بخش اول مبانی

شامل فصل‌های :

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

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

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

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

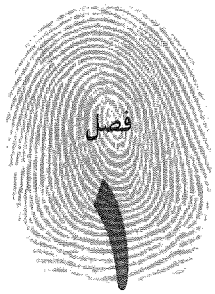
این نماد در فصل ۳ به طور دقیق تعریف می‌شود، که در آن جا به آن نماد حدی می‌گوییم. این فصل با نمادهای حدی متعددی آغاز می‌شود، که از آن‌ها برای تعیین کران بالا و/یا پایین زمان اجرای

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

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

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

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



نقش الگوریتم‌ها در محاسبات

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

۱-۱ الگوریتم‌ها

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

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

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

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

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

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

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

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

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

چه نوع مسائلی با الگوریتم حل می‌شوند؟

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

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

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

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

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

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

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

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

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

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

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

ساختمان‌های داده

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

تکنیک‌ها

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

مسائل سخت

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

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

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

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

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

محاسبه‌ی موازی

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

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

تمرین‌ها

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

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

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

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

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

۱-۲ الگوریتم به عنوان یک فناوری

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

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

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

کارایی

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

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

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

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

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

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

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

الگوریتم‌ها و فناوری‌های دیگر

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

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

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

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

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

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

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

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

تمرین‌ها

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

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

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

مسائل

۱-۱ *مقایسه‌ی زمان‌های اجرا*

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

مسائل

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

آغاز

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

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

۲-۱ مرتب‌سازی درجی

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

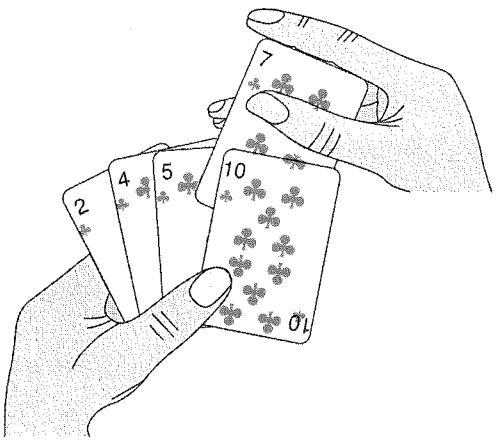
- **ورودی:** دنباله‌ای از n عدد $\langle a_1, a_2, \ldots, a_n \rangle$.

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

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

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

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



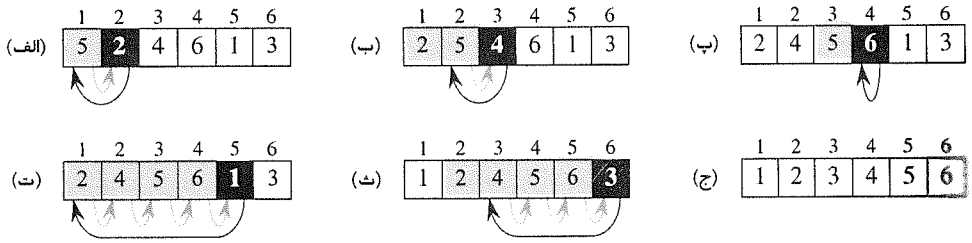
شکل ۲-۱ مرتب‌سازی دسته‌ی کارت‌ها با استفاده از روش درجی.

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

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

ثابت‌های حلقه و درستی مرتب‌سازی درجی

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



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

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

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

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

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

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

اجازه دهید ببینیم چطور این خصوصیات برای مرتب‌سازی درجی برقرارند.

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

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

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

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

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

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

قراردادهای شبه‌کد

در شبه‌کدهای خود از قراردادهای زیر استفاده خواهیم کرد.

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

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

---

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

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

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

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

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

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

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

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

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

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

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

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

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

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

تمرین‌ها

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

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

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

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

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

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

۲-۲ تحلیل الگوریتم‌ها

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

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

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

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

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

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

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

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

تحلیل مرتب‌سازی درجی

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

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

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

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

---

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

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

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

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

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

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

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

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

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

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

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

و

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

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

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

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

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

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

تحلیل بدترین حالت و حالت متوسط

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

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

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

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

مرتبه‌ی رشد

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

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

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

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

تمرین‌ها

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

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

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

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

۲-۳ طراحی الگوریتم‌ها

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

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

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

۲-۳-۱ رویکرد تقسیم و حل

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

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

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

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

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

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

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

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

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

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

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

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

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

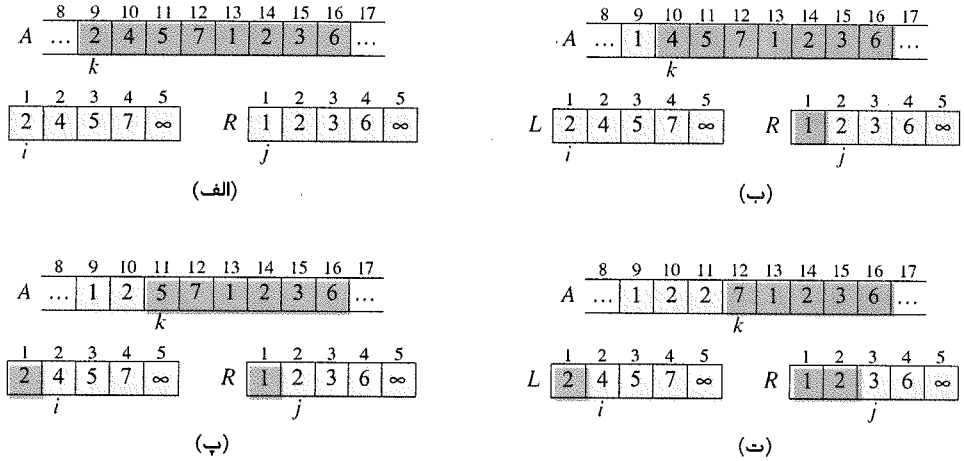
می‌کند، و خط ۲ طول زیرآرایه‌ی $A[q+۱..r]$، یعنی $n_۲$ را. سپس در خط ۳ آرایه‌های L و R (چپ و راست) را با طول‌های $n_۱+۱$ و $n_۲+۱$ می‌سازیم؛ فضای اضافی در هر آرایه مقدار نگهبان را ذخیره خواهد کرد. حلقه‌ی for در خطوط ۴-۵ زیرآرایه‌ی $A[p..q]$ را در $L[۱..n_۱]$ کپی می‌کند، و حلقه‌ی for خطوط ۶-۷، زیرآرایه‌ی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

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

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

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

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



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

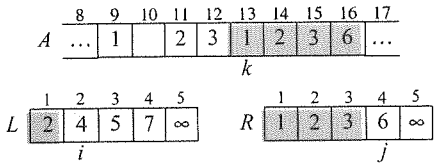
نشان می‌دهیم رویه‌ی MERGE در زمان $\theta(n)$ اجرا می‌شود، که در آن $n = r - p + 1$. توجه کنید که خطوط ۱-۳ و ۸-۱۱ در زمان ثابت اجرا می‌شوند، زمان اجرای حلقه‌ی for در خطوط ۴-۷ برابر است با $\theta(n_1 + n_2) = \theta(n)$[1]، حلقه‌ی for خطوط ۱۲-۱۷ تعداد n بار تکرار می‌شود، و هر بار تکرار به زمان ثابت نیاز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

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

---

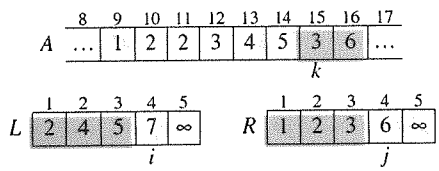
[1] در فصل ۳ خواهیم دید که چگونه با تساوی‌هایی که حاوی نماد θ هستند برخورد کنیم.

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



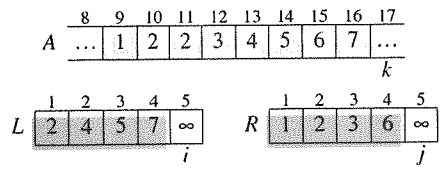
(ج)

(ث)



(ح)

(چ)



(خ)

شکل ۲-۳ ادامه

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

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

۲-۳-۲ تحلیل الگوریتم‌های تقسیم و حل

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

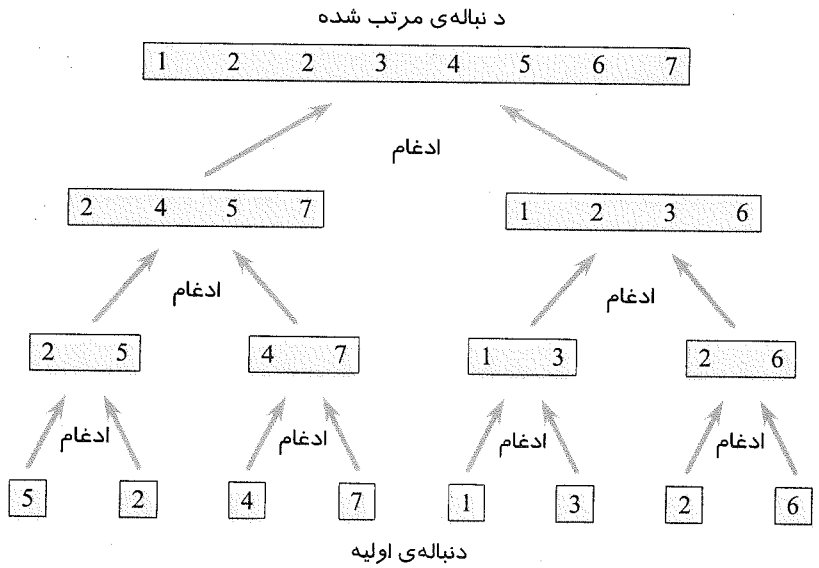
مقداردهی q با $\lfloor (p+r)/2 \rfloor$، زیرآرایه‌های $A[p..q]$ و $A[q+1..r]$ را با اندازه‌های $\lceil n/2 \rceil$ و $\lfloor n/2 \rfloor$ تولید می‌کند، این است که چهار حالتی را که از زوج یا فرد بودن p یا r به وجود می‌آیند، بررسی کنیم.

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

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

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

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



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

تحلیل مرتب‌سازی ادغامی

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

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

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

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

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

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

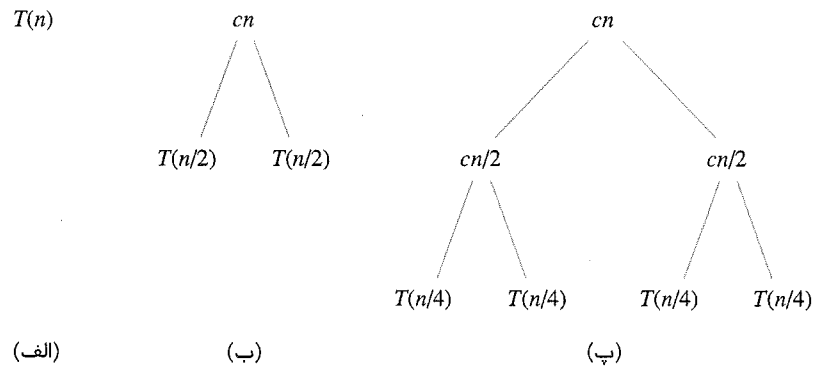
برای درک این که چرا جواب رابطه‌ی ۲-۱ عبارت است از $T(n) = \theta(n \lg n)$ احتیاجی به قضیه‌ی اصلی نداریم. اجازه دهید معادله‌ی بازگشتی ۲-۱ را به صورت زیر بازنویسی کنیم:

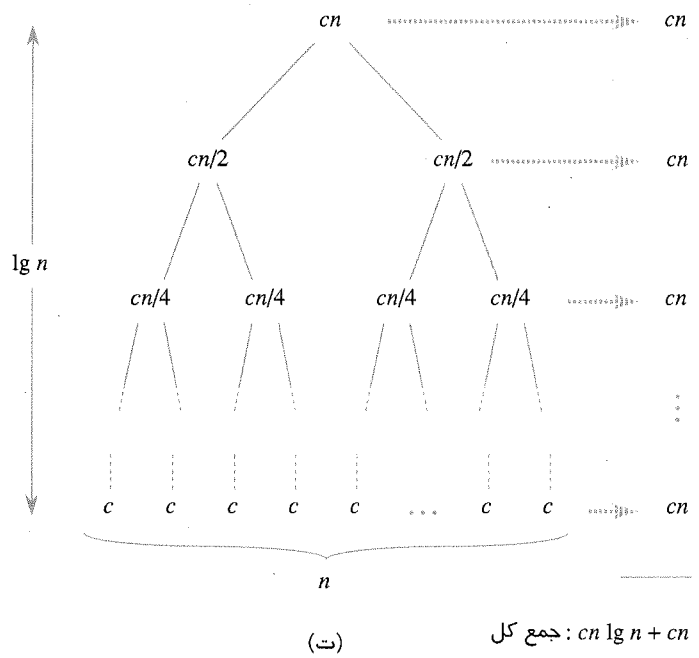
$$T(n) = \begin{cases} c & \text{اگر } n = ۱ \\ ۲T(n/۲) + cn & \text{اگر } n > ۱ \end{cases} \qquad (۲-۲)$$

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

---

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





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

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

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

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

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

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

تمرین‌ها

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

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

کار را به پایان رساند.

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

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

برابر است با $T(n)=n\lg n$.

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

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

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

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

مسائل

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

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

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

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

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

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

IV. در عمل k باید چطور انتخاب شود؟

۲-۳ *درستی مرتب‌سازی حبابی*

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

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

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

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

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

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

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

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

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

۲-۳ درستی قانون هورنر (Horner's rule)

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

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

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

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

I. زمان اجرای این قطعه کد بر حسب نماد θ چقدر است؟

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

III. ثابت حلقه‌ی زیر را در نظر بگیرید.

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

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

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

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

۲-۴ وارونگی‌ها

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

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

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

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

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

رشد توابع

۳-۰ مقدمه

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

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

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

۱-۳ نمادهای حدی

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

نمادهای حدی، توابع، و زمان‌های اجرا

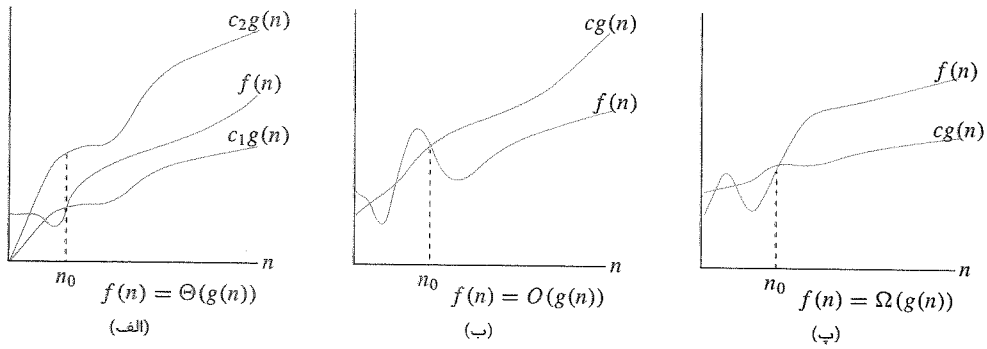
معمولاً از نمادهای حدی برای توصیف زمان اجرای الگوریتم‌ها استفاده می‌کنیم، همان طور که قبلا در مورد بدترین حالت زمان اجرای زمان اجرای $\theta(n^2)$ برای مرتب‌سازی درجی این کار را کردیم. ولی باید بدانید که نمادهای حدی در واقع برای توابع به کار می‌روند. به خاطر بیاورید که بدترین حالت زمان اجرای مرتب‌سازی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

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

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

نماد θ

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

نماد O

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

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

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

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

---

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

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

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

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

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

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

نماد Ω

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

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

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

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

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

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

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

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

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

نمادهای حدی در تساوی‌ها و نامساوی‌ها

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

نماد o

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

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

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

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

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

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

نماد ω

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

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

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

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

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

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

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

مقایسه‌ی توابع

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

تعدی:

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

انعکاس پذیری:

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

تقارن:

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

تقارن ترانهاده:

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

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

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

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

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

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

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

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

تمرین‌ها

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

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

$$(n+a)^b=\theta(n^b) \qquad (۲-۳)$$

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

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

۳-۱-۵ قضیه‌ی ۳-۱ را اثبات کنید.

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

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

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

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

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

۳-۲ نمادهای استاندارد و توابع متعارف

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

یکنواختی (monotonicity)

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

کف و سقف

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

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

برای تمام اعداد صحیح n،

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

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

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

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

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

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

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

حساب پیمانه‌ای (Modular arithmetic)

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

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

نتیجه می‌شود که

$$0 \le a \bmod n < n \qquad (۹-۳)$$

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

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

چندجمله‌ای‌ها

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

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

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

توابع نمایی

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

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

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

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

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

که به کمک آن می‌توانیم نتیجه بگیریم

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

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

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

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

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

$$e^x \geq ۱ + x \qquad (۱۲-۳)$$

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

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

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

$$e^x = ۱ + x + \theta(x^۲)$$

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

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

لگاریتم‌ها

در این کتاب، از نمادهای زیر استفاده خواهیم کرد:

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

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

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

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

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

که در تمام تساوی‌های بالا پایه‌ی لگاریتم‌ها ۱ نیستند.

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

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

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

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

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

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

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

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

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

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

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

فاکتوریل

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

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

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

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

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

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

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

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

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

که در آن

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

تکرار توابع

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

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

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

تابع لگاریتم تکراری

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

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

تابع لگاریتم تکراری رشد به شدت کندی دارد:

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

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

اعداد فیبوناچی

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

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

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

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

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

$$x^۲ = x + ۱ \qquad (۲۳-۳)$$

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

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

به خصوص داریم:

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

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

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

که نتیجه می‌دهد:

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

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

تمرین‌ها

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

۳-۲-۲ تساوی ۳-۱۶ را اثبات کنید.

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

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

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

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

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

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

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

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

مسائل

۳-۱ ***رفتار حدی چندجمله‌ای‌ها***

فرض کنید

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

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

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

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

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

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

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

۳-۲ رشد حدی نسبی

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

A	B	O	o	Ω	ω	θ
$\lg^k n$	n^ϵ					
n^k	c^n					
$\sqrt{n}$	$n^{\sin n}$					
2^n	$2^{n/2}$					
$n^{\lg c}$	$c^{\lg n}$					
$\lg(n!)$	$\lg(n^n)$					

۳-۳ درجه‌بندی نرخ رشد حدی توابع

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

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

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

۳-۴ خصوصیات نمادهای حدی

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

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

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

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

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

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

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

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

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

۳-۵ نسخه‌های مختلف O و Ω

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

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

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

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

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

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

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

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

۳-۶ *توابع تکراری*

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

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

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

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

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