متفاوت برای استقرا و به خصوص، به کار بستن استقرا برای زیرمجموعه‌های ویژه، به جای عناصر دل‌خواه است. موردی را هم دیدیم که نشان می‌داد «تحلیل» باید پا به پای «طراحی» پیش رود. با کمی تمرین یاد می‌گیرید که چگونه شهودی را که درباره‌ی کارایی الگوریتم دارید، حتا پیش از انجام تحلیل آن، گام به گام پیش ببرید. این شهود، شما را در یافتن الگوریتمی بهتر یاری می‌دهد. معمولاً (ولی نه همیشه!) حقیقت، چندان دور از شهود نیست.

۶-۴-۶ حد پایین مرتب‌سازی

مرتب‌سازی را با الگوریتمی از $O(n^2)$ آغاز کردیم و آن را تا الگوریتمی از $O(n \log n)$ بهبود بخشیدیم. آیا می‌توان بیش‌تر از این هم، الگوریتم مرتب‌سازی را بهبود داد؟ اثبات وجود یک حد پایین برای حل یک مسأله‌ی مشخص، در واقع، اثبات این است که مسأله، راه‌حلی بهتر از این حد ندارد. یافتن حد پایین برای یک مسأله کار بسیار دشواری است، زیرا باید تمام الگوریتم‌های ممکن (و نه تنها یک رویکرد خاص) را در نظر بگیریم. پس لازم است مدلی تعریف کنیم که متناظر با الگوریتمی دل‌خواه (و نامشخص) باشد و سپس ثابت کنیم زمان اجرای هر الگوریتمی که با این مدل جور است، بزرگ‌تر یا مساوی حد پایین مورد نظر خواهد بود. در این بخش با مدلی از این دست آشنا می‌شویم که «درخت

تصمیم‌گیری» نامیده می‌شود. محاسباتی که بیش‌تر شامل مقایسه‌اند، برای مدل شدن با درخت تصمیم‌گیری مناسبند. درخت‌های تصمیم‌گیری برخلاف ماشین‌های تورینگ یا ماشین‌های با دسترسی تصادفی، مدلی عمومی برای محاسبه نیستند؛ اما حد پایینی که با درخت‌های تصمیم‌گیری به دست می‌آید، حد پایین مدل‌های عمومی هم خواهد بود و کار کردن با درخت‌های تصمیم‌گیری از جهات بسیاری ساده‌تر و آسان‌تر است. درخت‌های تصمیم‌گیری گونه‌های فراوانی دارند و چندین برهان آشنا برای یافتن بهینه‌ی حد پایین کاملاً وابسته به آن‌هاست.

درخت‌های تصمیم‌گیری را درخت‌هایی دودویی تعریف می‌کنیم که دو نوع گره دارند: گره‌های داخلی و برگ‌ها. هر گره داخلی متناظر با یک پرسش است که پاسخ آن دو حالت دارد و هر حالت پاسخ نیز با یکی از شاخه‌های آن گره متناظر است. هر برگ این درخت، متناظر با یکی از خروجی‌های ممکن است. فرض می‌کنیم ورودی، دنباله‌ای از اعداد x_1 ، x_2 ، ... و x_n باشد. هر محاسبه از ریشه‌ی درخت آغاز می‌گردد. در هر گره، ورودی در برابر پرسش قرار می‌گیرد و بنا به پاسخ آن، یکی از دو شاخه‌ی چپ یا راست برگزیده می‌شود. هر برگ، متناظر با یکی از ترتیب‌های ورودی است و ترتیب عناصر ورودی از روی برگی که به آن رسیده‌ایم، مشخص می‌شود. زمان اجرای بدترین حالتِ متناظر با درخت T، ارتفاع آن درخت است که نشان‌دهنده‌ی کار لازم برای ورودی در بدترین حالت است. به این ترتیب، هر درخت تصمیم‌گیری متناظر با یک الگوریتم است. هرچند درخت‌های تصمیم‌گیری نمی‌توانند همه‌ی الگوریتم‌ها را مدل کنند (برای نمونه، الگوریتم محاسبه‌ی ریشه دوم یک عدد) اما برای الگوریتم‌های مبتنی بر مقایسه، مدل‌هایی پذیرفتنی هستند. با یافتن حد پایینی برای درخت تصمیم‌گیری درمی‌یابیم که هیچ الگوریتمی با این قالب (یعنی با مقایسه - مترجمان) نمی‌تواند عمل‌کردی بهتر از این حد داشته باشد. حال، از درخت‌های تصمیم‌گیری کمک می‌گیریم تا حد پایینی برای مرتب‌سازی بیابیم.

□ قضیه‌ی ۶-۱

اگر یک الگوریتم مرتب‌سازی بر مبنای درخت تصمیم‌گیری باشد، زمان اجرای آن از $\Omega(n \log n)$ خواهد بود.

برهان: دنباله‌ی ورودیِ الگوریتم مرتب‌سازی را x_1 ، x_2 ، ... و x_n بگیرید. خروجی، همین دنباله است اما به صورت مرتب‌شده. می‌توان خروجی را به صورت «جای‌گشتی» از ورودی دید؛ یعنی خروجی مشخص می‌کند که چگونه این عناصر را بچینیم تا مرتب شوند. از آنجا که ترتیب ورودی دل‌خواه است، پس خروجی می‌تواند هر یک از جای‌گشت‌های ورودی باشد. یک الگوریتم مرتب‌سازی، درست محسوب می‌شود، اگر از پس تمام ورودی‌های ممکن برآید. بنابراین هر جای‌گشت (چینش) از (۱، ۲، ... و n) باید بیانگر یک خروجی ممکن در درخت تصمیم‌گیری مرتب‌سازی باشد. خروجی یک درخت تصمیم‌گیری، برگ‌های آن هستند. از آنجا که دو جای‌گشت متفاوت، بیانگر دو خروجی گوناگون هستند، پس این دو جای‌گشت باید با برگ‌های متفاوتی نیز متناظر باشند. بنابراین، باید دست‌کم یک برگ برای هر جای‌گشت ممکن وجود داشته باشد. تعداد کل جای‌گشت‌های n عنصر، $n!$ است. چون درخت را

دودویی در نظر گرفته‌ایم، پس ارتفاع درخت، دست‌کم $\log_2^{(n!)}$ خواهد بود، اما بنا به تقریب استرلینگ:

$$n! = \sqrt{2\pi n}(\frac{n}{e})^n (1 + O(1/n))$$

از این رو داریم: $\log_2^{(n!)} = \Omega(n \log n)$ و برهان کامل می‌شود.

□

این نوع حد پایین، حد پایین نظری نامیده می‌شود، چراکه بر مبنای اندازه‌گیری عملی زمان اجرا با زمان‌سنج به دست نیامده است (حتا نوع پرسش‌ها را نیز تعریف نکرده‌ایم) بلکه تنها بر مبنای حجم اطلاعات خروجی حاصل شده است. در این مورد، حد پایین نشان می‌دهد که در بدترین حالت، هر الگوریتم مرتب‌سازی به $\Omega(n \log n)$ مقایسه نیاز دارد، زیرا باید بتواند !n حالت گوناگون را از یکدیگر تشخیص دهد، در حالی که در هر بار تنها می‌تواند دو تای آن‌ها را از هم متمایز سازد. ما می‌توانستیم درخت تصمیم‌گیری را درختی با سه فرزند تعریف کنیم (که برای نمونه، فرزندان به ترتیب متناظر با $<$، $=$ و $>$ باشند). در این حالت، ارتفاع، دست‌کم $\log_3^{n!}$ می‌شد که بازهم از $\Omega(n \log n)$ است. به عبارت دیگر، اگر در یک درخت تصمیم‌گیری تعداد شاخه‌های خارج‌شده از هر گره ثابت باشد، حد پایین $\Omega(n \log n)$ برقرار است.

از اثبات این حد پایین، تنها نتیجه می‌شود که هیچ الگوریتمی بر مبنای مقایسه، برای مرتب‌سازی وجود ندارد که از $\Omega(n \log n)$ سریع‌تر باشد. شاید بتوان با بهینه‌سازی ویژگی‌هایی از کلیدها و انجام اعمال جبری روی آن‌ها، عناصر را بسیار سریع‌تر از این حد پایین هم مرتب کرد. برای نمونه، اگر n عنصر داشته باشیم که مقدارشان بین ۱ تا ۴n باشد، آنگاه مرتب‌سازی سطلی در زمانی از $O(n)$، فهرست مرتب‌شده‌ی آن‌ها را به ما می‌دهد. این موضوع، با حد پایین گفته‌شده هیچ تناقضی ندارد، زیرا مرتب‌سازی سطلی از مقایسه استفاده نمی‌کند؛ بلکه از این واقعیت بهره می‌برد که می‌توان به گونه‌ای کارآمد، ارقام را به عنوان نشانی (برای سطل‌ها) در نظر گرفت.

معمولاً هنگام بحث درباره‌ی درخت‌های تصمیم‌گیری از اندازه‌ی آن‌ها چشم‌پوشی کرده، تنها روی ارتفاعشان متمرکز می‌شویم. حتا یک الگوریتم ساده که زمان خطی دارد نیز، ممکن است متناظر با یک درخت تصمیم‌گیری با تعداد گره‌هایی نمایی باشد. چون بنا نیست واقعاً درخت را بسازیم، اندازه‌ی آن بی‌اهمیت است. از این درخت، تنها در جایگاه ابزاری برای اثبات حد پایین بهره می‌بریم. نادیده گرفتن اندازه، اثبات را قدرتمندتر می‌کند و حتا می‌توان برهان را در برنامه‌هایی با اندازه‌های نمایی نیز به کار گرفت. از سویی، این روش ممکن است آن قدر زمخت باشد که نتوان آن را برای یافتن حد پایین برخی مسائل به کار برد (یعنی مسائلی که نمی‌توان آن‌ها را با برنامه‌هایی با اندازه‌های عملی حل کرد، اما با برنامه‌هایی که اندازه‌ی نمایی دارند، حل‌شدنی هستند؛ مانند برنامه‌ای با یک جدول برای همه‌ی

حالت‌های ممکن). درخت‌های تصمیم‌گیری مدل‌های یک‌نواختی برای محاسبه نیستند؛ یعنی این درخت‌ها به n (اندازه ورودی) وابسته‌اند. پس می‌توان برای مقدارهای گوناگون n، درخت‌های متفاوتی ساخت. این نگرانی بی‌جا نیست؛ بعداً روشن خواهد شد که در مسأله‌هایی که احتمالاً به زمان اجرایی نمایی نیاز دارند، می‌توان درخت‌های تصمیم‌گیری‌ای با ارتفاع چندجمله‌ای - اما با اندازه‌ی نمایی - ساخت. بنابراین، گاهی درخت‌های تصمیم‌گیری بیش از حد خوش‌بینانه هستند؛ یعنی شاید حد پایین یک درخت تصمیم‌گیری، بسیار کم‌تر از پیچیدگی واقعی مسأله باشد. از سوی دیگر، اگر حد پایین دسته‌ای از الگوریتم‌ها (چنان که در مسأله‌ی مرتب‌سازی دیدیم) با حد بالای یک الگوریتم مشخص از آن دسته برابر گردد، آنگاه از حد پایین نتیجه می‌شود که حتا اگر حافظه‌ی فراوانی را برای این الگوریتم خاص به کار گیریم، زمان اجرای الگوریتم بیش از این بهبودپذیر نیست.

جالب است که بدانیم زمان اجرای هر الگوریتم مرتب‌سازیِ مبتنی بر مقایسه، در حالت میانگین هم از $\Omega(n \log n)$ است. اثبات این مطلب را (که پیچیده است) در اینجا نمی‌آوریم (برای نمونه Aho، Hopcroft و Ullman [۱۹۷۴] را ببینید).

۶-۵ مرتبه‌ی آماری

فرض کنید $S = x_1, x_2, ..., x_n$ دنباله‌ای از عناصر باشد. رتبه‌ی x_i را k گوییم، اگر x_i، kامین عنصر از نظر کوچکی باشد. به آسانی با مرتب کردن عناصر می‌توانیم رتبه‌ی همه‌ی آن‌ها را به صورت یک دنباله مشخص کنیم. پرسش‌های زیادی درباره‌ی رتبه مطرح است که پاسخ آن‌ها را بدون مرتب‌سازی هم می‌توان یافت. در این بخش از کتاب، با چنین پرسش‌هایی سر و کار داریم. کار را با مسأله‌ی یافتن بیشینه و کمینه‌ی عناصر آغاز می‌کنیم. سپس به بررسی مسأله‌ی عمومی «پیدا کردن kامین عنصر از نظر کوچکی» می‌پردازیم. (در این بخش به kامین عنصر از نظر کوچکی «آماری kام» و به kامین عنصر از نظر بزرگی، «آماری معکوس kام» گفته‌ایم - مترجمان)

۶-۵-۱ بیشینه و کمینه‌ی عناصر

یافتن عنصر بیشینه یا عنصر کمینه‌ی یک دنباله، عمل ساده‌ای است. اگر بیشینه‌ی دنباله‌ای به اندازه‌ی n-1 را بشناسیم، برای یافتن بیشینه‌ی دنباله‌ای به اندازه‌ی n، کافی است این عنصر بیشینه را با عنصر nام مقایسه کنیم (یافتن بیشینه‌ی دنباله‌ای به اندازه‌ی 1 نیز روشن است). از عنصر دوم به بعد، این فرایند به یک مقایسه به ازای هر عنصر نیازمند است؛ از این رو، تعداد مقایسه‌ها n-1 خواهد بود.

حال، فرض کنید می‌خواهیم هر دو عنصر بیشینه و کمینه را بیابیم:

مسأله: عنصر بیشینه‌ی یک دنباله‌ی داده‌شده را به همراه عنصر کمینه‌ی آن بیابید.

راه‌حل سرراست، حل جداگانه‌ی هر دو مسأله است. تعداد کل مقایسه‌ها 2n-3 خواهد بود: n-1 مقایسه برای یافتن عنصر بیشینه و n-2 مقایسه برای یافتن عنصر کمینه (چراکه در مورد کمینه، دیگر نیازی به در نظر گرفتن عنصر بیشینه نیست). آیا انجام این کار در زمان کم‌تری هم ممکن است؟ بازهم رویکرد استقرایی را در نظر بگیرید. فرض کنید چگونگی حل مسأله را برای n-1 عنصر می‌دانیم و حال می‌خواهیم حل آن را برای n عنصر بیابیم (حالت پایه روشن است). باید عنصر تازه را با عنصر بیشینه و عنصر کمینه‌ای که از پیش داریم، مقایسه کنیم. این کار به دو مقایسه نیاز دارد که از آن نتیجه می‌شود تعداد کل مقایسه‌ها بازهم 2n-3 خواهد بود، زیرا عنصر نخست به هیچ مقایسه‌ای نیاز ندارد و برای عنصر دوم نیز تنها یک مقایسه انجام می‌شود. با پویش عناصر در ترتیبی متفاوت هم، نمی‌توان این راه‌حل را بهبود بخشید، زیرا محل عناصر دنباله نقشی در این مسأله ندارد.

مسأله راه‌حل دیگری نیز دارد: هر بار راه‌حل را با بیش از یک عنصر گسترش دهیم. بیایید هر بار، دو عنصر به راه‌حل بیفزاییم. به عبارت دیگر، فرض می‌کنیم که راه‌حل مسأله را برای n-2 عنصر می‌دانیم و می‌کوشیم مسأله را برای n عنصر حل کنیم. (برای کامل کردن این رویکرد به دو حالت پایه‌ی n=1 و n=2 نیازمندیم تا با گسترش آن‌ها، همه‌ی اعداد طبیعی پوشش داده شوند.) x_{n-1} و x_n را در نظر بگیرید و بیشینه و کمینه‌ی n-2 عنصر نخست را به ترتیب در MAX و min قرار دهید. (بنا به فرض استقرا این دو عنصر را می‌شناسیم.) به آسانی می‌توان دید که برای یافتن بیشینه و کمینه‌ی تازه، تنها به سه مقایسه نیاز داریم. نخست x_{n-1} را با x_n، سپس عنصر بزرگ‌تر در بین این دو را با MAX و عنصر کوچک‌تر آن‌ها را با min مقایسه می‌کنیم. پس الگوریتمی داریم که در کل به جای 2n مقایسه، تقریباً به 3n/2 مقایسه نیاز دارد! آیا می‌توان با افزودن سه (یا چهار) عنصر در هر بار، کار را بهتر از این هم انجام داد؟ با پی‌گیری این روش درمی‌یابیم تعداد مقایسه‌ها تغییری نخواهد کرد. از هر شیوه‌ای هم که بهره گیریم، نمی‌توانیم تعداد مقایسه‌ها را برای این مسأله کاهش دهیم. جالب است بدانید که با رویکرد تقسیم‌وحل نیز به تعداد مقایسه‌هایی در حدود 3n/2 می‌رسیم (تمرین ۶-۱۴). (می‌توان ثابت کرد حل این مسأله، دست‌کم نیازمند $\lceil 3n/2 \rceil - 2$ مقایسه است - مترجمان)

۶-۵-۲ یافتن آماری kام یک دنباله

حال به حالت کلی مسأله می‌پردازیم:

مسأله: $S = x_1, x_2, \ldots, x_n$ همراه با عدد k در بازه‌ی [1,n] به شما داده شده است. آماری kامِ S را بیابید.

این مسأله، مرتبه‌ی آماری یا گزینش نام دارد. اگر k بسیار نزدیک به ۱ یا n باشد، آنگاه می‌توانیم آماری (معکوس) kام را با k بار اجرای الگوریتم یافتن عنصر کمینه (بیشینه) پیدا کنیم. این روش تقریباً به kn مقایسه نیاز دارد. انجام عمل مرتب‌سازی از این الگوریتم ابتدایی، بهتر است، مگر آن‌که k از $O(\log n)$ یا $n - O(\log n)$ باشد؛ اما الگوریتمی کارآمد هم وجود دارد که می‌تواند به ازای هر k، آماری kام را بیابد.

ایده‌ی کار بهره‌گیری از روش تقسیم‌وحل، شبیه الگوریتم مرتب‌سازی سریع است؛ اما در اینجا تنها حل یک زیرمسأله لازم است. در مرتب‌سازی سریع، دنباله را به کمک یک محور به دو زیردنباله، تقسیم و سپس هر دو زیردنباله را به صورت بازگشتی مرتب می‌کردیم. در اینجا، لازم است تنها مشخص کنیم کدام زیردنباله دربرگیرنده‌ی آماری kام است و سپس الگوریتم را تنها برای آن زیردنباله به صورت بازگشتی ادامه دهیم، زیرا می‌توان عناصر زیردنباله‌ی دیگر را نادیده گرفت. این الگوریتم در شکل ۶-۱۶ آمده است.

الگوریتم: Selection(X,n,k)

ورودی: X (آرایه‌ای با اندیس‌های ۱ تا n) و k (یک عدد صحیح)

خروجی: S (آماری kام؛ اثر جانبی: آرایه‌ی ورودی تغییر می‌کند.)

```
begin
    if (k<1) or (k>0) then print "خطا"
    else
        S := Select(1,n,k)
end

procedure Select(Left,Right,k);
begin
    if Left = Right then
      Select := Left
    else
        Partition(X,Left,Right); {شکل ۶-۹ را ببینید.}
        خروجی Partition را در Middle قرار بده;
        if Middle-Left +1 ≥ k then
          Select(Left,Middle,k)
        else
            Select (Middle+1,Right,k-(Middle-Left+1))
end
```

شکل ۶-۱۶ الگوریتم Selection

پیچیدگی: مانند مرتب‌سازی سریع، در اینجا هم، گزینش نامناسب محور به الگوریتمی از درجه‌ی دوم منجر می‌شود. از آنجا که در هر فراخوانی بازگشتی لازم است تنها یک زیرمسأله حل گردد، زمان اجرای

این الگوریتم از مرتب‌سازی سریع کم‌تر خواهد بود. میانگین تعداد مقایسه‌ها از $O(n)$ است و حتا می‌توان کاری کرد که در بدترین حالت هم، تعداد گام‌های پیدا کردن آماری kام از $O(n)$ باشد، اما این مطلب را در اینجا ثابت نمی‌کنیم. در عمل، الگوریتم شکل ۶-۱۶ کارآمدتر است.

توجه: بیش‌تر کاربردهای مرتبه‌ی آماری به یافتن میانه، یعنی n/2امین عنصر از نظر کوچکی نیاز دارند. الگوریتم Selection، الگوریتم بسیار خوبی برای یافتن میانه است. هیچ الگوریتم ساده‌تری وجود ندارد که صرفاً میانه را پیدا کند. به عبارت دیگر، گسترش مسأله‌ی یافتن میانه به یافتن هر آماری kام، الگوریتم را ساده‌تر هم می‌کند! این هم نمونه‌ی دیگری از تقویت فرض استقراست (چگونگی بازگشت وابسته به مقدار k است).

۶-۶ فشرده‌سازی داده‌ها

فشرده‌سازی داده‌ها ترفند جالب‌توجهی برای صرفه‌جویی در فضای ذخیره‌سازی است. فرض کنید پرونده‌ای داریم که در قالب رشته‌ای از کاراکترها ذخیره شده است. می‌خواهیم تا جایی که می‌توانیم این پرونده را فشرده کنیم، به گونه‌ای که از روی پرونده‌ی فشرده‌شده بتوان پرونده‌ی اصلی را بازسازی کرد. اگر مراجعه به پرونده نسبتاً کم باشد، فشرده‌سازی داده‌ها سودمند است. در این صورت، کاری که برای فشرده‌سازی و تبدیل پرونده‌ی فشرده‌شده به پرونده‌ی اصلی انجام می‌شود، با توجه به صرفه‌جویی در فضای ذخیره‌سازی توجیه‌پذیر است. اگر هزینه‌ی فرستادن اطلاعات از هزینه‌ی پردازش آن‌ها بیش‌تر باشد، این روش در برطرف کردن مشکلات ارتباطی نیز بااهمیت است. فشرده‌سازی داده‌ها کاربردهای دیگری هم دارد و زمینه‌ای بسیار گسترده است. در این بخش، تنها یک الگوریتم برای جنبه‌ی خاصی از فشرده‌سازی را مطرح می‌کنیم.

برای سادگی، فرض می‌کنیم پرونده، دنباله‌ای از حروف الفبای انگلیسی باشد. هر یک از این ۲۶ کاراکتر را با رشته‌ای یکتا از بیت‌ها نمایش می‌دهیم و این رشته را کد آن کاراکتر می‌خوانیم. اگر طول همه‌ی کدها (مانند بسیاری از کدهای استاندارد) یکسان باشد، تعداد کل بیت‌های پرونده، تنها به تعداد کاراکترهای پرونده وابسته خواهد بود. از سوی دیگر، می‌توان رشته‌های بیتی کوتاه‌تر را برای کاراکترهایی برگزید که بیش‌تر در پرونده ظاهر شده‌اند (مانند A) و رشته‌های بیتی بلندتر را برای کاراکترهایی برگزید که کم‌تر در پرونده آمده‌اند (مانند Z). برای مثال، در کد ASCII (کوتاه‌شده‌ی American Standard Code for Information Interchange به معنای کد استاندارد آمریکایی برای تبادل اطلاعات) همه‌ی کاراکترها با رشته‌هایی ۷ بیتی نمایش داده می‌شوند. رشته‌ی متناظر با حرف A، ۱۰۰۰۰۰۱ و رشته‌ی متناظر با B، ۱۰۰۰۰۱۰ و ... است. (در این روش کدگذاری، برای ۱۲۸ کاراکتر جا وجود دارد که حروف کوچک انگلیسی و کاراکترهای ویژه را نیز در بر می‌گیرد.) واژه‌ی «AND» (و البته هر واژه‌ی سه حرفی دیگر) به ۲۱ بیت فضا نیاز دارد. اگر نمایش A را مثلاً به ۱۰۰۱

تغییر دهیم، هر بار که A در پرونده ظاهر شود، ۳ بیت صرفه‌جویی کرده‌ایم. هر کدگذاری‌ای هم درست نیست، زیرا ممکن است موجب ابهام شود. برای نمونه، در حالی که کد M، ۱۰۰۱۱۰۱ است، نمی‌توانیم کد A را ۱۰۰۱ قرار دهیم؛ زیرا اگر با ۱۰۰۱ روبه‌رو شویم، نمی‌توانیم دریابیم با A برخورد کرده‌ایم یا با بخش آغازین M. می‌توانستیم از جداکننده‌های ویژه‌ای نیز برای جدا کردن کاراکترها یاری گیریم، اما این کار پرونده را طولانی‌تر می‌کند. در حالت کلی، پیشوند کد هیچ کاراکتری نباید با کد کامل کاراکتر دیگری برابر باشد. این نیاز را محدودیت پیشوندی می‌نامیم. ممکن است برای کوتاه کردن یک کاراکتر ناچار شویم کد برخی دیگر را بلندتر کنیم. مسأله، یافتن بهترین موازنه است، با این فرض که بسامد یا دفعات تکرار کاراکترها را از پیش می‌دانیم.

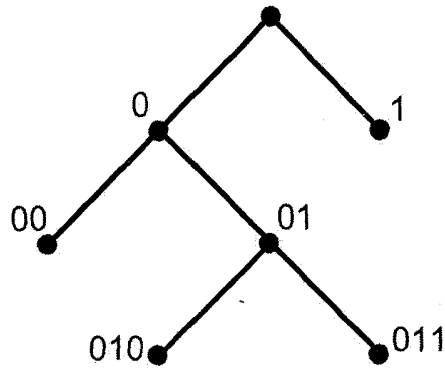
> **مسأله:** متنی (یعنی دنباله‌ای از کاراکترها) داده شده است. روشی برای کدگذاری آن بیابید که هم محدودیت پیشوندی را برآورد و هم برای کدگذاری متن، تعداد کل بیت‌های مورد نیاز را کمینه کند.

نخست باید تعداد دفعاتی را که هر کاراکتر در متن ظاهر شده است، محاسبه کنیم. به این مقدار، بسامد کاراکتر می‌گوییم. (در بسیاری موارد می‌توانیم به جای محاسبه‌ی جدول دقیق بسامد برای متن خود، جدول‌های استاندارد بسامد را که برای آن نوع متن از پیش محاسبه شده‌اند، به کار ببریم.) کاراکترها را با C_1، C_2، ... و C_n و بسامدهایشان را با f_1، f_2، ... و f_n نشان می‌دهیم. کدگذاری E که در آن رشته‌ی بیتی S_i به طول s_i بیانگر کاراکتر C_i است، به ما داده شده است. طول پرونده‌ی F، فشرده‌شده با کدگذاری E عبارت است از:

$$L(E,F) = \sum_{i=1}^{n} s_i . f_i$$

هدف ما یافتن یک کدگذاری مانند E است، به گونه‌ای که محدودیت پیشوندی را برآورده سازد و L(E,F) را نیز کمینه کند.

محدودیت پیشوندی برای جلوگیری از بروز ابهام در تبدیل متن کدگذاری‌شده به متن اصلی، ضروری است. بیایید نگاهی به یک روال کدگشایی بیندازیم. لازم است دنباله‌ی بیت‌ها را یکی‌یکی پویش کنیم تا به دنباله‌ای برسیم که با کد یکی از کاراکترها برابر باشد. یک درخت دودویی را در نظر بگیرید که یا هر گره آن، دو یال با برچسب‌های ۱ و ۰ دارد و یا بدون یال است. برگ‌های این درخت، متناظر با کاراکترها هستند و دنباله‌ای از ۰ و ۱ها نیز که مسیری از ریشه به یک برگ را مشخص می‌کند، متناظر با کد همان برگ است (شکل ۶-۱۷ را ببینید). محدودیت پیشوندی حکم می‌کند که هر کاراکتر، متناظر با یک برگ باشد. اگر یک پرونده‌ی کدگذاری‌شده را پویش کنیم و به یک برگ برسیم، با اطمینان می‌توانیم کاراکتر متناظر با آن را مشخص کنیم. هدف ما ساخت چنین درختی است که L(E,F) را کمینه کند. اگرچه برای حل مسأله هیچ نیازی به نمایش درختی نیست، اما داشتن توضیحی تصویری از مسأله (و محدودیت‌هایش) سودمند است.



شکل ۶-۱۷ نمایش درختی کدگذاری

این الگوریتم بر پایه‌ی کاهش مسأله‌ی n کاراکتری به مسأله‌ی n-1 کاراکتری بنا می‌شود (حالت پایه روشن است). طبق معمول، مشکل اصلی، چگونگی تعریف فرض استقرا و ترتیب حذف کاراکترهاست. (حذف کاراکترها برای کاهش اندازه‌ی مسأله انجام می‌شود - مترجمان) کاهشی که در اینجا صورت می‌گیرد با کاهش‌هایی که پیش‌تر دیدیم، متفاوت است. به جای آن‌که یک کاراکتر را از فرض استقرا حذف کنیم، یک کاراکتر «مصنوعی» به جای دو کاراکتر قرار می‌دهیم. این روش قدری پیچیده‌تر از روش‌های پیش است، اما همان هدف (یعنی کاهش اندازه‌ی ورودی) را برآورده می‌کند. C_i و C_j را دو کاراکتر با کم‌ترین بسامد در نظر بگیرید (اگر بیش از دو کاراکتر، این ویژگی را داشته باشند، آنگاه به دل‌خواه دو تا از آن‌ها را برمی‌گزینیم). ادعا می‌کنیم درختی وجود دارد که مقدار L(E,F) را کمینه می‌کند و در این درخت C_i و C_j با برگ‌هایی متناظر می‌شوند که فاصله‌ی آن‌ها تا ریشه بیش‌تر از دیگر برگ‌هاست؛ چراکه در غیر این صورت، اگر کاراکتری با بسامد بیش‌تر و جایگاه پایین‌تر در درخت وجود داشته باشد، می‌توان به کمک جابه‌جایی آن با C_i و C_j، مقدار L(E,F) را کاهش داد. (اگر بسامد این کاراکتر با بسامد C_i و C_j برابر باشد، می‌توان عمل جابه‌جایی را انجام داد بدون آن‌که L(E,F) تغییر کند.) از آنجا که هر گره درخت، یا دو فرزند دارد و یا بدون فرزند است (چراکه در غیر این صورت می‌توانستیم درخت را کوتاه‌تر کنیم) می‌توانیم C_i و C_j را با یکدیگر در نظر بگیریم. حال، کاراکتر تازه‌ای که آن را C_{ij} می‌نامیم و بسامدش هم برابر f_i+f_j است، جای‌گزین C_i و C_j می‌کنیم.

بدین ترتیب، تعداد کاراکترهای مسأله، n-1 (n-2 کاراکتر قدیمی و یک کاراکتر تازه) می‌شود و بنا به فرض استقرا مسأله حل‌شدنی است. مسأله‌ی اصلی با جای‌گزین کردن یک گره داخلی در مسأله‌ی کاهش‌یافته با دو برگ متناظر با C_i و C_j، در محلی که بیانگر C_{ij} است، حل می‌شود. اثبات بهینه بودن این راه‌حل را به عنوان تمرین به خواننده واگذار می‌کنیم. (برای درک بهتر مطلب به مثال ۶-۱ مراجعه کنید - مترجمان)

پیاده‌سازی: اعمالی که برای کدگذاری به این روش لازم است عبارتند از: (۱) درج در ساختمان داده، (۲) حذف دو کاراکتری از ساختمان داده که بسامد آن‌ها کمینه است و (۳) ساخت درخت. هرم برای دو

عمل نخست، ساختمان داده‌ای مناسب است که با یاری آن هر یک از این عمل‌ها در بدترین حالت در $O(\log n)$ گام انجام‌پذیر است. الگوریتم، در شکل ۶-۱۸ ارائه شده است. این روش کدگذاری، به یاد D. Huffman که این الگوریتم را پیش‌نهاد کرد، کدگذاری هافمن نامیده می‌شود. (D. Huffman [۱۹۵۲] را ببینید.)

الگوریتم: Huffman_Encoding(S,f)

ورودی: S (رشته‌ای کاراکتری) و f (آرایه‌ای از بسامدها)

خروجی: T (درخت هافمن برای S)

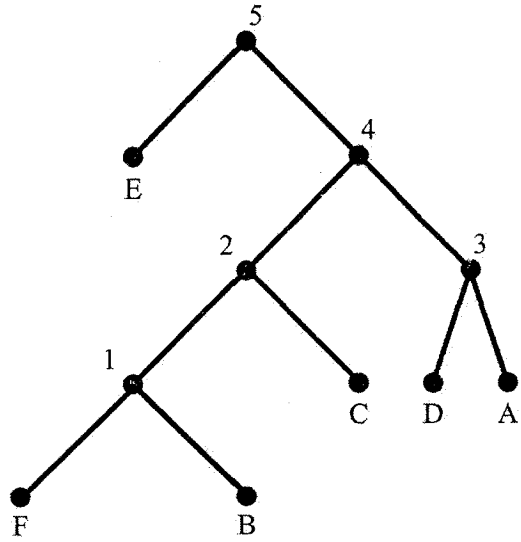
```
begin
    تمام کاراکترها را با توجه به بسامدشان به یک هرم بیفزا;
    while H تهی نیست do
        if H تنها یک کاراکتر مانند A دارد then
            A را ریشه‌ی T قرار بده
        else
            دو کاراکتر X و Y با کم‌ترین بسامد را پیدا کن و آن دو را از H حذف کن;
            کاراکتر تازه‌ی Z را که بسامدش برابر مجموع بسامدهای X و Y است،
            جای‌گزین X و Y کن {یعنی Z را به H اضافه کن.}
            {هنوز Z بدون والد است.} X و Y را فرزندان Z در T قرار بده
end
```

شکل ۶-۱۸ الگوریتم Huffman_Encoding

☐ مثال ۶-۱

فرض کنید کل داده‌ها شامل شش کاراکتر A، B، C، D، E و F، به ترتیب با بسامدهای ۵، ۲، ۳، ۴، ۱۰ و ۱ باشد. درخت هافمن برای این کاراکترها در شکل ۶-۱۹ نشان داده شده است. گره‌های داخلی، بنا به ترتیب ایجادشان شماره‌گذاری شده‌اند.

☐



شکل ۶-۱۹ درخت هافمن برای مثال ۶-۱

پیچیدگی: ایجاد هر گره برای ساخت درخت، نیازمند زمان ثابتی است. تعداد گام‌های هر درج و هر حذف نیز از $O(\log n)$ است، پس زمان اجرای کل الگوریتم از $O(n \log n)$ خواهد بود.

۶-۷ تطابق رشته‌ای

$A = a_1 a_2 ... a_n$ و $B = b_1 b_2 ... b_m$ را دو رشته‌ی کاراکتری در نظر بگیرید به گونه‌ای که $m \le n$ و فرض کنید کاراکترها اعضای مجموعه‌ای متناهی باشند. (با آن‌که لازم نیست اما برای راحتی، کاراکترها را حروف الفبای انگلیسی در نظر می‌گیریم.) یک زیررشته از A، دنباله‌ای متوالی از کاراکترهای $a_i a_{i+1} ... a_j$ از A است. زیررشته‌ی ویژه‌ی $a_1 a_2 ... a_i$ ($b_1 b_2 ... b_i$) را با A(i) (B(i)) نشان می‌دهیم.

> **مسأله:** دو رشته‌ی A و B داده شده‌اند. نخستین رخداد B را در A (در صورت وجود) بیابید. به عبارت دیگر، کوچک‌ترین k را بیابید به گونه‌ای که برای همه‌ی iهایی که در بازه‌ی [1,m] قرار دارند، داشته باشیم: $a_{k+i} = b_i$.

روشن‌ترین نمونه‌ی این مسأله، جست‌وجوی واژه یا الگوی مشخص در یک پرونده‌ی متنی است.[۱] همه‌ی برنامه‌های ویرایشگر متن باید دستوراتی برای یافتن الگو داشته باشند. این مسأله، در دیگر زمینه‌ها نیز کاربردهایی دارد – مثلاً در زیست‌شناسی مولکولی برای یافتن الگوهای مشخصی درون مولکول‌های بزرگ RNA یا DNA.

۱- دست‌کم برای من چنین است، چون الان دارم یک پرونده‌ی متنی را ویرایش می‌کنم.

در نگاه نخست، این مسأله، ساده به نظر می‌رسد: می‌توانیم با شروع از نخستین کاراکتر A که با b_1 مطابقت دارد و ادامه دادن کار (یعنی تطبیق دادن b_2 به بعد) تا هنگامی که یا تطبیق کامل شود یا به یک عدم تطبیق برسیم، عملیات را پی‌گیری کنیم. در صورت عدم تطبیق، ناچاریم به یک کاراکتر پس از جایی که آغاز کرده بودیم، برگردیم و عملیات را از نو شروع کنیم. این فرایند، در شکل ۶-۲۰ با مثالی که در این بخش چند بار به کار خواهد رفت، شرح داده شده است. در این مثال A=xyxxyxyxyyxyxyxyyxyxyxx و B=xyxyyxyxyxx است. نخستین عدم تطبیق در a_4 رخ می‌دهد، زیرا $b_4 \neq a_4$. پس از آن b_1 را با a_2 مقایسه می‌کنیم که در همان شروع به عدم تطبیق منجر می‌شود. سپس کار را از a_3 آغاز می‌کنیم که تطابق دارد، اما $a_4 \neq b_2$. تلاش بعدی امیدوارکننده‌تر است: از a_4 تا a_7 تطابق دارد، اما در a_8 چنین نیست. اینک، لازم است چندین گام به عقب برگردیم و b_1 را با a_5 مقایسه کنیم (که با عدم تطبیق روبه‌رو می‌شویم) سپس b_1 را با a_6 مقایسه می‌کنیم و به همین ترتیب کار را ادامه می‌دهیم. سرانجام، به یک تطبیق می‌رسیم که از a_{13} شروع می‌شود. ممکن است چندین و چند بار ناچار به عقب‌گرد و مقایسه‌ی دوباره شویم که در نتیجه، تعداد مقایسه‌ها در بدترین حالت از $O(mn)$ خواهد شد. دقت کنید که بخش بزرگی از کار، زائد است. برای نمونه، دوبار زیرالگوی xyxy را با شروع از a_{11} در A می‌یابیم (خط ۶ و ۱۱). هنگام یافتن یک واژه در پرونده‌ای متنی، تعداد گام‌های عقب‌گرد بسیار اندک است، زیرا بیش‌تر وقت‌ها فوراً عدم تطبیق رخ می‌دهد. الگوریتم ساده‌ای که گفتیم، برای چنین کاربردهایی نسبتاً خوب است. در دیگر موارد که الفبا کوچک است و الگوها زیاد تکرار می‌شوند، احتمالاً تعداد گام‌های عقب‌گرد زیاد خواهد شد و ممکن است الگوریتم گفته‌شده زیرالگویی را چندین و چند بار با جای یکسانی از متن مقایسه کند، اما علاقه‌مندیم به الگوریتمی برسیم که از چنین تکرارهایی پرهیز کند. مسأله، نظم دادن به اطلاعاتی است که هنگام اجرای الگوریتم به دست آورده‌ایم، به گونه‌ای که بعداً هنگام رویارویی با همان تطبیق‌ها در جاهای دیگر بتوانیم به صورتی کارآمد، آن اطلاعات را به کار ببریم.

A=xyxxyxyxyyxyxyxyyxyxyxx B=xyxyyxyxyxx

•	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	x	y	x	x	y	x	y	x	y	y	x	y	x	y	x	y	y	x	y	x	y	x	x
1:	x	y	x	y	.	.	.																
2:		x	.	.	.																		
3:			x	y	.	.	.																
4:				x	y	x	y	y	.	.	.												
5:					x	.	.	.															
6:						x	y	x	y	y	x	y	x	y	x	x	.	.	.				
7:							x	.	.	.													
8:								x	y	x	.	.	.										
9:									x	.	.	.											
10:										x	.	.	.										
11:											x	y	x	y	y	.	.	.					
12:												x	.	.	.								
13:													x	y	x	y	y	x	y	x	y	x	x

شکل ۶-۲۰ نمونه‌ای از تطبیق ساده و سرراست رشته

برای بهبود این الگوریتم سرراست، نخست باید بفهمیم که ناکارآمدی آن از کجاها سرچشمه می‌گیرد. حالت بدی که درباره‌اش بحث کردیم، از عقب‌گرد سرچشمه می‌گرفت. یک حالت بد ویژه، هنگامی رخ می‌دهد که الگو، yyyyyx و متن، yyyyyyyyyyyyyx باشد. پنج yیی را که در الگو قرار دارند، با متن مقایسه می‌کنیم تا این‌که در الگو به x می‌رسیم و با عدم تطبیق روبه‌رو می‌شویم، در نتیجه یک گام به سمت راست می‌رویم و چهار بار دیگر مقایسه‌هایی را انجام می‌دهیم که پیش‌تر نیز انجام داده بودیم. (برطرف کردن مشکل برای این حالت ساده، آسان است اما به یاری آن، مشکل حالت کلی را نشان داده‌ایم.) از سوی دیگر، الگوی xyyyyy را در نظر بگیرید. برای تطبیق این الگو در متن، به دنبال رخدادی از x می‌گردیم که پس از آن پنج y آمده باشد. اگر تعداد این yها کافی نباشد، نیازی به عقب‌گرد نیست. در این مورد، باید x بعدی را بیابیم؛ زیرا حتا تطبیق تمام yها هیچ فایده‌ای ندارد. الگوریتم سرراستی که متناسب با الگوی xyyyyy طرح‌ریزی شود، در زمانی خطی اجرا خواهد شد، زیرا هیچ نیازی به عقب‌گرد ندارد.

بیایید به الگوی پیشین B=xyxyyxyxyxx بازگردیم. فرض کنید هنگام پویش پنجمین کاراکتر B، یک عدم تطبیق رخ داده باشد (مانند هنگامی که در خط ۴ از شکل ۶-۲۰، a_8 با این کاراکتر مقایسه شد). دو کاراکتر پیشین A، باید xy باشد (زیرا تطبیق انجام شده است) اما از سویی، xy دو کاراکتر نخست B نیز هست. حال، می‌خواهیم B را به سمت راست «بلغزانیم» تا کاراکتر فعلی A را با کاراکتری در میانه‌های B مقایسه کنیم (البته با در نظر گرفتن تطبیق‌های پیشین). دوست داریم (برای صرفه‌جویی در تعداد مقایسه‌ها) B را هر چه بیش‌تر به سمت راست بلغزانیم، بدون آن‌که تطبیق‌های احتمالی را جا بگذاریم. در این مورد، می‌توانیم B را دو گام به سمت راست بلغزانیم. عمل تطبیق را به کمک مقایسه‌ی b_3 با همان کاراکتری از A که سبب عدم تطبیق شد (در این مثال a_8) ادامه می‌دهیم، زیرا از پیش می‌دانیم که b_1 و b_2 تطبیق یافته‌اند (در واقع، همان کاری را انجام دادیم که قرار بود بعداً در خط ۶ از شکل ۶-۲۰ انجام دهیم، با این تفاوت که دیگر مجبور نیستیم سه مقایسه‌ی تکراری، یعنی x در خط ۵ و xy در آغاز خط ۶ را انجام دهیم.) توجه کنید که کل این بحث کاملاً مستقل از A است! تنها چند کاراکتر انگشت‌شماری از A را که آخر همه بررسی شده‌اند - چون پیش از این با B تطبیق یافته‌اند - می‌شناسیم.

در بحثی که در اینجا مطرح می‌شود فرض بر این نیست که در متن (و الگو) تنها دو نوع کاراکتر وجود دارد، ولی برای سادگی مثال‌ها، الفبای آن‌ها را دو کاراکتری در نظر گرفته‌ایم. می‌توان در این مورد، الگوریتم را بیش از این هم بهبود بخشید (موضوع تمرین ۶-۴۵ نیز همین است).

بیایید با پی‌گیری تطبیق پیش، مورد دیگری را بررسی کنیم. عدم تطبیق در خط ۶ از شکل ۶-۲۰ در آخرین کاراکتر B، یعنی b_{11} رخ می‌دهد. در اینجا می‌توانیم لغزاندن را بیش‌تر هم ادامه دهیم. زیرالگوی $B(10) = b_1 b_2 ... b_{10}$ را در نظر بگیرید. می‌دانیم B(10) دقیقاً مطابق ۱۰ کاراکتر از بخش‌های نزدیک به آغاز A است؛ یعنی: $B(10) = A[6..15]$. می‌خواهیم به صورت دقیق مشخص

کنیم B چند گام دیگر باید به راست جابه‌جا شود تا به تطبیق دیگری برسیم. برای یافتن تعداد این گام‌ها، به دنبال پسوند بیشینه‌ای در B(10) می‌گردیم که با پیشوندی از B برابر باشد. در این مورد، همان‌گونه که در شکل ۶-۲۱ آمده، طول چنین پسوندی ۳ است (یعنی پسوند xyx). در این شکل، B(10) هر بار یک گام جابه‌جا می‌شود و با خودش مقایسه می‌گردد تا آن‌که یک پیشوند با یک پسوند مطابقت یابد (علت نادیده گرفتن آخرین کاراکتر، یعنی b_{11}، رخ دادن عدم تطبیق در همین کاراکتر است). از آنجا که می‌دانیم $B[1..3] = B[8..10]$، می‌توانیم کار را با مقایسه‌ی a_{16} با b_4 و ... ادامه دهیم تا هنگامی‌که به تطبیق کامل برسیم. بدین ترتیب، تمام مقایسه‌های خط‌های ۷ تا ۱۲ و نیمی از مقایسه‌های خط ۱۳ را انجام ندادیم. تفاوت شکل ۶-۲۰ و شکل ۶-۲۱ در این است که اطلاعات شکل ۶-۲۱ تنها به B بستگی دارد و همین تفاوت، نکته مهمی است، زیرا نشان می‌دهد که می‌توانیم یک بار B را پیش‌پردازش کنیم و اطلاعات مربوط به آن را بدون توجه به A به دست آوریم. پس از پیش‌پردازش B می‌توانیم از تطبیق‌های انجام‌شده در خط ۶ از شکل ۶-۲۰ سود ببریم؛ چراکه دیگر هیچ یک از آن‌ها تکرار نخواهند شد.

```
B=  x  y  x  y  y  x  y  x  y  x  x
       x  .  .  .
          x  y  x  .  .  .
             x  .  .  .
                x  .  .  .
                   x  y  x  y  y
                      x  .  .  .
                         x  y  x
```

شکل ۶-۲۱ تطبیق یک رشته با خودش

اساس الگوریتم بهبودیافته، پیش‌پردازش B است. تمام الگوهای تکراری B را بررسی می‌کنیم و برای موارد عدم تطبیق تدبیری می‌اندیشیم که ناچار به عقب‌گرد نشویم؛ به این ترتیب، رشته‌ی A همواره رو به جلو پویش می‌شود و هیچ عقب‌گردی در آن صورت نمی‌گیرد، اگرچه ممکن است گاهی یک کاراکتر از آن (هنگام بروز عدم تطبیق) با چندین کاراکتر از B مقایسه گردد. هنگام رویارویی با یک عدم تطبیق، برای دانستن مقدار عقب‌گرد در B، به یک جدول (به نام next) مراجعه می‌کنیم. در این جدول برای هر کاراکتر B، خانه‌ای وجود دارد که نشان می‌دهد در صورت عدم تطبیق در آن کاراکتر، چه مقدار عقب‌گرد (یا چه تعداد جابه‌جایی) باید انجام گیرد. به زودی، روشی کارآمد برای ساخت این جدول ارائه خواهیم کرد. نخست، جدول را به طور دقیق تعریف می‌کنیم و توضیح می‌دهیم که چگونه باید آن را برای مسأله تطابق رشته به کار ببریم.

بیایید ایده‌ی ساخت جدول next را روشن‌تر کنیم. می‌خواهیم برای هر b_i، بزرگ‌ترین پسوند B(i-1) را چنان بیابیم که با پیشوندی از B(i-1) برابر باشد. اگر طول این پسوند j شود، آنگاه می‌توانیم

کاراکتری از A را که دچار عدم تطبیق شده است (با چشم‌پوشی از چند کاراکتر) یک‌راست با b_{j+1} تطبیق دهیم. آخرین j کاراکتری از A را که با ابتدای B تطبیق دارند، از پیش می‌شناسیم؛ در ضمن، از آنجا که پسوند گفته‌شده در بین همه‌ی پسوندهایی که با یک پیشوند برابر بوده‌اند، بزرگ‌ترین است؛ دیگر B با تعداد بیش‌تری کاراکتر از سمت چپ A تطبیق‌پذیر نخواهد بود. پس جدول next را چنین تعریف می‌کنیم:

next(i)= یا بیش‌ترین j در بازه‌ی (0,i-1) که به ازای آن $B(j)=b_{i-j}\ b_{i-j+1}\ldots b_{i-1}$ و یا عدد ۰، اگر چنین j‌یی وجود نداشته باشد.

برای راحتی کار next(1) را ۱- تعریف می‌کنیم تا از بقیه متمایز گردد. روشن است که همیشه next(2)، صفر خواهد بود (زیرا هیچ j‌یی نمی‌تواند در شرط 1-2>j>0 صادق باشد). در شکل ۶-۲۲، جدول next برای الگوی B از شکل ۶-۲۱ نشان داده شده است. اگرچه می‌توان مقدارهای جدول next را با پیروی کورکورانه از روش شکل ۶-۲۲ حساب کرد، اما روش هوشمندانه‌ای نیز برای محاسبه‌ی آن‌ها در زمانی از O(m) وجود دارد. بیایید نخست فرض کنیم مقدارهای next را به ما داده‌اند و ما می‌خواهیم چگونگی انجام تطبیق بر اساس این مقدارها را ببینیم. اندکی بعد به چگونگی محاسبه‌ی next نیز خواهیم پرداخت.

i=	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱
B=	x	y	x	y	y	x	y	x	y	x	x
next=	-۱	۰	۰	۱	۲	۰	۱	۲	۳	۴	۳

شکل ۶-۲۲ مقدارهای next

تطبیق به این ترتیب پیش می‌رود: کاراکترهای A با کاراکترهای B مقایسه می‌شوند تا آن‌که عدم تطبیق در کاراکتری از B رخ دهد. در این کاراکتر از B، مثلاً b_i، به جدول next نگاه کرده، سپس کاراکتر فعلی A را با $b_{next(i)+1}$ مقایسه می‌کنیم (چراکه نخستین next(i) کاراکتر از پیش تطبیق یافته‌اند). اگر بازهم عدم تطبیق رخ دهد، مقایسه‌ی بعدی را با $b_{next(next(i)+1)+1}$ انجام می‌دهیم و به همین ترتیب، کار را دنبال می‌کنیم. تنها استثنای این قاعد زمانی است که عدم تطبیق در b_1 رخ دهد؛ در این حالت، باید در A پیش برویم. مقدار ویژه‌ی next(1) (یعنی ۱-) برای مشخص کردن همین حالت است. برنامه‌ی تطابق رشته‌ای در شکل ۶-۲۳ ارائه شده است.

الگوریتم: String_Match(A,n,B,m)

ورودی: A (رشته‌ای به طول n) و B (رشته‌ای به طول m)

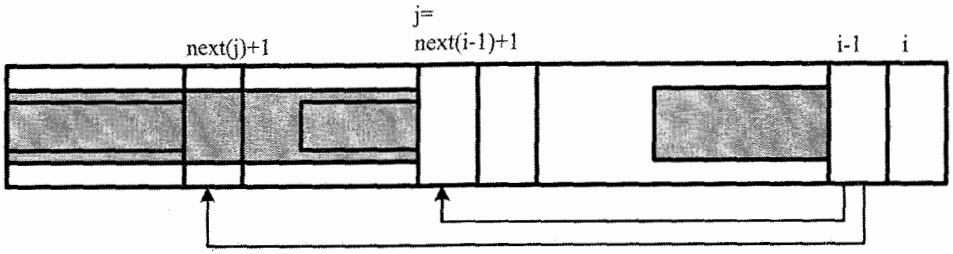
{فرض می‌کنیم جدول next از پیش داده شده است؛ شکل ۶-۲۵ را ببینید.}

خروجی: Start (اندیس آغاز نخستین رخداد B در A)

```
begin
    j := 1; i := 1;
    Start := 0 ;
    while Start = 0 and i ≤ n do
        if B[j] = A[i] then
            j := j + 1;
            i := i + 1;
        else
            j := next[j] + 1;
            if j = 0 then
              j := 1;
              i := i + 1;
        if j = m+1 then Start := i - m
end
```

شکل ۶-۲۳ الگوریتم String_Match

آنچه ناگفته ماند، الگوریتمی برای محاسبه مقدارهای جدول next است. از استقرا کمک می‌گیریم: چنان که گفته شد، برای حالت پایه، next(2) را صفر می‌گیریم. فرض می‌کنیم next را برای ۱، ۲، ... و i-1 محاسبه کرده‌ایم؛ حال next(i) را در نظر می‌گیریم. در بهترین حالت، هنگامی که $b_{i-1}=b_{next(i-1)+1}$، next(i) برابر next(i-1)+1 خواهد شد. به عبارت دیگر، می‌توان به انتهای پسوند بیشینه‌ی یافته‌شده‌ی مورد نظر (یعنی پسوند بیشینه‌ای از B(i-2) که با پیشوندی از B برابر است - مترجمان) b_{i-1} را نیز افزود. این حالت، دشوار نیست؛ اما اگر b_{i-1} برابر $b_{next(i-1)+1}$ نباشد، کار دشوار می‌شود. لازم است پسوند تازه‌ای بیابیم که با یک پیشوند برابر گردد. می‌دانیم چگونه باید بزرگ‌ترین پسوند B(i-2) را تطبیق دهیم؛ این پسوند با $b_1b_2...b_{next(i-1)}$ تطبیق می‌یابد (شکل ۶-۲۴ را ببینید) اما $b_{i-1} \neq b_{next(i-1)+1}$، دقیقاً مانند یک عدم تطبیق معمولی در $b_{next(i-1)+1}$ است! پس راه برخورد با آن را می‌دانیم. اگر در اندیسی مانند j، یک عدم تطبیق رخ دهد، به next(j) می‌رویم؛ پس هنگام عدم تطبیق در اندیس next(i-1)+1 باید به next(next(i-1)+1) برویم؛ یعنی بکوشیم b_{i-1} را با $b_{next(next(i-1)+1)+1}$ تطبیق دهیم. هنگام تطبیق، مقدار next(next(i-1)+1)+1 را در next(i) قرار می‌دهیم؛ ولی هنگام عدم تطبیق، کار را به همین روش آن قدر ادامه می‌دهیم تا یا به یک تطبیق کامل و یا به آغاز الگو برسیم.



شکل ۶-۲۴ محاسبه‌ی next(i)

☐ مثال ۶-۲

B را xyxyyyxyxyxx (مانند شکل ۶-۲۱) در نظر بگیرید و به next(11) توجه کنید. نخست به next(10) نگاه می‌کنیم و می‌بینیم که مقدارش ۴ است؛ سپس b_{10} را با b_5 مقایسه می‌کنیم. اگر یکسان باشند، طول بزرگ‌ترین پیشوندی که با یک پسوند برابر است، ۵ خواهد بود؛ اما b_{10} با b_5 یکسان نیست. پس، در b_5 یک عدم تطبیق داریم و به next(5) توجه می‌کنیم که مقدارش ۲ است. حال b_{10} را با b_3 مقایسه می‌کنیم و می‌بینیم که یکسان هستند. از این رو next(11)=3 و به آسانی می‌توان درستی این تساوی را به روش دستی هم بررسی کرد.

☐

هرچند درک الگوریتم محاسبه‌ی جدول next آسان نیست، اما پیاده‌سازی آن بسیار ساده است. این برنامه را می‌توانید در شکل ۶-۲۵ ببینید.

الگوریتم: Compute_Next(B,m)

ورودی: B (رشته‌ای به طول m)

خروجی: next (آرایه‌ای با اندازه‌ی m)

```
begin
    next(1) := -1;
    next(2) := 0;
    for i := 3 to m do
        j := next(i-1) + 1
        while b_{i-1} ≠ b_j and j > 0 do
            j := next(j) + 1;
        next(i) := j
end
```

شکل ۶-۲۵ الگوریتم Compute_Next

پیچیدگی: ممکن است یک کاراکتر از A با کاراکترهای زیادی از B مقایسه شود. اگر عدم تطبیق رخ دهد، آنگاه همان کاراکتر از A با کاراکتری از B – که جدول next آن را مشخص کرده است – مقایسه می‌شود. اگر بازهم عدم تطبیق رخ دهد، به مقایسه‌ی آن کاراکتر از A آن قدر ادامه می‌دهیم تا یا به یک تطبیق برسیم، یا به ابتدای B. با این حال، ادعا می‌کنیم زمان اجرای الگوریتم هنوز هم از O(n)

است. برای هر کاراکتر A، مثلاً a_i تا چند بار ممکن است عقب‌گرد کنیم؟ فرض کنید نخستین عدم تطبیق در b_k باشد. از آنجا که با هر عقب‌گرد به اندیسی کوچک‌تر در B می‌رسیم، پس حداکثر k عقب‌گرد ممکن است. به هر حال، برای رسیدن به b_k بدون عقب‌گرد، بایستی k بار به جلو رفته باشیم! اگر هزینه‌های حرکات رو به جلو را دو برابر در نظر بگیریم، خیالمان راحت است که هزینه‌ی کل (یعنی هزینه‌ی حرکت‌های رو به جلو همراه با هزینه‌ی عقب‌گردها) از مجموع هزینه‌های تازه‌ی حرکات رو به جلو بیش‌تر نخواهد بود. از سویی، دقیقاً n حرکت رو به جلو وجود دارد؛ پس تعداد مقایسه‌ها از O(n) خواهد بود.

از آنجا که Kunth، Moris و Pratt [۱۹۷۷] این الگوریتم را ابداع کرده‌اند، آن را الگوریتم KMP نام‌گذاری کرده‌اند. Boyer و Moore [۱۹۷۷] نیز الگوریتم سریعی برای این مسأله طراحی کرده‌اند که نگاه کوتاهی هم به آن می‌اندازیم. تفاوت دو الگوریتم در این است که الگوریتم Boyer-Moore، B را از انتها به ابتدا پویش می‌کند؛ پس، نخستین مقایسه بین b_m و a_m انجام می‌شود. اگر تطبیق رخ دهد، مقایسه‌ی بعدی بین b_{m-1} و a_{m-1} خواهد بود و ... . اگر با عدم تطبیق روبه‌رو شویم، تقریباً مانند الگوریتم پیش، اطلاعات موجود را به کار برده، کل الگو را به راست جابه‌جا می‌کنیم؛ مثلاً اگر داشته باشیم: a_m="Z" و Z اصلاً در B ظاهر نشده باشد، آنگاه می‌توان کل الگو را m گام به راست جابه‌جا کرد و مقایسه‌ی بعدی را بین a_{2m} و b_m انجام داد. اگر Z در B، مثلاً در b_i ظاهر شده باشد، آنگاه می‌توانیم عمل جابه‌جایی را m-i گام انجام دهیم. در صورت وجود تطبیق‌های جزئی، تصمیم‌گیری درباره‌ی مقدار جابه‌جایی پیچیده‌تر می‌گردد. از یک سو، می‌خواهیم تطبیق‌های پیداشده را بهینه کنیم؛ از سوی دیگر، جابه‌جایی بیش‌تر کل الگو، کارآمدتر خواهد بود، حتا اگر ناچار شویم برخی مقایسه‌ها را دو بار انجام دهیم. از جزئیات می‌گذریم. ویژگی جالب الگوریتم این است که احتمال دارد تعداد مقایسه‌هایش از n هم کم‌تر شود (البته در متن‌های معمولی)! چراکه گاهی از روی یک عدم تطبیق (بدون انجام هیچ مقایسه‌ی دیگری) می‌توانیم به اندازه‌ی m گام جابه‌جا شویم.

۶-۸ مقایسه‌ی دنباله‌ها

به تازگی، موضوع مقایسه‌ی دنباله‌ها بسیار مورد توجه قرار گرفته است. علت اصلی این توجه کاربردهایی است که این موضوع در حل مسائل زیست‌شناسی مولکولی دارد. ما در اینجا تنها روی یک مسأله تمرکز می‌کنیم: یافتن کم‌ترین گام‌های ویرایشی لازم برای تبدیل یک رشته به رشته‌ای دیگر. روش اصلی به‌کاررفته در اینجا برنامه‌نویسی پویاست (پیش‌تر در بخش ۵-۱۰ به این روش اشاره شده است).

$A = a_1 a_2 ... a_n$ و $B = b_1 b_2 ... b_m$ را دو رشته‌ی کاراکتری بگیرید که کاراکترهای آن‌ها از یک مجموعه‌ی متناهی گرفته شده‌اند (مثلاً از الفبای انگلیسی). علاقه‌مندیم A را کاراکتر به کاراکتر

چنان تغییر دهیم که به B تبدیل شود. سه نوع تغییر (یا گام ویرایشی) مجاز است و هزینه‌ی هر تغییر نیز ۱ در نظر گرفته می‌شود: (۱) درج؛ یعنی افزودن یک کاراکتر به رشته، (۲) حذف؛ یعنی کنار گذاشتن یک کاراکتر از رشته و (۳) جای‌گزینی؛ یعنی قرار دادن یک کاراکتر دیگر به جای کاراکتری از رشته. مثلاً برای تغییر رشته‌ی abbc به babb می‌توانیم نخست a را حذف کنیم تا bbc به دست آید، سپس یک a بین bها قرار دهیم (babc) و دست آخر به جای c، b بگذاریم تا با سه تغییر، کار مورد نظر انجام شود، اما برای انجام این کار راه دیگری نیز وجود دارد: یک b در ابتدای abbc قرار می‌دهیم (تا babbc حاصل شود) و سپس c را از آن حذف می‌کنیم تا با دو تغییر، کار مورد نظر انجام گردد. هدف ما کمینه ساختن تعداد این تغییرات تک کاراکتری است.

این مسأله، یعنی مسأله‌ی ویرایش رشته، کاربردهایی نیز در مقایسه‌ی پروندها و نگه‌داری نسخه‌های گوناگون آن‌ها دارد. ممکن است دو پرونده‌ی متنی (یا دو برنامه) داشته باشیم که یکی از آن‌ها تغییریافته‌ی دیگری باشد. یافتن اختلاف این دو پرونده دشوار نیست. اگر یک برنامه، چندین نسخه‌ی شبیه یکدیگر داشته باشد و بایگانی کردن همه‌ی این نسخه‌ها لازم باشد؛ راحت‌تر است که به جای ذخیره‌ی نسخه‌های بعدی، اختلاف بین آن‌ها با نسخه‌ی پیشین ذخیره گردد. در چنین مواردی امکان دارد تنها استفاده از اعمال درج و حذف را آزاد بگذاریم و یا شاید حتا برای گام‌های گوناگون ویرایشی، هزینه‌های متفاوتی در نظر بگیریم.

برای تبدیل یک رشته به رشته‌ای دیگر، روش‌های نسبتاً زیادی وجود دارد و به نظر می‌رسد که یافتن بهترین آن‌ها دشوار باشد. طبق معمول، از استقرا یاری می‌گیریم. زیررشته‌ی پیشوندی $a_1a_2...a_i$ را با A(i) نشان می‌دهیم (و $b_1b_2...b_i$ را با B(i)). مسأله همان تبدیل A(n) به B(m) با کم‌ترین گام‌های ویرایشی است. بنا بر استقرا فرض کنید بهترین راه تبدیل A(n-1) به B(m) را می‌شناسیم. (ممکن است راه‌حل بهینه یکتا نباشد؛ در این صورت، یافتن هر کدام از این راه‌حل‌ها برای ما کافی است.) با حذف a_n می‌توانیم A(n) را به B(m) تبدیل کنیم، اما ممکن است برای انجام کار روش بهتری نیز وجود داشته باشد. شاید بهتر باشد b_m را جای‌گزین a_n کنیم و حتا شاید a_n با b_m برابر باشد.

برای تبدیل بهینه‌ی A به B لازم است با یاری بهترین روش ممکن برای تبدیل زیردنباله‌های درون A به زیردنباله‌های درون B، همه‌ی حالت‌های گوناگون را شناسایی کنیم. C(i,j) را کم‌ترین هزینه‌ی تبدیل A(i) به B(j) بگیرید. حال، فرض کنید تنها به یافتن هزینه‌ی تبدیل A به B علاقه‌مند هستیم؛ نه خود روش تبدیل. می‌خواهیم رابطه‌ی بین C(n,m) را با C(i,j)هایی که i<n یا j<m بیابیم. در هر گام، یا ممکن است یکی از سه عمل ویرایشی انجام شود و یا ممکن است دو کاراکتر، یکسان باشند که در این صورت، انجام هیچ عملی لازم نیست؛ پس، روشن است که یکی از چهار حالت

صفحه‌ی بعد رخ می‌دهد:

حذف: اگر در تبدیل بهینه‌ی A به B، a_n حذف شده باشد، آنگاه روشی که پیش‌تر گفته شد، کارساز است؛ یعنی بهترین روش، تبدیل A(n-1) به B(m) و سپس حذف کاراکتر آخر است. به عبارت دیگر داریم: C(n,m)=C(n-1,m)+1.

درج: اگر در تبدیل بهینه‌ی A به B، کاراکتری برای تطبیق با B_m درج شده باشد، آنگاه C(n,m)=C(n,m-1)+1؛ یعنی تبدیل بهینه‌ی A(n) به B(m-1) را (به یاری استقرا) می‌یابیم و کاراکتری برابر با b_m در آن درج می‌کنیم.

جای‌گزینی: اگر در تبدیل بهینه‌ی A به B، a_n جای‌گزین b_m شده باشد، نخست باید تبدیل بهینه لازم برای تبدیل A(n-1) به B(m-1) را بیابیم و سپس اگر $a_n \neq b_m$، یک کاراکتر به آن بیفزاییم.

تطبیق: اگر a_n برابر با b_m باشد، داریم: C(n,m)=C(n-1,m-1). در این حالت c(i,j) را چنین تعریف می‌کنیم:

$$c(i,j)=\begin{cases}0 & ,a_i=b_j\\ 1 & ,a_i\neq b_j\end{cases}$$

این بحث را می‌توان در یک رابطه‌ی بازگشتی خلاصه کرد:

$$C(n,m)=\min\begin{cases}C(n-1,m)+1 & (\text{حذف } a_n)\\ C(n,m-1)+1 & (\text{درج } b_m)\\ C(n-1,m-1)+c(n,m) & (\text{جای‌گزینی یا تطبیق } a_n)\end{cases}$$

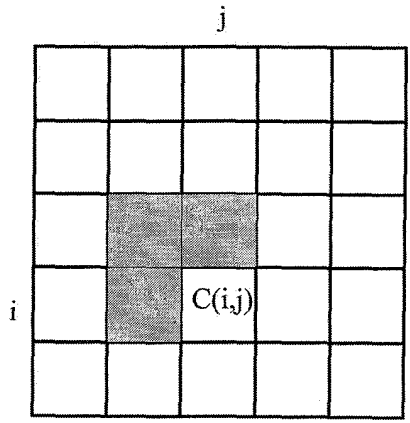
برای همه‌ی iهای بازه‌ی [0,n] و برای همه‌ی jهای بازه‌ی [0,m] داریم: C(i,0)=i و C(0,j)=j.

اثبات این مطلب که تنها همین حالت‌ها ممکن هستند، چندان دشوار نیست. a_n را در نظر بگیرید. به هر حال، کاری روی آن انجام می‌شود: یا حذف می‌شود (که طبق حالت نخست با آن رفتار می‌کنیم) یا به کاراکتری از B نگاشته می‌گردد که در این صورت یا a_n به b_m نگاشته می‌شود (که طبق حالت سوم یا چهارم با آن رفتار می‌کنیم) یا به کاراکتری پیش از b_m نگاشته می‌شود (که طبق حالت دوم باید چیزی پس از a_n درج گردد).

اشکال این روش، استفاده‌ی زیاد از استقراست! در این روش، مسأله‌ای با اندازه‌ی (n,m) را به مسأله‌هایی کاهش دادیم که اندازه‌ی آن‌ها تنها اندکی کوچک‌تر بود. اگر برای هر یک از این مسأله‌های کوچک‌تر روش بازگشتی را جداگانه به کار ببریم، در هر کاهش، مسأله به اندازه‌ی یک مقدار ثابت، کوچک می‌شود، در حالی که کار لازم سه برابر خواهد شد. به این ترتیب به الگوریتمی با زمان اجرای نمایی می‌رسیم. خوش‌بختانه در این مورد نیازی به حل جداگانه‌ی زیرمسأله‌ها نیست. نکته کلیدی، مشابه بودن بسیاری از زیرمسأله‌هاست. هر زیرمسأله، شامل محاسبه‌ی C(i,j) برای برخی از i و jهای محدوده‌ی $0 \le i \le n$ و $0 \le j \le m$ است. تعداد ترکیبات چنین i و jهایی، $n \times m$ می‌شود. پس

تعداد زیرمسأله‌های متفاوت از nm بیش‌تر نخواهد شد (و زیرمسأله‌های دیگر تکراری هستند) یعنی تنها حل nm زیرمسأله لازم است. پیش‌تر هم در مسأله‌ی کوله‌پشتی (بخش ۵-۱۰) با چنین موردی برخورد کرده بودیم (در اینجا نویسنده به اشتباه، خوانندگان را به بخش ۵-۱۱ ارجاع داده بود - مترجمان). برای حل این مسأله از استقرای قوی کمک می‌گیریم. به جای گسترش مسأله‌ی با اندازه‌ی n-1 به مسأله‌ی با اندازه‌ی n، از روی حل همه‌ی زیرمسأله‌های با اندازه‌ی کوچک‌تر از n، مسأله‌ی با اندازه‌ی n را حل می‌کنیم. از آنجا که این مسأله دوبعدی است، ناگزیر باید همه‌ی زیرمسأله‌هایی که اندازه‌ی آن‌ها از (n,m) کوچک‌تر است، به مسأله‌ای با اندازه‌ی (n,m) گسترش داده شوند. هرگاه نماد (n,m)> به کار رود، معنایش این است: «هر ترکیبی از (i,j) که دست‌کم یکی از آن دو از حد متناظر خودش کوچک‌تر بوده، دیگری نیز از حد متناظر خود بزرگ‌تر نباشد.»

هنگامی می‌توانیم استقرای قوی را به کار ببریم که حل همه‌ی زیرمسأله‌های کوچک‌تر را داشته باشیم. بنابراین، جدولی از نتیجه‌ی همه‌ی زیرمسأله‌ها می‌سازیم (به شکل ۶-۲۶ توجه کنید). برای محاسبه‌ی مقدار C(i,j) به سه مقدار دیگر نیاز داریم که در شکل، خانه‌های این سه مقدار تیره شده است. بهتر است این جدول را به گونه‌ای پویش کنیم که هنگام رسیدن به هر خانه، هر سه خانه‌ی مورد نیاز برای محاسبه‌ی آن خانه را از پیش دیده باشیم. در این مورد، پیمایش سطری (یعنی سطر به سطر از چپ به راست) روشی مناسب است. این روش، نمونه‌ای از برنامه‌نویسی پویاست.



شکل ۶-۲۶ وابستگی‌های C(i,j)

پیاده‌سازی: ماتریس دوبعدی C[1..n,1..m] را به کار می‌گیریم. هر خانه‌ی C[i,j] از ماتریس، مقدار C(i,j) را در خود نگه‌داری می‌کند. M[i,j] را آخرین حرکتی (تغییری) بگیرید که منجر به مقدار کمینه C[i,j] شده است. علت این که تنها به آخرین تغییر نیاز داریم، این است که به کمک آن می‌توانیم عقب‌گرد کرده، همه‌ی تغییرات انجام‌شده را از روی ماتریس بیابیم. این تغییر، یا delete(i) یا insert(j) و یا replace(i,j) است. برای محاسبه‌ی C[i,j] لازم است مقدار C[i-1,j]، C[i,j-1] و C[i-1,j-1] را

بدانیم. می‌توان بنا به هر یک از این سه حالتی که منجر به مقدار کمینه‌ی C[i,j] شده است، آخرین تغییر را مشخص کرد. الگوریتم در شکل ۶-۲۷ ارائه شده است.

پیچیدگی: بنا به برنامه‌ی شکل ۶-۲۷ روشن است که زمان اجرا از O(nm) خواهد بود. ضعف اصلی برنامه نیاز آن به فضایی از O(nm) است.

الگوریتم: Minimum_Edit_Distance(A,n,B,m)

ورودی: A (رشته‌ای به طول n) و B (رشته‌ای به طول m)

خروجی: C (ماتریس هزینه‌ها)

```
begin
    for i := 0 to n do C[i,0] := i;
    for j := 1 to m do C[0,j] := j;
    for i := 1 to n do
        for j := 1 to m do
            x := C[i-1,j] + 1;
            y := C[i,j-1] + 1;
            if a_i = b_i then
              z := C[i-1,j-1]
            else
                z := C[i-1, j-1] + 1;
            C[i,j] := min(x,y,z)
            {می‌توان M[i,j] را به طور مناسب تنظیم کرد.}
end
```

شکل ۶-۲۷ الگوریتم Minimum_Edit_Distance

توجه: اگر حل مسأله به حل تعداد زیادی زیرمسأله‌ی کمی کوچک‌تر وابسته باشد، برنامه‌نویسی پویا سودمند خواهد بود. در برنامه‌نویسی پویا بهره‌گیری از یک جدول برای نگه‌داری نتایج پیشین بسیار رایج است. معمولاً این جدول به ترتیبی مشخص (بیش‌تر، ترتیب سطری) پویش می‌شود و در نتیجه، زمان اجرا دست‌کم از درجه‌ی دو خواهد بود. بنابراین، کارایی شیوه‌ی برنامه‌نویسی پویا از روش‌هایی مانند روش تقسیم‌وحل کم‌تر است.

۶-۹ الگوریتم‌های مبتنی بر احتمال

تاکنون تنها درباره‌ی الگوریتم‌های مشخص و قطعی بحث کرده‌ایم؛ یعنی الگوریتم‌هایی که همه‌ی گام‌های آن‌ها از پیش روشن بوده است. اگر یک الگوریتم قطعی را دو (یا چند) بار روی یک ورودی مشخص به کار گیریم، تمامی اجراها و نتایجشان یکسان خواهند شد. الگوریتم‌های مبتنی بر احتمال

چنین نیستند. در این الگوریتم‌ها گام‌هایی وجود دارد که هم به ورودی و هم به نتایج برخی «رویدادهای تصادفی» وابسته‌اند. الگوریتم‌های مبتنی بر احتمال گونه‌های بسیاری دارند که دو تا از آن‌ها را بررسی خواهیم کرد. کار را با یک نمونه‌ی ساده، آغاز و سپس با یک روش رسمی‌تر پی‌گیری می‌کنیم.

فرض کنید مجموعه‌ای از اعداد x_1، x_2، ... و x_n داده شده است و می‌خواهیم یکی از آن‌ها را که به «نیمه‌ی بالا» تعلق داشته باشد، برگزینیم؛ یعنی می‌خواهیم عنصری را برگزینیم که بزرگ‌تر یا مساوی با میانه‌ی اعداد باشد. برای مثال، ممکن است بخواهیم برحسب نمره‌ی دانش‌جویان، یک دانش‌جوی خوب را برگزینیم. یک راه، گزینش بزرگ‌ترین عدد است (که بی‌شک در نیمه‌ی بالاست). پیش‌تر دیدیم که برای یافتن عنصر بیشینه به n-1 مقایسه نیازمندیم. یک روش دیگر، آن است که الگوریتم یافتن عنصر بیشینه را اجرا کنیم، اما درست پس از انجام نیمی از کار، آن را متوقف سازیم. عددی که از نیمی از اعداد بزرگ‌تر باشد، بی‌گمان به نیمه‌ی بالا تعلق دارد. این الگوریتم حدوداً به n/2 مقایسه نیاز دارد. آیا می‌توان کار را بهتر از این هم انجام داد؟ چندان دشوار نیست که ثابت کنیم با کم‌تر از n/2 مقایسه نمی‌توان مطمئن شد که عدد یافته‌شده به نیمه‌ی بالا تعلق دارد؛ پس به نظر می‌رسد که این الگوریتم بهینه است.

اگر بر «اطمینان از نتیجه» پافشاری کنیم، این الگوریتم بهینه است، اما در بسیاری موارد نیازی نیست که از نتیجه کاملاً مطمئن باشیم؛ تنها کافی است راه‌حل، با احتمالی مناسب، درست باشد. برای مثال، در مورد درهم‌سازی، نمی‌توانیم ضمانت کنیم که برخوردی رخ نخواهد داد، اما در صورت بروز آن می‌توانیم برایش چاره‌ای بیندیشیم. (چنان که در آینده نشان داده خواهد شد، درهم‌سازی را می‌توان الگوریتمی مبتنی بر احتمال پنداشت.) اگر نخواهیم ضمانت کنیم که عنصر یافته‌شده حتماً در نیمه‌ی بالا باشد، الگوریتم بهتری نیز وجود دارد: x_i و x_j را به طور تصادفی از بین اعداد در نظر می‌گیریم، به گونه‌ای که $i \neq j$. فرض می‌کنیم $x_i \geq x_j$. احتمال تعلق عددی، که به طور تصادفی برگزیده شده است، به نیمه‌ی بالای اعداد، دست‌کم 1/2 است (اگر شمار زیادی از اعداد با میانه برابر باشند، این احتمال از 1/2 هم بیش‌تر خواهد بود). پس احتمال این که x_i و x_j هیچ یک به نیمه‌ی بالا تعلق نداشته باشند، حداکثر 1/4 است. از طرفی چون $x_i \geq x_j$، این احتمال، با احتمال تعلق نداشتن x_i به نیمه‌ی بالا برابر خواهد بود. بنابراین، احتمال این که x_i متعلق به نیمه‌ی بالا باشد، دست‌کم 3/4 است.

معمولاً احتمال 3/4 برای درستی راه‌حل، ما را راضی نمی‌کند، اما می‌توان با به‌کارگیری روش پیش، راه‌حل را گسترش داد: k عدد را به طور تصادفی برگزینید و بزرگ‌ترین آن‌ها را انتخاب کنید. بنا به همان استدلال پیش، احتمال آن که بیشینه‌ی این k عنصر، به نیمه‌ی بالا تعلق داشته باشد، $1-2^{-k}$ خواهد بود. برای مثال، اگر k=10، احتمال موفقیت الگوریتم، ۰/۹۹۹ و اگر k=20، احتمال موفقیت الگوریتم ۰/۹۹۹۹۹۹ است. اگر k=100، معمولاً در عمل، می‌توانیم از احتمال اشتباه الگوریتم صرف‌نظر کنیم، چراکه احتمال خطا در برنامه‌نویسی یا سخت‌افزار و یا حتا بروز زلزله‌ای که سبب خطا گردد، از احتمال موفق نشدن الگوریتم بیش‌تر است. بدین ترتیب، الگوریتمی با احتمال درستی بسیار

زیاد برای گزینش عددی در نیمه‌ی بالا داریم که بدون توجه به اندازه‌ی ورودی، حداکثر باید ۱۰۰ مقایسه انجام دهد. (فرض می‌کنیم که گزینش تصادفی هر عنصر در یک عمل انجام‌پذیر باشد. کمی بعد، در بخش ۶-۹-۱ تولید اعداد تصادفی را بررسی خواهیم کرد.)

گاهی به این نوع الگوریتم‌ها، الگوریتم‌های Monte Carlo می‌گویند. احتمال این که نتیجه‌ی چنین الگوریتمی اشتباه باشد، بسیار ناچیز است، اما زمان اجرایش، احتمالاً از تمام الگوریتم‌های قطعی بهتر است. نوع دیگری از الگوریتم‌های مبتنی بر احتمال، آن‌هایی هستند که هرگز پاسخشان اشتباه نمی‌شود، اما زمان اجرای آن‌ها تضمین‌شده نیست؛ یعنی ممکن است گاهی خیلی زود به پایان برسند و گاهی نیز ممکن است اجرایشان مدت نامعلومی ادامه داشته باشد. این نوع از الگوریتم‌ها - که گاهی به آن‌ها الگوریتم‌های Las Vegas نیز می‌گویند – هنگامی سودمند هستند که زمان اجرای مورد انتظار از آن‌ها پایین باشد. در بخش ۶-۹-۲ یک الگوریتم Las Vegas نشان داده شده است که مسأله‌ی رنگ‌آمیزی مشخصی را حل می‌کند. در بخش ۶-۹-۳ روش بسیار جالبی برای تبدیل برخی الگوریتم‌های Las Vegas به الگوریتم‌های قطعی نشان خواهیم داد. از این روش بهره خواهیم گرفت و الگوریتمی قطعی و کارآمد برای مسأله‌ی رنگ‌آمیزی بخش ۶-۹-۲ به دست خواهیم آورد، اما با این ترفند نمی‌توان هر الگوریتم کارآمدی از نوع Las Vegas را به یک الگوریتم قطعی کارآمد تبدیل کرد.

ایده‌ی الگوریتم‌های مبتنی بر احتمال مستقیماً از روش‌های اثبات ریاضی الگوبرداری شده است. بهره جستن از احتمال برای اثبات خواص ترکیبیاتی، روشی نیرومند است. ایده‌ی اصلی این است که در بین مجموعه‌ای از اشیاء ثابت کنیم احتمال وجود یک شیء با ویژگی‌های مورد نظر از صفر بیش‌تر است که به صورت غیرمستقیم، وجود شیئی را با این ویژگی‌ها ثابت می‌کند. این شیوه، در الگوریتم‌ها به این صورت به کار گرفته می‌شود: فرض کنید در جست‌وجوی شیئی با خواصی مشخص هستیم و می‌دانیم اگر به صورت تصادفی شیئی را ایجاد کنیم، احتمال آن که شیء ایجادشده، ویژگی‌های مورد نظر را داشته باشد، بزرگ‌تر از صفر است. (اثبات مبتنی بر احتمال برای وجود یک شیء دل‌خواه، همین است.) در موارد مناسب، می‌کوشیم اثبات مبتنی بر احتمال را با تولید رخدادهای احتمالی پی‌گیری کنیم، سپس شیء مورد نظر را با احتمالی مثبت می‌یابیم. می‌توانیم این فرایند را بارها و بارها تکرار کنیم تا سرانجام موفق شویم. اگر احتمال یافتن شیء، مورد پسند ما باشد، یک الگوریتم از نوع Las Vegas با کارایی رضایت‌بخش به دست آورده‌ایم.

۶-۹-۱ اعداد تصادفی

در الگوریتم‌های مبتنی بر احتمال، لازم است اعدادی را به صورت تصادفی تولید کنیم. می‌خواهیم این کار به شیوه‌ای کارآمد انجام شود. هر روال قطعی، اعداد را به روشی ثابت (وابسته به گام‌های خود) تولید می‌کند. اگر این روش، کاملا قطعی باشد، دیگر، اعداد تولیدشده به معنای واقعی کلمه، تصادفی نیستند و

با یکدیگر رابطه‌ی مشخصی دارند؛ خوش‌بختانه، این موضوع مشکل عملی عمده‌ای نیست و در عمل، می‌توان از اعداد شبه‌تصادفی سود جست. یک روال قطعی، این اعداد را تولید می‌کند - بنابراین اعداد، واقعاً تصادفی نیستند - اما این کار را به گونه‌ای انجام می‌دهد که رابطه‌ی بین اعداد تولیدشده در بیش‌تر کاربردها مشکلی ایجاد نمی‌کند.

بحث ژرف‌تر درباره‌ی این موضوع، فراتر از حد این کتاب است. ما تنها شیوه‌ای بسیار کارآمد به نام «روش هم‌نهشتی خطی» را بررسی می‌کنیم. این شیوه برای تولید اعداد شبه‌تصادفی به کار می‌رود. گام نخست، برگزیدن عدد صحیح r(1) به عنوان هسته‌ی آغازین است که به طور تصادفی از محیط خارج الگوریتم گرفته می‌شود. (زمان جاری به میلیونیم ثانیه یا رکورد فعلی یک تیم ورزشی برای مقدار هسته مناسبند.) بقیه‌ی اعداد تصادفی با رابطه‌ی $r(i) = (r(i-1).b+1) \bmod t$ محاسبه می‌شوند که در آن، b و t اعدادی ثابت هستند. گزینش b و t باید با دقت بسیار انجام شود. Knuth [۱۹۸۱] پیروی از این ره‌نمود را سفارش می‌کند: t باید دست‌کم عددی میلیونی و در صورت امکان توانی از ۲ (یا ۱۰) باشد. b باید تقریباً یک رقم کم‌تر از t داشته باشد و نمایش ده‌دهی‌اش به x21 ختم شود که در آن، x عددی زوج است. این پیش‌نهادهای (عجیب) به گونه‌ای طرح شده‌اند که از بروز حالات بد جلوگیری کنند؛ حالت‌های بدی که سبب تکرار زیاد دنباله‌ای یکسان از اعداد می‌شوند. اعداد تولیدشده با روش هم‌نهشتی خطی، در محدوده‌ی ۰ تا t-1 قرار می‌گیرند. می‌توانیم با ضریبی مناسب، این محدوده را تغییر دهیم (در این حالت، t باید مضربی از محدوده‌ی مورد نظر باشد).

۶-۹-۲ یک مسأله‌ی رنگ‌آمیزی

S را مجموعه‌ای با n عنصر و S_1، S_2، ... و S_k را دسته‌ای از زیرمجموعه‌های متمایز آن بگیرید که هر یک از این زیرمجموعه‌ها دقیقاً r عنصر دارند و $k \le 2^{r-2}$.

> **مسأله:** هر یک از عناصر S را با یکی از دو رنگ قرمز و آبی رنگ کنید، چنان که هر S_i دربرگیرنده‌ی دست‌کم یک عنصر قرمز رنگ و یک عنصر آبی رنگ باشد.

هر رنگ‌آمیزیِ برآورنده‌ی این شرط، یک رنگ‌آمیزی معتبر خوانده می‌شود. مشخص خواهد شد که با شرایط داده‌شده برای زیرمجموعه‌ها، همواره رنگ‌آمیزی معتبری وجود دارد. ساده‌ترین الگوریتم مبتنی بر احتمال برای این کار از روی اثباتی برای وجود چنین رنگ‌آمیزی معتبری، الگوبرداری شده است (خودِ این اثبات نیز بر پایه‌ی احتمال است). الگوریتم چنین است:

عناصر S را یکی یکی، به صورت تصادفی، یعنی با احتمال 1/2 قرمز یا آبی کنید (بدون توجه به رنگ دیگر عناصر).

روشن است که رنگ‌آمیزی حاصل از این روش همیشه معتبر نیست. بیایید احتمال شکست (معتبر نبودن رنگ‌آمیزی) را محاسبه کنیم. احتمال این که همه‌ی عناصر S_i قرمز شوند، 2^{-r} است. احتمال این که همه‌ی عناصر دست‌کم یکی از k زیرمجموعه قرمز شود، از ۴ / ۱ بیش‌تر نیست (زیرا به علت محدودیت موجود بر روی k داریم: $k2^{-r} \leq 1/4$). ازاین رو، احتمال معتبر نبودن این رنگ‌آمیزی تصادفی، حداکثر ۲ / ۱ است (زیرا احتمال این که زیرمجموعه‌ای آبی شود، نیز وجود دارد و این احتمال هم حداکثر ۱/۴ است). بدین ترتیب ثابت شد که همواره رنگ‌آمیزی معتبری وجود خواهد داشت (وگرنه احتمال شکست ۱ می‌شد). همچنین از این بحث می‌توان نتیجه گرفت که این الگوریتم تصادفی بسیار مناسب است. به آسانی می‌توان اعتبار یک رنگ‌آمیزی مشخص را بررسی کرد: عناصر هر زیرمجموعه را بررسی می‌کنیم تا آن که به دو عنصر با رنگ‌های متفاوتی برسیم. احتمال موفقیت، چنان که گفته شد، ۲ / ۱ است. اگر موفق نشویم، رنگ‌آمیزی معتبر نبوده است؛ بنابراین رنگ‌آمیزی را دوباره انجام می‌دهیم. تعداد دفعات مورد انتظار برای اجرای الگوریتم تا رسیدن به یک رنگ‌آمیزی معتبر، ۲ بار است. پیداست که این الگوریتم از نوع Las Vegas است، زیرا بررسی رنگ‌آمیزی انجام‌شده را تکرار می‌کند تا آن که به یک رنگ‌آمیزی معتبر برسد. این الگوریتم، کاربرد ساده‌ای از شیوه‌ی مبتنی بر احتمال بود، اما بدبختانه، بیش‌تر الگوریتم‌های مبتنی بر احتمال به این سادگی نیستند. در بخش بعد، نشان می‌دهیم چگونه می‌توان این الگوریتم را به گونه‌ای تغییر داد که به طور قطعی، یک رنگ‌آمیزی معتبر بیابد.

۶-۹-۳ روشی برای تبدیل الگوریتم‌های مبتنی بر احتمال (یا احتمال‌گرا) به الگوریتم‌های قطعی[۲]

در این بخش نشان می‌دهیم که چگونه می‌توان با بهره‌گیری از استقرا، یک الگوریتم رنگ‌آمیزیِ مبتنی بر احتمال را به الگوریتمی قطعی تبدیل کرد. روشی که ارائه می‌کنیم برای همه‌ی الگوریتم‌های Las Vegas کارساز نیست. گمان هم نمی‌کنیم که بتوان همه الگوریتم‌های Las Vegas را به صورت کارآمد به الگوریتمی قطعی تبدیل کرد. این شیوه جالب است، زیرا از ایده‌ی تقویت فرض استقرا به شکلی نیرومند بهره می‌گیرد. الگوریتمی که در اینجا به دست می‌آید، نه تنها قطعی و کارآمد است، بلکه می‌تواند به شرط کنار گذاشتن برخی محدودیت‌های تحمیل‌شده به مسأله‌ی اصلی، یک مسأله‌ی عمومی‌تر را نیز حل کند.

بازهم، S را مجموعه‌ای n عنصری و S_1، S_2، ... و S_k را دسته‌ای از زیرمجموعه‌های متمایز آن در نظر بگیرید. این الگوریتم احتمال‌گرا بر این واقعیت استوار است که احتمال دست‌یابی به یک رنگ‌آمیزی معتبر، پس از رنگ‌آمیزی تصادفی عنصر، دست‌کم ۲ / ۱ است. فرض کنید می‌توانیم یک

۲- در نخستین بارِ خواندن کتاب، می‌توانید از این بخش چشم‌پوشی کنید.

عنصر را قرمز یا آبی کنیم، به گونه‌ای که پس از رنگ‌آمیزی تصادفی آن عنصر، احتمال دست‌یابی به یک رنگ‌آمیزی معتبر برای بقیه‌ی عناصر، صفر نباشد. با داشتن چنین فرضی و با استقرا روی n می‌توانیم به یک الگوریتم دست پیدا کنیم. اگر بتوانیم یک عنصر را چنان رنگ‌آمیزی کنیم که احتمال موفقیت همچنان مخالف صفر باقی بماند، پس با استقرا می‌توانیم همه‌ی عناصر را رنگ‌آمیزی کنیم.

از آنجا که می‌کوشیم هر بار، یک عنصر را رنگ‌آمیزی کنیم، پس باید فرض استقرا به گونه‌ای تقویت شود که لازم نباشد همه‌ی زیرمجموعه‌ها هم‌اندازه باشند. مهم‌ترین شرط آن است که احتمال موفقیت همچنان مخالف صفر باقی بماند. s_i را اندازه‌ی زیرمجموعه S_i بگیرید. احتمال آن که عناصر S_i هم‌رنگ باشند، 2^{-s_i+1} است. احتمال شکست (یعنی احتمال این که رنگ‌آمیزی تصادفی همه‌ی عناصر، یک رنگ‌آمیزی معتبر نباشد) از این عبارت بیش‌تر نیست:

$$F(n)=\sum_{i=1}^{k}2^{-s_i+1}$$

تابع احتمال F(n)، تابعی از اندازه‌ی زیرمجموعه‌هاست، اما برای راحتی، آن را به صورت تابعی از n نوشته‌ایم. تا زمانی که F(n)<1، زیر پایمان محکم و پایدار است. بیایید تا این فرض را برای استقرا بیازماییم:

فرض آزمایشی استقرا: می‌دانیم چگونه مجموعه‌ی S را که کم‌تر از n عنصر دارد، رنگ‌آمیزی کنیم؛ مشروط برآن که F(n)<1.

اگر یکی از زیرمجموعه‌ها تک‌عنصری باشد، در F(n) به اندازه‌ی عدد ۱ تأثیر و سهم دارد؛ در این صورت، F(n) کم‌تر از ۱ نخواهد شد. اگر n=2، با توجه به این که زیرمجموعه‌ها متمایز فرض شده بودند، تنها یک «زیرمجموعه‌ی دوعنصری» وجود دارد و ما می‌توانیم یک عنصر آن را آبی و عنصر دیگر را قرمز کنیم. از این رو، حالت پایه برقرار است. حال می‌کوشیم تا مسأله‌ی رنگ‌آمیزی n عنصر را به مسأله‌ی رنگ‌آمیزی n-1 عنصر کاهش دهیم.

x را عنصری دل‌خواه از S بگیرید. می‌توان آن را با یکی از دو رنگ آبی یا قرمز رنگ‌آمیزی کرد. فرض کنید x را آبی کرده‌ایم. احتمال آن که با رنگ‌آمیزی تصادفی بتوان بقیه‌ی n-1 عنصر را به طور معتبر رنگ‌آمیزی کرد، چقدر است؟ احتمال شکست در رنگ‌آمیزی عناصر زیرمجموعه‌ی S_i – که x به آن تعلق ندارد – همان 2^{-s_i+1} است. یک عنصر زیرمجموعه‌ی S_j (که x متعلق به آن است) آبی شده است، پس کافی است که دست‌کم یک عنصر آن قرمز شود. بنابراین احتمال شکست در رنگ‌آمیزی زیرمجموعه‌ی S_j، $2^{-(s_j-1)}$ است. دقت کنید که این احتمال، با احتمال شکست پش از رنگ‌آمیزی x برابر است! بنابراین، F(n) همچنان کم‌تر از ۱ باقی می‌ماند و حال تنها لازم است n-1 عنصر را رنگ‌آمیزی کنیم. آیا این مطلب بدان معناست که ما اکنون یک الگوریتم داریم؟ نه! معنایش این است که می‌توانیم نخستین گزینش را به دل‌خواه انجام دهیم. پس از انجام نخستین انتخاب، مسأله تغییر می‌کند.

دیگر نمی‌توانیم از فرض پیشین استقرا بهره‌برداری کنیم، چون پس از رنگ‌آمیزی نخستین عنصر، لازم است برخی زیرمجموعه‌ها، با دو رنگ و برخی دیگر، با یک رنگ رنگ‌آمیزی شوند. ناچار بازهم باید فرض استقرا تقویت شود تا این تغییر در آن بازتاب یابد. فرض کنید برخی عناصر از پیش رنگ‌آمیزی شده‌اند. با این فرض، یک زیرمجموعه در یکی از این چهار وضعیت قرار می‌گیرد: (۱) زیرمجموعه، هم عنصر آبی دارد و هم عنصر قرمز که در این صورت لازم نیست کاری روی آن انجام دهیم؛ (۲) زیرمجموعه، دست‌کم یک عنصر قرمز دارد، ولی عنصر آبی ندارد که در این حالت باید دست‌کم یکی از عناصر رنگ‌نشده‌ی آن آبی شود؛ (۳) زیرمجموعه، دست‌کم یک عنصر آبی دارد، اما هیچ عنصر قرمزی ندارد که در این حالت لازم است دست‌کم یکی از عناصرِ بدون رنگ آن قرمز شود؛ (۴) زیرمجموعه، هیچ عنصر رنگ‌آمیزی‌شده‌ای ندارد. زیرمجموعه‌ی حالت (۲) را زیرمجموعه‌ی قرمز، زیرمجموعه‌ی حالت (۳) را زیرمجموعه‌ی آبی و زیرمجموعه‌ی حالت (۴) را زیرمجموعه‌ی خنثا می‌نامیم. u_i را تعداد عناصر بدون‌رنگِ زیرمجموعه‌ی S_i بگیرید. اگر S_i در وضعیت (۱) باشد، رنگ‌آمیزی آن درست و موفق بوده است. اگر S_i در وضعیت قرمز یا آبی باشد (یعنی وضعیت‌های (۲) یا (۳)) آنگاه احتمال شکست در رنگ‌آمیزی تصادفی آن، 2^{-u_i} است. اگر S_i خنثا باشد، احتمال شکست در رنگ‌آمیزی تصادفی آن، 2^{-u_i+1} است. f_i را احتمال شکست در رنگ‌آمیزی تصادفی S_i بگیرید. باید این خاصیت را برقرار نگه داریم:

$$F(n)=\sum_{i=1}^{k} f_i < 1 \qquad (۱-۶)$$

فرض استقرا باید وضعیت همه‌ی زیرمجموعه‌ها را در نظر گرفته باشد. بنابراین، مسأله را به گونه‌ای گسترش می‌دهیم که دربرگیرنده‌ی زیرمجموعه‌های دل‌خواه قرمز، آبی و خنثا باشد. به عبارت دیگر، اینک ورودی مسأله، زیرمجموعه‌هایی است که برچسب‌های قرمز، آبی و خنثا خورده‌اند، در حالی که فرض می‌کنیم شرط (۶-۱) هم برقرار است.

مسأله: هر یک از عناصر S را با یکی از دو رنگ آبی و قرمز چنان رنگ‌آمیزی کنید که پس از رنگ‌آمیزی، هر زیرمجموعه‌ی قرمز، دست‌کم یک عنصر آبی و هر زیرمجموعه‌ی آبی، دست‌کم یک عنصر قرمز و هر زیرمجموعه‌ی خنثا، دست‌کم یک عنصر قرمز و یک عنصر آبی داشته باشد.

فرض استقرا برای این گسترش (عجیب و غریب) مسأله، کاملاً سرراست است:

فرض استقرا: اگر شرط (۶-۱) بر قرار باشد، می‌دانیم چگونه باید عنصرهای مجموعه‌ی S را که تعدادشان کم‌تر از n است، رنگ‌آمیزی کنیم تا شرایط مسأله برآورده شود.

این حالت پایه نیز مانند حالت پایه‌ی فرض پیشین استقراست. مجموعه‌ی n عنصری S به ما داده شده است و شرط (۶-۱) برای آن برقرار است. لازم است یک عنصر S را چنان رنگ‌آمیزی کنیم که شرط (۶-۱) بر قرار بماند.

دوباره از S، عنصر دل‌خواه x را برمی‌گزینیم. برای رنگ‌آمیزی x دو راه وجود دارد که هر یک از آن‌ها وضعیت متفاوتی را برای زیرمجموعه پیش می‌آورند. اگر x را قرمز کنیم، همه‌ی زیرمجموعه‌های قرمز دربرگیرنده‌ی x در همان حالت قرمز باقی می‌مانند (اما از تعداد عناصر رنگ‌آمیزی‌نشده‌ی آن‌ها یکی کم می‌شود) و همه‌ی زیرمجموعه‌های آبی دربرگیرنده‌ی x، به درستی رنگ‌آمیزی می‌گردند (و می‌توان آن‌ها را کنار گذاشت)؛ همه‌ی زیرمجموعه‌های خنثای دربرگیرنده‌ی x نیز، زیرمجموعه‌هایی قرمز می‌شوند. وضعیت زیرمجموعه‌هایی که x در آن‌ها نیست هم، بدون تغییر باقی می‌ماند. زدن رنگ آبی به x هم، به تغییرات مشابهی منجر می‌شود. حال، می‌توانیم مقدار F(n-1) را برای این حالت محاسبه کنیم. چون در این حالت، x را قرمز کرده‌ایم، این مقدار را با F_R(n-1) نشان می‌دهیم. در حالتی هم که x را آبی کرده باشیم، مقدار F(n-1) را با F_B(n-1) نشان می‌دهیم. کلید قفل الگوریتم، این لم است:

□ لم ۶-۲

F(n) را احتمال شکست، در همان آغاز کار، $F_R(n-1)$ را احتمال شکست، پس از قرمز کردن x و $F_B(n-1)$ را احتمال شکست، پس از آبی کردن x بگیرید. در این صورت:

$$F_R(n-1)+F_B(n-1)\le 2F(n).$$

برهان: زیرمجموعه‌ای که x در آن نباشد، بدون تغییر باقی می‌ماند. در آن صورت، $F_R(n-1)$، $F_B(n-1)$ و F(n) برای این زیرمجموعه با هم برابرند که با ادعای گفته‌شده سازگار است. اینک، زیرمجموعه‌های دربرگیرنده‌ی x را در نظر می‌گیریم. بسته به وضعیت زیرمجموعه، سه حالت ممکن است: (۱) زیرمجموعه‌ی قرمز تأثیری در مقدار $F_B(n-1)$ ندارد، چراکه هم‌اینک با موفقیت رنگ شده است، اما تأثیر آن در $F_R(n-1)$ دو برابر مقدار تأثیرش در F(n) است، زیرا اولی یک عنصر کم‌تر از دومی دارد. بازهم می‌بینیم که ادعای گفته‌شده برقرار است. (۲) بحث درباره‌ی زیرمجموعه‌ی آبی دقیقاً مانند بحث درباره‌ی زیرمجموعه‌ی قرمز است. (۳) تأثیر زیرمجموعه‌ی خنثایی با u_i عنصر در F(n)، 2^{-u_i+1} است. چون پس از رنگ‌آمیزی یک عنصر، این مجموعه، به یک زیرمجموعه‌ی قرمز یا یک زیرمجموعه‌ی آبی تبدیل می‌شود که در هر دو حالت، تعداد عناصر رنگ نشده‌ی آن یکی کم‌تر است؛ پس تأثیرش در $F_R(n-1)$ و $F_B(n-1)$ یکسان و برابر با $2^{-(u_i-1)}$ خواهد بود. می‌بینیم که در هر یک از این دو حالت، تأثیر آن در F(n)، $F_R(n-1)$ و $F_B(n-1)$ یکسان است و ادعای لم ثابت می‌شود.

□

لم ۶-۲ راه‌نمای ما در رسیدن به الگوریتم است. کار در حالت پایه‌ی تک‌عنصری ساده است، چون با برقراری شرط (۶-۱) تنها یک زیرمجموعه‌ی قرمز یا آبی ممکن است وجود داشته باشد که این تک‌عنصر را شامل شود. در این صورت، می‌توانیم این تک‌عنصر را با رنگ دیگر رنگ‌آمیزی کنیم. اگر داشته باشیم: $F_R(n-1)+F_B(n-1)\le 2F(n)$، آنگاه یا $F_R(n-1)\le F(n)$، یا $F_B(n-1)\le F(n)$ (و یا هر دو). می‌توانیم این مقادیر را محاسبه کنیم و اگر $F_R(n-1)$ از $F_B(n-1)$ کم‌تر است، x را آبی، وگرنه قرمز کنیم. به کمک لم ۶-۲، شرط (۶-۱) در فرض استقرا، برقرار می‌ماند و الگوریتم به پیش می‌رود. پیاده‌سازی این الگوریتم را به خواننده واگذار می‌کنیم.

۶-۱۰ یافتن اکثریت

E را دنباله‌ای از اعداد صحیح x_1، x_2، ... و x_n بگیرید. فراوانی x در E، تعداد دفعاتی است که x در E ظاهر شده است. اگر فراوانی z در E بزرگ‌تر از n/2 باشد، می‌گوییم عدد z در E یک اکثریت است.

> **مسأله:** دنباله‌ای از عددها به ما داده شده است. یا مشخص کنید که دنباله هیچ اکثریتی ندارد، یا یک اکثریت در آن بیابید.

برای مثال، هر عدد صحیح می‌تواند بیانگر یک رأی در انتخابات باشد. در این صورت، مسأله، بررسی این موضوع است که آیا کسی در انتخابات برنده شده است یا نه. اگر شمار نامزدها اندک باشد، آنگاه مرتب‌سازی سطلی می‌تواند مسأله را به صورت کارآمد، در زمانی از O(n) حل کند؛ اما اگر (مانند زمان حاضر) تعداد نامزدها بسیار زیاد باشد، آنگاه دیگر نمی‌توان مرتب‌سازی سطلی را به کار برد. در اینجا فرض می‌کنیم هیچ محدودیتی روی تعداد نامزدها وجود ندارد و هر یک از آن‌ها با یک عدد صحیح نشان داده شده‌اند. در سامانه‌های رایانه‌ای نیز رأی‌گیری (مثلاً برای سازگار کردن تصمیم‌ها) انجام می‌شود. (بهترین واژه در زبان پارسی برای سیستم، «هَنداد» است، اما به دلیل رایج نبودن این واژه، مترجمان واژه‌ی «سامانه» را به کار برده‌اند؛ نکته‌ی دیگر این که در سامانه‌های حساسی مانند ماهواره‌های فضایی، محاسبات را چندین بخش جداگانه انجام می‌دهند، سپس با رأی‌گیری در بین نتایجِ به‌دست‌آمده، پاسخ را بر می‌گزینند تا با خراب شدن یک بخش کل سامانه از کار نیفتد - مترجمان)

گاهی (مانند این نمونه) راه‌حلی که باید برای آن قدری فکر کرد، از راه‌حل سرراستی که در ابتدا به ذهن می‌رسد، بهتر است؛ راه‌حل درخشانی که پیاده‌سازی آن ساده‌تر هم هست. نخست، چند رویکرد سرراست را بررسی و سپس راه‌حل دقیق و زیبای مسأله را بیان می‌کنیم.

سرراست‌ترین راه برای حل این مسأله به کارگرفتن مرتب‌سازی است. با مرتب‌سازی رأی‌ها، شمارش تعداد رأی‌های هر نامزد آسان می‌شود، اما مرتب‌سازی، در بدترین حالت، به مقایسه‌هایی از $O(n\log n)$ نیاز دارد. خواهیم دید که کار را بهتر از این هم می‌توان انجام داد. می‌توانیم الگوریتم

یافتن میانه را به کار ببریم. اگر اکثریتی وجود داشته باشد، باید همان میانه باشد (زیرا میانه n/2امین عنصر از نظر کوچکی است و اکثریت، بیش از n/2 بار ظاهر می‌شود). بنابراین پس از یافتن میانه می‌توان تعداد دفعات ظهور آن را در دنباله حساب کرد و چنان‌چه میانه دارای اکثریت نباشد، در دنباله اکثریتی وجود نخواهد داشت. از آنجا که یافتن میانه از مرتب‌سازی دنباله آسان‌تر است؛ پس یافتن میانه، رویکرد بهتری است. روش دیگر، بهره‌گیری از یک الگوریتم مبتنی بر احتمال، به این صورت است که نمونه‌ی تصادفی کوچکی از رأی‌ها برگزینیم، اکثریتش را پیدا کرده، تعداد دفعات تکرار این اکثریت را در کل رأی‌ها حساب کنیم. اگرچه با این الگوریتم به آسانی می‌توان دریافت که آیا یک نامزد، اکثریت دارد یا نه، اما با آن، اثبات وجود نداشتن اکثریت ممکن نیست. خروجی چنین الگوریتمی ممکن است «نمی‌دانم» باشد. (از چنین روشی در همه‌پرسی‌ها یا انتخابات عادی نیز سود می‌جویند؛ زیرا بر پایه‌ی نظرسنجی‌های پیش از آن می‌توان دریافت کدام نامزدها شانس دست‌یابی به اکثریت را دارند.) از این گذشته، تعیین اندازه‌ی مناسب نمونه هم، کار آسانی نیست.

حال، الگوریتمی با زمان خطی برای یافتن اکثریت ارائه می‌کنیم که با هر تعداد نامزد کار می‌کند. این الگوریتم از الگوریتم یافتن میانه سریع‌تر و آسان‌تر است. مانند آنچه در الگوریتم یافتن ستاره‌ی مشهور (بخش ۵-۵) انجام دادیم، نخست، می‌کوشیم تا جای ممکن، عناصری را که می‌دانیم اکثریت نخواهند شد، کنار بگذاریم. این روش، سرانجام به جایی ختم می‌شود که همه‌ی عناصر را به جز یکی می‌توانیم کنار بگذاریم. یافتن نامزد با پی‌گیری این فرض (که به ما اجازه می‌دهد اندازه‌ی مسأله را کاهش دهیم) انجام می‌شود:

اگر $x_i \neq x_j$ و ما هر دوی این عناصر را از فهرست کنار بگذاریم، اکثریت فهرست اصلی، در فهرست تازه نیز اکثریت خواهد بود.

(دقت کنید که عکس این مطلب درست نیست؛ مثلاً ۱، ۲، ۵، ۵ و ۳ اکثریت ندارد، اما اگر ۱ و ۲ را از آن حذف کنیم، اکثریت فهرست تازه، ۵ خواهد شد.)

پس، اگر دو رأی نابرابر را از فهرست حذف کنیم و اکثریت فهرست تازه را (که کوچک‌تر هم هست) بیابیم؛ باید پس از آن بررسی کنیم که آیا این اکثریت، در فهرست اصلی هم اکثریت است یا نه. اگر نتوانیم دو رأی نابرابر در فهرست پیدا کنیم، چطور؟ اگر همه‌ی رأی‌ها بررسی شوند و با هم برابر باشند، آنگاه همین مقداری که همه با آن برابر هستند، تنها نامزد اکثریت خواهد شد. در حالت عادی، همین که یک رأی نابرابر با نامزد یافتیم، می‌توانیم فرض پیش را به کار ببریم، اما اگر رأی متفاوتی پیدا نشد، کافی است اکثریت بودنِ همان نامزد یافته‌شده را بررسی کنیم؛ هسته‌ی مرکزی ایده، همین است. اینک، روش پیاده‌سازی این ایده را توضیح می‌دهیم:

رأی‌ها به ترتیبی که در دنباله (فهرست) ظاهر شده‌اند، بررسی می‌شوند. دو متغیر C و M را به کار می‌بریم (حرف اول دو واژه‌ی نامزد و فراوانی در زبان انگلیسی). هنگامی که با x_i روبه‌رو می‌شویم، C، تنها نامزد اکثریت در بین x_1، x_2 ... و x_{i-1} است و M نشان می‌دهد که این نامزد، صرف نظر از

تعداد دفعاتی که آن را حذف کرده‌ایم، تا به حال چند بار ظاهر شده است. به عبارت دیگر رأی‌های x_1، x_2، ... و x_{i-1} را می‌توان به دو گروه به اندازه‌ی 2k و M تقسیم کرد، به گونه‌ای که 2k+M=i-1. گروه نخست، شامل k جفت از رأی‌های نابرابر (که بنا به فرض می‌توان آن‌ها را حذف کرد) و گروه دوم شامل M بار ظهور C در دنباله است. اگر در بین x_1، x_2، ... و x_{i-1} اکثریتی وجود داشته باشد، آنگاه باید از این روش حذف، جان سالم به در برده، در C قرار گرفته باشد. (بازهم دقت کنید که عکس این مطلب درست نیست؛ یعنی ممکن است C از فرایند حذف جان سالم به در ببرد، ولی اکثریت نباشد.) هنگامی که به x_i می‌رسیم، آن را با C مقایسه می‌کنیم و بنا به برابر یا نابرابر بودنِ x_i با C، فراوانی نامزد را یک واحد افزایش یا کاهش می‌دهیم. البته باید مراقب حالتی هم باشیم که در آن هیچ نامزدی وجود ندارد (برای مثال، اگر $x_2 \neq x_1$ در x_3 چنین وضعیتی رخ می‌دهد). این حالت، هنگام صفر بودنِ M رخ می‌دهد که ما باید طبق الگوریتم در C، x_i و در M، ۱ قرار دهیم. در پایان، تنها یک نامزد (C) داریم و می‌توانیم تعداد دفعات ظهور آن در فهرست را محاسبه کنیم و مشخص سازیم که آیا C، اکثریت است، یا این که اصلاً اکثریتی وجود ندارد. این الگوریتم در شکل ۶-۲۸ ارائه شده است.

الگوریتم: Majority(X,n)

ورودی: X (آرایه‌ای به اندازه‌ی n از اعداد صحیح مثبت)

خروجی: Majority (اکثریت X، اگر X اکثریت داشته باشد، وگرنه ۱-)

```
begin
    C := X[1];
    M := 1;
    {نخستین پویش: همه، به جز نامزد C حذف می‌شوند.}
    for i := 2 to n do
        if M = 0 then
          C := X[i];
          M := 1
        else
            if C = X[i] then M := M + 1
            else M := M-1;
    {دومین پویش: بررسی این که آیا C اکثریت است یا نه.}
    if M = 0 then Majority := -1
    else
        Count := 0;
        for i := 1 to n do
            if X[i] = C then Count := Count + 1;
        if Count > n/2 then Majority := C
        else Majority := -1
end
```

شکل ۶-۲۸ الگوریتم Majority

پیچیدگی: n-1 مقایسه برای یافتن یک نامزد و n-1 مقایسه نیز در بدترین حالت، برای تعیین اکثریت بودن یا نبودن آن انجام می‌شود. پس، در کل، حداکثر 2n-2 مقایسه لازم است. می‌توان تعداد مقایسه‌ها را به 3n/2+1 کاهش داد و تعداد بهینه هم همین است (Fischer و Salzberg [۱۹۸۲]). در هر صورت، زمان کل اجرا از $O(n)$ خواهد بود، چراکه شمار اعمال دیگری که به ازای هر مقایسه انجام می‌شود، ثابت است.

۶-۱۱ سه نمونه از روش‌های جالب اثبات[۳]

در این بخش با سه مسأله‌ی کاملاً متفاوت درباره‌ی دنباله‌ها، مجموعه‌ها و مجموعه‌های چندگانه (مجموعه‌ی چندگانه مفهومی شبیه مجموعه‌ی عادی دارد، با این تفاوت که ممکن است عناصر، چند بار عضو آن باشند؛ مانند {۱,۳,۲,۱} – مترجمان) آشنا می‌شویم که الگوریتم هر یک از آن‌ها را با روشی متفاوت به دست می‌آوریم. نخستین الگوریتم از اصل تقویت فرض استقرا بهره می‌گیرد. برای این الگوریتم، فرض استقرا چهار بار تقویت می‌شود تا الگوریتمی کارآمد به دست آید. دومین الگوریتم مثالی از یک شیوه‌ی واضح، یعنی بهبود «قضیه» با کنار گذاشتن تمام فرضیات غیرضروری است؛ این مثال نشان خواهد داد که این قاعده همیشه هم ساده و سرراست نیست. سومین مثال نیز نشان می‌دهد که چگونه می‌توان با گزینش هوشمندانه‌ی پایه‌ی استقرا، یک الگوریتم را بهبود بخشید.

۶-۱۱-۱ بلندترین زیردنباله‌ی صعودی (یا افزایشی)

S را دنباله‌ای از اعداد صحیح و متمایز x_1، x_2، ... و x_n بگیرید. یک زیردنباله‌ی صعودی (یا IS) عبارت است از x_{i_1}، x_{i_2}، ... و x_{i_k} به گونه‌ای که $i_1<i_2<\ldots<i_k$ و برای هر j در فاصله‌ی [1,k] داشته باشیم: $x_{i_j} < x_{i_{j+1}}$. بلندترین زیردنباله‌ی افزایشی در S (یا LIS) یک زیردنباله‌ی افزایشی با بیش‌ترین طول ممکن است.

مسأله: در دنباله‌ای از اعداد صحیح متمایز، بلندترین زیردنباله‌ی صعودی را پیدا کنید.

الگوریتمی که در اینجا می‌سازیم، نمونه‌ی بسیار خوبی از اصل تقویت فرض استقراست. فرض را چندین بار و هر بار برای برطرف ساختن مشکلی که بار قبل پیش آمده است، تقویت می‌کنیم. نخست فرض ساده و سرراست را در نظر بگیرید:

۳- در نخستین بارِ خواندن کتاب، می‌توانید از این بخش چشم‌پوشی کنید.

فرض استقرا (نخستین تلاش): می‌دانیم چگونه یکی از بلندترین زیردنباله‌های افزایشی را در هر دنباله‌ای با طول کم‌تر از m بیابیم.

حل حالت پایه (دنباله‌ای با طول ۱) روشن است. با داشتن دنباله‌ای به طول m، یک LIS در m-1 عنصر نخست می‌یابیم و سپس x_m را در نظر می‌گیریم. اگر x_m از عنصر پایانی LIS – که بنا به استقرا آن را داریم – بزرگ‌تر باشد، آنگاه می‌توانیم x_m را به پایان این LIS بیفزاییم که به این ترتیب، مسأله برای m نیز حل می‌شود؛ اما اگر x_m از عنصر پایانی LIS بزرگ‌تر نباشد، ادامه‌ی حل مسأله به این سادگی نیست. برای مثال، ممکن است چندین LIS متمایز وجود داشته باشد و بتوان x_m را به یکی از آن‌ها افزود، اما شاید LIS مورد نظر، همان بلندترین زیردنباله‌ی صعودی یافته‌شده در فرض استقرا نباشد. حال، در استقرا فرض را به این صورت تقویت می‌کنیم:

فرض استقرا (دومین تلاش): می‌دانیم چگونه همه‌ی بلندترین زیردنباله‌های افزایشی را در دنباله‌ای با طول کم‌تر از m بیابیم.

بازهم حالت پایه روشن و بدیهی است و از استقرا به همان روش پیش سود می‌جوییم، با این تفاوت که حال می‌توانیم x_m را با همه‌ی LISها بسنجیم و بلندترین IS را بیابیم. بدین ترتیب مشکل پیش حل می‌شود، اما مشکل دیگری – یعنی لزوم یافتن تمام LISها – پیش می‌آید. اگر نتوان x_m را به هیچ یک از LISها افزود، آنگاه احتمال دارد یک IS با طول یک واحد کم‌تر از بلندترین IS وجود داشته باشد که بتوان x_m را به آن افزود تا LIS تازه‌ای به وجود آورَد. گویا از چاله در آمده‌ایم و به چاه افتاده‌ایم، چراکه حالا ناچاریم هم تمام ISهای با بیش‌ترین طول و هم تمام ISهای یک مرتبه کوتاه‌تر از بیش‌ترین طول را بیابیم؛ اما برای یافتن ISهای دسته‌ی دوم ناچاریم ISهای دو مرتبه کوتاه‌تر از بیش‌ترین طول را نیز بیابیم و ... . این مثال نمونه‌ی خوبی از زیاده‌روی در تقویت فرض استقراست.

بگذارید به عقب بازگردیم و نگاه دیگری به فرض قوی‌تر استقرا بیندازیم. آیا واقعاً به همه‌ی LISها نیاز داریم؟ خیر! کافی است بدانیم که آیا می‌توان x_m را به یکی از آن‌ها افزود یا نه. آیا راهی وجود دارد که «بهترین» آن‌ها را از نظر «امکان افزودن x_m» پیدا کنیم؟ پاسخ مثبت است. بهترین LIS آن است که با عدد کوچک‌تری به پایان برسد چراکه اگر بتوانیم x_m را به این LISها بیفزاییم، بی‌شک می‌توان x_m را به این LIS هم افزود. (شاید چندین LIS متمایز، با عدد پایانی یکسان، این قابلیت را داشته باشند. برای سادگی، به جای گفتن «یک LIS دل‌خواه از بهترین‌ها» می‌گوییم «بهترین LIS».) بیایید فرض دیگری برای استقرا در نظر بگیریم که از قبلی، اندکی ضعیف‌تر باشد:

فرض استقرا (سومین تلاش): می‌دانیم چگونه یکی از بلندترین زیردنباله‌های افزایشی را در دنباله‌ای با طول کوچک‌تر از m بیابیم، به گونه‌ای که عدد پایانیِ هیچ یک از بلندترین زیردنباله‌های دیگر، از عدد پایانی این زیردنباله کوچک‌تر نباشد.

حالت پایه در اینجا نیز روشن است. با توجه به x_m درمی‌یابیم که آیا می‌توان آن را به «LIS پیداشده با استقرا» افزود یا نه. فرض کنید طول این LIS، s باشد. اگر بتوان x_m را به آن افزود، LIS تازه‌ای داریم

که از قبلی بلندتر است؛ پس، این LIS تازه که یکتاست «بهترین» هم هست و کار انجام شده است. اگر هم نتوان x_m را به این LIS افزود، خیالمان راحت است که هیچ زیردنباله‌ی افزایشی بلندتری وجود ندارد، اما هنوز کارمان به پایان نرسیده است؛ زیرا ممکن است با حالتی برخورد کنیم که نتوان x_m را به بهترین LIS افزود (به این دلیل که از عدد پایانیِ LIS کوچک‌تر است) اما بتوان آن را به یک IS با طول s-1 افزود و LIS دیگری ایجاد کرد که عدد پایانی آن کوچک‌تر باشد. برای بررسی این حالت باید بهترین IS با طول s-1 را بشناسیم، اما باز دوباره، اگر فرض استقرا بگوید که بهترین IS به طول s-1 را می‌شناسیم، آنگاه امکان دارد x_m به یک IS با طول s-2 اضافه شود و یک IS تازه به طول s-1 بسازد که از بقیه بهتر باشد. باید تعیین کنیم آیا افزودن x_m به چنین ISی کارساز خواهد بود یا نه، تا بتوانیم بر اساس استقرا پیش برویم. بنابراین، لازم خواهد شد که بهترین ISها به طول s-2، s-3، ... و ۱ را بشناسیم. روشن است که بهترین IS به طول ۱ کوچک‌ترین عدد درون دنباله است. (بدون به‌کارگیری استقرا نیز می‌توان دریافت که نباید ISهای کوتاه‌تر را به دل‌خواه کنار گذاشت و آن‌ها را نادیده گرفت، چراکه همواره احتمال دارد یکی از این ISها آغازگر LIS نهایی باشد.)

دوباره می‌کوشیم فرض را تقویت کنیم. بهترین زیردنباله‌ی افزایشی به طول k را - یعنی همان را که با عدد کوچک‌تری به پایان می‌رسد - با BIS(k) نشان می‌دهیم (اگر هم تعداد چنین زیردنباله‌هایی بیش‌تر از یکی باشد، یکی از آن‌ها را به دل‌خواه برمی‌گزینیم). آخرین عدد دنباله‌ی BIS(k) را نیز با BIS(k).last نشان می‌دهیم.

فرض استقرا (چهارمین تلاش): می‌دانیم در دنباله‌ای با طول کوچک‌تر از m، چگونه برای هر k که k>m-1، BIS(k) را - در صورت وجود - بیابیم.

هنوز هم حالت پایه بدیهی است. با توجه به x_m، باید بفهمیم که کدام BISها ممکن است تغییر کنند. x_m به یک BIS(k) افزوده می‌شود، اگر و تنها اگر این دو شرط درست باشند: (۱) x_m>BIS(k).last که در این صورت، x_m می‌تواند به BIS(k) اضافه شود و (۲) x_m<BIS(k+1).last که در آن صورت با قرار گرفتن x_m در انتهای BIS(k) نتیجه‌ی بهتری نسبت به BIS(k+1) به دست خواهد آمد. ادعا می‌کنیم: $BIS(1).last < BIS(2).last < \ldots < BIS(s).last$ که در آن s، طول LIS است. این ادعا درست است، چراکه اگر دست‌کم برای یک j داشته باشیم: $BIS(j).last \le BIS(j-1).last$، آنگاه j-1 عدد از ابتدای BIS(j) از BIS(j-1) بهتر خواهد بود. الگوریتم بدین ترتیب پیش می‌رود: با توجه به x_m، به مقادیر $BIS(i).last$ برای i=s، i=s-1، i=s-2 و ... نگاه می‌کند تا آن که یکی، مثلاً $BIS(j).last$ را بیابد که از x_m کوچک‌تر باشد. اگر نتوان j را چنان یافت که $BIS(j).last < x_m$، آنگاه x_m کوچک‌ترین عدد دنباله تا اینجاست و همان BIS(1) می‌شود. اگر j=s، آنگاه به BIS(s)، x_m را می‌افزاییم که یک BIS(s+1) تازه ایجاد خواهد شد. ((BIS(s) قبلی بدون تغییر باقی می‌ماند.) اگر هم $j \ne s$، آنگاه: $BIS(j).last < x_m < BIS(j+1).last$ که در آن صورت به جای BIS(j+1)، دنباله‌ی «BIS(j)، x_m» را قرار می‌دهیم.

کل الگوریتم بر همین پایه است و به سادگی با نخستین استفاده‌ی درست از استقرا، کار پیش خواهد رفت. دقت کنید که می‌توان از جست‌وجوی دودویی کمک گرفت، چون مجموعه مورد جست‌وجو مرتب است. از این رو، هر x_m به کارهایی که باید انجام شود، حداکثر $O(\log m)$ مقایسه می‌افزاید و در نتیجه زمان کل اجرا از $O(n \log n)$ خواهد شد. جزئیات این الگوریتم را به خواننده واگذار می‌کنیم که البته کار چندان ساده‌ای هم نیست.

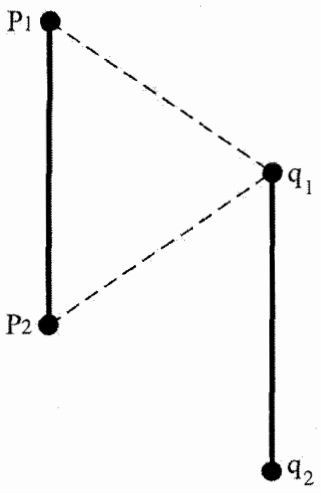
۶-۱۱-۲ یافتن بزرگ‌ترین دو عنصر یک مجموعه

یک ترفند - که تقریباً در اثبات هر قضیه‌ای اهمیت دارد - بررسی کامل اثبات از آغاز تا پایان برای یافتن فرضیات یا گام‌هایی است که وجودشان ضرورتی ندارد. حذف چنین فرضیاتی معمولاً ما را به قضیه‌ای بهتر می‌رساند. گاهی نیز داشتن فرضیات اضافی نشانه‌ی وجود اشتباه در برهان است. به گفته‌ی Polya و Szego [۱۹۲۷]: «باید سرتاسر برهان را با دقت بررسی کنیم تا دریابیم که آیا واقعاً همه‌ی فرضیات به کار گرفته شده‌اند یا نه؛ باید بررسی کنیم که آیا با فرضیاتی کم‌تر هم می‌توانیم به همان نتایج برسیم یا نه ... نباید به سادگی قانع شویم تا آن که روشن گردد با حذف هر یک از فرضیات می‌توان برای قضیه مثال نقضی آورد.» با الگوریتم‌ها هم باید همین‌گونه برخورد کرد. اگرچه این اصول، ساده به نظر می‌رسند، اما چنان که در مثال بعد خواهیم دید، همیشه هم این‌گونه نیست.

مسأله: مجموعه‌ی S از n عدد x_1، x_2، ... و x_n داده شده است. نخستین و دومین عدد از نظر بزرگی را بیابید.

به دنبال الگوریتمی هستیم که تعداد مقایسه‌های بین عناصر را کمینه سازد (با اعمالِ دیگر کاری نداریم). برای سادگی، n را توانی از ۲ می‌گیریم.

به یاری روش تقسیم‌وحل، مجموعه‌ی S (با اندازه‌ی n) را به دو زیرمجموعه‌ی P و Q با اندازه‌های n/2 تقسیم می‌کنیم. با بهره‌گیری از استقرایی سرراست، فرض می‌کنیم که بزرگ‌ترین دو عنصر مجموعه‌های P و Q را می‌شناسیم و آن‌ها را به ترتیب با p_1، p_2 و q_1، q_2 نشان می‌دهیم. حال، می‌کوشیم بزرگ‌ترین دو عنصر S را بیابیم. روشن است که برای انجام این دو کار دو مقایسه لازم و کافی است: یک مقایسه بین بزرگ‌ترین عناصر، یعنی p_1 و q_1 انجام می‌شود و مقایسه‌ی دیگر بین «بازنده‌ی» این مقایسه و دومین عنصر از نظر بزرگی در سمت «برنده» است (شکل ۶-۲۹ را ببینید). این شیوه به رابطه‌ی بازگشتی $T(2n) = 2T(n) + 2$ و $T(2) = 1$ می‌انجامد که حلش $T(n) = 3n/2 - 2$ است. این تعداد مقایسه از تعداد مقایسه‌های رویکرد سرراست (یعنی 2n-3) بهتر است. این شیوه بسیار شبیه یافتن عنصر بیشینه و کمینه (ارائه شده در بخش ۶-۵-۱) است. اکنون به دنبال راه‌حل بهتری می‌رویم.



شکل ۶-۲۹ یافتن بزرگ‌ترین دو عنصر (خط‌چین‌ها بیانگر مقایسه‌های لازم هستند.)

اگر در هر گام استقرا دو مقایسه لازم باشد، چگونه می‌توان تعداد کل مقایسه‌ها را کم‌تر کرد؟ با نگاه دقیقی به این دو مقایسه در شکل ۶-۲۹، می‌بینیم که دیگر از q_2 در الگوریتم استفاده نمی‌شود. بنابراین مقایسه‌ای که پیش‌تر برای یافتن آن انجام داده بودیم، لازم نبوده است. اگر از انجام این مقایسه‌ها بپرهیزیم، شمار قابل‌توجهی از تعداد مقایسه‌ها کاسته خواهد شد. از سویی، تا p_1 و q_1 را با هم مقایسه نکنیم، روشن نمی‌شود که از p_2 و q_2 کدام یک را باید کنار بگذاریم. اگر می‌دانستیم کدام زیرمجموعه بازنده‌ی مقایسه است، می‌توانستیم الگوریتم معمولی یافتن عنصر بیشینه را برای آن به کار ببریم تا اعمال کم‌تری انجام شود. پس با این که می‌دانیم می‌توان از مقایسه‌های بسیاری چشم‌پوشی کرد، اما نمی‌دانیم که این مقایسه‌ها کدام‌ها هستند.

ترفندی که در اینجا به کار می‌گیریم، به عقب انداختن محاسبه‌ی دومین عنصر بزرگ تا نزدیک پایان کار است. در فرض استقرا، تنها فهرستی از نامزدهای دومین عنصر بزرگ را نگه‌داری می‌کنیم، بدون آن که مشخص کنیم کدام یک دومین عنصر بزرگ است.

فرض استقرا: می‌دانیم چگونه در مجموعه‌های با اندازه‌ی کوچک‌تر از n، عنصر بیشینه و مجموعه‌ی «کوچکی» از نامزدهای دومین عنصر بزرگ را بیابیم.

اندازه‌ی مجموعه‌ی «کوچک» را در فرض استقرا تعریف نکرده‌ایم، بلکه هنگام پیش‌روی برای ساخت الگوریتم، آن را خواهیم یافت.

الگوریتم این‌گونه جلو می‌رود: مجموعه‌ی S با اندازه‌ی n را به دو زیرمجموعه‌ی P و Q با اندازه‌های n/2 تقسیم می‌کنیم. بنا به فرض استقرا، هم بزرگ‌ترین عنصر این دو زیرمجموعه، یعنی p_1 و q_1 را می‌شناسیم و هر دو مجموعه‌ی نامزدهای دومین عنصر بزرگ برای هر یک از این دو زیرمجموعه، یعنی C_P و C_Q را. p_1 و q_1 را با هم مقایسه می‌کنیم و بزرگ‌ترین آن‌ها، مثلاً p_1 را، بزرگ‌ترین عنصر S

می‌گیریم. سپس C_Q را دور می‌اندازیم، چراکه عناصر آن از q_1 کوچک‌ترند و تنها q_1 را به C_P می‌افزاییم. در پایان، بزرگ‌ترین عنصر مجموعه و مجموعه‌ای از نامزدهای دومین عنصر بزرگ را داریم و از این مجموعه به طور مستقیم، دومین عنصر بزرگ را پیدا می‌کنیم. تعداد مقایسه‌ها برای یافتن بزرگ‌ترین عنصر در رابطه‌ی بازگشتی $T(n)=2T(n/2)+1$ و $T(2)=1$ صدق می‌کند که از آن نتیجه می‌شود: $T(n)=n-1$. به آسانی می‌توان دید که اندازه‌ی $\log_2^n$ برای مجموعه‌ی نامزدها کافی است، زیرا هر بار که اندازه‌ی مجموعه‌ی مورد بررسی را دو برابر می‌کنیم، یک عنصر نیز به مجموعه‌ی نامزدها می‌افزاییم. بنابراین یافتن دومین عنصر بزرگ نیاز به $\log_2^n-1$ مقایسه‌ی اضافی دارد. پس تعداد کل مقایسه‌ها $n-1+\log_2^n-1$ است که قطعاً بهترین تعداد مقایسه‌های ممکن خواهد بود (Knuth [۱۹۷۳b] را ببینید). بدین ترتیب، اگر n توانی از ۲ باشد، فرض استقرا چنین می‌شود:

فرض استقرا: می‌دانیم چگونه در مجموعه‌های با اندازه‌ی کوچک‌تر از n، عنصر بیشینه و مجموعه‌ای از نامزدهای دومین عنصر بزرگ را بیابیم که اندازه‌ی مجموعه‌ی نامزدها حداکثر $\log_2^n$ باشد.

توجه: پس از آن که الگوریتمی ساخته شد، خوب است که آن را به دقت بررسی کنیم تا بخش‌هایی را که دخالتی در پاسخ نهایی ندارند، بیابیم. بیش‌تر اوقات، می‌توان چنین بخش‌هایی را کنار گذاشت. حتا اگر نتوان اعمال اضافی را حذف کرد، ممکن است بتوان به جای آن‌ها اعمال ساده‌تری قرار داد که این کار، الگوریتم را کارآمدتر می‌سازد.

۶-۱۱-۳ پیدا کردن مد در یک مجموعه‌ی چندگانه

$S=(x_1,x_2,...,x_n)$ را مجموعه‌ای چندگانه از عناصری متعلق به «مجموعه‌ای با ترتیب کلی» در نظر بگیرید (برخی از این عناصر می‌توانند برابر باشند) مد یک مجموعه‌ی چندگانه، عنصری است که بیش از دیگر عناصر تکرار شده باشد (ممکن است یک مجموعه‌ی چندگانه بیش از یک مد داشته باشد). به تعداد دفعات تکرار یک عنصر، فراوانی آن عنصر می‌گویند، بنابراین، مد یک مجموعه‌ی چندگانه عنصری است که بیش‌ترین فراوانی را داشته باشد.

مسأله: یک مد برای مجموعه‌ی چندگانه داده‌شده‌ی S بیابید.

هدف ما کمینه‌سازی تعداد مقایسه‌هاست. یک راه برای یافتن مد، بهره‌گیری از مرتب‌سازی است. پس از آن که عناصر را مرتب کردیم، می‌توانیم با پویش دنباله‌ی مرتب‌شده، فراوانی عناصر را حساب کنیم (زیرا در دنباله‌ی مرتب‌شده، عناصر برابر، پشت‌سرهم قرار دارند). خواهیم دید که همیشه هم لازم نیست از مرتب‌سازی کمک بگیریم. از آنجا که یافتن اکثریت در زمانی خطی شدنی است (بخش ۶-۱۲) اما مرتب‌سازی به زمانی از $O(n\log n)$ نیاز دارد؛ به این فکر می‌افتیم که شاید مرتب‌سازی لازم نباشد.

از این رو، چون فراوانی مد زیاد است، حدس می‌زنیم که شاید روش سریعی برای یافتن آن، بدون مرتب‌سازی وجود داشته باشد.

بگذارید رویکرد ساده‌ی استقرایی را بیازماییم. فرض کنید مد یک مجموعه‌ی چندگانه‌ی n-1 عنصری را می‌شناسیم و می‌خواهیم مد یک مجموعه‌ی چندگانه n عنصری را بیابیم. این کار چندان آسان نیست، چراکه ممکن است چندین عنصر بیش‌ترین فراوانی را در بین n-1 عنصر داشته باشند و عنصر nام سرنوشت مد را تعیین کند. اگر فرض استقرا بگوید که همه‌ی عناصر دارای بیش‌ترین فراوانی را می‌شناسیم، آنگاه می‌توان مشخص کرد که آیا عنصر nام تساوی آن‌ها را به هم خواهد زد یا نه؛ اما از سویی، ممکن است عنصر nام فراوانی عنصری به جز این عناصر را افزایش دهد و آن عنصر را به فهرست عناصر با بیش‌ترین فراوانی بیفزاید. پیش‌تر (در بخش ۶-۱۱-۱) دیدیم که ردگیری تمام «بهترین» پاسخ‌ها شدنی است، اما هزینه‌اش هم احتمالاً بسیار بالاست. (در اینجا نویسنده به اشتباه، خواننده را به بخش ۶-۱۳-۱ ارجاع داده بود - مترجمان) از طرفی، لازم نیست عنصر nام دل‌خواه باشد، بلکه خودمان می‌توانیم عنصری ویژه را به عنوان عنصر nام برگزینیم. اگر nامین عنصر، بزرگ‌ترین عنصر باشد، بازهم با همان مشکل روبه‌رو هستیم؛ هرچند که به پاسخ نزدیک‌تر شده‌ایم. می‌توانیم اندازه‌ی مسأله را نه تنها با حذف یک عنصر بیشینه، بلکه با حذف تمامی آن‌ها کاهش دهیم. سپس، مسأله‌ی کاهش‌یافته را حل و فراوانی مد آن را با فراوانی عنصر بیشینه مقایسه می‌کنیم.

هرچند اینک نیز الگوریتم حل مسأله را در اختیار داریم، اما بدبختانه این الگوریتم بسیار کند است. یافتن بیشینه‌ی یک مجموعه‌ی چندگانه‌ی n عنصری به n-1 مقایسه نیاز دارد. پس، اگر تعداد عناصر متمایز مجموعه‌ی چندگانه زیاد باشد، آنگاه محاسبات بسیاری برای یافتن عنصر بیشینه باید انجام شود. به خصوص اگر مجموعه‌ی چندگانه، در واقع، یک مجموعه‌ی عادی باشد (یعنی همه‌ی عناصرش متمایز باشند) آنگاه این الگوریتم مانند «مرتب‌سازی با انتخاب» و از $O(n^2)$ خواهد بود.

برای بهبود کارایی این الگوریتم به روش تقسیم‌وحل متوسل می‌شویم. به جای آن که در فرض استقرا یک عنصر یا مجموعه‌ی کوچکی از عناصر را به کار ببریم، می‌کوشیم مجموعه‌ی چندگانه را به دو بخش با اندازه‌ی تقریباً برابر تقسیم کنیم. این دو بخش باید جداازهم باشند تا به زیرمسأله‌هایی مستقل از یکدیگر تبدیل شوند. چگونه می‌توان یک مجموعه‌ی چندگانه را به دو بخش جداازهم با اندازه‌های تقریباً برابر تقسیم کرد؟ می‌توانیم نخست، میانه‌ی مجموعه‌ی چندگانه را بیابیم و سپس آن را به سه بخش تقسیم کنیم - یک بخش، با عناصری کوچک‌تر از میانه، یک بخش، با عناصری برابر با آن و بخش دیگر، با عناصری بزرگ‌تر از میانه. پیش‌تر دیدیم که چگونه می‌توان در حالت میانگین، میانه را با $O(n)$ مقایسه یافت (بخش ۶-۵). هرچند اثبات نکردیم، اما بدانید که زمان یافتن میانه در بدترین حالت از $O(n)$ است. الگوریتم یافتن میانه را به صورت گامی از الگوریتم اصلی به کار می‌بریم. با داشتن مجموعه‌ی چندگانه‌ای با اندازه‌ی n، نخست، میانه‌ی آن را می‌یابیم و سپس بر اساس آن، مجموعه‌ی چندگانه را به سه بخش تقسیم می‌کنیم، سپس دو زیرمسأله با اندازه‌ی نابزرگ‌تر از n/2 را

حل می‌کنیم (اعضای یکی از سه زیرمسأله همگی برابرند - مترجمان). مد مجموعه‌ی چندگانه‌ی اصلی به آسانی از روی مد دو مجموعه‌ی چندگانه‌ی کوچک‌تر به دست می‌آید، چراکه این دو مجموعه‌ی چندگانه‌ی کوچک‌تر جداازهم هستند. از آنجا که یافتن میانه و تقسیم مسأله در زمانی خطی انجام‌شدنی است، به این رابطه‌ی بازگشتی آشنا می‌رسیم:

$$T(n) \le 2T(n/2) + O(n) \text{ و } T(2) = 1$$

که از آن نتیجه می‌شود: $T(n) = O(n \log n)$؛ اما این زمان از زمان مرتب‌سازی بهتر نیست. در واقع، اگر عنصری را که تقسیم بر مبنای آن انجام شده است، به جای میانه، به طور تصادفی برگزیده باشیم، آنگاه در اصل، این الگوریتم همان «مرتب‌سازی سریع» خواهد بود.

حال، به قلب الگوریتم می‌رسیم. برای بهبود کارایی به پایه‌ی استقرا توجه می‌کنیم. فراوانی مد را M بگیرید. ادعا می‌کنیم که پایه‌ی استقرا می‌تواند از زیرمجموعه‌ی چندگانه‌ای با اندازه‌ی M آغاز شود. به عبارت دیگر، مجبور نیستیم عمل تقسیمِ مجموعه‌ی چندگانه را پس از رسیدن به بخش‌هایی با اندازه‌ی M بازهم ادامه دهیم. مد، در همین مرحله مشخص می‌شود، زیرا فراوانی هیچ عنصر دیگری از M بیش‌تر نخواهد شد. بنابراین، دیگر لازم نیست مجموعه‌ی چندگانه را بیش از این تقسیم کنیم.

پیاده‌سازی این الگوریتم چندان سرراست نیست. نمی‌توانیم الگوریتم را به روش بازگشتی پیاده‌سازی کنیم، چراکه از پیش نمی‌دانیم عمل بازگشت چقدر باید ادامه پیدا کند. عمل بازگشت باید زمانی پایان پذیرد که اندازه‌ی مجموعه‌ی چندگانه، M یا کوچک‌تر از M شود، اما مقدار M در طی اجرای الگوریتم و با بررسی مجموعه‌های چندگانه‌ی کوچک‌تر به دست می‌آید. در هر گام، همه‌ی زیرمجموعه‌های چندگانه بررسی می‌شوند تا اگر در یکی از آن‌ها همه‌ی عناصر یکسان بود، بازگشت را متوقف کنیم؛ وگرنه عمل تقسیم بازهم ادامه می‌یابد. جزئیات پیاده‌سازی این الگوریتم را به خواننده واگذار می‌کنیم.

پیچیدگی: در الگوریتم تازه، تنها پایه‌ی رابطه‌ی بازگشتی پیش تغییر کرده است:

$$T(n) \le 2T(n/2) + O(n) \text{ و } T(M) = O(M)$$

که از آن نتیجه می‌شود تعداد مقایسه‌ها از $O(n \log(n/M))$ است. به طور شهودی برای توضیح رابطه‌ی بازگشتی می‌توان گفت: عمل بازگشت تا هنگامی ادامه می‌یابد که به یک مجموعه‌ی چندگانه با اندازه‌ی M برسیم که در کل، چنین رخدادی $\log(n/M)$ مرتبه اتفاق می‌افتد و هر بار نیز مقایسه‌های مورد نیاز برای تقسیم و بررسی همه‌ی زیرمسأله‌ها در زمانی خطی انجام می‌شود. به ویژه، اگر $M = cn$ (c یک ثابت است) آنگاه زمان این الگوریتم خطی خواهد بود. اگر M ثابت باشد، الگوریتم از $O(n \log n)$ می‌شود. پس، این الگوریتم تنها در صورتی که هم M و هم هزینه‌ی مقایسه‌ها نسبتاً زیاد باشد، از حل مسأله به روش مرتب‌سازی بهتر خواهد بود. (در این روش، سربار نگه‌داری زیرمسأله‌ها نیز زیاد است.)

۶-۱۲ خلاصه

در این فصل درباره‌ی موضوعات گوناگونی بحث شد: جست‌وجو، مرتب‌سازی، مرتبه‌ی آماری، فشرده‌سازی داده‌ها، کار با رشته‌ها، الگوریتم‌های مبتنی بر احتمال و ... . در هر یک از این موضوعات، تنها یک یا دو مسأله‌ی پایه‌ای ارائه گردید. در عمل، برخلاف مسأله‌های این فصل، تعریف ساده و روشن بیش‌تر مسأله‌ها ممکن نیست. پس باید کوشید تا درکی مجرد و انتزاعی از بخش‌های اصلی و کلیدی آن‌ها به دست آورد. روش‌ها و ترفندهایی که در این فصل به کار بردیم، کاملاً شبیه همان‌هایی هستند که در فصل ۵ با آن‌ها آشنا شدید. در اینجا، بازهم نقش اصلی بر عهده‌ی استقراست.

بسیاری از مسأله‌هایی که در این فصل بررسی شدند، راه‌حل‌های ساده و سرراستی دارند که با اندکی تلاش به دست می‌آیند (جست‌وجوی خطی و مرتب‌سازی با انتخاب، دو نمونه از این مسأله‌ها هستند). اگر اندازه‌ی ورودی کوچک باشد، اغلب این راه‌حل‌ها، هم مناسبند، هم بر راه‌حل‌های پیچیده برتری دارند؛ اما اگر اندازه‌ی ورودی کوچک نباشد (مثلاً بیش از ۱۰۰ عنصر در ورودی داشته باشیم) یافتن راه‌حل‌های بهتر پراهمیت‌تر می‌شود. برای نمونه، جست‌وجوی خطی و مرتب‌سازی‌های از $O(n^2)$ بسیار رایج هستند و بدبختانه، این‌گونه الگوریتم‌های ناکارآمد را برای ورودی‌های بزرگ نیز به کار می‌برند.[۴]

مراجعی برای مطالعه‌ی بیش‌تر

می‌توانید در Knuth [۱۹۷۳b] گنجینه‌ای از مطالب را درباره‌ی مرتب‌سازی و جست‌وجو همراه با تاریخچه‌ی آن‌ها بیابید. در Stanton و White [۱۹۸۶]، الگوریتم‌های بیش‌تری درباره‌ی دنباله‌ها و مجموعه‌ها و ترکیبیات ارائه شده است. نمونه‌ای از یک الگوی جست‌وجوی دودویی در Manna و Waldinger [۱۹۸۷] آمده است. مسأله‌ی زیردنباله‌ی ناپایدار که در بخش ۶-۲ ارائه گردید، از Mirzaian [۱۹۸۷] گرفته شده است (در این مرجع الگوریتمی خطی برای حل این مسأله وجود دارد). Perl، Itai و Avni [۱۹۷۸] کارایی جست‌وجو با درون‌یابی را در حالت میانگین بررسی کرده‌اند و برخی نتایج عملی آن نیز در van der Nat [۱۹۷۹] آمده است.

شاید نخستین بار در سال ۱۹۴۵ von Neumann مرتب‌سازی ادغامی را ساخته باشد و شاید این الگوریتم، از نخستین برنامه‌های ذخیره‌شده‌ای باشد که پیاده‌سازی گردیده‌اند. یک نسخه‌ی «درجا» از

۴- نمونه‌ای که این موضوع را به طور غیرمنتظره‌ای برجسته کرد، زمانی بود که در دوم نوامبر ۱۹۸۸ یک ویروس (یا کرم) به بیش از ۶۰۰۰ رایانه در سرتاسر ایالات متحده حمله کرد و کار آن‌ها را به گونه‌ی چشم‌گیری کند ساخت. بخشی از علت کند شدن رایانه‌ها، الگوریتم جست‌وجوی خطی این ویروس بود (Spafford [۱۹۸۸] را ببینید).

این مرتب‌سازی، نخستین بار از سوی Kronrod [۱۹۶۹] ارائه شد؛ Huang و Langston [۱۹۸۸] و Dvorak و Durian [۱۹۸۸] را نیز ببینید. مرتب‌سازی سریع ازHoare [۱۹۶۲] است و بحث مفصلی درباره‌ی آن در Sedgewick [۱۹۷۸] آمده است. مرتب‌سازی هرمی را Williams [۱۹۶۴] ارائه کرد. گروه گرافیک رایانه‌ای دانشگاه Toronto [۱۹۸۱] فیلمی معرکه با پویانمایی زیبا برای آشنایی با ۹ روش اصلی مرتب‌سازی ساخته است. با وجود این که در چند سال گذشته، مرتب‌سازی مورد مطالعه گسترده‌ای قرار گرفته است، اما هنوز مسأله‌های باز و حل‌نشده‌ی بسیاری در این زمینه وجود دارد. هنوز تعداد دقیق مقایسه‌های لازم برای مرتب‌سازی n عدد روشن نشده است. الگوریتم طرح‌شده در تمرین ۶-۳۰ از Ford و Johnson [۱۹۵۹] است. زمانی، این الگوریتم به دلیل کم بودن تعداد مقایسه‌ها عالی بود، اما Manacher [۱۹۷۹] ثابت کرد که این الگوریتم، بهینه نیست. یکی دیگر از روش‌های مرتب‌سازی پرکاربرد Shellsort است که Shell [۱۹۵۹] آن را ابداع کرد. هم الگوریتم این روش ساده است و هم پیاده‌سازی آن، اما هنوز پیچیدگی آن شناخته نشده است؛ برای دیدن برخی نتایج و مشاهدات تجربی به Incerpi و Sedgewick [۱۹۸۷] مراجعه کنید. به یاری درخت‌های تصمیم‌گیری، حد پایین در چندین مسأله‌ی پایه‌ای یافته شده است؛ Moret [۱۹۸۲] نیز بررسی جامعی درباره‌ی این درخت‌ها انجام داده است.

Floyd و Rivest [۱۹۷۵] تحلیلی برای گونه‌ی مبتنی بر احتمال الگوریتم گزینش ارائه کرده‌اند. نخستین بار Blum، Floyd، Pratt، Rivest و Tarjan [۱۹۷۲] الگوریتمی خطی و قطعی برای مرتبه‌ی آماری طراحی کردند، هرچند مقدار ثابت آن بسیار بزرگ و زمان اجرای آن در واقع از O(n) است. Schönhage، Paterson و Pippenger [۱۹۷۶] الگوریتمی برای یافتن میانه ارائه کرده‌اند که حداکثر به ۳n مقایسه نیاز دارند. بهترین حد پایین شناخته‌شده (بنا به تعداد مقایسه‌ها) برای یافتن میانه ۲n است (به Bent و John [۱۹۸۵] مراجعه کنید). این مقاله نتیجه‌های مسأله‌ی مرتبه‌ی آماری را در حالت کلی در بر دارد؛ البته عبارت‌هایی که حدهای پایین را در حالت کلی نشان دهند، بسیار پیچیده‌اند.

از آنجا که فشرده‌سازی داده‌ها اهمیت بسیاری دارد، مورد مطالعه‌ی گسترده‌ای قرار گرفته است. الگوریتم بخش ۶-۶ به Huffman [۱۹۵۲] برمی‌گردد (Knuth [۱۹۷۳a] را نیز ببینید). Knuth [۱۹۸۵] و Vitter [۱۹۸۵] گونه‌هایی از الگوریتم هافمن را توصیف کرده‌اند که تنها با یک گذر انجام می‌شوند. Ziv و Lempel [۱۹۸۷] یک الگوریتم رایج و کارآمد دیگر برای این کار مطرح کرده‌اند. درباره‌ی فشرده‌سازی داده‌ها مطالب بیش‌تری نیز در Lynch [۱۹۸۵] یافت می‌شود.

الگوریتم تطابق رشته‌ای، ارائه‌شده در بخش ۶-۷، از knuth، Morris و Pratt [۱۹۷۷] و نیز از Boyer و Moore [۱۹۷۷] است. Galil [۱۹۷۹] بدترین حالت زمان اجرای الگوریتم Boyer-Moore را بهبود بخشیده است. درباره‌ی پیچیدگی این الگوریتم مطالب بیش‌تری در Guibas و Odlyzko [۱۹۸۰] و نیز در Schaback [۱۹۸۸] یافت می‌شود. چندین مقایسه‌ی عملی بین شماری از الگوریتم‌های تطبیق رشته‌ای گوناگون در Smit [۱۹۸۲] آمده است. Karp و Rabin [۱۹۸۷] یک

الگوریتم تطبیق رشته‌ای مبتنی بر احتمال ساخته‌اند. این الگوریتم از ایده‌ی «انگشت‌نگاری» بهره می‌گیرد و رشته‌های بزرگ را در قالبی کوچک ارائه می‌کند تا مقایسه‌ی آن‌ها کارآمدتر شود. می‌توان از این الگوریتم در الگوهای دوبعدی نیز سود جست. مسأله‌ی تطبیق رشته‌ای را می‌توان گسترش داد و الگوهایی برای حل مسأله‌هایی پیچیده‌تر از رشته پیدا کرد؛ برای نمونه «کلیدهای عمومی یا جای‌گزین» که روشی سودمند است و می‌توانیم به کمک آن‌ها به دنبال رخدادهای رشته‌هایی در قالب B*C بگردیم (که در اینجا، B و C رشته‌هایی مشخص و * بیانگر وجود هر رشته‌ی ممکن است). مسأله‌ی کلی‌تر، جست‌وجوی هر مجموعه‌ی قابل توصیف با «عبارت منظم» است. (عبارت منظم یا regular expression، روشی برای توصیف مجموعه‌هایی از رشته‌ها به دست می‌دهد و در بیش‌تر دانشگاه‌های ایران در درس نظریه‌ی زبان‌ها و ماشین‌ها بررسی می‌شود - مترجمان) برای آشنایی با نکات بیش‌تری درباره‌ی این موضوع، Aho و Corasick [۱۹۷۵] را ببینید. دیگر مسأله‌ی مهم، جست‌وجوی رشته‌ها در متنی «پیش‌پردازش‌شده» و ثابت است. درخت‌های پسوندی (Weiner [۱۹۷۳] و Mc Creight [۱۹۷۶]) و آرایه‌های پسوندی (Manber و Myres [۱۹۹۰]) امکان جست‌وجوی سریع را فراهم می‌کنند.

مقایسه‌ی دنباله‌ها و کاربردهای فراوانشان در کتابی که Sankoff و Kruskal [۱۹۷۳] آن را ویرایش کرده‌اند، بررسی شده است. در کتابی ویراسته‌ی Apostolico و Galil [۱۹۸۵] مسائل گوناگونی آمده است که رشته‌ها را نیز در بر دارد. الگوریتم بخش ۶-۸ از Wagner و Fischer [۱۹۷۴] است. می‌توان این الگوریتم را از جنبه‌های گوناگونی بهبود بخشید: از نظر صرفه‌جویی در حافظه (Hirschberg [۱۹۷۵])، از نظر زمان اجرا در صورت این که الفبایش خیلی بزرگ باشد (Hunt و Szymanski [۱۹۷۷]) و در حالتی که دنباله‌ها به هم نزدیک باشند (Ukkonen [۱۹۸۵] و Myres [۱۹۸۶]). Hirschberg [۱۹۷۳] بررسی جامعی درباره‌ی نتایج مرتبط با این موضوع انجام داده است.

الگوریتم مبتنی بر احتمالی که یک عنصر را در نیمه‌ی بالا می‌یابد از Yao [۱۹۷۷] است. روش تولید اعداد تصادفی با جزئیات کامل در Knuth [۱۹۸۱] شرح داده شده است. الگوریتم مبتنی بر احتمالی که برای رنگ‌آمیزی در بخش ۶-۹-۲ ارائه شده، بر پایه‌ی یک «اثبات وجودِ مبتنی بر احتمال» در Bollobás [۱۹۸۶] است. شیوه‌ای که در بخش ۶-۹-۳ برای تبدیل الگوریتم‌های مبتنی بر احتمال به الگوریتم‌های قطعی توضیح داده شده، از Raghavan [۱۹۸۶] است. K. Pruhs، بهره‌گیری از این شیوه را برای حل مسأله‌ی رنگ‌آمیزی بخش ۶-۹-۲ به نویسنده گوش‌زد کرد. مسأله‌ی یافتن یک رنگ‌آمیزی معتبر برای زیرمجموعه‌هایی با اندازه‌هایی دل‌خواه، در حالت کلی، یک مسأله‌ی NP-تمام است (Lovász [۱۹۷۳]). (NP-تمام در فصل ۱۱ کتاب بررسی شده است - مترجمان) Erdös و Spencer [۱۹۷۴] نمونه‌های بسیاری از روش‌های مبتنی بر احتمال برای اثبات ویژگی‌های ترکیبیاتی ارائه کرده‌اند.

مسأله‌ی یافتن اکثریت، پیش‌تر نیز بررسی شده بود؛ برای نمونه Misra و Gries [۱۹۸۲]. Fischer و Salzberg [۱۹۸۲] با کمک یک ساختمان داده‌ای پیچیده‌تر، نسبت به آنچه در بخش ۶-۱۰ ارائه گردید، نشان داده‌اند که تعداد مقایسه‌ها (و نه تعداد دیگر گام‌ها) را می‌توان در بدترین حالت به 3n/2+1 کاهش داد و ثابت کرده‌اند که این حد، بهینه است.

Gries [۱۹۸۱] یک توصیف عالی برای حل مسأله‌ی بلندترین زیردنباله‌ی افزایشی ارائه کرده است (که در این کتاب از آن بسیار سود جستیم). Erdös و Szekeres [۱۹۳۵] با به کاربردن «اصل لانه‌ی کبوتر» به گونه‌ای دقیق و زیبا ثابت کرده‌اند هر دنباله از عناصر متمایز به طول n^2+1 باید یک زیردنباله‌ی افزایشی یا کاهشی به طول n+1 داشته باشد. مسأله‌ی یافتن نخستین و دومین عنصر از نظر بزرگی در یک مجموعه، نخستین بار، برای رقابت‌های تنیس و از سوی Lewis Carroll پیش‌نهاد شده است Knuth [۱۹۷۳b] را ببینید). Dobkin و Munro [۱۹۸۰] الگوریتم دیگری برای یافتن مد ارائه کرده‌اند (Gonnet [۱۹۸۴] را هم ببینید).

حل تمرین ۶-۲۷ در Aho، Hopcroft وUllman [۱۹۷۴] شرح داده شده و تمرین ۶-۳۴ از karp، Saks و Wigderson [۱۹۸۶] گرفته شده است. یک راه‌حل برای تمرین ۶-۳۹ در Rodeh [۱۹۸۲] وجود دارد. موضوع تمرین ۶-۲۴ در Choueka، Fraenkel، Klein و Perl [۱۹۸۵] بررسی شده است. ایده‌ی دنباله‌های دست‌یافتنی از Ryser [۱۹۵۷] است.

تمرین‌های آموزشی

۶-۱ راه‌بردی مناسب برای این بازی مشهور طراحی کنید: در این بازی، یک نفر عددی دل‌خواه در بازه‌ی ۱ تا n را پیش خودش در نظر می‌گیرد و دیگری می‌کوشد تا با پرسیدن پرسش‌هایی در قالب «آیا از x کوچک‌تر (بزرگ‌تر) است؟» آن را بیابد. هدف، کم کردن پرسش‌ها تا جای ممکن است. (فرض کنید هیچ یک از بازیگران تقلب نخواهند کرد.)

۶-۲ روشی برای بازی «عدد را حدس بزن» (تمرین ۶-۱) بیابید که در آن محدوده‌ی عدد برگزیده‌شده نامعلوم است؛ یعنی این عدد می‌تواند هر عدد مثبتی باشد.

۶-۳ فرض کنید برنامه‌ای دارید که متن‌های بزرگ را مدیریت می‌کند؛ مانند یک برنامه‌ی واژه‌پرداز. این برنامه، متنی را به صورت دنباله‌ای از کاراکترها در ورودی می‌گیرد و خروجی متناسب با آن را می‌سازد. ناگهان، برنامه با اشکالی روبه‌رو می‌شود که نمی‌توانید آن را برطرف کنید. بدتر آن که نه می‌توانید بفهمید اشکال چیست و نه می‌توانید جای آن را مشخص کنید. به عبارت دیگر، تنها واکنش برنامه، توقف و درج عبارت «خطا» در خروجی است. فرض کنید این اشکال به سبب وجود رشته‌ای خاص در متن باشد که برنامه به دلیل ناشناخته‌ای آن را دوست ندارد. این اشکال

به کل متنی که رشته‌ی اشکال‌زا در آن آمده است، وابسته نیست. راه‌بردی برای یافتن رشته‌ای که سبب بروز خطاست، پیش‌نهاد کنید.

۶-۴ یک ورودی مثال بزنید که در آن جست‌وجو با درون‌یابی، برای یافتن یک عنصر در جدولی به اندازه‌ی n، نیازمند $\Omega(n)$ مقایسه باشد.

۶-۵ مرتب‌سازی تعویض مرتبه را کامل کنید. ورودی این برنامه، دنباله‌ای از n عدد صحیح kرقمی است. مقدار هر رقم ممکن است از ۱ تا m باشد. فرض کنید به فضایی از $O(m)$ دست‌رسی دارید. نخست، برنامه را به صورت روالی بازگشتی بنویسید و مشخص کنید چه مقدار فضای اضافی برای این روال بازگشتی مورد نیاز است. سپس، برنامه‌ای غیربازگشتی طراحی کنید و بکوشید مقدار فضای اضافیِ مورد استفاده‌ی آن را کمینه سازید.

۶-۶ برنامه‌های کامل مرتب‌سازی درجی (هم با جست‌وجوی خطی و هم با جست‌وجوی دودویی) و مرتب‌سازی با انتخاب را بنویسید.

۶-۷ تعداد مقایسه‌ها را برای مرتب‌سازی ورودی شکل ۶-۸ (با مرتب‌سازی ادغامی) و برای ورودی شکل ۶-۱۱ (با مرتب‌سازی سریع) بشمارید. تعداد مقایسه‌ها را برای مرتب‌سازی همین ورودی‌ها به کمک مرتب‌سازی درجی و مرتب‌سازی با انتخاب حساب کنید.

۶-۸ به یاری یک قانون ثابت حلقه، ثابت کنید وجود نخستین دستور if، در الگوریتم Mergesort (شکل ۶-۷) لازم نیست؛ یعنی ثابت کنید اگر این دستور if را برداریم و الگوریتم را با "if Left≠Right" آغاز کنیم، در نتیجه‌ی الگوریتم تغییری رخ نمی‌دهد.

۶-۹ مرتب‌سازی ادغامی را با راه‌حل مسأله‌ی نمای افقی (فصل ۵) مقایسه کنید. شباهت‌های این دو را به طور رسمی ارائه دهید. اگر راه‌حل یکی از این مسأله‌ها را به صورت یک «جعبه‌ی سیاه» در اختیار داشته باشیم، آیا می‌توانیم به یاری آن مسأله‌ی دیگر را نیز حل کنیم؟

۶-۱۰ یک قانون ثابت حلقه‌ی مناسب برای حلقه‌ی اصلی الگوریتم Partition (شکل ۶-۹) بنویسید و برهانی برای درستی آن ارائه کنید.

۶-۱۱ یک ورودی مثال بزنید که مرتب‌سازی سریع هنگام کار روی آن حتماً نیازمند $\Omega(n^2)$ مقایسه باشد. محور این مرتب‌سازی را میانه‌ی عنصرهای نخست، میانی و پایانی دنباله بگیرید.

۶-۱۲ در برخی موارد، ورودی یک الگوریتم مرتب‌سازی، از پیش تقریباً مرتب‌شده است؛ یعنی تنها تعداد اندکی از عناصر در جای درست خود قرار ندارند. عمل‌کرد الگوریتم‌های گوناگون مرتب‌سازی (بخش ۶-۴) را برای چنین ورودی‌هایی شرح دهید. در چنین مواردی کدام الگوریتم را برمی‌گزینید؟ (سفارش می‌کنیم خودتان الگوریتمی برای این کار طراحی کنید.)

۶-۱۳ جدولی مانند شکل ۶-۱۵ برای ساخت یک هرم، با روش بالا به پایین، بسازید.

۶-۱۴ الگوریتمی با رویکرد تقسیم‌وحل طراحی کنید که کم‌ترین و بیش‌ترین عنصر یک مجموعه را بیابد. این الگوریتم باید حداکثر 3n/2 مقایسه (برای $n = 2^k$) انجام دهد. آیا می‌توانید دقیقاً نشان دهید که چرا این الگوریتم به تعداد مقایسه‌هایی کم‌تر از رویکرد سرراست (یعنی 2n-3) نیاز دارد؟

۶-۱۵ درخت هافمن را برای حروف الفبای این تمرین بسازید. همه‌ی حرف‌ها را در نظر بگیرید. درخت هافمن در مقایسه با بهترین «کد با طول ثابت» چه تعداد بیت در مصرف حافظه صرفه‌جویی می‌کند؟

۶-۱۶ جدول next (بخش ۶-۷) را برای رشته‌ی aabbaababababbaabbaabb بسازید.

۶-۱۷ با بهره‌گیری از الگوریتم Minimum_Edit_Distance (شکل ۶-۲۷) ماتریس‌های C و M را بر اساس مقایسه‌ی دنباله‌ی aabccbbaabca با رشته‌ی baacbabaccaba بسازید.

۶-۱۸ قانون ثابت حلقه‌ی مناسبی برای نخستین حلقه‌ی الگوریتم Majority (شکل ۶-۲۸) بنویسید و سپس درستی نخستین مرحله‌ی این الگوریتم را ثابت کنید.

تمرین‌های خلاقانه

اندازه‌ی دنباله‌ها و مجموعه‌ها n بوده، عناصرشان اعداد حقیقی هستند، مگر آن که به صراحت خلاف آن گفته شود. اگر زمان اجرای الگوریتمی از $O(n)$ باشد، به آن خطی می‌گوییم و منظور از زمان اجرا، زمان اجرا در بدترین حالت است.

۶-۱۹ آرایه‌ی A[1..n] از اعداد صحیح داده شده است، چنان که برای هر i در فاصله‌ی [1,n] داریم: $|A[i] - A[i+1]| \le 1$. فرض کنید: $A[1] = x$ و $A[n] = y$ به گونه‌ای که $x < y$. الگوریتمی کارآمد برای جست‌وجو طراحی کنید که j را چنان بیابد که A[j]=z و z در فاصله [x,y] باشد. بیش‌ترین تعداد مقایسه‌هایی که این الگوریتم با z انجام می‌دهد، چقدر است؟

۶-۲۰ با کمک درخت‌های تصمیم‌گیری ثابت کنید الگوریتمی که برای تمرین ۶-۱۹ ارائه کردید، در بدترین حالت، بهینه است. (اگر بهینه نیست، الگوریتم را آن قدر بهبود دهید تا بتوانید این مطلب را ثابت کنید.)

۶-۲۱ ورودی، مجموعه S شامل n عدد حقیقی است. الگوریتمی از $O(n)$ برای یافتن یک عدد که در این مجموعه نباشد، طراحی کنید. ثابت کنید حد پایینِ تعداد گام‌های لازم برای حل مسأله از $\Omega(n)$ است.

۶-۲۲ مجموعه‌ی S شامل n عدد حقیقی به همراه عدد حقیقی x داده شده است.

الف- الگوریتمی طراحی کنید که مشخص کند آیا می‌توان دو عدد در S یافت که حاصل جمعشان دقیقاً x شود. زمان اجرای این الگوریتم باید از $O(n\log n)$ باشد.

ب- حال، فرض کنید که عناصر مجموعه S مرتبند. الگوریتمی طراحی کنید که مسأله را در زمانی از $O(n)$ حل کند.

۶-۲۳ مجموعه‌های S_1 و S_2 از اعداد حقیقی به همراه عدد حقیقی x داده شده‌اند. الگوریتمی برای یافتن دو عنصر، یکی از S_1 و دیگری از S_2 بنویسید به گونه‌ای که حاصل جمع آن دو دقیقاً x شود. اگر تعداد کل عناصر دو مجموعه را n بنامیم، زمان اجرای این الگوریتم باید از $O(n\log n)$ باشد.

۶-۲۴ الگوریتمی طراحی کنید که مشخص کند آیا دو مجموعه جداازهم هستند یا نه. اگر m و n تعداد عناصر این دو مجموعه باشند، پیچیدگی الگوریتم را بیابید. حالتی را هم که m از n بسیار کوچک‌تر است، بررسی کنید.

۶-۲۵ الگوریتمی برای محاسبه‌ی اجتماع دو مجموعه - که اندازه‌ی هر دو از $O(n)$ است و به صورت «آرایه‌هایی از عناصر» داده شده‌اند - طراحی کنید. خروجی الگوریتم، باید آرایه‌ای از عناصر متمایز باشد که اجتماع دو مجموعه‌ی ورودی را نشان دهد. پس، هیچ عنصری نباید بیش از یک بار در خروجی دیده شود. زمان اجرای الگوریتم در بدترین حالت باید از $O(n\log n)$ باشد.

۶-۲۶ اگر دنباله‌ی x_1، x_2، ... و x_n از اعداد حقیقی داده شده باشد (n زوج است) الگوریتمی برای افراز ورودی به n/2 زوج چنان طراحی کنید که اگر جمع اعداد هر زوج را با s_1، s_2، ... و $s_{n/2}$ نشان دهیم، بیشینه‌ی این جمع‌ها کم‌ترین مقدار ممکن را داشته باشد.

۶-۲۷ ✯ مرتب‌سازی واژه‌نامه‌ای را چنان تغییر دهید که برای رشته‌هایی با طول متغیر نیز کار کند. به عبارت دیگر، اعداد دقیقاً kرقمی نیستند، بلکه برخی کوتاه‌تر و برخی بلندترند. البته می‌توان اعداد را با افزودن ارقام «پوچ» صفر هم‌طول کرد. الگوریتمی بنویسید که بدون هم‌طول کردن رشته‌ها نیز کار کند و برحسب تعداد کل ارقام، زمان اجرایی خطی داشته باشد.

۶-۲۸ دو دنباله در ورودی داده شده است: دنباله‌ی x_1، x_2، ... و x_n از اعداد صحیح با ترتیبی دل‌خواه و تصادفی و دنباله‌ی a_1، a_2، ... و a_n از اعداد صحیح متمایز ۱ تا n (یعنی این دنباله یک جای‌گشت از اعداد ۱، ۲، ... و n است). هر دو دنباله در قالب آرایه داده شده‌اند. الگوریتمی از $O(n\log n)$ طراحی کنید که دنباله‌ی نخست را برحسب ترتیب گرفته‌شده از دنباله‌ی دوم بچیند. به عبارت دیگر، می‌خواهیم x_iها به جای ترتیب عادی طبق «ترتیب داده‌شده در آرایه دوم» مرتب شوند. برای مثال اگر x_iها به ترتیب ۱۷، ۵، ۱ و ۹ و a_iها به ترتیب ۳، ۲، ۴ و ۱ باشند، خروجی x_iها باید به صورت ۹، ۵، ۱۷ و ۱ باشد (یعنی نخست،

سومین عنصر از نظر کوچکی و ... می‌آید - مترجمان). این الگوریتم باید «درجا» باشد؛ یعنی نمی‌توانید از هیچ آرایه‌ی دیگری بهره گیرید.

۶-۲۹ d دنباله از عناصر در ورودی داریم، به گونه‌ای که هر دنباله از پیش مرتب‌شده و تعداد کل عناصر n تاست. الگوریتمی از $O(n \log d)$ بنویسید که همه‌ی این دنباله‌ها را در قالب یک دنباله‌ی مرتب ادغام کند.

۶-۳۰ توصیفی خلاصه و ناکامل از یک مرتب‌سازی به نام مرتب‌سازی Ford و Johnson چنین است:

۱- n/2 زوج متمایز از عناصری با قالبی دل‌خواه داریم،

۲- عناصر هر زوج را با هم مقایسه می‌کنیم،

۳- به طور بازگشتی، عناصر بزرگ‌تر زوج‌ها را (که تعدادشان n/2 است) مرتب می‌کنیم،

۴- عناصر باقی‌مانده را هم که تعدادشان n/2 است، به ترتیبی به فهرست مرتب‌شده‌ی عناصر بزرگ‌تر می‌افزاییم.

تعداد مقایسه‌های به‌کاررفته در این الگوریتم، تقریباً از هر الگوریتم دیگری کم‌تر است، به این شرط که در گام ۴، عمل افزودن عناصر کوچک‌تر زوج‌ها به شیوه‌ای «مناسب» انجام شود. حالت‌هایی را که در آن‌ها n برابر با ۵، ۶ یا ۸ است، در نظر بگیرید و برای آن‌ها شیوه‌ای مناسب برای افزودن عناصر در گام ۴ پیدا کنید. باید الگوریتم مرتب‌سازیِ به‌دست‌آمده برای این مقادیر n بهینه باشد. (در واقع با این روش، الگوریتمی بهینه برای ورودی‌هایی که کم‌تر از ۱۲ عنصر دارند، به دست خواهید آورد.)

۶-۳۱ ورودی، دنباله‌ای از اعداد صحیح با تعداد زیادی عناصر تکراری است، به گونه‌ای که تعداد اعداد متمایز دنباله از $O(\log n)$ است.

الف- برای این دنباله یک الگوریتم مرتب‌سازی طراحی کنید که تعداد مقایسه‌های آن در بدترین حالت از $O(n \log \log n)$ باشد.

ب- چرا در این حالت، «حد پایین $\Omega(n \log n)$ برای مرتب‌سازی» برقرار نیست؟

۶-۳۲ ثابت کنید جمع ارتفاع گره‌های یک درخت دودویی متوازن با n گره، حداکثر n-1 است. (در روش ذخیره‌ی ضمنی می‌توان هر درخت دودویی متوازن با n گره را در یک آرایه با اندازه‌ی n جا داد.) درختی ارائه کنید که جمع ارتفاع گره‌های آن دقیقاً n-1 باشد.

۶-۳۳ جمع ارتفاع همه‌ی گره‌های یک هرم (بخش ۶-۴-۵) را نیز می‌توان مستقیماً با توجه به این مطلب محاسبه کرد که ارتفاع گره متناظر با مکان i از آرایه‌ای با اندازه‌ی n، حداکثر $\lceil \log_2^{(n-i+1)} \rceil$ است. جمع ارتفاع‌ها را با توجه به این نکته بیابید.

۶-۳۴ ورودی، عدد حقیقی x و هرمی با اندازه‌ی n در قالب یک آرایه است (روشن است که بزرگ‌ترین عنصر در بالای هرم قرار دارد). الگوریتمی بنویسید که مشخص کند آیا kامین

عنصر هرم (از نظر بزرگی) کم‌تر یا مساوی x هست یا نه. زمان اجرای الگوریتمی که طراحی می‌کنید باید از $O(k)$ و مستقل از اندازه‌ی هرم باشد. می‌توانید فضایی به اندازه‌ی $O(k)$ را به کار ببرید. (دقت کنید که یافتن kامین عنصر از نظر بزرگی اهمیت ندارد، بلکه باید رابطه‌ی آن با x را بیابید.)

۶-۳۵ فرض کنید ورودی، دنباله‌ای از اعداد متمایز x_1، x_2، ... و x_n است و هر x_i، وزن مثبت $w(x_i)$ را دارد. W را جمع همه‌ی وزن‌ها بگیرید. مسأله، یافتن یک x_j است، چنان که به ازای مقدار داده‌شده‌ی X $(0 \le X \le W)$ داشته باشیم:

$$\sum_{x_i>x_j} w(x_i) < X \quad و \quad w(x_j) + \sum_{x_i>x_j} w(x_i) \ge X$$

الگوریتمی کارآمد برای حل این مسأله طراحی کنید. (به این مسأله، مسأله‌ی گزینش وزن‌دار می‌گویند و اگر همه‌ی وزن‌ها ۱ باشند، این مسأله، همان مسأله‌ی گزینش معمولی خواهد شد.)

۶-۳۶ فرض کنید A، الگوریتمی بر اساس مقایسه برای یافتن kامین عنصر از نظر بزرگی در بین n عنصر باشد. ثابت کنید با اطلاعاتی که A دارد، می‌تواند تشخیص دهد کدام عناصر از عنصر یافته‌شده بزرگ‌تر و کدام عناصر از آن کوچک‌ترند. (به عبارت دیگر، با این الگوریتم می‌توانید بدون هیچ مقایسه‌ی دیگری، اعضای مجموعه‌ی ورودی را برحسب رابطه‌ی آن‌ها با kامین عنصر از نظر بزرگی به بخش‌هایی تقسیم کنید.)

۶-۳۷ مسأله‌ی یافتن kامین عنصر از نظر بزرگی را در حالتی در نظر بگیرید که هر عنصر، یک خانه‌ی حافظه را پر کند و ما تنها به کوچک‌ترین فضای حافظه علاقه‌مند باشیم. ورودی، دنباله‌ای از عناصر است که هر بار، یکی از آن‌ها در خانه‌ی مشخص C قرار می‌گیرد؛ یعنی، در iامین گام ورودی، x_i در C قرار داده می‌شود (محتویات پیشین C پاک می‌گردد). در بین هر دو گام از ورودی می‌توانید هر محاسبه‌ای را انجام دهید (مثلاً قرار دادن محتویات C در یک محل موقت). هدف، کم کردن تعداد خانه‌های مورد نیاز الگوریتم است. در این الگوریتم، یک حد بالا و یک حد پایین برای تعداد خانه‌های حافظه‌ی مورد نیاز ارائه دهید.

۶-۳۸ هدف این مسأله یافتن آماری kام، یعنی kامین عنصر از نظر کوچکی است و مانند تمرین ۶-۳۷ می‌خواهیم با به کاربردن حافظه‌ای بسیار اندک، زمان اجرا کمینه شود (اما دیگر لازم نیست حافظه‌ی به‌کاررفته کم‌ترین مقدار ممکن باشد). در اینجا نیز دنباله‌ای از عناصر x_1، x_2، ... و x_n یکی‌یکی داده می‌شوند. الگوریتمی با زمان اجرای $O(n)$ طراحی کنید که تنها با بهره‌گیری از $O(k)$ خانه‌ی حافظه، آماری kام را بیابد. مقدار k را از پیش می‌دانیم (پس به الگوریتم می‌توان حافظه‌ی کافی تخصیص داد) اما تا همه‌ی عناصر را نبینیم، مقدار n آشکار نخواهد شد.

۶-۳۹ فرض کنید A و B دو مجموعه‌ی n عنصری در رایانه‌های P و Q باشند. این دو رایانه، هم می‌توانند برای یکدیگر پیام بفرستند و هم خودشان می‌توانند هر نوع محاسبه‌ای را انجام دهند. الگوریتمی برای یافتن آماری nام در اجتماع A و B طراحی کنید (یعنی میانه‌ی اجتماع A و B را بیابید). برای سادگی می‌توانید فرض کنید همه‌ی عناصر متمایزند. اگر تنها یک عنصر یا یک عدد صحیح را بتوان در هر پیام قرار داد، چگونه می‌توان تعداد پیام‌های لازم را کمینه کرد؟ تعداد این پیام‌ها در بدترین حالت چقدر خواهد شد؟

۶-۴۰ با داشتن مجموعه‌ی $S = \{x_1, x_2, ..., x_n\}$ از اعداد صحیح، زیرمجموعه‌ی ناتهی R در آن را ($R \subseteq S$) چنان بیابید که داشته باشیم:

$$\sum_{x_i \in R} x_i \equiv 0 \text{ (پیمانه‌ی n)}$$

(این رابطه، یعنی هم‌نهشت بودن حاصل جمع با صفر به پیمانه‌ی n، نشانگر بخش‌پذیر بودن حاصل‌جمع بر عدد n است - مترجمان)

۶-۴۱ در مسأله‌ی یافتن عنصری در دنباله‌ی x_1، x_2، ... و x_n که $x_i = i$، با بهره‌گیری از ایده‌ی حدنظری ثابت کنید حد پایین تعداد مقایسه‌ها، $\Omega(\log n)$ است. (بخش‌های ۶-۲ و ۶-۴-۶ را ببینید.)

۶-۴۲ فرض کنید می‌خواهید کد هافمن را به کار ببرید، اما زبان برنامه‌نویسی شما امکان دسترسی به بیت‌ها را نمی‌دهد و تنها می‌توانید دنباله‌ی بیت‌ها را به صورت دنباله‌ی از بایت‌ها (یا با توجه به سخت افزار به صورت واحدهایی kبیتی) بخوانید. از آنجا که هر بایت (یا واحد بیتی) متناظر با یک عدد صحیح است، پس هر رشته‌ای در این زبان برنامه‌نویسی با دنباله‌ای از اعداد صحیح (هر یک کوچک‌تر از 2^8) متناظر می‌شود. شیوه‌ای برای تبدیل دنباله‌ی اعداد صحیح پیدا کنید که هرگاه دنباله‌ی بیتی اصلی را خواستید، بتوانید آن را به یاری درخت هافمن بیابید. کار را با ساخت یک جدول $k \times 2^k$ انجام دهید که در آن، k اندازه‌ی هر واحد بیتی (۸، برای بایت) است. جدول، به درخت (که آن را دارید) وابسته است. تنها می‌توانید از اعمال ضرب، جمع و تفریق برای اعداد صحیح کمک بگیرید، اما به کار بردن اعمال بیتی مجاز نیست. این جدول باید به شما امکان دسترسی به هر یک از بیت‌های هر عدد از دنباله را بدهد. مسأله را یک بار دیگر نیز با این فرض که اندازه‌ی جدول 2×2^k است، حل کنید.

۶-۴۳ فرض کنید کد هافمن به یک متن مشخص اعمال شده باشد؛ درخت هافمن نیز ساخته شده، در دسترس شما باشد و بسامد هر کاراکتر متن را هم بدانید. می‌خواهیم هنگام تغییر متن، درخت بهینه را برای متن تغییریافته داشته باشیم. تغییرات متن کند است، به گونه‌ای که هر بار تنها بسامد یک کاراکتر (موجود) یک واحد افزوده می‌شود.

می‌دانیم که در درخت هافمن با نزدیک شدن به ریشه، بسامد گره‌هایی که به ترتیب می‌بینیم، کاهشی نخواهد بود (به عبارت دیگر، در این درخت، محل گرهی با بسامد کم‌تر نمی‌تواند از محل گرهی با بسامد بیش‌تر بالاتر باشد). بسامد یک گره داخلی v، برابر با حاصل جمع بسامدهای گره‌های خارجی پایین‌دست آن گره تعریف می‌شود؛ با توجه به این مطالب، دوستی به شما پیش‌نهاد می‌کند تا با نگاه به یک سطح بالاتر بررسی کنید که آیا هنوز بسامد افزایش‌یافته، ویژگی گفته‌شده را برآورده می‌کند یا نه. اگر گرهی در سطح بالاتر با بسامدی کم‌تر از گره X وجود نداشته باشد، بگذارید X سر جای خود بماند؛ در غیر این صورت، کاراکتر سطح بالاتر را (که بسامد آن از X کم‌تر است) به جای X قرار دهید. گاهی ممکن است این الگوریتم کار کند، اما در حالت کلی چنین نیست. چرا گاهی این الگوریتم نادرست است و چگونه می‌توان آن را درست کرد؟ هم باید علت نارسایی الگوریتم را بگویید و هم باید شرح دهید که چرا در حالت کلی، الگوریتم کار نمی‌کند؛ یعنی یا یک مثال نقض پیدا کنید که این الگوریتم نتواند درخت بهینه‌ی آن را بسازد، یا نشان دهید که اگر الگوریتم درست باشد با یک تناقض (یا برخی استلزامات بسیار مشکوک) روبه‌رو می‌شویم. پس تنها، اشاره به چند حالت که الگوریتم از انجام درست آن‌ها ناتوان است، کافی نیست؛ چراکه می‌توان از این حالت‌ها چشم‌پوشی کرد. بنابراین، باید ثابت کنید الگوریتم واقعاً نادرست است.

۴۴-۶ اگر ورودی، دو رشته‌ی کاراکتری $A = a_1a_2...a_n$ و $B = b_1b_2...b_n$ باشد، الگوریتمی از $O(n)$ طراحی کنید که مشخص سازد آیا B، چرخشی از A هست یا نه؛ به عبارت دیگر، این الگوریتم باید روشن کند که آیا اندیسی مانند k در بازه‌ی [1,n] وجود دارد که به ازای هر i از این بازه a_i برابر $b_{(k+i) \bmod n}$ باشد.

۴۵-۶ می‌توان الگوریتم تطبیق رشته‌ای KMP را بدین ترتیب برای رشته‌های دودویی بهبود بخشید: هنگام ساختن جدول next همراه با بررسی پسوندی از رشته که تا به حال دیده‌اید، کاراکتر تطبیق‌نیافته را نیز به این جدول بیفزایید؛ چراکه در پی بلندترین پسوندی از $B(i-1)\overline{b_i}$ هستیم که با پیشوندی از B تطبیق داشته باشد ($\overline{b_i}$ کاراکتر مکمل b_i است.) در این روش، هر کاراکتر A با هر کاراکتر B دقیقاً یک بار مقایسه می‌شود.

الف- جدول تازه‌ی next را دقیقاً تعریف کرده، آن را برای مثال شکل ۶-۲۱ پر کنید.

ب- الگوریتم تطبیق رشته‌ای را برای بهره‌گیری از این تغییر بازنویسی کنید.

۴۶-۶ الگوریتمی هم‌زمان برای تطبیق رشته‌ای: فرض کنید الگوی ورودی، کاراکتر به کاراکتر و به آرامی وارد برنامه شود (مثلاً از راه صفحه کلید) اما متن اصلی را از پیش داشته باشیم. می‌خواهیم عمل تطبیق را تا جایی که می‌توانیم پیش ببریم، بدون آن که منتظر دریافت همه‌ی الگو شویم. به عبارت دیگر، می‌خواهیم هنگامی که کاراکتر kام وارد می‌شود، برنامه

سرگرم بررسی کاراکتری از متن باشد که نخستین تطبیق با k-1 کاراکتر آغازین الگو از آنجا شروع می‌شود. الگوریتم KMP را برای دست‌یابی به این هدف بازنویسی کنید.

۶-۴۷ الگوریتم تطبیق رشته‌ای KMP را برای یافتن بلندترین پیشوندی از B که با زیررشته‌ای از A تطبیق دارد، تغییر دهید. به عبارت دیگر، لازم نیست در A دنبال تطبیقی از همه‌ی B باشید؛ بلکه کافی است بلندترین تطبیق را (که از b_1 شروع می‌شود) بیابید.

۶-۴۸ T و P را دنباله‌های کاراکتری t_1، t_2، ... و t_n و p_1، p_2، ... و p_k بگیرید، به گونه‌ای که $k \le n$. الگوریتمی از $O(n)$ طراحی کنید که روشن سازد آیا P زیردنباله‌ای از T هست یا نه. (P را زیردنباله‌ی T گوییم، اگر دنباله‌ای از اندیس‌های $1 \le i_1 < i_2 < \ldots < i_k \le n$ وجود داشته باشند که برای همه‌ی jهای بازه‌ی [1,k]، t_{i_j} برابر p_j باشد.)

۶-۴۹ الگوریتمی برای تمرین ۶-۴۸ بسازید که اگر چندین زیردنباله از T با P برابر باشند، زیردنباله‌ای از اندیس‌های $1 \le i_1 < i_2 < \ldots < i_k \le n$ را بیابد که هم به ازای هر j از بازه‌ی [1,k]، t_{i_j} با p_j برابر باشد و هم $\sum_{j=1}^{k} i_j$ بیشینه گردد.

۶-۵۰ تمرین ۶-۴۸ را دوباره در نظر بگیرید، اما فرض کنید iامین کاراکتر از T، هزینه‌ی c(i) داشته باشد که c(i)>0. زیردنباله‌ی تطبیق‌یافته‌ای را پیدا کنید که جمع کل هزینه‌هایش بیشینه شود؛ یعنی دنباله‌ای از اندیس‌های $1 \le i_1 < i_2 < \ldots < i_k \le n$ را بیابید که هم به ازای هر j از بازه‌ی [1,k]، t_{i_j} با p_j برابر باشد و هم $\sum_{j=1}^{k} c(i_j)$ بیشینه گردد.

۶-۵۱ بلندترین زیردنباله‌ی مشترک دو دنباله را LCS و کوتاه‌ترین ابردنباله‌ی مشترک آن‌ها را (یعنی کوتاه‌ترین دنباله‌ای که هر دو دنباله، زیردنباله‌ای از آن هستند) SCS می‌گویند.

الف- الگوریتمی کارآمد طراحی کنید که در دو دنباله‌ی داده‌شده، LCS و SCS را بیابد.

ب- فرض کنید d(T,P) کوتاه‌ترین فاصله‌ی ویرایشی بین دنباله‌های T و P، در حالتی باشد که امکان جای‌گزینی وجود ندارد (یعنی برای یک جای‌گزینی باید یک درج و یک حذف انجام شود). ثابت کنید: $d(T,P) = |SCS(T,P)| - |LCS(T,P)|$ (که در آن، منظور از $|k|$، طول دنباله‌ی k است).

۶-۵۲ مسأله‌ی کوتاه‌ترین فاصله ویرایشی (بخش ۶-۸) را به حالتی تعمیم دهید که در آن اعمال درج در آغاز یا پایان یکی از دنباله‌ها شمارش نمی‌شوند؛ یعنی اگر B در A وجود داشته باشد، اعمال درج لازم برای گسترش B به حساب نمی‌آیند و تنها، فاصله‌ی ویرایشی B را از زیردنباله‌ای از A که با آن تطبیق دارد، در نظر می‌گیریم. (توجه: اگر به آغاز B، بدون هزینه، کاراکترهایی را بیفزایید، باید اعمال درج پایان A را به حساب آورید و برعکس.)

۶-۵۳ می‌توان مسأله‌ی مقایسه‌ی دو دنباله را به سه (یا بیش از سه) دنباله تعمیم داد؛ بدین ترتیب که در هر گام می‌توانیم عمل درج، حذف یا جای‌گزینی کاراکترها را روی هر یک از دنباله‌ها انجام دهیم. اگر کاراکترهای متناظر در همه‌ی دنباله‌ها یکسان باشند، هزینه‌ی آن گام ۰، و گرنه ۱ خواهد بود (حتا اگر دو دنباله با هم تطبیق یابند و تنها یک عمل درج یا حذف لازم شود). برای نمونه فرض کنید دنباله‌ها aabb، bbb و cbb باشند. یک دنباله‌ی ویرایشی ممکن، افزودن a به آغاز bbb و cbb (با هزینه‌ی ۱)، جای‌گزینی یک b در bbb و یک c در cbb با یک a است. پس از این دو کار تطابق صورت می‌گیرد و کل هزینه‌ی ویرایش ۲ خواهد بود. الگوریتمی از $O(n^3)$ برای یافتن فاصله‌ی ویرایشی کمینه، برای سه دنباله‌ی ورودی بنویسید.

۶-۵۴ $A = a_1a_2....a_n$ و $B = b_1b_2...b_m$ را دو رشته‌ی کاراکتری بگیرید و رشته‌ی $a_ia_{i+1}...a_n$ را (که iامین پسوند A هم گفته می‌شود) با $A[i]$ نشان دهید. d_i را نیز فاصله‌ی ویرایشی کمینه بین B و $A[i]$ فرض کنید. الگوریتمی از $O(n^2)$ برای پیدا کردن کم‌ترین مقدار d_i (در بین همه‌ی iهای بازه‌ی [1,n]) طراحی کنید.

۶-۵۵ ورودی، دنباله‌ی اعداد x_1، x_2 ... و x_n است. ثابت کنید هر الگوریتم قطعی برای گزینش یک عدد از نیمه‌ی بالایی این مجموعه (یعنی عددی بزرگ‌تر یا مساوی میانه) دست‌کم به $\lfloor n/2 \rfloor$ مقایسه نیازمند است.

۶-۵۶ در بخش ۶-۹-۲ الگوریتمی مبتنی بر احتمال برای رنگ‌آمیزی گفته شد. تعداد گام‌های مورد انتظار این الگوریتم را برحسب k و r مشخص کنید.

۶-۵۷ k را عددی نابزرگ‌تر از n بگیرید و فرض کنید روالی برای تولید اعداد تصادفی در محدوده‌ی ۱ تا k دارید. الگوریتمی برای تولید یک جای‌گشت تصادفی از n عدد ارائه کنید که احتمال تولید همه‌ی جای‌گشت‌های ممکن با یکدیگر برابر باشد.

۶-۵۸ نظرسنجی، نمونه‌ای از الگوریتم‌های مبتنی بر احتمال است. فرض کنید ۲ نامزد و n رأی‌دهنده وجود دارند. الگوریتمی رایج، پرسش از k رأی‌دهنده به تصادف و گرفتن میانگین پاسخ‌هاست. اگر دقیقاً نیمی از رأی‌دهندگان، هوادار یکی از دو نامزد باشند، احتمال آن که نتیجه‌ی نظرسنجی (با k رأی‌دهنده) در محدوده‌ی ۴۵ تا ۵۵ درصد باشد، چقدر است؟ (عبارتی که به عنوان پاسخ ارائه می‌دهید، علاوه بر مقدارهای ثابت، تنها می‌تواند پارامترهای n و k را در خود داشته باشد.)

۶-۵۹ نتایج نظرسنجی‌ها را معمولاً همراه با محدوده‌ی «خطا» ارائه می‌کنند؛ برای مثال، می‌گویند درصد آرای نامزد X، x با حاشیه‌ی خطای ± 3 درصد است. توضیح دهید چرا بیان خطا به صورت درصد به اندازه‌ی بیان آن به صورت مطلق، دقیق نیست؟ راه دقیق تعریف خطا چیست؟

۶-۶۰ هدف این تمرین، مقایسه‌ی الگوریتم‌های Monte Carlo با الگوریتم‌های Las Vegas است. به طور خلاصه، می‌توان گفت الگوریتم‌های Monte Carlo تنها زمان اجرای الگوریتم را تضمین می‌کنند، نه درستی آن را. فرض کنید با یک مسأله‌ی تصمیم‌گیری روبه‌رو هستیم، پس پاسخ «بله» یا «خیر» است. احتمال خطای الگوریتم Monte Carlo را حداکثر ¼ بگیرید. (همین مقدار خطا مناسب است؛ چراکه می‌توانیم به سادگی با چندین و چند بار اجرای الگوریتم، اکثریت خروجی را پاسخ الگوریتم در نظر بگیریم و به این ترتیب، احتمال خطا را به میزان چشم‌گیری کاهش دهیم.) کدام یک از این دو دسته الگوریتم بهتر است؟ به عبارت دیگر، آیا امکان دارد یکی از این دو دسته الگوریتم را به دیگری تبدیل کرد؟

(فرض کنید برای مسأله‌ای دو نوع راه‌حل A و B داشته باشیم. تا به حال، منظور از بهتر بودن یک راه‌حل، زمان اجرای کوتاه‌تر یا حافظه‌ی مورد نیاز کم‌تر بوده است، اما در اینجا منظور از بهتر بودن راه‌حل نوع A از راه‌حل نوع B این است که بتوان از روی راه‌حل نوع A، راه‌حلی از نوع B برای مسأله ساخت. منظور نویسنده این است که اگر ضریب خطا کم‌تر از ½ باشد، الگوریتم‌های Monte Carlo و Las Vegas را با یکدیگر مقایسه کنید - مترجمان)

۶-۶۱ اگر فهرستی از n عنصر داشته باشید، الگوریتمی برای یافتن عناصری که بیش از n/4 بار در فهرست ظاهر شده‌اند، طراحی کنید. تعداد مقایسه‌های این الگوریتم باید از $O(n)$ باشد. (راه‌نمایی: این کار با تغییر الگوریتم «یافتن اکثریت» انجام‌پذیر است.)

۶-۶۲ از شما خواسته شده است که یک رقابت دوره‌ای تنیس را برنامه‌ریزی کنید. تعداد بازی‌کنان $n = 2^k$ است. هر بازی‌کن باید با همه‌ی بازی‌کنان دیگر بازی کند. هر بازی‌کن هر روز یک بار بازی می‌کند و باید تمام بازی‌های خود را در n-1 روز انجام دهد. بازی‌کنان را با P_1، P_2، ... و P_n نشان می‌دهیم. برنامه‌ی بازی‌های هر بازی‌کن را ارائه دهید. (راه‌نمایی: روش تقسیم‌وحل را این گونه به کار ببرید: نخست، بازی‌کنان را به دو گروه مساوی تقسیم کنید و برنامه‌ی بازی‌های هر گروه را در n/2-1 روز نخست پیدا کنید. سپس بازی‌های بین دو گروه را در n/2 روز بعد تنظیم کنید.)

۶-۶۳ ✯ الگوریتمی برای برگزاری یک رقابت دوره‌ای تنیس با تعداد دل‌خواهی بازی‌کن بنویسید (تمرین ۶-۶۲ را ببینید). اگر تعداد بازی‌کنان فرد باشد، آنگاه در هر دور، یک بازی‌کن استراحت می‌کند.

۶-۶۴ ✯ r_1، r_2، ... و r_n و c_1، c_2، ... و c_n را دو دنباله از اعداد صحیح با مجموعی برابر بگیرید؛ یعنی: $\sum_{i=1}^{n} r_i = \sum_{i=1}^{n} c_i$. چنین دنباله‌هایی را هنگامی دست‌یافتنی گوییم که ماتریسی $n \times n$ از عناصر ۰ یا ۱ وجود داشته باشد و به ازای هر i، جمع عناصر سطر i، دقیقاً r_i و جمع عناصر ستون i، دقیقاً c_i شود. البته، همه‌ی دنباله‌ها دست‌یافتنی نیستند؛ مثلاً دو دنباله‌ی

«۰،۲» و «۰،۲» دست‌یافتنی نیستند، چراکه تنها دومین عنصر از دومین سطر می‌تواند ناصفر باشد و چون این عنصر نباید از ۱ بیش‌تر شود؛ پس این دو دنباله دست‌یافتنی نیستند. الگوریتمی برای تشخیص دست‌یافتنی بودن یا دست‌یافتنی نبودن هر دو دنباله‌ی دل‌خواه طراحی کنید و اگر دو دنباله دست‌یافتنی بودند، ماتریس متناظر با آن‌ها را نیز بسازید. (راه‌نمایی: در استقرایی که به کار می‌برید، نخست، فرض را برای گسترش مسأله به ماتریس‌های $n \times m$ تقویت کنید و آنگاه استقرا روی n (تعداد سطرها) را به کار گیرید. بکوشید محل ۱های سطر نخست را به گونه‌ای تعیین کنید که اگر و تنها اگر مسأله‌ی اصلی قابل‌حل باشد، مسأله برای n-1 سطر دیگر نیز حل شود.)

فصل ۷

الگوریتم‌های گراف

راه میان‌بر، طولانی‌ترین راه بین دو نقطه است.
ناشناس

۷-۱ آشنایی

در فصل پیش، الگوریتم‌هایی را برای کار با مجموعه‌ها یا دنباله‌هایی از اشیاء بررسی کردیم و به رابطه‌هایی مانند ترتیب، تکرار و هم‌پوشانی برخوردیم. در این فصل، با رابطه‌های بیش‌تری بین اشیاء آشنا خواهیم شد و گراف‌ها را برای مدل‌سازی این رابطه‌ها به کار خواهیم گرفت. گراف‌ها می‌توانند وضعیت‌های بسیار گوناگونی را مدل کنند و در زمینه‌های بسیاری از باستان‌شناسی گرفته تا روان‌شناسی اجتماعی کاربرد دارند. در این فصل، برای کار با گراف‌ها و محاسبه‌ی ویژگی‌های مشخصی از آن‌ها، چندین الگوریتم مهم و پایه‌ای را به شما معرفی خواهیم کرد.

نخست، به نمونه‌هایی از مدل‌سازی با گراف نگاهی می‌اندازیم:

۱- یافتن مسیری خوب از خانه تا رستورانی در شهر؛ خیابان‌ها متناظر با یال‌ها (اگر خیابان‌ها یک‌طرفه باشند، یال‌های جهت‌دار) و تقاطع خیابان‌ها متناظر با رأس‌ها هستند. برای هر رأس و هر یال (هر تکه از خیابان که بین دو تقاطع است) زمان تأخیری مورد انتظار است و مسأله، یافتن «سریع‌ترین» مسیر بین دو رأس است.

۲- برخی برنامه‌های رایانه‌ای را می‌توان به وضعیت‌های گوناگون تقسیم کرد. ممکن است در هر یک از این وضعیت‌ها، حالت‌های مختلفی برای پیش‌روی وجود داشته باشد و برخی از این وضعیت‌ها نیز ممکن است نامطلوب باشند. یافتن وضعیت‌هایی که به یک حالت نامطلوب منجر می‌شوند، مسأله‌ی دیگری از نظریه‌ی گراف است که در آن هر وضعیت، متناظر با یک رأس و هر یال متناظر با امکان پیش‌روی از یک وضعیت به وضعیت دیگر خواهد بود.

۳- به مسأله‌ی برنامه‌ریزی کلاس‌های یک دانشگاه می‌توان به صورت مسأله‌ای از نظریه‌ی گراف نگریست. رأس‌ها بیانگر کلاس‌ها هستند و اگر دانش‌آموزی بخواهد دو کلاس را با هم بگیرد، یا استادی بخواهد در هر دو کلاس تدریس کند، آنگاه آن دو کلاس را به یکدیگر

متصل می‌کنیم. مسأله، برنامه‌ریزی کلاس‌هاست، به گونه‌ای که تعداد تداخل کلاس‌ها کمینه گردد. این مسأله دشوار است و به راحتی نمی‌توان راه‌حل مناسبی برای آن یافت.

۴- یک سامانه‌ی رایانه‌ای با چندین حساب کاربری در نظر بگیرید که در آن هر کاربر برای دست‌رسی به حساب خود، مجوز یا امتیازی امنیتی دارد. ممکن است کاربران بخواهند با یکدیگر همکاری کنند و اجازه‌ی دست‌رسی به حساب خود را در اختیار کاربر دیگری هم بگذارند. از سویی، اگر کاربر A به حساب کاربر B و کاربر B به حساب کاربر C دست‌رسی داشته باشد، آنگاه A نیز به حساب کاربر C دست‌رسی خواهد داشت. مسأله‌ی تعیین دست‌رسی کاربران به حساب‌های یکدیگر، مسأله‌ای دیگر از نظریه‌ی گراف است. در اینجا، کاربران با رأس‌ها متناظر می‌شوند و اگر کاربر A اجازه‌ی دست‌رسی به حساب خود را به کاربر B بدهد، یالی جهت‌دار از A به B وجود خواهد داشت.

کتاب‌های درسی بسیاری درباره‌ی نظریه‌ی گراف و کاربردهای فروان آن وجود دارند (مراجع پایان فصل را ببینید).

چند ساختمان‌داده که برای نگه‌داری گراف مناسبند، پیش‌تر در بخش ۴-۶ بررسی شدند. در این کتاب، برای نگه‌داری گراف‌ها، بیش‌تر، لیست همسایگی را به کار می‌گیریم که در گراف‌های تنک یا خلوت (گراف‌های کم‌یال) از دیگر روش‌ها کارآمدتر است. نخست، با واژه‌های رایج آشنا می‌شویم. گراف G=(V,E) از مجموعه‌ی رأس‌های (یا گره‌های) V و مجموعه‌ی یال‌های E تشکیل می‌شود. هر یال، با زوجی متمایز از دو رأس متناظر است. (گاهی طوقه هم، یعنی یالی از یک رأس به خودش مجاز است، ولی ما فرض می‌کنیم به کار بردن طوقه مجاز نباشد.) یک گراف ممکن است «جهت‌دار» یا «بدون‌جهت» باشد. یال‌های گراف جهت‌دار، زوج‌هایی مرتبند؛ یعنی ترتیب دو رأسی که یک یال آن‌ها را به هم متصل می‌کند، بااهمیت است. در گراف جهت‌دار یال را به صورت پیکانی از یک رأس (دم) به رأس دیگر (سر) می‌کشیم. یال‌های گراف بدون‌جهت، زوج‌هایی از رأس‌ها بدون توجه به ترتیب آن‌هاست و آن‌ها را به راحتی با خطی بین دو رأس نشان می‌دهیم. گراف چندگانه گرافی است که در آن بین هر دو زوج از رأس‌ها ممکن است چندین یال وجود داشته باشد (یعنی E در این گراف، یک مجموعه‌ی چندگانه است). گاهی به گرافی که چندگانه نیست، گراف ساده می‌گویند. فرض می‌کنیم گراف‌هایی که با آن‌ها کار می‌کنیم، همگی ساده‌اند مگر آن که خلافش گفته شود. برای رأس v، درجه‌ی $d(v)$، تعداد یال‌های متصل به آن است. در گراف‌های جهت‌دار بین درجه‌ی ورودی و درجه‌ی خروجی فرق می‌گذاریم؛ اولی، تعداد یال‌هایی است که به v وارد می‌شوند و دومی، تعداد یال‌هایی است که از v خارج می‌شوند.

مسیری از رأس v_1 به رأس v_k دنباله‌ای از رأس‌های v_1، v_2، ... و v_k است که به ترتیب با یال‌های (v_1,v_2)، (v_2,v_3)، ... و (v_{k-1},v_k) به یکدیگر متصل شده‌اند. (معمولاً یال‌ها نیز بخشی از مسیر در نظر گرفته می‌شوند.) مسیر را ساده گویند، اگر هر رأس حداکثر یک بار در آن دیده شود. اگر

مسیری (بنا به نوع گراف، جهت‌دار یا بدون‌جهت) از رأس v به رأس u وجود داشته باشد، می‌گوییم u از v دست‌رس‌پذیر است (یا v به u دست‌رسی دارد). مدار، مسیری است که رأس‌های آغاز و پایانش یکسانند. مدار را ساده گویند، اگر در آن به جز نخستین رأس (یا همان آخرین رأس) رأس دیگری بیش از یک بار ظاهر نشود. به مدار ساده، دور نیز می‌گویند. (گاهی به مدارهای غیرساده نیز دور می‌گویند، ولی ما فرض می‌کنیم که «دور»ها همگی ساده‌اند.) منظور از شکل بدون‌جهتِ یک گراف جهت‌دار، خود آن گراف است بدون در نظر گرفتن جهت یال‌هایش. گراف را همبند گویند، اگر در شکل بدون‌جهت آن، از هر رأس مسیری به هر رأس دیگر وجود داشته باشد. جنگل، گرافی است که در شکل بدون‌جهت آن دور وجود نداشته باشد. درخت، جنگلی همبند است. درخت ریشه‌دار (که به آن arborescence نیز می‌گویند) درختی جهت‌دار است با یک رأس خاص به نام «ریشه» به گونه‌ای که همه‌ی یال‌ها از آن دور می‌شوند.

زیرگرافی از گراف G=(V,E)، هر گرافی همچون H=(U,F) است چنان که $U \subseteq V$ و $F \subseteq E$. درخت پوشا یا پوششیِ گراف بدون‌جهت G، زیرگرافی از G است که هم درخت باشد و هم تمام رأس‌های G را در بر گیرد. جنگل پوشا یا پوششی گراف بدون‌جهت G، زیرگرافی از G است که هم جنگل باشد و هم تمام رأس‌های G را در بر گیرد. در گراف G=(V,E)، هر زیرگراف القاشده با رأس‌ها، زیرگرافی مانند H=(U,F) است که $U \subseteq V$ و F تمام یال‌هایی از E را در بر گیرد که هر دو رأس آن‌ها متعلق به U باشد. معمولاً به زیرگراف القاشده با رأس‌ها، زیرگراف القایی می‌گویند. اگر گراف G=(V,E) همبند نباشد، می‌توان آن را به طور یکتا به زیرگراف‌هایی همبند افراز کرد. این زیرگراف‌ها را مؤلفه‌های همبند G می‌گویند. یک مؤلفه‌ی همبند G زیرگرافی همبند از آن است، به گونه‌ای که زیرگراف همبند دیگری از G آن را در بر نگرفته باشد؛ به عبارت دیگر، مؤلفه‌های همبند، بزرگ‌ترین زیرگراف‌های همبند هستند. گراف دوبخشی، گرافی است که می‌توان رأس‌هایش را به دو مجموعه تقسیم کرد، به گونه‌ای که هر یال گراف، رأسی از یک مجموعه را به رأسی از مجموعه‌ی دیگر متصل کند. گراف وزن‌دار، گرافی است که یال‌هایش وزن (یا هزینه، یا طول) داشته باشند.

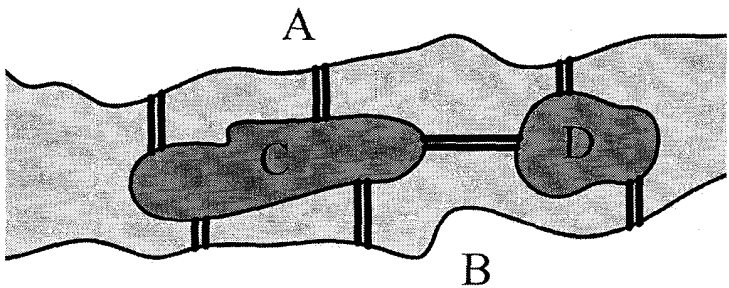
صرف‌نظر از چند مورد روشن، بسیاری از تعریف‌ها در گراف‌های جهت‌دار و گراف‌های بدون‌جهت به یکدیگر شبیه هستند؛ برای مثال، مسیرهای جهت‌دار و مسیرهای بدون‌جهت دقیقاً مانند یکدیگر تعریف می‌شوند، اما روشن است که جهت یال‌ها در مسیرهای جهت‌دار مشخص است. هرگاه درباره‌ی یکی از این دو دسته‌ی کلی گراف‌ها سخن می‌گوییم، نمادی متفاوت برای آن به کار نخواهیم برد. پس، برای مثال، اگر در مبحث گراف‌های جهت‌دار سخنی درباره‌ی مسیرها می‌گوییم، منظورمان مسیرهای جهت‌دار است.

کار را با مثالی ساده آغاز می‌کنیم که نخستین مسأله از نظریه‌ی گراف محسوب می‌شود: عبور از پل‌های شهر Königsberg. سپس درباره‌ی چگونگی کارهایی مانند پیمایش گراف، ترتیب‌دهی به رأس‌های گراف (یعنی یافتن یک رابطه‌ی ترتیب بین رأس‌های گراف - مترجمان)، یافتن کوتاه‌ترین

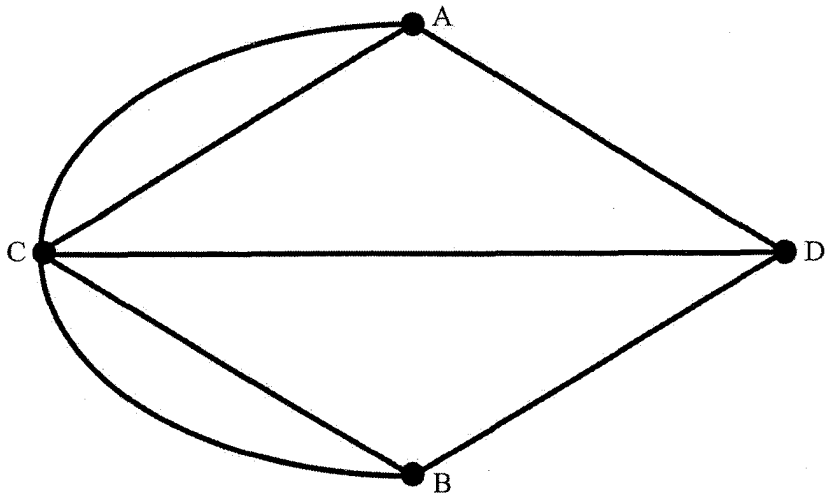
مسیرها در گراف و افراز گراف به بخش‌هایی برای برآوردن ویژگی‌های مشخص بحث می‌کنیم. در فصل ۱۰ نیز مبحثی درباره‌ی رابطه‌ی بین الگوریتم‌های گراف و الگوریتم‌های ماتریس آمده و چند الگوریتم دیگر گراف هم در آنجا ارائه شده است.

۷-۲ گراف‌های اویلری

گراف‌های اویلری از نخستین مسأله‌های طرح و حل‌شده در نظریه‌ی گراف هستند. سال ۱۷۳۶ میلادی، ریاضی‌دان سوئیسی، Leonhard Euler با این معما روبه‌رو شد. شهر Königsberg (که امروزه Kaliningrad نامیده می‌شود) بر دو کرانه‌ی رودخانه‌ی Pregel قرار داشت و دو جزیره‌ی درون رودخانه را نیز در بر می‌گرفت (شکل ۷-۱). بخش‌های مختلف شهر با هفت پل به یکدیگر متصل شده بودند. بسیاری از شهروندان برای حل این معما کوشیده بودند: «آیا می‌توان از جایی در شهر قدم زدن را آغاز کرد و پس از دقیقاً یک بار عبور از روی همه‌ی پل‌ها به همان نقطه‌ی شروع بازگشت؟» راه‌حل مسأله با تجرید آن به دست آمد. گراف شکل ۷-۲ معادل با مسأله‌ی شکل ۷-۱ است. از دیدگاه نظریه‌ی گراف، معما، یافتن مداری در گراف (در صوت وجود) است که هر یال گراف دقیقاً یک بار در آن دیده شود. راه دیگر طرح این معما چنین است: «آیا می‌توانیم گراف شکل ۷-۲ را بدون برداشتن مداد به گونه‌ای رسم کنیم که نقطه‌ی پایان ترسیم همان نقطه‌ی شروع آن باشد و از هیچ یالی بیش از یک بار نگذریم؟» اویلر این مسأله را حل کرد. او ثابت کرد چنین پیمایشی شدنی است، اگر و تنها اگر گراف مسأله، همبند و درجه‌ی همه‌ی رأس‌هایش زوج باشد. چنین گراف‌هایی گراف‌های اویلری نامیده می‌شوند. از آنجا که گراف شکل ۷-۲، رأس‌هایی با درجه‌ی فرد دارد، در نتیجه، مسأله‌ی پل‌های Königsberg حل‌شدنی نیست. برای این قضیه، برهانی بر پایه‌ی استقرا ارائه می‌شود که به الگوریتمی کارآمد، برای یافتن مسیر بسته‌ی مورد نظر منجر می‌گردد.



شکل ۷-۱ مسأله‌ی پل‌های Königsberg



شکل ۷-۲ گراف متناظر با مسأله‌ی پل‌های Königsberg

مسأله: گراف همبندی (G=(V,E)) را به شما داده‌اند که درجه‌ی همه‌ی رأس‌هایش زوج است. مسیر بسته‌ی P را چنان بیابید که هر یال E دقیقاً یک بار در این مسیر بسته ظاهر شود.

به آسانی می‌توان ثابت کرد برای آن که چنین مسیر بسته‌ای وجود داشته باشد، باید درجه‌ی همه‌ی رأس‌ها زوج باشد: هنگام پیمایش یک مسیر بسته تعداد دفعات ورود به یک رأس با تعداد دفعات خروج از آن برابر است. از آنجا که قرار است از هر یال دقیقاً یک بار بگذریم، پس تعداد یال‌های گذرنده از هر رأس باید زوج باشد. برای آن که با استقرا ثابت کنیم چنین شرطی کافی است، نخست باید روشن سازیم استقرا روی کدام پارامتر بنا می‌شود. در آغاز می‌کوشیم بدون تغییر دادن مسأله، اندازه‌ی آن را کاهش دهیم. اگر یک رأس یا یک یال را حذف کنیم، ممکن است دیگر، درجه‌ی رأس‌های گراف حاصل زوج نباشد. باید مجموعه‌ی یال‌های S را به گونه‌ای برای حذف برگزینیم که تعداد یال‌هایی از S که از هر رأس v می‌گذرند، زوج باشد (توجه کنید که صفر هم زوج است). هر مدار، این شرط را برآورده می‌کند؛ پس پرسش این است که آیا یک گراف اویلری، همواره مدار دارد یا نه. فرض کنید با آغاز از رأس دل‌خواه v بدون آن که از یالی بیش از یک بار بگذریم، به ترتیبی دل‌خواه، شروع به پیمایش گراف کنیم. ادعا می‌کنیم این پیمایش سرانجام به v باز خواهد گشت، چراکه هر بار که وارد رأسی شویم، درجه‌ی آن رأس یک واحد کاهش یافته، عددی فرد می‌گردد؛ پس بی هیچ مشکلی می‌توانیم از آن خارج شویم. (توجه کنید که شاید این مدار همه‌ی یال‌ها را در بر نگیرد.)

اینک آماده‌ایم تا با بیان فرض استقرا، قضیه را ثابت کنیم. (با آن که استقرا روی مسیرهای بسته انجام می‌شود، بیان فرض روی تعداد یال‌ها از بیان آن روی تعداد مسیرها آسان‌تر است.)

فرض استقرا: یک گراف همبند با تعداد یال‌هایی کم‌تر از m که درجه‌ی همه‌ی رأس‌هایش زوج باشد، مسیری بسته دارد که در این مسیر هر یال دقیقاً یک بار آمده است و ما می‌دانیم چگونه این مسیر بسته را بیابیم.

گراف G=(V,E) را که m یال دارد، در نظر گرفته، فرض کنید P مسیری بسته در این گراف باشد. گرافی را که با حذف تمام یال‌های P از G به دست می‌آید، G' بنامید. درجه‌ی همه‌ی رأس‌های G' باید زوج باشد، چراکه تعداد یال‌های حذف‌شده‌ی گذرنده از هر رأس زوج است، اما بازهم نمی‌توان از فرض استقرا سود جست، چراکه شاید G' همبند نباشد. G'_1، G'_2، ... و G'_k را مؤلفه‌های همبند G' بگیرید. درجه‌ی تک‌تک رأس‌های هر مؤلفه‌ی همبند زوج است. همچنین، تعداد یال‌های هر مؤلفه (و بی‌شک تمام مؤلفه‌ها با همدیگر) از m کوچک‌تر است. به این ترتیب، اِعمال فرض استقرا به هر مؤلفه امکان‌پذیر می‌شود؛ یعنی هر مؤلفه، مسیری بسته دارد که هر یال آن مؤلفه دقیقاً یک بار در آن مسیر آمده است و ما می‌دانیم چگونه این مسیرهای بسته را پیدا کنیم. این k مسیر بسته را با P_1، P_2، ... و P_k نشان می‌دهیم. حال، لازم است همه‌ی این مسیرها را در قالب یک مسیر بسته چنان با هم ادغام کنیم که کل گراف را در بر گیرد. کار را با پیمایش رأسی دل‌خواه از P آغاز می‌کنیم تا آن که به نخستین رأسی برسیم که متعلق به یکی از مؤلفه‌های همبند باشد. این رأس را v_j و مؤلفه‌ی مربوط به آن را G'_j بگیرید. در اینجا، مسیر P_j را پیموده، دوباره به v_j باز می‌گردیم. می‌توان با پی‌گیری این شیوه، نخستین بار که با رأسی از یکی از این مؤلفه‌ها روبه‌رو می‌شویم، مسیر آن مؤلفه را بپیماییم. سرانجام به رأس آغازین P باز خواهیم گشت و بنابراین از همه‌ی یال‌های گراف دقیقاً یک بار گذاشته‌ایم. به این مسیر بسته، یک مدار اویلری می‌گویند؛ اما هنوز الگوریتم کامل نشده است، زیرا لازم است هم شیوه‌ای کارآمد برای یافتن این مؤلفه‌های همبند و هم روشی کارآمد برای پیمایش گراف بیابیم. اندکی بعد درباره‌ی این دو موضوع بحث خواهد شد. پیاده‌سازی الگوریتم مدار اویلری را به عنوان تمرین به خواننده واگذار می‌کنیم.

۷-۳ روش‌های پیمایش گراف

هنگام طراحی الگوریتم‌های گراف با این پرسش روبه‌رو می‌شویم که چگونه باید به ورودی بنگریم. این مسأله در فصل پیش به علت تک‌بعدی بودن ورودی، مسأله‌ای ساده و سرراست بود؛ چراکه به آسانی می‌توان دنباله‌ها و مجموعه‌ها را به ترتیب خطی پویش کرد، ولی پویش یک گراف که آن را پیمایش گراف هم می‌گوییم، به این سادگی نیست. دو الگوریتم برای پیمایش گراف ارائه می‌کنیم: جست‌وجوی نخست-ژرفا (DFS) و جست‌وجوی نخست-پهنا (BFS). (در بیش‌تر کتاب‌ها، DFS را جست‌وجوی عمق-اول و BFS را جست‌وجوی سطح-اول ترجمه کرده‌اند - مترجمان) بیش‌تر الگوریتم‌های این فصل احتمالاً به گونه‌ای به یکی از این دو شیوه برمی‌گردند.

۷-۳-۱ جست‌وجوی نخست-ژرفا

انجام جست‌وجوی نخست-ژرفا در گراف‌های جهت‌دار و گراف‌های بدون‌جهت تقریباً یکسان است، اما چون می‌خواهیم چندین ویژگی را هم در این دو دسته گراف بررسی کنیم و این ویژگی‌ها در هر دسته متفاوت از دیگری است، پس این بحث را برای هر یک از این دو دسته گراف به طور جداگانه ارائه می‌کنیم.

گراف‌های بدون‌جهت

فرض کنید گراف بدون‌جهت G=(V,E) را از روی یک نمایشگاه آثار هنری (شامل چند راهرو که نقاشی‌هایی به دیوارهای راهروهای آن آویزان است) ساخته باشیم و بخواهیم از نمایشگاه به گونه‌ای بگذریم که تمام نقاشی‌ها را تماشا کنیم. (فرض می‌کنیم جهت حرکت هر چه باشد، هنگام عبور از هر راهرو نقاشی‌های هر دو سوی آن را می‌بینیم.) هنگامی که گراف اویلری باشد، می‌توانیم طوری در نمایشگاه قدم بزنیم که از هر راهرو دقیقاً یک بار بگذریم، اما فعلاً فرض نکرده‌ایم که گراف G اویلری است؛ پس اجازه داریم از هر یالی هر چند بار که می‌خواهیم بگذریم (هنگامی که بحث به نتیجه برسد، خواهید دید که هر یال دقیقاً دو بار پیموده شده است). ایده‌ی الگوریتم جست‌وجوی نخست-ژرفا چنین است: درون نمایشگاه قدم می‌زنیم و به محض آن که توانستیم، وارد راهروی تازه‌ای می‌شویم. نخستین باری که به یک تقاطع می‌رسیم، در آنجا یک سنگ‌ریزه می‌گذاریم و وارد راهروی دیگری می‌شویم (مگر آن که راهروی تازه بن‌بست باشد) اما اگر به تقاطعی رسیدیم که در آنجا سنگ‌ریزه وجود داشت، به همان راهرویی که در آن بودیم، بازمی‌گردیم و می‌کوشیم راهرویی بیابیم که تا به حال وارد آن نشده‌ایم. هنگامی که تمام مسیرهای خارج‌شونده از یک تقاطع را دیدیم، سنگ‌ریزه را از آن تقاطع برمی‌داریم و به راهرویی وارد می‌شویم که در ابتدا از آنجا آمده بودیم؛ یعنی دوباره وارد این تقاطع نمی‌شویم. (منظور از برداشتن سنگ‌ریزه تنها تمیز کردن نمایشگاه است، پس این کار جزئی ضروری از الگوریتم نیست.) هر بار کوشیدیم راهروی تازه‌ای را بررسی کنیم. پس از بررسی تمام راهروهای هر تقاطع نیز از همان راهرویی که از آن وارد تقاطع شده بودیم، بازگشتیم. این شیوه را از این رو جست‌وجوی نخست-ژرفا می‌گویند که هر بار کوشیدیم به یک راهروی تازه برویم؛ یعنی در نمایشگاه به ژرفای بیش‌تری نفوذ کنیم. سودمندی اصلی روش DFS در شیوه‌ی آن برای تقسیم گراف، به همراه قابلیت اجرای بازگشتی آن روی بخش‌های تقسیم‌شده است.

DFS را به شکل قدم زدن و علامت‌گذاری با سنگ‌ریزه توضیح دادیم. حال، ببینیم چگونه DFS برای گراف بدون‌جهتِ داده‌شده با یک لیست همسایگی پیاده‌سازی می‌شود. پیمایش گراف را از رأس دل‌خواهی همچون r آغاز می‌کنیم و آن را ریشه‌ی DFS می‌نامیم. به ریشه علامت «مشاهده‌شده»

می‌زنیم. یک رأس دل‌خواه از رأس‌های علامت‌نخورده‌ی متصل به r برمی‌گزینیم و آن را r_1 می‌نامیم. حال، عمل DFS را به صورت بازگشتی با شروع از r_1 انجام می‌دهیم. بازگشت‌ها زمانی متوقف می‌شوند که به رأسی همچون v برسیم که تمام همسایگان آن علامت‌خورده باشند. اگر پس از آن که DFS روی r_1 به پایان رسید، تمام رأس‌های همسایه‌ی r علامت‌خورده باشند، DFS روی r نیز به پایان می‌رسد؛ در غیر این صورت، رأس علامت‌نخورده‌ی دل‌خواهی همچون r_2 را از رأس‌های متصل به r برمی‌گزینیم و DFS را با شروع از r_2 انجام می‌دهیم و به همین ترتیب کار را ادامه می‌دهیم تا هنگامی که همه‌ی رأس‌ها مشاهده شوند.

معمولاً از پیمایش گراف هدفی داریم؛ یعنی الگوریتم DFS انجام می‌شود و گراف را می‌پیماییم تا کاری را روی رأس‌ها یا یال‌های آن انجام دهیم. پیمایش «پیش‌ترتیب» گراف یعنی کار مورد نظر را (هر چه که باشد) هنگام رسیدن به یک رأس یا یال و علامت‌گذاری آن انجام می‌دهیم و پیمایش «پس‌ترتیب»، یعنی عمل مورد نظر پس از بازگشت از یک یال، یا پس از آن انجام می‌شود که دریافتیم یال به یک رأس مشاهده‌شده می‌رسد. برگزیدن روش پیش‌ترتیب یا روش پس‌ترتیب به مسأله‌ای که DFS را برای آن به کار می‌بریم، بستگی دارد. با یاری این دو اصطلاح جای اعمال را در کاربردهای گوناگون به صورت پیش‌ترتیب یا پس‌ترتیب مشخص می‌کنیم. الگوریتم DFS در شکل ۷-۳ داده شده است. در فراخوانی بازگشتی الگوریتم این شکل، v رأس آغاز است. برای سادگی در ابتدا گراف را همبند در نظر می‌گیریم. در شکل ۷-۴ مثالی از اجرای الگوریتم DFS آورده شده است که در آن اعدادِ روی رأس‌ها نشان‌دهنده‌ی ترتیب پیمایش آن‌هاست.

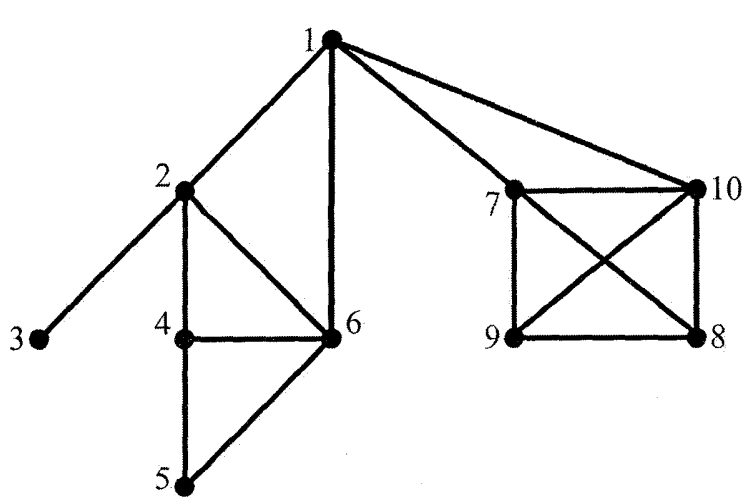
الگوریتم: Depth_First_Search(G,v)

ورودی: G=(V,E) (یک گراف همبند بدون‌جهت) و v (رأسی از G)

خروجی: وابسته به کاربرد مورد نظر است.

```
begin
    v را علامت بزن;
    {کار مورد نظر وابسته به هدفی
    است که از پیمایش گراف داریم.}
    ; انجام کار پیش‌ترتیب مورد نظر روی v
    for (v,w) تمام یال‌های do
        if w علامت‌نخورده است then Depth_First_Search(G,w);
        ; انجام کار پس‌ترتیب مورد نظر روی یال (v,w)
    {انجام کار مورد نظر وابسته به هدفی است که از پیمایش گراف داریم؛ گاهی این کار
    تنها روی یال‌هایی انجام می‌شود که به رأس‌های تازه علامت‌خورده می‌روند.}
end
```

شکل ۷-۳ الگوریتم Depth_First_Search (در متن اصلی کتاب، شماره‌ی این شکل به اشتباه ۷-۴ درج شده است - مترجمان)



شکل ۷-۴ اجرای DFS بر روی یک گراف بدون‌جهت

□ لم ۷-۱

اگر G همبند باشد، با اجرای الگوریتم Depth_First_Search تمام رأس‌ها علامت‌گذاری می‌شوند و در طی اجرای الگورتیم نیز هر یال دست‌کم یک بار دیده خواهد شد.

برهان: فرض کنید این‌گونه نباشد و U را مجموعه‌ی رأس‌های علامت‌نخورده بگیرید. از آنجا که G همبند است دست‌کم یکی از رأس‌های U باید به یک یا بیش از یک رأس از رأس‌های علامت‌خورده متصل باشد، اما چنین چیزی ممکن نیست، چون هر بار که به یک رأس می‌رسیم و آن را علامت‌گذاری می‌کنیم، به سراغ تمام رأس‌های همسایه‌ی آن هم می‌رویم و آن‌ها را نیز علامت‌گذاری می‌کنیم؛ بدین ترتیب تمام رأس‌ها پیموده خواهند شد و از آنجا که پس از دیدن هر رأس، تمام یال‌های آن را هم بررسی می‌کنیم، پس تمام یال‌های گراف نیز پیموده خواهند شد.

□

اگر گراف ورودی یعنی G=(V,E) همبند نباشد، باید DFS را اندکی تغییر دهیم. هرگاه تمام رأس‌ها در دور نخست علامت بخورند، گراف همبند است و کار به پایان می‌رسد؛ در غیر این صورت، باید دوباره کار را از یک رأس دل‌خواه علامت‌نخورده آغاز کنیم و یک DFS دیگر انجام دهیم و این کار را تا پیمایش کامل گراف ادامه دهیم. بنابراین می‌توانیم DFS را برای فهمیدن این که گراف همبند است یا نه و یافتن مؤلفه‌های همبند آن به کار ببریم. این الگوریتم در شکل ۷-۵ آورده شده است. ما عموماً با گراف‌های همبند کار خواهیم کرد، چون اگر گراف همبند نباشد، می‌توانیم هر یک از مؤلفه‌های همبند آن را جداگانه در نظر بگیریم. پس DFS را همان‌گونه که در شکل ۷-۳ آمده است، به کار می‌بریم؛ بدون آن که به صراحت بگوییم این الگوریتم ممکن است چند دور اجرا گردد.

الگوریتم: Connected_Components(G)

ورودی: G=(V,E) (یک گراف بدون‌جهت)

خروجی: برای هر رأس v، v.Component نشان‌دهنده‌ی شماره‌ی مؤلفه‌ی همبندی از گراف خواهد شد که دربردارنده‌ی رأس v است.

```
begin
    Component_Number := 1;
    while رأس علامت‌نخورده‌ای همچون v وجود دارد do
        Depth_First_Search(G,v);
                        {این عمل پیش‌ترتیب انجام شود:
        {v.Component_Number:=Component_Number;
        Component_Number := Component_Number + 1
end
```

شکل ۷-۵ الگوریتم Connected_Components

پیچیدگی: روشن است که هر یال دقیقاً یک بار از هر سوی خود (یعنی در مجموع دو بار) پیموده می‌شود. پس زمان اجرای الگوریتم متناسب با تعداد یال‌هاست. به علاوه ممکن است گراف تعدادی رأس هم داشته باشد که به هیچ جا متصل نباشند (و تمام این رأس‌ها نیز باید بررسی شوند). پس باید $O(|V|)$ را هم به عبارت زمان اجرا بیفزاییم. بنابراین کل زمان اجرا از $O(|V|+|E|)$ خواهد بود.

ساخت درخت DFS

حالا دو کاربرد ساده‌ی DFS را ارائه می‌کنیم: شماره‌گذاری یال‌ها با اعداد DFS و ساخت یک درخت پیمایش خاص که آن را درخت DFS می‌نامیم. اعداد درخت DFS ویژگی خاصی دارند که حتا اگر درخت را به طور صریح هم نسازیم، در بسیاری الگوریتم‌ها سودمند خواهد بود. با در نظر گرفتن این اعداد درک بسیاری از الگوریتم‌ها آسان‌تر می‌شود. برای توصیف این الگوریتم‌ها تنها لازم است تعیین کنیم که شیوه‌ی پیش‌ترتیب را به کار برده‌ایم یا شیوه‌ی پس‌ترتیب را. الگوریتم شماره‌گذاری رأس‌ها با اعداد DFS در شکل ۷-۶ و الگوریتم ساخت درخت DFS در شکل ۷-۷ آمده است. لازم نیست این دو الگوریتم جدا از هم اجرا شوند.

الگوریتم: DFS_Numbering(G,v)

ورودی: G=(V,E) (یک گراف بدون‌جهت) و v (یک رأس از G)

خروجی: به ازای هر رأس v، v.DFS شماره‌ی DFS برای v خواهد بود.

```
DFS_Number := 1; {مقداردهی اولیه}
```

این اعمال را در DFS به صورت پیش‌ترتیب به کار ببرید:

```
v.DFS := DFS_Number;
DFS_Number := DFS_Number + 1;
```

شکل ۷-۶ الگوریتم DFS_Numbering

الگوریتم: Build_DFS_Tree(G,v)

ورودی: G=(V,E) (یک گراف بدون‌جهت) و v (یک رأس از G)

خروجی: T (یک درخت DFS از G که در آغاز تهی است.)

این عمل را در DFS به صورت پس‌ترتیب به کار ببرید:

```
if w علامت‌نخورده است then بیفزا (v,w) را به T ;یال
{این دستور را می‌توان به فرمان if، از خط ۴ الگوریتم Depth_First_Search افزود.}
```

شکل ۷-۷ الگوریتم Build_DFS_Tree

رأسی همچون v را «بالادست» رأس w در درخت T با ریشه‌ی r گوییم، اگر v روی مسیر یکتای موجود از w به r در T باشد. اگر v بالادست w باشد، w را «پایین‌دست» v گوییم.

□ **لم ۷-۲** (ویژگی اصلی درخت‌های DFS برای گراف‌های بدون‌جهت)

اگر G=(V,E) یک گراف همبند بدون‌جهت و T=(V,F) یکی از درخت‌های DFS در G باشد که با الگوریتم Build_DFS_Tree ساخته شده است، آنگاه هر یال مانند e $(e \in E)$ یا متعلق به T است (یعنی $e \in F$) و یا دو رأس از G را به هم متصل می‌کند که یکی از آن‌ها در T بالادست دیگری است.

برهان: اگر (v,u) یک یال G باشد، فرض کنید با الگوریتم DFS، v پیش از u دیده شود. پس از آن که v را علامت زدیم، DFS را از رأس‌های علامت نخورده‌ی همسایه‌ی v آغاز می‌کنیم. چون u همسایه‌ی v است، پس یا DFS از u آغاز شده است که در این حالت یال (v,u) جزو درخت T است و یا DFS پیش از عقب‌گرد از رأس v، u را دیده است که در این حالت u در درخت T پایین‌دست v است.

□

به عبارت دیگر DFS به سراغ یال‌های جانبی (یعنی یال‌هایی که بین رأس‌ها علاوه بر مسیرهای درخت، مسیرهای جانبی به وجود می‌آورند) نمی‌رود و چنان که بعداً خواهیم دید، پرهیز از این‌گونه یال‌ها در روال بازگشتی‌ای که روی گراف اعمال می‌گردد، اهمیت دارد.

از آنجا که DFS الگوریتمی پراهمیت است، نسخه‌ای غیربازگشتی هم از آن ارائه می‌دهیم. ابزار اصلی برای پیاده‌سازی بازگشتی برنامه‌ها پشته است که اطلاعات لازم برای برگشت از فراخوانی‌های بازگشتی تودرتو را در خود نگه می‌دارد. کامپایلر، تمام داده‌های محلیِ مربوط به هر فراخوانی از هر روال بازگشتی را روی پشته نگه می‌دارد. پس هرگاه یکی از فراخوانی‌های بازگشتی به پایان برسد، می‌توانیم (بدون کوچک‌ترین تغییری در اطلاعات) دقیقاً به همان نقطه‌ی فراخوانی در روال صدازننده بازگردیم (که ممکن است فراخوانی دیگری از همین روال بازگشتی باشد). یکی از دلایل کاراتر بودن روال‌های غیربازگشتی این است که بیش‌تر اوقات لازم نیست تمام داده‌های محلی روی پشته نگه‌داری شوند. نسخه‌ی غیربازگشتی‌ای که بعداً ارائه می‌دهیم، نمونه‌ی خوبی از تبدیل یک برنامه‌ی بازگشتی به یک برنامه‌ی غیربازگشتی است.

یک مشکل عمده در تبدیل روال بازگشتی به نسخه‌ای غیربازگشتی از آن، لزوم نگه‌داری صریح محل بازگشت است. در داخل یک حلقه‌ی for، DFS را به طور بازگشتی فراخوانی می‌کنیم و از برنامه انتظار داریم محلی را به خاطر بسپارد که پس از پایان فراخوانی بازگشتی، اجرا باید از آنجا ادامه یابد. در گونه‌ی غیربازگشتی روال، خودمان باید این اطلاعات را نگه‌داری کنیم. فرض می‌کنیم هر رأس، لیستی پیوندی (به ترتیبی معین) از یال‌های گذرنده از خود دارد. (الگوریتم DFS هم به همین ترتیب به پیمایش گراف می‌پردازد.) v.First به ابتدای این لیست اشاره می‌کند و هر عنصر لیست، رکوردی شامل دو متغیر است که یکی از آن‌ها (Vertex) نشان‌دهنده‌ی رأس دیگر یال است و دیگری (Next) به عنصر بعدی لیست اشاره می‌کند. در آخرین یال هم مقدار مؤلفه‌ی Next را nil می‌گذاریم. DFS مانند پیش، تا جایی که نتواند رأس تازه‌ای بیابد، به پیمایش درخت می‌پردازد. در طی جست‌وجو، در یک پشته، رأس‌هایی را که روی مسیر ریشه به رأس فعلی قرار دارند، به ترتیب نگه می‌دارد. پس برای هر دو رأس Parent و Child در پشته اشاره‌گری به یک یال از Parent نگه‌داری می‌شود که این یال در پیمایش با DFS، هنگام عقب‌گرد از Child، یال بعدی است. نسخه‌ی غیربازگشتی DFS در شکل ۷-۸ آمده است.

الگوریتم: Nonrecursive_Depth_First_Search(G,v)

ورودی: G=(V,E) (یک گراف همبند بدون‌جهت) و v (یک رأس از G)

خروجی: وابسته به کاربرد است.

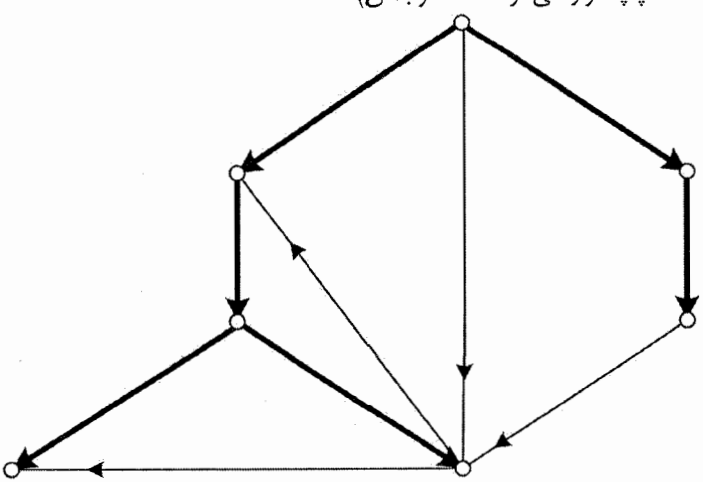
{در اینجا برخلاف بقیه‌ی فصل، نماد اشاره‌گر زبان پاسکال، یعنی ^ را به صورت صریح به کار می‌بریم.}

```
begin
    while رأس علامت‌نخورده‌ای همچون v وجود دارد do
        v را علامت بزن;
        روی رأس v کارهای پیش‌ترتیب مورد نظر را انجام بده;
        Edge := v.First;
        v و Edge را به سر پشته وارد کن;
        Parent := v;
        {تا اینجا مقداردهی اولیه انجام شد؛ حال به حلقه‌ی اصلی بازگشت می‌پردازیم.}
        while پشته تهی نیست do
            سر پشته را بردار و مقدار آن را در v بریز;
            while Edge ≠ nil do
                Child := Edge^.Vertex;
                if علامت‌نخورده است Child then
                    Child را علامت بزن;
                    روی Child کارهای پیش‌ترتیب مورد نظر را انجام بده;
                    Edge^.Next; را بالای پشته قرار بده
                    {این عمل برای این بود که پس از انجام کارهای مورد
                    نظر روی Child بتوانیم به یال بعدی بازگردیم.}
                    Edge := Child.First;
                    Parent := Child;
                    Parent را به سر پشته وارد کن;
                else {یعنی Edge یک یال عقب‌رو است.}
                    روی (Parent,Child) کارهای پس‌ترتیب دل‌خواه را انجام بده;
                    {اگر کارهای پس‌ترتیب تنها برای یال‌های درختی لازم باشد،
                    از این تکه چشم‌پوشی می‌کنیم.}
                    Edge := Edge^.Next;
            سر پشته را بردار و مقدار آن را در Child قرار بده;
            if پشته خالی نیست then
            {پشته هنگامی خالی می‌شود که Child ریشه باشد.}
                دو مقدار بالای پشته را (بدون حذف) به ترتیب در Edge و Parent بریز;
                روی (Parent,Child) کارهای پس‌ترتیب مورد نظر را انجام بده
end
```

شکل ۷-۸ الگوریتم Nonrecursive_Depth_First_Search

گراف‌های جهت‌دار

روال DFS برای گراف‌های جهت‌دار با رول DFS برای گراف بدون‌جهت یکسان است، اما درخت‌های DFS برای گراف‌های جهت‌دار ویژگی‌های متفاوتی دارند. همان‌گونه که در شکل ۷-۹ دیده می‌شود، دیگر نمی‌توان گفت که یال جانبی وجود ندارد. حالا با چهار نوع یال سر و کار داریم: یال‌های درختی، عقب‌رو، جلورو و جانبی. سه نوع یال نخست، دو رأس را که یکی پایین‌دست دیگری در درخت است، به هم متصل می‌کنند: یال‌های درختی، والدها (در درخت) را به فرزندان متصل می‌کنند؛ یال‌های عقب‌رو رأس‌های پایین‌دست را به رأس‌های بالادست وصل می‌کنند و یال‌های جلورو هم بالادست‌ها را به پایین‌دست‌ها متصل می‌کنند. تنها، یال‌های جانبی، رأس‌هایی را به هم متصل می‌کنند که خود این رأس‌ها در درخت به هم وصل نیستند؛ البته یال‌های جانبی باید مطابق آنچه در لم بعدی نشان داده خواهد شد «از راست به چپ» باشند. (می‌دانیم در الگوریتم DFS به ترتیب، درخت‌هایی از روی گراف ساخته می‌شوند. منظور نویسنده از این که درختی سمت راست‌تر است، این است که آن درخت دیرتر ساخته شده است و توجه کنید هیچ یالی نمی‌تواند از رأسی در یک درخت به رأسی از درختی دیگر در سمت راست درخت نخست برود، چراکه در این صورت، رأس انتهای این یال باید در درخت نخست یا همان درخت سمت چپ قرار می‌گرفت - مترجمان)



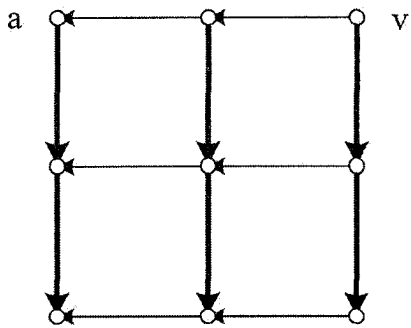
شکل ۷-۹ یال‌هایی که پررنگ کشیده شده‌اند، نشان‌دهنده‌ی یک درخت DFS از گرافی جهت‌دار هستند.

□ **لم ۷-۳** (ویژگی اصلی درخت‌های DFS برای گراف‌های جهت‌دار)
اگر G=(V,E) گرافی جهت‌دار و T=(V,F) یک درخت DFS از G و (v,w) یالی در E باشد به گونه‌ای که v.DFS_Number<w.DFS_Number، آنگاه w در درخت T رأسی پایین‌دست v است.

برهان: از آنجا که شماره‌ی DFS در رأس w از شماره‌ی DFS در رأس v بزرگ‌تر است، پس w بعد از v دیده شده است و چون یال (v,w) عضوی از E است، پس یال (v,w) باید هنگام اجرای DFS روی v بررسی شده باشد که اگر در آن زمان، w علامت‌نخورده بود، یال (v,w) به درخت اضافه می‌شد. بنابراین $(v,w) \in F$ و شرط مورد نظر برقرار است، ولی اگر هنگام اجرای DFS روی v، w علامت‌خورده بود، روشن است که w پس از v و حین یکی از فراخوانی‌های بازگشتیِ آغازشده از v علامت خورده است. پس در درخت T، w باید پایین‌دست v باشد.

□

در گراف‌های همبند بدون‌جهت، DFS از هر رأسی که آغاز شود می‌تواند کل گراف را بپیماید، اما در گراف‌های جهت‌دار چنین نیست. گراف جهت‌دار شکل ۷-۱۰ را در نظر بگیرید. اگر DFS از رأس a آغاز شود، تنها ستون سمت چپ گراف پیموده خواهد شد. DFS تنها در صورتی کل گراف شکل ۷-۱۰ را خواهد پیمود که کار خود را از رأس v آغاز کرده باشد. اگر رأس v و دو یال آن را از این گراف برداریم، دیگر هیچ رأسی نیست که DFS بتواند با آغاز از آن، کل گراف را بپیماید و ما باید دوباره انجام DFS را از رأسی علامت‌نخورده آغاز کنیم و این کار را آن قدر ادامه دهیم تا همه‌ی رأس‌های گراف علامت بخورند. بنابراین هرگاه در این کتاب سخن از DFS برای گرافی جهت‌دار به میان بیاید، منظور این است که DFS را آن قدر اجرا می‌کنیم تا همه‌ی رأس‌ها علامت بخورند و تمام یال‌های گراف بررسی شوند.



شکل ۷-۱۰ نمونه‌ای از DFS بر روی گرافی جهت‌دار که تمام گراف را به‌یک‌باره نمی‌پیماید.

برای نمونه، نشان می‌دهیم چگونه باید DFS را به کار برد تا روشن شود که آیا گراف بدون‌دور است یا نه.

مسأله: گراف جهت‌دار G=(V,E) داده شده است. تعیین کنید که آیا این گراف دوری (جهت‌دار) در خود دارد یا نه.

□ لم ۷-۴

G=(V,E) گرافی جهت‌دار و T را یک درخت DFS از آن بگیرید. G دارای دوری جهت‌دار است، اگر و تنها اگر یالی عقب‌رو (نسبت به T) داشته باشد.

برهان: اگر یالی عقب‌رو وجود داشته باشد، این یال به رأسی بالاتر در درخت منتهی می‌گردد، پس یک دور به وجود می‌آورد و برعکس، اگر C دوری در G و v رأسی از C با کم‌ترین شماره‌ی DFS باشد، ادعا می‌کنیم یال (w,v) که در C به رأس v می‌رود، یالی عقب‌رو است. این یال نمی‌تواند جلورو یا درختی باشد، چراکه از رأسی با شماره‌ی DFS بزرگ‌تر به رأسی با شماره‌ی DFS کوچک‌تر می‌رود. فرض کنید در درخت، v بالادست w نباشد و u را پایین‌ترین بالادست مشترک v و w بگیرید. روشن است که v و w در دو زیردرخت از u قرار دارند. از آنجا که شماره‌ی DFS برای v از شماره‌ی DFS برای w کم‌تر است، پس زیردرخت دربردارنده‌ی v پیش از زیردرخت دربردارنده‌ی w دیده شده است. بنابراین، برای رفتن از v به w حتماً از u یا یکی از بالادست‌های آن می‌گذریم (چراکه رفتن از چپ به راست امکان‌پذیر نیست). C مسیری از v و w در خود دارد و چون v دارای کم‌ترین شماره‌ی DFS در C است، پس هیچ یک از گره‌های بالادست v نمی‌توانند درون C باشند. (پس به این ترتیب حتماً یالی از یک گره پایین‌دست به یک گره بالادست وجود خواهد داشت - مترجمان)

□

الگوریتم تشخیص دوردار بودن گراف در شکل ۷-۱۱ آمده است.

الگوریتم: Find_a_Cycle(G)

ورودی: G=(V,E) (گرافی جهت‌دار)

خروجی: Find_a_Cycle (اگر G دور داشته باشد، این متغیر true وگرنه false خواهد شد.)

DFS را از رأسی دل‌خواه آغاز کنید، اما در این الگوریتم، هم لازم است اعمالی به صورت پیش‌ترتیب انجام شود و هم لازم است کارهایی را به صورت پس‌ترتیب انجام دهیم.

کارهای پیش‌ترتیب:

```
v.on_the_path := true;
{اگر رأس x روی مسیر ریشه به رأس فعلی قرار داشته باشد x.on_the_path، true خواهد بود.}
{در آغاز الگوریتم، Find_a_Cycle و x.on_the_path برای هر رأس x، false است.}
```

کارهای پس‌ترتیب:

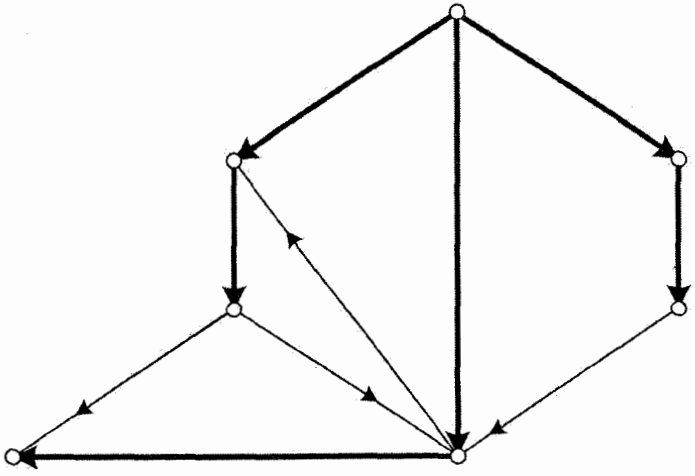
```
if w.on_the_path then Find_a_Cycle := true; halt;
if w آخرین رأس از لیست همسایگی v است then v.on_the_path := false;
```

شکل ۷-۱۱ الگوریتم Find_a_Cylcle

۷-۳-۲ جست‌وجوی نخست-پهنا

به نظر می‌رسد که جست‌وجوی نخست-پهنا (BFS) گراف را سازمان‌یافته‌تر می‌پیماید؛ یعنی سطح به سطح کار خود را انجام می‌دهد. اگر کار را از رأسی مانند v آغاز کنیم، نخست فرزندان v بررسی

می‌شوند، سپس در سطح دوم همه‌ی نوه‌های v را بررسی می‌کنیم و ... . (شکل ۷-۱۲ را ببینید.) پیاده‌سازی این پیمایش مانند پیاده‌سازی نوع غیربازگشتی DFS انجام می‌شود، اما به جای پشته یک صف را به کار می‌بریم. در اینجا می‌توانیم به جای شماره‌های DFS، به رأس‌ها شماره‌های BFS نسبت دهیم؛ یعنی برای رأسی مانند w شماره‌ی BFS نشان می‌دهد که w چندمین رأسی است که الگوریتم BFS آن را علامت‌گذاری کرده یا دیده است. برای ساخت درخت BFS کافی است تنها یال‌هایی را که به رأس‌های تازه دیده‌شده منتهی می‌شوند، در نظر بگیریم. الگوریتم BFS در شکل ۷-۱۳ داده شده است. (چون الگوریتم BFS از نظر شهودی، تنها رو به پایین حرکت می‌کند، پس برخلاف DFS نمی‌توان به راحتی برای آن اعمال پس‌ترتیب تعریف کرد. ما هم از اعمال پس‌ترتیب برای BFS چشم‌پوشی کرده‌ایم.)



شکل ۷-۱۲ یک درخت BFS برای گرافی جهت‌دار

لم ۷-۵

اگر (u,w) چنان یالی از درخت BFS باشد که u والد w گردد، آنگاه بین رأس‌هایی که یالی از آن‌ها به w می‌رسد، u دارای کم‌ترین شماره‌ی BFS است.

برهان: در صف، عنصری که زودتر وارد شود، زودتر هم خارج می‌شود. اثبات از روی این ویژگی صف انجام می‌شود.

□

لم ۷-۶

مسیر ریشه به هر رأس w در درخت، کوتاه‌ترین (یعنی کم‌یال‌ترین) مسیر از ریشه به آن رأس در گراف است.

برهان: به خواننده واگذار می‌شود.

□

الگوریتم: Breadth_First_Search(G,v)

ورودی: G=(V,E) (یک گراف همبند بدون‌جهت) و v (راسی از G)

خروجی: وابسته به کاربرد راست.

```
begin
    v را علامت بزن;
    v را در صف بگذار
    while صف تهی نیست do
        نخستین رأس صف را بردار و مقدارش را در w بگذار;
        کارهای پیش‌ترتیب مورد نظر را روی w انجام بده;
        {کارهای پیش‌ترتیب وابسته به کاربرد BFS است.}
        for تمام یال‌های (w,x) که در آن‌ها، x علامت‌نخورده است do
            x را علامت بزن;
            (w,x) را به درخت T بیفزا;
            x را در صف بگذار;
end
```

شکل ۷-۱۳ الگوریتم Breadth_First_Search

سطح رأسی مانند w، طول (یعنی تعداد یال‌های) مسیر ریشه تا آن رأس در درخت است. BFS، گراف را سطح به سطح می‌پیماید.

□ لم ۷-۷

اگر یال (u,w) در E، متعلق به درخت T نباشد، آنگاه تفاوت شماره‌ی سطح رأس‌هایی که با این یال به یکدیگر متصل می‌شوند، بیش از ۱ نیست.

برهان: به خواننده واگذار می‌شود.

□

اینک که می‌دانیم چگونه باید یک گراف را پیمود، چند الگوریتم گراف را می‌آوریم. بازهم از رویکرد «طراحی به یاری استقرا» بسیار بهره می‌بریم.

۷-۴ ترتیب توپولوژیک

فرض کنید باید چند کار را یکی‌یکی انجام دهیم. انجام برخی از این کارها وابسته به انجام برخی دیگر است؛ یعنی تا پیش‌نیاز کاری کاملاً انجام نشده باشد، نمی‌توان خود آن کار را شروع کرد. می‌دانیم که این کارها چگونه به یکدیگر وابسته‌اند و می‌خواهیم برنامه‌ای متناسب با این وابستگی‌ها برای انجام کارها ارائه دهیم. (زمان آغاز هر کار را به گونه‌ای تنظیم می‌کنیم که پیش‌نیازهای آن کار از قبل انجام

شده باشند.) می‌خواهیم الگوریتمی طراحی کنیم که بتواند به سرعت برنامه‌ی زمانی لازم را ارائه دهد. این مسأله را ترتیب توپولوژیک گویند. می‌توانیم از روی کارها و رابطه‌ی پیش‌نیازیِ بین آن‌ها گرافی جهت‌دار بسازیم. هر کار با یک رأس متناظر می‌شود و اگر کار x پیش‌نیاز کار y باشد، یالی جهت‌دار از رأس x به رأس متناظر با y می‌کشیم. روشن است که گراف نباید دور داشته باشد، وگرنه برخی کارها را هرگز نمی‌توان آغاز کرد.

مسأله: گراف جهت‌دار بدون‌دور G=(V,E) با n رأس داده شده است. به این رأس‌ها برچسب‌های ۱ تا n را چنان نسبت دهید که اگر برچسب رأسی مانند v، k بود، آنگاه برچسب تمام رأس‌هایی که با مسیری جهت‌دار از v دست‌رس‌پذیرند، از k بزرگ‌تر باشد.

به یاری استقرا راه‌حل ساده و سرراستی برای مسأله می‌یابیم.

فرض استقرا: می‌دانیم چگونه باید رأس‌های هر گراف جهت‌دار بدون‌دوری را که کم‌تر از n رأس داشته باشد، طبق شرایط مسأله برچسب بزنیم.

حالت پایه روشن است. طبق معمول، گرافی با n رأس در نظر می‌گیریم و یک رأس آن را کنار می‌گذاریم، پس از به‌کارگیری فرض استقرا روی باقی‌مانده‌ی گراف می‌کوشیم برچسب‌گذاری را گسترش دهیم. هر رأسی را که بخواهیم، می‌توانیم به عنوان رأس nام برگزینیم. پس به دنبال رأسی می‌گردیم که کارمان را ساده‌تر کند. برچسب زدن به کدام رأس آسان‌تر است؟ رأسی (کاری) که به دیگر رأس‌ها (کارها) اصلاً وابستگی نداشته باشد؛ یعنی رأسی که درجه‌ی ورودی آن صفر باشد. بدون هیچ مشکلی می‌توان به این رأس برچسب ۱ زد؛ اما آیا یافتن رأسی با درجه‌ی ورودی صفر همواره ممکن است؟ بالاخره کار را باید از جایی آغاز کنیم، پس به طور شهودی می‌بینیم که پاسخ مثبت است. لم بعدی، این موضوع را ثابت می‌کند.

□ لم۷-۸

در هر گراف جهت‌دار بدون‌دور همواره رأسی با درجه‌ی ورودی صفر وجود دارد.

برهان: اگر درجه‌ی ورودی تمام رأس‌ها مثبت باشد، می‌توانیم گراف را در خلاف جهت یال‌هایش بپیماییم، بدون آن که ناچار به توقف شویم. از آنجا که تعداد رأس‌ها متناهی است، پس حتماً وارد یک دور می‌شویم؛ اما در گراف بدون‌دور چنین چیزی ممکن نیست. (با همین استدلال می‌توان نشان داد که رأسی هم با درجه‌ی خروجی صفر وجود دارد.)

□

به زودی خواهیم دید که چگونه می‌توان رأسی با درجه‌ی ورودی صفر یافت. پس از یافتن این رأس به آن برچسب ۱ می‌دهیم و سپس این رأس را به همراه یال‌های گذرنده از آن کنار می‌گذاریم و به برچسب‌گذاری باقی گراف - که بی‌گمان هنوز هم بدون‌دور است - با اعداد ۲ تا n می‌پردازیم. (اگر بخواهیم واقعاً دقیق باشیم، فرض استقرا بر پایه‌ی برچسب‌هایی از ۱ تا n-۱ است، نه ۲ تا n؛ اما تفاوت

این دو بازه هیچ مشکلی ایجاد نمی‌کند.) توجه کنید که پس از برگزیدن رأسی با درجه‌ی ورودی صفر (برای کاهش) دیگر چندان کاری برای الگوریتم باقی نمی‌ماند.

پیاده‌سازی: تنها مشکل پیاده‌سازی، یافتن روشی برای پیدا کردن رأسی با درجه‌ی ورودی صفر و تنظیم دوباره‌ی درجه‌های ورودی، پس از کنار گذاشتن آن رأس است. متغیری به نام Indegree به هر رأس نسبت می‌دهیم و در آغازِ الگوریتم درجه‌ی ورودی هر رأس مانند v را در متغیر v.Indegree می‌ریزیم. مقداردهی اولیه‌ی متغیرهای Indegree با یک بار پیمایش تمام یال‌های گراف (به هر ترتیب دل‌خواه) ممکن است (مثلاً با DFS)؛ به این ترتیب که پس از پیمودن یال (v,w)، مقدار v.Indegree را یک واحد بیفزاییم. رأس‌های با درجه‌ی ورودی صفر را در یک صف (یا پشته) قرار می‌دهیم. بنا به لم ۷-۸ دست‌کم رأسی مانند v با درجه‌ی ورودی صفر وجود دارد. v را به آسانی یافته، از صف کنار می‌گذاریم. سپس به ازای هر یال خارج‌شونده از v مانند (v,w)، شمارنده‌ی رأس w را یک واحد می‌کاهیم. هنگام صفر شدنِ این شمارنده، خود رأس را هم در صف قرار می‌دهیم. پس از برداشتن v گراف همچنان بدون‌دور باقی می‌ماند. پس بنا به لم ۷-۸ باید دست‌کم یک رأس با درجه‌ی ورودی صفر در گراف باقی‌مانده وجود داشته باشد. الگوریتم هنگامی به پایان می‌رسد که صف خالی شود. در این زمان همه‌ی رأس‌ها برچسب خورده‌اند. الگوریتم در شکل ۷-۱۴ داده شده است.

الگوریتم: Topological_Sorting(G)

ورودی: G=(V,E) (یک گراف جهت‌دار بدون‌دور)

خروجی: میدان label از هر رأس نشان‌دهنده‌ی ترتیب توپولوژیک آن در G خواهد بود.

```
begin
    {برای مثال با DFS} ;v.Indegree را به ازای هر رأس v در گراف مقداردهی اولیه کن
    G_label := 0;
    for i := 1 to n do
        if v_i.Indegree = 0 then v_i را در صف بگذار;
    repeat
        رأس سر صف را بردار و در v بریز;
        G_label := G_label + 1;
        v.label := G_label;
        for (v,w) هر یال do
            w.Indegree := w.Indegree - 1;
            if w.Indegree = 0 then w را در صف بگذار;
    until صف خالی شود
end
```

شکل ۷-۱۴ الگوریتم Topological_Sorting

پیچیدگی: مقداردهی اولیه به متغیرهای Indegree نیازمند زمانی از $O(|V|+|E|)$ است. زمان یافتن رأسی با درجه‌ی ورودی صفر (چون از صف استفاده می‌کنیم) مقداری ثابت است. هر یال (v,w) یک بار

(هنگام برداشتن v از صف) بررسی می‌شود. پس، زمان لازم برای به روز کردن مقدار متغیرها دقیقاً برابر تعداد یال‌های گراف است؛ بنابراین زمان اجرای الگوریتم از $O(|V|+|E|)$ خواهد بود که نسبت به اندازه‌ی ورودی خطی است.

۷-۵ کوتاه‌ترین مسیر از یک رأس به رأس‌های دیگر

در این بخش با گراف‌های وزن‌دار سر و کار داریم. G=(V,E) را گرافی جهت‌دار بگیرید که به یال‌های آن مقداری نامنفی به عنوان وزن نسبت داده شده است. دراین بخش وزن را طول می‌گوییم، چراکه معمولاً این مسأله را مسأله‌ی کوتاه‌ترین مسیر (نه سبک‌ترین مسیر) می‌نامند. (طول مسیر گاهی هم تعداد یال‌های مسیر را نشان می‌دهد؛ دقت کنید این دو را با یکدیگر اشتباه نگیرید.) اگر گراف بدون‌جهت باشد، آن را گرافی جهت‌دار در نظر می‌گیریم، اما به جای هر یال گراف بدون‌جهت، دو یال در هر دو جهت با همان طول اولیه قرار می‌دهیم. پس می‌توان بحث این بخش را برای گراف‌های بدون‌جهت نیز به کار برد. طول مسیر، مجموع طول یال‌های آن است.

مسأله گراف جهت‌دار G=(V,E) و رأس v از آن داده شده‌اند. کوتاه‌ترین مسیر از v به هر یک از رأس‌های دیگر را بیابید.

برای سادگی، تنها، روش یافتن طول کوتاه‌ترین مسیرها را بررسی می‌کنیم. سپس الگوریتم را چنان گسترش می‌دهیم تا خودِ مسیرها را نیز بیابد. نمونه‌های بسیاری از این مسأله وجود دارد. برای مثال، ممکن است گراف را از روی نقشه‌ای از راه‌ها ساخته باشیم و برچسب هر یال، بنا به مسأله، متناسب با طول واقعی، یا زمان لازم برای پیمودن آن، یا هزینه‌ی ساخت آن باشد.

حالت بدون‌دور

نخست، بیایید فرض کنیم گراف G بدون‌دور است. در این حالت، حل مسأله آسان‌تر است و به حل مسأله در حالت کلی نیز کمک می‌کند. نخست می‌کوشیم با استقرا روی تعداد رأس‌ها مسأله را حل کنیم. حالت پایه روشن است. حال $|V|$ را برابر n بگیرید. می‌توانیم ترتیب توپولوژیک را که در بخش پیش گفته شد، به کار گیریم. اگر برچسب v، k باشد، دیگر لازم نیست به بررسی رأس‌هایی که برچسبی کم‌تر از k دارند، بپردازیم (این رأس‌ها از v دست‌رس‌پذیر نیستند). ترتیب توپولوژیک برای استقرا هم ترتیبی مناسب است. آخرین رأس را که برچسب n دارد، در نظر بگیرید (آن را z می‌نامیم). فرض می‌کنیم (بنا به استقرا) کوتاه‌ترین مسیرها را از v به همه‌ی رأس‌های دیگر به جز z می‌دانیم. طول کوتاه‌ترین مسیر از v به w را با w.SP نشان می‌دهیم. برای یافتن z.SP تنها لازم است

رأس‌هایی را بررسی کنیم که یالی از آن‌ها به z می‌رود. چون از پیش، کوتاه‌ترین مسیر به هر یک از رأس‌های دیگر را می‌شناسیم، پس اگر به ازای هر رأس w که یالی به z دارد، طول یال (w,z) را به w.SP بیفزاییم؛ می‌توانیم با برگزیدن کم‌ترین مقدار در بین این حاصل‌جمع‌ها، z.SP را بیابیم. آیا کار به پایان رسیده است؟ باید دقت کنیم که با در نظر گرفتن z کوتاه‌ترین مسیر به رأس‌های دیگر تغییر نکند. از آنجا که z آخرین رأس در ترتیب توپولوژیک است، پس در گراف رأس دیگری از z دست‌رس‌پذیر نیست و این رأس بر هیچ مسیر دیگری تأثیر نخواهد داشت. بنابراین z را کنار می‌گذاریم و کوتاه‌ترین مسیرها را بدون در نظر گرفتن آن می‌یابیم. حال، اگر z را سر جایش برگردانیم، مسأله حل شده است. پس فرض استقرا این‌گونه خواهد بود:

فرض استقرا: اگر یک ترتیب توپولوژیک داده شده باشد، می‌دانیم چگونه کوتاه‌ترین مسیرها از v به n−۱ رأس نخست را بیابیم.

گرافی بدون‌دور با n رأس و ترتیبی توپولوژیک از این رأس‌ها داده شده است. nامین رأس را کنار می‌گذاریم، مسأله‌ی کاهش‌یافته را با استقرا حل می‌کنیم و سپس به ازای هر یال $(w,z)\in E$ طول یال (w,z) را به مقدار w.SP می‌افزاییم و کم‌ترین مقدار را در بین تمام این حاصل‌جمع‌ها برمی‌گزینیم. این الگوریتم را در شکل ۷-۱۵ نشان داده‌ایم. اینک الگوریتم را به گونه‌ای بهبود می‌دهیم که بتوان همراه با یافتن کوتاه‌ترین مسیرها، ترتیب توپولوژیک را نیز پیدا کرد. به عبارت دیگر، می‌خواهیم دو بار پیمایش گراف، یکی برای محاسبه‌ی ترتیب توپولوژیک و دیگری برای یافتن کوتاه‌ترین مسیرها را در هم ادغام کنیم.

الگوریتم: Acyclic_Shortest_Paths(G,v,n)

ورودی: G=(V,E) (یک گراف وزن‌دار بدون‌دور)، v (یک رأس) و n (تعداد رأس‌ها)

خروجی: به ازای هر رأس $w\in V$، w.SP برابر طول کوتاه‌ترین مسیر از v به w خواهدشد.

{فرض می‌کنیم مرتب‌سازی توپولوژیک از پیش انجام شده است. در شکل ۷-۱۶ گونه‌ی بهبودیافته‌ای از الگوریتم آمده است که ترتیب توپولوژیک را نیز حساب می‌کند.}

```
begin
    z را برابر رأسی قرار بده که در ترتیب توپولوژیک، برچسب n دارد;
    if z ≠ v then
        Acyclic_Shortest_Paths(G-z,v,n-1);
        {G-z با حذف z و یال‌های آن از G به دست می‌آید.}
        for همه‌ی wهایی که (w,z) متعلق به E است do
            if (w.SP + (طول یال (w,z)) < z.SP then
                z.SP := w.SP + (طول یال (w,z));
    else v.SP := 0
end
```

شکل ۷-۱۵ الگوریتم Acyclic_Shortest_Paths

به روش اجرای بازگشتی الگوریتم (پس از یافتن ترتیب توپولوژیک) دقت کنید. برای سادگی، برچسب v را در ترتیب توپولوژیک ۱ فرض کنید. گام نخست، فراخوانی بازگشتی روال است. این روال تا رسیدن به v مرتباً خود را فراخوانی می‌کند. آنگاه طول کوتاه‌ترین مسیر به v را صفر قرار می‌دهد و شروع به بازگشت می‌کند. سپس به سراغ رأس u با برچسب ۲ می‌رود و طول کوتاه‌ترین مسیر به آن را برابر طول یال v به u (در صورت وجود این یال) قرار می‌دهد؛ اگر چنین یالی در گراف نباشد، آنگاه مسیری از v به u وجود نخواهد داشت. گام بعد، بررسی رأسی است که برچسب ۳ دارد (آن را x می‌نامیم). در این مورد، ممکن است یال‌هایی از v یا u به x وجود داشته باشند که باید مسیرهای ساخته‌شده با این یال‌ها را با یکدیگر مقایسه کنیم. دراینجا، به جای به‌کارگیری روش فراخوانی بازگشتی برای عقب‌گرد، تلاش می‌کنیم همین گام‌ها را در حرکت رو به جلو، یعنی به ترتیب افزایش برچسب‌ها برداریم.

استقرا با شروع از v، به ترتیب افزایشی برچسب‌ها به کار گرفته می‌شود. به این ترتیب، دیگر لازم نیست برچسب‌ها را از پیش بدانیم و می‌توانیم هر دو الگوریتم را هم‌زمان اجرا کنیم. فرض می‌کنیم طول کوتاه‌ترین مسیرها را تا رأس‌هایی که برچسب‌های ۱ تا m دارند، می‌دانیم. حال رأس با برچسب m+۱ را در نظر گرفته، آن را z می‌نامیم. برای یافتن کوتاه‌ترین مسیر تا z، باید همه‌ی یال‌های واردشونده به آن را بررسی کنیم. چون از ترتیب توپولوژیک پیروی کرده‌ایم، همه‌ی این یال‌ها از رأس‌هایی آمده‌اند که برچسب آن‌ها از برچسب z کم‌تر است. بنا به فرض استقرا همه‌ی این رأس‌ها را پیش‌تر بررسی کرده‌ایم؛ یعنی طول کوتاه‌ترین مسیر به هر یک از آن‌ها را می‌دانیم. به ازای هر یال مانند (w,z)، طول کوتاه‌ترین مسیر تا w یعنی w.SP را می‌دانیم؛ پس طول «کوتاه‌ترین مسیر به z با گذر از این یال» برابر w.SP به اضافه‌ی طول یال (w,z) خواهد بود. بنابراین طول کوتاه‌ترین مسیر تا z کمینه‌ی «w.SP به اضافه‌ی طول یال (w,z)» در بین تمام wهاست. بازهم لازم نیست نگران تنظیم کوتاه‌ترین مسیرها تا رأس‌هایی باشیم که برچسب کوچک‌تری دارند؛ چراکه هنوز هم این رأس‌ها از z دست‌رس‌پذیر نیستند. الگوریتم بهبودیافته را در شکل ۷-۱۶ آورده‌ایم.

پیچیدگی: هر یال، یک بار هنگام مقداردهی اولیه‌ی درجه‌های ورودی و یک بار هنگام برداشتن رأس انتهایی آن از صف، بررسی می‌شود. زمان دست‌رسی به صف نیز مقداری ثابت است. هر رأس هم یک بار بررسی می‌شود. بنابراین زمان اجرا در بدترین حالت از $O(|V|+|E|)$ خواهد شد.

الگوریتم: Improved_Acyclic_Shortest_Paths(G,v)

ورودی: G=(V,E) (یک گراف وزن‌دار بدون‌دور) و v (یک رأس از G)

خروجی: به ازای هر رأس w، w.SP برابر طول کوتاه‌ترین مسیر از v به w خواهد شد.

{این الگوریتم نسخه‌ی غیربازگشتی الگوریتم پیش است، اما مرتب‌سازی توپولوژیک را هم انجام می‌دهد.}

```
begin
    for هر رأس w do
        w.SP := ∞;
    برای هر رأس v، v.indegree را مقداردهی اولیه کن;   {مثلاً با DFS}
    for i := 1 to n
        if v_i.indegree = 0 then v_i را در صف بگذار;
    v.SP := 0;
    repeat
        رأس سر صف را بردار و در w بگذار;
        for هر یال (w,z) do
            if (w.SP +(w,z) طول یال ) < z.SP then
                z.SP := w.SP +(w,z) طول یال;
            z.indegree := z.indegree - 1;
            if z.indegree = 0 then z را در صف بگذار;
    until صف خالی شود
end
```

شکل ۷-۱۶ الگوریتم Improved_Acyclic_Shortest_Paths

حالت کلی

اگر گراف، بدون‌دور نباشد، ترتیب توپولوژیک معنا ندارد و دیگر نمی‌توان مستقیماً الگوریتم‌های گفته‌شده را به کار برد، اما شاید بتوان از ایده‌ی آن‌ها برای حالت کلی نیز سود جست. سادگی الگوریتم‌های گفته‌شده نتیجه‌ی این دو ویژگی ترتیب توپولوژیک است:

اگر z رأسی با برچسب k باشد، آنگاه: (۱) مسیری از z به رأس‌هایی که برچسب آن‌ها از k کوچک‌تر است، وجود ندارد؛ (۲) از رأس‌هایی که برچسب آن‌ها از k بزرگ‌تر است، مسیری به z وجود ندارد.

به دلیل این دو ویژگی بود که توانستیم بدون بررسی رأس‌هایی که درترتیب توپولوژیک پس از z قرار می‌گیرند، کوتاه‌ترین مسیر را از v به z بیابیم. آیا می‌توان ترتیبی روی رأس‌ها چنان تعریف کرد که امکان چنین کاری را در حالت کلی هم برای ما فراهم کند؟

ایده‌ی این کار، بررسی رأس‌های گراف با ترتیبی ناشی از طول کوتاه‌ترین مسیر از v تا آن‌هاست. اگرچه در آغاز، مقدار این طول‌ها را نمی‌دانیم، هنگام اجرای الگوریتم، آن‌ها را خواهیم یافت. نخست، همه‌ی یال‌های خارج‌شونده از v را بررسی می‌کنیم. (v,x) را کوتاه‌ترین یال در بین آن‌ها بگیرید. چون طول تمام یال‌ها مثبت است، کوتاه‌ترین مسیر از v به x، همان یال (v,x) خواهد بود. هیچ یک از دیگر مسیرهای آغازشونده از v کوتاه‌تر از این مسیر نیستند. پس کوتاه‌ترین مسیری را که به x می‌رسد، می‌شناسیم و می‌توانیم آن را به عنوان پایه‌ای برای استقرا به کار ببریم. بیایید یک گام دیگر هم برداریم. چگونه می‌توانیم کوتاه‌ترین مسیر تا یک رأس دیگر را هم پیدا کنیم؟ رأسی را برمی‌گزینیم که از نظر نزدیک بودن به v در جایگاه دوم است (x در جایگاه نخست بود). تنها لازم است دو دسته مسیر بررسی شوند: (۱) مسیرهایی که با یال دیگری از v آغاز شده باشند؛ (۲) مسیرهایی که دو یال داشته باشند و یال نخستشان همان (v,x) بوده، یال دومشان از x آغاز شود. به ازای هر رأس y به جز x، طول هر یال (v,y) را حساب می‌کنیم. همچنین به ازای هر رأس z به جز v، مجموع طول دو یال (v,x) و (x,z) را به دست می‌آوریم. در بین این دو دسته‌مقدار، کم‌ترین را برمی‌گزینیم. (یعنی: min({ ∀ y∈V | y≠x: (v,y)طول } ∪ { ∀ z∈V | z≠v: ((x,z)طول + (v,x)طول)}) - مترجمان)
در اینجا نیز لازم نیست مسیرهای دیگر را در نظر بگیریم، زیرا کوتاه‌ترین مسیر خارج‌شونده از v تا هر رأس به جز x، در بین همین مسیرهاست. فرض استقرا در حالت کلی چنین می‌شود:

فرض استقرا: یک گراف و رأس v از آن داده شده‌اند. k رأسی را که از دیگر رأس‌ها به v نزدیک‌ترند، می‌شناسیم و کوتاه‌ترین مسیر از v تا آن‌ها را نیز می‌دانیم.

توجه کنید که استقرا روی تعداد رأس‌هایی است که کوتاه‌ترین مسیر تا آن‌ها از پیش محاسبه شده است؛ نه روی اندازه‌ی گراف. به علاوه، استقرا فرض می‌کند که این رأس‌ها نزدیک‌ترین رأس‌ها به v هستند و ما می‌توانیم آن‌ها را شناسایی کنیم. می‌دانیم چگونه نزدیک‌ترین رأس تا v (در اینجا x) را بیابیم، بنابراین حالت پایه (k=۱) حل شده است. مسأله هنگامی که k به |V|-1 برسد، حل می‌شود.

مجموعه‌ی رأس v همراه با نزدیک‌ترین k رأس به آن را با V_k نشان می‌دهیم. مشکل، یافتن نزدیک‌ترین رأسی به v است که عضو V_k نباشد (این رأس را w بنامید). به علاوه باید کوتاه‌ترین مسیر از v به w را نیز بیابیم. نزدیک‌ترین مسیر از v به w، تنها می‌تواند از رأس‌های V_k بگذرد (اگر این مسیر از رأسی بگذرد که در V_k نباشد، آنگاه این رأس از w به v نزدیک‌تر خواهد بود). بنابراین هنگام یافتن w، تنها کافی است به یال‌هایی توجه کنیم که رأس ابتدای آن‌ها عضو V_k است و رأس انتهای آن‌ها عضو V_k نیست. (u,z) را یکی از این یال‌ها بگیرید؛ یعنی $u \in V_k$ و $z \notin V_k$. چنین یالی برقرارکننده‌ی مسیری از v به z است که کوتاه‌ترین مسیر از v به u را (که بنا به استقرا آن را می‌شناسیم) همراه با یال (u,z) در بر دارد. باید چنین مسیرهایی را با هم مقایسه کرده، کوتاه‌ترین آن‌ها را برگزینیم.

الگوریتمِ برآمده از فرض استقرا چنین است: هر بار رأسی تازه مانند w افزوده می‌شود که مقدار رابطه‌ی (۷-۱) را برای تمام wهایی که در V_k نیستند، کمینه کند.

$$\min_{u \in V_k}(u.SP + (u,w)\text{طول}) \qquad (۱-۷)$$

بنا به استدلال پیش، w حتماً k+۱امین رأس نزدیک به v است و با افزودن آن، فرض استقرا گسترش می‌یابد.

حالا دیگر الگوریتم به درستی کار می‌کند، اما می‌توان کارایی آن را بهبود بخشید. گام اصلی الگوریتم دربردارنده‌ی یافتن نزدیک‌ترین رأس است و با محاسبه‌ی طول مسیر کمینه طبق رابطه‌ی (۷-۱) می‌توانیم این رأس را بیابیم، اما لازم نیست هر بار تمام مقدارهای (طول یال(u,w) + u.SP) را بیازماییم. هنگام افزودن رأسی تازه، تنها برخی از این مقدارها ممکن است تغییر کنند؛ یعنی آن‌هایی که متناظر با مسیرهایی باشند که از این رأس تازه می‌گذرند. می‌توانیم طول کوتاه‌ترین مسیر تا رأس‌های درون V_k را نگه‌داری کنیم و هنگام افزوده شدن رأس تازه به V_k، مقدار این طول‌ها را به روز کنیم. برای این که ببینیم پس از افزوده شدن w به V_k، کوتاه‌ترین مسیرها کدام‌ها هستند، کافی است تنها مسیرهایی را که از w می‌گذرند، در نظر بگیریم. پس باید تمام یال‌های خارج‌شونده از w به رأس‌هایی را در نظر بگیریم که این رأس‌ها عضو V_k نباشند. به ازای هر یالی از این دست مانند (w,z)، باید مقدار (w.SP + طول یال (w,z)) را حساب کنیم و در صورت نیاز مقدار z.SP را تغییر دهیم. پس، هر بار باید رأسی را بیابیم که مقدار SP برای آن کمینه خواهد شد و مقدار SP را برای برخی از رأس‌های باقی‌مانده تغییر دهیم. به این الگوریتم، الگوریتم Dijkstra گویند.

پیاده‌سازی: لازم است در مجموعه‌ی طول‌های مسیر، کم‌ترین مقدار را بیابیم و پیوسته این مقدارها را به روز کنیم. هرم، ساختار مناسبی برای یافتن کوچک‌ترین عنصر و به‌روزرسانی طول مسیرهاست. باید رأسی را بیابیم که طول مسیر تا آن رأس، کم‌ترین مقدار باشد؛ پس همه‌ی رأس‌هایی را که هنوز در V_k قرار نگرفته‌اند، در یک هرم می‌گذاریم و کلید هر عنصر این هرم را طول کوتاه‌ترین مسیر شناخته‌شده از v تا آن رأس قرار می‌دهیم. در آغاز کار، همه‌ی این طول مسیرها به جز یکی ∞ هستند، بنابراین عناصر هرم ترتیب مشخصی ندارند (به جز v که در بالای آن قرار گرفته است). یافتن w به آسانی با برداشتن عنصر بالای هرم انجام می‌شود. می‌توان هر یال (w,u) را بررسی کرده، به راحتی، طول مسیر را تغییر داد. هنگامی که طول مسیری تا یک رأس مانند z تغییر کند، ممکن است جای z هم در هرم عوض شود و ما باید بتوانیم این کار را انجام دهیم. برای انجام این لازم است محل z را در هرم بشناسیم. (به خاطر بیاورید که هرم یک ساختار جست‌وجو نیست؛ یعنی امکانی برای یافتن عنصر فراهم نمی‌آورد.) یافتن مکان z در هرم با ساختمان داده‌ی دیگری انجام می‌شود. چون رأس‌ها را از پیش می‌شناسیم، می‌توانیم آن‌ها را همراه با اشاره‌گرهایی به محلشان در هرم درون یک آرایه بگذاریم. پس برای یافتن هر عنصر در هرم کافی است این آرایه را بررسی کنیم. عنصرهای هرم ، رأس‌های گرافند، پس فضای مورد نیاز برای این آرایه از O(|V|) خواهد بود که مقداری پذیرفتنی و معقول است. طول‌های مسیر، تنها کم می‌شوند؛ یعنی اگر عنصری در هرم از والدش کوچک‌تر باشد، جابه‌جا شده، به

بالا می‌رود تا آن که در جایگاه درست خود قرار گیرد. روال‌های عادی کار با هرم نیز همین‌گونه هستند (مثلاً افزودن یک عنصر به هرم). الگوریتم یافتن کوتاه‌ترین مسیرها در شکل ۷-۱۷ آمده است.

الگوریتم: Single_Source_Shortest_Paths(G,v)

ورودی: G=(V,E) (یک گراف وزن‌دار جهت‌دار) و v (رأس آغازین رأس‌ها)

خروجی: برای هر رأس w، w.SP طول کوتاه‌ترین مسیر از v به w خواهد بود.

{فرض می‌کنیم هیچ یک از طول‌ها منفی نباشد.}

```
begin
    for هر رأس w do
        w.mark := false;
        w.SP := ∞;
    v.SP := 0;
    while رأسی علامت‌نخورده وجود دارد do
            رأس علامت‌نخورده‌ای را که مقدار SP برای آن کمینه است، در w قرار بده;
            w.mark := true;
            for هر یال (w,z) که z علامت‌نخورده است do
                if (w.SP + (طول یال (w,z)) < z.SP then
                    z.SP := (w.SP + (طول یال (w,z))
end
```

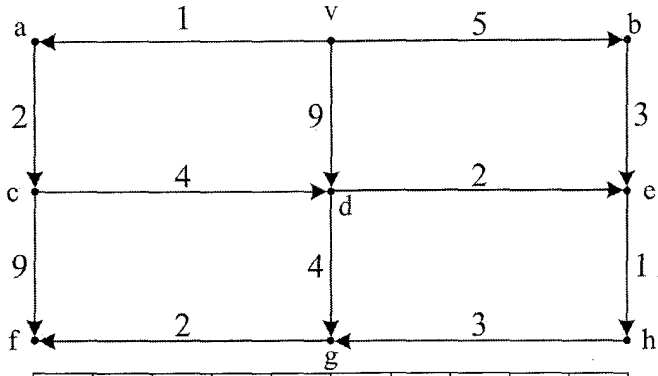
شکل ۷-۱۷ الگوریتم Single_Source_Shortest_Paths

پیچیدگی: به روز کردن طول یک مسیر به O(logm) مقایسه نیاز دارد که m اندازه‌ی هرم است. $|V|$ بار نیز عمل برداشتن رأس از هرم انجام خواهد شد و حداکثر $|E|$ بار هم باید به‌روزرسانی کرد (زیرا هر یال حداکثر یک بار باعث تغییر می‌شود)؛ پس تعداد مقایسه‌ها در هرم از $O(|E|\log|V|)$ خواهد شد. بنابراین زمان کل اجرا از $O((|E|+|V|)\log|V|)$ است. توجه کنید که این الگوریتم از الگوریتم مشابه برای گراف‌های بدون‌دور کندتر است، چراکه در آنجا، رأس بعدی از صفی (با یک ترتیب دل‌خواه) گرفته می‌شد و نیازی به تغییر مقدارهای یافته‌شده نبود.

☐ مثال ۷-۱

نمونه‌ای از اجرای الگوریتم Single_Source_Shortest_Paths در شکل ۷-۱۸ آمده است. سطر نخست، تنها مسیرهای تک‌یالی خارج‌شونده از V را شامل می‌شود. کوتاه‌ترین مسیر برگزیده‌شده در این مورد، ما را به رأس a می‌رساند. سطر دوم، با در نظر گرفتن همه‌ی مسیرهای تک‌یالی خارج‌شونده از v یا a، به‌روزرسانی مسیرها را انجام می‌دهد و کوتاه‌ترین مسیر برگزیده‌شده، ما را به رأس c می‌رساند. به همین ترتیب در هر سطر، رأس تازه‌ای برگزیده شده، کوتاه‌ترین مسیر فعلی از v به رأس دیگری پیدا می‌شود. اعداد درون دایره کوتاه‌ترین مسیرهای شناخته‌شده هستند.

☐



	v	a	b	c	d	e	f	g	h
a	۰	۱	۵	∞	۹	∞	∞	∞	∞
c	۰	(۱)	۵	۳	۹	∞	∞	∞	∞
b	۰	(۱)	۵	(۳)	۷	∞	۱۲	∞	∞
d	۰	(۱)	(۵)	(۳)	۷	۸	۱۲	∞	∞
e	۰	(۱)	(۵)	(۳)	(۷)	۸	۱۲	۱۱	∞
h	۰	(۱)	(۵)	(۳)	(۷)	(۸)	۱۲	۱۱	۹
g	۰	(۱)	(۵)	(۳)	(۷)	(۸)	۱۲	۱۱	(۹)
f	۰	(۱)	(۵)	(۳)	(۷)	(۸)	۱۲	(۱۱)	(۹)

شکل ۷-۱۸ نمونه‌ای از اجرای الگوریتم کوتاه‌ترین مسیر از یک رأس به رأس‌های دیگر (توجه کنید هر بار در ستون سمت چپ، نام رأسی قرار دارد که درآن گام به V_k افزوده شده است - مترجمان)

توجه: گاهی این نوع الگوریتم را جست‌وجوی اولویت‌دار می‌گویند، چراکه هر رأس یک اولویت نیز دارد (در اینجا اولویت هر رأس، طول کوتاه‌ترین مسیر شناخته‌شده از رأس مبدأ تا آن رأس است) و رأس‌ها برحسب این اولویت بررسی می‌شوند. هنگام در نظر گرفتن یک رأس، همه‌ی یال‌های گذرنده از آن رأس نیز بررسی می‌شوند. این بررسی ممکن است برخی اولویت‌ها را تغییر دهد. برای انجام این تغییرات، باید هنگام جست‌وجو، طول بهترین مسیر شناخته‌شده تا هر رأس را داشته باشیم. جست‌وجوی

اولویت‌دار از جست‌وجوی عادی پرهزینه‌تر است، اما برای مسأله‌هایی که گراف آن‌ها وزن دارند، سودمند است.

هر بار کوتاه‌ترین مسیر از رأس v به یک رأس دیگر را یافتیم تا سرانجام کوتاه‌ترین مسیر از رأس v به همه‌ی رأس‌های دیگر پیدا شد. هر مسیر تازه با یک یال شناخته می‌شود که این یال به کوتاه‌ترین مسیر شناخته‌شده تا یک رأس افزوده می‌گردد و کوتاه‌ترین مسیر تا یک رأس نو را مشخص می‌کند. همه‌ی این یال‌ها با یکدیگر درختی را تشکیل می‌دهند که ریشه‌ی آن v است (تمرین ۷-۶). این درخت را درخت کوتاه‌ترین مسیر می‌گویند. این درخت، هنگام رویارویی با انواع گوناگونی از مسأله‌های مربوط به مسیر، بااهمیت است.

۷-۶ درخت پوشای کمینه

شبکه‌ای را در نظر بگیرید که امکان ارتباط دوسویه را بین همه‌ی رایانه‌های خود فراهم می‌کند. فرستادن یک پیام از راه هر اتصال هزینه‌ای با مقدار مثبت دارد. فرض می‌کنیم هزینه‌ی فرستادن پیام روی یک اتصال در هر دو جهت یکسان است. می‌خواهیم از یک رایانه‌ی دل‌خواه، پیامی را به تمام رایانه‌های دیگر بفرستیم (یعنی آن پیام را در شبکه پخش کنیم). هزینه‌ی کل، حاصل جمع تک‌تک هزینه‌هایی است که برای انجام کار صرف فرستادن پیام‌ها روی اتصال‌های گوناگون می‌کنیم. (گاهی هم مانند تمرین ۷-۶۳، هزینه، زمان لازم برای رسیدن پیام به تمام رایانه‌های دیگر است.) می‌توانیم این شبکه را با گرافی بدون‌جهت نشان دهیم که مقدار مثبت هزینه، برچسب یال‌های آن است. مسأله، پیدا کردن یک زیرگراف همبند (متناظر با اتصالات به‌کاررفته در این فرایند) است که همه‌ی رأس‌ها را در بر می‌گیرد و مجموع هزینه‌های آن (یعنی مجموع برچسب یال‌های زیرگراف) در بین تمام زیرگراف‌های ممکن کمینه شود. به آسانی می‌توان دریافت که این زیرگراف باید درخت باشد؛ چون اگر در آن دوری وجود داشته باشد، می‌توانیم یکی از یال‌های این دور را برداریم؛ زیرگراف بازهم همبند خواهد بود، اما چون تمام برچسب‌ها مثبت هستند، هزینه‌ی آن کم‌تر خواهد شد. به این زیرگراف، درخت پوشای کمینه (MCST) می‌گوییم که به جز پخش پیام کاربردهای بسیار دیگری نیز دارد.[۱] برای سادگی، فرض می‌کنیم تمام هزینه‌ها با یکدیگر متفاوتند. این فرض، باعث یکتا شدن MCST می‌شود (تمرین ۷-۱۱) و بحث را ساده‌تر می‌کند. بدون این فرض هم می‌توانیم بدون هیچ تغییری الگوریتم را به کار ببریم؛ تنها هنگام دیدن چند یال هم‌هزینه، باید یکی از آن‌ها را به دل‌خواه برگزینیم (همان‌گونه

۱- فرض می‌کنیم کل گراف را می‌شناسیم، اما در دنیای واقعی، شبکه‌های محلیِ متصل به یک شبکه‌ی سراسری، تنها اطلاعات مربوط به خودشان را دارند؛ بنابراین به الگوریتمی توزیع‌شده نیاز خواهیم داشت.

که برای حذف یک دور می‌توانیم هر یک از یال‌های آن را به دل‌خواه کنار بگذاریم). اثبات درستی در این حالت، پیچیده‌تر است.

مسأله: یک گراف وزن‌دار، همبند و بدون‌جهت مانند G=(V,E) داده شده است. یک درخت پوشای کمینه (مانند T) را در آن بیابید.

(توجه کنید در اینجا وزن یال‌ها هزینه است.) فرض سرراست استقرا چنین است:

فرض استقرا (نخستین تلاش): می‌دانیم چگونه برای گراف همبندی با کم‌تر از m یال، MCST را بیابیم.

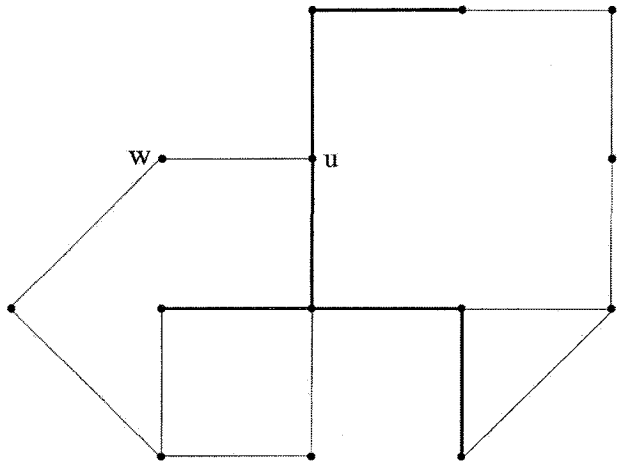
حالت پایه بدیهی است. اگر یک مسأله‌ی MCST با m یال داده شده باشد، چگونه می‌توانیم آن را به مسأله‌ای با کم‌تر از m یال کاهش دهیم؟ ادعا می‌کنیم کم‌هزینه‌ترین یال باید در MCST باشد، چراکه اگر این‌گونه نباشد، افزودن آن به MCST یک دور ایجاد می‌کند. با برداشتن هر یال دیگری از این دور، باز یک درخت داریم اما با هزینه‌ای کم‌تر، که با کمینه بودن هزینه در MCST تناقض دارد. پس یکی از یال‌های MCST را می‌شناسیم. می‌توانیم این یال را برداریم تا بتوان از استقرا روی بقیه‌ی گراف (که حالا تعداد یال‌هایش کم‌تر است) سود جست. آیا استقرا به درستی به کار گرفته شده است؟ نه! چون پس از حذف یک یال باید مسأله‌ای متفاوت با مسأله‌ی اصلی حل شود؛ چراکه هم با برگزیدن یک یال، انتخاب یال‌های دیگر محدود می‌شود و هم شاید پس از برداشتن یک یال، دیگر گراف همبند نباشد. دیگر بیش از این نمی‌توانیم بر این موضوع پافشاری کنیم که فرض استقرا باید به دقت تعریف شده، درستی آن در هر گام ثابت شود.

راه‌حل، اصلاح فرض استقراست. می‌دانیم چگونه یال نخست را برگزینیم، اما نمی‌توانیم به سادگی آن را کنار بگذاریم و فراموش کنیم، چراکه گزینش یال‌های دیگر به آن وابسته است. بنابراین به جای حذف این یال، آن را پس از برگزیدن، علامت می‌زنیم و از این واقعیت (برگزیده شدن این یال) در الگوریتم سود می‌جوییم. الگوریتم، هر بار با برگزیدن یکی از یال‌های MCST پیش خواهد رفت. پس استقرا، نه بر روی اندازه‌ی گراف، که بر روی تعداد یال‌های برگزیده‌شده در یک گراف خاص است.

فرض استقرا: گراف همبند G=(V,E) داده شده است و می‌دانیم چگونه زیرگراف T با کمتر از k رأس را (k<|V|-1) در آن چنان بیابیم که T هم درخت باشد و هم زیرگرافی از درخت پوشای کمینه‌ی G.

بحث حالت پایه برای این فرض استقرا همان است که در حالت پیش بیان شد (که یال نخست را برمی‌گزیند). فرض می‌کنیم درخت T را که برآورنده‌ی فرض استقراست، یافته باشیم و حالا لازم است با یالی دیگر، آن را گسترش دهیم. چگونه می‌توان یالی یافت که بی‌گمان متعلق به MCST باشد؟ بازهم همان استدلالِ یافتن نخستین یال را به کار می‌گیریم می‌دانیم T بخشی از MCST است. بنابراین در MCST دست‌کم یک یال وجود دارد که T را به رأسی متصل می‌کند که آن رأس در T

نیست. می‌کوشیم یکی از این یال‌ها را بیابیم. E_k را مجموعه‌ی همه‌ی یال‌هایی بگیرید که T را به رأس‌هایی که در T نیستند، متصل می‌کنند. ادعا می‌کنیم کم‌هزینه‌ترین یال از E_k متعلق به MCST است. این یال را (u,w) می‌نامیم (شکل ۷-۱۹ را ببینید). چون MCST یک درخت پوشاست، مسیری یکتا از u به w در آن وجود دارد (در درخت بین هر دو رأس، مسیری یکتا وجود دارد). اگر (u,w) متعلق به MCST نباشد، پس این یال در مسیر u به w هم قرار ندارد، اما چون $u \in T$ و $w \notin T$، پس باید دست‌کم یک یال مانند (x,y) در این مسیر وجود داشته باشد که T را به رأسی خارج آن متصل کند. هزینه‌ی این یال بیش از هزینه‌ی (u,w) است، چراکه (u,w) در بین تمام یال‌ها کم‌ترین هزینه را داشت. اینک می‌توانیم همان استدلالِ زمان برگزیدن یال نخست را به کار گیریم. اگر یال (u,w) را به MCST افزوده، یال (x,y) را از آن برداریم، به درخت پوشای دیگری با هزینه‌ی کم‌تر می‌رسیم؛ که تناقض است.



شکل ۷-۱۹ یافتن یال بعدی در MCST

پیاده‌سازی: این الگوریتم، بسیار شبیه الگوریتم یافتن کوتاه‌ترین مسیر موجود از یک رأس به رأس‌های دیگر (ارائه‌شده در بخش پیش) است. نخست، کم‌هزینه‌ترین یال را برمی‌گزینیم و T را درختی در نظر می‌گیریم که دارای همین یک یال است. هر بار بین یال‌هایی که T را به رأسی خارج از آن متصل می‌کنند، کم‌هزینه‌ترین را می‌یابیم. در الگوریتم کوتاه‌ترین مسیر، کوتاه‌ترین مسیری را می‌یافتیم که T را به رأسی خارج از آن متصل می‌کرد. بنابراین تنها تفاوت الگوریتم MCST با الگوریتم کوتاه‌ترین مسیر این است که کمینه‌سازی روی هزینه‌ی یال انجام می‌شود؛ نه روی طول مسیر. باقی‌مانده‌ی الگوریتم کم و بیش همان الگوریتم کوتاه‌ترین مسیر است. برای هر رأس w که $w \notin T$، کم هزینه‌ترین یالی را که به رأسی از T می‌رسد، نگه می‌داریم (اگر چنین یالی وجود نداشت، به آن رأس هزینه‌ی ∞ را نسبت می‌دهیم). هر بار با کم‌هزینه‌ترین یالی که رأسی خارج از T را به T متصل می‌کند، رأس تازه‌ای به T می‌افزاییم. (در اینجا، این رأس را w نامیده‌ایم.) سپس همه‌ی یال‌های

گذرنده از w را بررسی می‌کنیم؛ اگر هزینه‌ی یال (w,z) (به ازای هر رأس z که $z \notin T$) از هزینه‌ی بهترین یال فعلیِ منتهی به z کم‌تر بود، هزینه‌ی z را تغییر می‌دهیم. این الگوریتم در شکل ۷-۲۰ ارائه شده است.

پیچیدگی: پیچیدگی این الگوریتم با پیچیدگی یافتن کوتاه‌ترین مسیر از یک رأس به رأس‌های دیگر (که در بخش پیش ارائه شد) یکسان است؛ پس زمان اجرا در بدترین حالت از $O((|V|+|E|)\log|V|)$ است.

الگوریتم: MCST(G)

ورودی: G (یک گراف وزن‌دار بدون‌جهت)

خروجی: T (یک درخت پوشای کمینه در G)

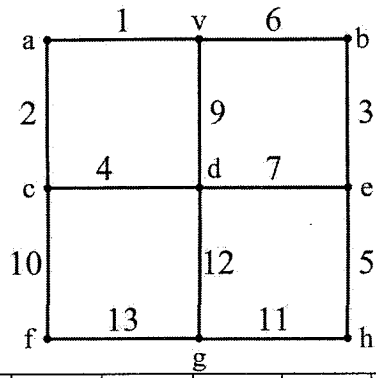
```
begin
    در مقداردهی اولیه، T را مجموعه‌ای تهی قرار بده;
    for G از w هر رأس do
        w.Mark := false;   {اگر w در T باشد، w.Mark، true خواهد شد.}
        w.Cost := ∞;
    کم‌هزینه‌ترین یال G را (x,y) نام‌گذاری کن;
    x.Mark := true;  {علامت‌گذاری y در حلقه‌ی اصلی انجام خواهد شد.}
    for (x,z) هر یال do
        z.Edge := (x,z);
        z.Cost := (x,z) هزینه‌ی یال;   {همان هزینه‌ی یال z.Edge}
    while رأس علامت‌نخورده‌ای وجود دارد do
            رأس علامت‌نخورده‌ای را که هزینه‌اش کمینه است، w بنام;
            if w.Cost = ∞ then
              print " G همبند نیست."
              halt
            else
                w.Mark := true;
                w.Edge را به درخت T بیفزا;
                {حال، هزینه‌ی رأس‌های علامت‌نخورده‌ی متصل به w را به روز می‌کنیم.}
                for (w,z) هر یال do
                    if not z.Mark then
                      if (w,z) هزینه‌ی یال < z.Cost then
                        z.Edge := (w,z);
                        z.Cost := (w,z) هزینه‌ی یال
end
```

شکل ۷-۲۰ الگوریتم MCST

☐ مثال ۷-۲

نمونه‌ای از اجرای الگوریتم MCST در شکل ۷-۲۱ نشان داده شده است. در هر گام، رأس نشان داده‌شده در ستون نخست جدول به درخت MCST افزوده می‌شود. رأس نخست، v است. پس از برگزیدن v، یال‌های گذرنده از آن همراه با هزینه‌هایشان در نظر گرفته می‌شوند. در هر سطر، رأسی را برمی‌گزینیم که می‌توان با کم هزینه‌ترین یال، آن را به MCST متصل کرد. در هر گام، بهترین یال‌های فعلی که به رأس‌هایی علامت‌نخورده (یعنی خارج MCST) متصل هستند (و هزینه‌ی آن‌ها) به روز می‌شود (این یال‌ها با رأس‌های انتهایی خود نشان داده شده‌اند).

☐



	v	a	b	c	d	e	f	g	h
v	–	v(۱)	v(۶)	∞	v(۹)	∞	∞	∞	∞
a	–	–	v(۶)	a(۲)	v(۹)	∞	∞	∞	∞
c	–	–	v(۶)	–	c(۴)	∞	c(۱۰)	∞	∞
d	–	–	v(۶)	–	–	d(۷)	c(۱۰)	d(۱۲)	∞
b	–	–	–	–	–	b(۳)	c(۱۰)	d(۱۲)	∞
e	–	–	–	–	–	–	c(۱۰)	d(۱۲)	e(۵)
h	–	–	–	–	–	–	c(۱۰)	h(۱۱)	–
f	–	–	–	–	–	–	–	h(۱۱)	–
g	–	–	–	–	–	–	–	–	–

شکل ۷- ۲۱ نمونه‌ای از اجرای الگوریتم درخت پوشای کمینه

توجه: الگوریتم یافتن MCST، نمونه‌ای – هرچند ناقص – از شیوه‌ای است که آن را «آزمندانه» یا «حریصانه» گویند. فرض کنید با مجموعه‌ای از عناصر سر و کار داریم که هر یک از آن‌ها هزینه‌ای هم دارد و ما می‌خواهیم عنصرهایی را با بیش‌ترین (یا کم‌ترین) هزینه به گونه‌ای بیابیم که شرایط ویژه‌ای را برآورده سازند. در مسأله‌ی MCST، هر عنصر، یالی از گراف است و شرط، این است که یال‌ها باید

درختی پوشا تشکیل دهند. در شیوه‌ی آزمندانه باید آزمند بود؛ یعنی در هر گام، عنصری را برگزید که از نظر هزینه دارای بهترین شرایط باشد. در الگوریتم MCST، شرایط یا محدودیت‌های بیش‌تری برای گزینش یال‌ها داشتیم؛ یعنی تنها یال‌هایی را بررسی می‌کردیم که به درخت فعلی متصل بودند. بنابراین الگوریتم MCST کاملاً آزمندانه نیست. برای یافتن MCST می‌توانیم در هر گام کم‌هزینه‌ترین یال را از هر جای گراف برگزینیم؛ مشروط به این که باعث ایجاد دور نگردد (تمرین ۷-۵۹). روش آزمندانه همیشه هم به راه‌حل بهینه نمی‌انجامد، بلکه معمولاً یافتن راه‌حلی نزدیک به بهینه با این روش، نیازمند خلاقیت و ابتکار در طراحی الگوریتم است؛ هرچند گاهی نیز مانند الگوریتم MCST می‌توان با روش آزمندانه به راه‌حل بهینه دست یافت.

۷-۷ کوتاه‌ترین مسیرها بین تمام زوج‌رأس‌های گراف

اینک مسأله‌ی یافتن کوتاه‌ترین مسیرها بین تمام زوج‌رأس‌های گراف را بررسی می‌کنیم.

مسأله: گراف وزن‌دار G=(V,E) با وزن‌های نامنفی داده شده است (که ممکن است جهت‌دار یا بدون‌جهت باشد). کوتاه‌ترین مسیر بین هر دو رأس را بیابید.

از آنجا که سخن از کوتاه‌ترین مسیر به میان آمده است، پس به جای «وزن» واژه‌ی «طول» را به کار می‌بریم. نام این مسأله «کوتاه‌ترین مسیرها بین تمام زوج‌رأس‌های گراف» است. برای سادگی، تنها، طول کوتاه‌ترین مسیرها را به جای خود آن‌ها می‌یابیم. گراف را جهت‌دار می‌گیریم؛ اما همین استدلال را می‌توان برای گراف‌های بدون‌جهت نیز به کار برد. همه‌ی وزن‌ها (طول‌ها) را نامنفی فرض می‌کنیم (در تمرین ۷-۷۳ طول‌های منفی نیز در نظر گرفته شده‌اند).

بازهم بیایید طبق معمول، کار را با فرضی سرراست برای استقرا آغاز کنیم. استقرا می‌تواند بر روی یال‌ها، یا بر روی رأس‌ها بنا شود. در فرایند یافتن کوتاه‌ترین مسیر، اثر افزودن یالی تازه مانند (u,w) به گراف چیست؟ شاید این یال، خود، کوتاه‌ترین مسیر بین (u,w) و شاید هم یالی از کوتاه‌ترین مسیر بین دو رأس دیگر باشد. در بدترین حالت، باید برای هر دو رأس v_1 و v_2 طول کوتاه‌ترین مسیر از v_1 تا u و طول (u,w) و طول کوتاه‌ترین مسیر از w تا v_2 را جمع بزنیم و سپس ببینیم که آیا حاصل این جمع از طول کوتاه‌ترین مسیر شناخته‌شده بین v_1 و v_2 کم‌تر است یا نه. به ازای هر یال تازه ممکن است تعداد بررسی‌های لازم از $O(|V|^2)$ شود. پس در بدترین حالت، زمان اجرا از $O(|E||V|^2)$ خواهد بود. (تعداد یال‌ها می‌تواند از $O(|V|^2)$ باشد، پس الگوریتم از $O(|V|^4)$ خواهد شد.)

در فرایند یافتن کوتاه‌ترین مسیر، اثر افزودن رأسی تازه مانند u به گراف چیست؟ نخست باید طول کوتاه‌ترین مسیرها از u به تمام رأس‌های دیگر به همراه طول کوتاه‌ترین مسیرها از همه‌ی رأس‌های دیگر به رأس u پیدا شود. چون همه‌ی کوتاه‌ترین مسیرهایی را که از u نمی‌گذرند، از پیش

می‌شناسیم، می‌توانیم به ترتیبی که خواهیم گفت، کوتاه‌ترین مسیر از u به w را بیابیم. تنها لازم است نخستین یال این مسیر را (که از u خارج می‌شود) پیدا کنیم. اگر این یال را (u,v) بنامیم، طول کوتاه‌ترین مسیر از u به w برابر مجموع طول (u,v) و طول کوتاه‌ترین مسیر از v به w است (طول کوتاه‌ترین مسیر از v به w را از پیش می‌دانیم). بنابراین، پس از بررسی این طول‌ها برای تمام همسایگان u، کوچک‌ترین مقدار را برمی‌گزینیم. با همین شیوه می‌توان کوتاه‌ترین مسیر از w به u را نیز یافت، اما هنوز کار به پایان نرسیده است؛ چراکه به ازای هر دو رأس دیگری از گراف ممکن است مسیری کوتاه‌تر، از راه u به وجود آمده باشد. برای هر دو رأس v و w، طول کوتاه‌ترین مسیر از v به u را با طول کوتاه‌ترین مسیر از u به w جمع می‌زنیم و حاصل را با طول کوتاه‌ترین مسیر شناخته‌شده مقایسه می‌کنیم. پس هنگام در نظر گرفتن هر رأس تازه باید تعدادی مقایسه و جمع انجام دهیم. این تعداد از $O(|V|^2)$ است و زمان اجرای کل الگوریتم از $O(|V|^3)$ خواهد شد. بنابراین استقرا روی رأس‌ها از استقرا روی یال‌ها بهتر است، اما با ثابت نگه داشتن تعداد یال‌ها و رأس‌ها و اعمال محدودیت روی نوع مسیرهای مجاز، این مسأله راه‌حل استقرایی بهتری هم دارد. استقرا بر پایه‌ی محدودیت‌های روی مسیرهاست. طی گام‌های استقرا این محدودیت‌ها به گونه‌ای کاهش می‌یابند که در پایان، همه مسیرهای ممکن بررسی شده باشند. به رأس‌ها برچسب‌هایی از ۱ تا $|V|$ می‌زنیم. به مسیری از u به w که بزرگ‌ترین برچسب رأس‌های درونی آن (یعنی رأس‌های مسیر به جز خود u و w) k باشد، یک «k-مسیر» می‌گوییم. پس، در حالت خاص، ۰-مسیر یک یال است (چون این مسیر نمی‌تواند رأس درونی داشته باشد).

فرض استقرا: اگر به ازای یک مقدار k که از m کوچک‌تر است، تنها k-مسیرها را بین هر دو رأس، مجاز بدانیم؛ آنگاه طول کوتاه‌ترین مسیر مجاز بین هر دو رأس را می‌دانیم.

در حالت پایه، m برابر ۱ است که در این حالت باید تنها مسیرهای تک‌یالی را در نظر بگیریم و راه‌حل به سادگی به دست می‌آید. فرض استقرا بر روی m بنا می‌شود و می‌کوشیم آن را به m+۱ گسترش دهیم. حال باید همه‌ی k-مسیرهایی را در نظر بگیریم که k<m+1؛ یعنی تنها m-مسیرها اضافه شده‌اند. ما باید کوتاه‌ترین m-مسیرها را بین تمام زوج‌رأس‌ها بیابیم و ببینیم آیا این مسیرهای تازه، بهتر از مسیرهای پیشین هستند یا نه. رأسی را که برچسب m خورده است، v_m بنامید. هر یک از کوتاه‌ترین m-مسیرها باید دقیقاً یک بار از این رأس بگذرند. کوتاه‌ترین m-مسیر بین u و w از اضافه کردن کوتاه‌ترین j-مسیر بین v_m و w (برای یک j کوچک‌تر از m) به انتهای کوتاه‌ترین k-مسیر بین u و v_m (برای یک k کوچک‌تر از m) به دست می‌آید (j لزوماً برابر k نیست). بنا به استقرا، طول همه‌ی کوتاه‌ترین k-مسیرها را برای همه‌ی kهای کم‌تر از m می‌دانیم. بنابراین تنها کافی است برای یافتن کوتاه‌ترین m-مسیر بین u و w، دو طول گفته‌شده را جمع بزنیم. این الگوریتم، هم از الگوریتمی که بر

پایه‌ی فرض سرراست استقرا روی رأس‌ها ساخته می‌شود، (به اندازه‌ی یک ضریب ثابت) سریع‌تر است و هم، راحت‌تر می‌توان برنامه‌ی آن را نوشت. الگوریتم در شکل ۷-۲۲ داده شده است.

در دو حلقه‌ی داخلی‌ترِ الگوریتم همه‌ی زوج‌رأس‌ها بررسی می‌شوند. توجه کنید که بررسی زوج‌رأس‌ها با هر ترتیب دل‌خواهی امکان‌پذیر است، چراکه این بررسی‌ها مستقل از یکدیگرند. این انعطاف‌پذیری، برای مثال در الگوریتم‌های موازی اهمیت دارد.

الگوریتم: All_Pairs_Shortest_Paths(Weight)

ورودی: Weight (یک ماتریس همسایگی $n \times n$ که گرافی وزن‌دار را نشان می‌دهد.) {Weight[x,y] وزن یال (x,y) را نشان می‌دهد. اگر این مقدار ∞ باشد؛ یعنی در گراف یال (x,y) وجود ندارد. برای هر x ، Weight[x,x] صفر است.}

خروجی: در پایان کار، ماتریس Weight دربردارنده‌ی طول کوتاه‌ترین مسیرهاست.

```
begin
    for m := 1 to n do {حلقه‌ی دنباله‌ی استقرا}
        for x := 1 to n do
            for y := 1 to n do
                if Weight[x,m] + Weight[m,y] < Weight[x,y] then
                    Weight[x,y] := Weight[x,m] + Weight[m,y]
end
```

شکل ۷-۲۲ الگوریتم All_Pairs_Shortest_Paths

پیچیدگی: این الگوریتم برای هر m به ازای هر دو رأس، تنها یک جمع و یک مقایسه انجام می‌دهد. طول دنباله‌ی استقرا $|V|$ است، پس تعداد کل جمع‌ها (و نیز مقایسه‌ها) حداکثر $|V|^3$ خواهد شد. زمان اجرای الگوریتم «کوتاه‌ترین مسیر از یک رأس به رأس‌های دیگر» از $O(|E|\log|V|)$ بود. اگر گراف چگال (یعنی پر و متراکم) باشد، تعداد یال‌های آن از $\Omega(n^2)$ خواهد بود. هنگامی که با چنین گراف‌هایی سر و کار داریم، این الگوریتم بهتر از آن است که الگوریتم «کوتاه‌ترین مسیر از یک رأس به رأس‌های دیگر» را برای تک‌تک رأس‌ها به کار گیریم؛ هرچند می‌توان الگوریتم «کوتاه‌ترین مسیر از یک رأس به رأس‌های دیگر» را در زمانی از $O(|V|^2)$ هم پیاده‌سازی کرد (تمرین ۷-۴۳) که با به کار بردن آن برای تمام رأس‌ها زمان اجرای الگوریتم «کوتاه‌ترین مسیر بین همه‌ی زوج‌رأس‌ها» از $O(|V|^3)$ خواهد شد. چون پیاده‌سازی الگوریتم این بخش ساده‌تر است، برای گراف‌های چگال بهتر عمل می‌کند؛ اما اگر گراف تقریباً تنک یا خلوت باشد، با $|V|$ بار به کار بردن الگوریتم «کوتاه‌ترین مسیر از یک رأس به رأس‌های دیگر» به زمان اجرای بهتری (از $O(|E||V|\log|V|)$) می‌رسیم.

۷-۸ بسط ترایا

بسط ترایا برای یک گراف جهت‌دار مانند G=(V,E)، گراف جهت‌دار C=(V,F) است که در آن یالی از v به w وجود دارد، اگر و تنها اگر مسیری جهت‌دار از v به w در G وجود داشته باشد. برای نمونه، می‌توان الگوریتم بسط ترایا را به مسأله‌ی امنیت حساب کاربران که در آغاز فصل گفته شد، ربط داد؛ یعنی هر کاربر را با یک رأس و هر اجازه‌ی دسترسی را با یک یال متناظر کرد. بسط ترایای هر کاربر، کاربرانی را مشخص می‌کند که (به طور مستقیم یا غیرمستقیم) اجازه‌ی دسترسی به حساب این کاربر را دارند. یافتن الگوریتمی کارآمد برای بسط ترایا بااهمیت است، زیرا این موضوع کاربردهای فراوان دیگری هم دارد.

مسأله: بسط ترایای گراف جهت‌دار داده‌شده‌ی G=(V,E) را بیابید.

این مسأله را به یاری کاهش حل می‌کنیم؛ یعنی هر نمونه از مسأله‌ی بسط ترایا را به نمونه‌ای از یک مسأله‌ی دیگر که حل آن را از پیش می‌دانیم، تبدیل می‌کنیم. (این کاهش از نوع کاهش‌های فصل ۱۰ است، نه کاهش اندازه‌ی مسأله برای استقرا که پیش‌تر گرفته شد - مترجمان) سپس از روی راه‌حل آن مسأله‌ی دیگر، پاسخ مسأله‌ی اصلی را می‌یابیم. کاهش به مسأله‌ی «کوتاه‌ترین مسیر بین تمام زوج‌رأس‌ها» صورت می‌گیرد.

$G'=(V,E')$ را یک گراف جهت‌دار و کامل بگیرید (پس بین هر دو رأس در هر دو جهت، یال وجود دارد). به هر یال e در E'، بسته به آن که e متعلق به E باشد یا نه، طول ۰ یا ۱ را نسبت می‌دهیم. حال مسأله‌ی «کوتاه‌ترین مسیر بین هر دو رأس G'» را حل می‌کنیم. از آنجا که طول تمام یال‌های G، در G' صفر است؛ پس اگر در G، مسیری از v به w وجود داشته باشد؛ طول این مسیر در G' صفر خواهد شد. بنابراین در G مسیری بین v و w وجود دارد، اگر و تنها اگر در G' طول کوتاه‌ترین مسیر بین v و w صفر باشد. به این ترتیب، پاسخ مسأله‌ی «کوتاه‌ترین مسیر بین همه‌ی زوج‌رأس‌ها» به طور مستقیم به پاسخ مسأله‌ی بسط ترایا تبدیل می‌شود.

روش کاهش مسأله‌ها را به طور گسترده در فصل ۱۰ بررسی خواهیم کرد. در اینجا کاهش را بیش‌تر به این منظور به کار بردیم تا با یک مثال ساده، روش را توضیح داده باشیم، وگرنه تبدیل مستقیم الگوریتم کوتاه‌ترین مسیر بین هر دو رأس، به الگوریتم بسط ترایا (نشان داده‌شده در شکل ۷-۲۳) کار ساده‌ای است.

هنگامی که می‌گوییم یک مسأله به مسأله‌ی دیگری کاهش می‌یابد، یعنی راه‌حل مسأله‌ی نخست آن قدر کلی است که راه‌حل مسأله‌ی دیگر هم در آن می‌گنجد؛ اما معمولاً راه‌حل‌های کلی‌تر پرهزینه‌تر هستند. به مواردی هم برخوردیم که حل مسأله‌ی کلی‌تر آسان‌تر بود، اما اغلب، هنگامی چیز

بیش‌تری به دست می‌آوریم که هزینه‌ی بیش‌تری هم پرداخت کرده باشیم. هنگام بهره‌گیری از کاهش همواره باید با توجه به ویژگی‌های مسأله برای بهبود راه‌حل بکوشیم.

الگوریتم: Transitive_Closure(A)

ورودی: A (یک ماتریس همسایگی $n \times n$ که نشان‌دهنده‌ی گرافی جهت‌دار است.)
{اگر یال (x,y) متعلق به گراف باشد، A[x,y]، true وگرنه false است. توجه کنید که برای هرx، A[x,x]، true است.}

خروجی: در پایان الگوریتم، ماتریس A نشانگر بسط ترایای گراف است.

```
begin
    for m := 1 to n do  {حلقه‌ی دنباله‌ی استقرا}
        for x := 1 to n do
            for y := 1 to n do
                if A[x,m] and A[m,y] then A[x,y] := true
                {در الگوریتم بعدی این گام را بهبود خواهیم داد.}
end
```

شکل ۷-۲۳ الگوریتم Transitive_Closure

گام اصلی الگوریتم، یعنی دستور if را در نظر بگیرید. این دستور هم A[x,m] و هم A[m,y] را می‌آزماید و تنها هنگام درستی هر دوی آن‌ها، دستور پس از then اجرا می‌گردد. این دستور if به ازای هر زوج‌رأس n بار اجرا می‌شود. بهبود این دستور در بهبود الگوریتم بسیار اثرگذار است. آیا لازم است هر بار، هم درستی A[x,m] و هم درستی A[m,y] را بیازماییم؟ A[x,m] تنها وابسته به x و m، اما A[m,y] تنها وابسته به m و y است. پس برای هر x و m مشخص تنها یک بار بررسی A[x,m] کافی است. اگر A[x,m] نادرست باشد، دیگر نیازی به بررسی A[m,y] نیست و اگر هم A[x,m] درست باشد، دیگر نیازی به بررسی دوباره‌ی آن نیست. پس از بهبود دادن الگوریتم پیش با توجه به این مطلب به الگوریتم ارائه‌شده در شکل ۷-۲۴ می‌رسیم. اگرچه پیچیدگی الگوریتم از نظر جانبی تغییر نمی‌کند، اما الگوریتم تازه تقریباً دو برابر سریع‌تر است.

پیاده‌سازی: پیاده‌سازی این الگوریتم سرراست است. توجه کنید که آخرین دستور الگوریتم همان اثر عمل or روی سطر xام ماتریس را دارد. در هر خانه‌ی (x,y) از سطر xام، یا مقدار خود آن خانه یا مقدار خانه‌ی(m,y) قرار می‌گیرد. انجام این عمل روی خانه‌های این سطر هم‌ارز or کردن سطر x با سطر m است. از آنجا که عمل or در بسیاری از رایانه‌ها می‌تواند به‌یک‌باره بر روی چندین بیت انجام شود و یک عمل سطری or، سریع‌تر از چندین عمل بیت به بیت است؛ پس تعداد گام‌های این الگوریتم، در عمل از $O(n^3/w)$ خواهد بود که در آن w، اندازه‌ی کلمه - یعنی تعداد بیت‌هایی که می‌توانند هم‌زمان با هم or شوند - است. این مثال، نمونه‌ی ساده‌ای از یک الگوریتم موازی است. درباره‌ی این مطلب در بخش ۹-۵-۳ بازهم بحث خواهد شد.

الگوریتم: Improved_Transitive_Closure(A)

ورودی: A (یک ماتریس همسایگی $n \times n$ که نشان‌دهنده‌ی گرافی جهت‌دار است.)
{اگر در گراف، یال (x,y) وجود داشته باشد، A[x,y]، true وگرنه false است. A[x,x] برای هر x، true است.}

خروجی: در پایان، ماتریس A نشان‌دهنده‌ی بسط ترایای G است.

```
begin
    for m := 1 to n do  {حلقه‌ی دنباله‌ی استقرا}
        for x := 1 to n do
            if A[x,m] then
                for y:= 1 to n do
                    if A[m,y] then A[x,y] := true
end
```

شکل ۷-۲۴ الگوریتم Improved_Transitive_Closure

۷-۹ شیوه‌های تجزیه‌ی گراف

پیش‌تر گونه‌ای از تجزیه‌ی گراف را دیده‌ایم: افراز گراف به مؤلفه‌های همبند. گاهی گراف را بر اساس برخی ویژگی‌ها افراز می‌کنیم. به این ترتیب ممکن است هنگام نیاز به طراحی الگوریتمی برای کار با گراف بتوانیم هر زیرگراف را جداگانه در نظر گرفته، از ویژگی‌های آن برای انجام کار مورد نظر سود جوییم. پیش‌تر به چندین الگوریتم برخوردیم که به عنوان ورودی باید گرافی همبند را دریافت می‌کردند. با افراز گراف به مؤلفه‌های همبند توانستیم الگوریتم‌های گفته‌شده را جداگانه روی هر مؤلفه‌ی همبند به کار گیریم؛ به این ترتیب از پیچیدگی‌های بسیاری پرهیز کردیم. این بخش کتاب، دو نوع تجزیه‌ی دیگر را بررسی می‌کند: مؤلفه‌های دوهمبند برای گراف‌های بدون‌جهت و مؤلفه‌های قویاً همبند برای گراف‌های جهت‌دار. این دو شیوه‌ی تجزیه در طراحی الگوریتم‌ها سودمند هستند؛ به ویژه، این دو نوع تجزیه ارتباط تنگاتنگی با دورهای درون گراف دارند (به ترتیب، دورهای بدون‌جهت و دورهای جهت‌دار). بنابراین هنگامی که یک مسأله‌ی گراف به دور ربط داشته باشد (که بسیاری از مسأله‌های گراف این‌گونه هستند) بهره‌گیری از این تجزیه‌ها ایده‌ی خوبی است. این تجزیه‌ها همواره هم به کار نمی‌آیند، اما دست‌کم باید بررسی شوند. در این بخش گراف‌ها را همبند فرض می‌کنیم.

۷-۹-۱ مؤلفه‌های دوهمبند

مفهوم دوهمبندی از گسترش طبیعی مفهوم عادی همبند بودن به دست می‌آید. یک گراف بدون‌جهت همبند است، اگر از هر رأس آن مسیری به هر رأس دیگر وجود داشته باشد. یک گراف بدون‌جهت دوهمبند است، اگر از هر رأس آن دست‌کم دو مسیر به هر رأس دیگر وجود داشته باشد و رأس‌های این دو مسیر جداازهم باشند. بنابراین به طور شهودی می‌توان گفت همبندی در گراف‌های دوهمبند بیش‌تر است؛ زیرا اگر بنا به دلیلی نتوانیم یکی از این دو مسیر را به کار ببریم، بازهم دو رأس واقع در دو سر این مسیر به یکدیگر متصل هستند. خواهیم دید هنگامی هم که گراف دوهمبند نباشد، می‌توان آن را به زیرگراف‌هایی دوهمبند افراز کرد. بیش از هر چیز باید به عمل افراز توجه کنیم. در حالت کلی، اگر بین هر دو رأس (غیرمجاور - مترجمان) یک گراف بدون‌جهت، دست‌کم k مسیر با رأس‌های جداازهم وجود داشته باشد، آن گراف را k-همبند می‌گویند. اینک چند ویژگی را در گراف‌های k-همبند بررسی می‌کنیم.

نخستین ویژگی مهم گراف‌های k-همبند قضیه‌ای از Menger [۱۹۲۷] است. این قضیه، تعداد مسیرهای با رأس‌های جداازهم در گراف را به تعداد رأس‌هایی که لازم است برای ناهمبند کردن گراف از آن حذف شوند، مربوط می‌کند.

قضیه‌ی Menger

$G=(V,E)$ را گرافی همبند و بدون‌جهت و u و v را دو رأس غیرمجاور در آن بگیرید. حداقل تعداد رأس‌هایی که باید برای قطع اتصال u و v حذف شوند، برابر با حداکثر تعداد مسیرهای با رأس‌های جداازهم بین u و v است. (حذف هر رأس، باعث حذف یال‌های گذرنده از آن هم می‌شود.)

□

قضیه‌ی Whitney [۱۹۳۲] نتیجه‌ای ساده از این قضیه است.

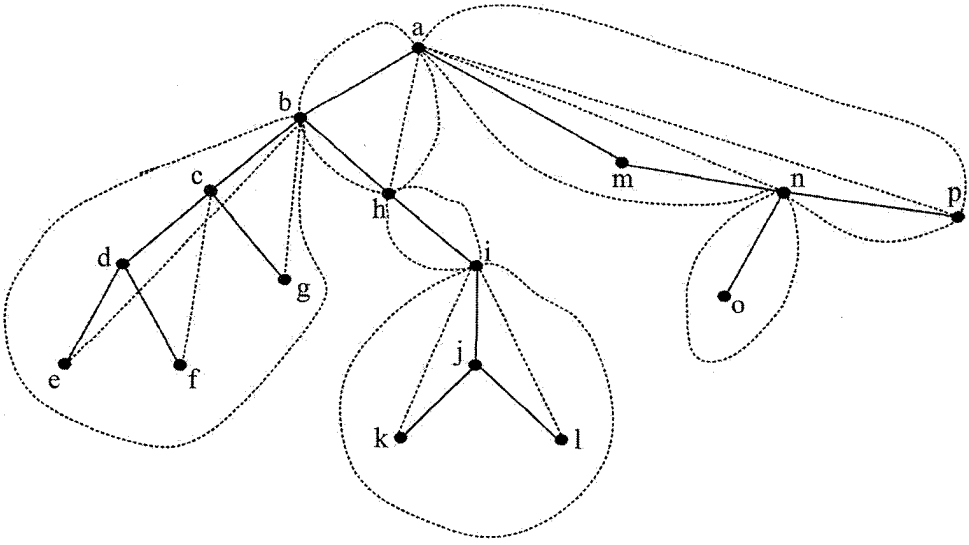
قضیه‌ی Whitney

یک گراف بدون‌جهت k-همبند است، اگر و تنها اگر برای ناهمبند کردن آن حذف دست‌کم k رأس لازم باشد.

□

از قضیه‌ی Whitney شرطی هم‌ارز با k-همبندی به دست می‌آید، پس می‌توانیم هر یک از این دو شرط را به جای یکدیگر به کار ببریم. یک روش اثبات این دو قضیه، در Chartrand و Lesniak [۱۹۸۶] آمده است. (اثبات این دو قضیه، از یک سمت روشن است: اگر k رأس وجود داشته باشند که حذف آن‌ها گراف را ناهمبند کند، آنگاه تعداد مسیرهای با رأس‌های جداازهم از k بیش‌تر نخواهد بود، اما اثبات سمت دیگر آن‌ها پیچیدگی بیش‌تری دارد.)

قضیه‌ی Menger یکی از مهم‌ترین قضیه‌ها در نظریه‌ی گراف است. برای کار ما نتیجه‌ی عمده‌ی دو قضیه‌ی گفته‌شده این است: یک گراف دوهمبند نیست، اگر و تنها اگر رأسی وجود داشته باشد که حذف آن، گراف را ناهمبند کند. چنین رأسی را «نقطه‌ی پیوند» گویند. شکل ۷-۲۵ ساختار گرافی را نشان می‌دهد که دوهمبند نیست. چنین گرافی دست‌کم یک نقطه‌ی پیوند دارد. هر یک از بخش‌های بین نقاط پیوند (این بخش‌ها در شکل با نقطه‌چین مشخص شده‌اند) به تنهایی دوهمبند هستند. این بخش‌های گراف، مؤلفه‌های دوهمبند آن را تشکیل می‌دهند. در آینده این مفهوم را روشن‌تر خواهیم کرد.



شکل ۷-۲۵ ساختار یک گراف نادوهمبند

تعریف: یک مؤلفه‌ی دوهمبند زیرمجموعه‌ای گسترش‌ناپذیر از یال‌های گراف است که زیرگراف القایی آن دوهمبند باشد (یعنی زیرمجموعه‌ی دیگری از یال‌ها وجود نداشته باشد که هم این زیرمجموعه را در بر گیرد و هم زیرگراف القایی آن دوهمبند باشد).

یک مؤلفه‌ی دوهمبند را با مجموعه‌ای از یال‌ها تعریف می‌کنیم، اما یک رأس ممکن است به چندین مؤلفه تعلق داشته باشد. هر نقطه‌ی پیوند بی‌گمان به بیش از یک مؤلفه تعلق دارد. (در واقع با این توصیف، ویژگی دیگری از نقطه‌ی پیوند را روشن کرده‌ایم.) از آنجا که هر یال، تنها به یک مؤلفه تعلق دارد، پس افراز مجموعه یال‌های هر گراف به مؤلفه‌های دوهمبند یکتاست؛ دو لم بعد، وجود و یکتایی چنین افرازی را ثابت می‌کند.

□ لم ۷-۹

هر دو یال e و f متعلق به یک مؤلفه‌ی دوهمبند هستند، اگر و تنها اگر دوری در گراف وجود داشته باشد که هر دو را در بر گیرد. (توجه کنید که یک مؤلفه‌ی دوهمبند ممکن است

تنها از یک یال تشکیل شود؛ این لم، تنها برای مؤلفه‌های دوهمبندی است که دست‌کم دو یال داشته باشند.)

برهان: نخست، نشان می‌دهیم که یک دور همواره به طول کامل درون یک مؤلفه‌ی دوهمبند قرار می‌گیرد؛ چراکه اگر برخی از یال‌های یک دور متعلق به یک مؤلفه‌ی دوهمبند و برخی دیگر متعلق به یک یا چند مؤلفه‌ی دوهمبند دیگر باشند، آنگاه می‌توانیم هر یک از این مؤلفه‌های دوهمبند را با افزودن دیگر یال‌های دور گسترش دهیم. زیرگراف گسترش‌یافته دوهمبند باقی می‌ماند، چراکه نمی‌توان با حذف یک رأس از یک دور، آن دور را ناهمبند کرد. پس دوهمبند بودن زیرگراف گسترش‌یافته با بیشینگیِ مؤلفه‌ی دوهمبند در تناقض است. برای اثبات سمت دیگر قضیه که در آن دو یال متعلق به یک مؤلفه‌ی دوهمبند هستند، روشی ارائه می‌دهیم که با آن بتوان دور دربرگیرنده‌ی آن دو یال را به دست آورد. دو رأس جدید (مصنوعی) روی e و f می‌گذاریم (یعنی اگر e = (v,w)، رأس تازه‌ی z را افزوده، دو یال (v,z) و (z,w) را به جای e قرار می‌دهیم؛ همین کار را برای f هم انجام می‌دهیم). این مؤلفه، هنوز هم یک زیرگراف دوهمبند است، زیرا نقطه‌ی پیوند ندارد. (حذف هر یک از رأس‌های تازه، معادل حذف یال‌های قدیمی است. پس موجب ناهمبند شدن مؤلفه نخواهد شد؛ پس از افزودن این رأس‌های تازه هم، اثر حذف هر یک از رأس‌های قدیمی روی همبندی مؤلفه تغییر نمی‌کند.) بنابراین، بین دو رأس تازه، دو مسیر با رأس‌هایی جداازهم وجود دارد، اما خودِ این دو مسیر دوری دربردارنده‌ی e و f به وجود می‌آورند.

□

□ لم ۷-۱۰

هر یال دقیقاً متعلق به یک مؤلفه‌ی دوهمبند است.

برهان: بی‌گمان، هر یال دست‌کم متعلق به یک مؤلفه‌ی دوهمبند است (شاید این مؤلفه، همین یک یال باشد). هیچ یالی نمی‌تواند متعلق به بیش از یک مؤلفه‌ی دوهمبند باشد، چراکه در آن صورت، در این مؤلفه‌های دوهمبند دورهایی وجود خواهد داشت که این یال را نیز در بر می‌گیرند؛ اما به تازگی دیدیم که چنین چیزی ممکن نیست.

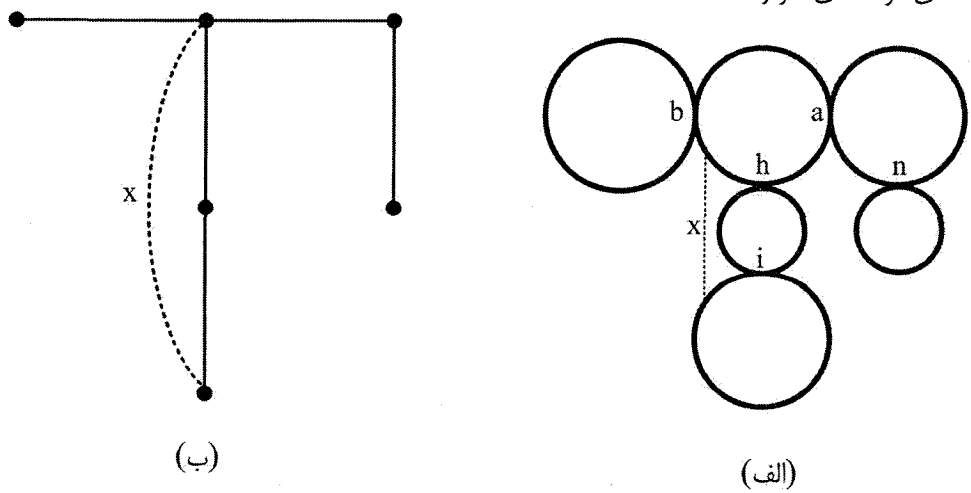
□

می خواهیم افراز به مؤلفه‌های دوهمبند را بیابیم. بیایید طبق معمول با فرض سرراستی برای استقرا کار را آغاز کنیم.

فرض استقرا: برای گراف‌های همبندی با کم‌تر از m یال می‌دانیم چگونه مؤلفه‌های دوهمبند را بیابیم.

یک گراف همبند تک‌یالی، دوهمبند است (حالت پایه). گرافی با m یال را در نظر گرفته، یال دل‌خواه x را برگزینید. x را از گراف کنار می‌گذاریم و بنا به استقرا مؤلفه‌های دوهمبند را می‌یابیم. حال باید روشن کنیم که افزودن دوباره‌ی x چه تأثیری روی افراز دارد. ساده‌ترین حالت هنگامی است که x دو رأس از

یک مؤلفه را به یکدیگر متصل کند (مانند یال (a,n) در شکل ۷-۲۵). در این حالت، افزودن x هیچ اثری روی افراز ندارد (تنها همبندی آن مؤلفه را بیش‌تر می‌کند). حالت آسان دیگر هنگامی است که برداشتن x گراف را کاملاً ناهمبند می‌کند (مانند یال‌های (h,i) و (n,o) در شکل ۷-۲۵). در این حالت، روشن است که هر دو نقطه‌ی انتهایی x نقاط پیوند هستند، پس x خود، یک مؤلفه‌ی دوهمبند است (چنین یالی را پل نامیده‌اند که نامی به‌جاست). روشن است هیچ یک از مؤلفه‌های دیگر تغییر نمی‌کنند. حالت دشوار هنگامی است که برداشتن x گراف را ناهمبند نکند، اما این یال رأس‌هایی از دو مؤلفه‌ی جداگانه را به یکدیگر متصل کند؛ مانند (b,e) در شکل ۷-۲۵. این حالت در شکل ۷-۲۶ (الف) نیز نشان داده شده است. روشن است x دو مؤلفه‌ای را که به یکدیگر متصل کرده است، با چندین مؤلفه‌ی دیگرِ سر راه ادغام می‌کند و موجب تشکیل مؤلفه‌ای بزرگ‌تر می‌شود. پس مسأله، یافتن و ادغام کارآمد همه‌ی مؤلفه‌های سر راه است.



شکل ۷-۲۶ یالی (x) که دو مؤلفه‌ی دوهمبند جداگانه را به یکدیگر متصل می‌کند. (الف) مؤلفه‌های متناظر با گراف شکل ۷-۲۵ همراه با نقاط پیوند آن‌ها (ب) درخت دوهمبندی مؤلفه‌ها

با نگاهی دقیق به شکل‌های ۷-۲۵ و ۷-۲۶ درمی‌یابیم که می‌توان (به صورتی که گفته خواهد شد) از روی مؤلفه‌های دوهمبند یک درخت تعریف کرد. هر مؤلفه‌ی دوهمبند متناظر با یک گره است (آن‌ها را گره نامیده‌ایم تا با رأس‌های گراف اشتباه نشوند). مؤلفه‌ی دل‌خواه R را ریشه‌ی درخت می‌گیریم و کار را آغاز می‌کنیم (در شکل ۷-۲۵ مؤلفه‌ای را ریشه‌ی درخت گرفته‌ایم که رأس‌های a، b و h در آن قرار دارند). فرزندان R مؤلفه‌های دوهمبندی هستند که نقطه‌ی پیوند مشترکی با R دارند. نوه‌های R مؤلفه‌های دوهمبندی هستند که هنوز در درخت قرار نگرفته‌اند، اما با دست‌کم یکی از فرزندان R نقطه‌ی پیوند مشترکی دارند. با ادامه‌ی این روند درخت را می‌سازیم. به عبارت دیگر، روش ساخت درخت، نخست-پهنا است. نمی‌توان ساده‌انگارانه گفت که هرگاه دو مؤلفه‌ی دوهمبند دارای نقطه‌ی پیوند مشترکی باشند به یکدیگر متصل هستند، چراکه ممکن است این نقطه‌ی پیوند بین بیش

از دو مؤلفه مشترک باشد و ما نباید دور تشکیل دهیم. چندان دشوار نیست که ثابت کنیم با چنین روشی همواره یک درخت ساخته می‌شود (تمرین ۷-۱۷). این درخت، درخت دوهمبندی نام دارد. شکل ۷-۲۶ (الف) بیانگر مؤلفه‌های دوهمبند گراف شکل ۷-۲۵ و شکل ۷-۲۶ (ب) نشانگر درخت دوهمبندی متناظر با آن است. در شکل ۷-۲۶، افزودن یک یال (x) را نشان داده‌ایم؛ برای مثال این یال می‌تواند رأس‌های a و k در گراف اصلی را به یکدیگر متصل کند.

حال اگر به درخت دوهمبندی دقت کنیم، درمی‌یابیم یالی که دو رأس از دو مؤلفه‌ی جداگانه را به یکدیگر متصل می‌کند، باعث تشکیل دور در درخت می‌شود. همه‌ی گره‌های این دور - که هر یک متناظر با یک مؤلفه‌ی دوهمبند هستند - باید در یک مؤلفه ادغام شوند. بدین ترتیب به یک الگوریتم می‌رسیم. دانستن شیوه‌ی ساخت درخت را هم به فرض استقرا می‌افزاییم؛ به این ترتیب برای هر یک از سه حالت پیش‌گفته راه‌کاری یافته‌ایم. چون راه‌حل بهتری هم وجود دارد، وارد جزئیات این الگوریتم نمی‌شویم.

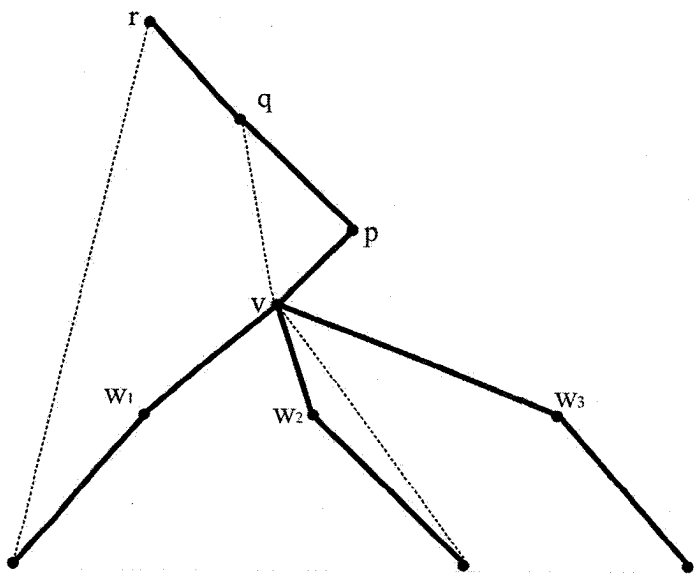
ضعف این الگوریتم در مقدار زمان لازم برای یافتن دورهای حاصل از یال افزوده‌شده، در درخت دوهمبندی است. برای یافتن دور ایجادشده در درخت ممکن است پیمایش کل درخت لازم باشد که در بدترین حالت، بررسی همه‌ی یال‌های درخت لازم خواهد شد. شاید تعداد یال‌های درخت از $O(|V|)$ باشند و ما ناچاریم این بررسی را به ازای هر یک از یال‌های گراف اصلی انجام دهیم. پس ممکن است زمان اجرای این الگوریتم از $O(|V||E|)$ شود (این تحلیل چندان دقیق نیست) اما بهتر است کاری کنیم که دیگر در هر گام، جست‌وجوی دور لازم نباشد.

یک روش رایج برای بهبود الگوریتم برآمده از فرض سرراست استقرا، گزینش سنجیده‌ی ترتیب استقراست. پیش‌تر یالی را به دل‌خواه برمی‌گزیدیم، اما اگر یال‌ها را به ترتیبی برگزینیم که کار با درخت دوهمبندی آسان‌تر شود، می‌توانیم الگوریتم را بهبود دهیم. نخستین تلاش طبیعی، بهره‌گیری از یک روش پیمایش مناسب در گراف است. بعداً روشن خواهیم ساخت که DFS برای این کار عالی است. دوباره شکل ۷-۲۵ را ببینید. فرض کنید که DFS از رأس a آغاز شود و نقطه‌ی پیوند b را در نظر بگیرید. B را نخستین مؤلفه‌ای بگیرید که DFS پس از دیدن b با آن روبه‌رو می‌شود. (در شکل ۷-۲۵، این مؤلفه از یال‌هایی تشکیل می‌شود که رأس‌های b، c، d، e، f و g را به یکدیگر متصل می‌کنند.) الگوریتم چگونه دریابد که b یک نقطه‌ی پیوند است؟ بنا به تعریف، اگر همه‌ی مسیرها از B به بقیه‌ی گراف، از b بگذرند، آنگاه b یک نقطه پیوند است. پس باید مشخص کنیم که آیا یالی از B به بقیه‌ی گراف رفته است یا نه.

فرض کنید با DFS رأس‌های درون B را بپیماییم. اگر یالی از B خارج نشود، پیمایش، درون B خواهد بود. همه‌ی یال‌های B را می‌پیماییم و دوباره به b خواهیم رسید. از آنجا که DFS یال‌های جانبی را کنار می‌گذارد، B، تنها با یال‌های عقب‌رو می‌تواند به بقیه‌ی گراف متصل شود. به عبارت دیگر، برداشتن b، B را ناهمبند می‌کند، اگر و تنها اگر هیچ یال عقب‌رویی از B خارج نشده باشد که به

درخت بالای b برسد. (تنها استثنای این قاعده در ریشه‌ی درخت DFS رخ می‌دهد.) اینک ببینیم چگونه می‌توان دریافت که چنین یالی وجود دارد یا نه.

می‌خواهیم ببینیم از یک زیردرخت تا کجای یک درخت DFS بالا خواهیم رفت. گراف را به یاری DFS می‌پیماییم. در هر رأس v، نخست، یک زیردرخت از پایین‌دست‌های v را به طور کامل می‌پیماییم، سپس به سراغ زیردرخت بعدی می‌رویم و کار را به همین ترتیب ادامه می‌دهیم. T_1 را زیردرختی بگیرید که ریشه‌اش یکی از فرزندان v است و DFS هم ریشه‌ی T_1 را پیش از دیگر فرزندان v بررسی می‌کند. فرض کنید هم مؤلفه‌های دوهمبند T_1 و هم بالاترین رأسی از درخت را که از راه یالی عقب‌رو به T_1 متصل است، می‌یابیم. (چنان که خواهید دید، با این روش، فرض استقرا واقعاً تقویت می‌شود.) بالاترین رأس درخت DFS را که از راه یالی عقب‌رو به v یا یکی از پایین‌دست‌های v (در درخت DFS) متصل است با High(v) نشان می‌دهیم. فرض کنید فرزندان v در درخت DFS، w_1، w_2، ... و w_k باشند (شکل ۷-۲۷ را ببینید). اگر برای همه‌ی w_iها، High(w_i) را داشته باشیم، می‌توانیم به آسانی High(v) را محاسبه کنیم: بین همه High(w_i)ها و سر دیگر همه‌ی یال‌های عقب‌رو از v، بالاترین رأس را به دست می‌آوریم. (اندکی بعد، روشی کارآمد برای تشخیص بالاتر بودن یک رأس از رأس دیگر خواهیم گفت.) بنابراین اگر DFS را اجرا کنیم، می‌توانیم به آسانی همه‌ی مقادیر High را به دست آوریم. در شکل ۷-۲۷، High(w_1)=r، High(w_2)=v، High(w_3)=w_3 و بالاترین یال عقب‌رو از v به q می‌رسد؛ پس High(v)=r.



شکل ۷-۲۷ محاسبه‌ی مقدار High

اینک فرض کنید همه‌ی مقدارهای تابع High محاسبه شده است. ادعا می‌کنیم رأس v یک نقطه‌ی پیوند است، اگر و تنها اگر v، فرزندی (مانند w_i) داشته باشد که High(w_i) از v بالاتر نباشد.

بی‌گمان اگر بتوان w_i را چنان یافت که این شرط برای آن درست باشد، آنگاه از رأس‌های زیردرختی که ریشه‌اش w_i است، در درخت هیچ یالی به رأس‌های بالاتر از v وجود نخواهد داشت؛ از این رو v یک نقطه‌ی پیوند است. (زیبایی DFS در این است که گراف را دقیقاً بنا به ترتیبی که برای کار ما مناسب است، می‌پیماید.)

با فرض بعدی استقرا، محاسبه‌ی مقادیر High هم‌گام با DFS پیش می‌رود.

فرض استقرا: هنگامی که با DFS، kامین رأس را ببینیم، می‌دانیم چگونه برای رأس‌های پایین‌دست و از پیش دیده‌شده‌ی این رأس، مقدار High را بیابیم.

ترتیب استقرا از ترتیب DFS پیروی می‌کند. هنگامی که به رأس v می‌رسیم، برای همه‌ی فرزندان v (به صورت بازگشتی) یک DFS اجرا می‌کنیم و برای هر یک (بنا به استقرا) مقدار High را می‌یابیم؛ سپس High(v) را محاسبه می‌کنیم. هم‌زمان با این کار می‌توانیم بررسی کنیم که آیا یک رأس، نقطه‌ی پیوند هست یا نه.

ریشه‌ی درخت DFS حالت خاصی دارد. روشن است که هیچ یک از مقادیر High نمی‌تواند از ریشه بالاتر باشد. به سادگی می‌توان دید خودِ ریشه نیز یک نقطه‌ی پیوند است، اگر و تنها اگر در درخت DFS ریشه بیش از یک فرزند داشته باشد که بررسی این موضوع هم کار آسانی است.

کارآمدی این الگوریتم در محاسبه‌ی مقادیر High به این دلیل است که پس از اجرای DFS، همه‌ی اطلاعات مورد نیاز در دست‌رس هستند. تنها مشکل باقی‌مانده، تشخیص بالاتر بودن یک رأس از رأس دیگر در درخت DFS است. از شماره‌های DFS برای انجام این کار یاری می‌گیریم. همه‌ی رأس‌هایی که در محاسبه‌ی مقدار High برای یک رأس نقش دارند، بالادست آن رأس در درخت DFS هستند. بنابراین، پیش از رسیدن به آن رأس، شماره‌ی DFS گرفته‌اند. به علاوه، هر چه یک رأس بالادست، بالاتر باشد، شماره‌ی DFS کوچک‌تری دارد! این مطلب برای رأس‌هایی که به این درخت مربوط نباشند درست نیست، اما خوش‌بختانه تنها یال‌های عقب‌رو مورد توجه هستند. پس روشی عملی برای کار با مقادیر High، سود جستن از شماره‌های DFS است. تعریف High(v) را تغییر می‌دهیم، تا به جای بالاترین رأس، شماره‌ی DFS برای آن رأس را برگرداند. توصیف الگوریتم با شماره‌های DFS گیج‌کننده است، زیرا شماره‌ی DFS در رأس‌های بالاتر کوچک‌تر است. بنابراین، شماره‌های کاهشی DFS را تعریف می‌کنیم: شماره‌ی کاهشی DFS برای ریشه، $|V|$ است و هر بار که به رأسی تازه برمی‌خوریم، این شماره را یک واحد می‌کاهیم؛ حتا می‌توانیم شماره‌های منفی را به کار گیریم: شماره‌ی DFS را برای ریشه ۱- می‌گیریم و هر بار که به رأسی تازه برخوردیم، آن را یک واحد می‌کاهیم. برتری این روش نسبت به روش نخست، این است که دیگر نیازی نیست مقدار $|V|$ را از پیش بدانیم.

تنها کار باقی‌مانده، یافتن خود مؤلفه‌های دوهمبند است. می‌توان با روشی کورکورانه این کار را انجام داد، اما شیوه‌ی درخشانی نیز وجود دارد. دوباره به شکل ۷-۲۵ نگاه کنید. توجه کنید هنگامی که

الگوریتم تشخیص داد b یک نقطه‌ی پیوند است، آخرین یال‌های پیموده‌شده، یال‌های B بودند. هم، رأس‌های تازه و هم، یال‌ها را (پس از دیدن) درون یک پشته می‌گذاریم. هنگامی که روشن شد یک رأس، نقطه‌ی پیوند است، می‌توانیم همه‌ی یال‌های روی پشته را (که دقیقاً تشکیل‌دهنده‌ی یک مؤلفه‌ی دوهمبند هستند) حذف کرده، تا رسیدن به همان نقطه‌ی پیوند به عقب برگردیم. حال می‌توانیم آن یال‌ها را از گراف کنار بگذاریم و سپس کار را با همان روش پیش ادامه دهیم. برنامه‌ی کامل یافتن مؤلفه‌های دوهمبند در شکل ۷-۲۸ ارائه شده است. (این الگوریتم را می‌توانستیم تنها براساس اعمالی پیش‌ترتیب و پس‌ترتیب در دل الگوریتم DFS تعریف کنیم، اما آن را به صورت کامل ارائه کرده‌ایم.)

پیچیدگی: روشن است در هر رأس، هم مقدار کار لازم برای اجرای DFS و هم مقدار کار اضافی ثابت است. پس زمان اجرای این الگوریتم از $O(|V|+|E|)$ است. از آنجا که هنگام پیمایش مؤلفه‌ها باید آن‌ها را نگه‌داری کنیم، حافظه‌ی مورد نیاز الگوریتم هم از $O(|V|+|E|)$ خواهد بود.

☐ مثال ۷-۳

شکل ۷-۲۹ نمونه‌ای از اجرای الگوریتم Biconnected_Componenets برای گراف شکل ۷-۲۵ است. در بالای جدول، رأس‌های گراف و سپس شماره‌ی (کاهشی) DFS برای آن‌ها را نشان داده‌ایم. هر یک از سطرهای بعدی، به‌روزرسانی مقادیر High را پس از هر فراخوانی تازه‌ی روال بازگشتی نشان می‌دهد. در این جدول، هنگام یافتن نقاط پیوند، دور آن‌ها دایره کشیده شده است.

☐

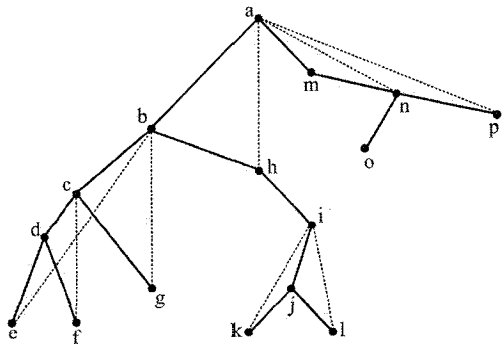
الگوریتم: Biconnected_Components(G,v,n)

ورودی: G=(V,E) (یک گراف همبند بدون‌جهت)، v (رأسی که نقش ریشه‌ی درخت DFS را برعهده دارد) و n (تعداد رأس‌های G)

خروجی: مؤلفه‌های دوهمبند، علامت‌گذاری و مقادیر High محاسبه شده است.

```
begin
    for هر رأس v از گراف do
        v.DFS_Number := 0;
    {شماره‌های DFS نشان می‌دهند که رأس‌های متناظر بررسی شده‌اند یا نه.}
    DFS_N := n;
    {شماره‌های کاهشی DFS را به کار برده‌ایم؛ توضیح متن کتاب را ببینید.}
    BC(v)
end

procedure BC(v);
begin
    v.DFS_Number := DFS_N;
    DFS_N := DFS_N - 1;
    {این پشته در آغاز تهی است.}  v را به پشته اضافه کن;
    v.High := v.DFS_Number;  {مقدار اولیه}
    for  هر یال (v,w) do
        (v,w) را به پشته بیفزا;
        {هر یال دوبار (و هر بار برای یکی از جهت‌هایش) افزوده خواهد شد.}
        if  w والد v نیست  then
          if w.DFS_Number = 0 then
            BC(w);
            if w.High ≤ v.DFS_Number then
            {v، اتصال w را از بقیه‌ی گراف قطع می‌کند.}
              همه‌ی یال‌ها و رأس‌ها را از پشته حذف کن تا آن که به v;
              برسی، سپس زیرگراف حاصل از این یال‌ها و رأس‌ها را به عنوان
                                        مؤلفه‌ای دوهمبند علامت بزن
              v را دوباره در پشته قرار بده; {v بخشی از مؤلفه‌ی w است و شاید بخشی
                                        از مؤلفه‌ی رأس‌های دیگر هم باشد.}
            v.High := max(v.High,w.High)
          else {(v,w) یا یک یال عقب‌روست و یا یک یال جلورو.}
            v.High := max(v.High,w.DFS_Number)
end
```

شکل ۷-۲۸ الگوریتم Biconnected_Components



	a ۱۶	b ۱۵	c ۱۴	d ۱۳	e ۱۲	f ۱۱	g ۱۰	h ۹	i ۸	j ۷	k ۶	l ۵	m ۴	n ۳	o ۲	p ۱
a	۱۶	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
b	۱۶	۱۵	–	–	–	–	–	–	–	–	–	–	–	–	–	–
c	۱۶	۱۵	۱۴	–	–	–	–	–	–	–	–	–	–	–	–	–
d	۱۶	۱۵	۱۴	۱۳	–	–	–	–	–	–	–	–	–	–	–	–
e	۱۶	۱۵	۱۴	۱۳	۱۵	–	–	–	–	–	–	–	–	–	–	–
d	۱۶	۱۵	۱۴	۱۵	۱۵	–	–	–	–	–	–	–	–	–	–	–
f	۱۶	۱۵	۱۴	۱۵	۱۵	۱۴	–	–	–	–	–	–	–	–	–	–
d	۱۶	۱۵	۱۴	۱۵	۱۵	۱۴	–	–	–	–	–	–	–	–	–	–
c	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	–	–	–	–	–	–	–	–	–	–
g	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	–	–	–	–	–	–	–	–	–
c	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	–	–	–	–	–	–	–	–	–
(b)	۱۶	۱۵	۱۵	۱۵	۱۴	۱۳	۱۵	–	–	–	–	–	–	–	–	–
h	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	–	–	–	–	–	–	–	–
i	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	–	–	–	–	–	–	–
j	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۷	–	–	–	–	–	–
k	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۷	۸	–	–	–	–	–
j	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	–	–	–	–	–
i	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	–	–	–	–
j	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	–	–	–	–
(j)	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	–	–	–	–
(h)	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	–	–	–	–
b	۱۶	۱۶	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	–	–	–	–
a	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	–	–	–	–
m	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	۴	–	–	–
n	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	۴	۱۶	–	–
o	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	۴	۱۶	۲	–
(n)	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	۴	۱۶	۲	–
p	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	۴	۱۶	۲	۱۶
n	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	۴	۱۶	۲	۱۶
m	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	۱۶	۱۶	۲	۱۶
(a)	۱۶	۱۵	۱۵	۱۵	۱۵	۱۴	۱۵	۱۶	۸	۸	۸	۸	۱۶	۱۶	۲	۱۶

شکل ۷-۲۹ مثالی از محاسبه‌ی مقادیر High و مؤلفه‌های دوهمبند

۷-۹-۲ مؤلفه‌های قویاً همبند

در این بخش تنها گراف‌های جهت‌دار را بررسی می‌کنیم. یک گراف جهت‌دار را قویاً همبند گوییم، اگر برای هر زوج‌رأس v و w، هم مسیری از v به w و هم مسیری از w به v وجود داشته باشد. به عبارت دیگر، در این گراف هر رأس از هر رأس دیگر دست‌رس‌پذیر است.

تعریف: یک مؤلفه‌ی قویاً همبند، زیرمجموعه‌ای گسترش‌ناپذیر از رأس‌هاست که زیرگراف القایی آن قویاً همبند باشد (یعنی هیچ زیرمجموعه‌ای از رأس‌ها وجود نداشته باشد که هم این مؤلفه را در بر گیرد و هم زیرگراف القایی آن قویاً همبند باشد).

توجه کنید برخلاف مؤلفه‌ی دوهمبند، مؤلفه‌ی قویاً همبند، با مجموعه‌ای از رأس‌ها تعریف می‌شود. افراز یک گراف به مؤلفه‌های قویاً همبند یکتاست. هر رأس دقیقاً به یک مؤلفه تعلق دارد، اما هر یال گراف یا به یک مؤلفه تعلق دارد یا دو مؤلفه‌ی جداگانه را به یکدیگر متصل می‌کند. به یاری دو لم بعد، وجود چنین افرازی را ثابت می‌کنیم. این دو لم، شبیه لم‌هایی هستند که در بخش پیش برای مؤلفه‌های دوهمبند به کار بردیم.

□ لم ۷-۱۱

دو رأس، متعلق به مؤلفه‌ی قویاً همبند یکسانی هستند، اگر و تنها اگر مداری دربرگیرنده‌ی هر دوی آن‌ها در گراف وجود داشته باشد. (به خاطر بیاورید که یک مدار، مسیری جهت‌دار و بسته است که این مسیر لزوماً ساده نیست؛ یعنی ممکن است از برخی رأس‌ها بیش از یک بار بگذرد. دور نیز مداری ساده است.)

برهان: یک مدار به تنهایی قویاً همبند است. امکان ندارد یک مؤلفه‌ی قویاً همبند تنها برخی از رأس‌های یک مدار را در بر گیرد، چراکه دیگر بزرگ‌ترین زیرمجموعه‌ی ممکن نخواهد بود (چون می‌توانیم دیگر رأس‌های مدار را نیز به این مؤلفه بیفزاییم). اگر دو رأس v و w از یک مؤلفه‌ی قویاً همبند یکسان داده شده باشند، ادعا می‌کنیم این دو رأس در یک مدار قرار دارند. بنا به تعریف قویاً همبند بودن، هم مسیری از v به w و هم مسیری از w به v وجود خواهد داشت. از کنار هم گذاشتن این دو مسیر، یک مدار به دست می‌آید (اما این مدار لزوماً یک دور نیست، چراکه ممکن است رأس‌های این دو مسیر جداازهم نباشند).

□

□ لم ۷-۱۲

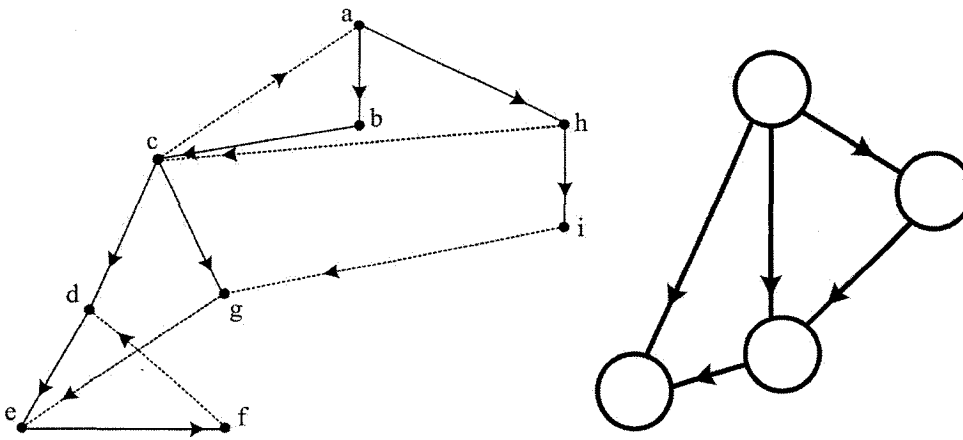
هر رأس دقیقاً متعلق به یک مؤلفه‌ی قویاً همبند است.

برهان: اگر رأس v به بیش از یک مؤلفه‌ی قویاً همبند تعلق داشته باشد، آنگاه مدارهایی وجود خواهند داشت که هم v و هم رأس‌هایی از مؤلفه‌های دیگر را در بر می‌گیرند؛ اما از ترکیب این مدارها،

مدار دیگری تشکیل می‌شود که بنا به لم ۷-۱۱ باید کاملاً در یک مؤلفه‌ی قویاً همبند قرار داشته باشد. پس به تناقض رسیدیم.

□

در اینجا می‌توانیم گراف مؤلفه‌ی قویاً همبند (SCC) را همانند درخت دوهمبندی مؤلفه‌ها تعریف کنیم. (به این گراف، گراف فشرده‌شده هم می‌گویند.) گره‌های گراف SCC متناظر با مؤلفه‌های قویاً همبند هستند. (برای رأس‌های این گراف، واژه‌ی «گره» را به کار برده‌ایم تا با رأس‌های گراف اصلی اشتباه نشوند.) یالی جهت‌دار از گره a به گره b وجود دارد، اگر (در گراف اصلی) از مؤلفه‌ی متناظر با a، یالی جهت‌دار به مؤلفه‌ی متناظر با b وجود داشته باشد. گراف SCC بدون‌دور است، زیرا دورها نمی‌توانند بیش از یک مؤلفه را در بر گیرند. شکل ۷-۳۰، گراف جهت‌دار G و گراف فشرده‌شده‌ی آن را نشان می‌دهد.



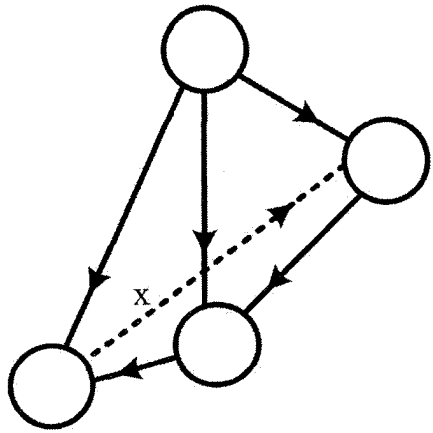
شکل ۷-۳۰ یک گراف جهت‌دار و گراف مؤلفه‌های قویاً همبند آن

در اینجا هم می‌توانیم مانند مؤلفه‌های دوهمبند، به یاری استقرا یک الگوریتم طراحی کنیم.

فرض استقرا: می‌دانیم چگونه در گراف‌هایی با کم‌تر از m یال، مؤلفه‌های قویاً همبند را بیابیم و برای آن‌ها گراف SCC را بسازیم.

حالت پایه روشن و بدیهی است. گرافی با کم‌تر از m یال در نظر بگیرید و در آن یال دل‌خواهی مانند x را برگزینید. x را حذف می‌کنیم و بنا به استقرا در گراف باقی‌مانده، مؤلفه‌های قویاً همبند را می‌یابیم. سپس باید اثر افزودن x روی مؤلفه‌های یافته‌شده را روشن کنیم. بازهم حالت آسان هنگامی است که x، دو رأس از یک مؤلفه‌ی یکسان را به یکدیگر متصل کند. در این حالت، افزودن x نه تأثیری بر افراز دارد و نه تأثیری بر گراف SCC. حالت دشوار هنگامی است که x، رأس‌هایی از دو مؤلفه‌ی جداگانه را به یکدیگر متصل کند. این حالت در شکل ۷-۳۱ نشان داده شده است که در آن، یال x دو مؤلفه از گراف فشرده‌شده‌ی شکل ۷-۳۰ را به یکدیگر متصل می‌کند. روشن است که x این دو مؤلفه را با یکدیگر ادغام خواهد کرد، اگر و تنها اگر این یال در گراف SCC، یک دور (جهت‌دار) را تکمیل کند. در

این حالت، همه‌ی مؤلفه‌های متناظر با گره‌های این دور باید در یک مؤلفه ادغام شوند و با این عمل، کار به پایان می‌رسد. اگر x، درون هیچ دوری از گراف SCC نباشد، روشن است که مؤلفه‌ی دربردارنده‌ی x نیز تغییر نخواهد کرد. مانند مؤلفه‌های دوهمبند، اینجا نیز می‌توانیم الگوریتم را با در نظر گرفتن ترتیب مشخصی برای یال‌ها بهبود دهیم. دوباره، DFS نقش اصلی را بر عهده می‌گیرد.



شکل ۷-۳۱ افزودن یک یال که دو مؤلفه‌ی قویاً همبند جداگانه را به یکدیگر متصل می‌کند.

بیایید اینجا نیز از همان گام‌هایی که برای الگوریتم مؤلفه‌های دوهمبند برداشتیم، پیروی کنیم و تنها در صورت لزوم، این گام‌ها را تغییر دهیم. هنگامی که با DFS به رأسی برمی‌خوریم، می‌خواهیم بدانیم آیا آن رأس با دیگر رأس‌ها، در یک مدار قرار می‌گیرد یا نه؛ به ویژه با رأس‌هایی که در درخت DFS، بالاتر از آن رأس قرار دارند. بازهم می‌توان ایده‌ی مقادیر High را به کار گرفت. در جست‌وجوی رأس‌هایی هستیم که از خودشان یا پایین‌دست‌های آن‌ها، راهی به دیگر بخش‌های گراف وجود نداشته باشد. همان‌گونه که برای مؤلفه‌های دوهمبند، نقاط پیوند را یافتیم، حالا هم نیازمند شیوه‌ای هستیم که با آن بتوانیم «نقاط گسست» را بیابیم. (مترجمان بر این باورند که در ادامه‌ی این پاراگراف، به جای درخت DFS باید جنگل DFS گفته می‌شد، اما برای رعایت امانت‌داری، عین مطلب کتاب را ترجمه کرده‌اند.) درخت DFS را در نظر بگیرید. هر مؤلفه‌ی قویاً همبند، متناظر با یکی از بخش‌های همبند این درخت است (تمرین ۷-۸۸) یعنی همه‌ی رأس‌های یک مؤلفه‌ی قویاً همبند باید به یک زیردرخت همبند از درخت DFS متعلق باشند. برای یک مؤلفه‌ی داده‌شده، بالاترین رأس در درخت را در نظر بگیرید؛ این رأس را ریشه‌ی آن مؤلفه می‌گوییم. ریشه‌ی یک مؤلفه، نخستین رأسی است که DFS به آن برخورد می‌کند. (برای مثال، ریشه‌های مؤلفه‌ها در شکل ۷-۳۰، a، d، g و i هستند.) اگر بتوانیم با روشی شبیه یافتن نقاط پیوند این ریشه‌ها را نیز پیدا کنیم، آنگاه می‌توانیم تمام اجزای افراز را بیابیم. خواهیم دید که این ریشه‌ها مانند نقاط پیوند هستند.

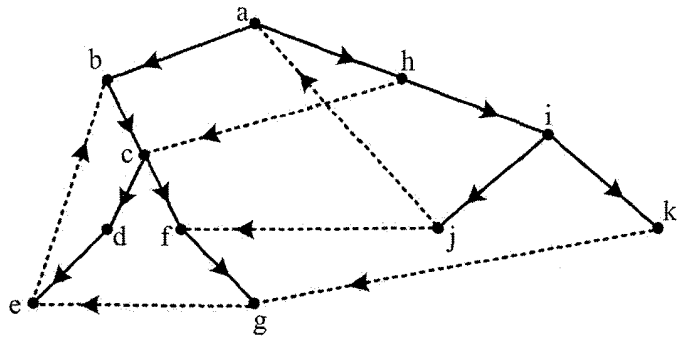
الگوریتم را بر پایه‌ی «استقرایی با ترتیب DFS» می‌سازیم. r را ریشه‌ی نخستین مؤلفه‌ای بگیرید که DFS، آن را به طور کامل می‌بیند. این مؤلفه در شکل رایج DFS، مؤلفه‌ی گوشه‌ی سمت

چپ پایین است (در شکل ۷-۳۰: r=d). این مؤلفه باید در درخت، همه‌ی پایین‌دست‌های r را در بر گیرد (هیچ رأسی از پایین‌دست‌های r نمی‌تواند به مؤلفه‌ای کوچک‌تر - توضیح مترجمان: منظور نویسنده از مؤلفه‌ی کوچک‌تر، مؤلفه‌ای است که پیش‌تر دیده شده باشد، نه مؤلفه‌ای که اندازه‌ی آن کوچک‌تر است - متعلق باشد، چراکه مؤلفه‌های کوچک‌تر زودتر پیمایش می‌شوند). اگر هنگام اجرای DFS، بتوانیم تشخیص دهیم که r نخستین ریشه است، آنگاه می‌توانیم مؤلفه‌ی دربرگیرنده‌ی r را بیابیم و از گراف کنار بگذاریم تا بتوانیم فرایند را با استقرا ادامه دهیم. البته کار به این سادگی هم نیست؛ اما ایده‌ی اصلی همین است. نخست، ببینیم چگونه می‌توانیم r را شناسایی کنیم.

اگر r، ریشه‌ی یک مؤلفه باشد، آنگاه هیچ یک از رأس‌های پایین‌دست آن، یالی عقب‌رو به رأس‌های بالادست r نخواهند داشت؛ چراکه چنین یال عقب‌رویی همراه با سر یا رأس انتهای آن، یک دور تشکیل خواهد داد که در این صورت، رأس بالاتر و r متعلق به مؤلفه‌ی یکسانی خواهند بود. برای بررسی وجود چنین یال‌های عقب‌رویی در اینجا نیز می‌توانیم مانند مؤلفه‌های دوهمبند، مقادیر High را به کار ببریم؛ اما باید بیش‌تر دقت کنیم، چراکه DFS در گراف‌های جهت‌دار، یال‌های جانبی را حذف نمی‌کند. چنان که در شکل ۷-۳۲ می‌بینید، با آن که رأس g یال عقب‌رو ندارد، اما از آن، یک یال جانبی به e رفته است؛ به این ترتیب، هرچند یال عقب‌رویی وجود ندارد که از یک رأس پایین‌دست g آغاز شود، اما والد g (یعنی f) ریشه‌ی هیچ مؤلفه‌ای نیست. پس باید یال‌های جانبی را هم بررسی کنیم.

تأثیر یال‌های جانبی چیست؟ جهت این یال‌ها حتماً از راست به چپ است؛ به عبارت دیگر، به رأس‌هایی می‌روند که پیش‌تر دیده شده‌اند. به یاد داشته باشید که در حال جست‌وجوی نخستین ریشه هستیم. اگر یالی جانبی از g به e وجود داشته باشد و هنوز ریشه را نیافته باشیم، ادعا می‌کنیم که f ریشه نیست، بلکه ریشه باید بالادست هر دو رأس f و e باشد، چراکه اگر ریشه بالادست f نبود، باید پیش از f به آن می‌رسیدیم. در ضمن، این واقعیت هم که هنوز مؤلفه‌ی دربرگیرنده‌ی e شناسایی نشده است، نشان می‌دهد راهی برای بالا رفتن از e وجود دارد. پس اگر یالی جانبی از g به رأسی که پیش از f بررسی شده است، وجود داشته باشد، f ریشه نخواهد بود. آیا با چنین حالتی روبه‌رو شده‌ایم؟ تشخیص این موضوع به سادگی تشخیص عقب‌رو بودن یال است. تنها کافی است به شماره‌های DFS توجه کنیم! هنگام بررسی اثر «یال g به e» عقب‌رو بودن یا نبودن آن اهمیتی ندارد؛ تنها باید به شماره‌ی DFS رأس e (و رابطه‌ی این مقدار با شماره‌ی DFS رأس f) توجه کنیم. مقدار High را نیز مانند آنچه در مورد مؤلفه‌های دوهمبند گفته شد، می‌توانیم با یافتن یالی که به رأس با کم‌ترین شماره‌ی DFS می‌رود، تعریف کنیم. مقدار High برای یک رأس، بالاترین مقدار High بین فرزندان آن رأس، رأس پایانی یال‌های عقب‌رو و رأس پایانی یال‌های جانبی آن تعریف می‌شود. نخستین ریشه، اولین رأسی است که مقدار High در آن از خود رأس بالاتر نباشد. دقت کنید که مقادیر High واقعاً بالاترین رأس‌ها را نشان نمی‌دهند. مقدار High برای g شماره‌ی DFS رأس e است، هرچند رسیدن به b از e (و در نتیجه از g) امکان‌پذیر است. تنها، امکان رسیدن به رأسی بالاتر از g (یا f) را بررسی می‌کنیم؛

شناسایی بالاترین رأس برای ما اهمیتی ندارد. (همان‌گونه که هنگام رسیدن به یالی عقب‌رو نیز لازم نیست یال‌های خارج‌شونده از رأس پایانی آن را بررسی کنیم.)



شکل ۷-۳۲ تأثیر یال‌های جانبی

با یافتن نخستین ریشه، می‌توانیم نخستین مؤلفه‌ی قویاً همبند را بیابیم: همه‌ی پایین‌دست‌های ریشه در درخت DFS. سپس می‌توانیم این مؤلفه را از گراف کنار بگذاریم. برای این کار همه‌ی رأس‌ها و یال‌های مؤلفه را به همراه همه‌ی یال‌هایی که از دیگر رأس‌ها به این مؤلفه آمده‌اند، حذف می‌کنیم؛ البته می‌توان از حذف یال‌هایی که از دیگر رأس‌ها آمده‌اند، چشم‌پوشی کرد، چراکه دیگر راهی برای خروج از این مؤلفه وجود ندارد. از آنجا که گراف کوچک‌تر شده است، حالا می‌توان باقی‌مانده‌ی کار را با استقرا پی‌گیری کرد! (بررسی درستی تمام فرض‌ها بر عهده‌ی خواننده است.) توجه کنید که تعریف مقدار High تعریفی پویاست. با حذف یال‌هایی که به مؤلفه‌ی تازه کشف‌شده می‌روند، دیگر این یال‌ها نقشی در محاسبه‌ی مقادیر High نخواهند داشت. (این تعریف، با تعریف ایستای مقادیر High برای مؤلفه‌های دوهمبند تفاوت دارد؛ در آنجا این مقدارها به مؤلفه‌های پیشین وابسته نبودند.) در عمل، لازم نیست واقعاً رأس‌ها یا یال‌ها را حذف کنیم. می‌توانیم به سادگی رأس‌های هر مؤلفه‌ی شناسایی شده را علامت بزنیم تا در آینده یال‌هایی را که به این رأس‌های علامت‌خورده می‌روند، نادیده بگیریم. الگوریتم مؤلفه‌ی قویاً همبند در شکل ۷-۳۳ ارائه شده است (بازهم برای پیش‌گیری از ابهام شمارنده‌های کاهشی DFS را به کار برده‌ایم).

پیچیدگی: از آنجا که این الگوریتم مانند الگوریتم مؤلفه‌ی دوهمبند است؛ پیچیدگی زمانی و فضایی آن از $O(|V|+|E|)$ خواهد بود.

الگوریتم: Strongly_Connected_Components(G,v,n)

ورودی: G=(V,E) (یک گراف جهت‌دار)، v (رأسی که نقش ریشه‌ی درخت DFS را بر عهده دارد) و n (تعداد رأس‌های G)

خروجی: همان گراف ورودی که در آن مؤلفه‌های قویاً همبند، علامت‌گذاری و مقادیر High محاسبه شده‌اند.

{مانند همیشه، «روال DFS برای گراف‌های جهت‌دار» آن قدر فراخوانی می‌شود تا همه‌ی رأس‌ها دیده شوند.}

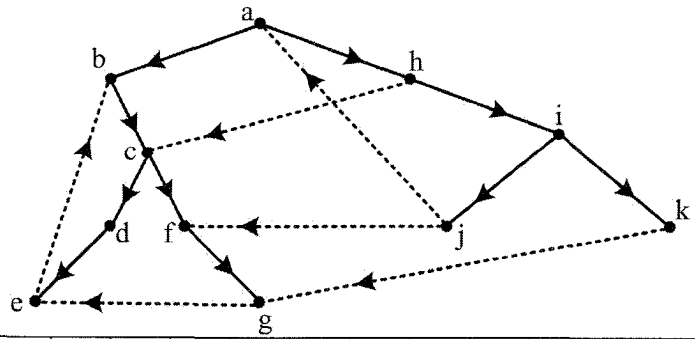
```
begin
    for G در v هر رأس do
        v.DFS_Number := 0;
        v.Componenet := 0;
    Current_Component := 0;
    DFS_N := n;
    {از شماره‌های کاهشی DFS سود می‌جوییم؛ بخش ۷-۹-۱ را ببینید.}
    while v.DFS_Number 0 رأسی مانند v وجود دارد که do
        SCC(v)
end

procedure SCC(v);
begin
    v.DFS_Number := DFS_N;
    DFS_N := DFS_N - 1;
    v را به پشته بیفزا;
    v.High := v.DFS_Number; {مقدار اولیه}
    for (v,w) هر یال do
        if w.DFS_Number = 0 then
            SCC(w);
            v.High := max(v.High,w.High)
        else
            if w.DFS_Number > v.DFS_Number and w.Component = 0 then
            {(v,w) یا یک یال جانبی و یا یک یال عقب‌رو است که باید آن را نیز در
                v.High := max(v.High,w.DFS_Number);
            نظر بگیریم.}
    if v.High = v.DFS_Number then {ریشه‌ی یک مؤلفه است.}
        Current_Component := Current_Component + 1;
        repeat {علامت‌گذاری رأس‌های مؤلفه‌ی تازه}
            مقدار سر پشته را حذف کن و در x بگذار ;
            x.Component := Current_Component;
        until x = v
end
```

شکل ۷-۳۳ الگوریتم Strongly_Connected_Components

☐ مثال ۷-۴

در شکل ۷-۳۴ نمونه‌ای از اجرای الگوریتم Strongly_Connected_Components برای گراف شکل ۷-۳۲ نشان داده شده است. سطر نخست جدول، رأس‌های گراف و سطر دوم، شماره‌های (کاهشی) DFS را برای هر یک از آن‌ها نشان می‌دهد. هر یک از سطرهای بعدی هم، نشان‌دهنده‌ی شماره‌های به‌روزشده‌ی High، پس از یک فراخوانی تازه‌ی روال بازگشتی است. پس از آن هم که دریافتیم یک رأس، ریشه‌ی مؤلفه‌ای قویاً همبند است، دور آن دایره کشیده‌ایم.

☐



	a ۱۱	b ۱۰	c ۹	d ۸	e ۷	f ۶	g ۵	h ۴	i ۳	j ۲	k ۱
a	۱۱	–	–	–	–	–	–	–	–	–	–
b	۱۱	۱۰	–	–	–	–	–	–	–	–	–
c	۱۱	۱۰	۹	–	–	–	–	–	–	–	–
d	۱۱	۱۰	۹	۸	–	–	–	–	–	–	–
e	۱۱	۱۰	۹	۸	۱۰	–	–	–	–	–	–
d	۱۱	۱۰	۹	۱۰	۱۰	–	–	–	–	–	–
c	۱۱	۱۰	۱۰	۱۰	۱۰	–	–	–	–	–	–
f	۱۱	۱۰	۱۰	۱۰	۱۰	۶	–	–	–	–	–
g	۱۱	۱۰	۱۰	۱۰	۱۰	۶	۷	–	–	–	–
f	۱۱	۱۰	۱۰	۱۰	۱۰	۷	۷	–	–	–	–
c	۱۱	۱۰	۱۰	۱۰	۱۰	۷	۷	–	–	–	–
(b)	۱۱	۱۰	۱۰	۱۰	۱۰	۷	۷	–	–	–	–
a	۱۱	۱۰	۱۰	۱۰	۱۰	۷	۷	–	–	–	–
h	۱۱	۱۰	۱۰	۱۰	۱۰	۷	۷	۴	–	–	–
i	۱۱	۱۰	۱۰	۱۰	۱۰	۷	۷	۴	۳	–	–
j	۱۱	۱۰	۱۰	۱۰	۱۰	۷	۷	۴	۳	۱۱	–
i	۱۱	۱۰	۱۰	۱۰	۱۰	۷	۷	۴	۱۱	۱۱	–
(k)	۱۱	۱۰	۱۰	۱۰	۱۰	۷	۷	۴	۱۱	۱۱	۱
i	۱۱	۱۰	۱۰	۱۰	۱۰	۷	۷	۴	۱۱	۱۱	۱
h	۱۱	۱۰	۱۰	۱۰	۱۰	۷	۷	۱۱	۱۱	۱۱	۱
(a)	۱۱	۱۰	۱۰	۱۰	۱۰	۷	۷	۱۱	۱۱	۱۱	۱

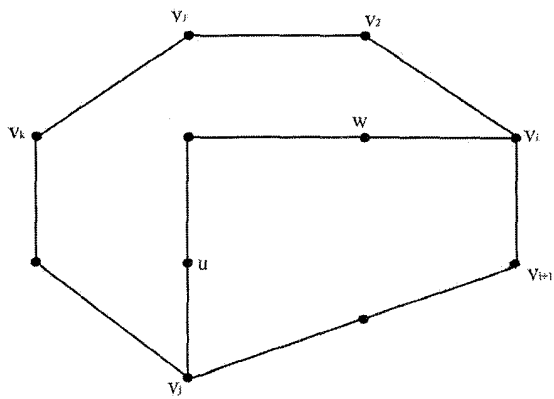
شکل ۷-۳۴ نمونه‌ای از محاسبه‌ی مقادیر High و مؤلفه‌های قویاً همبند

۷-۹-۳ نمونه‌هایی از کاربرد تجزیه‌ی گراف

در این بخش کوتاه، راه‌حل دو مسأله را بررسی می‌کنیم و می‌بینیم که با تجزیه‌ی گراف، کار بسیار ساده‌تر می‌شود. نخستین مسأله، درباره‌ی گراف‌های بدون‌جهت و دومی، درباره‌ی گراف‌های جهت‌دار است.

مسأله: گراف همبند و بدون‌جهت G=(V,E) داده شده است. مشخص کنید آیا این گراف، دوری با طول زوج دارد یا نه.

دیدیم که یک دور باید درون یک مؤلفه‌ی دوهمبند باشد. از این رو، نخست گراف را به مؤلفه‌های دوهمبند افراز می‌کنیم و هر یک از مؤلفه‌ها را جداگانه در نظر می‌گیریم. به این ترتیب، می‌توانیم فرض کنیم که گراف دوهمبند است! گراف دوهمبندی که بیش از یک یال داشته باشد، دست‌کم یک دور نیز دارد (در واقع، هر دو یال گراف در یک دور قرار دارند). بیایید یک دور دل‌خواه بیابیم (فرض کنید این دور $C_1 = v_1, v_2, \ldots, v_k, v_1$ باشد). اگر k زوج باشد که کار به پایان می‌رسد. اگر در گراف، یال دیگری وجود نداشته باشد - یعنی گراف دقیقاً از یک دور با طول فرد تشکیل شده باشد - آنگاه پاسخ مسأله قطعاً منفی است، اما اگر یال دیگری به جز یال‌های این دور وجود داشته باشد، پس یالی وجود دارد که در این دور نیست، اما یکی از رأس‌هایش متعلق به دور است. این یال را (vi,w) بگیرید. از آنجا که گراف دوهمبند است، پس یال‌های (vi,w) و (vi, vi+1) در یک دور (که آن را C_2 می‌نامیم) قرار خواهند گرفت. C_2 را با آغاز از w می‌پیماییم تا آن که دوباره، مثلاً در رأس v_j، به C_1 برسیم (شکل ۷-۳۵ را ببینید). روشن است که $v_i \neq v_j$. مسیر vi، w، ...، u و vj چنان که در شکل ۷-۳۵ نشان داده‌ایم، سازنده‌ی دو دور تازه است. به آسانی می‌توان دید که طول یکی از این سه دور باید زوج باشد. پس قضیه‌ی ۷-۱۳ را ثابت کردیم.



شکل ۷-۳۵ یافتن یک دور با طول زوج

قضیه ۷-۱۳

هر گراف دوهمبند با بیش از یک یال که خود، دوری با طول فرد نباشد، دربردارنده‌ی دوری با طول زوج خواهد بود.

□

مسأله‌ی دوم مانند مسأله‌ی نخست است، اما برای گراف‌های جهت‌دار.

مسأله: گراف جهت‌دار G=(V,E) داده شده است. مشخص کنید آیا این گراف، دوری (جهت‌دار) با طول فرد دارد یا نه.

اینجا نیز، می‌دانیم که یک دور باید در مؤلفه‌ای قویاً همبند قرار داشته باشد، پس می‌توانیم گراف را قویاً همبند فرض کنیم. DFS را از رأسی دل‌خواه مانند r آغاز می‌کنیم و به رأس‌ها به روشی که می‌گوییم، برچسب «زوج» یا «فرد» می‌زنیم: به r برچسب زوج زده، برای هر یال (v,w) به w برچسب مخالف v می‌زنیم. از آنجا که (بنا به فرض قویاً همبند بودن) r از هر رأس دیگر دست‌رس‌پذیر است، ادعا می‌کنیم یک دور به طول فرد وجود دارد، اگر و تنها اگر طی فرایند علامت‌گذاری بخواهیم به رأسِ از پیش برچسب‌خورده‌ای، علامتی مخالف برچسب آن بزنیم (چشم‌گیرترین حالت هنگامی است که دوباره به r رسیده باشیم و بخواهیم به آن برچسب فرد بزنیم). اثبات را که به فرض «قویاً همبند بودن» بسیار وابسته است، بر عهده‌ی خواننده می‌گذاریم.

حل هر یک از این دو مسأله، بدون تجزیه‌ی گراف بسیار دشوارتر می‌شود. از آنجا که می‌توان هر کدام از این دو نوع تجزیه را به گونه‌ای کارآمد در زمانی خطی انجام داد، پس خوب است هنگام روبه‌رو شدن با یک مسأله‌ی گراف، فرض اضافی «دوهمبند بودن» یا «قویاً همبند بودن» را به مسأله بیفزاییم. این فرض در مسأله‌های مربوط به دور بیش‌تر اثرگذار است. در گراف جهت‌دار، مسأله‌ی داشتن دوری با طول زوج را در نظر بگیرید. جالب است بدانید هنوز راه‌حل کارآمدی برای این مسأله یافته نشده است و بحث در مورد آن ادامه دارد (بخش مراجع پایان فصل را ببینید).

۷-۱۰ تطابق

گراف بدون‌جهت G=(V,E) داده شده است؛ به مجموعه‌ای از یال‌ها که هیچ زوجی از آن‌ها رأس مشترکی نداشته باشند، یک تطابق می‌گویند. به آن تطابق می‌گوییم، چون در آن هر یال مطابقت دو رأس را نشان می‌دهد. بازهم تکرار می‌کنیم که در بین یال‌های تطابق هیچ رأسی متعلق به بیش از یک یال نیست؛ پس در چنین تطابقی، هر عنصر تنها یک عنصر همتا دارد و تطابق از نوع تک‌همتایی است. گاهی رأس‌هایی را که هیچ یک از یال‌های تطابق از آن‌ها نگذرند، رأس‌های تطبیق‌نیافته می‌نامیم و گاهی هم می‌گوییم آن رأس‌ها متعلق به تطابق نیستند. یک تطابق کامل، تطبیقی است که در آن همه‌ی رأس‌های گراف تطبیق‌یافته باشند. یک تطابق بیشینه، تطابقی است که تعداد یال‌هایش بیشینه

باشد. یک تطابق گسترش‌ناپذیر هم، تطبیقی است که نتوانیم یال دیگری به آن بیفزاییم. با مسأله‌های تطابق در زمینه‌های بسیاری (علاوه بر همسریابی در جوامع انسانی!) برخورد می‌کنیم: کارکنان با شغل‌ها، ماشین‌ها با قطعات و ... . (matching در زبان انگلیسی معنای «همسریابی» هم دارد. نویسنده، این معنای دیگر را دست‌مایه‌ی شوخی قرار داده است – مترجمان) مسأله‌های بسیاری هم هستند که ظاهراً به تطبیق ربطی ندارند، اما دارای گونه‌ای هم‌ارز در مسأله‌های تطابق هستند.

تطابق رأس‌های گراف در حالت کلی مسأله‌ای دشوار است. در این بخش، تنها درباره‌ی دو مسأله‌ی خاص از این زمینه بحث می‌کنیم. مسأله نخست، یعنی یافتن همه‌ی تطبیق‌های کامل در دسته‌ی خاصی از گراف‌های بسیار چگال، اهمیت چندانی ندارد؛ اما راه‌حل آن رویکرد جالبی را توضیح می‌دهد که در آینده، این راه‌حل را تعمیم می‌دهیم و از آن برای حل یک مسأله‌ی مهم تطابق در گراف‌های دوبخشی سود می‌جوییم.

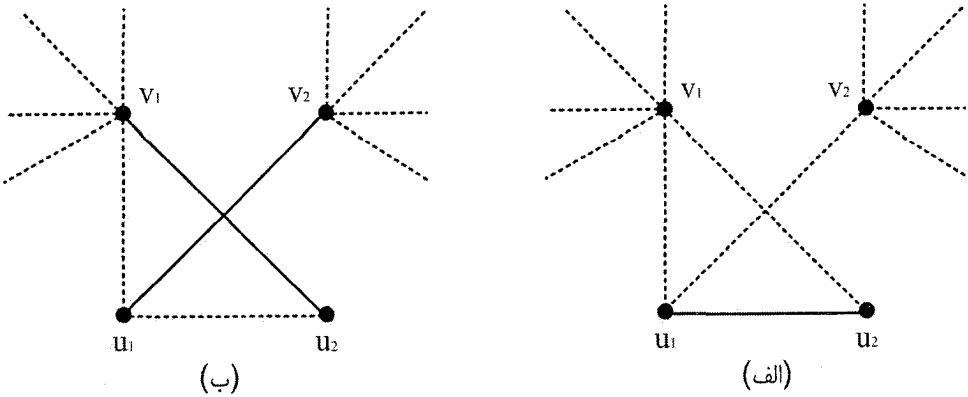
۷-۱۰-۱ یافتن تطبیق‌های کامل در گراف‌های بسیار چگال

در این مثال، حالت خاصی از مسأله‌ی تطابق کامل را در نظر می‌گیریم. گراف G=(V,E) را گرافی بدون‌جهت بگیرید که در آن $|V| = 2n$ و درجه‌ی هر رأس دست‌کم n است. الگوریتمی برای یافتن یک تطبیق کامل در چنین گرافی ارائه می‌کنیم. به عنوان یک نتیجه نشان می‌دهیم که با این شرایط همواره یک تطابق کامل وجود خواهد داشت.

استقرا روی اندازه‌ی تطبیق (m) انجام می‌شود. در حالت پایه (m=1) می‌توانیم هر یالی را به دل‌خواه برگزینیم. خواهیم دید که می‌توان تطبیق‌های ناکامل را با افزودن یک یال نو، یا با جای‌گزینی یک یال موجود با دو یال تازه گسترش داد. در هر یک از این دو حالت، اندازه‌ی تطابق افزایش می‌یابد و به نتیجه نزدیک‌تر می‌شویم.

در گراف G، M را با m یال (m<n) در نظر بگیرید. نخست، همه‌ی یال‌هایی را که در M نیستند، بررسی می‌کنیم تا یالی بیابیم که بتوان آن را به M افزود. اگر چنین یالی بیابیم، کار تمام می‌شود؛ وگرنه M یک تطبیق گسترش‌ناپذیر است. M تطابق کاملی نیست، پس دست‌کم دو رأس غیرمجاور v_1 و v_2 وجود دارند که متعلق به M نیستند. از این دو رأس، دست‌کم 2n یال متمایز خارج شده است و همه‌ی این یال‌ها به رأس‌هایی در M می‌روند؛ چراکه اگر یالی از آن‌ها به رأسی خارج از M می‌رفت، می‌توانستیم همان یال را به M بیفزاییم. از آنجا که تعداد یال‌های M کم‌تر از n است و 2n یال از v_1 و v_2 به رأس‌هایی از Mمی‌روند، دست‌کم یک یال از M – مثلاً (u_1,u_2) – با سه یال خارج‌شده از v_1 و v_2 همسایه‌اند. این سه یال را (u_1,v_1)، (u_1,v_2) و (u_2,v_1) می‌گیریم (شکل ۷-۳۶ (الف) را ببینید) و البته این فرض به کلیت مسأله آسیبی نمی‌رساند. به آسانی می‌توان دید اگر پس

از حذف یال (u_1,u_2) از M، دو یال (u_1,v_2) و (u_2,v_1) را به M بیفزاییم؛ تطبیق بزرگ‌تری به دست می‌آوریم (شکل ۷-۳۶ (ب) را ببینید).



شکل ۷-۳۶ گسترش یک تطابق

در تمرین ۷-۲۱، پیاده‌سازی این الگوریتم را بر عهده‌ی خواننده گذشته‌ایم. شیوه‌ی به کار گرفته‌شده در حل این مسأله، نمونه‌ی دیگری از روش آزمندانه در طراحی الگوریتم بود. در هر گام از گسترش، حداکثر سه یال دخالت دارند. برای این مثال، همین مقدار تلاش کافی بود، اما در حالت کلی، یافتن یک تطبیق خوب دشوارتر است؛ چراکه ممکن است برگزیدن یک یال، روی گزینش یال‌های دورتر گراف هم تأثیر بگذارد. در بخش بعد نشان می‌دهیم چگونه می‌توان این رویکرد را به مسأله‌های دیگر تطابق تعمیم داد.

۷-۱۰-۲ تطابق دوبخشی

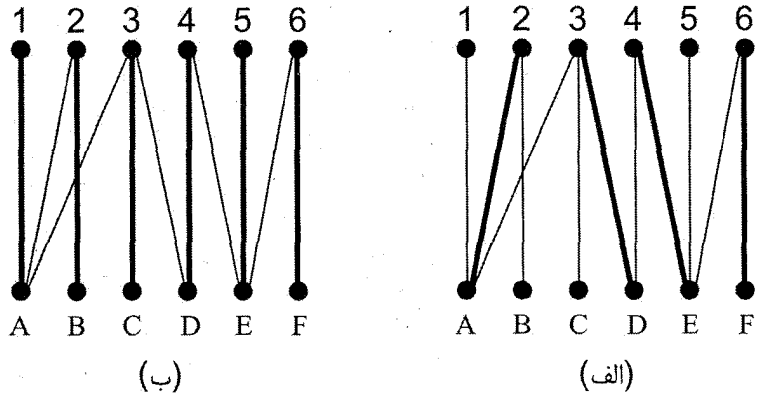
G=(V,E,U) را گرافی دوبخشی بگیرید (یعنی مجموعه‌ی رأس‌های گراف به {U,V} افراز شده است و هر یال گراف، رأسی از V را به رأسی از U متصل می‌کند).

مسأله: تطبیق بیشینه را در گراف دوبخشی G بیابید.

همسریابی گونه‌ای از این مسأله در دنیای واقعی است. V را مجموعه‌ی دختران، U را مجموعه‌ی پسران و E را مجموعه‌ی ازدواج‌های ممکن بگیرید. می‌خواهیم به گونه‌ای برای هر پسر، یک دختر بیابیم که بیش‌ترین تعداد ممکن از پسرها و دخترها متأهل شوند. (ازدواج، تنها بین پسر و دختری ممکن است که با یالی به یکدیگر متصل شده باشند - مترجمان)

رویکرد سرراست استقرا تلاش برای انجام تطبیق با راه‌بردی مشخص تا زمانی است که دیگر همسریابی ممکن نباشد. برای آن‌که راه‌برد به‌کاررفته به بهترین حالت ممکن نزدیک شود، می‌توانیم چند ایده را بیازماییم؛ مثلاً می‌توانیم به شیوه‌ای آزمندانه رفتار کنیم، یعنی نخست رأس‌هایی را تطبیق

دهیم که درجه‌ی کوچک‌تری دارند. برای رأس‌هایی که درجه‌ی آن‌ها بزرگ‌تر است، در آینده، شانس بیش‌تری برای یافتن «همتایی تطبیق‌نیافته» باقی می‌ماند. (به عبارت دیگر، نخست پسری، همسر خود را برمی‌گزیند که مشکل‌پسندتر است؛ سپس فکری برای دیگر پسران می‌کنیم.) از آنجا که تحلیل چنین راه‌بردهایی دشوار است، می‌کوشیم رویکرد مسأله‌ی پیش را به کار گیریم. فرض کنید کار را با تطبیقی گسترش‌ناپذیر که لزوماً تطابقی بیشینه نیست، آغاز می‌کنیم. آیا راهی برای بهبود تطابق وجود دارد؟ شکل ۷-۳۷ (الف) را در نظر بگیرید که در آن، یال‌های متعلق به تطابق پررنگ‌تر هستند. روشن است که می‌توانیم تطابق را با جای‌گزینی دو یال ۱A و ۲B به جای ۲A گسترش دهیم. این کار گونه‌ی دیگری از همان روش به‌کاررفته در مسأله‌ی پیش است، اما خود را به «جای‌گزینی یک یال با دو یال» محدود نمی‌کنیم. اگر به روشی بتوانیم k+۱ یال را به جای k یال قرار دهیم، بازهم تطبیق را بهبود داده‌ایم. برای مثال، در شکل ۷-۳۷ (الف) می‌توان با جای‌گزینی ۳C، ۴D و ۵E به جای ۳D و ۴E، تطابق را گسترش داد.



شکل ۷-۳۷ گسترش یک تطابق دوبخشی

بیایید این جای‌گزینی‌ها را بررسی کنیم. هدف ما افزایش شمار رأس‌های تطبیق‌یافته است. با رأسی تطبیق‌نیافته (v) کار را آغاز می‌کنیم و می‌کوشیم همتایی برای آن بیابیم. اگر تطبیق از پیش، گسترش‌ناپذیر بوده است، تمام همسایگان v تطبیق‌یافته هستند. پس باید یکی از تطبیق‌های پیشین را کنار بگذاریم. یکی از همسایگان تطبیق‌یافته‌ی v را برمی‌گزینیم (مثلاً u که با w تطابق یافته است). مطابقت بین u و w را بر هم می‌زنیم و v را با u تطبیق می‌دهیم. حالا باید همتایی برای w بیابیم. اگر w به رأسی تطبیق‌نیافته متصل باشد، کار تمام است (نخستین حالت گفته‌شده برای شکل ۷-۳۷) اما اگر همسایگان w همگی تطبیق‌یافته باشند، می‌توانیم با همین روش برخی تطبیق‌ها را کنار گذاشته، بکوشیم تطبیق‌های تازه‌ای را جای‌گزین کنیم. باید دو کار انجام دهیم تا بتوانیم این رویکرد را به یک الگوریتم تبدیل کنیم: هم باید مطمئن شویم این روال، سرانجام به پایان می‌رسد و هم باید نشان دهیم

اگر گسترشی ممکن باشد، این روال، بی‌گمان آن را خواهد یافت. نخست، این ایده را با زبان ریاضی بیان می‌کنیم.

«مسیر یک‌درمیان P» برای تطبیقی مانند M، مسیری از رأسی مانند $v \in V$ به رأسی مانند $u \in U$ است که نه v، نه u، هیچ یک در M تطبیق نیافته باشند و یال‌های P یک در میان متعلق به E-M و M باشند. (چون v به M تعلق ندارد، پس یال نخست P (یعنی (u,w)) متعلق به M نیست، یال دوم P (یعنی (w,x)) متعلق به M است و ... . آخرین یال P (یعنی (z,u)) هم به M تعلق ندارد.) توجه کنید که همین مسیرهای یک‌درمیان را برای بهبود تطابق به کار بردیم. P از رأسی در V آغاز می‌شود و به رأسی در U پایان می‌یابد، پس تعداد یال‌های آن فرد است. در ضمن، تعداد یال‌هایی از P که متعلق به M نیستند، دقیقاً یکی بیش‌تر از تعداد یال‌هایی از P است که به M تعلق دارند. بنابراین، اگر به جای همه‌ی یال‌هایی از P که به M تعلق دارند، یال‌هایی از همین مسیر را بگذاریم که به M تعلق ندارند، به تطابقی با یک یال بیش‌تر می‌رسیم. برای مثال، نخستین مسیر یک‌درمیانی که برای بهبود تطابق شکل ۷-۳۷ (الف) به کاربردیم، (۱A، ۲A و ۲B) بود که یال‌های ۱A و ۲B را جای‌گزین یال ۲A کرد. دومین مسیر یک‌درمیان (۳C، ۳D، ۴D، ۴E و ۵E) بود که یال‌های ۳C، ۴D و ۵E را جای‌گزین یال‌های ۳D و ۴E کرد.

تا به حال دیگر باید برای شما روشن شده باشد که اگر برای یک تطابق داده‌شده‌ی M، مسیری یک‌درمیان وجود داشته باشد، آنگاه M بیشینه نخواهد بود. عکس این قضیه نیز درست است.

□ قضیه‌ی مسیر یک‌درمیان

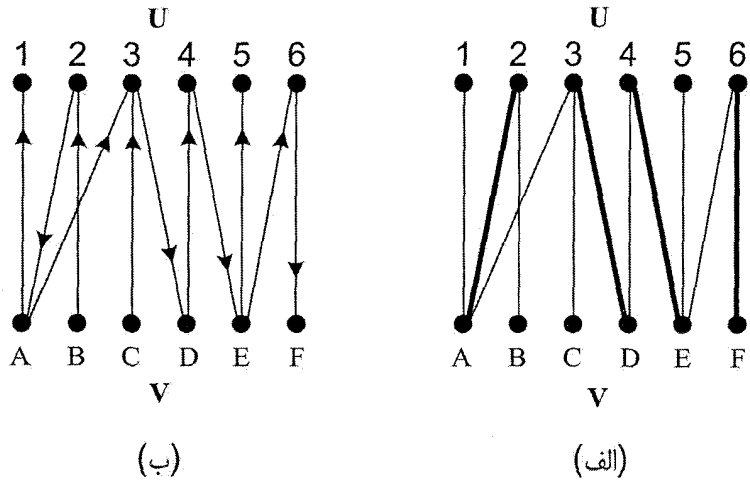
یک تطابق، بیشینه است، اگر و تنها اگر هیچ مسیر یک‌درمیانی نداشته باشد.

□

این ادعا حالتی ویژه از قضیه‌ای است که در بخش بعد ثابت خواهد شد.

از قضیه‌ی مسیر یک‌درمیان به سادگی به یک الگوریتم می‌رسیم، چراکه هر تطبیق نابیشینه، مسیری یک‌درمیان دارد و هر مسیر یک‌درمیان موجود می‌تواند تطابق را گسترش دهد. کار را با الگوریتمی آزمندانه آغاز می‌کنیم و به تطابق تا جایی که می‌توانیم یال می‌افزاییم و این کار را تا رسیدن به تطبیقی گسترش‌ناپذیر ادامه می‌دهیم. سپس به دنبال مسیرهای یک‌درمیان می‌گردیم و تطابق را آن قدر به یاری این مسیرها گسترش می‌دهیم تا دیگر هیچ مسیر یک‌درمیانی پیدا نشود. تطابق به‌دست‌آمده بیشینه خواهد بود. هر مسیر یک‌درمیان، یک یال به تطابق می‌افزاید و هیچ تطبیقی (با n رأس) بیش از n/۲ یال نخواهد داشت؛ پس فرایند یافتن مسیر یک‌درمیان حداکثر n/۲ بار انجام می‌شود. تنها مسأله‌ای که باقی می‌ماند، شیوه‌ی یافتن مسیرهای یک‌درمیان است. برای حل این مسأله، گراف بدون‌جهت G را به گراف جهت‌دار G' چنان تبدیل می‌کنیم که جهت هر یال متعلق به M، از U به V و جهت هر یال متعلق به E-M، از V به U باشد. شکل ۷-۳۸ (الف) نشانگر تطابق به‌دست‌آمده برای گراف شکل ۷-۳۷ (الف) است؛ شکل ۷-۳۸ (ب) نیز گراف جهت‌دار G' را نشان می‌دهد. هر مسیر

یک‌درمیان دقیقاً با مسیری جهت‌دار از رأسی تطبیق‌نیافته در V به رأسی تطبیق‌نیافته در U متناظر است. هر یک از الگوریتم‌های جست‌وجوکننده‌ی گراف مانند DFS می‌توانند چنین مسیرهایی را بیابند. از آنجا که می‌توان چنین جست‌وجویی را در زمانی از $O(|V|+|E|)$ انجام داد، پس پیچیدگی الگوریتم اصلی از $O(|V|(|V|+|E|))$ خواهد بود.



(ب) (الف)

شکل ۷-۳۸ یافتن مسیرهای یک‌درمیان

بهبود الگوریتم

هنگام تحلیل بدترین حالت، زمان پیمایش کل گراف با زمان پیمودن مسیرهای مورد نظر ما برابر است. پس بهتر بود در هر پیمایش گراف می‌کوشیدیم چند مسیر یک‌درمیان بیابیم؛ البته باید مطمئن شویم که این مسیرها روی یکدیگر اثر نمی‌گذارند. یک راه تضمین استقلال چنین مسیرهایی، محدود کردن آن‌ها به مسیرهایی با رأس‌های جداازهم است. اگر رأس‌های هر یک از این مسیرها جداازهم باشند، هر مسیر بر رأس‌های متفاوتی اثر می‌گذارد، پس می‌توان آن‌ها را هم‌زمان با یکدیگر یافت. الگوریتم بهبودیافته‌ی پیدا کردن مسیرهای یک‌درمیان چنین خواهد بود: نخست در G'، BFS را از همه‌ی رأس‌های تطبیق‌نیافته V آغاز می‌کنیم و سطح به سطح پیش می‌رویم تا به سطحی برسیم که رأس‌هایی تطبیق‌نیافته از U پیدا شوند. پس از یافتن این رأس‌ها، از گراف القاشده با BFS، مجموعه‌ای گسترش‌ناپذیر از مسیرهایی در G' بیرون می‌کشیم که رأس‌های آن‌ها جداازهم باشد (در G این مسیرها یک‌درمیان هستند). برای انجام این کار مسیری می‌یابیم، رأس‌های آن را حذف می‌کنیم، مسیر دیگری می‌یابیم، رأس‌های آن را حذف می‌کنیم و ... . (نتیجه‌ی کار یک مجموعه‌ی گسترش‌ناپذیر است، نه یک مجموعه‌ی بیشینه.) مجموعه‌ای گسترش‌ناپذیر را برمی‌گزینیم تا تعداد یال‌هایی را که در

هر جست‌وجو به تطابق افزوده می‌شوند، بیشینه سازیم. (هر یک از این مسیرهای یک‌درمیانی که رأس‌های آن‌ها جداازهم بود، یک یال به تطبیق می‌افزاید.) در پایان، تطابق را با به‌کارگیری مجموعه‌ی مسیرهای یک‌درمیان بهبود می‌دهیم. این فرایند تا جایی تکرار می‌شود که دیگر هیچ مسیر یک‌درمیانی پیدا نکنیم (یعنی در گراف جهت‌دار تازه‌ی G' مسیری از رأس‌های تطبیق‌نیافته‌ی V به رأس‌های تطبیق‌نیافته‌ی U وجود نداشته باشد).

پیچیدگی: در بدترین حالت، تعداد دفعات تکرار در الگوریتم بهبودیافته از $O(\sqrt{n})$ است. برهان را که از Hopcroft و karp [۱۹۷۳] است، در اینجا نمی‌آوریم. بنابراین زمان کل اجرا در بدترین حالت از $O((|V|+|E|)\sqrt{|V|})$ خواهد بود.

۷-۱۱ شارهای شبکه

مسأله‌ی شارهای شبکه یکی از مسأله‌های بنیادی در نظریه‌ی گراف و بهینه‌سازی ترکیبیاتی است. در ۳۵ سال اخیر، این مسأله را بسیار مورد توجه قرار داده‌اند و به تدریج الگوریتم‌ها و ساختمان‌های داده‌ای بسیاری برای آن طراحی کرده‌اند. این مسأله گونه‌ها و تعمیم‌های فراوانی دارد. مسأله‌های بسیاری را – که ظاهراً هیچ ربطی به یکدیگر ندارند – می‌توان به شبکه‌ی شاره تبدیل کرد. گونه‌ی اصلی مسأله چنین است: G=(V,E) را گرافی جهت‌دار با دو رأس مشخص s (چشمه) با درجه‌ی ورودی ۰ و t (چاهک) با درجه‌ی خروجی ۰ در نظر بگیرید. به هر یال $e \in E$ وزن مثبت c(e) نسبت داده شده است که آن را گنجایش e می‌گوییم. گنجایش یا ظرفیت هر یال، حداکثر شاری را نشان می‌دهد که می‌تواند از آن یال بگذرد. چنین گرافی را یک شبکه می‌گوییم. برای راحتی، ظرفیت یال‌هایی را که در گراف وجود ندارند، صفر می‌گیریم. شار، تابعی همچون f روی یال‌های شبکه است که دارای این دو شرط باشد:

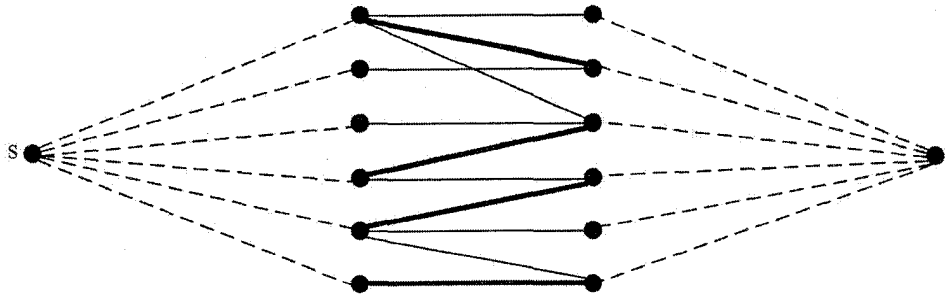
۱- $0 \le f(e) \le c(e)$ ؛ یعنی شار هیچ یالی بیش از گنجایش آن نیست.

۲- به ازای هر $v \in V - \{s,t\}$: $\sum_u f(u,v) = \sum_w f(v,w)$ ؛ یعنی مجموع شارهای ورودی به هر رأس (به جز چشمه و چاهک) برابر مجموع شارهای خروجی از آن است.

از این دو شرط نتیجه می‌شود که مجموع کل شارهای خروجی از s برابر با مجموع کل شارهای ورودی به t است. مسأله، بیشینه کردن این مجموع است. (اگر ظرفیت‌ها اعداد حقیقی باشند، حتا روشن نیست که آیا شار ممکن، مقدار بیشینه‌ای دارد یا نه؛ نشان خواهیم داد که شار ممکن واقعاً مقداری بیشینه دارد.) یک راه تجسم این مسأله، در نظر گرفتن شبکه‌ای از لوله‌های آب است. می‌خواهیم آب گذرنده از شبکه‌ی لوله‌ها بیش‌ترین مقدار ممکن باشد، اما اگر آب واردشونده به یک بخش بیش از حد باشد، لوله‌ها می‌ترکند.

نخست نشان می‌دهیم که مسأله‌ی تطبیق دوبخشی را - که در بخش پیش بررسی شد - می‌توان به صورت یک مسأله‌ی شبکه‌ی شاره بیان کرد. از آنجا که مسأله‌ی تطبیق دوبخشی را حل کرده‌ایم، اما هنوز راه‌حل مسأله‌ی شبکه‌ی شاره را نمی‌دانیم، این کار ظاهراً بی‌فایده است (یعنی جهت این کاهش نادرست است). این کاهش را با وجود جهت نادرست آن ارائه می‌کنیم، چراکه ترفندهای به‌کاررفته برای حل مسأله‌ی شبکه‌ی شاره مانند ترفندهای حل مسأله‌ی تطبیق دوبخشی است و درک این شباهت‌ها به فهم الگوریتم‌های شبکه‌ی شاره کمک می‌کند.

گراف دوبخشی $G=(V,E,U)$ داده شده است و می‌خواهیم تطبیقی با بیش‌ترین اندازه‌ی ممکن در آن بیابیم. دو رأس تازه‌ی s و t را به گراف اضافه می‌کنیم. از s به تمام رأس‌های V و از تمام رأس‌های U به t یال می‌افزاییم. جهت همه‌ی یال‌های E را نیز از V به U قرار می‌دهیم (شکل ۷-۳۹ را ببینید که در آن، جهت همه‌ی یال‌ها از چپ به راست است). ظرفیت همه‌ی یال‌ها را ۱ قرار می‌دهیم. حال به یک مسأله‌ی شبکه‌ی شاره روی گراف تغییریافته‌ی G' رسیده‌ایم. M را یک تطابق در G بگیرید. M با یک شار یا جریان در G' متناظر است. به همه‌ی یال‌های M و همه‌ی یال‌هایی که s یا t را به رأسی تطبیق‌یافته در M متصل می‌کنند، شار ۱ و به یال‌های دیگر شار ۰ نسبت می‌دهیم. کل شار برابر تعداد یال‌های تطبیق است. روشن خواهد شد که M تطبیقی بیشینه است، اگر و تنها اگر شار متناظر با آن در G' بیشینه باشد. یک سمت این ادعا روشن است: اگر شار، بیشینه و متناظر با یک تطبیق باشد، آنگاه نمی‌توانیم تطبیق بزرگ‌تری داشته باشیم؛ چراکه این تطبیقِ بزرگ‌تر، متناظر با شاری بیش‌تر خواهد شد. برای اثبات سمت دیگر ادعا باید معادل مناسبی برای مسیرهای یک‌درمیان، در شبکه‌ی شاره بیابیم و نشان دهیم اگر مسیر یک‌درمیانی وجود نداشته باشد، شار متناظر بیشینه است.



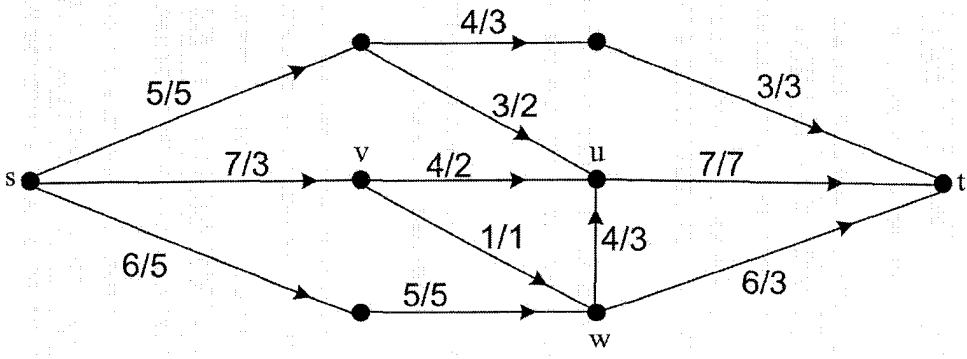
شکل ۷-۳۹ کاهش مسأله‌ی تطبیق دوبخشی به شبکه‌ی شاره (جهت همه‌ی یال‌ها از چپ به راست است).

یک مسیر افزاینده یا افزایشی برای شار f، مسیری است جهت‌دار از s به t که تعدادی از یال‌های G را اما نه لزوماً با همان جهت در برداشته باشد و هر یک از این یال‌ها - مثلاً (v,u) - دقیقاً در یکی از دو شرط صفحه‌ی بعد صادق باشد:

۱- جهت (v,u) با جهت آن در G یکسان است و f(v,u)<c(v,u). در این حالت به یال (v,u)، یک یال جلورو می‌گوییم. هر یال جلورو برای شار بیش‌تری نیز ظرفیت دارد. به c(v,u)-f(v,u) ظرفیت مانده‌ی یال می‌گویند.

۲- (v,u) با جهت مخالف در G است (یعنی $(u,v) \in E$) و $f(u,v)>0$. در این حالت، یال (v,u) را عقب‌رو می‌گوییم. می‌توان از یک یال عقب‌رو مقداری شار «قرض گرفت».

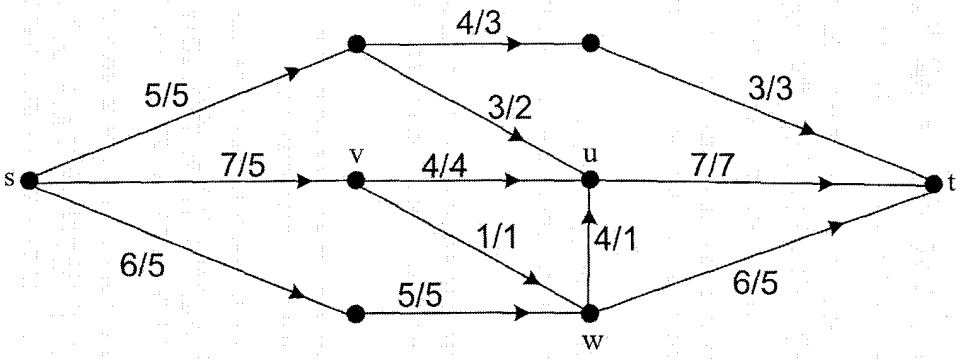
مسیرهای افزاینده، تعمیم مسیرهای یک‌درمیان هستند و همان نقشی را که مسیرهای یک‌درمیان در تطابق دوبخشی بر عهده داشتند، در شبکه‌ی شاره بازی می‌کنند. اگر در شار f مسیری افزایشی وجود داشته باشد (که در این صورت می‌گوییم f مسیری افزایشی را می‌پذیرد) آنگاه f بیشینه نخواهد بود. می‌توانیم f را با فرستادن شار بیش‌تری از راه هر مسیر افزاینده تغییر دهیم. اگر همه‌ی یال‌های مسیر جلورو باشند، می‌توانیم از راه آن‌ها شار بیش‌تری بفرستیم و همه‌ی شرایط لازم بازهم برقرار خواهد بود. در این حالت، شار اضافی دقیقاً برابر کم‌ترین مقدار در بین «ظرفیت مانده‌ی» یال‌های مسیر است، اما اگر یال عقب‌رو هم وجود داشته باشد، مسأله کمی پیچیده‌تر می‌شود. به شکل ۷-۴۰ نگاه کنید که در آن به هر یال برچسبی به صورت a/b زده شده است. در این برچسب، a، ظرفیت و b، مقدار فعلی شار را نشان می‌دهد. چون مسیری از s به t وجود ندارد که تنها از یال‌های جلورو تشکیل شده باشد، پس به نظر می‌رسد که در این شکل نمی‌توان شار بیش‌تری از چشمه به چاهک فرستاد؛ اما چنین نیست.



شکل ۷-۴۰ نمونه‌ای از یک شبکه با شار (نابیشینه)

مسیر s، v، u، w و t یک مسیر افزایشی است و می‌توان با این مسیر، ۲ واحد شار اضافه از s به u فرستاد. (۲ کمینه‌ی ظرفیت‌های مانده در بین همه‌ی یال‌های جلورو مسیر تا u است.) می‌توانیم ۲ واحد شار از f(w,u) کم کنیم. ۲ واحد شار از راه مسیر افزایشی به u افزوده شده و ۲ واحد شار از راه یال عقب‌رو، یعنی (w,u) از آن کاسته شده است؛ بدین ترتیب، هنوز شرایط پایستگی در u برقرار است. حال، ۲ واحد شار اضافی در w داریم که باید از آن خارج شود؛ یعنی دقیقاً آنچه ما می‌خواهیم. می‌توانیم این ۲ واحد شار را از راه یال جلورو، یعنی (w,t) بگذرانیم و چون ۲ واحد از شار یال عقب‌رو، یعنی

(w,u) کاسته‌ایم؛ شرایط لازم در w نیز برقرار می‌ماند. در این مورد، چون یال جلوروی (w,t) به t می‌رسد، کار تمام است. از آنجا که تنها یال‌های جلورو از s خارج و به t وارد می‌شوند، شار کل افزایش خواهد یافت. مقدار این افزایش برابر با کم‌ترین مقدار بین کمینه‌ی ظرفیت‌های مانده‌ی یال‌های جلورو و کمینه‌ی شارهای فعلی یال‌های عقب‌رو خواهد بود. شکل ۷-۴۱ همین شبکه را با شار تغییریافته نشان می‌دهد. (در واقع، این شار بیشینه است.)



شکل ۷-۴۱ نتیجه‌ی افزایش شار در شکل ۷-۴۰

بحث پیش نشان می‌دهد که اگر مسیری افزایشی وجود داشته باشد، شار، بیشینه نیست. عکس این مطلب نیز درست است.

☐ قضیه‌ی مسیر افزایشی

شار f بیشینه است، اگر و تنها اگر هیچ مسیر افزاینده‌ای نداشته باشد.

برهان: درستی یک سمت قضیه را نشان دادیم؛ پس باید بدانید اگر شبکه‌ای مسیر افزایشی داشته باشد، آنگاه شار آن بیشینه نیست. حالا بیایید فرض کنیم شار f مسیر افزایشی ندارد. باید ثابت کنیم f بیشینه است. مفهوم «برش» را به کار می‌بریم. یک برش به صورت شهودی مجموعه‌ای است از یال‌ها که s را از t جدا می‌کند. به بیان دقیق‌تر، A را مجموعه‌ای از رأس‌های V بگیرید به گونه‌ای که $s \in A$ و $t \notin A$. بقیه‌ی رأس‌ها را با B نشان می‌دهیم؛ یعنی B=V-A. یک برش مجموعه‌ای از یال‌های $\{(v,w) \in E\}$ است، به گونه‌ای که $v \in A$ و $w \in B$. ظرفیت یک برش را برابر مجموع ظرفیت یال‌های آن تعریف می‌کنیم. روشن است که شار نمی‌تواند بیش‌تر از ظرفیت هیچ یک از برش‌های شبکه باشد. (اگر لوله‌های آب را ببندید، از آن‌ها آب نخواهد گذشت.) از این رو، اگر شاری بیابید که مقدارش با ظرفیت یک برش برابر گردد، آنگاه این شار باید بیشینه باشد. برهان را با اثبات این مطلب کامل می‌کنیم: اگر در شاری اصلاً مسیر افزاینده‌ای وجود نداشته باشد، آنگاه مقدار شار برابر با ظرفیت یک برش و در نتیجه، بیشینه است.

f را شاری بگیرید که هیچ مسیر افزاینده‌ای ندارد. $A(A \subset V)$ را مجموعه‌ای از رأس‌ها به گونه‌ای در نظر بگیرید که به ازای هر $v \in A$ یک مسیر افزایشی برای شار f از s به v وجود داشته

باشد. روشن است که $s \in A$ و $t \notin A$ (چون فرض کردیم که f هیچ مسیر افزاینده‌ای ندارد). پس A یک برش است. ادعا می‌کنیم که برای هر یال (v,w) از این برش، f(v,w) برابر c(v,w) است؛ چراکه در غیر این صورت، (v,w) باید یالی جلورو باشد و مسیری افزایشی به w وجود داشته باشد که با فرض $w \notin A$ در تناقض است. با همین روش می‌توان ثابت کرد که یالی مانند (w,v) وجود ندارد که هم $w \notin A$، هم $v \in A$ و هم $f(w,v)0$ (چراکه اگر چنین یالی وجود داشته باشد، حتماً عقب‌رو است و در نتیجه، مسیری افزایشی تشکیل می‌دهد). بدین ترتیب، مقدار شار f با ظرفیت برش A برابر و در نتیجه، بیشینه است.

□

پس یک قضیه‌ی بنیادی را ثابت کردیم:

□ قضیه‌ی شار بیشینه - برش کمینه

شار بیشینه در هر شبکه برابر با کمینه‌ی ظرفیت برش‌های آن است.

□

از قضیه‌ی مسیر افزایشی قضیه‌ی بعدی نیز به دست می‌آید.

□ قضیه‌ی شار مجموع

اگر ظرفیت هر یال شبکه عددی صحیح باشد، آنگاه شاری بیشینه وجود خواهد داشت و مقدار آن هم یک عدد صحیح است.

برهان: این قضیه مستقیماً از قضیه‌ی مسیر افزایشی به دست می‌آید. در واقع، هر الگوریتم که تنها مسیرهای افزایشی را به کار ببرد، هنگامی که همه‌ی ظرفیت‌ها اعدادی صحیح باشند؛ به یک شار مجموع منتهی خواهد شد. از آنجا که کار را با مقدار شار ۰ آغاز می‌کنیم و هر مسیر افزایشی، یک عدد صحیح را به شار کل می‌افزاید، پس درستی این قضیه روشن است.

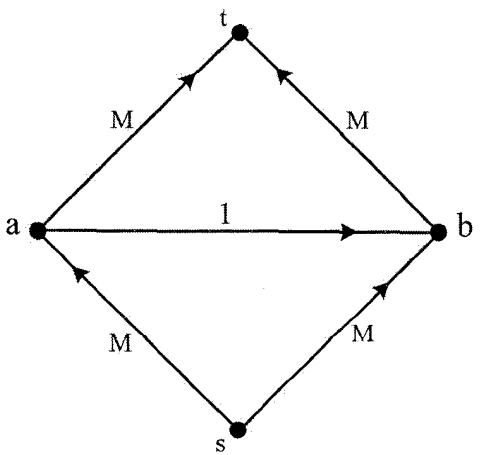
□

اینک دوباره به مسأله‌ی تطابق دوبخشی بازمی‌گردیم. پیداست که هر مسیر یک‌درمیان در G با یک مسیر افزایشی در G' متناظر است و برعکس. از قضیه‌ی مسیر افزایشی، درستی قضیه‌ی مسیر یک‌درمیان (از بخش پیش) به دست می‌آید. اگر M تطابقی بیشینه باشد، آنگاه هیچ مسیر یک‌درمیانی برای آن وجود نخواهد داشت؛ پس مسیر افزاینده‌ای هم در G' وجود ندارد؛ یعنی شار، بیشینه است. از سوی دیگر، یک شار مجموع بیشینه وجود دارد که به روشنی متناظر با یک تطبیق است، چراکه هر رأس از V تنها با یک یال (با ظرفیت ۱) به s متصل شده است؛ از این رو، از هر رأس V تنها می‌توان شاری با مقدار ۱ گذراند. همین استدلال برای رأس‌های U هم درست است. این تطابق، بیشینه است، چراکه اگر قابل‌گسترش بود، می‌توانستیم شار بزرگ‌تری هم به دست آوریم.

از قضیه‌ی مسیر افزایشی مستقیماً یک الگوریتم به دست می‌آید. کار را با شار ۰ آغاز می‌کنیم، به دنبال مسیری افزایشی می‌گردیم و شار را برحسب آن افزایش می‌دهیم و این کار را آن قدر تکرار

می‌کنیم تا دیگر مسیر افزاینده‌ای وجود نداشته باشد. در طی این فرایند، داریم شار را می‌افزاییم، پس کار در جهتی درست پیش می‌رود. می‌توان با روشی که در ادامه‌ی بحث می‌آید، مسیرهای افزایشی را یافت. برای شبکه‌ی G=(V,E) با شار f، «گراف ظرفیت‌های باقی‌مانده» را چنین تعریف می‌کنیم: هر شبکه‌ی R=(V,F) که همان رأس‌های G، همان چشمه و چاهک و همان یال‌ها را داشته باشد، اما جهت و ظرفیت یال‌ها می‌تواند متفاوت باشد. یال‌های «گراف ظرفیت‌های باقی‌مانده» متناظر با یال‌هایی هستند که می‌توانند در مسیری افزایشی قرار بگیرند. ظرفیت یال‌های «گراف ظرفیت‌های باقی‌مانده» برابر با مقدار شار افزایشی احتمالی از راه همین یال‌ها در گراف اصلی است. به بیان دقیق تر، یال (v,w) متعلق به F است، اگر یالی جلورو و یا یالی عقب‌رو باشد. گنجایش یال، در حالت نخست، برابر با c(v,w)-f(v,w) و در حالت دوم، برابر با f(v,w) خواهد بود. پس، یک مسیر افزایشی، مسیری جهت‌دار و معمولی از s به t در «گراف ظرفیت‌های باقی‌مانده» است. از آنجا که هر یال باید دقیقاً یک بار بررسی شود، پس ساخت «گراف ظرفیت‌های باقی‌مانده» نیازمند $|E|$ گام است.

بدبختانه، با برگزیدن آزادانه‌ی مسیرهای افزایشی ممکن است به الگوریتمی بسیار کند برسیم. زمان اجرای بدترین حالت برای چنین الگوریتمی ممکن است حتا تابعی از اندازه‌ی گراف نباشد. شبکه‌ی شکل ۷-۴۲ را در نظر بگیرید. شار بیشینه بی‌گمان 2M است؛ اما شاید فردی، کار را با مسیر s، a، b و t آغاز کند که مقدار شار آن ۱ است. سپس همین فرد ممکن است مسیر افزایشی s، b، a و t را برگزیند که بازهم شار را ۱ واحد افزایش می‌دهد. درست است که گراف، تنها چهار رأس و پنج یال دارد، اما شاید M بسیار بزرگ باشد و احتمال دارد فرایند گفته‌شده، 2M بار تکرار شود. (از آنجا که مقدار M را می‌توان با تعداد بیت‌هایی از O(logM) ذخیره کرد، پس در بدترین حالت، زمان اجرای این الگوریتم نسبت به اندازه‌ی ورودی، نمایی خواهد شد.)



شکل ۷-۴۲ یک شبکه‌ی شاره که با گزینش آزادانه‌ی مسیرهای افزایشی در آن ممکن است برای یافتن شار بیشینه به زمان بسیار زیادی نیازمند باشیم.

هرچند احتمال بروز چنین حالتی بسیار اندک است، اما باید هشیار باشیم و نگذاریم چنین حالتی رخ دهد. به علاوه، ما می‌خواهیم تعداد افزایش‌ها را کمینه کنیم تا سرعت الگوریتم بیش‌تر شود. Edmond و Karp [۱۹۷۲] چند پیش‌نهاد ارائه کرده‌اند؛ مثلاً مسیر افزایشی بعدی، مسیری افزاینده با حداقل تعداد یال‌ها باشد و ثابت کرده‌اند که اگر با این روش، کار را ادامه دهیم، آنگاه تعداد افزایش‌های لازم حداکثر $(|V|^3 - |V|)/4$ خواهد شد. با این روش، در بدترین حالت، الگوریتم برحسب اندازه‌ی ورودی چندجمله‌ای است. از آن هنگام تاکنون الگوریتم‌های فراوانی پیش‌نهاد شده‌اند. برخی از این الگوریتم‌ها پیچیده‌اند و برخی ساده‌تر (اما هیچ یک واقعاً ساده نیستند). حد بالای پیچیدگی برخی از این الگوریتم‌های شبکه‌ی شاره از $O(|V|^3)$ است. این الگوریتم‌ها را شرح نخواهیم داد (می‌توانید شرح کامل آن‌ها را در مراجع پایان فصل بیابید).

۷-۱۲ دورهای هامیلتونی

این فصل را با بحث درباره‌ی دوری دربردارنده‌ی همه‌ی یال‌های گراف آغاز کردیم و حال، آن را با بحثی درباره‌ی دوری دربرگیرنده‌ی تمام رأس‌های گراف به پایان می‌رسانیم. نام این مسأله‌ی مشهور از ریاضی‌دان ایرلندی Sir William R. Hamilton گرفته شده است که در سال ۱۸۷۵ میلادی بر اساس این مسأله، یک بازی عامه‌پسند طراحی کرد.

مسأله: گراف G=(V,E) داده شده است. دوری ساده در آن بیابید که از هر رأس V دقیقاً یک بار بگذرد.

به چنین دوری، دور هامیلتونی و به گراف‌هایی که چنین دورهایی داشته باشند، گراف‌های هامیلتونی می‌گویند. این مسأله را می‌توان هم در گراف‌های جهت‌دار و هم در گراف‌های بدون‌جهت مطرح کرد، ولی ما تنها نوع بدون‌جهت آن را بررسی می‌کنیم.

مسأله‌ی یافتن دورهای هامیلتونی (یا همان تشخیص گراف‌های هامیلتونی) برخلاف مسأله‌ی دورهای اویلری بسیار دشوار است و مسأله‌ای NP-تمام (فصل ۱۱) محسوب می‌گردد. در این بخش، نمونه‌ی ساده‌ای از مسأله ارائه می‌کنیم که دورهای هامیلتونی را تنها در دسته‌ای از گراف‌های بسیار چگال می‌یابد. جالب‌ترین بخش این مثال، بهره‌گیری از روشی زیبا به نام استقرای معکوس است.

۷-۱۲-۱ استقرای معکوس

پیش‌تر در بخش ۲-۱۱ استقرای معکوس را بررسی کردیم. ایده‌ی این استقرا به‌کارگیری مجموعه‌ی نامتناهی S (مثلاً $S = \{2^k\}$ و $k=1,2,\ldots$) به عنوان حالت پایه است؛ یعنی ثابت می‌کنیم به ازای هر

$n \in S$، قضیه‌ی P(n) برقرار است. سپس رو به عقب حرکت کرده، ثابت می‌کنیم از درستی P(n)، درستی P(n-1) به دست می‌آید. معمولاً در ریاضیات، حرکت از n به n-1 آسان‌تر از حرکت از n-1 به n نیست. اثبات حالت پایه‌ی نامتناهی نیز از اثبات حالت پایه‌ی ساده بسیار دشوارتر است؛ اما در طراحی الگوریتم رفتن از n به n-1 تقریباً همیشه آسان‌تر است؛ یعنی حل مسأله برای ورودی کوچک‌تر ساده‌تر است. برای مثال، می‌توانیم ورودی‌های «پوچ» یا «قلابی» را که اثری روی خروجی ندارند، به کار ببریم. پس در بسیاری موارد کافی است تنها برای ورودی‌هایی که اندازه‌ی آن‌ها از یک مجموعه‌ی نامتناهی گرفته می‌شود، الگوریتم طراحی کنیم؛ نه برای هر ورودی دل‌خواهی. رایج‌ترین شیوه‌ی به‌کارگیری اصل استقرا به صورت معکوس، طراحی الگوریتم برای ورودی‌هایی است که اندازه‌ی آن‌ها توانی از ۲ باشد. با این کار، طراحی تروتمیزتر شده، بسیاری از جزئیات دست‌وپاگیر کنار گذاشته می‌شود. روشن است که سرانجام این جزئیات نیز حل خواهند شد؛ اما اگر نخست، مسأله‌ی اصلی را حل کنیم، ادامه‌ی کار راحت‌تر می‌شود. چند جای کتاب (مثلاً در بخش‌های ۸-۲ و ۹-۴) فرض کرده‌ایم که اندازه‌ی ورودی توانی از ۲ است.

بهره‌گیری از این شیوه، هنگامی هم که تعداد عناصر ممکن محدود باشد، سودمند است. حالت پایه‌ی استقرا می‌تواند به جای نمونه‌ای با تعداد عناصر کمینه، نمونه‌ای با تعداد عناصر بیشینه باشد و پس از آن، با حرکت رو به عقب قضیه ثابت شود. برای مثال، فرض کنید بخواهیم قضیه‌ای را درباره‌ی گراف‌ها با به‌کارگیری استقرا روی تعداد یال‌ها ثابت کنیم. می‌توانیم کار را با گراف کامل آغاز کنیم. برای تعداد ثابتی رأس، تعداد یال‌های این گراف بیشینه است. سپس می‌توانیم ثابت کنیم اگر یالی را حذف کنیم، بازهم قضیه برقرار است (برخلاف حالت معمولی که یالی را به گراف می‌افزاییم و ثابت می‌کنیم که هنوز قضیه درست است). با این روش دستمان در کاربرد استقرا بازتر است. با الگوریتم بعد، اصل استقرای معکوس را روشن‌تر می‌سازیم.

۷-۱۲-۲ یافتن دورهای هامیلتونی در گراف‌های بسیار چگال

G=(V,E) را گرافی بدون‌جهت و همبند و d(v) را درجه‌ی رأس v از آن بگیرید. مسأله‌ی بعد، یافتن دورهای هامیلتونی در گراف‌های بسیار چگال است. این مسأله را با شرایطی ویژه تعریف می‌کنیم و نشان خواهیم داد که این شرایط، هامیلتونی بودن گراف را تضمین می‌کنند. حال، مسأله را (که نمونه‌ی خوبی از کاربرد استقرای معکوس است) شرح می‌دهیم:

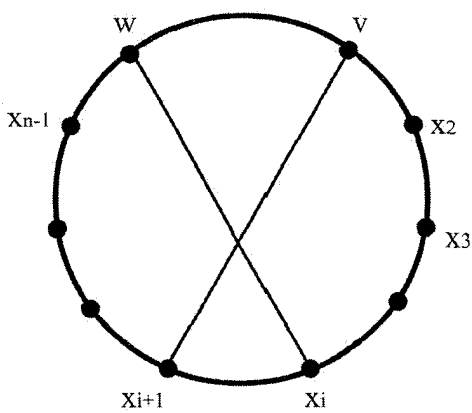
مسأله: گراف بدون‌جهت و همبند G=(V,E) با n رأس $(n \geq 3)$ داده شده است، به گونه‌ای که برای هر دو رأس غیرمجاور v و w داریم: $d(v)+d(w) \geq n$. در این گراف یک دور هامیلتونی بیابید.

الگوریتم حل مسأله بر پایه‌ی استقرایی معکوس روی تعداد یال‌های گراف است. در حالت پایه با یک گراف کامل سر و کار داریم. هر گراف کاملی که دست‌کم سه رأس داشته باشد، دوری (دورهایی) هامیلتونی دارد و یافتن یک دور هامیلتونی نیز در آن آسان است (کافی است رأس‌ها را به ترتیبی دل‌خواه در نظر بگیرید و آن‌ها را با یک دور به هم متصل کنید).

فرض استقرا: می‌دانیم چگونه در گراف‌هایی با دست‌کم m یال که شرایط گفته‌شده در آن‌ها برقرار است، یک دور هامیلتونی بیابیم.

باید در گرافی با m−۱ یال که شرایط مسأله را برآورده می‌کند، روش یافتن دوری هامیلتونی را بیابیم. G=(V,E) را چنین گرافی بگیرید. در این گراف، دو رأس غیرمجاور و دل‌خواه v و w را در نظر بگیرید و G' را همان G فرض کنید، با این تفاوت که در آن v و w به یکدیگر متصل هستند. بنا به فرض استقرا می‌دانیم چگونه دوری هامیلتونی در G' بیابیم. x_1، x_2، ...، x_n و x_1 را چنین دوری بگیرید (شکل ۷-۴۳ را ببینید). اگر این دور از یال (v,w) نگذرد، آنگاه G نیز باید این دور را در بر داشته باشد، پس کار به پایان می‌رسد؛ وگرنه بدون آن که به کلیت مسأله آسیب برسد، می توانیم فرض کنیم: $v=x_1$ و $w=x_n$. بنا به شرایط داده‌شده در G داریم: $d(v)+d(w)\geq n$. اینک، همه چیز برای یافتن یک دور هامیلتونی تازه آماده است.

همه‌ی یال‌هایی را که از v یا w خارج می‌شوند، در نظر بگیرید. (بنا به شرایط مسأله) تعداد این یال‌ها دست‌کم n است. از آنجا که G، n−۲ رأس دیگر نیز دارد، پس دو رأس همسایه‌ی x_i و x_{i+1} در دور وجود دارند، به گونه‌ای که v به x_{i+1} و w به x_i متصل است. حال، با یال‌های (v,x_{i+1}) و (w,x_i) یک دور هامیلتونی تازه به گونه‌ای می‌سازیم که یال (v,w) را در بر نگیرد. این دور عبارت است از: $v(=x_1)$، x_{i+1}، x_{i+2}، ...، $w(=x_n)$، x_i، x_{i-1}، ... و v (شکل ۷-۴۳ را ببینید).



شکل ۷-۴۳ یافتن یک دور هامیلتونی در G به یاری G'

پیاده‌سازی: راه سرراست برای پیاده‌سازی اثبات، آغاز کار با گرافی کامل است. سپس باید یال‌ها را یکی‌یکی جای‌گزین کنیم. می‌توانیم کار را به روشی که گفته خواهد شد، با گرافی بسیار کوچک‌تر آغاز

کنیم. گراف ورودی G را می‌گیریم و در آن (مثلاً با DFS) یک مسیر بلند می‌یابیم. سپس یال‌های لازم از E' (مکمل E؛ یعنی یال‌هایی که متعلق به G نیستند) را به آن می‌افزاییم تا این مسیر به یک دور هامیلونی تبدیل شود. حال، به گراف بزرگ‌تر G' رسیده‌ایم که یک دور هامیلونی هم دارد. معمولاً با این روش، تعداد اندکی یال به گراف افزوده می‌شود؛ به هر حال، در بدترین حالت، حداکثر n-۱ یال به گراف افزوده خواهد شد. می‌توانیم کار را با G' آغاز کنیم و فرایند گفته‌شده را بارها به کار گیریم تا آن در G مسیری هامیلتونی پیدا شود. تعداد کل گام‌های لازم برای جای‌گزینی یک یال و تعداد یال‌هایی که باید جای‌گزین شوند، هر دو از O(n) هستند؛ پس زمان اجرای الگوریتم از $O(n^2)$ خواهد بود.

۷-۱۳ خلاصه

گراف‌ها برای مدل کردن رابطه‌ی بین زوج‌هایی از اشیاء به کار می‌روند. از آنجا که در بیش‌تر الگوریتم‌ها لازم است کل ورودی بررسی گردد، نخستین موضوع الگوریتم‌های گراف، چند روش برای پیمایش آن‌هاست. دو گونه از پیمایش گراف را بررسی کردیم: جست‌وجوی نخست-ژرفا (DFS) و جست‌وجوی نخست-پهنا (BFS). چندین نمونه از مسأله‌های گراف را دیدیم که در آن‌ها DFS مناسب‌تر از BFS بود. بنابراین پیش‌نهاد می‌کنیم نخست DFS را بیازمایید (هرچند، مسأله‌هایی هم هستند که BFS برای آن‌ها بهتر است). DFS برای الگوریتم‌های بازگشتی گراف بسیار مناسب است. BFS معمولاً به فضای بیش‌تری نیاز دارد (هرچند، در حالت کلی چنین نیست و به نوع گراف بستگی دارد). مثالی از جست‌وجوی اولویت‌دار را هم دیدیم که برای محاسبه‌ی کوتاه‌ترین مسیر از یک رأس به رأس‌های دیگر گراف به کار برده می‌شد. جست‌وجوی اولویت‌دار پرهزینه‌تر از جست‌وجوی معمولی است. این روش جست‌وجو در بهینه‌سازی مسأله‌های مربوط به گراف‌های وزن‌دار به کار می‌آید.

معمولاً وجود دور در گراف، الگوریتم‌ها را با مشکلات عمده‌ای روبه‌رو می‌کند. به همین دلیل است که معمولاً طراحی الگوریتم در گراف‌های جهت‌دار بدون‌دور و درخت‌ها آسان‌تر و اجرای آن‌ها سریع‌تر است. لازم است دقت کنیم که حتا شاید گراف‌هایی دورهای بسیار گوناگونی داشته باشند که تعداد یال‌هایشان اندک است (تمرین ۷-۵۴). الگوریتم‌هایی که باید همه یا بخش عمده‌ی دورهای گراف را بررسی کنند، ممکن است برای بیش‌تر گراف‌ها بسیار کند باشند.

تجزیه‌ی گراف بسیار سودمند و خوش‌بختانه هزینه‌اش هم نسبتاً معقول است. تجزیه‌ی گراف به مؤلفه‌های همبند، دوهمبند و قویاً همبند را دیدیم. اساساً تجزیه‌ی گراف به ما اجازه می‌دهد فرض کنیم برخی ویژگی‌ها (مانند همبند بودن) در مؤلفه‌ها وجود دارند؛ هرچند که گراف مورد نظر، آن ویژگی‌ها را نداشته باشد.

کاهش، ترفند سودمند دیگری برای الگوریتم‌های گراف است. می‌توان گراف را با ماتریس نشان داد، پس رابطه‌ای طبیعی بین الگوریتم‌های گراف و ماتریس وجود دارد. بحث کلی درباره‌ی این رابطه و

کاهش‌های مربوط به آن در فصل ۱۰ خواهد آمد. مسأله‌های شبکه‌ی شاره و تطابق نمونه‌هایی عالی از کاهش هستند. کاهش‌ها به ما کمک می‌کنند تا تشخیص دهیم که آیا یک مسأله، دشوار است یا نه. در فصل ۱۱ دسته‌ای از مسأله‌ها به نام NP–تمام را بررسی خواهیم کرد که احتمالاً الگوریتمی برای حل آن‌ها وجود ندارد که در بدترین حالت، زمان اجرایش نسبت به اندازه‌ی ورودی، چندجمله‌ای باشد. شمار فراوانی از مسأله‌های گراف در این دسته جای می‌گیرند. گاه به نظر می‌رسد مسأله‌های آسان و دشوار تفاوت اندکی با یکدیگر دارند. برای نمونه، الگوریتمی کارآمد دیدیم که تشخیص می‌داد آیا یک گراف جهت‌دار، دوری ساده با طول فرد دارد یا نه. همین مسأله، تنها با افزودن این شرط که آن دور باید دربردارنده‌ی رأس (یا یال) مشخصی باشد، NP–تمام خواهد بود. لازم است به درکی شهودی از این تفاوت‌ها برسیم. پس مفاهیم فصل ۱۱ برای درک الگوریتم‌های گراف بسیار مهم هستند.

مراجعی برای مطالعه‌ی بیش‌تر

نظریه‌ی گراف، زمینه‌ای نسبتاً تازه در ریاضیات است. بیش‌تر نتایج پایه‌ای در این زمینه در قرن بیستم به دست آمده‌اند. با این حال، امروزه نظریه‌ی گراف زمینه‌ای توسعه‌یافته و بررسی‌شده همراه با هزاران نتیجه است. کتاب‌های بسیاری در زمینه‌ی گراف منتشر شده‌اند که برخی از آن‌ها عبارتند از: Berge [۱۹۶۲]، Ore [۱۹۶۳]، Harray [۱۹۶۹]، Berge [۱۹۷۳]، Deo [۱۹۷۴] Bondy و Murty [۱۹۷۶]، Chartrand [۱۹۷۷]، Capobianco و Mulluzzo [۱۹۷۸]، Bollobás [۱۹۷۹]، Tutte [۱۹۸۴] و Chartrand و Lesniak [۱۹۸۶]. چند کتاب نیز الگوریتم‌های گراف را بررسی کرده‌اند که برخی از آن‌ها عبارتند از: Even [۱۹۷۹]، Golumbie [۱۹۸۰] (که بر گراف‌های کامل و ارتباط انواع گراف‌ها تأکید دارد)، Gondran و Minoux [۱۹۸۴] (که تکیه‌اش بیش‌تر بر مسائل بهینه‌سازی است)، Gibbons [۱۹۸۵]، Nishizeki و Chiba [۱۹۸۸] (که ویژه‌ی گراف‌های مسطح است) و بررسی جامعی از van Leeuwen [۱۹۸۶].

ایده‌ی گراف‌های اویلری از Euler [۱۷۳۶] است و نخستین نتیجه در نظریه‌ی گراف به شمار می‌رود. به آسانی می‌توان از روی اثبات وجود مسیرهای اویلری، الگوریتمی برای یافتن آن‌ها به دست آورد (برای نمونه Even [۱۹۷۹] یا Ebert [۱۹۸۸] را ببینید). نخستین بار Lucas [۱۸۸۲] (شرح از Trémaux) و Tarry [۱۸۹۵] جست‌وجوی نخست–ژرفا را برای طراحی الگوریتم‌های پیمایش هزارتو توصیف کرده‌اند. اهمیت این جست‌وجو با کار Tarjan [۱۹۷۲] آشکار شد. او الگوریتم‌هایی برای مؤلفه‌های دوهمبند و قویاً همبند نیز ارائه کرد.

مسأله‌ی درخت پوشای کمینه را به صورت گسترده بررسی کرده‌اند. الگوریتمی که در بخش ۷–۶ برای حل این مسأله (بدون پیاده‌سازی) ارائه شد، از Prim [۱۹۷۵] است. Kruskal [۱۹۵۶] الگوریتم دیگری برای حل این مسأله ارائه کرده است (تمرین ۷–۵۹). افرادی که نامشان می‌آید، الگوریتم‌های

دیگری برای یافتن درخت پوشای کمینه طراحی کرده‌اند: Yao [۱۹۷۵]، Cheriton و Tarjan [۱۹۷۶]، Fredman و Tarjan [۱۹۸۷] و Gabow، Galil، Spencer و Tarjan [۱۹۸۶].

Dijkstra [۱۹۵۹] الگوریتم کوتاه‌ترین مسیر از یک رأس به رأس‌های دیگر را (ارائه‌شده در بخش ۷-۵) ساخته و Johnson [۱۹۷۷] آن را با یک هرم پیاده‌سازی کرده است (Tarjan [۱۹۸۳] را هم ببینید). اگر گراف تنک (خلوت) باشد (در عمل هم معمولاً چنین است) این الگوریتم سریع خواهد بود. اگر تعداد یال‌ها متناسب با $|V|^2$ باشد، زمان اجرای الگوریتم از $O(|V|^2 \log|V|)$ خواهد شد. موضوع تمرین ۷-۴۳ پیاده‌سازی بهتر این الگوریتم برای گراف‌های چگال با زمان اجرایی از $O(|V|^2)$ است. نتیجه‌ای از Fredman و Tarjan [۱۹۸۷] نشان می‌دهد که بهترین زمان اجرای مجانبی برای این مسأله (که از ساختارهای داده‌ای کاملاً پیچیده‌ای یاری می‌گیرد) از $O(|E|+|V|\log|V|)$ است. Floyd [۱۹۶۲] الگوریتم کوتاه‌ترین مسیر بین هر دو رأس را (ارائه شده در بخش ۷-۷) طراحی کرده است. این الگوریتم برای گراف‌های وزن‌داری که وزن‌های منفی هم داشته باشند، به درستی کار می‌کند؛ مشروط به این که هیچ دوری با وزن منفی در گراف وجود نداشته باشد (تمرین ۷-۷۳). در حالت میانگین می‌توان کوتاه‌ترین مسیرها را سریع‌تر هم یافت؛ Spira [۱۹۷۳] الگوریتمی دارد که زمان اجرای آن در حالت میانگین از $O(|V|^2 \log^2|V|)$ است. Moffat و Takaoka [۱۹۸۷] با ترکیبی از الگوریتم Spira و الگوریتمی قدیمی‌تر از Dantzig [۱۹۶۰]، الگوریتمی به وجود آورده‌اند که میانگین زمان اجرای آن از $O(|V|^2 \log|V|)$ است. برای یافتن اطلاعات بیش‌تری درباره‌ی الگوریتم‌های کوتاه‌ترین مسیر بررسی جامع Deo و Pang [۱۹۸۴] را ببینید (که در کنار مطالب دیگر، ۲۲۲ مرجع را هم در بر می‌گیرد). Warshall [۱۹۶۲] الگوریتم بسط ترایا را (ارائه شده در بخش ۷-۸) طراحی کرده است.

Ford و Fulkerson [۱۹۵۶] قضیه‌ی مسیر افزایشی و کاربرد آن را در شارهای شبکه کشف کردند. توصیفی بسیار خوب از ساختمان‌های داده‌ای و الگوریتم‌های ترکیبیاتی برای شارهای شبکه در Tarjan [۱۹۸۳] آمده است. Goldberg و Tarjan [۱۹۸۸] الگوریتم تازه‌تری برای شبکه‌ی شاره طراحی کرده‌اند. اطلاعات بیش‌تری درباره‌ی مسأله‌ی شبکه‌ی شاره و تعمیم‌های آن در این مراجع وجود دارد: Ford و Fulkerson [۱۹۶۲]، Christofides [۱۹۷۵]، Lawler [۱۹۷۶]، Minieka [۱۹۷۸]، Papadimitriou و Steiglitz [۱۹۸۲] و Gondran و Minoux [۱۹۸۴]. مبانی یا پایه‌های ریاضی نظریه‌ی تطابق و الگوریتم‌هایی برای مسأله‌های گوناگون آن، در کتابی از Lovász و Plummer [۱۹۸۶] وجود دارد. Galil [۱۹۸۶] بررسی جامعی درباره‌ی الگوریتم‌های تطابق، هم برای حالت کلی گراف‌ها و هم برای گراف‌های دوبخشی انجام داده است. الگوریتم بخش ۷-۱۲-۲ برای یافتن دورهای هامیلتونی در گراف‌های بسیار چگال بر پایه‌ی یک قضیه (و اثبات آن) از Ore [۱۹۶۰] است.

در این فصل دو موضوع مهم نظریه‌ی گراف را بررسی نکردیم: مسطح بودن و یک‌ریختی گراف. مسأله‌ی تشخیص گراف‌های مسطح و رسم آن‌ها در صفحه از قدیمی‌ترین مسأله‌ها در نظریه گراف است. نخستین الگوریتم‌های حل این مسأله از این افراد است: Auslander و Parter [۱۹۶۱]، Lempel، Even و Cederbaum [۱۹۶۶]. Hopcroft و Tarjan [۱۹۷۴] الگوریتمی برای تشخیص مسطح بودن گراف ارائه کرده‌اند که زمان اجرای آن خطی است. این الگوریتم که از روالی با زمان خطی (برپایه‌ی DFS) برای تجزیه‌ی گراف به مؤلفه‌های ۳-همبند سود می‌جوید (Hopcroft و Tarjan [۱۹۷۳]) مبنای ساخت ساختمان‌های داده‌ای و الگوریتم‌های فراوان دیگری شده است. تا زمان نگارش کتاب، هنوز الگوریتمی با زمان اجرای چندجمله‌ای برای تشخیص یک‌ریختی گراف‌ها پیدا نشده بود. یک‌ریختی گراف‌ها یکی از مسأله‌های انگشت‌شمار مهمی است که هنوز وضعیت مشخصی ندارد و روشن نیست که آیا این مسأله NP-سخت است یا این که راه‌حلی چندجمله‌ای دارد. (در این مورد، در فصل ۱۱ بیش‌تر بحث خواهیم کرد.) می‌توانید بررسی این موضوع را در Hoffman [۱۹۸۲] یا Lukas [۱۹۸۲] هم ببینید.

بحث و بررسی درباره‌ی دنباله‌های de Bruijn (تمرین ۷-۲۸) در Even [۱۹۷۹] یافت می‌شود. تمرین ۷-۴۶ از Sedgewick و Vitter [۱۹۸۶] است. ایده‌ی تمرین ۷-۵۵، مسأله‌ای از Bollobás [۱۹۸۶] و ایده‌ی تمرین ۷-۵۸ هم، مسأله‌ای از Lovász [۱۹۷۹] است. می‌توانید الگوریتمی را که برای حل تمرین ۷-۵۷ لازم است، در Ford [۱۹۵۶] بیابید. راه‌نمایی تمرین ۷-۸۱ برای بسط ترایا به الگوریتمی از Warren [۱۹۷۵] می‌انجامد. تمرین ۷-۹۷ از Lovász و Plummer [۱۹۸۶] است. Gabow و Tarjan [۱۹۸۸] برای مسأله‌ی گلوگاهِ تمرین ۷- ۱۰۰ الگوریتمی کارآمد ارائه کرده‌اند. نام قضیه تمرین ۷-۱۰۱ و ۷-۱۰۲ قضیه‌ی Gomory است. تمرین ۷-۱۰۵ از Lovász [۱۹۷۹] است. تمرین ۷-۱۲۱ به مسأله‌ای برای طراحی جدول‌های مسیریابی برمی‌گردد که بیش از حد حافظه به کار نبرد. این مسأله در Manber و McVoy [۱۹۸۸] حل شده است.

تمرین‌های آموزشی

۷-۱ مسأله‌ی یافتن عامل‌های توازن در درخت‌های دودویی را (که در بخش ۵-۸ بررسی شد) در نظر بگیرید و آن را با به‌کارگیری DFS حل کنید؛ یعنی اعمال پیش‌ترتیب و پس‌ترتیب لازم را تعریف کنید.

۷-۲ G=(V,E) را گرافی بدون‌جهت و همبند بگیرید و فرض کنید درخت DFS برای آن، T با ریشه‌ی v باشد.

الف- H را یک زیرگراف القایی دل‌خواه از G بگیرید و نشان دهید اشتراک T و H لزوماً درختی پوشا در H نیست.

ب- R را زیردرختی از H بگیرید و فرض کنید S زیرگرافی از G، القاشده با رأس‌های R باشد. ثابت کنید R می‌تواند یک درخت DFS از S باشد.

۳-۷ گراف بدون‌جهت و همبند G=(V,E)، درخت پوشای T در G و رأس v داده شده‌اند. الگوریتمی طراحی کنید که مشخص کند آیا T یک درخت معتبر DFS از G با ریشه‌ی v هست یا نه. به عبارت دیگر، این الگوریتم باید مشخص کند که آیا T می‌تواند خروجی یک DFS باشد که از v آغاز شده و همه‌ی یال‌ها را (با ترتیبی ممکن) پیموده باشد. زمان اجرای الگوریتم باید از $O(|E|+|V|)$ باشد.

۴-۷ ویژگی همه‌ی گراف‌های بدون‌جهتی را مشخص کنید که رأسی مانند v دارند، به گونه‌ای که هم درخت DFS پوشایی با ریشه‌ی v و هم درخت BFS پوشایی با ریشه‌ی v در آن‌ها وجود داشته باشد و این دو درخت با هم برابر گردند. (دو درخت پوشا با یکدیگر برابرند، اگر مجموعه‌ی یال‌های آن‌ها یکسان باشد؛ ترتیب پیمایش یال‌ها در اینجا بی‌اهمیت است، اما ریشه‌ی هر دو درخت باید رأسی یکسان (v) باشد.)

۵-۷ الگوریتم Topological_Sorting (شکل ۷-۱۴) را با این فرض که دیگر نمی‌دانید گراف، بدون‌دور است یا نه، تغییر دهید. روشن است که اگر گراف دور داشته باشد، نمی‌توان در آن هیچ ترتیب توپولوژیکی یافت. الگوریتمی طراحی کنید که برحسب آن که گراف بدون‌دور یا دارای دور باشد، برچسب‌های ترتیب توپولوژیک یا برچسب‌های یک دور گراف را در خروجی قرار دهد.

۶-۷ الگوریتم Single_Source_Shortest_Paths (شکل ۷-۱۷) را در نظر بگیرید. ثابت کنید زیرگراف دربردارنده‌ی همه‌ی یال‌های متعلق به «کوتاه‌ترین مسیرها از v به رأس‌های دیگر» که هنگام اجرای این الگوریتم به دست می‌آید، درختی با ریشه‌ی v است.

۷-۷ G=(V,E) را گرافی بدون‌جهت و وزن‌دار بگیرید و فرض کنید T درخت کوتاه‌ترین مسیرها با ریشه‌ی v باشد (تمرین ۷-۶). اگر وزن هر یال G به اندازه‌ی ثابت c افزایش یابد، آیا هنوز هم T، درخت کوتاه‌ترین مسیرها از v است؟

۸-۷ یا این موضوع را ثابت کنید یا مثالی نقض ارائه دهید: الگوریتم Single_Source_Shortest_Paths (شکل ۷-۱۷) برای گراف وزن‌داری هم که وزن برخی یال‌های آن منفی است، اما هیچ دوری با وزن منفی ندارد؛ درست کار می‌کند.

۹-۷ G=(V,E) را گرافی وزن‌دار و بدون‌جهت بگیرید. ثابت کنید اگر وزن یال‌ها متمایز باشد، آنگاه درخت پوشای کمینه یکتاست.

۱۰-۷ الگوریتم MCST (شکل ۷-۲۰) را به گونه‌ای تغییر دهید که یک درخت پوشای بیشینه بیابد.

۱۱-۷ یا ثابت کنید الگوریتم MCST (شکل ۷-۲۰) برای گراف‌های وزن‌داری که یال‌هایی با وزن منفی هم داشته باشند، درست کار می‌کند و یا برای این ادعا یک مثال نقض بیاورید.

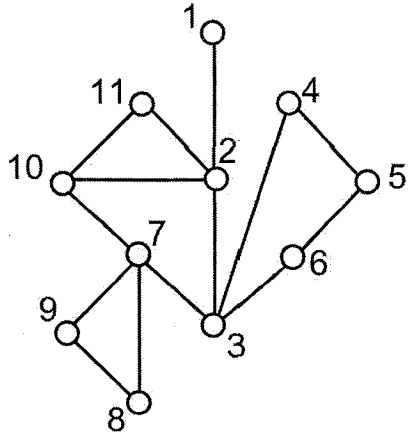
۱۲-۷ الف- یک گراف بدون‌جهت، وزن‌دار و همبند و یک رأس مانند v از آن، چنان ارائه دهید که درخت پوشای کمینه‌ی گراف، همان درخت کوتاه‌ترین مسیرها با ریشه‌ی v باشد.

ب- یک گراف بدون‌جهت، وزن‌دار و همبند و یک رأس مانند v از آن، چنان ارائه دهید که درخت پوشای کمینه‌ی گراف از «درخت کوتاه‌ترین مسیرها با ریشه‌ی v» بسیار متفاوت باشد. آیا امکان دارد که این دو درخت کاملاً جداازهم باشند؟

۱۳-۷ اثر حذف رأس c از گراف شکل ۷-۲۵ را در مؤلفه‌های دوهمبند و درخت دوهمبندی حاصل از آن شرح دهید.

۱۴-۷ الف- الگوریتم مؤلفه‌های دوهمبند را روی گراف شکل ۷-۴۴ اجرا کنید. این الگوریتم باید از شماره‌های DFS، به همان ترتیب داده‌شده در شکل پیروی کند. در هر گام الگوریتم، مقادیر High را محاسبه کنید.

ب- اگر یال (4,8) را به این گراف بیفزاییم، در اجرای الگوریتم چه تغییراتی به وجود می‌آید؟

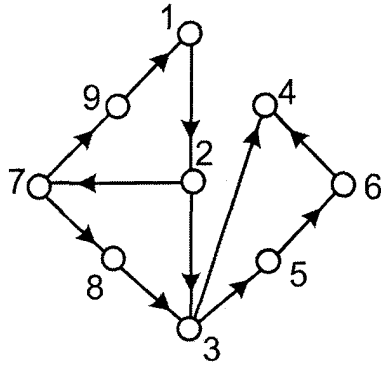


شکل ۷-۴۴ گرافی بدون‌جهت همراه با شماره‌های DFS برای رأس‌های آن (تمرین ۷-۱۴)

۱۵-۷ ثابت کنید تعریف درخت دوهمبندی در بخش ۷-۹-۱ واقعاً نشان‌دهنده‌ی یک درخت است؛ یعنی نشان دهید که همه‌ی مؤلفه‌های دوهمبند به یکدیگر متصل هستند و دوری هم وجود ندارد.

۱۶-۷ الف- الگوریتم مؤلفه‌های قویاً همبند را روی گراف شکل ۷-۴۵ اجرا کنید. این الگوریتم باید از شماره‌های DFS، به همان ترتیب داده‌شده در شکل پیروی کند. در هر گام الگوریتم، مقادیر High را محاسبه کنید.

ب- اگر یال (4,1) را به گراف بیفزاییم، در اجرای الگوریتم چه تغییراتی به وجود می‌آید؟



شکل ۷-۴۵ یک گراف جهت‌دار با شماره‌های DFS برای رأس‌های آن (تمرین ۷-۱۶)

۷-۱۷ G=(V,E) را گرافی قویاً همبند بگیرید و فرض کنید T یک درخت DFS از G باشد. ثابت کنید اگر در G همه‌ی یال‌های جلورو - با توجه به T - حذف شوند، گراف حاصل بازهم قویاً همبند است.

۷-۱۸ الف- ثابت کنید الگوریتم یافتن دوری با طول فرد در گراف جهت‌دار (بخش ۷-۹-۴) درست کار می‌کند.

ب- گرافی ارائه دهید که قویاً همبند نباشد و الگوریتم بند «الف» روی آن درست کار کند.

۷-۱۹ الگوریتم بررسی‌شده در بخش ۷-۱۰-۲ را پیاده‌سازی کنید. این الگوریتم در گرافی با ۲n رأس، که درجه‌ی هر رأس دست‌کم n باشد، یک تطابق کامل می‌یابد. زمان اجرای الگوریتمی که طراحی می‌کنید باید در بدترین حالت از $O(|V|+|E|)$ باشد.

۷-۲۰ این تمرین، اثبات وجود تطابق کامل در گراف‌های چگال را قدری تعمیم می‌دهد. فرض کنید گرافی با ۲n رأس به شما داده‌اند که درجه‌ی همه‌ی رأس‌های آن زیاد نیست، اما مجموع درجه‌های هر دو رأس نامجاور، دست‌کم ۲n است. آیا بازهم همواره تطبیقی کامل وجود دارد؟ آیا هنوز هم الگوریتم به‌دست‌آمده در تمرین ۷-۱۹ درست است؟

تمرین‌های خلاقانه

فرض کنید همه‌ی گراف‌ها به صورت لیست‌های همسایگی داده شده باشند؛ مگر آن که به صراحت خلاف آن را گفته باشیم. چنین روش ذخیره‌ای به فضایی از $O(|V|+|E|)$ نیاز دارد؛ از این رو، اگر زمان اجرای الگوریتمی از $O(|V|+|E|)$ باشد، می‌گوییم زمان اجرای آن خطی است. منظور از زمان اجرای یک الگوریتم، زمان اجرای آن در بدترین حالت است؛ مگر آن که به صراحت خلاف آن را بگوییم. در برخی موارد، زمان اجرای مشخصی داده شده است و برای حل تمرین باید به آن زمان دست

یافت؛ در موارد دیگر، تنها باید «الگوریتمی کارآمد» برای حل مسأله پیدا کنید. در این موارد، حتا اگر بهترین الگوریتم ممکن را پیدا نکردید، بکوشید تا جایی که می‌توانید به الگوریتم بهتری دست پیدا کنید.

۷-۲۱ گراف بدون‌جهت G=(V,E) و عدد صحیح k داده شده‌اند. یا بزرگ‌ترین زیرگراف القایی H در G را چنان بیابید که درجه‌ی هر رأس H از k بزرگ‌تر باشد و یا مشخص کنید که چنین زیرگرافی وجود ندارد. (یک زیرگراف القایی در G=(V,E)، گرافی مانند H=(U,F) است، به گونه‌ای که $U \subseteq V$ و F دربرگیرنده‌ی تمام یال‌هایی از E است که هر دو سر آن یال در U باشد.) برای این مسأله (که در بخش ۵-۳ بررسی شد) الگوریتمی با زمان اجرای خطی بیابید.

۷-۲۲ G=(V,E) را گرافی بدون‌جهت و همبند بگیرید. می‌خواهیم رأسی با درجه‌ی ۱ را برگزینیم، آن رأس و یال‌های گذرنده از آن را حذف کنیم و این فرایند (یعنی برگزیدن رأسی دیگر با درجه‌ی ۱ در گراف باقی‌مانده و حذف آن) را تکرار کنیم، تا آن که همه‌ی یال‌های گراف حذف شوند. اگر چنین فرایندی برای گراف‌های مشخصی امکان‌پذیر باشد، آنگاه ممکن است طراحی الگوریتم در این گراف‌ها با استقرا آسان‌تر شود. ویژگی‌های گراف‌های بدون‌جهت و همبندی را بیابید که این شرط‌ها را برآورده کنند. به عبارت دیگر، شرط لازم و کافی برای گراف G را چنان بیابید که فرایند گفته‌شده امکان‌پذیر باشد.

۷-۲۳ الگوریتم گراف اویلری را که در بخش ۷-۲ بررسی شد، به صورتی کارآمد پیاده‌سازی کنید. زمان و فضای این الگوریتم باید خطی باشد.

۷-۲۴ G=(V,E) را گرافی بدون‌جهت بگیرید که درجه‌ی همه‌ی رأس‌های آن زوج است. الگوریتمی با زمان اجرای خطی طراحی کنید که یال‌های G را به گونه‌ای جهت‌دار کند که در هر رأس، درجه‌ی ورودی با درجه‌ی خروجی برابر باشد.

۷-۲۵ یک مدار جهت‌دار اویلری، مداری جهت‌دار است که از هر یال دقیقاً یک بار بگذرد. ثابت کنید هر گراف جهت‌دار، مدار جهت‌داری اویلری دارد، اگر و تنها اگر در هر رأس درجه‌ی ورودی با درجه‌ی خروجی، برابر و گراف بدون‌جهت متناظر با گراف اصلی، همبند باشد. الگوریتمی کارآمد طراحی کنید که یکی از این گونه مدارها را (در صورت وجود) بیابد.

۷-۲۶ G=(V,E) را گرافی بدون‌جهت و همبند بگیرید که تعداد رأس‌های با درجه‌ی فرد آن k است.

الف- ثابت کنید k زوج است.

ب- الگوریتمی برای پیدا کردن k/2 مسیر باز طراحی کنید، به گونه‌ای که هر یال G دقیقاً در یکی از این مسیرها باشد.

۷-۲۷ الگوریتمی طراحی کنید که در گرافی همبند و بدون‌جهت رأسی را بیابد که حذف آن، گراف را ناهمبند نکند. زمان اجرای الگوریتم باید خطی باشد. (از الگوریتم مؤلفه‌های دوهمبند کمک نگیرید.) به عنوان یک نتیجه ثابت کنید هر گراف همبند چنین رأسی دارد.

۲۸-۷ یک دنباله‌ی دودویی de Bruijn، دنباله‌ای (دوردار) از 2^n بیت به صورت $a_{2^n}...a_2a_1$ است، به گونه‌ای که هر رشته‌ی دودویی ممکن به طول n مانند S در جایی از این دنباله ظاهر شده باشد؛ یعنی یک اندیس یکتای i وجود دارد به گونه‌ای که $S = a_ia_{i+1}...a_{i+n}$ (از اندیس‌ها باید تنها باقی‌مانده‌ی تقسیم آن‌ها بر 2^n را در نظر بگیریم). برای مثال، دنباله‌ی ۱۰۰۰۱۰۱۱ یک دنباله‌ی دودویی de Bruijn به ازای n=۳ است. $G_n = (V,E)$ را گرافی جهت‌دار بگیرید که این‌گونه تعریف شده است: مجموعه‌ی رأس‌های V متناظر با مجموعه‌ی تمام رشته‌های دودویی به طول n-۱ است ($|V| = 2^{n-1}$). رأس متناظر با رشته‌ی $a_{n-1}...a_2a_1$ یالی دارد که به رأس متناظر با رشته‌ی $b_{n-1}...b_2b_1$ می‌رسد، اگر و تنها اگر $a_{n-1}...a_3a_2$ با $b_{n-2}...b_2b_1$ برابر باشد. ثابت کنید G_n یک گراف جهت‌دار اویلری است و ارتباط آن با دنباله‌های de Bruijn را بیابید.

۲۹-۷ n عدد صحیح مثبت d_1، d_2، ... و d_n داده شده‌اند، به گونه‌ای که $d_1 + d_2 + ... + d_n = 2n - 2$. می‌خواهیم درختی با n رأس بسازیم که درجه‌ی رأس‌های آن دقیقاً d_1، d_2، ... و d_n باشد. برای حل این مسأله الگوریتمی کارآمد بسازید.

۳۰-۷ (i_1,o_1)، (i_2,o_2)، ... و (i_n,o_n) را دنباله‌ی از زوج‌اعداد صحیح بگیرید که:

الف- $i_1 = 0$ و برای هر k از بازه‌ی [1,n] داشته باشیم: $i_k = 1$.

ب- $\sum_{j=1}^{n} o_j = n - 1$.

درختی ریشه‌دار با n رأس بسازید که در آن درخت، درجه‌ی ورودی رأس k، i_k و درجه‌ی خروجی این رأس، o_k باشد. این الگوریتم باید در زمانی از O(n) اجرا شود.

۳۱-۷ G=(V,E) را گرافی جهت‌دار فرض کنید که می‌تواند دور هم داشته باشد. w را سلف v گوییم، اگر $(w,v) \in E$. الگوریتمی بیابید که یا رأس‌های G را با اعداد متمایز ۱ تا $|V|$ برچسب بزند، به گونه‌ای که برچسب هر رأس دست‌کم از برچسب یکی از سلف‌های آن (در صورت وجود چنین رأسی) بزرگ‌تر باشد و یا مشخص کنید برچسب‌گذاری G با این روش ممکن نیست.

۳۲-۷ گراف بدون‌جهت G=(V,E) رنگ‌شدنی با k رنگ گفته می‌شود، اگر هنگام داشتن k رنگ گوناگون بتوان همه‌ی رأس‌های G را چنان رنگ‌آمیزی کرد که هیچ دو رأس همسایه‌ای در G، هم‌رنگ نباشند. الگوریتمی با زمان خطی ارائه دهید که یا گراف ورودی را با دو رنگ رنگ بزند و یا مشخص کند که دو رنگ برای رنگ‌آمیزی گراف کافی نیست.

۳۳-۷ G=(V,E) را گرافی بدون‌جهت بگیرید که می‌توان آن را با دو رنگ رنگ‌آمیزی کرد. ممکن است با چندین روش گوناگون بتوان این کار را انجام داد. به یاری الگوریتم تمرین ۷-۳۲ ثابت

کنید رنگ‌آمیزی G یکتاست (البته به جز جابه‌جا کردن دو رنگ با یکدیگر که همواره شدنی است) اگر و تنها اگر G همبند باشد.

۳۴-۷ T را درختی بدون‌جهت با ریشه‌ی r بگیرید (که لزوماً دودویی نیست). این درخت با لیست‌های همسایگی ذخیره شده است و هر رأس آن متناظر با یک کاراکتر از الفبایی متناهی و ثابت است. P را یک رشته‌ی الگو بگیرید (که به صورت آرایه‌ای از کاراکترها داده شده است). الگوریتمی طراحی کنید که روشن کند آیا این الگو، دست‌کم یک بار در مسیری از ریشه به یک برگ درخت ظاهر شده است یا نه. در بدترین حالت، این الگوریتم باید در زمانی از O(n+m) اجرا شود که در آن n، تعداد رأس‌های درخت و m، اندازه‌ی الگوست.

۳۵-۷ گراف همبند و بدون‌جهت G=(V,E) که دقیقاً یک دور دارد، داده شده است. جهت یال‌ها را به گونه‌ای تنظیم کنید که درجه‌ی ورودی هر رأس حداکثر ۱ باشد. (چنین گراف‌های جهت‌داری یک‌به‌یک نامیده می‌شوند، چراکه با توابع یک‌به‌یک متناظر هستند.) پیچیدگی این الگوریتم چقدر است؟

۳۶-۷ G=(V,E) را گرافی بدون‌جهت بگیرید. الگوریتمی طراحی کنید که مشخص کند آیا امکان دارد جهت یال‌های G را به گونه‌ای تنظیم کرد تا درجه‌ی ورودی هر رأس، دست‌کم ۱ شود. اگر چنین کاری امکان‌پذیر باشد، الگوریتم شما باید روش کار را نیز نشان دهد.

۳۷-۷ گراف همبند و بدون‌جهت G=(V,E) داده شده است. جهت یال‌های آن را چنان تنظیم کنید که این دو شرط برآورده شوند:

الف- گراف جهت‌دار حاصل، درختی ریشه‌دار را در بر گیرد (یعنی درختی که همه‌ی یال‌هایش از ریشه دور شوند).

ب- هر یال گراف که به این درخت تعلق نداشته باشد، با یال‌های درخت، دوری جهت‌دار تشکیل دهد.

پیچیدگی این الگوریتم را بیابید.

۳۸-۷ گراف جهت‌دار و بدون‌دور G=(V,E) داده شده است. یک مسیر (جهت‌دار) ساده در G بیابید که تعداد یال‌های آن در بین تمام مسیرهای ساده‌ی G بیشینه باشد. زمان اجرای این الگوریتم باید خطی باشد.

۳۹-۷ الف- مسأله‌ی تمرین ۷-۳۸ را برای گراف‌های وزن‌دار حل کنید؛ یعنی مسیری را بیابید که وزن آن در بین همه‌ی مسیرها بیشینه باشد.

ب- آیا در صورت وجودِ یال‌هایی با هزینه‌ی منفی، الگوریتم شما بازهم درست کار می‌کند؟

پ- آیا این الگوریتم در حالت کلی (هنگامی که ممکن است گراف ورودی، دور نیز داشته باشد) بازهم درست کار می‌کند؟

۴۰-۷ G=(V,E) را گرافی جهت‌دار و بدون‌دور در نظر بگیرید و فرض کنید k، تعداد یال‌های طولانی‌ترین مسیر در G باشد. الگوریتمی طراحی کنید که رأس‌های گراف را حداکثر به k+۱ دسته چنان تقسیم کند که برای هر دو رأس v و w از یک دسته‌ی یکسان، نه مسیری از v به w وجود داشته باشد و نه مسیری از w به v. این الگوریتم باید در زمانی خطی اجرا گردد.

۴۱-۷ G=(V,E) را گرافی جهت‌دار با این ویژگی در نظر بگیرید که از یک زیرگراف بدون‌دور مانند H، به همراه چند یال عقب‌رو تشکیل شده است و در هیچ یک از مسیرهای ساده‌ی G بیش از یکی از این یال‌های عقب‌رو وجود ندارد. اگر H دربرگیرنده‌ی همه‌ی رأس‌های G باشد، الگوریتمی با زمان خطی بنویسید که کوتاه‌ترین مسیر از یک رأس خاص به دیگر رأس‌ها را بیابد. (دقت کنید که خود H داده نشده است.)

۴۲-۷ G=(V,E) را گرافی جهت‌دار و v و w را دو رأس از آن بگیرید. الگوریتمی با زمان خطی طراحی کنید که تعداد کوتاه‌ترین مسیرهای متمایز (نه لزوماً با رأس‌های جداازهم) را بین v و w محاسبه کند. (یال‌ها بدون‌وزن هستند.)

۴۳-۷ الگوریتم Single_Source_Shortest_Paths (شکل ۷-۱۵) را به گونه‌ای پیاده‌سازی کنید که زمان اجرای آن در بدترین حالت از $O(|V|^2)$ (و مستقل از اندازه‌ی E) باشد.

۴۴-۷ اگر G=(V,E) گرافی وزن‌دار و جهت‌دار باشد، الگوریتمی برای یافتن کم‌وزن‌ترین دور در آن ارائه دهید. زمان اجرای الگوریتم باید از $O(|V|^3)$ باشد.

۴۵-۷ الگوریتم یافتن کوتاه‌ترین مسیر که در بخش ۷-۵ ارائه شد، در بین مسیرهای هم‌طول، یکی را به دل‌خواه برمی‌گزیند. روشی را برای تغییر این الگوریتم بیابید، به گونه‌ای که اگر چندین مسیر گوناگون با طول یکسان وجود داشته باشند، کم‌یال‌ترین مسیر را برگزیند. (در بین مسیرهای هم‌طولی که تعداد یال‌هایشان برابر است، یکی به دل‌خواه برگزیده می‌شود.) می‌توانید فضایی اضافی از $O(|E|)$ را نیز به کار ببرید.

۴۶-۷ یک گراف اقلیدسی، گرافی وزن‌دار و بدون‌جهت است که هر رأس آن متناظر با یک نقطه‌ی صفحه و وزن هر یال آن برابر با فاصله‌ی بین دو نقطه‌ی متناظر با دو سر آن یال است. برای یافتن کوتاه‌ترین مسیر بین دو رأس s و t از یک گراف اقلیدسی، این روش مکاشفه‌ای پیش‌نهاد شده است: از الگوریتم Dijkstra برای یافتن کوتاه‌ترین مسیرها از یک رأس یاری بگیرید، اما در هر تکرار رأسی مانند x را در بین رأس‌هایی که تاکنون انتخاب نشده‌اند، به گونه‌ای برگزینید که حاصل جمع dist(s,x) و Euclid_dist(x,t) کمینه شود. (dist بیانگر کوتاه‌ترین مسیر و Euclid_dist نشان‌دهنده‌ی مساحت اقلیدسی است و فرض بر این است که Euclid_dist را داریم.) هنگامی که رأس برگزیده‌شده t باشد، کوتاه‌ترین مسیر از s تا t به دست می‌آید.

الف- چگونه الگوریتم را پیاده‌سازی می‌کنید؟ تنها کافی است تفاوت‌های پیاده‌سازی Dijkstra با پیاده‌سازی خود را بیان کنید.

ب- چرا این شیوه برای حالت کلی گراف‌ها (گراف‌های نااقلیدسی) کار نمی‌کند.

پ- مثالی بیاورید که در آن، این روش، از الگوریتم Dijkstra بسیار (یعنی بیش از یک مقدار ثابت) سریع‌تر باشد. مثالی هم بیاورید که این الگوریتم سریع‌تر از الگوریتم Dijkstra نباشد. زمان اجرای بدترین حالت برحسب تعداد یال‌ها چقدر است؟

۴۷-۷ ورودی مسأله، گراف جهت‌دار G=(V,E) همراه با رأس v از آن است. به هر رأس w از G، هزینه‌ی مثبت c(w) نسبت داده شده است. هزینه‌ی مسیری جهت‌دار مانند v، x_1، x_2، ...، x_k و u را $\sum_{i=1}^{k} c(x_i)$ تعریف می‌کنیم؛ یعنی هزینه‌ی رأس آغاز و پایان را در نظر نمی‌گیریم. پس اگر $(v,u) \in E$ آنگاه هزینه‌ی رفتن از v به u صفر است. الگوریتمی کارآمد طراحی کنید که کم‌هزینه‌ترین مسیر از رأس v به هر رأس دیگر را بیابد.

۴۸-۷ G=(V,E) را گرافی وزن‌دار و جهت‌دار با وزن‌های مثبت فرض کنید. v و w را دو رأس از G و k را عددی صحیح بگیرید که $k \leq |V|$. الگوریتمی برای یافتن کوتاه‌ترین مسیری از v به w که دقیقاً k یال داشته باشد، طراحی کنید. (نیازی نیست که این مسیر ساده باشد.)

۴۹-۷ دسته‌ی بزرگی از مسأله‌ها که مسائل گلوگاه نامیده می‌شوند، چنین قالبی دارند: ورودی، گرافی وزن‌دار است و ما به ویژگی مشخصی از گراف (در این مورد، کوتاه‌ترین مسیرهای آن) علاقه‌مندیم. وزن گلوگاهِ یک زیرگراف را وزن سنگین‌ترین یال آن تعریف می‌کنیم که با تعریف عادی (یعنی مجموع وزن یال‌ها) متفاوت است. (سنگین‌ترین یال همان گلوگاه است.) در این مسأله منظور از هزینه‌ی یک مسیر، هزینه‌ی سنگین‌ترین یال آن است؛ یعنی کوتاه‌ترین مسیرها را از نظر گلوگاه بررسی می‌کنیم.

الف- الگوریتمی برای حل مسأله‌ی کوتاه‌ترین مسیر از یک رأس به دیگر رأس‌ها طراحی کنید که در آن، هزینه‌ی مسیر بنا به آنچه گفته شد، از روی گلوگاه محاسبه شود. آیا درخت کوتاه‌ترین مسیرهایی که در این الگوریتم به دست می‌آید، ویژگی بارزی دارد؟

ب- الگوریتمی برای حل مسأله‌ی کوتاه‌ترین مسیر از هر رأس به رأس‌های دیگر - با توجه به تعریف گفته‌شده برای هزینه‌ی مسیر - طراحی کنید.

۵۰-۷ G=(V,E) را گرافی وزن‌دار و جهت‌دار بگیرید که دور ندارد، اما وزن برخی از یال‌های آن احتمالاً منفی است. الگوریتمی با زمان خطی برای حل مسأله‌ی کوتاه‌ترین مسیر از یک رأس به همه‌ی رأس‌های دیگر در G بیابید.

۵۱-۷ اگر d(v) درجه رأس v را نشان دهد، الگوریتمی با زمان خطی طراحی کنید که همه‌ی لیست‌های همسایگی گراف جهت‌دار G=(V,E) را به ترتیب افزایشیِ درجه‌ی رأس‌ها مرتب

کند؛ یعنی اگر d(u)<d(v)، آنگاه در هر لیست همسایگی که هم u و هم v را در بر داشته باشد، یال‌هایی که به u می‌روند، پیش از یال‌های رونده به v بیایند. در صورت برابر شدن درجه‌ی دو رأس، می‌توانید آن‌ها را به هر ترتیب دل‌خواهی بیاورید. این الگوریتم می‌تواند از فضایی خطی نیز کمک بگیرد.

۷-۵۲ می‌خواهیم مجموعه‌ی یال‌های E از گراف بدون‌جهت G=(V,E) را به زیرمجموعه‌های جداازهم E_1، ... و E_k چنان افراز کنیم که هر E_i متناظر با یک دور ساده باشد. شرط(های) لازم و کافی برای انجام این کار چیست؟ الگوریتمی کارآمد بنویسید که گراف‌های دارنده‌ی این شرط(ها) را به صورتی که گفته شد، افراز کند.

۷-۵۳ گرافی همبند و بدون‌جهت به شما داده شده است که یال‌های آن وزن ندارند. دوری ساده با کم‌ترین طول در آن بیابید. (طول این دور را محیط گراف می‌گویند.)

۷-۵۴ ثابت کنید گراف‌های بدون‌جهتی با n رأس و O(n) یال وجود دارند که $2^{\Omega(n)}$ دور متمایز داشته باشند. (از این ادعا نتیجه می‌شود که حتا ممکن است تعداد دورهای یک گراف تنک هم نمایی باشد؛ پس الگوریتمی که به بررسی تمام دورهای گراف نیاز دارد، نمی‌تواند برای حالت کلی گراف‌ها کارآمد باشد.)

۷-۵۵ الگوریتمی برای حل این مسأله طراحی کنید:

ورودی: گراف جهت‌دار G= (V,E) با n+۱ رأس و n یال که گراف بدون‌جهت متناظر با آن یک درخت است. هر یال (u,w) با عدد صحیح یکتای $\lambda(u,w)$ برچسب خورده است و $1 \le \lambda(u,w) \le n$.

خروجی: تابع S از رأس‌های گراف به زیرمجموعه‌هایی از {1,2,...,n} به گونه‌ای که این دو شرط برقرار شود:

۱- اگر $(u,w) \in E$ آنگاه $S(w) = S(u) \cup \{\lambda(u,w)\}$.

۲- اگر $u \neq w$ آنگاه $S(u) \neq S(w)$.

(توجه: S(u) می‌تواند هر یک از زیرمجموعه‌های {1,2,...,n} باشد؛ مثلاً ϕ یا {1,2,...,n}.)

۷-۵۶ دوباره مسأله‌ی تمرین ۷-۵۵ را در نظر بگیرید. ثابت کنید نمی‌توان این مسأله را برای هیچ یک از گراف‌های دارای دور (نه لزوماً جهت‌دار) حل کرد و روش‌های گوناگون برچسب‌گذاری نیز کمکی به حل مسأله دراین حالت نخواهند کرد؛ یعنی باید ثابت کنید محدود کردن مسأله به درخت لازم است.

۷-۵۷ یک هسته در گراف جهت‌دار G=(V,E)، زیرمجموعه‌ای مانند V' از رأس‌های آن است ($V' \subseteq V$) که دو سر هیچ یالی متعلق به V' نباشد و به ازای هر $w \notin V'$، یالی مانند (v,w) وجود داشته باشد که $v \in V'$. ورودی مسأله، گراف جهت‌دار G=(V,E) است که

n+۱ رأس و n یال دارد، اما گراف بدون‌جهت متناظر با آن یک درخت است. الگوریتمی طراحی کنید که یا در G یک هسته بیابد و یا مشخص کند که G هسته ندارد.

۵۸-۷ G=(V,E) را گرافی جهت‌دار و f را تابعی تعریف‌شده روی تمام یال‌های آن بگیرید، به گونه‌ای که اگر مسیر e_1، ... و e_k مداری در G باشد، آنگاه: $\sum_{i=1}^{k} f(e_i) = 0$. تابع P را روی رأس‌های G چنان تعریف کنید که برای هر یال (v,w) داشته باشیم: f(v,w)=P(w)-P(v).

۵۹-۷ الگوریتم MCST را با رویکرد تازه‌ای پیاده‌سازی کنید: به جای نگه‌داری یک درخت و هر بار افزودن یک یال به آن، مجموعه‌ای از درخت‌های جداازهم را نگه‌داری کنید (که همگی بخش‌هایی از MCST هستند). این درخت‌ها را با هم در نظر بگیرید و هر بار یالی به یکی از آن‌ها اضافه کنید. در آغاز، همه‌ی رأس‌ها درخت‌هایی جداازهم با اندازه‌ی ۰ هستند. این الگوریتم در هر گام کم‌هزینه‌ترین یالی را می‌یابد که دو درخت جداازهم را به یکدیگر متصل می‌سازد. سپس با افزودن این یال، آن دو درخت را در یکدیگر ادغام می‌کند. باید ثابت کنید این رویکرد درست و مناسب است. پیچیدگی الگوریتم MCST با این رویکرد چقدر است؟ (راه‌نمایی: به‌کارگیری ساختمان داده‌ی union-find برای حل مسأله به شما کمک می‌کند.)

۶۰-۷ G=(V,E) را گرافی وزن‌دار و بدون‌جهت و F را زیرگرافی از آن بگیرید که جنگل باشد (یعنی در Fهیچ دوری وجود ندارد). الگوریتمی کارآمد طراحی کنید که کم‌هزینه‌ترین درخت پوشایی از G را بیابد که دربرگیرنده‌ی همه‌ی یال‌های F باشد.

۶۱-۷ G=(V,E) را گرافی وزن‌دار و بدون‌جهت بگیرید که همبند است و T را یک درخت پوشای کمینه در G فرض کنید. اگر هزینه‌ی یالی از G (مانند e) تغییرکند، در چه شرایطی T دیگر یک MCST نخواهد بود؟ الگوریتمی کارآمد طراحی کنید که یا روشن سازد که T هنوز هم یک MCST است و یا یک MCST تازه در G بیابد. (e ممکن است متعلق به T باشد یا نباشد.)

۶۲-۷ شبکه‌ای ارتباطی را در نظر بگیرید که می‌توان مدلی از آن با یک گراف همبند، وزن‌دار و بدون‌جهت ساخت. هر پایگاه در این شبکه، با یک رأس نشان داده می‌شود. هر خط ارتباطی، دوسویه و متناظر با یک یال وزن‌دار است. برچسب یال، ممکن است نشان‌دهنده‌ی تأخیر مورد انتظار خط و یا تعرفه‌ی استفاده از آن باشد. اطلاعات هر پایگاه این شبکه، محلی است؛ یعنی هر پایگاه، تنها یال‌ها (و در نتیجه رأس‌های) همسایه‌ی خود را می‌شناسد. می‌توان از یک MCST (درخت پوشای کمینه) برای پخش سراسری پیام به همه‌ی پایگاه‌ها سود جست؛ یعنی اگر برای پخش سراسری، یال‌های این MCST را به کار ببریم، هزینه‌ی کل کمینه می‌شود. فرض کنید هر پایگاه از پیش می‌داند کدام یک از یال‌های همسایه‌اش جزو MCST است. همچنین فرض کنید زیرمجموعه‌ی مشخصی از پایگاه‌ها مانند U ($U \subset V$) وجود دارد

که همه‌ی پایگاه‌های آن، اطلاعات یکدیگر از شبکه را در اختیار داشته باشند؛ یعنی هر پایگاه از U، هم یال‌ها و رأس‌های همسایه‌ی خود را و هم یال‌ها و رأس‌های همسایه‌ی همه‌ی پایگاه‌های دیگر U را بشناسد. در ضمن، فرض کنید زیرگراف القایی U در درخت پوشای کمینه‌ی گراف، همبند (یعنی درخت) باشد. حال اگر هزینه‌ی یالی مانند e از MCST $(e \in U)$ تغییر کند:

الف- با چه شرایطی می‌توان مطمئن شد که این تغییر هزینه اثری روی MCST ندارد؟ پاسخی تنها برحسب اطلاعات درون پایگاه‌های U بدهید. به عبارت دیگر، پایگاه‌های درون U چگونه می‌توانند تشخیص دهند که پس از تغییر هزینه‌ی یک یال آیا MCST هم باید تغییر کند یا نه؟

ب- در چه شرایطی تفاوت MCST تغییریافته با MCST اصلی، تنها در یال‌های متعلق به زیرگراف القایی با U است؟ پاسخی تنها برحسب اطلاعات درون پایگاه‌های U بدهید.

پ- الگوریتمی برای بررسی شرایط بندهای «الف» و «ب» (بازهم تنها با اطلاعات پایگاه‌های درون U) ارائه و تغییر لازم در MCST را به طور خلاصه شرح دهید. (پاسخ دادن الگوریتم هنگامی که تغییر MCST درون U رخ می‌دهد، کافی است.)

۶۳-۷ مسأله‌ی پخش سراسری پیام در شبکه را در نظر بگیرید، اما این بار فرض کنید که هدف اصلی افزایش سرعت پخش باشد، نه کاهش هزینه‌ی آن. به عبارت دیگر، در اینجا هزینه نشان‌دهنده‌ی زمان ارسال پیام است و ما می‌خواهیم زمان سپری‌شده برای پخش سراسری را کمینه کنیم. می‌توان یک پیام را به صورت هم‌زمان روی چند خط فرستاد. فرض کنید پیام از یک پایگاه مشخص به پایگاه‌های همسایه‌ی آن فرستاده می‌شود و آن پایگاه‌ها نیز پیام را به دیگر پایگاه‌ها به گونه‌ای می‌فرستند که هر پایگاه، تنها یک پیام دریافت کند. می‌توانید فرض کنید تمام ساختار شبکه را می‌شناسید.

الف- اگر همراه با هر خط ارتباطی، تأخیر آن نیز مشخص شده باشد، الگوریتمی برای یافتن بهترین ارسال طراحی کنید.

ب- اگر پایگاه‌ها نیز تأخیر داشته باشند، الگوریتمی برای یافتن ارسال بهینه ارائه کنید. t(v) واحد زمانی طول می‌کشد تا پایگاه v، پیامی را به همسایگانش بفرستد. زمان فرستادن پیامی از v به k همسایه، kt(v) خواهد بود. (برای هر v، مقدار t(v) را می‌دانیم.)

۶۴-۷ G=(V,E) را گرافی وزن‌دار، بدون‌جهت و همبند بگیرید. برای سادگی، وزن‌ها را مثبت و متمایز فرض کنید. اگر e یالی از G باشد، T(e) نشان‌دهنده‌ی کم هزینه‌ترین درخت پوشایی است که e را در بر داشته باشد. الگوریتمی بنویسید که برای هر یال e $(e \in E)$ T(e) را بیابد. زمان اجرای این الگوریتم باید از $O(|V|^2)$ باشد.

۷-۶۵ الگوریتمی کارآمد طراحی کنید که در یک گراف همبند، وزن‌دار و بدون‌جهت، درخت پوشای با کم‌ترین هزینه از نظر گلوگاه را بیابد. (به خاطر بیاورید که وزن گلوگاه، وزن سنگین‌ترین یال زیرگراف بود.) به عبارت دیگر، شما باید درخت پوشایی را بیابید که سنگین‌ترین یال آن کمینه باشد.

۷-۶۶ این گونه‌ی مسأله‌ی تمرین ۷-۶۵ را در گراف‌های جهت‌دار حل کنید: ورودی، گراف جهت‌دار وزن‌داری با یک رأس مشخص از آن (v) است. درخت پوشای ریشه‌داری با ریشه‌ی v بیابید که پرهزینه‌ترین یال آن کمینه باشد. (به یاد آورید که در یک درخت ریشه‌دار، جهت همه‌ی یال‌ها به گونه‌ای بود که از ریشه دور می‌شدند.)

۷-۶۷ G=(V,E) را گرافی وزن‌دار و بدون‌جهت و T را یک MCST در آن بگیرید. اگر وزن همه‌ی یال‌های G به اندازه‌ی مقدار ثابت c افزایش بیابد، آیا هنوز هم T یک MCST است؟ اگر T، MCST نیست، تغییر آن به یک MCST چقدر دشوار است؟

۷-۶۸ ✯ G=(V,E) را گرافی بدون‌جهت، وزن‌دار و همبند و T را یک MCST در آن بگیرید. حال، رأس تازه‌ی v را همراه با چند «یال وزن‌دار از این رأس به رأس‌های پیشین G» به گراف می‌افزاییم. الگوریتمی با زمان خطی برای یافتن یک MCST تازه که v را هم در بر گیرد، طراحی کنید.

۷-۶۹ فرض کنید هزینه‌ی یک درخت پوشا، به جای جمع هزینه‌ی یال‌های آن، حاصل‌ضرب هزینه‌ی یال‌ها باشد (هزینه‌ها از صفر بیش‌ترند). با این فرض، الگوریتمی برای یافتن درخت پوشای بیشینه ارائه کنید. (می توانید همه‌ی هزینه‌ها را متمایز فرض کنید.)

۷-۷۰ G=(V,E) را گرافی بدون‌جهت و همبند با n رأس که از 1 تا n شماره‌گذاری شده‌اند، در نظر بگیرید. الگوریتمی کارآمد طراحی کنید که کوچک‌ترین k را چنان بیابد که یکایک مؤلفه‌های همبند گراف حاصل از حذف پی‌درپی رأس‌های با شماره‌های ۱، ۲، ... و k (به همین ترتیب) حداکثر n/۲ رأس داشته باشند. دقت کنید که با حذف هر رأس، تمام یال‌های گذرنده از آن نیز حذف خواهند شد. (راهنمایی: از ساختمان داده‌ی union-find کمک بگیرید.)

۷-۷۱ G=(V,E) را گرافی بدون‌جهت فرض کنید. زیرمجموعه‌ی F از یال‌های گراف ($F \subseteq E$) یک مجموعه یال بازخوردی نامیده می‌شود، اگر هر دور از G دست‌کم یک یال در F داشته باشد. الگوریتمی بنویسید که کوچک‌ترین مجموعه از یال‌های بازخوردی را بیابد.

۷-۷۲ G=(V,E) را گرافی وزن‌دار و بدون‌جهت با وزن‌های مثبت بگیرید. الگوریتمی طراحی کنید که در G یک مجموعه یال بازخوردی (تعریف‌شده در تمرین ۷-۷۱) با وزن کمینه بیابد.

۷-۷۳ ثابت کنید الگوریتم All_Pairs_Shortest_Paths (ارائه‌شده در شکل ۷-۲۲) برای گراف‌های وزن‌داری هم که وزن منفی نیز دارند، اما هیچ دوری با وزن منفی در آن‌ها موجود نیست، درست کار می‌کند.

۷-۷۴ G=(V,E) را گرافی وزن‌دار و جهت‌دار بگیرید که وزن برخی از یال‌های آن منفی است، اما هیچ دوری با وزن منفی ندارد (یعنی هیچ دوری در گراف وجود ندارد که مجموع وزن یال‌های آن منفی شود). T را درختی پوشا از G بگیرید و ریشه‌ی آن را v بنامید. الگوریتمی ارائه دهید که روشن سازد آیا درخت T تنها، دربرگیرنده‌ی «کوتاه‌ترین مسیرها از v به دیگر رأس‌های G» است یا نه. (پس، خروجی باید به صورت «بله» یا «خیر» باشد.)

۷-۷۵ می‌خواهیم کوتاه‌ترین مسیرها از یک رأس به رأس‌های دیگر را در گراف‌های وزن‌داری محاسبه کنیم که وزن منفی هم دارند، اما دوری با وزن منفی در آن‌ها موجود نیست. مانند تمرین ۷-۷۴، با یک درخت پوشای ریشه‌دار دل‌خواه کار خود را آغاز می‌کنیم. سپس به یاری الگوریتم تمرین ۷-۷۴ روشن می‌کنیم آیا این درخت دل‌خواه، درخت کوتاه‌ترین مسیرها از رأس ریشه هست یا نه. الگوریتم به‌دست‌آمده در تمرین ۷-۷۴ باید برای تشخیص این که درخت یافته‌شده، درخت مورد نظر نیست، دلیلی داشته باشد. با توجه به این دلیل، درخت را تغییر می‌دهیم. باید این روال را تا یافتن درخت کوتاه‌ترین مسیرها ادامه دهیم.

الف- این الگوریتم را با جزئیات آن شرح دهید.

ب- ثابت کنید الگوریم پس از $O(|V|.|E|)$ گام به پایان می‌رسد.

پ- برای بهبود الگوریتم، درخت آغازین مناسبی پیش‌نهاد کنید. نیازی نیست که با این تغییر، بدترین حالت هم بهبود یابد؛ بلکه بهبود حالت میانگین الگوریتم نیز کافی است.

۷-۷۶ G=(V,E) را گرافی وزن‌دار و جهت‌دار بگیرید که وزن برخی یال‌های آن منفی است. الگوریتمی کارآمد طراحی کنید که روشن کند آیا گراف، دوری با وزن منفی دارد یا نه. (یعنی خروجی الگوریتم باید «بله» یا «خیر» باشد.)

۷-۷۷ الف- G=(V,E) را گرافی جهت‌دار و v را رأسی از V بگیرید. رنگ هر یال E یا سیاه است یا قرمز. الگوریتمی با زمان خطی بنویسید که روشن کند آیا G دور ساده‌ای دارد که هم v را در بر گیرد و هم رنگ‌های آن یک در میان باشد (یعنی هر یال قرمز (سیاه) از این دور، دو یال همسایه‌ی سیاه (قرمز) داشته باشد). اگر چنین دوری در گراف وجود داشته باشد، الگوریتم باید دست‌کم یکی از این دورها را بیابد.

ب- مسأله را دوباره پس از برداشتن محدودیت عبور دور از رأس v حل کنید.

۷-۷۸ گراف بدون‌جهت و همبند G=(V,E) به شما داده شده است. کم‌ارتفاع‌ترین درخت پوشای G را بیابید. (ارتفاع یا بلندی درخت، بیش‌ترین فاصله‌ی ریشه از برگ‌هاست.)

۷-۷۹ یک مسیر هامیلتونی، مسیری ساده است که همه‌ی رأس‌های گراف را در بر گیرد. الگوریتمی طراحی کنید که تشخیص دهد آیا در گراف جهت‌دار بدون‌دور داده‌شده، مسیری هامیلتونی وجود دارد یا نه. زمان اجرای این الگوریتم باید خطی باشد.

۷-۸۰ الگوریتم Improved_Transitive_Closure داده‌شده در شکل ۷-۲۴ سه حلقه‌ی تودرتو دارد. حلقه‌ی نخست (یعنی بیرونی‌ترین حلقه) یک ستون و حلقه‌ی دوم، یک سطر را برمی‌گزینند و حلقه‌ی سوم، روی سطر برگزیده‌شده کار می‌کند. فرض کنید دو حلقه‌ی نخست را با یکدیگر جابه‌جا کنیم؛ یعنی حلقه‌ی نخست، یک سطر و حلقه‌ی دوم، یک ستون را برگزینند. به عبارت دیگر، دو خط نخست الگوریتم را با یکدیگر جابه‌جا می‌کنیم. (حاصل این کار، در شکل ۷-۴۶ با نام WRONG_Transitive_Closure نشان داده شده است.) با یک مثال نقض نشان دهید که این الگوریتم، دیگر درست کار نخواهد کرد.

الگوریتم: WRONG_Transitive_Closure(A)

ورودی: A (یک ماتریس همسایگی $n \times n$ برای نشان دادن گرافی جهت‌دار)
{اگر یال (x,y) در گراف باشد، A[x,y]، ture وگرنه false خواهد بود.A[x,x] نیز به ازای هر x، ture است.}

خروجی: در پایان، ماتریس A باید بیانگر بسط ترایای گراف باشد.

```
begin
    for x := 1 to n do
        for m := 1 to n do
            if A[x,m] then
                for y := 1 to n do
                    if A[m,y] then A[x,y] := true
end
```

شکل ۷-۴۶ الگوریتم WRONG_Transitive_Closure

۷-۸۱ در الگوریتم بسط ترایا، اگر ماتریس آن قدر بزرگ باشد که نتوانیم آن را در حافظه‌ی اصلی نگه داریم، ناچاریم ماتریس را در حافظه‌ی کمکی ذخیره کنیم. به همین دلیل بود که ترتیب پویش عناصر ماتریس را در این الگوریتم جابه‌جا کردیم. (در تمرین ۷-۸۰ تلاش شد تا چنین کاری انجام شود، اما موفقیت‌آمیز نبود.) فرض کنید ماتریس، سطر به سطر ذخیره شده باشد، به گونه‌ای که هر سطر آن به اندازه‌ی یک صفحه‌ی حافظه باشد. می‌خواهیم از تعداد صفحه‌هایی که لازم است از حافظه‌ی کمکی واکشی شود، تا جای ممکن بکاهیم. اگر نخستین حلقه، ماتریس را ستون به ستون پویش کند، آنگاه برای بررسی هر ستون به همه‌ی سطرها نیاز داریم. از سوی دیگر، اگر دو حلقه‌ی نخست را با یکدیگر جابه‌جا کنیم و دریابیم که مقدار خانه‌ی مشخصی مانند (x,y)، false است، آنگاه در گام بعد، دیگر نیازی به واکشی yامین سطر نداریم. بنابراین، اگر ماتریس تنک یا خلوت باشد (یعنی تعداد اندکی ۱ داشته باشد) آنگاه به واکشی تعداد کم‌تری از صفحه‌ها نیاز خواهد بود. الگوریتم شکل ۷-۴۶ نادرست است، اما می‌توانیم آن را اصلاح کنیم.

الف- نشان دهید که اگر این الگوریتم را O(logn) بار اجرا کنیم، بسط ترایا به درستی حساب خواهد شد.

ب- نشان دهید دو بار اجرای الگوریتم برای دست‌یابی به نتیجه کافی است.

۸۲-۷ G=(V,E) را یک گراف چندگانه بگیرید؛ یعنی گرافی بدون‌جهت که ممکن است بین هر دو رأس آن، بیش از یک یال وجود داشته باشد. در این حالت، E یک مجموعه‌ی چندگانه و $|E|$ تعداد کل یال‌های گراف است. الگوریتمی از $O(|E|+|V|)$ بنویسید که همه‌ی رأس‌هایی را که درجه‌ی ۲ دارند، با جای‌گزینی یالی مانند (u,w) به جای یال‌هایی مانند (u,v) و (v,w) حذف کند و به جای همه‌ی یال‌های چندگانه بین هر دو رأس نیز تنها یک یال قرار دهد. (دقت کنید که جای‌گزینی چندین یال بین دو رأس با یک یال، ممکن است رأس تازه‌ای با درجه‌ی ۲ به وجود آورد که در این صورت، باید این رأس را هم حذف کرد و حذف این رأس نیز ممکن است یال‌هایی چندگانه پدید آورد که این یال‌ها هم باید حذف شوند.)

۸۳-۷ یک گراف بدون‌جهت و همبند، دوهمبند یالی نامیده می‌شود، اگر پس از حذف هر یال دل‌خواه از آن بازهم همبند باقی بماند. الگوریتمی با زمان خطی برای تشخیص دوهمبند یالی بودن گراف طراحی کنید.

۸۴-۷ ورودی، یک گراف بدون‌جهت و همبند همراه با سه یال a، b و c از آن است. مسأله، بررسی وجود دوری در گراف است که a و b را در بر گیرد، اما شامل c نباشد. الگوریتم حل این مسأله باید در زمانی خطی اجرا گردد.

۸۵-۷ G=(V,E) را گرافی بدون‌جهت و همبند و T=(V,F) را درختی پوشا در G بگیرید. ثابت کنید اشتراک F با مجموعه یال‌های یک مؤلفه‌ی دوهمبند، درختی پوشا در آن مؤلفه‌ی دوهمبند تشکیل می‌دهد.

۸۶-۷ یک گسترش دوهمبند از گراف G=(V,E) گرافی دوهمبند مانند $G'=(V,E')$ است به گونه‌ای که $E \subseteq E'$. گراف بدون‌جهت G=(V,E) داده شده است. کوچک‌ترین گسترش دوهمبند آن را بیابید؛ یعنی گسترشی دوهمبند برای G بیابید که تعداد یال‌های آن کمینه باشد. (راه‌نمایی: نخست، گراف‌های بسیار ساده را در نظر بگیرید، سپس کار را آن قدر ادامه دهید تا به حالت کلی گراف‌ها برسید.)

۸۷-۷ فرض کنید یک گراف بدون‌جهت همراه با همه‌ی نقاط پیوند آن به شما داده شده باشد. بدون اجرای کل الگوریتمِ مؤلفه‌های دوهمبند، روشی برای یافتن این مؤلفه‌ها پیدا کنید.

۸۸-۷ G=(V,E) را گرافی جهت‌دار و T را یک درخت DFS در آن بگیرید. ثابت کنید اشتراک یال‌های T با یال‌های هر مؤلفه‌ی قویاً همبند G، زیردرختی از T است.

۸۹-۷ هر یک از مقدارهای High که در الگوریتم Strongly_Connected_Components (شکل ۷-۳۳) حساب می‌شود، واقعاً بیانگر «بالاترین» رأس دست‌رس‌پذیر از رأس مورد نظر نیست؛

بلکه تنها نشان می‌دهد آیا مؤلفه‌ای قویاً همبند پیدا شده است یا نه. الگوریتمی با زمان خطی طراحی کنید که برای هر رأس v از گراف، رأسی دست‌رس‌پذیر از آن را چنان بیابد که شماره‌ی DFS برای رأس یافته‌شده بیش‌ترین مقدار ممکن باشد. (شماره‌های DFS بر پایه‌ی یک درخت DFS خاص به دست آمده‌اند که در آن درخت، شماره‌های DFS کاهشی است.)

۷-۹۰ G=(V,E) را گرافی همبند و بدون‌جهت بگیرید. الگوریتمی با زمان خطی ارائه کنید که روشن سازد آیا می‌توان یال‌های G را به گونه‌ای جهت‌گذاری کرد که گراف جهت‌دار حاصل، قویاً همبند گردد. اگر این کار امکان‌پذیر باشد، الگوریتم باید آن را انجام دهد.

۷-۹۱ الف- این قضیه را ثابت کنید: گراف جهت‌دار G=(V,E) قویاً همبند است، اگر و تنها اگر مداری داشته باشد که هر یال را دست‌کم یک بار در بر گیرد. (دقت کنید که هر یال می‌تواند بیش از یک بار در مدار ظاهر شود.)

ب- الگوریتمی طراحی کنید که پس از دریافت گرافی قویاً همبند، چنین مداری در آن بیابد.

۷-۹۲ یک «پایه‌ی رأسی» در گراف جهت‌دار G=(V,E)، کوچک‌ترین زیرمجموعه‌ای از رأس‌های آن مانند B است ($B \subseteq V$) که به ازای هر رأس $v \in V$ رأسی مانند $b \in B$ یافت شود که مسیری به طول ۰ یا بیش‌تر از b به v وجود داشته باشد. ادعاهای دو بند «الف» و «ب» را ثابت کنید و به یاری آن‌ها الگوریتمی با زمان خطی بسازید که برای هر گراف جهت‌دار دل‌خواه، یک پایه‌ی رأسی بیابد.

الف- رأسی با درجه‌ی ورودی غیرصفر که در هیچ دوری قرار نداشته باشد، هرگز نمی‌تواند متعلق به هیچ یک از پایه‌های رأسی باشد.

ب- پایه‌ی رأسی گراف جهت‌داری که دور نداشته باشد، یکتاست و یافتن این پایه‌ی رأسی نیز آسان است.

۷-۹۳ یک گراف جهت‌دار را یک‌جانبه می‌گویند، اگر به ازای هر دو رأس آن، دست‌کم یکی از دو رأس از دیگری دست‌رس‌پذیر باشد. پس هر گراف قویاً همبند یک‌جانبه است، اما بسیاری از گراف‌های یک‌جانبه قویاً همبند نیستند؛ مانند گرافی جهت‌دار، دارای دو رأس که با یک یال به یکدیگر متصل شده باشند. الگوریتمی با زمان (و فضای) خطی طراحی کنید که روشن سازد آیا گراف جهت‌دار داده‌شده، یک‌جانبه است یا نه. (راه‌نمایی: از گراف مؤلفه‌های قویاً همبند یاری بگیرید.)

۷-۹۴ گراف جهت‌دار G=(V,E)، تک‌مسیره خوانده می‌شود، اگر در صورت دست‌رس‌پذیری w از v، تنها یک مسیر ساده از v به w وجود داشته باشد. الگوریتمی کارآمد بنویسید که تشخیص دهد گراف داده‌شده، تک‌مسیره است یا نه. (راه‌نمایی: نخست، مسأله را برای گراف‌های بدون‌دور حل کنید.)

۷-۹۵ الگوریتمی با زمان خطی برای یافتن تطابقی بیشینه در یک درخت داده‌شده طراحی کنید.

۹۶-۷ قضیه‌ی مسیرهای یک‌درمیان را به صورت مستقیم و بدون کمک شارهای شبکه یا برش‌ها ثابت کنید. (راه‌نمایی: برای دو تطابق داده‌شده‌ی M_1 و M_2، ویژگی‌های تفاضل متقارن آن‌ها را - یعنی مجموعه‌ی همه‌ی یال‌هایی را که دقیقاً در یکی از دو تطبیق آمده‌اند - بررسی کنید.)

۹۷-۷ G را گرافی بدون‌جهت و دوبخشی و M را تطبیقی دل‌خواه از آن بگیرید.

الف- این قضیه را ثابت کنید: تطابقی بیشینه در G وجود دارد که همه‌ی رأس‌های زیرپوشش M را در بر می‌گیرد. (یک رأس، هنگامی زیرپوشش تطبیق M نامیده می‌شود که سر یالی از M باشد.)

ب- از روی برهان بند «الف»، الگوریتم یافتن چنین تطبیق بیشینه‌ای را برای G و M داده‌شده طراحی کنید.

۹۸-۷ ✯ ثابت کنید زمان اجرای الگوریتم تطابق دوبخشی Hopcroft و Karp (همان الگوریتم بهبودیافته‌ی بخش ۷-۱۰) در بدترین حالت از $O((m+n)\sqrt{n})$ است.

۹۹-۷ فرض کنید می‌خواهیم تطابقی در یک گراف بیابیم که تک‌همتایی نباشد. به عبارت دیگر، به جای یال‌های جداازهم، در جست‌وجوی گراف‌های ستاره‌ای جداازهم هستیم. (این گراف‌ها، درخت‌هایی هستند که در آن‌ها رأس ریشه به همه‌ی رأس‌های دیگر متصل است.) یک یال، حالت خاصی از گراف‌های ستاره‌ای است، اما یک رأسِ بدون یال را گراف ستاره‌ای در نظر نمی‌گیریم. اگر گراف همبند و بدون‌جهت G=(V,E) داده شده باشد، می‌خواهیم الگوریتمی طراحی کنیم که در G یک دسته گراف‌های ستاره‌ای با رأس‌هایی جداازهم را به گونه‌ای بیابد که هر یک از ستاره‌ها دست‌کم دو رأس داشته باشد. هر رأس باید تنها در یکی از گراف‌های ستاره‌ای (یا ستاره‌ها) باشد، اما نیازی نیست که همه‌ی یال‌ها پوشش داده شوند. به عبارت دیگر، ستاره‌ها باید همه‌ی رأس‌ها را و نه لزوماً همه‌ی یال‌ها را در بر گیرند. (در اینجا هیچ محدودیتی از نظر بیشینگی یا کمینگی وجود ندارد.)

الف- در اینجا الگوریتمی برای حل مسأله ارائه می‌کنیم. هم با تشریح نادرستی استدلال و هم با آوردن مثال نقض، اِشکال این الگوریتم را نشان دهید.

الگوریتم نادرست: از استقرا بهره می‌گیریم. فرض استقرا این است که می‌دانیم چگونه مسأله را برای گرافی با کم‌تر از n رأس حل کنیم. برای گراف داده‌شده‌ی G=(V,E) با n رأس، نخست رأسی دل‌خواه را همراه با همه‌ی همسایگانش حذف می‌کنیم. ممکن است گراف برجای‌مانده دیگر همبند نباشد، اما می‌توانیم هر مؤلفه‌ی همبند را به صورت جداگانه در نظر بگیریم و همین الگوریتم را بنا به استقرا برای آن مؤلفه به کار ببریم.

ب- الگوریتمی کارآمد (و البته درست) برای حل این مسأله طراحی کنید.

۷-۱۰۰ ورودی، یک گراف وزن‌دار و دوبخشی با n رأس و m یال است. الگوریتمی با زمان اجرایی از $O(\sqrt{nm}\log n)$ ارائه دهید که کم‌وزن‌ترین تطابق بیشینه را از نظر گلوگاه بیابد. (وزن گلوگاه تطبیق M، چنان که گفته شد، وزن سنگین‌ترین یال آن است.)

۷-۱۰۱ صفحه‌ای $N \times N$ مانند صفحه‌ی شطرنج در نظر بگیرید (یعنی خانه‌ها یک در میان سیاه و سفید هستند). به یاری قضیه‌ی مسیر یک‌درمیان ثابت کنید اگر به دل‌خواه، یک خانه‌ی سیاه و یک خانه‌ی سفید را کنار بگذاریم، می‌توانیم بقیه‌ی صفحه را با مهره‌های دومینو (که اندازه‌ی هر یک از آن‌ها ۱×۲ است) بپوشانیم.

۷-۱۰۲ خانه‌های صفحه‌ی تمرین ۷-۱۰۱ را رأس‌های یک گراف بگیرید و بین رأس‌های متناظر با خانه‌های همسایه یال بگذارید. حال، قضیه‌ی تمرین ۷-۱۰۱ را با یافتن دوری هامیلتونی در این گراف ثابت کنید.

۷-۱۰۳ فرض کنید G=(V,E) گرافی همبند و بدون‌جهت باشد و دو درخت پوشای T و R از آن را به شما داده باشند. کوتاه‌ترین دنباله‌ی T_0، T_1، ... و T_k از درخت‌هایی را بیابید که هم T_0=T، هم T_k=R و هم در هر درخت از این دنباله بتوان با حذف یک یال و افزودن یک یال دیگر، درخت بعدی را به دست آورد.

۷-۱۰۴ قرار است یک مسابقه‌ی دوره‌ای بین n نفر برگزار شود. پس، هر بازی‌کن با n-۱ بازی‌کن دیگر مسابقه می‌دهد. بازی‌ها نتیجه‌ی «مساوی» ندارند و نتیجه‌ی آن‌ها را به صورت یک ماتریس به ما داده‌اند. در حالت کلی، نمی‌توان بازی‌کنان را مرتب کرد، چراکه ممکن است A، B را ببرد، B، C را شکست دهد و C هم بر A پیروز شود (به عبارت دیگر، ممکن است نتیجه‌ی بازی‌ها ترایا نباشد). الگوریتمی طراحی کنید که با توجه به ماتریس نتیجه‌ی بازی‌ها، بازی‌کنان را به ترتیب P_1، P_2، ... و P_n چنان مرتب کند که P_1، P_2 را، P_2، P_3 را و ... (و سرانجام P_{n-1}، P_n را) شکست داده باشد. (چنین ترتیبی را یک ترتیب ضعیف می‌گوییم.) زمان اجرای این الگوریتم در بدترین حالت باید از O(nlogn) باشد. (زمان دست‌یابی به هر یک از خانه‌های ماتریس مقداری ثابت است.)

۷-۱۰۵ n عدد صحیح d_1، d_2، ... و d_n داده شده‌اند. می‌دانیم که $d_1 + d_2 + ... + d_n$ عددی زوج است و $0 \le d_1 \le d_2 \le ... \le d_n$. همچنین، برای هر i از بازه‌ی [2,n] داریم: $d_i \langle d_1 + d_2 + ... + d_{i-1}$. الگوریتمی برای ساختن یک گراف چندگانه و بدون‌جهت با n رأس که درجه‌ی رأس‌های آن دقیقاً d_1، d_2، ... و d_n باشد، ارائه دهید و درستی آن را ثابت کنید. (با این کار می‌توان نتیجه گرفت همواره چنین گراف چندگانه‌ای وجود دارد.)

۷-۱۰۶ ✯ «رنگ‌آمیزی یالی» گراف، نسبت دادن رنگ‌هایی به یال‌های آن است (یعنی هر یال، تنها یک رنگ خواهد داشت) به گونه‌ای که هر دو یال گذرنده از یک رأس، رنگ‌های متفاوتی داشته باشند. اگر گرافی دوبخشی و بدون‌جهت به شما بدهند که درجه‌ی همه‌ی

رأس‌هایش $k = 2^n$ باشد (یعنی k توانی از ۲ باشد) الگوریتمی برای یافتن یک رنگ‌آمیزی یالی گراف با k رنگ بیاورید. زمان اجرای الگوریتم باید از $O(|E|\log k)$ باشد.

۷-۱۰۷ یک پوشش یالی در یک گراف بدون‌جهت، مجموعه‌ای از یال‌های گراف است که هر رأس گراف دست‌کم سرِ یکی از این یال‌ها باشد. الگوریتمی طراحی کنید که برای هر گراف دوبخشی، کوچک‌ترین پوشش یالی را بیابد.

۷-۱۰۸ یک پوشش رأسی در گراف بدون‌جهت G=(V,E)، مجموعه‌ای از رأس‌ها مانند U است که دست‌کم یک سر هر یال گراف، در U باشد. الگوریتمی کارآمد طراحی کنید که پس از دریافت یک درخت در ورودی، کوچک‌ترین پوشش رأسی آن را بیابد. (حالت کلی مسأله‌ی پوشش رأسی گراف، در فصل ۱۱ مورد بحث و بررسی قرار می‌گیرد.)

۷-۱۰۹ G=(V,E) را درختی بگیرید که رأس‌های آن وزن دارند؛ وزن هر رأس هم برابر درجه‌ی آن است. الگوریتمی بنویسید که کم‌وزن‌ترین پوشش رأسی G (یعنی یک پوشش رأسی در G با وزن کمینه) را بیابد.

۷-۱۱۰ الگوریتمی کارآمد طراحی کنید که کوچک‌ترین پوشش رأسی را برای گراف دوبخشی ورودی بیابد. (راه‌نمایی: بین این مسأله و برش‌های کمینه‌ی گراف، رابطه‌ای بیابید.)

۷-۱۱۱ فرض کنید G=(V,E) گرافی بدون‌جهت باشد. یک مجموعه‌ی مستقل در G، مجموعه‌ای از رأس‌هاست که هیچ زوجی از آن‌ها به یکدیگر متصل نباشند. الگوریتمی کارآمد طراحی کنید که در یک گراف دوبخشی داده‌شده، بزرگ‌ترین مجموعه‌ی مستقل را بیابد. (راه‌نمایی: رابطه‌ی مسأله با تمرین ۷-۱۱۰ را پیدا کنید.) در فصل ۱۱، مسأله‌ی مجموعه‌های مستقل را برای گراف‌های دل‌خواه (بدون محدودیت) بررسی می‌کنیم.

۷-۱۱۲ الگوریتمی بیاورید که مجموعه‌ای مستقل و گسترش‌ناپذیر در گراف بدون‌جهت ورودی پیدا کند. (تعریف مجموعه‌ی مستقل در تمرین ۷-۱۱۱ آمده است.) لازم نیست بزرگ‌ترین مجموعه‌ی مستقل را بیابید؛ کافی است مجموعه‌ی مستقلی را پیدا کنید که پس از افزودن رأس دیگری به آن غیرمستقل شود.

۷-۱۱۳ G=(V,E) را درختی بگیرید که وزن هر رأس v از آن، w(v) باشد. الگوریتمی با زمان خطی طراحی کنید که در G سنگین‌ترین مجموعه‌ی مستقل را بیابد. (تعریف مجموعه‌ی مستقل در تمرین ۷-۱۱۱ آمده است.)

۷-۱۱۴ اگر G=(V,E) گرافی بدون‌جهت و همبند باشد، الگوریتمی ارائه کنید که روشن سازد آیا G پوششی رأسی (تمرین ۷-۱۰۸) با حداکثر k رأس دارد که همه‌ی آن‌ها مستقل باشند یا نه. (مستقل بودن رأس‌های پوشش، یعنی هیچ زوجی از آن‌ها همسایه نیستند.)

۷-۱۱۵ الگوریتمی طراحی کنید که پس از دریافت گرافی بدون‌جهت، مشخص سازد آیا این گراف، مجموعه‌ای از رأس‌ها مانند U دارد که هم پوشش رأسی کمینه و هم مجموعه‌ی مستقل بیشینه باشند. اگر چنین مجموعه‌ای وجود داشته باشد، الگوریتم باید آن را پیدا کند.

۷-۱۱۶ یک گراف بازه‌ای، گرافی است بدون‌جهت که رأس‌هایش متناظر با بازه‌هایی از اعداد حقیقی هستند و دو رأس هنگامی به یکدیگر متصل هستند که اشتراک بازه‌های متناظر با آن‌ها تهی نباشد. اگر G=(V,E) یک گراف بازه‌ای باشد و بازه‌های متناظر با رأس‌های آن را بدانیم، الگوریتمی کارآمد ارائه دهید که بزرگ‌ترین مجموعه‌ی مستقل را در G پیدا کند.

۷-۱۱۷ گراف بدون‌جهت G=(V,E) را گرافی دوپاره گویند، اگر بتوان مجموعه‌ی رأس‌های آن را به دو مجموعه‌ی جداازهم U و W به گونه‌ای افراز کرد که گراف القایی با U، بدون یال و گراف القایی با W، گرافی کامل باشد. (گراف کامل، گرافی است که همه‌ی یال‌های ممکن در آن وجود داشته باشند.) الگوریتمی با زمان خطی طراحی کنید که روشن سازد آیا گراف داده‌شده، دوپاره است یا نه.

۷-۱۱۸ الف- الگوریتمی طراحی کنید که گراف بدون‌جهت G=(V,E) را بگیرد و تشخیص دهد آیا این گراف، زیرگرافی به شکل مثلث دارد یا نه. زمان اجرای الگوریتم باید از $O(|V|.|E|)$ باشد.

ب- آیا الگوریتم شما می‌تواند همه‌ی زیرگراف‌های مثلثی G را پیدا کند.

۷-۱۱۹ الف- الگوریتمی بنویسید که گراف بدون‌جهت G=(V,E) را بگیرد و روشن کند آیا این گراف، زیرگرافی مربعی (یعنی دوری با طول ۴) دارد یا نه. زمان اجرای الگوریتم باید از $O(|V|^3)$ باشد.

ب- الگوریتم نوشته‌شده را تا رسیدن به زمان اجرایی از $O(|V|.|E|)$ بهبود دهید. می‌توانید به دل‌خواه، هر یک از دو روش ماتریس همسایگی یا لیست همسایگی را برای ذخیره‌ی گراف به کار ببرید.

۷-۱۲۰ ثابت کنید هیچ الگوریتمی که زمان اجرای آن در بدترین حالت از $O(|V|.|E|)$ باشد، نمی‌تواند در گراف بدون‌جهت داده‌شده‌ی G=(V,E)، همه‌ی زیرگراف‌های مربعی را پیدا کند.

۷-۱۲۱ ✯ T را درخت ریشه‌داری بگیرید که جهت‌دار است، اما شاید دودویی نباشد. هر رأس این درخت، وزنی دارد که از وزن والد آن رأس بیش‌تر است. (درخت، ساختار یک هرم را دارد که در آن رأس‌های کم‌وزن‌تر بالاتر قرار گرفته‌اند.) هر رأس می‌تواند به عنوان یک رأس معمولی یا یک رأس محور برگزیده شود. در رأس‌های محور، هزینه برابر وزن است، اما در رأس‌های معمولی، هزینه کم‌تر از وزن است (و برابر وزن رأس منهای وزن نزدیک‌ترین محور بالادست آن تعریف می‌شود). پس با گزینش یک رأس به عنوان محور ممکن است هزینه‌ی آن رأس

افزایش یابد، اما شاید هزینه‌ی برخی از رأس‌های پایین‌دست آن کاهش یابد. الگوریتمی کارآمد طراحی کنید که تعدادی از رأس‌ها را به عنوان محور چنان برگزیند که مجموع هزینه‌های تمام رأس‌ها کمینه گردد. (هیچ محدودیتی روی تعداد رأس‌های محور نیست.)

۷-۱۲۲ T را درخت دودویی کاملی به ارتفاع h، با $n = 2^h - 1$ رأس بگیرید. می‌خواهیم T را به صورتی که گفته خواهد شد، درون صفحه رسم کنیم. هر رأس، متناظر با نقطه‌ای یکتا از یک توری منظم است (یعنی طول و عرض هر نقطه، در دستگاه مختصات عددهایی صحیح هستند). رأس‌های همسایه با پاره‌خط‌های مستقیم به یکدیگر متصل شده‌اند. هیچ یک از پاره‌خط‌ها نیز، دیگری را قطع نمی‌کند. این روش ترسیم گراف، مسأله‌ای مهم در طراحی تراشه‌های الکترونیکی (به ویژه VLSI) است. هدف ما در این تمرین، کمینه کردن مساحت مستطیل دربرگیرنده‌ی گراف است؛ یعنی می‌خواهیم مساحت مستطیلی را که تنها از نقاط اشغال‌نشده می‌گذرد و گراف را هم در بر می‌گیرد، کمینه کنیم. پس برای مثال، زنجیره‌ای مستقیم از k رأس را باید در مستطیلی با مساحت ۲(k+۱) قرار داد. روشن است که برای هر گراف با n رأس، کم‌ترین مساحت ممکن از $\Omega(n)$ خواهد بود.

الف- مستطیل دربرگیرنده‌ی T را چنان بیابید که مساحتش از O(n) باشد. (راه‌نمایی: از شیوه‌ی تقسیم‌وحل کمک بگیرید، چراکه هر درخت دودویی کامل از دو درخت دودویی کامل و کوچک‌تر تشکیل شده است که هر دوی آن‌ها به یک ریشه‌ی مشترک متصل گشته‌اند. فرض کنید می‌دانید چگونه باید مستطیل مناسب را برای درخت‌های با ارتفاع h-۱ پیدا کنید؛ سپس مستطیل دربرگیرنده‌ی درختی به ارتفاع h را بیابید.)

ب- الگوریتمی بنویسید که مختصات هر رأس T را در مستطیل به‌دست‌آمده در بند «الف» حساب کند.

واژه‌نامه‌ی پارسی به انگلیسی

kامین عنصر از نظر بزرگی، آماری معکوس kام	kth-largest element
kامین عنصر از نظر کوچکی، آماری kام	kth-smallest element
kامین کلید (عنصر) از نظر کوچکی	kth smallest key(element)
k–مسیر	k-path
k–همبند	k-connected
NP–تمام	NP-complete
NP–سخت	NP-hard
آرایه‌ی پسوندی	suffix array
ابردنباله	supersequence
استقرای دوطرفه	two sided induction
استقرای دوگانه	double induction
استقرای معکوس	reversed induction
استلزام	implication
اشاره‌گر	pointer
اصل لانه‌ی کبوتر	pigeonhole principle
اعداد شبه‌تصادفی	pseudorandom numbers
افراز، بخش‌بندی	partition
اکثریت	majority
الحاق	concatenate
الگوریتم بازگشتی	recursive algorithm
الگوریتم تکراری	iterative algorithm
الگوریتم توزیع‌شده	distributed algorithm
الگوریتم قطعی	deterministic algorthim
الگوریتم مبتنی بر احتمال یا احتمال‌گرا	probabilistic algorithm
الگوریتم هافمن	Huffman's algorithm
الگوریتم‌های ترکیبیاتی	combinatorial algorithms
الگوریتم هم‌زمان	on-line algorithm

انگشت‌نگاری	fingerprinting
با ترتیب کلی	totally ordered
بالادست	ancestor
برخورد	collision
برخورد ثانویه	secondary collision
بررسی جامع	survey
بررسی خطی	linear probing
برش	cut
برنامه‌نویسی پویا	dynamic programming
بزرگ‌ترین زیردنباله‌ی متوالی یا به‌هم‌پیوسته	maximum consecutive subsequence
بزرگ‌ترین زیرگراف القایی	maximal induced subgraph
بسامد	frequency
بسط ترایا	transitive closure
بلندترین زیردنباله‌ی صعودی یا افزایشی	longest increasing subsequence
بهینه‌سازی ترکیبیاتی	combinatorial optimization
پایه‌ی رأسی	vertex basis
پایین‌دست	descendant
پخش، پخش سراسری	broadcast
پرونده	file
پشته	stack
پل	bridge
پوچ، قلابی، ساختگی	dummy
پوسته‌ی کوژ	convex hull
پوشش رأسی	vertex cover
پوشش یالی	edge cover
پویا	dynamic
پیچیدگی در حالت سرشکن‌شده	amortized complexity
پیمایش گراف	graph traversal
پیوند	join
تابع درهم‌ساز	hash function
تابع درهم‌ساز عمومی	universal hash

تابع صعودی	monotically growing
تابع لگاریتم پی‌درپی	iterated logarithm function
تبارشناسی	genealogical
تجزیه‌ی خوشه‌ای	ear decomposition
تحلیل سرشکن‌شده	amortized analysis
ترایایی	transitivity
ترتیب توپولوژیک	topological sorting
ترتیب چرخشی	cyclic order
ترتیب ضعیف	weak sorting
ترجمه	compile
تضاد ابداعی	inventor's paradox
تطابق بیشینه	maximum matching
تطابق تک‌همتایی	monogamous matching
تطابق دوبخشی	bipartite matching
تطابق کامل	perfect matching
تطابق گسترش‌ناپذیر	maximal matching
تطبیق، تطابق	matching
تطبیق جزئی	partial match
تطبیق رشته یا تطابق رشته‌ای	string matching
تطبیق کامل	perfect matching
تعمیم	generalization
تفاضل متقارن	symmetric difference
تقریب استرلینگ	Stirling's approximation
تقسیم‌وحل	divide and conquer
تکرار کلید	duplicate
توری منظم، شبکه	lattice
توزیع نسبتاً یک‌نواخت	relatively uniformly distributed
ثبّات	register
جای‌گشت	permutation
جداکننده	delimiter
جدول درهم	hash table

interpolation search	جست‌وجو با درون‌یابی
priority search	جست‌وجوی اولویت‌دار
binary search	جست‌وجوی دودویی
pure binary search	جست‌وجوی دودویی محض
breadth-first search	جست‌وجوی نخست-پهنا، جست‌وجوی سطح-اول
depth-first search	جست‌وجوی نخست-ژرفا، جست‌وجوی عمق-اول
forest	جنگل
sink	چاهک
double rotation	چرخش دوگانه
single rotation	چرخش منفرد
source	چشمه
multidimensional	چندبعدی
information-theoretic lower bound	حد پایین نظری
elimination of history	حذف حافظه
entry	خانه، درایه
successor	خلف
well-defined	خوش‌تعریف
clustering	خوشه‌ی پرشده
abstract data type	داده‌گونه‌ی مجرد
degree	درجه
outdegree	درجه‌ی خروجی
indegree	درجه‌ی ورودی
tree	درخت
2-3 tree	درخت ۲-۳
B-tree	درخت B
DFS tree	درخت DFS
free tree	درخت آزاد
optimal tree	درخت بهینه
suffix tree	درخت پسوندی
spanning tree	درخت پوشا
minimum-cost spanning tree	درخت پوشای کمینه

decision tree	درخت تصمیم‌گیری
weight-balance tree	درخت تناسب وزنی
search tree	درخت جست‌وجو
self-adjusting tree	درخت خودتنظیم
binary tree	درخت دودویی
binary search tree	درخت دودویی جست‌وجو
complete binary tree	درخت دودویی کامل
biconnected tree (biconnected component tree)	درخت دوهمبندی (درخت دوهمبندی مؤلفه‌ها)
rooted tree	درخت ریشه‌دار
red-black tree	درخت قرمز-سیاه
shortest path tree	درخت کوتاه‌ترین مسیر
balanced tree	درخت متوازن
balanced-search tree	درخت متوازن‌کننده‌ی جست‌وجو
Huffman tree	درخت هافمن
hashing	درهم‌سازی
double hashing	درهم‌سازی دوگانه
coalesced hashing	درهم‌سازی یک‌پارچه
reachable	دست‌رس‌پذیر
tail	دم، مبدأ
sequence	دنباله
cyclic sequence	دنباله‌ی چرخشی
realizable sequence	دنباله‌ی دست‌یافتنی
cycle	دور
Hamiltonian tour (Hamiltonian cycle)	دور هامیلتونی
biconnectivity	دوهمبندی
edge-biconnected	دوهمبند یالی
representation	ذخیره، نمایش
Euler's formula	رابطه‌ی اویلر
ordering relationship	رابطه‌ی ترتیب
vertex	رأس

rank	رتبه
formalize	رسمیت بخشیدن
string	رشته
Harmonic series	رشته‌ی توافقی
asymptotic behavior	رفتار مجانبی
valid coloring	رنگ‌آمیزی معتبر
edge coloring	رنگ‌آمیزی یالی
recurrence relations with full history	روابط بازگشتی با حافظه‌ی کامل
bisection method	روش تنصیف
linear congruential method	روش هم‌نهشتی خطی
flowchart	روندنما
separate chaining	زنجیره‌بندی مجزا
ordered pair	زوج مرتب
subsequence	زیردنباله
maximum subsequence	زیردنباله‌ی بیشینه
stuttering-subsequence	زیردنباله‌ی ناپایدار
vertex-induced subgraph	زیرگراف القاشده با رأس‌ها
induced subgraph	زیرگراف برآمده، زیرگراف القایی
data structure	ساختمان داده‌ای، ساختمان‌داده
consistent	سازگار
system	سامانه
head	سر، مقصد
straightforward	سرراست
hierarchial	سلسله‌مراتبی
predecessor	سلف
network flows	شارهای شبکه
network	شبکه
conservation constraint	شرط پایستگی
decreasing DFS number	شماره‌ی کاهشی DFS
decompositions of graphs	شیوه‌های تجزیه‌ی گراف
greedy method	شیوه‌ی آزمندانه یا حریصانه

explicit	صریح
queue	صف
priority queue	صف اولویت
first-in first-out (FIFO) queue	صف ترتیبی
implicit	ضمنی
slack of the edge	ظرفیت مانده‌ی یال
balance factor	عامل توازن
asymmetry	عدم تقارن
backtracking	عقب‌گرد
minimal edit distance	فاصله‌ی ویرایشی کمینه
multiplicity	فراوانی
child	فرزند
dictionary, data dictionary	فرهنگ داده‌ای
path compression	فشرده‌سازی مسیر
Fibonacci	فیبوناچی
loop invariant	قانون ثابت حلقه
max-flow min-cut theorem	قضیه‌ی شار بیشینه-برش کمینه
integral-flow theorem	قضیه‌ی شار مجموع
anti-Gray code	کد ضد گری
encoding	کدگذاری
Huffman's encoding	کدگذاری هافمن
open Gray code	کد گری باز
closed Gray code	کد گری بسته
wild card	کلید عمومی یا جای‌گزین
single-source shortest path	کوتاه‌ترین مسیر از یک رأس به رأس‌های دیگر
all shortest paths	کوتاه‌ترین مسیرها بین تمام زوج‌رأس‌های گراف
brute force	کورکورانه
Euclidean graph	گراف اقلیدسی
Eulerian graph	گراف اویلری
interval graph	گراف بازه‌ای
undirected graph	گراف بدون‌جهت

acyclic graph	گراف بدون‌دور
unipathic graph	گراف تک‌مسیره
sparse graph	گراف تنک یا خلوت
directed graph	گراف جهت‌دار
dense graph	گراف چگال
multigraph	گراف چندگانه
bipartite graph	گراف دوبخشی
split graph	گراف دوپاره
simple graph	گراف ساده
star graph (star)	گراف ستاره‌ای (ستاره)
residual graph	گراف ظرفیت‌های باقی‌مانده
condensation	گراف فشرده‌شده
planar graph	گراف مسطح
weighted graph	گراف وزن‌دار
connected graph	گراف همبند
unilateral graph	گراف یک‌جانبه
node	گره
critical node	گره بحرانی
internal node	گره داخلی
linked list	لیست پیوندی
adjacency list	لیست همسایگی
symmetric matrix	ماتریس متقارن
adjacency matrix	ماتریس همسایگی یا مجاورت
random-access machine	ماشین با دست‌رسی تصادفی
Turing machine	ماشین تورینگ
totally ordered set	مجموعه با ترتیب کلی
feedback-edge set	مجموعه یال بازخوردی
multiset	مجموعه‌ی چندگانه
independent set	مجموعه‌ی مستقل
fast Fourier transform	محاسبه‌ی سریع تبدیل فوریه
prefix constraint	محدودیت پیشوندی

pivot	محور
girth of the graph	محیط گراف
mode	مد
circuit	مدار
Eulerian circuit	مدار اویلری
simple circuit	مدار ساده
uniform model	مدل یک‌نواخت
sort	مرتب‌سازی
mergesort	مرتب‌سازی ادغامی
selection sort	مرتب‌سازی با انتخاب
radix sort	مرتب‌سازی بر اساس مرتبه
radix-exchange sort	مرتب‌سازی تعویض مرتبه
in-place sorting	مرتب‌سازی درجا
insertion sort	مرتب‌سازی درجی
quicksort	مرتب‌سازی سریع
bucket sort	مرتب‌سازی سطلی
straight-radix sort	مرتب‌سازی گام به گام بر اساس مرتبه
lexicographic sort	مرتب‌سازی واژه‌نامه‌ای یا الفبایی
heapsort	مرتب‌سازی هرمی
cyclically sorted	مرتب‌شده‌ی چرخشی
order statistics (selection)	مرتبه‌ی آماری (گزینش)
order of magnitude	مرتبه‌ی بزرگی
open problem	مسأله‌ی باز یا حل‌نشده
partition problem	مسأله‌ی بخش‌بندی
decision problem	مسأله‌ی تصمیم‌گیری
celebrity problem	مسأله‌ی ستاره‌ی مشهور
network-flow problem	مسأله‌ی شبکه‌ی شاره
knapsack problem	مسأله‌ی کوله‌پشتی
weighted selection problem	مسأله‌ی گزینش وزن‌دار
bottleneck problem	مسأله‌ی گلوگاه
skyline problem	مسأله‌ی نمای افقی

equivalence problem	مسأله‌ی هم‌ارزی
planarity	مسطح بودن
path	مسیر
augmenting path	مسیر افزایشی یا افزاینده
closed path	مسیر بسته
simple path	مسیر ساده
vertex disjoint paths	مسیرهای با رأس‌های جداازهم
edge-disjoint paths	مسیرهای با یال‌های جداازهم
alternating path	مسیر یک‌درمیان
characteristic equation	معادله‌ی مشخصه
valid	معتبر، درست
the towers of Hanoi puzzle	معمای برج‌های هانوی
heuristic	مکاشفه‌ای، اکتشافی
intuitive	ملموس، شهودی
component	مؤلفه
biconnected component	مؤلفه‌ی دوهمبند
strongly connected component	مؤلفه‌ی قویاً همبند
connected component	مؤلفه‌ی همبند
median	میانه
field	میدان
data field	میدان داده‌ای
nonbiconnected	نادوهمبند
candidate	نامزد
articulation point	نقطه‌ی پیوند
breakpoint	نقطه‌ی گسست
one-to-one mapping	نگاشت یک‌به‌یک
parent	والد
face	وجه
heap	هرم
maze	هزارتو
kernel	هسته

connected	همبند
sibling	هم‌شیر
edge	یال
cross edge	یال جانبی
forward edge	یال جلورو
back edge	یال عقب‌رو
one-dimensional	یک‌بعدی، تک‌بعدی
injective	یک‌به‌یک
isomorphism	یک‌ریختی