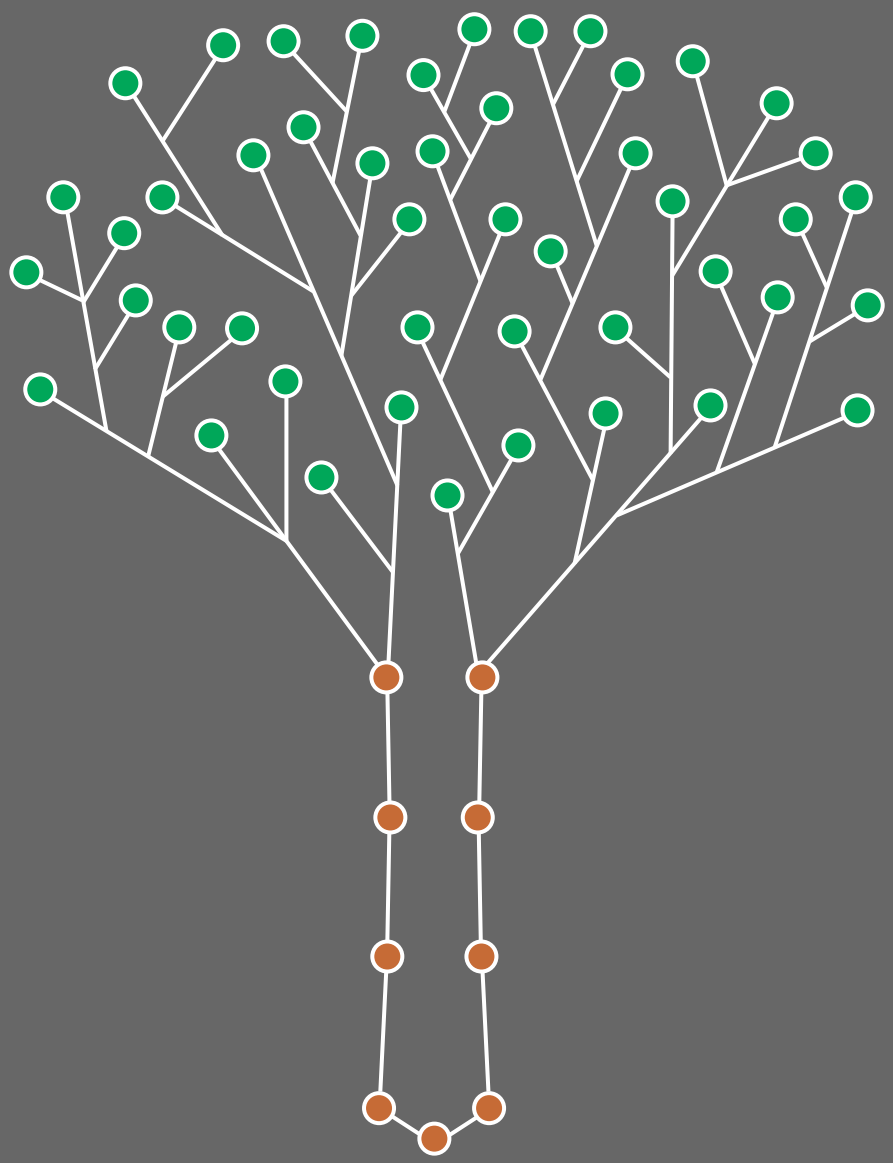
پرسش و پاسخ داده ساختارها



تالیف

مسعود فلاح پور

به نام یزدان نیک اندیش

پرسش و پاسخ داده ساختارها

تالیف

مسعود فلاح پور

آبان ۱۳۹۳

نگارش ۰/۲

مجوز

تقدیم به

پدر و مادرم؛ که همواره یار و یاورم بوده اند

فهرست مطالب

فهرست الگوریتم‌ها

پیش‌گفتار

در علوم کامپیوتر درس‌هایی وجود دارند که از آنها به عنوان درس‌هایی بنیادین یاد می‌شود. برخی از این درس‌ها عبارت‌اند از: داده‌ساختارها، تجزیه و تحلیل الگوریتم‌ها، طراحی کامپایلر و نظریه‌ی زبان‌ها و ماشین‌ها.

از میان این درس‌های بنیادین یکی از مهمترین آنها، درس داده‌ساختارها است که به عنوان پیشنیازی برای درس‌هایی همچون تجزیه و تحلیل الگوریتم‌ها و سیستم‌های عامل نیز مطرح است. اگر داده‌ساختار را به این صورت تعریف کنیم که «یک داده‌ساختار روشی برای ذخیره و سازماندهی داده‌ها است به طوریکه بازیابی و/یا تغییر داده‌ها به سادگی و با کارایی بالا انجام شود» آنگاه می‌توان به تعریفی از درس داده‌ساختارها نیز رسید. در درس داده‌ساختارها به بررسی دقیق و موشکافانه‌ی انواع داده‌ساختارها و چگونگی پیاده‌سازی آنها در یک زبان برنامه‌نویسی پرداخته می‌شود.

به دلیل اهمیت درس داده‌ساختارها کتاب‌های مختلفی در مورد آن نوشته شده است که بسیاری از آنها دارای قالب کم و بیش یکسانی هستند. قالب کلی این کتاب‌ها به این شکل است که در هر فصل از کتاب ابتدا به معرفی و بررسی یک داده‌ساختار خاص پرداخته شده و در انتهای فصل تمریناتی مرتبط با آن داده‌ساختار ارائه می‌شود. در کتاب حاضر سعی شده است از قالبی متفاوت استفاده شود.

در این کتاب فرض بر این است که خواننده با مباحث مختلف داده‌ساختارها آشنایی نسبی دارد و در نتیجه هر فصل از این کتاب دارای بخش نخست کتابهای معمول، یعنی معرفی و بررسی یک داده‌ساختار، نیست. تمرکز این کتاب بر روی مطرح کردن تعدادی سوال در مورد هر یک از انواع داده‌ساختارها و دادن پاسخ گام به گام و تشریحی به هر یک از سوالات است. به بیانی دیگر می‌توان قالب این کتاب را به صورت پرسش و پاسخ در نظر گرفت که به خواننده کمک می‌کند تا فهم عمیقتری از داده‌ساختارهای مختلف به دست آورد.

در نگارش حاضر، تنها فصل‌های اول، دوم و پنجم در کتاب گنجانده شده‌اند و سایر فصول پس از آماده‌سازی و کسب اطمینان از کیفیت علمی و ظاهری آنها در نگارش‌های بعدی به کتاب اضافه خواهند شد.

متن کتاب با استفاده از سیستم حروفچینی لاتک ، بسته‌ی زی‌پرشین و ویرایشگر bidiTexmaker آماده شده است. برای متن پارسی از قلم XB Niloofar و برای کلمات انگلیسی و شبه‌کدها از قلم Computer Modern استفاده شده است. برای طراحی جلد کتاب از نرم‌افزار Corel DRAW و برای رسم شکل‌ها از بسته‌ی PSTricks و نرم‌افزارهای Corel DRAW و LaTeXDraw استفاده شده است. برای دسترسی به متن خام کتاب می‌توانید به نشانی `https://github.com/MasoodFallahpoor/DS-Book` مراجعه کنید.

در آماده‌سازی این کتاب تلاش شده است تا چه از نظر علمی و چه از نظر ظاهری کتابی شایسته و خالی از خطا به خوانندگان تقدیم شود. اما از آنجایی که هیچ کتابی نمی‌تواند به طور کامل از خطا در امان باشد از این رو از شما خواننده‌ی گرامی خواهشمندم در صورت مشاهده هرگونه خطای املایی، نگارشی و یا علمی به نشانی `masood.fallahpoor@gmail.com` اطلاع دهید تا خطای موجود در ویرایش‌های بعدی کتاب رفع شود.

مسعود فلاح‌پور
آبان ۱۳۹۳

درباره مولف

مسعود فلاح‌پور در سال ۱۳۶۷ در تهران متولد شد و تحصیلات اولیه خود را در همین شهر پشت سر گذاشت. او در سال ۱۳۸۷ مدرک کاردانی و در سال ۱۳۹۰ مدرک کارشناسی ناپیوسته خود را از دانشکده فنی شماره دو تهران (شهید شمسی‌پور) دریافت کرد. مسعود دارای مدرک کارشناسی ارشد مهندسی کامپیوتر در گرایش مهندسی نرم‌افزار از دانشکده مهندسی برق و کامپیوتر دانشگاه شهید بهشتی است.

مسعود به هر دو جنبه‌ی نظری و عملی علم کامپیوتر علاقه‌مند است. از جمله علاقه‌مندی‌های او در بخش نظری می‌توان به سیستم‌های عامل، داده‌ساختارها، طراحی الگوریتم‌ها و طراحی کامپایلر اشاره کرد. علاقه به سیستم عامل گنو/لینوکس، برنامه‌نویسی به زبان‌های جاوا و سی و همچنین برنامه‌نویسی اندروید از جمله علاقه‌مندی‌های او در بخش عملی است.

قدردانی

قدردانی و نام بردن از تمام افرادی که در به ثمر رسیدن این کتاب نقش داشته‌اند کاری است بس دشوار. به همین جهت فقط از برخی افراد نام برده خواهد شد.

بر خود لازم می‌دانم از آقای مهدی جوانمرد که ایده اولیه نوشتن این کتاب را مطرح کردند و همچنین جمع‌آوری بخشی از سوالات هر فصل را بر عهده داشتند صمیمانه سپاسگزاری کنم.

همچنین قدردانی می‌کنم از استاد گرامی، دکتر محسن ابراهیمی مقدم، که فهم دقیق و عمیق بسیاری از مفاهیم داده‌ساختارها را مدیون ایشان هستم.

در نهایت نیز از تلاش‌های چندین و چند ساله‌ی آقای وفا کارن پَهلَو برای توسعه‌ی بسته‌ی زی‌پرشین کمال قدردانی را دارم زیرا با خلق این بسته کمک شایانی به جامعه‌ی دانشگاهی ایران کرده‌اند.

قواعد شبه‌کد

برای بیان الگوریتم‌های بیان شده در کتاب، به جای استفاده از یک زبان برنامه‌نویسی خاص، از شبه‌کد استفاده شده است. با استفاده از شبه‌کد می‌توان الگوریتم‌ها را به شکلی ساده بیان کرد و از بیان جزئیات غیر ضروری خودداری کرد. در ادامه توضیحاتی در مورد کلیات شبه‌کد استفاده شده در کتاب بیان خواهد شد.

توضیحات

اگر در قسمتی از شبه کد نیاز به توضیح وجود داشته باشد، مانند زبان C++ از دو علامت اسلش پشت سرهم برای شروع توضیح استفاده می‌شود. در ادامه نمونه‌ای از یک توضیح آورده شده است.

```
// This is a comment
```

زیربرنامه‌ها

تمامی الگوریتم‌های کتاب به صورت زیربرنامه تعریف می‌شوند. یک زیربرنامه دارای دو نوع است: تابع و رویه. اگر زیربرنامه بخواهد مقداری را به عنوان خروجی بازگرداند آنگاه از تابع استفاده می‌کنیم و اگر مقداری را برنگرداند از رویه استفاده خواهیم کرد.

تعریف یک تابع با کلمه کلیدی function آغاز می‌شود. سپس نام تابع بیان می‌شود و در صورتی که تابع دارای ورودی باشد، ورودی‌های تابع در داخل پرانتز آورده می‌شوند. در ادامه‌ی تعریف تابع، بدنه تابع شروع می‌شود و در انتها مقداری به عنوان خروجی تابع توسط دستور return برگشت داده می‌شود. عبارت end function نیز خاتمه تعریف تابع را نشان می‌دهد. شکل کلی تعریف یک تابع در ادامه نشان داده شده است.

```
function FunctionName(param1, param2, ... , paramN)
    // body of function
    return result
end function
```

شکل کلی تعریف یک رویه هم مانند یک تابع است با این تفاوت‌ها که تعریف یک رویه با کلمه کلیدی procedure آغاز می‌شود، مقداری توسط رویه بازگردانده نمی‌شود و همچنین خاتمه رویه توسط عبارت end procedure مشخص می‌شود.

متغیرها

برای تعریف متغیرها از قالب $VarName : VarType$ استفاده می‌شود. برای مثال اگر بخواهیم متغیر i را از نوع عدد صحیح تعریف کنیم به صورت $i : integer$ عمل می‌کنیم.

متغیرهای مورد استفاده در شبه‌کدها در اکثر مواقع به صورت صریح تعریف نمی‌شوند و فرض بر این است که با اولین استفاده از یک متغیر، آن متغیر به صورت ضمنی تعریف نیز می‌شود. در حالت تعریف ضمنی متغیرها، با توجه به شبه‌کدی که متغیر در آن مورد استفاده قرار گرفته است، به راحتی می‌توان به نوع آن نیز پی برد.

آرایه‌ها

اندیس تمامی آرایه‌ها از عدد یک آغاز می‌شود مگر اینکه در یک شبه‌کد صراحتاً چیز دیگری بیان شود. برای دسترسی به خانه‌ی i ام آرایه یک بعدی A از قالب $A[i]$ و برای دسترسی به عنصر سطر i ام و ستون j ام آرایه دو بعدی B از قالب $B[i, j]$ استفاده می‌شود.

اگر متغیر A نشان دهنده یک آرایه یک بعدی باشد آنگاه طول این آرایه در خصیصه $length$ آن قرار دارد و برای دسترسی به آن از قالب $A.length$ استفاده می‌شود. اگر A یک آرایه دو بعدی باشد تعدادی سطرهای آن در خصیصه row و تعداد ستون‌های آن در خصیصه $column$ قرار دارد و برای دسترسی به آنها به ترتیب از قالب $A.rows$ و $A.columns$ استفاده می‌شود.

جهت اشاره به بازه‌ای از یک آرایه از قالب $A[i\,..\,j]$ استفاده می‌شود که در آن i اندیس شروع بازه و j اندیس پایان بازه است.

حلقه‌ها

برای تکرار یک تا تعدادی دستور از دو نوع حلقه استفاده خواهد شد: حلقه for و حلقه while. از ساختار حلقه for زمانی استفاده می‌شود که تعداد تکرار بدنه حلقه از قبل مشخص باشد و از حلقه while زمانی استفاده می‌شود که تعداد تکرار بدنه حلقه از قبل معلوم نباشد.

تعریف حلقه for با کلمه کلیدی for آغاز می‌شود. سپس مقدار اولیه شمارنده حلقه به متغیر شمارنده حلقه انتساب داده می‌شود و پس از کلمه کلیدی to مقدار نهایی شمارنده حلقه مشخص می‌شود. بعد از تعریف سرآیند حلقه، بدنه حلقه تعریف می‌دهد و در نهایت عبارت end for پایان حلقه را نشان می‌شود. در ادامه شکل کلی تعریف حلقه for نشان داده شده است.

```
for counter = startValue to endValue
    // body of for loop
end for
```

با هر بار اجرای این حلقه یک واحد به متغیر شمارنده حلقه افزوده می‌شود و بدنه حلقه تا زمانی اجرا می‌شود که شرط $counter \leqslant endValue$ برقرار باشد. اگر بخواهیم شمارنده حلقه به جای افزایش، کاهش یابد آنگاه به جای کلمه کلیدی to از کلمه کلیدی downto استفاده می‌شود.

تعریف حلقه while با کلمه کلیدی while آغاز می‌شود. سپس یک عبارت منطقی قرار می‌گیرد و تا زمانی که عبارت منطقی برقرار باشد بدنه حلقه اجرا می‌شود. خاتمه حلقه while نیز با عبارت end while نشان

داده می‌شود.

```
while booleanExpression
    // body of while loop
end while
```

دستورات شرطی

برای شروع یک دستور شرطی از کلمه کلیدی if استفاده می‌شود و در ادامه یک عبارت منطقی آورده می‌شود. اگر عبارت منطقی درست باشد دستورات بخش اول و در غیر این صورت دستورات بخش دوم اجرا می‌شوند. خاتمه تعریف دستور شرطی نیز با عبارت end if نشان داده می‌شود. برای تعریف یک دستور شرطی از قالب کلی زیر استفاده می‌شود.

```
if booleanExpression
    // statements to be executed when booleanExpression is TRUE
else
    // statements to be executed when booleanExpression is FALSE
end if
```

وجود بخش else اجباری نیست و این یعنی اگر این بخش وجود نداشته باشد و شرط دستور شرطی برقرار نباشد آنگاه بدنه دستور شرطی اجرا نخواهد شد.

دستور return

با اجرای دستور return اجرای زیربرنامه بلافاصله پایان می‌یابد. از این دستور در صورت نیاز برای بازگرداندن مقدار یا مقادیری به عنوان خروجی یک تابع نیز می‌توان استفاده کرد. در ادامه شکل‌های مختلف دستور return آورده شده است.

```
return
return result
return (result1, result2, ... , resultN)
```

در صورتی که شکل خروجی تابعی مانند $\textsc{Func}(A, B)$ مانند حالت سوم دستور return باشد آنگاه از شکل زیر برای دریافت تمامی خروجی‌های آن استفاده خواهد شد. با اجرای دستور زیر مقدار result1 در var1 قرار می‌گیرد، result2 در var2 قرار می‌گیرد و به همین ترتیب تا resultN که در varN قرار می‌گیرد.

```
var1, var2, ... , varN=Func(A, B)
```

عملگرها

عملگرهای منطقی مورد استفاده در الگوریتم‌ها عبارت‌اند از $>$، $<$، $\leq$، $\geq$، $\neq$ و $==$ که از عملگر آخر برای بررسی تساوی دو مقدار استفاده می‌شود.

برای ترکیب عبارات منطقی از عملگرهای and، or و not استفاده می‌شود. جهت انتساب مقداری به یک متغیر از عملگر $=$ استفاده می‌شود.

برای انجام چهار عمل اصلی ریاضی از عملگرهای $+$، $-$، $\times$ و $/$ استفاده می‌شود. همچنین از عملگر mod به عنوان عملگر باقیمانده استفاده می‌شود.

اشاره‌گرها

برای تخصیص فضا به یک اشاره‌گر از زیربرنامه NEW استفاده می‌شود. برای مثال اگر p یک اشاره‌گر باشد و بخواهیم به آن فضا اختصاص دهیم باید زیربرنامه NEW را به صورت $\text{NEW}(p)$ فراخوانی کنیم.

اگر فرض کنیم p یک اشاره‌گر باشد که به یک ساختار[1] اشاره دارد و این ساختار دارای فیلدی به نام $next$ است آنگاه از قالب $p.next$ برای دسترسی به محتوای فیلد $next$ استفاده می‌شود.

برای بیان مقدار تهی در زمان کار با اشاره‌گرها از ثابت NULL استفاده می‌شود.

[1] منظور از ساختار، چیزی مانند struct در زبان C یا record در زبان پاسکال است.

فصل ۱

مرتبه‌ی زمانی

۱.۱ مقدمه

در بررسی داده‌ساختارها و الگوریتم‌ها معمولا دو موضوع حافظه و زمان مصرفی مورد توجه قرار می‌گیرند که از میان این دو زمان مصرفی از اهمیت بیشتری برخوردار است. در حالت ایده‌آل همیشه به دنبال داده‌ساختارها و الگوریتم‌هایی هستیم که با کمترین حافظه و بیشترین سرعت اجرا شوند.

مرتبه‌ی زمانی ابزاری است که به کمک آن می‌توان در مورد زمان اجرای عملیات داده‌ساختارها و الگوریتم‌ها بحث کرد و آنها را از نظر زمان اجرا با یکدیگر مقایسه کرد. با توجه به اهمیت مبحث مرتبه‌ی زمانی از آن به عنوان سنگ بنای دروس داده‌ساختارها و طراحی الگوریتم یاد می‌شود.

با توجه به گستردگی و اهمیت مبحث مرتبه‌ی زمانی این فصل به بخش‌هایی تقسیم شده است و سوالات هر بخش به بررسی جنبه‌ای از مرتبه‌ی زمانی می‌پردازد.

۲.۱ منابع مطالعاتی

معمولا در تمامی کتاب‌هایی که به بحث و بررسی داده‌ساختارها و طراحی الگوریتم‌ها می‌پردازند، فصل یا فصولی به معرفی مرتبه‌ی زمانی اختصاص داده می‌شود. برای مطالعه در مورد مفهوم مرتبه‌ی زمانی، نمادهای مجانبی و بزرگی توابع می‌توانید به فصل سوم [۱] و فصل اول [۲] مراجعه کنید. برای کسب اطلاعات در مورد الگوریتم‌های بازگشتی و به دست آوردن مرتبه‌ی زمانی چنین الگوریتم‌هایی می‌توانید به فصل چهارم [۱] و فصل دوم [۲] مراجعه کنید.

۳.۱ مفهوم مرتبه‌ی زمانی

◄ **سوال ۱.** آرایه‌ی A را در نظر بگیرید. هر یک از خانه‌های آرایه‌ی A شامل عددی صحیح در بازه‌ی [۱, ۲۵۶] است. الگوریتم (۱.۱) تعداد تکرار هر عدد در آرایه‌ی A را به دست می‌آورد.

الگوریتم ۱.۱ شمارش تعداد تکرار اعداد در یک آرایه یک بعدی

```
procedure CountRepetition(A)
    let C[1..256] be a new array of type integer
    n = A.length
    for i = 1 to 256
        C[i] = 0
    end for
    for i = 1 to n
        C[A[i]] = C[A[i]] + 1
    end for
end procedure
```

با در نظر گرفتن الگوریتم (۱.۱) به موارد زیر پاسخ دهید.

آ. با توجه به اندازه‌ی ورودی[۱]، چه تعداد عمل جمع و چه تعداد عمل انتساب در این الگوریتم انجام می‌شود؟

ب. اگر مجموع انتساب‌ها و جمع‌های انجام شده را به عنوان کل اعمال انجام شده در نظر بگیریم آنگاه چه درصدی از کل اعمال، هنگامی که آرایه‌ی A دارای ۵۰۰ خانه است، مربوط به حلقه اول و چه درصدی مربوط به حلقه دوم است؟ اگر آرایه‌ی A دارای ۵۰۰۰۰ خانه باشد، این درصدها چگونه خواهند بود؟

ج. کدامیک از دو حلقه‌ی موجود در الگوریتم تاثیر بیشتری در مرتبه‌ی زمانی الگوریتم دارد؟

◃ **پاسخ سوال ۱.**

آ. بدنه‌ی حلقه‌ی اول ۲۵۶ بار اجرا و در هر بار اجرا یک عمل انتساب انجام می‌شود. بدنه‌ی حلقه‌ی دوم n بار اجرا می‌شود و در هر بار اجرا یک عمل انتساب و یک عمل جمع صورت می‌پذیرد. در نتیجه تعداد انتساب‌های انجام شده در حلقه دوم n و تعداد جمع‌ها نیز برابر با n است. به این ترتیب می‌توان گفت برای دو حلقه در مجموع $n+۲۵۶$ عمل انتساب و n عمل جمع انجام می‌شود. به عبارت دیگر در الگوریتم (۱.۱) در مجموع $۲n+۵۱۲$ عمل انجام می‌شود که n همان تعداد خانه‌های آرایه‌ی ورودی است.

ب. اگر آرایه‌ی ورودی دارای ۵۰۰ خانه باشد آنگاه تعداد کل اعمال برابر با ۱۵۱۲ خواهد بود که از این تعداد، ۲۵۶ عمل متعلق به حلقه‌ی اول و ۱۲۵۶ عمل متعلق به حلقه‌ی دوم است. به این ترتیب می‌توان گفت به طور تقریبی ۱۷ درصد از کل اعمال انجام شده مربوط به حلقه‌ی اول و ۸۳ درصد مربوط به حلقه‌ی دوم است.

حال اگر فرض کنیم آرایه دارای ۵۰۰۰۰ خانه است آنگاه الگوریتم در کل ۱۰۰۵۱۲ عمل را انجام خواهد داد که از این تعداد دوباره ۲۵۶ عمل متعلق به حلقه‌ی اول است اما تعداد اعمال حلقه‌ی دوم به ۱۰۰۲۵۶ افزایش

[۱] منظور از اندازه‌ی ورودی همان تعداد خانه‌های آرایه‌ی A است.

می‌یابد. در این حالت به طور تقریبی ۰/۲۵ درصد از کل اعمال متعلق به حلقه‌ی اول و ۹۹/۷۵ درصد متعلق به حلقه‌ی دوم است.

ج. *تعداد* اعمال انجام شده در حلقه‌ی اول با افزایش اندازه‌ی ورودی تغییر نمی‌کند (تعداد اعمال این حلقه همواره ۲۵۶ است) اما *درصد* اعمال انجام شده در این حلقه، نسبت به کل اعمال، با افزایش اندازه‌ی ورودی کاهش می‌یابد. این یعنی اگر اندازه‌ی ورودی بسیار بزرگ باشد می‌توان اعمال انجام شده در حلقه‌ی اول را نادیده گرفت زیرا تعداد اعمال انجام شده در این حلقه تاثیری در تعیین مرتبه‌ی زمانی الگوریتم نخواهد داشت.

به این ترتیب می‌توان گفت برای تحلیل مرتبه‌ی زمانی یک الگوریتم می‌توان بدون در نظر گرفتن بخش‌هایی از الگوریتم که با تغییر اندازه‌ی ورودی زمان اجرای آنها تغییری نمی‌کند کار تحلیل را انجام داد. به بیان دیگر برای تحلیل مرتبه‌ی زمانی یک الگوریتم کافیست قسمت‌هایی از الگوریتم که زمان اجرای آنها به اندازه‌ی ورودی بستگی دارد مورد تحلیل قرار بگیرند.

◄ **سوال ۲.** چرا عبارت «زمان اجرای الگوریتم A حداقل از مرتبه $O(n^2)$ است» هیچ اطلاع مفیدی درباره‌ی زمان اجرای الگوریتم A در اختیار ما قرار نمی‌دهد؟

◁ **پاسخ سوال ۲.**

فرض کنید $T(n)$ نشان دهنده‌ی زمان اجرای الگوریتم A باشد. در این صورت عبارت مطرح شده در صورت سوال بیان می‌دارد که $T(n) \geq O(n^2)$. از آنجایی که تابعی مانند $f(n) = 0$ نیز عضو $O(n^2)$ است در نتیجه می‌توان گفت $T(n) \geq f(n)$ که این معادل است با $T(n) \geq 0$. از آنجایی که از قبل می‌دانیم زمان اجرای هر الگوریتم دلخواه مقداری غیر منفی است در نتیجه عبارت $T(n) \geq 0$ هیچ اطلاع مفیدی درباره‌ی زمان اجرای الگوریتم A در اختیار ما قرار نمی‌دهد.

◄ **سوال ۳.** دو کامپیوتر A و B را در اختیار داریم که اولی توانایی اجرای ده میلیارد دستور در ثانیه و دومی توانایی اجرای ده میلیون دستور در ثانیه را دارد. به بیانی دیگر کامپیوتر A از نظر محاسباتی هزار برابر از کامپیوتر B سریعتر است. لیست L به طول n حاوی اعداد صحیح را در اختیار داریم و قصد داریم آن را مرتب کنیم. دو الگوریتم مرتب‌سازی طراحی کرده‌ایم که الگوریتم اول با انجام $2n^2$ دستور و الگوریتم دوم با انجام $50n\lg n$ دستور می‌تواند لیست L را مرتب کند. الگوریتم اول را بر روی کامپیوتر A و الگوریتم دوم را بر روی کامپیوتر B اجرا می‌کنیم. اگر طول لیست L برابر با ده میلیون باشد آنگاه هر یک از این دو کامپیوتر به چه مقدار زمان برای مرتب‌سازی لیست L نیاز دارند؟ با مقایسه‌ی زمان‌های به دست آمده چه نتیجه‌ای می‌توان گرفت؟

◁ **پاسخ سوال ۳.**

زمان مورد نیاز کامپیوتر A برابر است با:

$$\frac{2 \times (10^7)^2}{10^{10}} = 20000 \text{ ثانیه}$$

و زمان مورد نیاز کامپیوتر B برابر است با:

$$\frac{50 \times 10^7 \times \lg 10^7}{10^7} \approx 1163 \text{ ثانیه}$$

با مقایسه‌ی زمان‌های به دست آمده به نتیجه‌ای جالب می‌رسیم. گرچه کامپیوتر A هزار برابر سریعتر از کامپیوتر B است اما چون مرتبه‌ی زمانی الگوریتمی که بر روی کامپیوتر A اجرا شد از مرتبه‌ی الگوریتم اجرا شده بر روی کامپیوتر B بزرگتر بود در نتیجه عملیات مرتب‌سازی بر روی کامپیوتر B بسیار سریع‌تر انجام شد. به این ترتیب می‌توان دریافت چرا همیشه دانشمندان علوم کامپیوتر به دنبال الگوریتم‌هایی با مرتبه‌ی زمانی پایین هستند!

◂ **سوال ۴.** برای حل مسئله‌ی P الگوریتم‌های A و B را در اختیار داریم. هر دو الگوریتم را بر روی یک کامپیوتر پیاده‌سازی کرده‌ایم. اگر اندازه‌ی ورودی مسئله‌ی P را با n نشان دهیم آنگاه الگوریتم A به $۸n^۲$ گام و الگوریتم B به $۶۴n\lg n$ گام برای حل مسئله‌ی P نیاز دارد. برای چه مقادیری از n الگوریتم A سریعتر از الگوریتم B مسئله‌ی P را حل می‌کند؟

◃ **پاسخ سوال ۴.**

برای رسیدن به پاسخ باید نامعادله‌ی $۸n^۲ < ۶۴n\lg n$ را حل کنیم. این نامعادله را به صورت زیر ساده می‌کنیم:

$$۸n^۲ < ۶۴n\lg n \Rightarrow n < ۸\lg n \Rightarrow ۲^{n/۸} < n$$

نامعادله‌ی فوق به ازای $۲ \leq n \leq ۴۳$ برقرار است. به این ترتیب می‌توان گفت گرچه مرتبه‌ی زمانی الگوریتم A از الگوریتم B بیشتر است اما این بدین معنی نیست که الگوریتم A همیشه کندتر از الگوریتم B عمل می‌کند.

۴.۱ نمادهای مجانبی و بزرگی توابع

◂ **سوال ۵.** آیا می‌توان به ازای توابع دلخواه f و g برقراری یکی از روابط $f(n) = O\big(g(n)\big)$ یا $f(n) = \Omega\big(g(n)\big)$ را اثبات کرد؟

◃ **پاسخ سوال ۵.**

خیر. ممکن است برای دو تابع دلخواه f و g نتوان درستی هیچ یک از دو رابطه‌ی $f(n) = O\big(g(n)\big)$ و $f(n) = \Omega\big(g(n)\big)$ را اثبات کرد. به طور مثال برای توابع $f(n) = n$ و $g(n) = n^{۱+\sin n}$ نمی‌توان این روابط را اثبات کرد زیرا مقدار تابع $g(n)$ در بازه‌ی $[۱, n^۲]$ نوسان می‌کند و این بدین معنی است که تابع $g(n)$ نه سقفی برای $f(n)$ و نه کفی برای آن است. در چنین حالتی گفته می‌شود که توابع $f(n)$ و $g(n)$ مقایسه‌پذیر نیستند.

◂ **سوال ۶.** اگر f و g توابع دلخواه غیرمنفی باشند، با توجه به تعریف نماد Θ، درستی عبارت زیر را ثابت کنید.

$$\max\Big(f(n), g(n)\Big) = \Theta\Big(f(n) + g(n)\Big)$$

◃ **پاسخ سوال ۶.**

ابتدا تابع $h(n)$، که همان تعریف تابع max است، را در نظر می‌گیریم:

$$h(n) = \begin{cases} f(n) & f(n) \geq g(n) \\ g(n) & f(n) < g(n) \end{cases} \tag{۱.۱}$$

با در نظر گرفتن تعریف (۱.۱) باید نشان دهیم $h(n) = \Theta\big(f(n)+g(n)\big)$. بدین منظور ابتدا نشان می‌دهیم $h(n) = O\big(f(n)+g(n)\big)$ و سپس نشان می‌دهیم $h(n) = \Omega\big(f(n)+g(n)\big)$.

با توجه به اینکه توابع f و g هر دو غیرمنفی هستند می‌توان نتیجه گرفت:

$$\begin{cases} f(n)+g(n) \geq f(n) \\ f(n)+g(n) \geq g(n) \end{cases} \implies f(n)+g(n) \geq h(n) \tag{۲.۱}$$

با در نظر گرفتن رابطه‌ی (۲.۱) و تعریف نماد مجانبی O خواهیم داشت:

$$h(n) \leq f(n)+g(n) \Rightarrow h(n) = O\Big(f(n)+g(n)\Big) \tag{۳.۱}$$

از طرف دیگر رابطه‌ی زیر نیز برقرار است:

$$\begin{cases} h(n) \geq f(n) \\ h(n) \geq g(n) \end{cases} \Rightarrow ۲h(n) \geq f(n)+g(n) \Rightarrow h(n) \geq \frac{۱}{۲}\Big(f(n)+g(n)\Big)$$

با در نظر گرفتن رابطه‌ی بالا و تعریف نماد مجانبی Ω داریم:

$$\frac{۱}{۲}\Big(f(n)+g(n)\Big) \leq h(n) \Rightarrow h(n) = \Omega\Big(f(n)+g(n)\Big) \tag{۴.۱}$$

از (۳.۱)، (۴.۱) و تعریف نماد Θ می‌توان به نتیجه‌ی دلخواه رسید:

$$h(n) = max\Big(f(n), g(n)\Big) = \Theta\Big(f(n)+g(n)\Big)$$

◄ **سوال ۷.** ثابت کنید برای هر دو عدد حقیقی و غیر منفی a و b داریم $(n+a)^b = \Theta(n^b)$.

◁ **پاسخ سوال ۷.**

بسط دوجمله‌ای نیوتن که در رابطه‌ی (۵.۱) نشان داده شده است را در نظر می‌گیریم.

$$(x+y)^k = \sum_{i=۰}^{k} \binom{k}{i} x^{k-i} y^i \tag{۵.۱}$$

اگر در (۵.۱) به ترتیب ابتدا n را جایگزین x ، سپس a را جایگزین y و در نهایت b را جایگزین k کنیم به رابطه‌ی (۶.۱) می‌رسیم.

$$(n+a)^b = \sum_{i=۰}^{b} \binom{b}{i} n^{b-i} a^i \tag{۶.۱}$$

با گسترش رابطه‌ی (۶.۱) خواهیم داشت:

$$(n+a)^b = \binom{b}{0}n^b + \binom{b}{1}n^{b-1}a^1 + \cdots + \binom{b}{b-1}n^1a^{b-1} + \binom{b}{b}a^b \tag{۷.۱}$$

اگر در (۷.۱) مقادیر ثابت را با c_r $(0 \leqslant r \leqslant b)$ نشان دهیم آنگاه می‌توان رابطه‌ی (۷.۱) را به صورت نشان داده شده در رابطه‌ی (۸.۱) بازنویسی کرد.

$$(n+a)^b = c_0n^b + c_1n^{b-1} + c_2n^{b-2} + \cdots + c_{r-1}n^1 + c_rn^0 \tag{۸.۱}$$

با توجه به رابطه‌ی (۸.۱) می‌توان نتیجه گرفت:

$$(n+a)^b \leq (b+1)n^b \tag{۹.۱}$$

از طرف دیگر برقراری رابطه‌ی (۱۰.۱) نیز بدیهی است.

$$n^b \leq (n+a)^b \tag{۱۰.۱}$$

با در نظر گرفتن روابط (۹.۱)، (۱۰.۱) و همچنین تعریف نماد Θ می‌توان به نتیجه‌ی زیر دست یافت:

$$(n+a)^b = \Theta(n^b)$$

به این ترتیب اثبات کامل است.

◄ **سوال ۸.** ثابت کنید رابطه‌ی $2^n < n! < n^n$ برای مقادیر بزرگ n برقرار است.

◃ **پاسخ سوال ۸.**

ابتدا نشان می‌دهیم نامساوی $2^n < n!$ برقرار است. برای این منظور نامساوی زیر، که درستی آن بدیهی است، را در نظر می‌گیریم:

$$\underbrace{\lg 2 + \lg 2 + \cdots + \lg 2}_{n} < \underbrace{\lg n + \lg(n-1) + \lg(n-2) + \cdots + \lg 1}_{n}$$

با خلاصه کردن نامساوی اخیر و گرفتن لگاریتم از طرفین آن داریم:

$$n \lg 2 < \lg n! \Rightarrow \lg 2^n < \lg n! \Rightarrow 2^n < n!$$

بدین صورت نشان دادیم 2^n از $n!$ کوچکتر است.

برای اثبات نامساوی $2^n < n!$ می‌توانستیم از تقریب استرلینگ[۲] نیز استفاده کنیم. این تقریب در رابطه‌ی (۱۱.۱) نشان داده شده است که در آن e نشان‌دهنده‌ی پایه‌ی لگاریتم طبیعی است.

$$n! = \frac{\sqrt{2n\pi}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)}{2^n} \tag{۱۱.۱}$$

[۲] Stirling's approximation

با در نظر گرفتن تقریب استرلینگ و استفاده از مفهوم حد می‌توان نوشت:

$$\begin{aligned}\lim_{n\to\infty}\frac{n!}{۲^n} &= \lim_{n\to\infty}\frac{\sqrt{۲n\pi}\left(\frac{n}{e}\right)^n\left(۱+\Theta\left(\frac{۱}{n}\right)\right)}{۲^n}\\ &= \lim_{n\to\infty}\sqrt{۲n\pi}\left(\frac{n}{۲e}\right)^n\left(۱+\Theta\left(\frac{۱}{n}\right)\right)\\ &= \infty\end{aligned}$$

چون حاصل حد بالا برابر با بی‌نهایت شد می‌توان گفت صورت کسر از مخرج آن بزرگتر است و این یعنی $۲^n < n!$.

در ادامه نشان خواهیم داد نامساوی $n! < n^n$ نیز برقرار است. درستی نامساوی (۱۲.۱) بدیهی است.

$$\underbrace{۱\times ۲\times ۳\times\cdots\times n}_{n} < \underbrace{n\times n\times n\times\cdots\times n}_{n} \tag{۱۲.۱}$$

اگر نامساوی (۱۲.۱) را به شکل خلاصه شده بنویسیم داریم $n! < n^n$.

اثبات اخیر را، مانند قسمت قبل، می‌توان با استفاده از تقریب استرلینگ هم انجام داد. بدین منظور خواهیم داشت:

$$\begin{aligned}\lim_{n\to\infty}\frac{n!}{n^n} &= \lim_{n\to\infty}\frac{\sqrt{۲n\pi}\left(\frac{n}{e}\right)^n\left(۱+\Theta\left(\frac{۱}{n}\right)\right)}{n^n}\\ &= \lim_{n\to\infty}\sqrt{۲n\pi}\left(\frac{n}{ne}\right)^n\left(۱+\Theta\left(\frac{۱}{n}\right)\right)\\ &= \lim_{n\to\infty}\frac{\sqrt{۲n\pi}\left(۱+\Theta\left(\frac{۱}{n}\right)\right)}{e^n}\\ &= ۰\end{aligned}$$

چون حاصل حد بالا برابر با صفر شد می‌توان گفت $n!$ از n^n کوچکتر است.

با توجه به اینکه درستی هر دو نامساوی $۲n < n!$ و $n! < n^n$ را اثبات کردیم در نتیجه می‌توان گفت رابطه‌ی $۲n < n! < n^n$ نیز برقرار است و به این ترتیب اثبات کامل است.

◄ **سوال ۹.** درستی رابطه‌ی $\lg n! = \Theta(n\lg n)$ را اثبات کنید.

◁ **پاسخ سوال ۹.**

به منظور اثبات درستی رابطه‌ی $\lg n! = \Theta(n\lg n)$ ابتدا درستی رابطه‌ی $\lg n! = O(n\lg n)$ و سپس درستی رابطه‌ی $\lg n! = \Omega(n\lg n)$ را اثبات می‌کنیم.

برای نشان دادن اینکه رابطه‌ی $\lg n! = O(n \lg n)$ برقرار است نامساوی (۱۳.۱) را در نظر می‌گیریم.

$$\underbrace{۱ \times ۲ \times ۳ \times \cdots \times n}_{n} \leq \underbrace{n \times n \times n \times \cdots \times n}_{n} \tag{۱۳.۱}$$

با خلاصه کردن و لگاریتم گرفتن از طرفین رابطه‌ی (۱۳.۱) به نامساوی $\lg n! \leqslant n \lg n$ می‌رسیم و این بدین معنی است که $\lg n! = O(n \lg n)$.

نشان دادن برقراری رابطه‌ی $\lg n! = \Omega(n \lg n)$ کمی مشکلتر است. بدین منظور نامساوی (۱۴.۱) را در نظر می‌گیریم.

$$\underbrace{\frac{n}{۲} \times \frac{n}{۲} \times \cdots \times \frac{n}{۲}}_{n/۲} \leq \underbrace{۱ \times ۲ \times \cdots \times n}_{n} \tag{۱۴.۱}$$

با خلاصه کردن و لگاریتم گرفتن از طرفین رابطه‌ی (۱۴.۱) به رابطه‌ی (۱۵.۱) می‌رسیم.

$$\lg \left(\frac{n}{۲}\right)^{\frac{n}{۲}} \leq \lg n! \Rightarrow \frac{n}{۲}(\lg n - ۱) \leq \lg n! \Rightarrow \frac{n}{۲} \lg n - \frac{n}{۲} \leq \lg n! \tag{۱۵.۱}$$

می‌دانیم که رابطه‌ی $(n/۲) \lg n - (n/۲) = \Theta(n \lg n)$ برقرار است. در نتیجه می‌توان گفت $n \lg n$ کفی برای $(n/۲) \lg n - (n/۲)$ است. از طرفی با در نظر گرفتن (۱۵.۱) می‌توان گفت $n \lg n$ کفی برای $\lg n!$ نیز هست و این یعنی $\lg n! = \Omega(n \lg n)$.

با توجه به اینکه نشان دادیم هر دو رابطه‌ی $\lg n! = O(n \lg n)$ و $\lg n! = \Omega(n \lg n)$ برقرار هستند پس می‌توان نتیجه گرفت رابطه‌ی $\lg n! = \Theta(n \lg n)$ نیز برقرار است و به این ترتیب اثبات کامل است.

◄ **سوال ۱۰.** می‌گوییم تابع f به صورت چندجمله‌ای کراندار است اگر به ازای ثابتی مانند k داشته باشیم $f(n) = O(n^k)$. آیا $\lceil \lg n \rceil!$ به صورت چندجمله‌ای کراندار است؟ در مورد $\lceil \lg \lg n \rceil!$ چه می‌توان گفت؟

◃ **پاسخ سوال ۱۰.**

ادعا می‌کنیم $\lceil \lg n \rceil!$ به صورت چندجمله‌ای کراندار نیست اما $\lceil \lg \lg n \rceil!$ به صورت چندجمله‌ای کراندار است.

قبل از اثبات ادعای مطرح شده به ذکر دو نکته که در اثبات‌ها از آنها استفاده خواهد شد می‌پردازیم.

نکته اول: تابع f به صورت چندجمله‌ای کراندار است اگر و فقط اگر $\lg f(n) = O(\lg n)$. دلایل این نتیجه‌گیری عبارت‌اند از:

- اگر تابع f به صورت چندجمله‌ای کراندار باشد آنگاه ثوابت n، c و k وجود دارند به طوری که برای تمامی nهای بزرگتر یا مساوی n، داریم $f(n) \leqslant cn^k$. با لگاریتم گرفتن از طرفین این نامساوی به عبارت $\lg f(n) \leqslant kc \lg n$ می‌رسیم و این یعنی $\lg f(n) = O(\lg n)$.
- اگر $\lg f(n) = O(\lg n)$ آنگاه f به صورت چندجمله‌ای کراندار است.

نکته دوم: دو رابطه‌ی زیر همواره برقرار هستند:

۱. $\lg n! = \Theta(n \lg n)$

۲. $\lceil \lg n \rceil = \Theta(\lg n)$

اثبات کراندار نبودن $\lceil \lg n \rceil!$:

اگر در رابطه‌ی اول از نکته‌ی دوم به جای n قرار دهیم $\lceil \lg n \rceil$ آنگاه داریم:

$$\lg\big(\lceil \lg n \rceil!\big) = \Theta\big(\lceil \lg n \rceil \lg\lceil \lg n \rceil\big)$$

با در نظر گرفتن رابطه‌ی دوم از نکته‌ی دوم و رابطه فوق می‌توان نوشت:

$$\lg\big(\lceil \lg n \rceil!\big) = \Theta\big(\lceil \lg n \rceil \lg\lceil \lg n \rceil\big) = \Theta\big(\lg n \lg\lg n\big) = \omega\big(\lg n\big)$$

بدین صورت نشان دادیم تابع $\lg n$ به عنوان یک مقدار کف برای تابع $\lg\big(\lceil \lg n \rceil!\big)$ است و نه سقفی برای آن. پس با در نظر گرفتن نکته‌ی اول می‌توان گفت $\lceil \lg n \rceil!$ به صورت چندجمله‌ای کراندار نیست.

اثبات کراندار بودن $\lceil \lg\lg n \rceil!$:

اگر در رابطه‌ی اول از نکته‌ی دوم به جای n قرار دهیم $\lceil \lg\lg n \rceil$ خواهیم داشت:

$$\lg\big(\lceil \lg\lg n \rceil!\big) = \Theta\big(\lceil \lg\lg n \rceil \lg\lceil \lg\lg n \rceil\big)$$

با در نظر گرفتن رابطه‌ی دوم از نکته‌ی دوم و رابطه بالا می‌توان نوشت:

$$\lg\big(\lceil \lg\lg n \rceil!\big) = \Theta\big(\lceil \lg\lg n \rceil \lg\lceil \lg\lg n \rceil\big) = \Theta\big(\lg\lg n \lg\lg\lg n\big)$$

چون $\lg\lg\lg n$ از $\lg\lg n$ کوچکتر است در نتیجه رابطه‌ی فوق را به صورتی که در ادامه آمده است در نظر می‌گیریم:

$$\lg\big(\lceil \lg\lg n \rceil!\big) = \Theta\big(\lg\lg n \lg\lg\lg n\big) = o\big(\lg^{۲} \lg n\big)$$

چون برای هر دو عدد حقیقی $a \geqslant ۰$ و $b \geqslant ۰$ رابطه‌ی $\lg^b n = o(n^a)$ برقرار است اگر در این رابطه به جای n قرار دهیم $\lg n$، به جای a مقدار ۱ و به جای b مقدار ۲ را قرار دهیم آنگاه می‌توان رابطه‌ی فوق را به صورت زیر نوشت:

$$\lg\big(\lceil \lg\lg n \rceil!\big) = o\big(\lg^{۲} \lg n\big) = o(\lg n)$$

به این ترتیب اثبات کردیم $\lg n$ سقفی برای $\lg\big(\lceil \lg\lg n \rceil!\big)$ است و طبق نکته‌ی اول، این بدین معنی است که تابع $\lceil \lg\lg n \rceil!$ به صورت چندجمله‌ای کراندار است.

◄ **سوال ۱۱.** توابعی که در ادامه آمده است را بر حسب نرخ رشد و به صورت صعودی مرتب کنید. در صورتیکه دو تابع دارای نرخ رشد یکسان بودند، از علامت تساوی بین دو تابع استفاده کنید (در تمام توابع فرض کنید n عددی مثبت و بزرگ است).

$$۲^{\lg n} \quad (\lg n)! \quad ۲^{\sqrt{۲\lg n}} \quad n^{۱/\lg n} \quad \left(\frac{۳}{۲}\right)^n \quad n! \quad \left(\sqrt{۲}\right)^{\lg n} \quad \lg n \quad ۸^{\lg n}$$

$$\ln n \quad \ln\ln n \quad n^n \quad n^۲ \quad ۲^n \quad (\lg n)^{\lg n} \quad \lg n! \quad ۲^{۲^n} \quad ۲^{۲^{n+۱}}$$

$$n\lg n \quad e^n \quad \sqrt{\lg n} \quad n۲^n \quad n^۳ \quad ۴^{\lg n} \quad \sqrt{n} \quad n \quad \lg^۲ n$$

◁ **پاسخ سوال ۱۱.**

لیست مرتب شده‌ی توابع در ادامه آمده است. در این لیست توابع از چپ به راست به ترتیب صعودی نوشته شده‌اند.

$$n^{۱/\lg n} \quad \ln\ln n \quad \sqrt{\lg n} \quad \lg n = \ln n \quad \lg^۲ n$$

$$۲^{\sqrt{۲\lg n}} \quad \left(\sqrt{۲}\right)^{\lg n} = \sqrt{n} \quad n = ۲^{\sqrt{\lg n}} \quad n\lg n = \lg n! \quad n^۲ = ۴^{\lg n}$$

$$n^۳ = ۸^{\lg n} \quad (\lg n)! \quad (\lg n)^{\lg n} \quad \left(\frac{۳}{۲}\right)^n \quad ۲^n$$

$$n۲^n \quad e^n \quad n! \quad n^n \quad ۲^{۲^n}$$

$$۲^{۲^{n+۱}}$$

در ادامه نشان داده می‌شود که ترتیب برخی از توابع چگونه به دست آمده است.

۱. تابع $n^{۱/\lg n}$ را به صورتی که در ادامه آمده است ساده می‌کنیم:

$$n^{۱/\lg n} = n^{\log_n ۲} = ۲^{\log_n n} = ۲$$

پس تابع $n^{۱/\lg n}$ معادل با عدد ۲ است. چون سایر توابع داده شده دارای رشد مثبت هستند در نتیجه عدد ۲ دارای کوچکترین مرتبه‌ی رشد است و تابع $n^{۱/\lg n}$ در ابتدای لیست، به عنوان تابعی با کمترین نرخ رشد، قرار می‌گیرد.

۲. می‌دانیم برای هر دو عدد حقیقی $a > ۱$ و b رابطه‌ی (۱۶.۱) برقرار است.

$$\lim_{n\to\infty} \frac{n^b}{a^n} = ۰ \tag{۱۶.۱}$$

اگر در (۱۶.۱) به جای n قرار دهیم $\lg n$ و به جای a قرار دهیم ۲^a آنگاه می‌توان نوشت:

$$\lim_{n\to\infty} \frac{\lg^b n}{(۲^a)^{\lg n}} = ۰ \Rightarrow \lim_{n\to\infty} \frac{\lg^b n}{(۲^{\lg n})^a} = ۰ \Rightarrow \lim_{n\to\infty} \frac{\lg^b n}{(n^{\lg ۲})^a} = ۰ \Rightarrow \lim_{n\to\infty} \frac{\lg^b n}{n^a} = ۰$$

چون حاصل حد فوق برابر با صفر است می‌توان رابطه‌ی $\lg^b n = O(n^a)$ را نتیجه گرفت. حال اگر قرار دهیم $b = ۱$ و $a = ۱/۲$ خواهیم داشت $\lg n = O(\sqrt{n})$ و این یعنی $\sqrt{n}$ دارای نرخ رشد بیشتری نسبت به $\lg n$ است. در نتیجه در لیست مرتب شده‌ی توابع، $\lg n$ باید قبل از $\sqrt{n}$ قرار بگیرد.

۳. می‌دانیم نامساوی $\lg n < n$ برای هر $n \geqslant ۱$ برقرار است. با ضرب n در طرفین این نامساوی به

نامساوی $n\lg n < n^{۲}$ می‌رسیم و این یعنی نرخ رشد $n\lg n$ از $n^{۲}$ کمتر است.

۴. درستی نامساوی (۱۷.۱) بدیهی است.

$$\underbrace{۱ \times ۲ \times ۳ \times \cdots \times \lg n}_{\lg n} < \underbrace{\lg n \times \lg n \times \cdots \times \lg n}_{\lg n} \tag{۱۷.۱}$$

عبارت سمت چپ در نامساوی (۱۷.۱) برابر با $(\lg n)!$ و عبارت سمت راست برابر با $(\lg n)^{\lg n}$ است. در نتیجه می‌توان گفت نرخ رشد $(\lg n)!$ از $(\lg n)^{\lg n}$ کمتر است.

۵. برای اثبات اینکه نرخ رشد تابع $\lg^{۲} n$ از $۲^{\sqrt{۲\lg n}}$ کمتر است، یعنی $\lg^{۲} n < ۲^{\sqrt{۲\lg n}}$، کافی است ابتدا از طرفین این نامساوی لگاریتم بر پایه‌ی دو بگیریم:

$$\lg\lg^{۲} n < \lg ۲^{\sqrt{۲\lg n}} \Rightarrow ۲\lg\lg n < \sqrt{۲\lg n} \Rightarrow \lg\lg n < \frac{\sqrt{۲}}{۲}\sqrt{\lg n}$$

با در نظر گرفتن نامساوی اخیر و اثبات مورد شماره‌ی ۲ می‌توان برقراری رابطه‌ی $\lg^{۲} n < ۲^{\sqrt{۲\lg n}}$ را نتیجه گرفت.

۶. برای اثبات نامساوی $۲^{\sqrt{۲\lg n}} < \sqrt{n}$ می‌توان مانند مورد ۵ از دو طرف نامساوی لگاریتم بر پایه‌ی دو گرفت و اثبات را به روشی مشابه انجام داد.

۷. این قسمت به اثبات برقراری نامساوی $(\lg n)^{\lg n} < (۳/۲)^{n}$ اختصاص دارد. می‌دانیم که برای هر $n \geqslant ۴$ نامساوی‌های $n^{۱/۲} \geqslant \lg n$ و $n^{۱/۲} \geqslant \lg\lg n$ برقرار هستند. با ضرب طرفین این دو نامساوی در یکدیگر می‌توان به نتیجه‌ای که در ادامه آمده است رسید.

$$n \geq \lg n \lg\lg n \Rightarrow n \geq \lg_{\frac{۳}{۲}} n \lg\lg n$$

اگر عدد ۳/۲ را به عنوان پایه‌ی توان برای طرفین نامساوی اخیر در نظر بگیریم آنگاه به نامساوی (۱۸.۱) می‌رسیم.

$$\left(\frac{۳}{۲}\right)^{n} > \left(\frac{۳}{۲}\right)^{\lg_{\frac{۳}{۲}} n \lg\lg n} \tag{۱۸.۱}$$

عبارت سمت راست نامساوی (۱۸.۱) را می‌توان به صورت زیر ساده کرد:

$$\left(\frac{۳}{۲}\right)^{\lg_{\frac{۳}{۲}} n \lg\lg n} = \left(\left(\frac{۳}{۲}\right)^{\lg_{\frac{۳}{۲}} n}\right)^{\lg\lg n} = n^{\lg\lg n} = (\lg n)^{\lg n}$$

به این ترتیب اگر به جای عبارت سمت راست نامساوی (۱۸.۱) عبارت معادل آن یعنی $(\lg n)^{\lg n}$ را قرار دهیم داریم:

$$\left(\frac{۳}{۲}\right)^{n} > (\lg n)^{\lg n}$$

و این یعنی نرخ رشد تابع $(۳/۲)^n$ بیشتر از $(\lg n)^{\lg n}$ است.

۸. اثبات برقراری نامساوی $n! < n^n < ۲^{۲^n}$ به راحتی صورت می‌پذیرد. می‌دانیم که نامساوی $n \lg n < ۲^n$ برقرار است. حال اگر عدد دو را به عنوان پایه‌ی توان برای طرفین این نامساوی در نظر بگیریم آنگاه می‌توان نتیجه‌گیری زیر را انجام داد:

$$۲^{n \lg n} < ۲^{۲^n} \Rightarrow \left(۲^{\lg n}\right)^n < ۲^{۲^n} \Rightarrow \left(n^{\lg ۲}\right)^n < ۲^{۲^n} \Rightarrow n^n < ۲^{۲^n}$$

پس نامساوی $n^n < ۲^{۲^n}$ برقرار است.

با توجه به برقراری نامساوی بدیهی زیر می‌توان به درستی نامساوی $n! < n^n$ نیز پی برد:

$$n! = \underbrace{۱ \times ۲ \times ۳ \times \cdots \times n}_{n} \leq \underbrace{n \times n \times n \times \cdots \times n}_{n} = n^n$$

چون هر دو نامساوی $n^n < ۲^{۲^n}$ و $n! < n^n$ برقرار هستند پس نامساوی $n! < n^n < ۲^{۲^n}$ نیز برقرار است و این بدین معنی است که در لیست مرتب شده‌ی توابع، $n!$ قبل از n^n و n^n قبل از $۲^{۲^n}$ قرار می‌گیرد.

◄ **سوال ۱۲.** با فرض اینکه توابع f و g هر دو غیرمنفی هستند درستی هر یک از موارد زیر را اثبات یا رد کنید.

۱. $f(n) = O\left(f^۲(n)\right)$

۲. $f(n) = O\big(g(n)\big) \Rightarrow ۲^{f(n)} = O\left(۲^{g(n)}\right)$

۳. $f(n) + o\big(f(n)\big) = \Theta\big(f(n)\big)$

◃ **پاسخ سوال ۱۲.**

مورد اول نادرست است. اگر تابع $f(n)$ دارای مقداری همواره کمتر از یک باشد، مثلاً قرار دهیم $f(n) = ۱/n$، آنگاه داریم $f^۲(n) \leqslant f(n)$ و این یعنی $f^۲(n)$ نمی‌تواند سقفی برای $f(n)$ باشد.

مورد دوم نیز نادرست است. با استفاده از یک مثال نقض می‌توان به این موضوع پی برد. اگر قرار دهیم $f(n) = ۲ \lg n$ و $g(n) = \lg n$ آنگاه می‌دانیم که $f(n) = O\big(g(n)\big)$ اما $۲^{f(n)}$ برابر است با $n^۲$ و $۲^{g(n)}$ برابر است با n و می‌دانیم که n نمی‌تواند سقفی برای $n^۲$ باشد.

مورد سوم درست است. فرض می‌کنیم تابعی مانند $g(n)$ وجود دارد بطوریکه $g(n) = o\big(f(n)\big)$. با توجه به اینکه توابع f و g هر دو غیرمنفی هستند می‌توان نوشت:

$$f(n) \leq f(n) + g(n) \tag{۱۹.۱}$$

از طرفی با در نظر گرفتن تعریف $o\big(f(n)\big)$ که در ادامه آمده است:

$$o\big(f(n)\big) = \{g(n) \mid \forall c > ۰, \exists n_۰ > ۰, \forall n \geq n_۰ : g(n) < cf(n)\} \tag{۲۰.۱}$$

و افزودن $f(n)$ به طرفین نامساوی موجود در (۲۰.۱) می‌توان نوشت:

$$f(n) + g(n) < cf(n) + f(n) = (c+1)f(n) \tag{۲۱.۱}$$

با ترکیب (۱۹.۱) و (۲۱.۱) داریم:

$$f(n) \leq f(n) + g(n) \leq (c+1)f(n)$$

و این یعنی $f(n) + g(n) = \Theta\big(f(n)\big)$.

چون $g(n)$ را به عنوان یک تابع دلخواه متعلق به $o\big(f(n)\big)$ انتخاب کردیم و نشان دادیم رابطه‌ی $f(n) + g(n) = o\big(f(n)\big)$ برقرار است پس می‌توان نتیجه گرفت برای تمام توابع متعلق به $o\big(f(n)\big)$ رابطه‌ی $f(n) + o\big(f(n)\big) = \Theta\big(f(n)\big)$ برقرار است.

۵.۱ روابط بازگشتی

◄ **سوال ۱۳.** بزرگترین مقسوم علیه مشترک دو عدد m و n برابر است با بزرگترین عددی که هم بر m و هم بر n بخش‌پذیر است. برای مثال بزرگترین مقسوم علیه مشترک دو عدد ۹ و ۱۵ برابر با ۳ است. از الگوریتم (۲.۱) می‌توان برای محاسبه‌ی بزرگترین مقسوم علیه مشترک دو عدد استفاده کرد (در این الگوریتم mod بیانگر عملگر باقیمانده است). یک رابطه‌ی بازگشتی ارائه دهید که نشان‌دهنده‌ی تعداد اعمال mod انجام شده توسط این الگوریتم باشد.

الگوریتم ۲.۱ به دست آوردن بزرگترین مقسوم علیه مشترک دو عدد

```
function GCD(m, n)
    if m < n
        SWAP(m, n)
    end if
    if n == 0
        return m
    else
        return GCD(n, m mod n)
    end if
end function
```

◃ **پاسخ سوال ۱۳.**

تعداد اعمال mod انجام شده در الگوریتم (۲.۱) را با $T(m,n)$ نشان می‌دهیم. حالت پایه‌ی رابطه‌ی بازگشتی هنگامی است که n برابر با صفر باشد که در این صورت تعداد اعمال mod انجام شده برابر با

صفر خواهد بود. یعنی داریم:

$$T(m,n) = ۰ \tag{۲۲.۱}$$

در صورتیکه n بزرگتر از صفر باشد آنگاه ابتدا یک عمل mod انجام شده و سپس دوباره الگوریتم با مقادیر جدید فراخوانی می‌شود. برای این حالت خواهیم داشت:

$$T(m,n) = ۱ + T(n, m \bmod n) \tag{۲۳.۱}$$

با ترکیب روابط (۲۲.۱) و (۲۳.۱) به رابطه‌ی بازگشتی (۲۴.۱) می‌رسیم که نشان‌دهنده‌ی تعداد اعمال mod انجام شده در الگوریتم (۲.۱) است.

$$T(m,n) = \begin{cases} ۰ & n = ۰ \\ ۱ + T(n, m \bmod n) & n > ۰ \end{cases} \tag{۲۴.۱}$$

◄ **سوال ۱۴.** مرتبه‌ی رابطه‌ی بازگشتی زیر را به دست آورید (راهنمایی: کار را با تقسیم طرفین بر n شروع کنید).

$$T(n) = \begin{cases} \Theta(۱) & n = ۱ \\ \sqrt{n}\,T(\sqrt{n}) + n & n > ۱ \end{cases}$$

◃ **پاسخ سوال ۱۴.**

ابتدا طرفین رابطه‌ی $T(n) = \sqrt{n}\,T(\sqrt{n}) + n$ را بر n تقسیم می‌کنیم که در این صورت داریم:

$$\frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + ۱ \tag{۲۵.۱}$$

اگر قرار دهیم $H(n) = T(n)/n$ آنگاه می‌توان رابطه‌ی بازگشتی (۲۵.۱) را به صورتی که در ادامه آمده است بازنویسی کرد:

$$H(n) = H(\sqrt{n}) + ۱ \tag{۲۶.۱}$$

با در نظر گرفتن تغییر متغیر $n = ۲^k$ می‌توان رابطه‌ی بازگشتی (۲۶.۱) را به شکل (۲۷.۱) نوشت:

$$H\left(۲^k\right) = H\left(۲^{k/۲}\right) + ۱ \tag{۲۷.۱}$$

اگر قرار دهیم $S(k) = H(۲^k)$ آنگاه می‌توان رابطه‌ی (۲۷.۱) را به شکل زیر در نظر گرفت:

$$S(k) = S\left(\frac{k}{۲}\right) + ۱$$

$S(k)$ دارای فرم کلی یک رابطه‌ی تقسیم و غلبه[۳] است پس با استفاده از قضیه‌ی اصلی[۴] و حالت دوم این

[۳] Divide and conquer
[۴] Master theorem

قضیه می‌توان نتیجه گرفت:

$$S(k) = \Theta(\lg k)$$

اگر به جای $S(k)$، معادل آن یعنی $H(۲^k)$ را قرار دهیم داریم:

$$H(۲^k) = \Theta(\lg k) \tag{۲۸.۱}$$

با در نظر گرفتن رابطه‌ی $n = ۲^k$ می‌توان نتیجه گرفت $\lg\lg n = \lg k$. با جایگذاری $\lg\lg n$ به جای $\lg k$ در (۲۸.۱) داریم:

$$H(n) = \Theta(\lg\lg n)$$

اگر به جای $H(n)$، معادل آن یعنی $T(n)/n$ را قرار دهیم به رابطه‌ی زیر می‌رسیم:

$$\frac{T(n)}{n} = \Theta(\lg\lg n) \Rightarrow T(n) = n \cdot \Theta(\lg\lg n) \Rightarrow T(n) = \Theta(n\lg\lg n)$$

به این ترتیب می‌توان گفت $T(n)$ از مرتبه‌ی $\Theta(n\lg\lg n)$ است.

◂ **سوال ۱۵.** نشان دهید رابطه‌ی بازگشتی زیر از مرتبه‌ی $O(n)$ است.

$$T(n) = \begin{cases} \Theta(۱) & n = ۱ \\ T\left(\frac{n}{۲}\right) + T\left(\frac{n}{۴}\right) + T\left(\frac{n}{۸}\right) + n & n > ۱ \end{cases}$$

▹ **پاسخ سوال ۱۵.**

برای به دست آوردن یک حدس مناسب برای مرتبه‌ی رابطه‌ی بازگشتی $T(n) = T(n/۲) + T(n/۴) + T(n/۸) + n$ از درخت بازگشت استفاده می‌کنیم. سپس با استفاده از روش جانشینی[۵] نشان می‌دهیم که حدس به دست آمده یک حدس درست است.

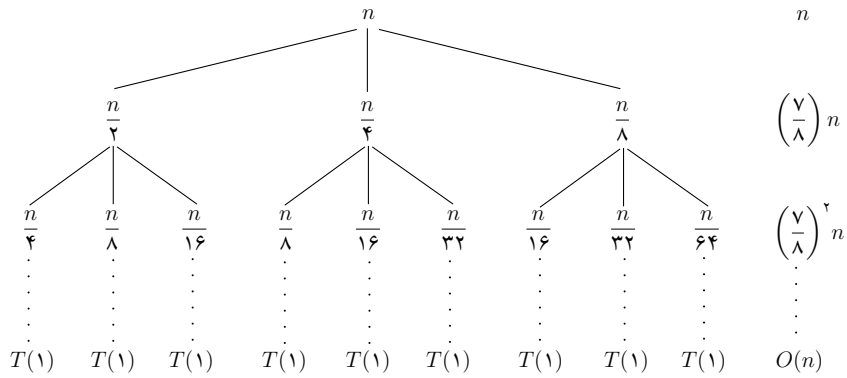
درخت بازگشت رابطه‌ی بازگشتی $T(n) = T(n/۲)+T(n/۴)+T(n/۸)+n$ در شکل (۱.۱) نشان داده شده است. هزینه‌ی هر سطح از درخت در سمت راست آن سطح نوشته شده است. با توجه به این شکل، مسیر زیر دارای بیشترین طول در درخت است و در نتیجه ارتفاع درخت بازگشت توسط این مسیر تعیین می‌شود.

$$n \to \frac{n}{۲} \to \frac{n}{۴} \to \frac{n}{۸} \to \cdots \to \frac{n}{۲^i} \to \cdots \to ۱$$

برای تعیین ارتفاع درخت باید دید چه زمانی عبارت $n/۲^i$ برابر با یک می‌شود. اگر قرار دهیم $n/۲^i = ۱$ آنگاه می‌توان گفت $i = \lg n$ و این یعنی ارتفاع درخت برابر با $\lg n$ است.

با توجه به ارتفاع درخت می‌توان گفت تعداد سطوح درخت برابر با $\lg n + ۱$ است (از سطح یک تا سطح

[۵] Substitution method



شکل (۱.۱): درخت بازگشت رابطه‌ی بازگشتی $T(n) = T(n/۲) + T(n/۴) + T(n/۸) + n$

$(\lg n + ۱)$. از طرفی هزینه‌ی هر سطح (به جز سطح آخر) از رابطه‌ی $(۷/۸)^{i-۱}n$ پیروی می‌کند که در آن i شماره‌ی سطح است. هزینه‌ی سطح آخر از فرمول کلی هزینه‌ی یک سطح پیروی نمی‌کند زیرا هزینه‌ی گره‌ای در سطح آخر نشان‌دهنده‌ی حالت پایه‌ی رابطه بازگشتی است و در نتیجه هزینه‌ی سطح آخر باید به صورت جداگانه محاسبه شود. اگر سطح آخر دارای حداکثر تعداد گره یعنی $۳^{\lg n}$ (معادل با $n^{\lg ۳}$) گره باشد چون هزینه‌ی هر گره $\Theta(۱)$ است در نتیجه هزینه‌ی سطح آخر برابر با $O(n^{\lg ۳})$ خواهد بود. اما از آنجایی که سطح آخر دارای حداکثر تعداد گره نیست در نتیجه هزینه‌ی این سطح کمتر از $O(n^{\lg ۳})$ است. در ادامه، هزینه‌ی سطح آخر را $O(n)$ در نظر می‌گیریم.

پس از به دست آوردن تعداد سطوح و هزینه‌ی هر سطح می‌توان به محاسبه‌ی هزینه‌ی کلی درخت بازگشت پرداخت که این هزینه بیانگر مرتبه‌ی رابطه‌ی بازگشتی است. هزینه‌ی درخت بازگشت توسط رابطه‌ی (۲۹.۱) بیان می‌شود.

$$T(n) = O(n) + \sum_{i=۱}^{\lg n} \left(\frac{۷}{۸}\right)^{i-۱} n \tag{۲۹.۱}$$

رابطه‌ی (۲۹.۱) را می‌توان به صورتی که در ادامه آمده است ساده کرد:

$$\begin{aligned} T(n) &= O(n) + \sum_{i=۰}^{\lg n - ۱} \left(\frac{۷}{۸}\right)^{i} n \\ &\leq O(n) + \sum_{i=۰}^{\infty} \left(\frac{۷}{۸}\right)^{i} n \\ &= O(n) + n \cdot \frac{۱}{۱ - \frac{۷}{۸}} \\ &= O(n) + ۸n \end{aligned}$$

به این ترتیب حدس می‌زنیم $T(n)$ از مرتبه‌ی $O(n)$ باشد. در ادامه نشان می‌دهیم حدس زده شده یک حدس مناسب است.

فرض می‌کنیم رابطه‌ی $T(k) \leqslant ck$، که در آن c یک مقدار ثابت غیرمنفی است، برای هر $k < n$ برقرار باشد. نشان می‌دهیم رابطه‌ی $T(k) \leqslant ck$ برای $k = n$ نیز برقرار است.

چون رابطه‌ی $T(k) \leqslant ck$ برای مقادیر کمتر از n برقرار است و از طرفی $n/۲$، $n/۴$ و $n/۸$ هر سه کمتر از n هستند پس روابط (۳۰.۱)، (۳۱.۱) و (۳۲.۱) برقرار هستند.

$$T\left(\frac{n}{۲}\right) \leq \frac{cn}{۲} \tag{۳۰.۱}$$

$$T\left(\frac{n}{۴}\right) \leq \frac{cn}{۴} \tag{۳۱.۱}$$

$$T\left(\frac{n}{۸}\right) \leq \frac{cn}{۸} \tag{۳۲.۱}$$

با جایگذاری (۳۰.۱)، (۳۱.۱) و (۳۲.۱) در رابطه‌ی بازگشتی و ساده‌سازی خواهیم داشت:

$$\begin{aligned}
T(n) &\leq \frac{cn}{۲} + \frac{cn}{۴} + \frac{cn}{۸} + n \\
&= \frac{۷cn}{۸} + n \\
&= \frac{۷cn}{۸} + \frac{cn}{۸} - \frac{cn}{۸} + n \\
&= cn - \frac{cn}{۸} + n \\
&= cn - \left(\frac{cn}{۸} - n\right)
\end{aligned}$$

حال باید مقدار ثابت c را طوری تعیین کنیم که داشته باشیم $(cn/۸) - n \geqslant ۰$ تا بتوانیم نتیجه بگیریم $T(n) \leqslant cn$. مقدار c به صورت زیر به دست می‌آید:

$$\frac{cn}{۸} - n \geq ۰ \Rightarrow \frac{cn}{۸} \geq n \Rightarrow cn \geq ۸n \Rightarrow c \geq ۸$$

به این ترتیب ثابت کردیم رابطه‌ی بازگشتی $T(n) = T(n/۲) + T(n/۴) + T(n/۸) + n$ از مرتبه‌ی $O(n)$ است.

◄ **سوال ۱۶.** رابطه‌ی بازگشتی زیر از چه مرتبه‌ای است؟

$$T(n) = \begin{cases} \Theta(۱) & n = ۱ \\ ۵T\left(\frac{n}{۵}\right) + \frac{n}{\log_۵ n} & n > ۱ \end{cases}$$

◃ **پاسخ سوال ۱۶.**

برای به دست آوردن مرتبه‌ی این رابطه‌ی بازگشتی از روش تکرار با جایگزینی استفاده می‌کنیم. می‌توان $T(n) = ۵T(n/۵) + n/\log_۵ n$ را به صورت زیر بسط داد:

$$T(n) = ۵T\left(\frac{n}{۵}\right) + \frac{n}{\log_۵ n}$$

$$
\begin{aligned}
&= ۵^۲T\left(\frac{n}{۵^۲}\right) + \frac{n}{\log_۵ n} + \frac{n}{\log_۵(n/۵)} \\
&= ۵^۳T\left(\frac{n}{۵^۳}\right) + \frac{n}{\log_۵ n} + \frac{n}{\log_۵(n/۵)} + \frac{n}{\log_۵(n/۵^۲)} \\
&= \quad \vdots \\
&= ۵^kT\left(\frac{n}{۵^k}\right) + \frac{n}{\log_۵ n} + \frac{n}{\log_۵(n/۵)} + \frac{n}{\log_۵(n/۵^۲)} + \cdots + \frac{n}{\log_۵(n/۵^{k-۱})} \\
&= ۵^kT\left(\frac{n}{۵^k}\right) + \sum_{i=۰}^{k-۱} \frac{n}{\log_۵(n/۵^i)} \qquad (۳۳.۱)
\end{aligned}
$$

اگر فرض کنیم $n = ۵^k$ آنگاه داریم $k = \log_۵ n$. با قرار دادن $\log_۵ n$ به جای k در (۳۳.۱) خواهیم داشت:

$$
\begin{aligned}
T(n) &= ۵^{\log_۵ n}T(۱) + \sum_{i=۰}^{\log_۵ n-۱} \frac{n}{\log_۵(n/۵^i)} \\
&= nT(۱) + n\sum_{i=۰}^{\log_۵ n-۱} \frac{۱}{\log_۵ n - \log_۵ ۵^i} \\
&= n\Theta(۱) + n\sum_{i=۰}^{\log_۵ n-۱} \frac{۱}{\log_۵ n - i} \\
&= n\Theta(۱) + n\sum_{i=۱}^{\log_۵ n} \frac{۱}{i} \\
&= n\Theta(۱) + n\lg\log_۵ n \\
&= \Theta(n) + n\lg\log_۵ n
\end{aligned}
$$

بدین ترتیب نشان دادیم $T(n) = \Theta(n) + n\lg\log_۵ n$. از آنجایی که پایه‌ی لگاریتم در مرتبه اهمیت ندارد در نتیجه می‌توان گفت $T(n)$ از مرتبه‌ی $\Theta(n\lg\lg n)$ است.

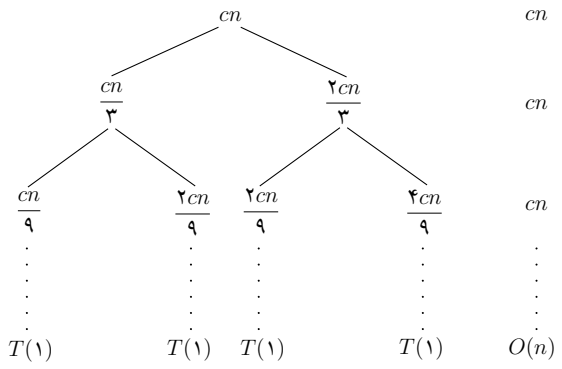
◄ **سوال ۱۷.** مرتبه‌ی رابطه‌ی بازگشتی زیر را تعیین کنید.

$$
T(n) = \begin{cases} \Theta(۱) & n = ۱ \\ T\left(\frac{n}{۳}\right) + T\left(\frac{۲n}{۳}\right) + cn & n > ۱ \end{cases}
$$

◃ **پاسخ سوال ۱۷.**

درخت بازگشت رابطه‌ی بازگشتی $T(n) = T(n/۳) + T(۲n/۳) + cn$ در شکل (۲.۱) نشان داده شده است.

با توجه به شکل (۲.۱)، مسیر زیر دیرتر از سایر مسیرها به $T(۱)$ (که همان شرط توقف رابطه‌ی بازگشتی



شکل (۲.۱): درخت بازگشت رابطه‌ی بازگشتی $T(n) = T(n/۳) + T(۲n/۳) + cn$

است) می‌رسد. در نتیجه از این مسیر می‌توان برای تعیین ارتفاع درخت بازگشت و تخمین سقف رابطه‌ی بازگشتی استفاده کرد.

$$cn \to \frac{۲cn}{۳} \to \frac{۴cn}{۹} \to \frac{۸cn}{۲۷} \to \cdots \to \frac{۲^i cn}{۳^i} \to \cdots \to ۱$$

با توجه به مسیر فوق می‌توان گفت ارتفاع درخت بازگشت برابر با $\log_{۳/۲} n$ است. همچنین هزینه‌ی هر سطح از این درخت برابر با cn است.

با توجه به اینکه حداکثر ارتفاع درخت بازگشت و هزینه‌ی هر سطح از این درخت را داریم یک حدس مناسب برای مرتبه‌ی این رابطه‌ی بازگشتی می‌تواند $O(n \log_{۳/۲} n)$ باشد. چون به ازای هر دو عدد غیرمنفی a و b داریم $O(\log_a n) = O(\log_b n)$ پس $O(n \log_{۳/۲} n)$ معادل با $O(n \lg n)$ است. در ادامه نشان می‌دهیم $O(n \lg n)$ سقف مناسبی برای رابطه‌ی بازگشتی $T(n) = T(n/۳) + T(۲n/۳) + cn$ است.

فرض می‌کنیم رابطه‌ی $T(k) \leqslant dk \lg k$، که در آن d یک مقدار ثابت غیرمنفی است، برای هر $k < n$ برقرار باشد. نشان می‌دهیم که این رابطه برای $k = n$ نیز برقرار است.

چون رابطه‌ی $T(k) \leqslant dk \lg k$ برای مقادیر کمتر از n برقرار است و از طرفی $n/۳$ و $۲n/۳$ هر دو کمتر از n هستند پس روابط (۳۴.۱) و (۳۵.۱) نیز برقرارند.

$$T\left(\frac{n}{۳}\right) \leq d\frac{n}{۳} \lg \frac{n}{۳} \tag{۳۴.۱}$$

$$T\left(\frac{۲n}{۳}\right) \leq d\frac{۲n}{۳} \lg \frac{۲n}{۳} \tag{۳۵.۱}$$

با جایگذاری (۳۴.۱) و (۳۵.۱) در $T(n) = T(n/۳) + T(۲n/۳) + cn$ داریم:

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

$$= dn\lg n - d\left(\frac{n}{۳}\lg ۳ + \frac{۲n}{۳}\lg ۳ - \frac{۲n}{۳}\lg ۲\right) + cn$$
$$= dn\lg n - dn\left(\lg ۳ - \frac{۲}{۳}\right) + cn$$
$$= dn\lg n - \left(dn\left(\lg ۳ - \frac{۲}{۳}\right) - cn\right)$$

در این مرحله باید مقدار ثابت d را طوری تعیین کنیم که عبارت $dn(\lg ۳ - (۲/۳)) - cn$ بزرگتر یا مساوی صفر باشد تا بتوانیم نتیجه بگیریم $T(n) \leqslant dn\lg n$. مقدار d به صورت زیر به دست می‌آید:

$$dn\left(\lg ۳ - \frac{۲}{۳}\right) - cn \geq ۰ \Rightarrow dn \geq \frac{cn}{\lg ۳ - \frac{۲}{۳}} \Rightarrow d \geq \frac{c}{\lg ۳ - \frac{۲}{۳}}$$

پس کافیست ثابت d را طوری انتخاب کنیم که رابطه‌ی $d \geqslant c/(\lg ۳ - (۲/۳))$ برقرار باشد.

به این ترتیب نشان دادیم رابطه‌ی بازگشتی $T(n) = T(n/۳) + T(۲n/۳) + cn$ از مرتبه‌ی $O(n\lg n)$ است.

◄ **سوال ۱۸.** رابطه‌ی بازگشتی $T(n) = ۸T(n/۲) + ۲n^۳\lg^۲ n + ۳n^۳$ از چه مرتبه‌ای است؟

◁ **پاسخ سوال ۱۸.**

برای به دست آوردن مرتبه‌ی رابطه‌ی بازگشتی $T(n) = ۸T(n/۲) + ۲n^۳\lg^۲ n + ۳n^۳$ می‌توان از حالت خاص قضیه‌ی اصلی استفاده کرد. حالت خاص قضیه‌ی اصلی در ادامه آمده است.

در رابطه‌ی بازگشتی $T(n) = aT(n/b) + f(n)$ که در آن $a \geqslant ۱$ و $b > ۱$ مقادیر ثابت هستند اگر داشته باشیم $f(n) = \Theta(n^{\log_b a}\lg^k n)$ آنگاه مرتبه‌ی رابطه‌ی بازگشتی $T(n)$ برابر با $\Theta(n^{\log_b a}\lg^{k+۱} n)$ خواهد بود.

با در نظر گرفتن رابطه‌ی بازگشتی $T(n) = ۸T(n/۲) + ۲n^۳\lg^۲ n + ۳n^۳$ داریم $a = ۸$، $b = ۲$، $n^{\log_b a} = n^۳$ و $f(n) = ۲n^۳\lg^۲ n + ۳n^۳$. با توجه به حالت خاص قضیه‌ی اصلی چون رابطه‌ی $۲n^۳\lg^۲ n + ۳n^۳ = \Theta(n^۳\lg^۲ n)$ برقرار است در نتیجه می‌توان گفت $T(n)$ از مرتبه‌ی $\Theta(n^۳\lg^۳ n)$ است.

◄ **سوال ۱۹.** مرتبه‌ی رابطه‌ی بازگشتی زیر را به دست آورید.

$$T(n) = \begin{cases} ۱ & n = ۰ \\ \sum_{i=۰}^{n-۱} T(i) + ۱ & n > ۰ \end{cases}$$

◁ **پاسخ سوال ۱۹.**

با توجه به تعریف رابطه‌ی بازگشتی $T(n)$، برای حالت $n = ۰$ داریم:

$$T(۰) = ۱$$

از مقدار $T(۰)$ می‌توان مقدار $T(۱)$ را به صورت زیر به دست آورد:

$$T(۱) = T(۰) + ۱$$

با در نظر گرفتن مقدار $T(۱)$، مقدار $T(۲)$ به صورتی که در ادامه آمده است خواهد بود:

$$T(۲) = T(۰) + T(۱) + ۱ = ۲T(۰) + ۱ + ۱$$

از مقدار $T(۲)$، می‌توان مقدار $T(۳)$ را به صورت زیر به دست آورد:

$$T(۳) = T(۰) + T(۱) + T(۲) + ۱ = ۴T(۰) + ۱ + ۱ + ۱ + ۱$$

اگر همین روند را ادامه دهیم مقدار $T(n)$ به صورت زیر به دست می‌آید:

$$T(n) = T(۰) + T(۱) + T(۲) + \cdots + T(n-۱) + ۱ = ۲^{n-۱}T(۰) + ۲^{n-۱}$$

اگر در رابطه‌ی $T(n) = ۲^{n-۱}T(۰) + ۲^{n-۱}$ به جای $T(۰)$ مقدار معادل آن یعنی ۱ را قرار دهیم به رابطه‌ی $T(n) = ۲^n$ می‌رسیم و به این ترتیب می‌توان نتیجه گرفت $T(n)$ از مرتبه‌ی $\Theta(۲^n)$ است.

۱.۶ تحلیل مرتبه‌ی زمانی الگوریتم‌ها

◄ **سوال ۲۰.** عدد صحیح m و آرایه‌ی A که حاوی اعداد صحیح است را در نظر بگیرید. الگوریتمی بنویسید که اگر تعداد اعداد بزرگتر یا مساوی m در آرایه‌ی A، بیشتر یا مساوی با تعداد اعداد کوچکتر از m است مقدار TRUE و در غیر اینصورت مقدار FALSE را به عنوان خروجی برگرداند. بهترین و بدترین حالت در اجرای این الگوریتم در چه صورتی رخ می‌دهد؟ الگوریتم خود را، در صورت لزوم، به گونه‌ای تغییر دهید که به محض تشخیص جواب، اجرای آن خاتمه یابد.

◃ **پاسخ سوال ۲۰.**

شبه کد الگوریتم مورد نظر در قالب الگوریتم (۳.۱) نشان داده شده است. الگوریتم ارائه شده به درستی کار می‌کند اما بدترین و بهترین حالت اجرای آن یکسان است. این بدین معنی است که حلقه‌ی موجود در این الگوریتم همواره n بار تکرار می‌شود.

در الگوریتم (۳.۱) بهترین حالت زمانی رخ می‌دهد که اعداد موجود در نیمه‌ی ابتدایی آرایه‌ی A بزرگتر یا مساوی با عدد m باشند. به طور مثال برای آرایه‌ای هشت خانه‌ای و $m = ۱۰$، یکی از بهترین حالت‌ها می‌تواند مانند آرایه‌ی نشان داده شده در شکل (۳.۱) باشد. در این صورت خروجی الگوریتم بعد از ارزیابی مقدار ۲۲ کاملاً مشخص خواهد شد و تکرارهای بعدی حلقه تاثیری بر خروجی الگوریتم ندارد.

یکی از بدترین حالات در اجرای الگوریتم (۳.۱) زمانی است که اعداد آرایه‌ی A به صورت یکی در میان بزرگتر و کوچکتر از عدد m باشند. در این صورت الگوریتم برای رسیدن به جواب باید تمام عناصر آرایه‌ی

الگوریتم ۳.۱ شمارش اعداد بزرگتر یا کوچکتر از یک مقدار خاص

```
function MoreOrLess(A, m)
    g = 0
    l = 0
    n = A.length
    for i = 1 to n
        if A[i] ≥ m
            g = g + 1
        else
            l = l + 1
        end if
    end for
    if g ≥ l
        return TRUE
    else
        return FALSE
    end if
end function
```

۱	۲	۳	۴	۵	۶	۷	۸
۱۲	۱۲	۱۴	۲۲	۱	۲	۳	۲۳

شکل (۳.۱): یکی از بهترین ورودی‌ها برای الگوریتم (۳.۱)

A را مورد بررسی قرار دهد. به عنوان مثال برای آرایه‌ای با هشت عنصر و $m = ۱۰$، یکی از بدترین حالات می‌تواند به صورت نشان داده شده در شکل (۴.۱) باشد.

با توجه به توضیحات ارائه شده شبه کد الگوریتم تغییر یافته در الگوریتم (۴.۱) آورده شده است. زوج یا فرد بودن تعداد خانه‌های آرایه در این الگوریتم مهم است و به همین جهت در این الگوریتم از تابع سقف استفاده شده است.

در هر دور از اجرای حلقه‌ی موجود در الگوریتم (۴.۱) یکی از سه حالت زیر روی می‌دهد:

- مقدار متغیر g بزرگتر یا مساوی نصف تعداد کل خانه‌های آرایه‌ی A باشد. در این صورت می‌توان گفت حداقل نیمی از اعداد آرایه بزرگتر یا مساوی عدد m هستند و در نتیجه مقدار l در تکرارهای بعدی حلقه هرگز از g بیشتر نخواهد شد.

۱	۲	۳	۴	۵	۶	۷	۸
۲۳	۹	۱۵	۹	۱۶	۸	۱۲	۳

شکل (۴.۱): یکی از بدترین ورودی‌های ممکن برای الگوریتم (۳.۱)

الگوریتم ۴.۱ شمارش اعداد بزرگتر یا کوچکتر از یک مقدار خاص

```
function ModifiedMoreOrLess(A, m)
    g = 0
    l = 0
    n = A.length
    for i = 1 to n
        if A[i] ≥ m
            g = g + 1
        else
            l = l + 1
        end if
        if g ≥ ⌈n/2⌉
            return TRUE
        else if l > ⌈n/2⌉
            return FALSE
        end if
    end for
    if g ≥ l
        return TRUE
    else
        return FALSE
    end if
end function
```

- مقدار متغیر l بزرگتر از نصف تعداد کل خانه‌های آرایه‌ی A باشد. اگر این‌طور باشد یعنی بیش از نیمی از اعداد آرایه کمتر از عدد m هستند و این یعنی با ادامه‌ی اجرای الگوریتم مقدار g نمی‌تواند بیشتر از l شود.
- هیچ یک از دو حالت قبلی برقرار نباشند و اجرای حلقه ادامه یابد.

به این ترتیب با تغییرات اعمال شده توانستیم بین بهترین و بدترین حالت اجرای الگوریتم تفاوت قائل شویم و به محض تشخیص جواب به اجرای الگوریتم پایان دهیم.

◄ **سوال ۲۱.** الگوریتم (۵.۱) نشان دهنده‌ی شبه کد الگوریتم جستجوی ترتیبی[۶] است. اگر عنصر با مقدار k در آرایه‌ی A پیدا شود آنگاه اندیس آن خانه برگشت داده شده و اجرای الگوریتم خاتمه می‌یابد. در غیر این صورت مقدار -1 به نشانه‌ی عدم وجود عنصر مورد نظر در آرایه بازگردانده می‌شود. تعداد مقایسات انجام شده در بدترین حالت و حالت متوسط این الگوریتم را به دست آورید (فرض کنید مقادیر آرایه‌ی A غیر تکراری هستند).

[۶] Sequential search

الگوریتم ۵.۱ جستجوی ترتیبی

```
function SequentialSearch(A, k)
    n = A.length
    for i = 1 to n
        if A[i] == k
            return i
        end if
    end for
    return −1
end function
```

◁ **پاسخ سوال ۲۱.**

تحلیل بدترین حالت

در الگوریتم جستجوی ترتیبی می‌توان دو حالت زیر را به عنوان بدترین حالت در نظر گرفت:

- مقدار مورد جستجو در آخرین خانه‌ی آرایه‌ی A باشد.
- مقدار مورد جستجو در آرایه‌ی A وجود نداشته باشد.

برای هر یک از این دو حالت به بررسی تعداد مقایسات انجام شده می‌پردازیم.

طبق فرض سوال هنگامی که مقدار مورد جستجو در آخرین خانه از آرایه قرار داشته باشد بدین معنی است که هیچ‌کدام از عناصر موجود در خانه‌های ۱ تا $n-1$ با مقدار مورد جستجو برابر نیستند. بنابراین الگوریتم در این حالت مقدار مورد جستجو را با تمام عناصر آرایه‌ی A مقایسه می‌کند تا مقدار مورد نظر را در خانه‌ی با اندیس n پیدا کند. پس در چنین حالتی n مقایسه انجام می‌شود.

در حالت دوم مقدار مورد جستجو با تمام عناصر آرایه مقایسه شده و به این نتیجه می‌رسیم که مقدار مورد نظر در آرایه وجود ندارد. پس در این حالت نیز به n مقایسه نیاز داریم.

به این ترتیب می‌توان گفت الگوریتم جستجوی ترتیبی در بدترین حالت دارای مرتبه‌ی زمانی $O(n)$ است.

تحلیل حالت متوسط

به طور کلی در الگوریتم‌های جستجو باید دو حالت مختلف را در تحلیل حالت متوسط در نظر بگیریم:

- مقدار مورد جستجو حتماً در آرایه‌ی A وجود دارد.
- ممکن است مقدار مورد جستجو در آرایه‌ی A وجود نداشته باشد.

اگر مقدار مورد جستجو در آرایه موجود باشد آنگاه این مقدار می‌تواند در هر یک از n خانه‌ی آرایه قرار داشته باشد. با این فرض که احتمال وجود مقدار مورد جستجو در هر یک از n خانه‌ی آرایه مساوی یکدیگر و برابر با $1/n$ باشد باید به این سوال پاسخ دهیم که اگر مقدار مورد جستجو در خانه‌ی اول باشد به چند مقایسه نیاز داریم؟ اگر در خانه‌ی دوم باشد به چند مقایسه نیاز داریم؟ و به همین ترتیب تا خانه‌ی nام.

اگر مقدار مورد جستجو در خانه‌ی اول باشد به یک مقایسه، اگر در خانه‌ی دوم باشد به دو مقایسه و به همین ترتیب اگر در خانه nام باشد به n مقایسه نیاز است. به این ترتیب متوسط تعداد مقایسات به صورتی که در ادامه آمده است به دست می‌آید:

$$A(n) = \sum_{i=۱}^{n} \frac{۱}{n} \cdot i = \frac{۱}{n} \sum_{i=۱}^{n} i = \frac{۱}{n} \cdot \frac{n(n+۱)}{۲} = \frac{n+۱}{۲}$$

حال به بررسی حالتی می‌پردازیم که ممکن است مقدار مورد جستجو در آرایه وجود نداشته باشد. در چنین حالتی دارای $n+۱$ احتمال مختلف هستیم. n حالت برای هنگامی که مقدار مورد جستجو یافت شود و یک حالت برای هنگامی که چنین مقداری در آرایه وجود نداشته باشد. طبق مطالب گفته شده در تحلیل بدترین حالت، اگر مقدار مورد جستجو در آرایه نباشد آنگاه به n مقایسه نیاز داریم. اگر تمام احتمالات ممکن را برابر با هم و مساوی با $۱/(n+۱)$ در نظر بگیریم آنگاه متوسط تعداد مقایسات به صورت زیر است:

$$\begin{aligned} A(n) &= \left(\frac{۱}{n+۱} \sum_{i=۱}^{n} i\right) + \left(\frac{۱}{n+۱} \cdot n\right) \\ &= \left(\frac{۱}{n+۱} \cdot \frac{n(n+۱)}{۲}\right) + \frac{n}{n+۱} \\ &= \frac{n}{۲} + \frac{n}{n+۱} \\ &= \frac{n}{۲} + ۱ - \frac{۱}{n+۱} \end{aligned} \tag{۳۶.۱}$$

از آنجا که اگر n به بی‌نهایت میل کند عبارت $۱/(n+۱)$ در (۳۶.۱) به صفر میل می‌کند در نتیجه می‌توان رابطه‌ی (۳۶.۱) را به صورتی که در ادامه آمده است در نظر گرفت:

$$A(n) \approx \frac{n}{۲} + ۱ = \frac{n+۲}{۲} = \frac{n+۱}{۲} + \frac{۱}{۲}$$

همان‌طور که مشخص است متوسط تعداد مقایسات در حالتی که ممکن است مقدار مورد جستجو در آرایه موجود نباشد تنها به اندازه‌ی ۱/۲ بیشتر از متوسط تعداد مقایسات در حالتی است که از وجود مقدار مورد جستجو در آرایه مطمئن هستیم. هنگامی که مقدار n بسیار بزرگ باشد می‌توان از مقدار ۱/۲ چشم‌پوشی کرد و این طور نتیجه گرفت که متوسط تعداد مقایسات در جستجوی ترتیبی، چه مقدار مورد جستجو در آرایه باشد و چه نباشد، برابر با $(n+۱)/۲$ است. به عنوان نتیجه‌گیری باید گفت الگوریتم جستجوی ترتیبی در حالت متوسط از مرتبه‌ی $O(n)$ است.

◄ **سوال ۲۲.** آرایه‌ی $A[۱..n]$ که حاوی اعداد متمایز است را در نظر بگیرید. می‌خواهیم با استفاده از الگوریتم جستجوی ترتیبی به دنبال مقداری خاص در آرایه‌ی A بگردیم. فرض کنید به احتمال ۲۵ درصد مقدار مورد جستجو در آرایه‌ی A وجود نداشته باشد. همچنین فرض کنید اگر مقدار مورد جستجو در آرایه‌ی A وجود داشته باشد آنگاه به احتمال ۷۵ درصد در نیمه‌ی ابتدایی آرایه قرار دارد. با در نظر گرفتن مفروضات بیان شده، متوسط تعداد مقایسات انجام شده توسط الگوریتم جستجوی ترتیبی چقدر است؟

◃ **پاسخ سوال ۲۲.**

رابطه‌ی (۳۷.۱) نشان دهنده‌ی فرمول متوسط تعداد مقایسات در جستجوی ترتیبی است. این رابطه بیانگر این است که به احتمال p_i، نیاز به C_i مقایسه برای یافتن مقدار مورد جستجو داریم.

$$A(n) = \sum_{i=۱}^{n} p_i.C_i \tag{۳۷.۱}$$

با توجه به فرضیات بیان شده در صورت سوال و رابطه‌ی (۳۷.۱) متوسط تعداد مقایسات برابر است با:

$$A(n) = \underbrace{(۰/۲۵ \cdot n)}_{\text{بخش اول}} + \underbrace{۰/۷۵ \cdot \left(\left(۰/۷۵ \sum_{i=۱}^{n/۲} \frac{۱}{n/۲} \cdot i \right) + \left(۰/۲۵ \sum_{i=(n/۲)+۱}^{n} \frac{۱}{n/۲} \cdot i \right) \right)}_{\text{بخش دوم}}$$

بخش اول رابطه‌ی فوق مربوط به حالتی است که به احتمال ۲۵ درصد مقدار مورد جستجو در آرایه وجود ندارد. در چنین حالتی نیاز به n مقایسه خواهد بود. بخش دوم این رابطه مربوط به حالتی است که به احتمال ۷۵ درصد مقدار مورد جستجو در آرایه وجود دارد. در این حالت با توجه به مفروضات سوال باید دو حالت مختلف را در نظر بگیریم. حالت اول هنگامی است که مقدار مورد جستجو در نیمه‌ی اول آرایه است و حالت دوم هنگامی است که این مقدار در نیمه‌ی دوم آرایه است.

با ساده‌سازی $A(n)$ به رابطه‌ی $A(n) = (۱۷n + ۱۲)/۳۲$ می‌رسیم که این مقدار به طور تقریبی برابر با $n/۲$ است.

◄ **سوال ۲۳.** از الگوریتم جستجوی ترتیبی برای جستجو در یک آرایه‌ی مرتب نیز می‌توان استفاده کرد. الگوریتمی بنویسید که با در اختیار داشتن آرایه‌ای مانند A که اعداد آن به صورت صعودی مرتب هستند، به جستجوی عنصری با مقدار k بپردازد. الگوریتم شما باید نسبت به الگوریتم جستجوی ترتیبی در یک آرایه نامرتب، تعداد مقایسات کمتری انجام دهد زیرا به محض اینکه مقدار مورد جستجو از عنصر نورد بررسی در آرایه کوچکتر باشد آنگاه عدم موفقیت برای یافتن مقدار مورد جستجو اعلام شده و اجرای الگوریتم خاتمه می‌یابد. مرتبه‌ی زمانی الگوریتم را در بدترین، بهترین و حالت متوسط به دست آورید.

◃ **پاسخ سوال ۲۳.**

شبه کد الگوریتم جستجوی ترتیبی در یک آرایه‌ی مرتب در قالب الگوریتم (۶.۱) نشان داده شده است.

تحلیل بهترین حالت

بهترین حالت زمانی رخ می‌دهد که مقدار مورد جستجو در خانه‌ی اول آرایه و یا کوچکتر از آن باشد که در این صورت با یک بار اجرا شدن حلقه از آن خارج شده و اجرای الگوریتم خاتمه می‌یابد. پس تعداد تکرار حلقه در بهترین حالت برابر با یک است و در نتیجه مرتبه‌ی زمانی بهترین حالت اجرای الگوریتم $O(۱)$ است.

تحلیل بدترین حالت

بیشترین تعداد تکرار حلقه در یکی از حالات زیر رخ می‌دهد:

- مقدار مورد جستجو در آخرین خانه‌ی آرایه قرار داشته باشد.
- مقدار مورد جستجو از مقدار آخرین خانه‌ی آرایه بزرگتر باشد.

الگوریتم ۶.۱ جستجوی ترتیبی در یک آرایه‌ی مرتب

```
function SortedSequentialSearch(A, k)
    n = A.length
    for i = 1 to n
        if A[i] == k
            return i
        else if A[i] > k
            return −1
        end if
    end for
    return −1
end function
```

- مقدار مورد جستجو بزرگتر از عنصر ماقبل آخر و کوچکتر از عنصر آخر آرایه باشد.

در هر یک از حالات بیان شده تعداد تکرار حلقه برابر با n است و این یعنی در بدترین حالت دارای مرتبه‌ی زمانی $O(n)$ هستیم.

تحلیل حالت متوسط

برای تحلیل حالت متوسط باید دو حالت مختلف را در نظر بگیریم:

- مقدار مورد جستجو حتماً در آرایه وجود دارد.
- ممکن است مقدار مورد جستجو در آرایه وجود نداشته باشد.

اگر فرض کنیم مقدار مورد جستجو در آرایه وجود دارد آنگاه n احتمال مختلف برای محل قرارگیری مقدار مورد جستجو قابل تصور است. حالت اول هنگامی است که مقدار مورد جستجو در خانه‌ی اول آرایه باشد. حالت دوم هنگامی است که مقدار مورد جستجو در خانه‌ی دوم آرایه باشد و به همین ترتیب تا حالت nام که مقدار مورد جستجو در خانه nام آرایه باشد. می‌دانیم اگر مقدار مورد نظر در خانه iام آرایه باشد برای یافتن آن به i مقایسه نیاز داریم. اگر احتمال وجود مقدار مورد جستجو در هر یک از خانه‌های ۱ تا n را یکسان و برابر با $۱/n$ در نظر بگیریم آنگاه متوسط تعداد مقایسات برابر است با:

$$A(n) = \sum_{i=۱}^{n} \frac{۱}{n} \cdot i = \frac{۱}{n} \sum_{i=۱}^{n} i = \frac{۱}{n} \cdot \frac{n(n+۱)}{۲} = \frac{n+۱}{۲}$$

اگر این احتمال وجود داشته باشد که مقدار مورد جستجو در آرایه موجود نباشد، آنگاه دارای $۲n + ۱$ حالت مختلف خواهیم بود. حالت اول هنگامی است که مقدار مورد جستجو کوچکتر از مقدار خانه‌ی اول آرایه باشد . حالت دوم هنگامی رخ می‌دهد که مقدار مورد جستجو در خانه‌ی اول آرایه قرار داشته باشد. حالت سوم زمانی اتفاق می‌افتد که مقدار مورد جستجو دارای مقداری بزرگتر از خانه‌ی اول آرایه و کوچکتر از خانه‌ی دوم آرایه باشد. حالت چهارم زمانی روی می‌دهد که مقدار مورد نظر در خانه‌ی دوم آرایه قرار داشته باشد. به همین

ترتیب تا حالت $۲n + ۱$ ام که زمانی روی می‌دهد که مقدار مورد جستجو از مقدار آخرین خانه‌ی آرایه بزرگتر باشد. در ادامه به تعیین تعداد مقایسات انجام شده در هر از این حالات می‌پردازیم.

در حالتی که مقدار مورد جستجو کوچکتر از مقدار خانه‌ی اول آرایه باشد و همچنین حالتی که مقدار مورد جستجو در خانه‌ی اول آرایه قرار دارد تنها یک مقایسه لازم است. در حالتی که مقدار مورد جستجو در خانه‌ی دوم آرایه باشد و همچنین حالتی که مقدار مورد جستجو عددی بین اولین عنصر و دومین عنصر آرایه باشد دو مقایسه لازم است و به همین ترتیب ادامه می‌دهیم تا در نهایت در هر سه حالت زیر به n مقایسه نیاز خواهد بود:

- مقدار مورد جستجو در آخرین خانه‌ی آرایه باشد.
- مقدار مورد جستجو بزرگتر از خانه‌ی ماقبل آخر و کوچکتر از خانه‌ی آخر آرایه باشد.
- مقدار مورد جستجو بزرگتر از آخرین خانه‌ی آرایه باشد.

با توجه به توضیحات ارائه شده متوسط تعداد مقایسات برابر است با:

$$\begin{aligned} A(n) &= \left(\sum_{i=۱}^{n-۱} \frac{۲}{۲n+۱} \cdot i\right) + \left(\frac{۳}{۲n+۱} \cdot n\right) \\ &= \left(\frac{۲}{۲n+۱} \cdot \frac{n(n-۱)}{۲}\right) + \frac{۳n}{۲n+۱} \\ &= \frac{n^۲ + ۲n}{۲n+۱} \\ &= \frac{n+۱}{۲} + \frac{n-۱}{۲n+۱} \end{aligned}$$

اگر مقدار n به بی‌نهایت میل کند می‌توان $A(n)$ را به صورت زیر تقریب زد:

$$A(n) \approx \frac{n+۱}{۲} + \frac{۱}{۲} \approx \frac{n+۱}{۲}$$

به این ترتیب می‌توان گفت الگوریتم جستجوی ترتیبی در یک آرایه مرتب، چه مقدار مورد جستجو در آرایه موجود باشد و چه نباشد، از مرتبه‌ی $O(n)$ است.

◄ **سوال ۲۴.** جستجوی یک مقدار در یک آرایه‌ی مرتب صعودی به روشی که در ادامه آمده است را در نظر بگیرید.

مقدار مورد جستجو با مقدار خانه‌ی میانی آرایه مقایسه می‌شود. سه حالت ممکن بعد از عمل مقایسه عبارت‌اند از:

۱. مقدار مورد جستجو با مقدار خانه‌ی میانی آرایه برابر است که در این صورت عمل جستجو با موفقیت خاتمه می‌یابد.

۲. مقدار مورد جستجو از مقدار خانه‌ی میانی آرایه کوچکتر است که در این صورت باید به جستجوی مقدار مورد نظر در خانه‌های قبل از خانه‌ی میانی، یعنی نیمه‌ی اول آرایه، بپردازیم.

۳. مقدار مورد جستجو از مقدار خانه‌ی میانی آرایه بزرگتر است که در این صورت باید به دنبال مقدار مورد نظر در خانه‌های بعد از خانه‌ی میانی، یعنی نیمه‌ی دوم آرایه، باشیم.

حالات ۲ و ۳ منجر به نادیده گرفتن بخشی از آرایه در ادامه‌ی روند اجرای الگوریتم می‌شوند. این روند به همین ترتیب ادامه یافته و در هر مرحله نیمی از عناصر بخش باقی مانده از آرایه نادیده گرفته می‌شوند تا اینکه یا مقدار مورد جستجو پیدا شود و یا به این نتیجه برسیم که این مقدار در آرایه وجود ندارد. به این روش جستجو، جستجوی دودویی[۷] گفته می‌شود. شبه کد الگوریتم جستجوی دودویی در الگوریتم (۷.۱) نشان داده شده است. مرتبه‌ی زمانی الگوریتم جستجوی دودویی را در بهترین، بدترین و حالت متوسط به دست آورید (احتمال وجود مقدار مورد جستجو در هر یک از خانه‌های آرایه و همچنین احتمال اینکه مقدار مورد جستجو در آرایه موجود نباشد برابر هستند).

الگوریتم ۷.۱ جستجوی دودویی

```
function BinarySearch(A, k)
    n = A.length
    begin = 1
    end = n
    while start ≤ end
        middle = ⌊start + end⌋/2
        if k == A[middle]
            return middle
        else if k < A[middle]
            middle = end − 1
        else
            middle = begin + 1
        end if
    end while
    return −1
end function
```

◁ **پاسخ سوال ۲۴.**

برای سادگی تحلیل، فرض می‌کنیم تعداد عناصر آرایه برابر با $n = ۲^k - ۱$ $(k \geqslant ۱)$ است. در حالت کلی می‌توان گفت اگر در ابتدای یک دور از اجرای حلقه دارای $۲^k - ۱$ عنصر در محدوده‌ی $begin$ تا end باشیم آنگاه $۲^{k-۱} - ۱$ عنصر در نیمه‌ی سمت چپ، یک عنصر به عنوان عنصر میانی و $۲^{k-۱} - ۱$ عنصر در نیمه‌ی سمت راست آرایه قرار خواهند داشت. در نتیجه در ابتدای دور بعدی اجرای حلقه تعداد عناصر موجود در محدوده‌ی $begin$ تا end به $۲^{k-۱} - ۱$ عنصر کاهش می‌یابد زیرا روند اجرای الگوریتم یا در نیمه‌ی سمت چپ و یا در نیمه‌ی سمت راست آرایه ادامه خواهد یافت.

تحلیل بدترین حالت

[۷] Binary search

با توجه به توضیحات بیان شده، در هر مرحله از اجرای حلقه یک واحد از توان عدد ۲ کم می‌شود. همچنین می‌دانیم که آخرین دور از اجرای حلقه زمانی روی می‌دهد که تعداد عناصر موجود در محدوده $begin$ تا end به یک عنصر برسد و این زمانی اتفاق می‌افتد که مقدار k یک باشد (مقدار عبارت $۲^k - ۱$ زمانی برابر با یک خواهد شد که مقدار k برابر با یک باشد). به این ترتیب می‌توان گفت در بدترین حالت از اجرای الگوریتم، با فرض اینکه تعداد عناصر آرایه برابر با $۲^k - ۱$ است، تعداد دفعات اجرای حلقه برابر با k است و این یعنی مرتبه‌ی زمانی الگوریتم جستجوی دودویی از مرتبه‌ی $O(k)$ است. با در نظر گرفتن رابطه‌ی $n = ۲^k - ۱$ و به دست آوردن k بر حسب n می‌توان نتیجه گرفت $k = \lg(n + ۱)$ و در نتیجه مرتبه‌ی زمانی الگوریتم جستجوی دودویی در بدترین حالت از مرتبه‌ی $O(\lg n)$ است.

تحلیل بهترین حالت

بهترین حالت زمانی روی می‌دهد که در اولین دور از اجرای حلقه مقدار مورد جستجو با مقدار خانه‌ی میانی آرایه برابر باشد که در این صورت با انجام یک مقایسه مقدار مورد جستجو پیدا می‌شود. پس الگوریتم جستجوی دودویی در بهترین حالت از مرتبه‌ی $O(۱)$ است.

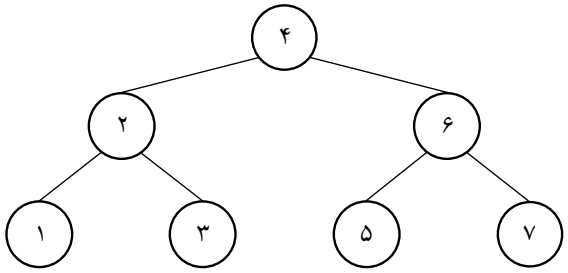
تحلیل حالت متوسط

به منظور درک بهتر تحلیل حالت متوسط بهتر است روند اجرای الگوریتم جستجوی دودویی را با استفاده از یک درخت تصمیم دودویی[۸] مدل کنیم. گره‌های چنین درختی با اندیس هر خانه‌ای از آرایه که در هر دور از اجرای حلقه با مقدار مورد جستجو مقایسه می‌شود برچسب‌گذاری می‌شود. عناصری که در صورت کوچکتر بودن مقدار مورد جستجو از مقدار میانی، در دورهای بعدی با مقدار مورد جستجو مقایسه می‌شوند در زیر درخت چپ قرار می‌گیرند. عناصری که در صورت بزرگتر بودن مقدار مورد جستجو از مقدار میانی، در دورهای بعدی با مقدار مورد جستجو مقایسه می‌شوند در زیر درخت راست قرار می‌گیرند. اگر فرض کنیم دارای آرایه‌ای با هفت عنصر هستیم و اندیس اولین خانه‌ی آرایه از عدد یک شروع شود آنگاه درخت تصمیم دودویی برای چنین آرایه‌ای مانند شکل (۵.۱) خواهد بود.

شکل (۵.۱) نشان می‌دهد که در دور اول از اجرای حلقه، عنصر میانی در خانه‌ی چهار آرایه قرار دارد. با مقایسه‌ی مقدار مورد جستجو با مقدار موجود در خانه‌ی چهارم، با فرض اینکه مقدار مورد جستجو در خانه‌ی چهارم آرایه نباشد، یا به نیمه‌ی سمت چپ آرایه (خانه‌های ۱ تا ۳) و یا به نیمه‌ی سمت راست آرایه (خانه‌های ۴ تا ۷) می‌رویم. اگر به نیمه‌ی سمت چپ آرایه برویم آنگاه در دور بعدی از اجرای حلقه، عنصر میانی در خانه با اندیس دو قرار دارد و اگر به نیمه‌ی سمت راست آرایه برویم آنگاه عنصر میانی در خانه با اندیس شش قرار دارد و به همین ترتیب.

در حالت کلی می‌دانیم که چنین درختی تقریباً متوازن است زیرا در هر مرحله از اجرای الگوریتم، آرایه از نقطه‌ی میانی به دو قسمت تقسیم می‌شود. با توجه به فرمول‌های موجود برای درختان دودویی می‌توان گفت درخت تصمیم یک آرایه‌ی n عنصری دارای حداکثر $\lfloor \lg n \rfloor + ۱$ سطح است (با این فرض که شماره‌ی سطوح از یک شروع شود). تعداد سطوح برابر است با حداکثر تعداد مقایسات برای یافتن یک مقدار در یک آرایه‌ی n عنصری با استفاده از الگوریتم جستجوی دودویی. با توجه به اینکه مقدار n را به صورت $n = ۲^k - ۱$ فرض کردیم پس درخت تصمیم آرایه‌ای با $۲^k - ۱$ عنصر یک درخت دودویی با حداکثر تعداد گره خواهد بود. با به دست آوردن k برحسب n می‌توان به این نتیجه رسید که درخت تصمیم دارای k سطح است و k برابر است

[۸] Binary decision tree



شکل (۵.۱): درخت تصمیم دودویی برای آرایه‌ای با هفت عنصر

با $k = \lg(n+۱)$.

در ادامه با استفاده از ایده‌ی درخت تصمیم دودویی به تحلیل دو حالت زیر در تحلیل حالت متوسط می‌پردازیم:

- مقدار مورد جستجو در آرایه وجود دارد.
- ممکن است مقدار مورد جستجو در آرایه وجود نداشته باشد.

اگر فرض کنیم مقدار مورد جستجو در آرایه وجود دارد آنگاه مقدار مورد جستجو می‌تواند با احتمال مساوی در هر یک از n خانه‌ی آرایه قرار داشته باشد (این احتمال برای هر خانه برابر با $۱/n$ است). اگر درخت تصمیم فرآیند جستجو را در نظر بگیریم آنگاه می‌توان دید که اگر مقدار مورد جستجو در ریشه‌ی درخت باشد آنگاه تنها به یک مقایسه برایب یافتن آن نیاز است. اگر مقدار مورد جستجو در هر یک از دو خانه‌ی واقع در سطح دوم درخت باشد به دو مقایسه نیاز است و به همین ترتیب. در حالت کلی می‌توان گفت اگر مقدار مورد جستجو در خانه‌ای در سطح kام باشد آنگاه نیاز به k مقایسه برای یافتن آن داریم. می‌دانیم که در یک درخت دودویی تعداد گره‌ها در سطح i برابر با $۲^{i-۱}$ است و تعداد سطوح در حالتی که دارای $۲^k - ۱$ گره هستیم برابر با k است. حال با توجه به اینکه تعداد سطوح، تعداد گره در هر سطح و همچنین تعداد مقایسه برای هر سطح را در اختیار داریم می‌توان متوسط تعداد مقایسات را از رابطه‌ی زیر به دست آورد:

$$\begin{aligned} A(n) &= \frac{۱}{n}\sum_{i=۱}^{k} i \cdot ۲^{i-۱} \\ &= \frac{۱}{۲n}\sum_{i=۱}^{k} i \cdot ۲^{i} \end{aligned} \tag{۳۸.۱}$$

با توجه به اینکه حاصل عبارت $\sum_{i=۱}^{k} i \cdot ۲^i$ برابر با $(k-۱)۲^{k+۱} + ۲$ است می‌توان رابطه‌ی (۳۸.۱) را به صورت نشان داده شده در رابطه‌ی (۳۹.۱) نوشت.

$$A(n) = \frac{۱}{۲n}\Big((k-۱)۲^{k+۱} + ۲\Big) \tag{۳۹.۱}$$

با ساده‌سازی (۳۹.۱) داریم:

$$A(n) = \frac{۱}{۲n}\Big((k-۱)۲^{k+۱} + ۲\Big)$$

$$
\begin{aligned}
&= \frac{۱}{n}\Big((k-۱)۲^k + ۱\Big) \\
&= \frac{۱}{n}\Big(k۲^k - ۲^k + ۱\Big) \\
&= \frac{k۲^k - (۲^k - ۱)}{n}
\end{aligned} \tag{۴۰.۱}
$$

اگر در (۴۰.۱) به جای $۲^k - ۱$ معادل آن یعنی n را قرار دهیم به رابطه‌ی (۴۱.۱) می‌رسیم.

$$A(n) = \frac{k۲^k}{n} - ۱ \tag{۴۱.۱}$$

با توجه به اینکه فرض کردیم $n = ۲^k - ۱$ پس $۲^k = n + ۱$. با جایگذاری $n + ۱$ به جای ۲^k در (۴۱.۱) خواهیم داشت:

$$
\begin{aligned}
A(n) &= \frac{k(n+۱)}{n} - ۱ \\
&= \frac{kn + k}{n} - ۱ \\
&= k + \frac{k}{n} - ۱
\end{aligned} \tag{۴۲.۱}
$$

هنگامی که n به بی‌نهایت میل کند k/n به صفر میل می‌کند. با در نظر گرفتن این موضوع می‌توان (۴۲.۱) را به صورت زیر در تقریب زد:

$$A(n) \approx k - ۱ = \lg(n+۱) - ۱$$

به این ترتیب می‌توان گفت متوسط تعداد مقایسات در حالتی که مقدار مورد جستجو در آرایه وجود دارد برابر با $\lg(n+۱) - ۱$ است.

اگر حالتی را در نظر بگیریم که ممکن است مقدار مورد جستجو در آرایه وجود نداشته باشد آنگاه اگر مقدار مورد جستجو در آرایه وجود داشته باشد دارای n مکان احتمالی برای قرارگیری آن هستیم. همچنین دارای $n+۱$ احتمال دیگر برای حالتی هستیم که مقدار مورد جستجو در آرایه وجود ندارد. احتمال اول از این $n+۱$ احتمال مختلف هنگامی است که مقدار مورد جستجو از عنصر اول آرایه کوچکتر باشد. احتمال دوم زمانی رخ می‌دهد که مقدار مورد جستجو از عنصر اول آرایه بزرگتر و از عنصر دوم کوچکتر باشد و به همین ترتیب تا احتمال $n + ۱$ ام که زمانی اتفاق می‌افتد که مقدار مورد جستجو از عنصر nام آرایه بزرگتر باشد. در هر یک از این $n + ۱$ حالت به k مقایسه نیاز داریم تا به عدم وجود مقدار مورد جستجو در آرایه پی ببریم. بدین ترتیب دارای $۲n + ۱$ احتمال مختلف هستیم که باید در محاسبات خود لحاظ کنیم. با توجه به توضیحات داده شده برای متوسط تعداد مقایسات داریم:

$$A(n) = \frac{۱}{۲n+۱}\left(\left(\sum_{i=۱}^{k} i \cdot ۲^{i-۱}\right) + (n+۱)k\right) \tag{۴۳.۱}$$

اگر در (۴۳.۱) به جای عبارت $\sum_{i=۱}^{k} i \cdot ۲^{i-۱}$ معادل آن یعنی $(k-۱)۲^k + ۱$ را قرار دهیم به رابطه‌ی (۴۴.۱) می‌رسیم.

$$A(n) = \frac{\Big((k-۱)۲^k + ۱\Big) + (n+۱)k}{۲n+۱} \tag{۴۴.۱}$$

با ساده‌سازی (۴۴.۱) داریم:

$$\begin{aligned}
A(n) &= \frac{\Big((k-۱)۲^k + ۱\Big) + \Big(۲^k - ۱ + ۱\Big)k}{۲(۲^k - ۱) + ۱} \\
&= \frac{\Big(k۲^k - ۲^k + ۱\Big) + k۲^k}{۲^{k+۱} - ۱} \\
&= \frac{k۲^{k+۱} - ۲^k + ۱}{۲^{k+۱} - ۱} \\
&\approx \frac{k۲^{k+۱} - ۲^k}{۲^{k+۱}} \\
&= k - \frac{۱}{۲} = \lg(n+۱) - \frac{۱}{۲}
\end{aligned}$$

پس متوسط تعداد مقایسات در حالتی که ممکن است مقدر مورد جستجو در آرایه وجود نداشته باشد برابر با $\lg(n+۱) - (۱/۲)$ است.

با توجه به اینکه متوسط تعداد مقایسات برای حالتی که مقدار مورد جستجو در آرایه وجود دارد برابر با $\lg(n+۱) - ۱$ و برای حالتی که ممکن است مقدار مورد جستجو در آرایه وجود نداشته باشد برابر با $\lg(n+۱) - (۱/۲)$ است پس می‌توان گفت مرتبه‌ی زمانی الگوریتم جستجوی دودویی در حالت متوسط برابر با $O(\lg n)$ است.

◄ **سوال ۲۵.** الگوریتم مرتب‌سازی یک آرایه‌ی n عنصری را در نظر بگیرید که در ابتدا کوچکترین عنصر آرایه را پیدا کرده و با عنصر ابتدایی آرایه، یعنی $A[۱]$، جابجا می‌کند. سپس دومین کوچکترین عنصر را پیدا کرده و با دومین عنصر، یعنی $A[۲]$، جابجا می‌کند و به همین ترتیب تا انتهای آرایه. این روش مرتب سازی، مرتب‌سازی انتخابی[۹] نام دارد. شبه کد الگوریتم مرتب‌سازی انتخابی را نوشته و سپس مرتبه‌ی زمانی بهترین و بدترین حالت اجرای آن را به دست آورید.

◃ **پاسخ سوال ۲۵.**

شبه کد الگوریتم مرتب‌سازی انتخابی در قالب الگوریتم (۸.۱) آورده شده است. مرتبه‌ی زمانی این الگوریتم در بهترین و بدترین حالت یکسان است زیرا در هر دوی این حالات مجموع تعداد تکرار دو حلقه‌ی موجود در الگوریتم یکسان هستند. تحلیل مرتبه‌ی زمانی الگوریتم مرتب‌سازی انتخابی در ادامه آمده است.

در هر دو حالت بهترین و بدترین، در دور اول از اجرای حلقه‌ی بیرونی، حلقه‌ی درونی $n-۱$ بار اجرا می‌شود و کوچکترین عنصر یافته شده در محدوده‌ی $A[۱..n]$ با مقدار موجود در $A[۱]$ جابجا می‌شود. در دور دوم از اجرای حلقه‌ی بیرونی، حلقه‌ی درونی $n-۲$ بار اجرا شده و کوچکترین عنصر در محدوده‌ی $A[۲..n]$

[۹] Selection sort

الگوریتم ۸.۱ مرتب‌سازی انتخابی

```
procedure SelectionSort(A)
    n = A.length
    for i = 1 to n − 1
        minIdx = i
        for j = i + 1 to n
            if A[j] < A[minIdx]
                minIdx = j
            end if
        end for
        Swap(A[i], A[minIdx])
    end for
end procedure
```

یافته شده و با مقدار موجود در $A[۲]$ جابجا می‌شود. این روند به همین ترتیب ادامه می‌یابد تا دور آخر از اجرای حلقه‌ی بیرونی که در نتیجه‌ی آن حلقه‌ی درونی تنها یک بار اجرا می‌شود. بدین ترتیب مجموع تعداد تکرار دو حلقه‌ی موجود در الگوریتم مرتب‌سازی انتخابی به صورت زیر به دست می‌آید:

$$\begin{aligned} C(n) &= \sum_{i=۱}^{n-۱} (n-i) \\ &= \sum_{i=۱}^{n-۱} n - \sum_{i=۱}^{n-۱} i \\ &= n(n-۱) - \frac{n(n-۱)}{۲} \\ &= \frac{n(n-۱)}{۲} \\ &= \frac{۱}{۲}n^۲ - \frac{۱}{۲}n \end{aligned}$$

به این ترتیب می‌توان گفت الگوریتم مرتب‌سازی انتخابی در بهترین و بدترین حالت از مرتبه‌ی $\Theta(n^۲)$ است.

◄ **سوال ۲۶.** شبه کد الگوریتم مرتب‌سازی درجی بازگشتی در قالب الگوریتم (۹.۱) آورده شده است. برای مرتب‌سازی آرایه‌ی $A[۱\,..\,n]$ این الگوریتم به صورت InsertionSort(A, n) فراخوانی می‌شود. این الگوریتم برای مرتب‌سازی صعودی عناصر آرایه‌ی A ابتدا به صورت بازگشتی زیرآرایه‌ی $A[۱\,..\,n-۱]$ را مرتب کرده و سپس مقدار موجود در خانه‌ی $A[n]$ را طوری در آرایه درج می‌کند که $A[۱\,..\,n]$ به صورت مرتب درآید. برای هر یک از حالات بهترین، متوسط و بدترین، رابطه‌ای بازگشتی بنویسید که نشان دهنده مرتبه‌ی زمانی الگوریتم در آن حالت باشد. با توجه به مرتب بودن زیرآرایه‌ی $A[۱\,..\,n-۱]$ استراتژی یافتن مکان مناسب برای درج عنصر $A[n]$ را در صورت امکان بهبود دهید.

الگوریتم ۹.۱ مرتب‌سازی درجی بازگشتی

```
1: procedure InsertionSort(A, n)
2:     if n==1
3:         return
4:     else
5:         InsertionSort(A, n − 1)
6:         i = n − 1
7:         x = A[n]
8:         while i > 0 and x < A[i]
9:             A[i + 1] = A[i]
10:            i = i − 1
11:        end while
12:        A[i + 1] = x
13:    end if
14: end procedure
```

◁ **پاسخ سوال ۲۶.**

بهترین حالت در مرتب‌سازی درجی بازگشتی زمانی روی می‌دهد که مقادیر آرایه از قبل به صورت صعودی مرتب باشند. در چنین حالتی شرط حلقه برقرار نبوده و در نتیجه زمان مصرفی خطوط ۶ تا ۱۲ از مرتبه‌ی $O(۱)$ خواهد بود. بدین ترتیب می‌توان رابطه‌ی بازگشتی بهترین حالت اجرای الگوریتم مرتب‌سازی درجی بازگشتی را به صورت زیر در نظر گرفت.

$$T(n) = \begin{cases} \Theta(۱) & n = ۱ \\ T(n-۱) + \Theta(۱) & n > ۱ \end{cases}$$

در حالت متوسط باید به این نکته توجه کرد که مقدار موجود در خانه‌ی $A[n]$ می‌تواند در هر یک از مکان‌های ۱ تا n از آرایه قرار بگیرد. اگر احتمال قرارگیری عنصر $A[n]$ در هر یک از این n خانه را برابر با $۱/n$ در نظر بگیریم آنگاه متوسط تعداد تکرار حلقه برابر با $(n+۱)/۲$ است و در نتیجه زمان مصرفی خطوط ۶ تا ۱۲ از مرتبه‌ی $O(n)$ خواهد بود. با توجه به توضیحات بیان شده می‌توان رابطه‌ی بازگشتی حالت متوسط را به صورت زیر نوشت.

$$T(n) = \begin{cases} \Theta(۱) & n = ۱ \\ T(n-۱) + \Theta(n) & n > ۱ \end{cases}$$

بدترین حالت در مرتب‌سازی درجی بازگشتی زمانی روی می‌دهد که آرایه از قبل به صورت نزولی مرتب باشد. در چنین حالتی در هر فراخوانی بازگشتی دارای بیشترین تعداد تکرار حلقه خواهیم بود و این تعداد برابر با $n-۱$ است. پس مرتبه‌ی زمانی اجرای خطوط ۶ تا ۱۲ برابر با $O(n)$ است و در نتیجه رابطه‌ی بازگشتی

بدترین حالت به صورت زیر است.

$$T(n) = \begin{cases} \Theta(۱) & n = ۱ \\ T(n-۱) + \Theta(n) & n > ۱ \end{cases}$$

با توجه به اینکه زیرآرایه‌ی $A[۱\,..\,n-۱]$ مرتب است می‌توان به جای جستجوی ترتیبی از جستجوی دودویی برای یافتن مکان مناسب برای درج مقدار موجود در خانه‌ی $A[n]$ استفاده کرد. شبه کد الگوریتم تغییر یافته در قالب الگوریتم (۱۰.۱) نشان داده شده است.

الگوریتم ۱۰.۱ مرتب‌سازی درجی دودویی بازگشتی

```
procedure BinInsertionSort(A, n)
    if n==1
        return
    else
        BinInsertionSort(A, n − 1)
        begin = 1
        end = n − 1
        x = A[n]
        while begin < end
            middle = ⌊(begin + end)/2⌋
            if x < A[middle]
                end = middle
            else
                begin = middle + 1
            end if
        end while
        for i = n  downto begin + 1
            A[i] = A[i − 1]
        end for
        A[begin] = x
    end if
end procedure
```

◄ **سوال ۲۷.** مرتبه‌ی زمانی قطعه کد زیر را تعیین کنید (در این قطعه کد mod به معنی عملگر باقیمانده است).

```
for i = 1 to n
    if (i mod 2) ≠ 0
        for j = 1 to i
```

```
        x = x + 1
    end for
    for j = i to n
        y = y + 1
    end for
  end if
end for
```

◁ **پاسخ سوال ۲۷.**

برای سادگی کار ابتدا دو حلقه‌ی داخلی را تبدیل به یک حلقه می‌کنیم. این کار به این دلیل امکان‌پذیر است که به دنبال مرتبه‌ی زمانی هستیم و اینکه حلقه‌ها چه عملیاتی انجام می‌دهند، مادامی که در بدنه‌ی حلقه‌ها شمارنده‌ها تغییر داده نشوند، اهمیت ندارد. کد تغییر یافته در ادامه آمده است.

```
for i = 1 to n
  if (i mod 2) ≠ 0
    for j = 1 to n + 1
      x = x + 1
      y = y + 1
    end for
  end if
end for
```

توجه کنید که در قطعه کد بالا تعداد تکرار حلقه‌ی داخلی برابر با مجموع تعداد تکرار دو حلقه‌ی for داخلی موجود در قطعه کد اولیه است. در نتیجه پیچیدگی زمانی قطعه کد جدید با قطعه کد اولیه تفاوتی ندارد.

با در نظر گرفتن قطعه کد جدید می‌توان گفت حلقه‌ی بیرونی n بار تکرار می‌شود در حالیکه حلقه‌ی درونی تنها زمانی اجرا می‌شود که n فرد باشد. با توجه به اینکه در بازه‌ی $[۱, n]$ تقریباً نیمی از اعداد فرد و نیمی زوج هستند می‌توان گفت در نیمی از تکرارها شرط if بررسی شده و حلقه‌ی درونی اجرا نمی‌شود اما در نیمی دیگر از تکرارها به دلیل درستی شرط if حلقه‌ی درونی نیز اجرا خواهد شد. با توجه به توضیحات داده شده می‌توان زمان مصرفی قطعه کد را به صورت زیر در نظر گرفت:

$$\frac{n}{۲}O(۱) + \frac{n}{۲}O(n) = O(n) + O(n^۲) = O(n^۲)$$

پس مرتبه‌ی زمانی قطعه کد مورد نظر از مرتبه‌ی $O(n^۲)$ است.

◀ **سوال ۲۸.** ایده‌ی مرتب‌سازی حبابی[۱۰] این است که به تدریج مقادیر بزرگتر در یک آرایه به سمت انتها و مقادیر کوچکتر به سمت ابتدای آرایه حرکت کنند. این الگوریتم برای مرتب‌سازی یک آرایه تعدادی گذر[۱۱] را انجام می‌دهد. در هر گذر مقادیر خانه‌های مجاور با یکدیگر مقایسه می‌شوند. اگر ترتیب دو خانه‌ی مجاور

[۱۰] Bubble sort
[۱۱] Pass

درست نباشد، مقادیر آن دو با یکدیگر جابجا می‌شوند. در هر گذر از ابتدای آرایه شروع کرده و مقادیر خانه‌های اول و دوم با یکدیگر مقایسه می‌شوند و در صورت نیاز به جابجایی با یکدیگر جابجا می‌شوند، سپس مقادیر خانه‌های دوم و سوم با یکدیگر مقایسه می‌شوند و در صورت نیاز به جابجایی با یکدیگر جابجا می‌شوند و به همین ترتیب تا مقایسه‌ی خانه $(n-۱)$ام با خانه‌ی nام. به این ترتیب پس از پایان گذر اول بزرگترین مقدار در خانه‌ی nام آرایه قرار می‌گیرد. در گذر دوم چون می‌دانیم بزرگترین مقدار در خانه‌ی nام آرایه قرار دارد می‌توانیم مقدار واقع در این خانه را نادیده بگیریم. در نتیجه در گذر دوم، دومین بزرگترین مقدار پیدا شده و در دومین خانه از انتهای آرایه قرار می‌گیرد و به همین ترتیب تا پایان گذر $(n-۱)$ام که آرایه کاملاً مرتب خواهد بود. در اجرای هر یک از این گذرها اگر حداقل یک جابجایی انجام نشود می‌توان گفت تمام خانه‌های آرایه مرتب شده‌اند و نیازی به اجرای ادامه‌ی الگوریتم وجود ندارد. الگوریتم (۱۱.۱) شبه کد الگوریتم مرتب‌سازی حبابی را نشان می‌دهد.

الگوریتم ۱۱.۱ مرتب‌سازی حبابی

```
procedure BubbleSort(A)
    n = A.length
    pairs = n − 1
    swapDone = TRUE
    while swapDone == TRUE
        swapDone = FALSE
        for i = 1 to pairs
            if A[i] > A[i + 1]
                Swap(A[i], A[i + 1])
                swapDone = TRUE
            end if
        end for
        pairs = pairs − 1
    end while
end procedure
```

با در نظر گرفتن الگوریتم مرتب‌سازی حبابی، مرتبه‌ی زمانی حالات بهترین، متوسط و بدترین را به دست آورید.

◁ **پاسخ سوال ۲۸.**

تحلیل بهترین حالت

برای تحلیل بهترین حالت باید بررسی کنیم که کمترین تعداد تکرار حلقه‌ها در چه شرایطی روی می‌دهد. در اولین دور از اجرای حلقه‌ی while، حلقه‌ی for به طور کامل اجرا شده و $n-۱$ مقایسه انجام می‌شود. در ادامه‌ی اجرای الگوریتم ممکن است دو حالت روی دهد:

- به دلیل نامرتب بودن آرایه، حداقل یک جابجایی رخ دهد که در این صورت مقدار متغیر $swapDone$

برابر با TRUE شده و سبب می‌شود حلقه‌ی for برای حداقل یک دور دیگر نیز اجرا شود.

- آرایه از قبل به صورت صعودی مرتب باشد که در این صورت مقدار متغیر $swapDone$ برابر با FALSE خواهد ماند و اجرای الگوریتم خاتمه می‌یابد.

به این ترتیب بهترین حالت زمانی رخ می‌دهد که آرایه از قبل به صورت صعودی مرتب باشد که در این حالت تنها $n-1$ مقایسه انجام می‌شود. پس می‌توان گفت الگوریتم مرتب‌سازی حبابی در بهترین حالت از مرتبه‌ی $O(n)$ است.

تحلیل بدترین حالت

برای تحلیل بدترین حالت باید شرایطی را در نظر بگیریم که تعداد اجرا شدن حلقه‌ها بیشترین تعداد ممکن باشد که در این صورت بیشترین تعداد مقایسات انجام خواهد شد. بدترین حالت در اجرای الگوریتم مرتب‌سازی حبابی زمانی رخ می‌دهد که آرایه به صورت نزولی مرتب باشد. به این ترتیب در گذر اول و در نتیجه‌ی مقایسه خانه‌ی اول (که حاوی بزرگترین مقدار آرایه است) با دیگر خانه‌های آرایه، مقدار خانه‌ی اول تا انتهای آرایه حرکت کرده و در خانه‌ی آخر قرار می‌گیرد. پس از اجرای گذر اول، دومین بزرگترین مقدار در خانه‌ی اول آرایه قرار گرفته است. در گذر دوم، دومین بزرگترین مقدار با تمامی خانه‌های آرایه، به جز خانه‌ی انتهایی، مقایسه شده و در دومین خانه از انتهای آرایه قرار می‌گیرد. این روند به همین ترتیب ادامه می‌یابد و در هر گذر حداکثر تعداد مقایسات انجام می‌شود. به این ترتیب می‌توان گفت برای قرار دادن بزرگترین مقدار در انتهای آرایه به $n-1$ مقایسه، برای قرار دادن دومین بزرگترین مقدار در خانه‌ی یکی مانده به آخر آرایه به $n-2$ مقایسه و به همین ترتیب تا nامین بزرگترین مقدار که به هیچ مقایسه‌ای نیاز ندارد زیرا چنین مقداری خود به خود در مکان درست خود قرار گرفته است. با جمع تعداد کل مقایسات انجام شده داریم:

$$W(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2)$$

به این ترتیب نشان دادیم الگوریتم مرتب‌سازی حبابی در بدترین حالت دارای مرتبه‌ی زمانی $O(n^2)$ است.

تحلیل حالت متوسط

فرض می‌کنیم احتمال اینکه در هر گذر دلخواه از الگوریتم، هیچ جابجایی‌ای انجام نشود برابر باهم باشند. با توجه به اینکه این الگوریتم در حالت کلی دارای $n-1$ گذر است پس احتمال اینکه بعد از یک گذر دلخواه اجرای الگوریتم متوقف شود برابر با $1/(n-1)$ است. به این ترتیب باید تعداد مقایسات انجام شده بعد از پایان هر یک از این گذرها را به دست آوریم و در احتمال خروج بعد از پایان آن گذر ضرب کنیم.

اگر اجرای الگوریتم بعد از گذر اول متوقف شود آنگاه دارای $n-1$ مقایسه خواهیم بود. اگر بعد از گذر دوم اجرای الگوریتم متوقف شود دارای $(n-1)+(n-2)$ مقایسه خواهیم بود و به همین ترتیب. در ادامه فرض می‌کنیم $C(i)$ نشان دهنده‌ی تعداد مقایسات بعد از i گذر باشد. با توجه به اینکه الگوریتم مرتب‌سازی حبابی زمانی متوقف می‌شود که در یک گذر هیچ جابجایی‌ای انجام نشود، باید تمام حالاتی که این الگوریتم امکان توقف دارد را در نظر گرفت. به این ترتیب متوسط تعداد مقایسات از رابطه‌ی (۴۵.۱) به دست می‌آید.

$$A(n) = \frac{۱}{n-۱}\sum_{i=۱}^{n-۱} C(i) \tag{۴۵.۱}$$

$C(i)$ در رابطه‌ی (۴۵.۱) برابر است با:

$$C(i) = \sum_{j=۱}^{i}(n-j) = \sum_{j=۱}^{i} n - \sum_{j=۱}^{i} j = ni - \frac{i(i+۱)}{۲} = ni - \frac{i^۲}{۲} - \frac{i}{۲}$$

با جایگذاری $C(i)$ در (۴۵.۱) به رابطه‌ی (۴۶.۱) می‌رسیم.

$$A(n) = \frac{۱}{n-۱}\sum_{i=۱}^{n-۱}\left(ni - \frac{i^۲}{۲} - \frac{i}{۲}\right) \tag{۴۶.۱}$$

می‌توان رابطه‌ی (۴۶.۱) را به صورتی که در ادامه آمده است ساده کرد.

$$\begin{aligned}
A(n) &= \frac{۱}{n-۱}\left(\sum_{i=۱}^{n-۱} ni - \sum_{i=۱}^{n-۱}\frac{i^۲}{۲} - \sum_{i=۱}^{n-۱}\frac{i}{۲}\right)\\
&= \frac{۱}{n-۱}\left(\frac{n^۳-n^۲}{۲} - \frac{n(n-۱)(۲n-۱)}{۱۲} - \frac{n(n-۱)}{۴}\right)\\
&= \frac{۱}{n-۱}\cdot\frac{۲n^۳-۳n^۲+n}{۶}\\
&= \frac{۱}{n-۱}\cdot\frac{(۲n^۲-n)(n-۱)}{۶}\\
&= \frac{(۲n^۲-n)(n-۱)}{۶(n-۱)}\\
&= \frac{۲n^۲-n}{۶}\\
&= \frac{۱}{۳}n^۲ - \frac{۱}{۶}n\\
&= O(n^۲)
\end{aligned}$$

به این ترتیب می‌توان گفت الگوریتم مرتب‌سازی حبابی در حالت متوسط، همچون بدترین حالت، از مرتبه‌ی $O(n^۲)$ است.

◄ **سوال ۲۹.** دنباله‌ای از n عمل بر روی داده‌ساختار D را در نظر بگیرید. فرض کنید این اعمال را به ترتیب از ۱ تا n شماره‌گذاری کرده‌ایم. اگر عمل iام توانی از دو باشد آنگاه هزینه‌ی این عمل برابر با i و در غیر اینصورت هزینه‌ی آن برابر با یک خواهد بود. برای مثال اگر دارای دنباله‌ای با چهار عمل باشیم آنگاه هزینه‌ی عمل اول برابر با یک، هزینه‌ی عمل دوم برابر با دو، هزینه‌ی عمل سوم برابر با یک و هزینه‌ی عمل چهارم برابر با چهار خواهد بود و مجموع هزینه‌ی این اعمال برابر با هشت است. با توجه به توضیحات ارائه شده، میانگین هزینه‌ی یک عمل در دنباله‌ای با n عمل از چه مرتبه‌ای است؟

◁ **پاسخ سوال ۲۹.**

در یک دنباله با n عمل، تعداد اعمالی که شماره‌ی آنها توانی از ۲ است برابر با $\lfloor \lg n \rfloor + ۱$ است. اگر این تعداد را از تعداد کل اعمال کم کنیم به تعداد اعمالی می‌رسیم که دارای هزینه‌ی یک هستند. پس تعداد اعمال با هزینه‌ی یک برابر است با $n - (\lfloor \lg n \rfloor + ۱)$. بدین ترتیب می‌توان مجموع هزینه‌ی n عمل را به صورت زیر به دست آورد:

$$\begin{aligned} C(n) &= n - \big(\lfloor \lg n \rfloor + ۱\big) + \sum_{j=۰}^{\lfloor \lg n \rfloor} ۲^i \\ &\le n - \lfloor \lg n \rfloor - ۱ + \sum_{j=۰}^{\lg n} ۲^j \\ &= n - \lfloor \lg n \rfloor - ۱ + ۲^{\lg n + ۱} - ۱ \\ &= ۲^{\lg n + ۱} + n - \lfloor \lg n \rfloor - ۲ \end{aligned}$$

با توجه به مقدار به دست آمده برای $C(n)$ می‌توان گفت $C(n) = O(۲^{\lg n})$. از طرفی چون می‌دانیم $۲^{\lg n} = n$ در نتیجه داریم $C(n) = O(n)$ و این یعنی مجموع هزینه‌ی انجام n عمل از مرتبه‌ی $O(n)$ است. به این ترتیب می‌توان گفت میانگین هزینه‌ی انجام یک عمل برابر با $O(n)/n$ یا همان $O(۱)$ است.

فصل ۲

آرایه و ماتریس

۱.۲ مقدمه

آرایه یکی از ساده‌ترین و در عین حال مهمترین داده‌ساختارها است. معمولا از این داده‌ساختار به عنوان زیربنای بسیاری از داده‌ساختارهای دیگر استفاده می‌شود. برای مثال یکی از روش‌های پیاده‌سازی داده‌ساختارهای صف و پشته استفاده از داده‌ساختار آرایه است.

در این فصل سوالاتی در مورد آرایه و و انواع ماتریس (که در واقع یک آرایه‌ی دو بعدی است) مطرح خواهد شد.

۲.۲ منابع مطالعاتی

یکی از منابع مناسب برای مطالعه‌ی بیشتر در مورد آرایه و ماتریس فصل سوم [۲] است. در فصل ذکر شده در مورد آرایه، ماتریس پایین مثلثی، ماتریس قطری و ماتریس اسپارس صحبت شده است. از دیگر منابع مناسب برای کسب اطلاعات بیشتر در مورد آرایه و ماتریس می‌توان به فصل دوم [۳] اشاره کرد.

۳.۲ آرایه

◄ **سوال ۱.** آرایه‌ی دوبعدی A دارای نقطه‌ی زین[۱] است اگر بتوان عنصری مانند $A[i,j]$ یافت به طوری که این عنصر دارای مقدار کمینه در سطر i و مقدار بیشینه در ستون j باشد. الگوریتمی ارائه دهید که نقطه‌ی زین را در یک آرایه دو بعدی، در صورت وجود، بیابد. سپس مرتبه‌ی زمانی الگوریتم را به دست آورید.

◃ **پاسخ سوال ۱.**

[۱] Saddle point

شبه کد یکی از روش‌های تعیین نقطه‌ی زین در یک آرایه‌ی دو بعدی در الگوریتم (۱.۲) آورده شده است. اگر آرایه‌ی A دارای نقطه‌ی زین باشد آنگاه شماره‌ی سطر و ستون نقطه‌ی زین به عنوان خروجی برگشت داده می‌شود. در غیر اینصورت زوج مرتب $(-۱,-۱)$ به معنی عدم وجود چنین نقطه‌ای بازگردانده خواهد شد.

الگوریتم ۱.۲ یافتن نقطه‌ی زین در یک آرایه‌ی دو بعدی

```
function SaddlePoint(A)
    r = A.rows
    c = A.columns
    for i = 1 to r
        for j = 1 to c
            if MinInRow(A, i, j) == TRUE
                if MaxInColumn(A, i, j) == TRUE
                    return (i, j)
                end if
            end if
        end for
    end for
    return (−1, −1)
end function
```

در الگوریتم (۱.۲) از دو زیربرنامه‌ی MinInRow و MaxInCol به ترتیب برای تعیین کمینه بودن مقدار عنصر $A[i,j]$ در سطر i و تعیین بیشینه بودن مقدار $A[i,j]$ در ستون j استفاده شده است. شبه کد این دو زیربرنامه به ترتیب در قالب الگوریتم‌های (۲.۲) و (۳.۲) نشان داده شده است.

الگوریتم ۲.۲ تعیین کمینه بودن عنصری خاص در یک سطر خاص

```
function MinInRow(A, i, j)
    c = A.columns
    for k = 1 to n
        if A[i, k] < A[i, j]
            return FALSE
        end if
    end for
    return TRUE
end function
```

همانطور که مشخص است تعداد تکرار حلقه‌ی موجود در زیربرنامه‌ی MinInRow برابر با تعداد ستون‌های آرایه‌ی A است و اگر تعداد ستون‌های این آرایه را با c نشان دهیم پس این زیربرنامه از مرتبه $O(c)$ است. از

الگوریتم ۳.۲ تعیین بیشینه بودن عنصری خاص در یک ستون خاص

```
function MaxInCol(A, i, j)
    r = A.rows
    for k = 1 to r
        if A[k, j] > A[i, j]
            return FALSE
        end if
    end for
    return TRUE
end function
```

طرف دیگر تعداد تکرار حلقه‌ی موجود در زیربرنامه‌ی MaxInCol برابر با تعداد سطرهای آرایه‌ی A است و اگر تعداد سطرهای این آرایه را با r نشان دهیم آنگاه مرتبه‌ی زمانی این زیربرنامه برابر با $O(r)$ است.

با توجه به اینکه مرتبه‌ی زمانی هر دو زیربرنامه‌ی مورد استفاده در SaddlePoint را در اختیار داریم، می‌توان مرتبه‌ی زمانی الگوریتم (۱.۲) را به راحتی به دست آورد. در این الگوریتم دو حلقه‌ی تو در تو وجود دارد که در مجموع $r \times c$ بار اجرا می‌شوند و مرتبه‌ی زمانی هر بار اجرای حلقه‌ی داخلی برابر با $O(r+c)$ است. بدین ترتیب الگوریتم SaddlePoint از مرتبه‌ی $O(r^۲c + rc^۲)$ است.

◄ **سوال ۲.** داده‌ساختار D را در نظر بگیرید. این داده‌ساختار یک آرایه‌ی دو بعدی است به طوری که در هر سطر، هر مقدار از مقدار سمت راست خود بزرگتر و از مقدار سمت چپ خود کوچکتر است و همچنین در هر ستون هر مقدار از مقدار پایینی خود بزرگتر و از مقدار بالایی خود کوچکتر است. در خانه‌های خالی داده‌ساختار D مقدار $-\infty$ قرار می‌گیرد. مثالی از چنین داده‌ساختاری در ادامه نشان داده شده است. الگوریتمی برای حذف مقدار بیشینه و همچنین الگوریتمی برای درج مقداری جدید در داده‌ساختار D ارائه دهید.

$$\begin{bmatrix} ۲۲ & ۱۹ & ۱۸ & ۱۵ & -\infty \\ ۲۰ & ۱۸ & ۱۶ & ۱۴ & -\infty \\ ۱۳ & ۱۲ & ۹ & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix}$$

◃ **پاسخ سوال ۲.**

برای درج یک مقدار جدید در داده‌ساختار D ابتدا آن را در سطر و ستون آخر داده‌ساختار، یعنی در مکان $D[D.rows, D.columns]$ قرار می‌دهیم و سپس با استفاده از زیربرنامه‌ی Percolate سعی می‌کنیم مقدار تازه درج شده را به مکان مناسب انتقال دهیم تا همچنان ویژگی مرتب بودن داده‌ساختار حفظ شود. روش کار بدین صورت است که همواره مقدار تازه درج شده، با این فرض که در حال حاضر در مکان $D[i,j]$ قرار دارد، را با دو مقدار $D[i,j-۱]$ و $D[i-۱,j]$ مقایسه کرده و با کوچکترین آن دو جابجا می‌کنیم. این روند را به همین ترتیب ادامه می‌دهیم تا به حالتی برسیم که مقدار تازه درج شده از هر دو مقدار $D[i,j-۱]$

و $D[i-۱,j]$ کوچکتر باشد. در چنین حالتی مقدار تازه درج شده در مکان درست خود قرار دارد و اجرای زیربرنامه‌ی PERCOLATE خاتمه می‌یابد.

الگوریتم (۴.۲) شبه کد عمل درج را نشان می‌دهد. در این الگوریتم D داده‌ساختاری است که قرار است مقدار v در آن درج شود.

الگوریتم ۴.۲ درج یک مقدار جدید در داده‌ساختار D

```
procedure INSERT(D, v)
    D[D.rows, D.columns] = v
    PERCOLATE(D, D.rows, D.columns)
end procedure
```

شبه کد زیربرنامه‌ی PERCOLATE در قالب الگوریتم (۵.۲) آورده شده است.

الگوریتم ۵.۲ جابجا کردن مقدار تازه درج شده به سمت چپ یا بالا

```
procedure PERCOLATE(D, i, j)
    min = MINIMUM(D[i, j], D[i, j − 1], D[i − 1, j])
    if min == D[i, j]
        return
    else if min == D[i, j − 1]
        SWAP(D[i, j], D[i, j − 1])
        PERCOLATE(D, i, j − 1)
    else
        SWAP(D[i, j], D[i − 1, j])
        PERCOLATE(D, i − 1, j)
    end if
end procedure
```

با توجه به مرتب بودن سطرها و ستون‌های داده‌ساختار D به صورت نزولی، می‌توان نتیجه گرفت که مقدار بیشینه در مکان $D[۱,۱]$ قرار دارد. برای حذف مقدار بیشینه ابتدا مقدار خانه‌ی $D[۱,۱]$ را در یک متغیر ذخیره می‌کنیم و سپس در مکان $D[۱,۱]$ مقدار $-\infty$ را قرار می‌دهیم. سپس با استفاده از زیربرنامه‌ی SIFTDOWN مقدار موجود در $D[۱,۱]$ را تا زمانی به سمت پایین یا راست جابجا می‌کنیم که دوباره خاصیت مرتب بودن داده‌ساختار برقرار شود. شبه کد زیربرنامه‌ی حذف مقدار بیشینه در الگوریتم (۶.۲) آمده است.

الگوریتم ۶.۲ حذف مقدار بیشینه از داده‌ساختار D

```
procedure DELETEMAX(D)
    max = D[1, 1]
    D[1, 1] = −∞
```

حذف مقدار بیشینه از داده‌ساختار D ــ ادامه

```
    SiftDown(D, 1, 1)
    return max
end procedure
```

شبه کد زیربرنامه‌ی SiftDown در الگوریتم (۷.۲) نشان داده شده است. روش کلی کار این زیربرنامه مانند زیربرنامه‌ی Percolate است با این تفاوت که مقدار $D[i,j]$ با مقادیر $D[i,j+1]$ و $D[i+1,j]$ مقایسه شده و به جای جابجایی به سمت بالا یا چپ، به سمت پایین یا راست جابجا می‌شود.

الگوریتم ۷.۲ جابجا کردن مقدار $-\infty$ به سمت راست یا پایین

```
procedure SiftDown(D, i, j)
    max = Maximum(D[i, j], D[i, j + 1], D[i + 1, j])
    if max == D[i, j]
        return
    else if max == D[i, j + 1]
        Swap(D[i, j], D[i, j + 1])
        SiftDown(D, i, j + 1 )
    else
        Swap(D[i, j], D[i + 1, j])
        SiftDown(D, i + 1, j)
    end if
end procedure
```

توجه داشته باشید که در شبه کد زیربرنامه‌های Percolate و SiftDown به دلیل جلوگیری از شلوغ شدن شبه کد، از قرار دادن دستورات شرطی مربوط به چک کردن محدوده‌ی اندیس‌های داده‌ساختار D خودداری شده است.

اگر تعداد سطرها و ستون‌های داده‌ساختار D را به ترتیب با r و c نشان دهیم آنگاه می‌توان گفت هر دو عمل درج و حذف از مرتبه‌ی $O(r+c)$ هستند. اثبات این موضوع به خواننده واگذار می‌شود (راهنمایی: قرار دهید $p = r + c$ و $T(p)$ را به عنوان رابطه‌ی بازگشتی نشان دهنده‌ی زمان مصرفی عمل درج یا حذف در نظر بگیرید. سپس نشان دهید $T(p)$ از مرتبه‌ی $O(p)$ است).

◂ **سوال ۳.** آرایه‌ی یک بعدی A یک آرایه‌ی تک‌قُله‌ای[۲] است اگر مقداری مانند $A[m]$ وجود داشته باشد به طوری که به ازای این مقدار هر دو شرط زیر برقرار باشند:

- برای هر i، $1 \leqslant i < m$، داشته باشیم $A[i] < A[i+1]$.
- برای هر j، $m \leqslant j \leqslant A.length$، داشته باشیم $A[j] > A[j+1]$.

[۲] Single peak

۱	۲	۳	۴	۵	۶	۷	۸
−۱	۳	۴	۹	۱۴	۱۶	۱۳	۵

شکل (۱.۲): یک آرایه‌ی تک‌قله‌ای با قله‌ی $A[۶]$

در این صورت می‌توان گفت خانه‌ی $A[m]$ قله یا همان بزرگترین مقدار آرایه‌ی A است. در شکل (۱.۲) یک آرایه‌ی تک‌قله‌ای نشان داده شده است و $A[۶]$ با مقدار ۱۶ قله‌ی این آرایه است. الگوریتمی بنویسید که یک آرایه‌ی تک‌قله‌ای را به عنوان ورودی دریافت کرده و قله‌ی آن را در زمان با مرتبه لگاریتمی نسبت به طول آرایه به دست آورد. (راهنمایی: از ایده‌ی موجود در الگوریتم جستجوی دودویی استفاده کنید).

◁ **پاسخ سوال ۳.**

با در نظر گرفتن تعریف آرایه‌ی تک‌قله‌ای، به ازای هر اندیس i، $۱ \leqslant i \leqslant A.length$، یکی از دو رابطه‌ی $A[i] < A[i+۱]$ یا $A[i] > A[i+۱]$ برقرار است. با توجه به این دو حالت می‌توان گفت:

- اگر $A[i] < A[i+۱]$ آنگاه باید در محدوده‌ی $A[i+۱\,..\,A.length]$ به دنبال قله بگردیم.
- اگر $A[i] > A[i+۱]$ آنگاه باید در محدوده‌ی $A[۱\,..\,i]$ به دنبال قله بگردیم.

با توجه به توضیحات داده شده می‌توان از الگوریتم (۸.۲) برای پیدا کردن قله‌ی یک آرایه‌ی تک‌قله‌ای استفاده کرد.

الگوریتم ۸.۲ یافتن قله در یک آرایه‌ی تک‌قله‌ای

```
function FindPeak(A)
    start = 1
    end = A.length
    while start < end
        mid = ⌊(start + end)/2⌋
        if A[mid] < A[mid + 1]
            start = mid + 1
        end if
        if A[mid] > A[mid + 1]
            end = mid
        end if
    end while
    return A[start]
end function
```

همانطور که مشاهده می‌شود روش کار این الگوریتم شباهت زیادی به الگوریتم جستجوی دودویی دارد و با انجام تحلیلی مشابه تحلیل الگوریتم جستجوی دودویی می‌توان به این نتیجه رسید که الگوریتم FindPeak، همچون الگوریتم جستجوی دودویی، از مرتبه‌ی $O(\lg n)$ است که n بیانگر طول آرایه ورودی است.

۱	۲	۳	۴	۵	۶
۹	۷	۲	۴	۸	۳

شکل (۲.۲): یک آرایه‌ی K-Flat

◄ **سوال ۴.** آرایه‌ی A که دارای زوج عنصر و حاوی اعداد غیر تکراری است و همچنین عدد k را در اختیار داریم. می‌گوییم آرایه‌ی A از نوع K-Flat است اگر به ازای هر $A[i]$، $۱ \leqslant i \leqslant A.length$، عنصری مانند $A[j]$، $۱ \leqslant j \leqslant A.length$ ، با شرط $i \neq j$ یافت شود به طوری که $A[i] + A[j] = k$. برای مثال آرایه‌ی نشان داده شده در شکل (۲.۲) را در نظر بگیرید. این آرایه به ازای $k = ۱۱$ یک آرایه‌ی K-Flat است زیرا داریم:

$$A[۱] + A[۳] = A[۲] + A[۴] = A[۵] + A[۶] = ۱۱$$

با در نظر گرفتن توضیحات داده شده الگوریتمی بنویسید که یک آرایه شامل اعداد غیر تکراری و عدد k را دریافت کرده و مشخص کند آیا آرایه‌ی ورودی K-Flat است یا خیر.

◃ **پاسخ سوال ۴.**

الگوریتم تعیین K-Flat بودن یک آرایه در قالب الگوریتم (۹.۲) آمده است.

الگوریتم ۹.۲ تعیین K-Flat بودن آرایه

```
function KFlat(A, k)
    Sort(A)
    i = 1
    j = A.length
    while i < j
        if A[i] + A[j] ≠ k
            return FALSE
        else
            i = i + 1
            j = j − 1
        end if
    end while
    return TRUE
end function
```

در این الگوریتم در ابتدا و به کمک الگوریتم Sort آرایه‌ی ورودی را به صورت صعودی و در زمان $O(n \lg n)$ مرتب می‌کنیم که در آن n طول آرایه است. سپس با استفاده از متغیر i از ابتدا به سمت انتهای آرایه و با استفاده از متغیر j از انتها به سمت ابتدای آرایه حرکت می‌کنیم. با مقایسه‌ی دو عنصر $A[i]$ و $A[j]$ ممکن است دو حالت روی دهد که عبارت‌اند از:

۱. $A[i] + A[j] = k$: در این حالت به ازای عنصر $A[i]$، عنصر $A[j]$ یافته شده است (و بالعکس) و این یعنی تا بدین جا شرط K-Flat بودن آرایه نقض نشده است و اجرای الگوریتم برای بررسی سایر خانه‌ها باید ادامه پیدا کند. برای ادامه‌ی اجرای الگوریتم به متغیر i یک واحد اضافه می‌شود تا به خانه‌ی بعدی و از متغیر j یک واحد کم می‌شود تا به خانه‌ی قبلی اشاره کند.

۲. $A[i] + A[j] \neq k$: این حالت، خود به دو حالت دیگر تقسیم می‌شود:

(آ) $A[i] + A[j] > k$: در این حالت به ازای $A[i]$ نمی‌توان عنصری یافت به طوری که شرط K-Flat بودن برقرار باشد.

(ب) $A[i] + A[j] < k$: در این حالت به ازای $A[j]$ نمی‌توان عنصری یافت به طوری که شرط K-Flat بودن برقرار باشد.

به این ترتیب در این حالت یا به ازای $A[i]$ و یا به ازای $A[j]$ شرط K-Flat بودن آرایه نقض می‌شود و در نتیجه مقدار FALSE به معنی K-Flat نبودن آرایه برگشت داده می‌شود و اجرای الگوریتم خاتمه می‌یابد.

اگر اجرای الگوریتم به خط ۱۳ برسد به این معنی است که شرط K-Flat بودن برای تمام عناصر آرایه برقرار بوده است و در نتیجه مقدار TRUE به نشانه‌ی K-Flat بودن آرایه‌ی ورودی برگشت داده می‌شود.

◄ **سوال ۵.** فرض کنید A آرایه‌ای n عنصری و شامل اعداد صحیح است. الگوریتمی با مرتبه‌ی زمانی $O(n \lg n)$ بنویسید که آرایه‌ی A و عدد صحیح k را دریافت و مشخص کند آیا دو عنصر متمایز مانند $A[i]$ و $A[j]$ می‌توان یافت به طوری که $A[i] + A[j] = k$.

◃ **پاسخ سوال ۵.**

شبه کد الگوریتم خواسته شد در قالب الگوریتم (۱۰.۲) آورده شده است. روش کار این الگوریتم در ادامه توضیح داده می‌شود.

در الگوریتم CheckPair در مرحله‌ی اول به کمک الگوریتم Sort آرایه‌ی A را به صورت صعودی و در زمان $O(n \lg n)$ مرتب می‌کنیم. سپس با استفاده از متغیر i از ابتدا به سمت انتها و با استفاده از متغیر j از انتها به سمت ابتدای آرایه‌ی A حرکت می‌کنیم. با مقایسه‌ی دو عنصر $A[i]$ و $A[j]$ ممکن است یکی از سه حالت زیر روی دهد:

۱. $A[i] + A[j] = k$: در این حالت دو عنصر $A[i]$ و $A[j]$ پیدا شده‌اند به طوری که حاصل جمع آنها برابر با k است. در چنین حالتی مقدار TRUE به عنوان خروجی الگوریتم و به نشانه‌ی اجرای موفقیت آمیز الگوریتم برگشت داده می‌شود.

۲. $A[i] + A[j] < k$: در این حالت چون $A[i] + A[j]$ کمتر از k است، با توجه به مرتب بودن آرایه‌ی A به صورت صعودی، یک واحد به مقدار i اضافه می‌شود تا در دور بعدی اجرای حلقه مقدار $A[i] + A[j]$ به k نزدیکتر و در حالت ایده‌آل برابر با k شود.

۳. $A[i] + A[j] > k$: در این حالت چون $A[i] + A[j]$ بزرگتر از k است، با توجه به مرتب بودن آرایه‌ی A به صورت صعودی، یک واحد از مقدار j کم می‌شود تا در دور بعدی اجرای حلقه مقدار

الگوریتم ۱۰.۲ یافتن دو عنصر با مجموع مشخص در یک آرایه یک بعدی

```
1: function CheckPair(A, k)
2:     Sort(A)
3:     i = 1
4:     j = A.length
5:     while i < j
6:         if A[i] + A[j] == k
7:             return TRUE
8:         else if A[i] + A[j] < k
9:             i = i + 1
10:        else
11:            j = j − 1
12:        end if
13:    end while
14:    return FALSE
15: end function
```

$A[i] + A[j]$ به k نزدیکتر و در حالت ایده‌آل برابر با k شود.

اگر اجرای الگوریتم به خط ۱۴ برسد مقدار FALSE به عنوان خروجی الگوریتم برگشت داده می‌شود تا نشان داده شود که هیچ دو عنصر متمایزی در آرایه‌ی A وجود ندارند به طوری که حاصل جمع آنها برابر با k شود.

◄ **سوال ۶.** فرض کنید A آرایه‌ای n عنصری ($n \geqslant ۲$) و شامل اعداد صحیح در بازه‌ی $[۱, n-۱]$ است. تمام اعداد موجود در آرایه‌ی A متمایز هستند به جز یک عدد که دو بار در آرایه ظاهر شده است. الگوریتمی کارا بنویسید که چنین آرایه‌ای را دریافت کرده و عدد تکراری را بازگرداند. مرتبه‌ی زمانی الگوریتم خود را به دست آورید.

◃ **پاسخ سوال ۶.**

شبه کد الگوریتم یافتن عدد تکرار شده در یک آرایه در الگوریتم (۱۱.۲) آورده شده است.

روش کار این الگوریتم بسیار ساده است. در ابتدای کار آرایه‌ای از نوع منطقی به نام B و به طول $n-۱$ تعریف شده است و مقدار اولیه‌ی تمام خانه‌های آن برابر با FALSE قرار داده شده است. سپس با استفاده از یک حلقه تمام خانه‌های آرایه‌ی A بررسی می‌شوند. در یک تکرار از حلقه یکی از دو حالت زیر روی می‌دهد:

- مقدار خانه‌ی $B[A[i]]$ برابر با FALSE باشد: این حالت بدین معنی است که عدد موجود در خانه‌ی $A[i]$، تا تکرار جاری در آرایه‌ی A مشاهده نشده است. در این حالت مقدار خانه‌ی $B[A[i]]$ برابر با TRUE قرار داده می‌شود تا نشان داده شود عدد موجود در خانه‌ی $A[i]$ تا تکرار جاری یک بار مشاهده شده است.
- مقدار خانه‌ی $B[A[i]]$ برابر با TRUE باشد: این حالت زمانی اتفاق می‌افتد که عدد موجود در خانه‌ی

الگوریتم ۱۱.۲ یافتن عدد تکرار شده در یک آرایه یک بعدی

```
function FindRepeated(A)
    n = A.length
    Let B[1..n − 1] be a new array of type boolean
    for i = 1 to n − 1
        B[i] = FALSE
    end for
    for i = 1 to n
        if B[A[i]] == FALSE
            B[A[i]] = TRUE
        else
            return A[i]
        end if
    end for
end function
```

$A[i]$ در یکی از تکرارهای قبلی حلقه مشاهده شده باشد و این یعنی عدد موجود در خانه‌ی $A[i]$ همان عدد تکرار شده در آرایه‌ی A است. در این حالت عدد موجود در خانه‌ی $A[i]$ به عنوان جواب الگوریتم برگشت داده می‌شود.

الگوریتم FindRepeated در بدترین حالت از مرتبه‌ی $\Theta(n)$ است زیرا حلقه‌ی اول همواره $n - ۱$ بار و حلقه‌ی دوم حداکثر n بار اجرا می‌شود.

◄ **سوال ۷.** تابعی بازگشتی بنویسید که بزرگترین مقدار در یک آرایه را بیابد.

◁ **پاسخ سوال ۷.**

الگوریتم (۱۲.۲) نشان دهنده‌ی شبه کد تابع بازگشتی یافتن بزرگترین مقدار در یک آرایه است. برای یافتن بزرگترین مقدار در آرایه‌ی A تابع FindMax باید به صورت $\text{FindMax}(A, 1, A.length)$ فراخوانی شود. در ادامه روش کار این تابع بیان می‌شود.

الگوریتم ۱۲.۲ یافتن بزرگترین مقدار در یک آرایه یک بعدی به صورت بازگشتی

```
function FindMax(A, low, high)
    if low == high
        max = A[low]
    else
        max = FindMax(A, low + 1, high)
        if A[low] > max
```

یافتن بزرگترین مقدار در یک آرایه یک بعدی به صورت بازگشتی ــ ادامه

```
        max = A[low]
    end if
end if
return max
end function
```

اگر دو متغیر low و $high$ برابر با یکدیگر باشند به این معنی است که زیرآرایه‌ی $A[low\,..\,high]$ دارای تنها یک خانه است و بدیهی است که مقدار همین تک خانه باید به عنوان بزرگترین مقدار زیرآرایه‌ی $A[low\,..\,high]$ بازگردانده شود. در صورتی که زیرآرایه‌ی $A[low\,..\,high]$ دارای بیش از یک خانه باشد آنگاه ابتدا به صورت بازگشتی بزرگترین مقدار زیرآرایه‌ی $A[low+۱\,..\,high]$ را به دست آورده و در متغیر max قرار می‌دهیم. سپس مقدار متغیر max را با خانه‌ی $A[low]$ مقایسه می‌کنیم. اگر شرط $A[low] > max$ برقرار باشد یعنی مقدار خانه‌ی $A[low]$ همان بزرگترین مقدار زیرآرایه‌ی $A[low\,..\,high]$ است و مقدار متغیر max بروزرسانی می‌شود در غیر این صورت مقدار متغیر max بزرگترین مقدار آرایه است.

◄ **سوال ۸.** تابعی بازگشتی بنویسید که آرایه‌ی دو بعدی A را به عنوان ورودی دریافت کرده و مجموع مقادیر آن را به عنوان خروجی بازگرداند.

◁ **پاسخ سوال ۸.**

شبه کد تابع بازگشتی جمع مقادیر یک آرایه‌ی دو بعدی در قالب الگوریتم (۱۳.۲) نمایش داده شده است. برای جمع مقادیر آرایه‌ی دو بعدی A تابع بازگشتی SumArray باید به صورت $\textsc{SumArray}(A, 1, 1)$ فراخوانی شود.

الگوریتم ۱۳.۲ جمع مقادیر یک آرایه‌ی دو بعدی به شکل بازگشتی

```
function SumArray(A, i, j)
    r = A.rows
    c = A.columns
    if i==r and j==c
        return A[i, j]
    else if i==r
        sum = 0
        for x = j to c
            sum = sum + A[i, x]
        end for
    else if j==c
        sum = 0
        for x = i to r
```

جمع مقادیر یک آرایه‌ی دو بعدی به شکل بازگشتی ــ ادامه

```
14:             sum = sum + A[x, j]
15:         end for
16:     else
17:         sum = 0
18:         for x = i to r
19:             sum = sum + A[x, j]
20:         end for
21:         for x = j to c
22:             sum = sum + A[i, x]
23:         end for
24:         return sum + SumArray(A, i + 1, j + 1)
25:     end if
26: end function
```

در الگوریتم (۱۳.۲) سه حالت زیر به عنوان حالات پایه‌ی تابع بازگشتی در نظر گرفته شده‌اند:

- آرایه‌ی A دارای تنها یک خانه باشد. در این حالت مقدار همان تک خانه به عنوان جمع مقادیر آرایه بازگردانده می‌شود.
- آرایه‌ی A دارای تنها یک سطر باشد. در این حالت با استفاده از یک حلقه، مقادیر آن تک سطر با یکدیگر جمع شده و به عنوان جمع مقادیر آرایه بازگردانده می‌شود (خطوط ۷ تا ۱۰).
- آرایه‌ی A دارای تنها یک ستون باشد. در چنین حالتی به کمک یک حلقه، مقادیر همان تک ستون با یکدیگر جمع شده و به عنوان جمع مقادیر آرایه برگردانده می‌شود (خطوط ۱۲ تا ۱۵).

اگر هیچ یک از حالات پایه‌ی تابع بازگشتی برقرار نباشند آنگاه می‌توان به صورت بازگشتی جمع مقادیر آرایه‌ی A را به دست آورد. برای این منظور ابتدا مقادیر سطر و ستون اول آرایه‌ی A را با یکدیگر جمع کرده و حاصل را در متغیر sum قرار می‌دهیم. با انجام این کار جمع مقادیر سطر و ستون اول ماتریس A را داریم و کافیست به صورت بازگشتی جمع مقادیر زیرآرایه‌ی $A[۲\,..\,A.rows, ۲\,..\,A.columns]$ را به دست آورده و مجموع به دست آمده را به متغیر sum اضافه کنیم تا جمع مقادیر آرایه‌ی A به دست آید.

◄ **سوال ۹.** آرایه‌ی یک بعدی A که حاوی اعداد صحیح است را در نظر بگیرید. زیرآرایه‌ی $A[i\,..\,j]$ یک زیرآرایه‌ی بیشینه[۳] است اگر مجموع مقادیر این زیرآرایه از مجموع مقادیر هر زیرآرایه‌ی دیگر A بزرگتر باشد. برای مثال در شکل (۳.۲) زیرآرایه‌ی $A[۸\,..\,۱۱]$ با مجموع ۴۳ یک زیرآرایه‌ی بیشینه است. الگوریتمی از مرتبه‌ی خطی بنویسید که زیرآرایه‌ی بیشینه یک آرایه یک بعدی را به دست آورد.

راهنمایی: از ابتدای آرایه به سمت انتهای آن حرکت کنید و در حین حرکت زیرآرایه‌ی بیشینه تا خانه‌ی جاری را ثبت کنید. اگر فرض کنیم زیرآرایه بیشینه‌ی بازه‌ی $A[۱\,..\,j]$ را می‌دانید آنگاه باید زیرآرایه بیشینه‌ی بازه‌ی

[۳] Maximum subarray

۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱	۱۲
۱۳	−۳	−۲۵	۲۰	−۳	−۱۶	−۲۳	۱۸	۲۰	−۷	۱۲	−۵

شکل (۳.۲): آرایه‌ای با زیرآرایه‌ی بیشینه‌ی $A[۸\,..\,۱۱]$

$A[۱\,..\,j+۱]$ را به دست آورید. برای این منظور به این نکته توجه کنید که زیرآرایه‌ی بیشینه‌ی بازه‌ی $A[۱\,..\,j+۱]$ در یکی از دو حالت زیر صدق می‌کند:

۱. زیرآرایه‌ی بیشینه‌ی بازه‌ی $A[۱\,..\,j+۱]$ همان زیرآرایه بیشینه‌ی بازه‌ی $A[۱\,..\,j]$ است،

۲. زیرآرایه‌ی بیشینه‌ی بازه‌ی $A[۱\,..\,j+۱]$ به صورت $A[i\,..\,j+۱]$ است به طوری که $۱ \le i \le j+۱$.

بر این اساس که زیرآرایه‌ی بیشینه به شکل $A[i\,..\,j]$ را در اختیار دارید، زیرآرایه‌ی بیشینه به شکل $A[i\,..\,j+۱]$ را در زمان ثابت به دست آورید تا مرتبه‌ی الگوریتم یافتن زیرآرایه بیشینه به صورت خطی به دست آید.

◁ **پاسخ سوال ۹.**

الگوریتم (۱۴.۲) نشان دهنده‌ی شبه کد الگوریتم یافتن زیرآرایه‌ی بیشینه در یک آرایه‌ی یک بعدی است. این الگوریتم با عنوان الگوریتم کادان[۴] شناخته می‌شود.

الگوریتم ۱۴.۲ یافتن زیرآرایه‌ی بیشینه در یک آرایه یک بعدی

```
function MaxSubarray(A)
    n = A.length
    maxSum = −∞
    endingHereSum = −∞
    for j = 1 to n
        endingHereHigh = j
        if endingHereSum + A[j] > A[j]
            endingHereSum = endingHereSum + A[j]
        else
            endingHereLow = j
            endingHereSum = A[j]
        end if
        if endingHereSum > maxSum
            maxSum = endingHereSum
            low = endingHereLow
            high = endingHereHigh
        end if
    end for
```

[۴] Kadane's algorithm

یافتن زیرآرایه‌ی بیشینه در یک آرایه یک بعدی ــ ادامه

```
    return (low, high, maxSum)
end function
```

در الگوریتم (۱۴.۲) تعدادی متغیر مهم وجود دارند که عبارت‌اند از:

- low و $high$ به ترتیب اندیس ابتدا و انتهای زیرآرایه‌ی بیشینه یافته شده تا تکرار جاری را نشان می‌دهند.
- $maxSum$ بیانگر مجموع مقادیر زیرآرایه بیشینه یافته شده تا تکرار جاری است.
- $endingHereLow$ و $endingHereHigh$ نشانگر اندیس ابتدا و انتهای زیرآرایه‌ی بیشینه‌ای هستند که به اندیس j ختم می‌شود. از آنجایی که اندیس انتهایی هر زیرآرایه بیشینه که به j ختم می‌شود باید j باشد در نتیجه در هر دور از تکرار حلقه قرار می‌دهیم $endingHereHigh = j$.
- $endingHereSum$: نشان دهنده‌ی مجموع مقادیر زیرآرایه بیشینه‌ای است که به اندیس j ختم می‌شود.

دستور شرطی ابتدای حلقه تعیین می‌کند آیا زیرآرایه بیشینه‌ای که به اندیس j ختم می‌شود فقط شامل $A[j]$ است یا خیر. با شروع هر تکرار حلقه متغیر $endingHereSum$ حاوی مجموع مقادیر زیرآرایه بیشینه‌ای است که به اندیس $j - ۱$ ختم می‌شود. اگر شرط $endingHereSum + A[j] > A[j]$ برقرار باشد آنگاه زیرآرایه بیشینه ختم شده به اندیس $j - ۱$ را گسترش می‌دهیم تا شامل اندیس j نیز بشود. در غیر این صورت زیرآرایه بیشینه جدیدی با اندیس ابتدایی j را شروع می‌کنیم که در این صورت متغیرهای low و $high$ هر دو برابر با j و $endingHereSum$ برابر با $A[j]$ قرار داده می‌شوند.

پس از به دست آوردن زیرآرایه بیشینه‌ای که به اندیس j ختم می‌شود با یک دستور شرطی دیگر بررسی می‌کنیم که آیا مجموع مقادیر این زیرآرایه بیشینه از مجموع مقادیر زیرآرایه‌ی بیشینه‌ای که تا تکرار جاری به دست آورده‌ایم بزرگتر است یا خیر. اگر اینگونه بود مقادیر متغیرهای low، $high$ و $maxSum$ را بروزرسانی می‌کنیم.

از آنجایی که هر تکرار از حلقه الگوریتم MaxSubarray از مرتبه ثابت است و حلقه در مجموع n بار تکرار می شود پس می‌توان گفت الگوریتم یافتن زیرآرایه بیشینه یک آرایه یک بعدی از مرتبه $\Theta(n)$ است که n بیانگر طول آرایه است.

◄ **سوال ۱۰.** در مواقعی ممکن است در یک آرایه به جای یک مقدار خاص، به دنبال مقداری با یک *ویژگی* خاص باشیم. برای مثال در آرایه‌ای شامل تعدادی عدد به دنبال kامین بزرگترین عدد باشیم. یک روش برای بدست آوردن kامین بزرگترین مقدار این است که آرایه را به صورت نزولی مرتب کنیم که در این صورت می‌توان گفت kامین بزرگترین مقدار در مکان kام آرایه قرار دارد. این روش در اکثر اوقات کاری بیش از آنچه که مورد نیاز است را انجام می‌دهد زیرا مقادیری که کوچکتر از kامین بزرگترین مقدار هستند اهمیتی ندارند. روش دیگر این است که بزرگترین مقدار موجود در آرایه را پیدا کرده و آن را در خانه‌ی اول آرایه قرار دهیم. سپس به دنبال دومین بزرگترین مقدار بگردیم و آن را در خانه‌ی دوم آرایه قرار دهیم و به همین ترتیب همین روند را ادامه دهیم. اگر این کار را k بار تکرار کنیم آنگاه kامین بزرگترین مقدار در خانه‌ی kام آرایه قرار خواهد گرفت. با در نظر گرفتن توضیحات داده شده، شبه کد الگوریتم روش دوم را نوشته و مرتبه‌ی زمانی آن را به دست آورید.

◁ **پاسخ سوال ۱۰.**

شبه کد الگوریتم یافتن kامین بزرگترین مقدار در یک آرایه در الگوریتم (۱۵.۲) آمده است.

الگوریتم ۱۵.۲ یافتن k امین بزرگترین عنصر در یک آرایه یک بعدی

```
function KthLargest(A, k)
    n = A.length
    for i = 1 to k
        maxIdx = i
        for j = i + 1 to n
            if A[j] > A[maxIdx]
                maxIdx = j
            end if
        end for
        Swap(A[i], A[maxIdx])
    end for
    return A[k]
end function
```

در دور اول از اجرای حلقه‌ی بیرونی، حلقه‌ی درونی $n-۱$ بار اجرا می‌شود. در دور دوم از اجرای حلقه‌ی بیرونی، حلقه‌ی درونی $n-۲$ بار اجرا می‌شود و به همین ترتیب تا اجرای kام حلقه‌ی بیرونی که به ازای آن حلقه‌ی درونی $n-k$ بار اجرا می‌شود. بدین ترتیب اگر مجموع تعداد تکرار دو حلقه الگوریتم KthLargest را با $S(n)$ نشان دهیم خواهیم داشت:

$$\begin{aligned} S(n) &= \sum_{i=۱}^{k} (n-i) \\ &= \sum_{i=۱}^{k} n - \sum_{i=۱}^{k} i \\ &= nk - \frac{k(k+۱)}{۲} \end{aligned}$$

به این ترتیب می‌توان گفت الگوریتم یافتن kامین بزرگترین مقدار یک آرایه از مرتبه‌ی $O(nk)$ است. برای حل این مسئله الگوریتمی با مرتبه‌ی زمانی بهتر نیز وجود دارد.

◀ **سوال ۱۱.** برای یافتن kامین بزرگترین مقدار در یک آرایه نیازی به تعیین مکان دقیق قرارگیری مقادیر کوچکتر یا بزرگتر از kامین بزرگترین مقدار نداریم. تنها کافی است بدانیم که این مقادیر، بزرگتر یا کوچکتر از kامین بزرگترین مقدار هستند. برای یافتن kامین بزرگترین مقدار در یک آرایه می‌توان از الگوریتمی که در ادامه بیان می‌شود استفاده کرد.

در ابتدا به صورت تصادفی عنصری از آرایه را انتخاب می‌کنیم. با توجه به عنصر انتخاب شده می‌توان عناصر

آرایه را به دو دسته افراز[۵] کرد:

۱. عناصری که بزرگتر از عنصر انتخاب شده هستند.

۲. عناصری که کوچکتر یا مساوی عنصر انتخاب شده هستند.

اگر عناصر بزرگتر از عنصر انتخابی را به خانه‌های قبل از این عنصر و عناصر کوچکتر یا مساوی عنصر انتخابی را به خانه‌های بعد از این عنصر انتقال دهیم در نهایت عنصر انتخابی در خانه‌ای مانند p در آرایه قرار خواهد گرفت. قرار گرفتن عنصر انتخابی در خانه‌ی p به این معنی است که این عنصر pامین بزرگترین مقدار آرایه است. اگر $p = k$ آنگاه kامین بزرگترین مقدار پیدا شده است و اجرای الگوریتم خاتمه می‌یابد. در صورتیکه k کوچکتر از p باشد آنگاه باید به صورت بازگشتی در زیرآرایه‌ی $A[۱\,..\,p-۱]$ به دنبال kامین بزرگترین مقدار بگردیم. در صورتیکه k بزرگتر از p باشد باید به صورت بازگشتی در زیرآرایه‌ی $A[p+۱\,..\,n]$ به دنبال kامین بزرگترین مقدار بگردیم و باید این نکته را در نظر بگیریم که باید به دنبال $(k-p)$امین بزرگترین مقدار باشیم زیرا مقادیر بزرگتر از pامین بزرگترین مقدار، که قبل از خانه‌ی p هستند را نادیده می‌گیریم. لذا در زیرآرایه‌ی سمت راست باید به دنبال $(k-p)$امین بزرگترین مقدار باشیم.

با در نظر گرفتن آنچه بیان شد، شبه کد الگوریتم شرح داده شده را بنویسید.

◁ **پاسخ سوال ۱۱.**

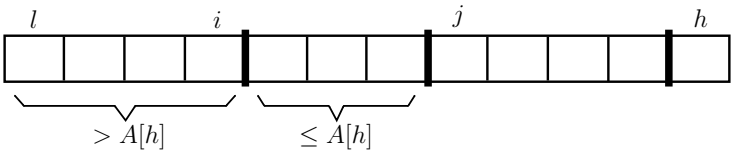
شبه کد الگوریتم یافتن kامین بزرگترین مقدار در یک آرایه در الگوریتم (۱۶.۲) نشان داده شده است. اگر قصد یافتن kامین بزرگترین مقدار در آرایه‌ی $A[۱\,..\,n]$ را داشته باشیم این الگوریتم باید به شکل $\textsc{KthLargest}(A, 1, n, k)$ فراخوانی شود.

الگوریتم ۱۶.۲ یافتن k امین بزرگترین عنصر در یک آرایه یک بعدی به صورت بازگشتی

```
function KthLargest(A, l, h, k)
    if l < h
        p = Partition(A, l, h)
        if p == k
            return A[p]
        else if k < p
            return KthLargest(A, l, p − 1, k)
        else
            return KthLargest(A, p + 1, h, k − p)
        end if
    end if
end function
```

زیربرنامه‌ی Partition مهمترین بخش الگوریتم یافتن kامین بزرگترین مقدار است. این زیربرنامه عنصر $A[h]$ در محدوده‌ی $A[l\,..\,h]$ را به عنوان عنصر محور[۶] انتخاب کرده و عناصر آرایه‌ی $A[l\,..\,h]$ را طوری

[۵] partition
[۶] pivot



شکل (۴.۲): زیرآرایه‌های ایجاد شده توسط زیربرنامه‌ی PARTITION

جابجا می‌کند که عناصر بزرگتر از عنصر محور به قبل از عنصر محور و عناصر کوچکتر یا مساوی عنصر محور به بعد از آن منتقل شوند. سپس مکان نهایی قرارگیری عنصر محور را به عنوان خروجی باز می‌گرداند. شبه کد زیربرنامه‌ی PARTITION در قالب الگوریتم (۱۷.۲) آورده شده است.

الگوریتم ۱۷.۲ افراز آرایه به دو بخش

```
function PARTITION(A, l, h)
    x = A[h]
    i = l − 1
    for j = l to h − 1
        if A[j] > x
            i = i + 1
            SWAP(A[i], A[j])
        end if
    end for
    SWAP(A[i + 1], A[r])
    return i + 1
end function
```

روش کار زیربرنامه‌ی PARTITION بسیار ساده است. در این زیربرنامه آرایه‌ی ورودی در هر لحظه به چهار زیرآرایه‌ی نشان داده شده در شکل (۴.۲) تقسیم می‌شود. این زیرآرایه‌ها عبارت‌اند از:

- زیرآرایه‌ی $A[l\,..\,i]$ حاوی مقادیر بزرگتر از عنصر محور،
- زیرآرایه‌ی $A[i+۱\,..\,j-۱]$ حاوی مقادیر کوچکتر یا مساوی عنصر محور،
- زیرآرایه‌ی $A[j\,..\,h-۱]$ حاوی مقادیری که هنوز بررسی نشده‌اند،
- زیرآرایه‌ی تک عنصری $A[h]$ که همان عنصر محور است.

در پایان اجرای زیربرنامه‌ی PARTITION اگر فرض کنیم عنصر محور در خانه‌ای با اندیس p قرار گرفته باشد آنگاه $A[p]$ کوچکتر از تمام عناصر زیرآرایه‌ی $A[l\,..\,p-۱]$ و بزرگتر یا مساوی عناصر $A[p+۱\,..\,h]$ است.

◄ **سوال ۱۲.** آرایه‌ی یک بعدی A که حاوی اعداد متمایز است را در نظر بگیرید. اگر $i < j$ و $A[i] > A[j]$ آنگاه زوج مرتب (i, j) یک وارونگی[۷] نامیده می‌شود. برای مثال آرایه‌ی نشان داده شده در شکل (۵.۲) دارای

[۷] inversion

۱	۲	۳	۴	۵	۶
۲	۳	۸	۶	۱	۱۵

شکل (۵.۲): آرایه‌ای با پنج وارونگی

پنج وارونگی است که عبارت‌اند از:

$$I = \{(۲, ۱), (۳, ۱)(۸, ۱), (۶, ۱), (۸, ۶)\}$$

حداکثر تعداد وارونگی‌ها در یک آرایه‌ی یک بعدی با n عنصری چه تعداد است؟

◁ **پاسخ سوال ۱۲.**

اگر آرایه به صورت نزولی مرتب باشد آنگاه دارای بیشترین تعداد وارونگی خواهیم بود. در چنین حالتی به ازای عنصر اول آرایه دارای $n - ۱$ وارونگی، به ازای عنصر دوم دارای $n - ۲$ وارونگی و به همین ترتیب تا عنصر یکی مانده به آخر آرایه که به ازای آن دارای تنها یک وارونگی خواهیم بود. به این ترتیب تعداد کل وارونگی‌ها به صورت زیر به دست می‌آید:

$$I(n) = \sum_{i=۱}^{n-۱} i = \frac{n(n-۱)}{۲} = \binom{n}{۲}$$

◀ **سوال ۱۳.** آرایه‌ی n عنصری A حاوی جایگشتی از اعداد ۱ تا n است. با n عدد متمایز می‌توان $n!$ جایگشت مختلف تولید کرد. آرایه‌ی A می‌تواند حاوی هر یک از این جایگشت‌ها باشد و از قبل نمی‌دانیم کدام جایگشت در این آرایه قرار دارد. احتمال وجود هر یک از این جایگشت‌ها در آرایه‌ی A را مساوی فرض کنید. اگر الگوریتم (۱۸.۲) بر روی آرایه‌ی A اجرا شود آنگاه دستور انتساب در خط ۶ در این الگوریتم به طور میانگین چند بار اجرا می‌شود؟

الگوریتم ۱۸.۲ یافتن بزرگترین مقدار در یک آرایه

```
1: function FindMax(A)
2:     n = A.length
3:     max = −∞
4:     for i = 1 to n
5:         if A[i] > max
6:             max = A[i]
7:         end if
8:     end for
9:     return max
10: end function
```

◁ **پاسخ سوال ۱۳.**

برای به دست آوردن میانگین تعداد دفعات اجرا شدن دستور انتساب در خط ۶، متغیر تصادفی X_i را به صورت زیر تعریف می‌کنیم:

$$X_i = \begin{cases} ۱ & \text{اگر عنصر } i\text{ام بزرگتر از } i-۱ \text{ عنصر قبلی باشد} \\ ۰ & \text{اگر عنصر } i\text{ام بزرگتر از } i-۱ \text{ عنصر قبلی نباشد} \end{cases}$$

به این ترتیب می‌توان متغیر تصادفی X را برابر با تعداد دفعات اجرای دستور انتساب در نظر گرفت و آن را به صورت زیر تعریف کرد:

$$X = X_۱ + X_۲ + \cdots + X_n = \sum_{i=۱}^{n} X_i$$

با به دست آوردن $\mathrm{E}[X]$ یا همان امید ریاضی متغیر تصادفی X در واقع میانگین تعداد دفعات اجرای دستور انتساب را به دست خواهیم آورد. برای به دست آوردن $\mathrm{E}[X]$ داریم:

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left(\sum_{i=۱}^{n} X_i\right) \\ &= \sum_{i=۱}^{n} \mathrm{E}[X_i] \end{aligned} \tag{۱.۲}$$

پس برای به دست آوردن $\mathrm{E}[X]$ باید مقدار $\mathrm{E}[X_i]$ را به دست آوریم. $\mathrm{E}[X_i]$ برابر با احتمال این رویداد است که عنصر iام بزرگتر از $i-۱$ عنصر قبلی خود باشد. یعنی

$$\mathrm{E}[X_i] = \mathrm{P}(\text{عنصر } i\text{ام بزرگتر از } i-۱ \text{ عنصر قبلی باشد})$$

چون هر یک از i عنصر ابتدایی آرایه با احتمال یکسان ممکن است بزرگترین عنصر در بین این i عنصر باشد در نتیجه احتمال اینکه iامین عنصر بزرگتر از $i-۱$ عنصر قبلی خود باشد برابر با $۱/i$ است و این یعنی $\mathrm{E}[X_i] = ۱/i$. با جایگذاری $۱/i$ در رابطه‌ی (۱.۲) خواهیم داشت:

$$\begin{aligned} \mathrm{E}[X] &= \sum_{i=۱}^{n} \frac{۱}{i} \\ &\approx \lg n \end{aligned}$$

به این ترتیب نشان دادیم دستور انتساب موجود در خط ۶ الگوریتم FindMax به طور میانگین $\lg n$ بار اجرا می‌شود.

◄ **سوال ۱۴.** الگوریتمی بازگشتی بنویسید که آرایه‌ای شامل اعداد صحیح را به عنوان ورودی دریافت کرده و با کمترین تعداد مقایسات، مقادیر بیشینه و کمینه‌ی آرایه‌ی ورودی را به عنوان خروجی بازگرداند.

◁ **پاسخ سوال ۱۴.**

شبه کد الگوریتمی کارا برای یافتن مقادیر بیشینه و کمینه در یک آرایه در الگوریتم (۱۹.۲) آمده است. برای یافتن مقادیر بیشینه و کمینه‌ی آرایه‌ی A این الگوریتم باید به صورت $\textsc{MinMax}(A, 1, A.length)$ فراخوانی شود. روش کار این الگوریتم در ادامه بیان شده است.

الگوریتم ۱۹.۲ یافتن مقادیر بیشینه و کمینه‌ی یک آرایه به صورت همزمان

```
function MinMax(A, low, high)
    if low == high
        min = A[low]
        max = A[low]
    else if low + 1 == high
        if A[low] < A[high]
            min = A[low]
            max = A[high]
        else
            min = A[high]
            max = A[low]
        end if
    else
        min1, max1 = MinMax(A, low + 2, high)
        if A[low] < A[low + 1]
            min = A[low]
            max = A[low + 1]
        else
            min = A[low + 1]
            max = A[low]
        end if
        if min1 < min
            min = min1
        end if
        if max1 > max
            max = max1
        end if
    end if
    return (min, max)
end function
```

اگر آرایه‌ی ورودی دارای تنها یک عنصر باشد آنگاه شرط خط ۲ برقرار بوده و در نتیجه مقدار همین تک عنصر به عنوان مقدار بیشینه و کمینه‌ی آرایه بازگردانده می‌شود و اجرای الگوریتم خاتمه می‌یابد. در صورتی که شرط خط ۵ برقرار باشد به این معنی است که آرایه دارای دو عنصر است و می‌توان با انجام یک مقایسه مقدار بزرگتر را در متغیر max و مقدار کوچکتر را در متغیر min قرار داد و به اجرای الگوریتم خاتمه داد. زمانی که آرایه دارای بیش از دو عنصر باشد با نادیده گرفتن دو عنصر ابتدایی زیرآرایه‌ی $A[low\,..\,high]$، مقادیر بیشینه و کمینه‌ی زیرآرایه‌ی $A[low+۲\,..\,high]$ را به دست می‌آوریم و مقدار کمینه را در متغیر $min1$ و مقدار بیشینه را در $max1$ قرار می‌دهیم. در نهایت با انجام سه مقایسه مقادیر بیشینه و کمینه‌ی کل آرایه را به دست آورده و برگشت می‌دهیم.

۴.۲ ماتریس اسپارس

◄ **سوال ۱۵.** آیا حاصل ضرب دو ماتریس اسپارس لزوماً یک ماتریس اسپارس خواهد بود؟ اگر جواب مثبت است اثبات کنید و اگر جواب منفی است یک مثال نقض ارائه دهید.

◃ **پاسخ سوال ۱۵.**

حاصل ضرب دو ماتریس اسپارس لزوماً یک ماتریس اسپارس نخواهد بود. به بیان دیگر ممکن است ماتریس حاصل شده از ضرب دو ماتریس اسپارس، ماتریسی باشد که حتی یک درایه با مقدار صفر هم ندارد. برای مثال به ضرب دو ماتریس زیر دقت کنید.

$$\begin{bmatrix} ۱ & ۰ & ۰ & ۰ \\ ۱ & ۰ & ۰ & ۰ \\ ۱ & ۰ & ۰ & ۰ \\ ۱ & ۰ & ۰ & ۰ \end{bmatrix} \begin{bmatrix} ۱ & ۱ & ۱ & ۱ \\ ۰ & ۰ & ۰ & ۰ \\ ۰ & ۰ & ۰ & ۰ \\ ۰ & ۰ & ۰ & ۰ \end{bmatrix} = \begin{bmatrix} ۱ & ۱ & ۱ & ۱ \\ ۱ & ۱ & ۱ & ۱ \\ ۱ & ۱ & ۱ & ۱ \\ ۱ & ۱ & ۱ & ۱ \end{bmatrix}$$

ماتریس‌های ضرب شونده هر دو اسپارس هستند اما ماتریس حاصل ضرب یک ماتریس اسپارس نیست.

قبل از پاسخ دادن به سوال ۱۶ قسمت زیر را بخوانید.

اگر از درایه‌های غیر صفر یک ماتریس اسپارس به عنوان درایه‌های مفید یاد کنیم آنگاه می‌توان گفت یک ماتریس اسپارس دارای درایه‌های مفید کمی است. اگر از یک آرایه‌ی دو بعدی برای ذخیره‌ی یک ماتریس اسپارس استفاده کنیم بسیاری از خانه‌های این آرایه حاوی مقادیر غیر مفید (صفر) خواهد بود. به منظور جلوگیری از مصرف بیهوده‌ی حافظه و عدم ذخیره‌ی مقادیر غیر مفید، به جای استفاده از یک آرایه‌ی دو بعدی از داده‌ساختاری که شرح آن در ادامه آمده است استفاده می‌شود.

ماتریس اسپارس M با m سطر و n ستون که دارای t درایه‌ی غیر صفر است را در نظر بگیرید. داده‌ساختاری که از آن برای برای ذخیره‌ی ماتریس اسپارس M استفاده می‌شود یک آرایه‌ی دو بعدی با تعریف $A[۰\,..\,t, ۱\,..\,۳]$ است. در خانه‌ی $A[۰,۱]$ تعداد درایه‌های غیر صفر، در خانه‌ی $A[۰,۲]$ تعداد سطرها و در خانه‌ی $A[۰,۳]$ تعداد ستون‌های ماتریس M قرار می‌گیرد. اگر درایه‌های غیر صفر ماتریس M را به صورت سطر به سطر و از چپ به راست از یک تا t شماره‌گذاری کنیم آنگاه برای پر کردن سایر سطرهای این داده‌ساختار به این صورت

	۱	۲	۳
۰	۵	۵	۸
۱	۱	۱	−۱
۲	۱	۳	۳
۳	۲	۲	۵
۴	۲	۵	۶
۵	۳	۱	۱
۶	۳	۴	۳
۷	۴	۴	۶
۸	۵	۳	۴

شکل (۶.۲): ماتریس اسپارس M که در داده‌ساختار SparseMat ذخیره شده است.

عمل می‌کنیم که شماره‌ی سطر درایه‌ی غیر صفر iام در خانه‌ی $A[i, ۱]$، شماره‌ی ستون این درایه در خانه‌ی $A[i, ۲]$ و مقدار درایه در خانه‌ی $A[i, ۳]$ ذخیره می‌شود. برای مثال اگر ماتریس اسپارس M به صورت زیر باشد آنگاه داده‌ساختار متناظر با این ماتریس به صورت نشان داده شده در شکل (۶.۲) خواهد بود. در ادامه‌ی این فصل برای اشاره به داده‌ساختار شرح داده شده از عنوان SparseMat استفاده خواهد شد.

$$M = \begin{bmatrix} -۱ & ۰ & ۳ & ۰ & ۰ \\ ۰ & ۵ & ۰ & ۰ & ۶ \\ ۱ & ۰ & ۰ & ۳ & ۰ \\ ۰ & ۰ & ۰ & ۶ & ۰ \\ ۰ & ۰ & ۴ & ۰ & ۰ \end{bmatrix}$$

◄ **سوال ۱۶.** الگوریتمی ارائه دهید که ماتریس اسپارس M، ذخیره شده در داده‌ساختاری از نوع SparseMat، را به عنوان ورودی دریافت کرده و ترانهاده‌ی آن را در زمان $O(n + t)$ به دست آورد که در آن t بیانگر تعداد عناصر غیر صفر ماتریس اسپارس M و n نشان‌دهنده‌ی تعداد ستونهای ماتریس M است.

◃ **پاسخ سوال ۱۶.**

شبه کد الگوریتم ترانهاده در الگوریتم (۲۰.۲) نشان داده شده است. ورودی این الگوریتم ماتریس اسپارس M است که در داده‌ساختاری از نوع SparseMat ذخیره شده است و خروجی الگوریتم ماتریس T است که حاوی ترانهاده‌ی ماتریس M است.

الگوریتم ۲۰.۲ ترانهاده‌ی سریع

```
procedure FastTranspose(M)
    let T be a data structure of type SparseMat
    let C[1..n] be a new array of type integer
    n = M[0, 2]
```

ترانهاده‌ی سریع ــ ادامه

```
    t = M[0,3]
    T[0,1] = n
    T[0,2] = M[0,1]
    T[0,3] = t
    if t==0
        return
    else
        for i = 1 to n
            C[i] = 0
        end for
        for i = 1 to t
            C[M[i,2]] = C[M[i,2]] + 1
        end for
        for i = 2 to n
            C[i] = C[i-1] + C[i]
        end for
        for i = 1 to t
            j = C[M[i,2]]
            T[j,1] = M[i,2]
            T[j,2] = M[i,1]
            T[j,3] = M[i,3]
            C[M[i,2]] = j + 1
        end for
    end if
end procedure
```

ایده‌ی مورد استفاده در الگوریتم (۲۰.۲) مانند ایده‌ی مورد استفاده در الگوریتم مرتب‌سازی شمارشی[۸] است. حلقه‌ی اول به تمام عناصر آرایه‌ی C مقدار صفر را نسبت می‌دهد. سپس حلقه‌ی دوم تعداد عناصر غیر صفر در هر ستون ماتریس اسپارس را به دست می‌آورد به طوری که پس از پایان این حلقه، $C[i]$ بیانگر تعداد عناصر غیر صفر ستون iام ماتریس اسپارس است. پس از پایان حلقه‌ی سوم، $C[i]$ بیانگر مجموع تعداد عناصر غیر صفر در ستون‌های ۱ تا i ماتریس اسپارس است. در نهایت حلقه‌ی چهارم کار ترانهاده کردن را انجام داده و هر یک از عناصر ماتریس M را در مکان مناسب از ماتریس T قرار می‌دهد. به این ترتیب در انتهای اجرای این الگوریتم ماتریس T حاوی ترانهاده‌ی ماتریس M خواهد بود.

برای محاسبه‌ی مرتبه‌ی زمانی الگوریتم FastTranspose باید گفت حلقه‌های اول و سوم از مرتبه‌ی $O(n)$

[۸] Counting sort

و حلقه‌های دوم و چهارم از مرتبه‌ی $O(t)$ هستند. در نتیجه مرتبه‌ی زمانی الگوریتم FastTranspose برابر با $O(n+t)$ است.

۵.۲ ماتریس‌های خاص

◄ **سوال ۱۷.** فرض کنید A و B دو ماتریس پایین مثلثی با n سطر و n ستون هستند. تعداد کل مقادیر غیر صفر این دو ماتریس در مجموع برابر با $n(n+۱)$ است (ماتریس A دارای $n(n+۱)/۲$ مقادیر غیر صفر و ماتریس B نیز دارای همین تعداد مقادیر غیر صفر است). با در اختیار داشتن ماتریس C با n سطر و $n+۱$ ستون روشی ارائه دهید که بتوان مقادیر غیر صفر دو ماتریس A و B را به طور همزمان در ماتریس C ذخیره کرد.

◃ **پاسخ سوال ۱۷.**

برای ذخیره‌سازی همزمان مقادیر غیر صفر دو ماتریس پایین مثلثی A و B در ماتریس C، کافیست مقادیر غیر صفر ماتریس A را به صورت پایین مثلثی و ترانهاده‌ی مقادیر غیر صفر ماتریس B را به صورت بالا مثلثی در ماتریس C ذخیره کنیم. برای درک بهتر نحوه‌ی ذخیره‌سازی به مثالی که در ادامه آمده است توجه کنید.

$$A=\begin{bmatrix} ۱ & ۰ & ۰ \\ ۲ & ۳ & ۰ \\ ۴ & ۵ & ۶ \end{bmatrix}, B=\begin{bmatrix} a & ۰ & ۰ \\ b & c & ۰ \\ d & e & f \end{bmatrix} \Rightarrow C=\begin{bmatrix} ۱ & a & b & d \\ ۲ & ۳ & c & e \\ ۴ & ۵ & ۶ & f \end{bmatrix}$$

با در نظر گرفتن این روش ذخیره‌سازی می‌توان فرمولی ارائه داد که به کمک آن بتوان بازیابی مقادیر از ماتریس C را به درستی انجام داد. با توجه به اینکه ماتریس A بدون هیچ تغییری در ماتریس C ذخیره شده است پس برای دسترسی به درایه‌ی $A[i,j]$ می‌توان مقدار $C[i,j]$ را بازیابی کرد. برای بازیابی $B[i,j]$ نیز می‌توان مقدار $C[j,i+۱]$ را بازیابی کرد.

◄ **سوال ۱۸.** به ماتریس‌های مربعی که بر روی قطر اصلی و همچنین چند قطر بالا و پایین قطر اصلی دارای مقادیر غیر صفر هستند، ماتریس‌های نواری گفته می‌شود. ماتریس‌های نواری نوع خاصی از ماتریس‌های اسپارس به شمار می‌آیند. در زیر نمونه‌ای از یک ماتریس سه نواری نمایش داده شده است. حالت کلی یک ماتریس a نواری به این صورت است که $a-۱$ قطر بالای قطر اصلی و $a-۱$ قطر پایین قطر اصلی و خود قطر اصلی دارای مقادیر غیر صفر هستند.

$$\begin{bmatrix} ۱ & ۳ & ۸ & ۰ & ۰ & ۰ \\ ۵ & ۲ & ۱ & ۱ & ۰ & ۰ \\ ۹ & ۱ & ۶ & ۸ & ۷ & ۰ \\ ۰ & ۶ & ۱ & ۱ & ۴ & ۲ \\ ۰ & ۰ & ۷ & ۳ & ۴ & ۹ \\ ۰ & ۰ & ۰ & ۲ & ۳ & ۱ \end{bmatrix}$$

با در نظر گرفتن توضیحات داده شده به موارد زیر پاسخ دهید.

آ. در یک ماتریس a نواری با n سطر و n ستون چه تعداد عنصر غیر صفر وجود دارد؟

ب. برای اینکه درایه‌ای مانند $M[i,j]$ از ماتریس a نواری M حاوی مقدار غیر صفر باشد، چه رابطه‌ای باید میان اندیس‌های i و j برقرار باشد؟

ج. یکی از روشهای ذخیره‌سازی یک ماتریس a نواری استفاده از یک آرایه یک بعدی مانند B است که در آن ابتدا سمت راست‌ترین قطر از بالا به پایین، سپس قطر کنار آن و به همین ترتیب تا چپ‌ترین قطر ذخیره می‌شوند. فرمولی ارائه دهید که به کمک آن بتوان تعیین کرد عنصری غیر صفر از ماتریس M مانند $M[i,j]$ در کدام خانه از آرایه‌ی B قرار می‌گیرد.

◁ **پاسخ سوال ۱۸ .**

در یک ماتریس a نواری با n سطر و n ستون دارای n عنصر غیر صفر بر روی قطر اصلی، $n-۱$ عنصر غیر صفر بر روی قطر بالای قطر اصلی و $n-۱$ عنصر غیر صفر بر روی پایین قطر اصلی هستیم و به همین ترتیب. می‌توان تعداد کل عناصر غیر صفر را در قالب مجموع زیر بیان کرد:

$$n+\sum_{i=۱}^{a-۱} ۲(n-i)$$

ب. اگر $M[i,j]$ عنصری با مقدار غیر صفر از ماتریس a نواری M باشد آنگاه $M[i,j]$ در یکی از سه حالت زیر صدق می‌کند:

۱. $M[i,j]$ بر روی قطر اصلی قرار دارد که در این صورت داریم $i=j$.

۲. $M[i,j]$ بر روی یکی از $a-۱$ قطر بالای قطر اصلی قرار دارد که در این صورت اندیس j باید حداکثر $a-۱$ واحد از اندیس i بزرگتر باشد یعنی $i<j\leq i+a-۱$.

۳. $M[i,j]$ بر روی یکی از $a-۱$ قطر پایین قطر اصلی قرار دارد که در این حالت اندیس i باید حداکثر $a-۱$ واحد از اندیس j بزرگتر باشد یعنی $j<i\leq j+a-۱$.

ج. اگر فرض کنیم ماتریس a نواری M با n سطر و n ستون را در آرایه‌ی یک بعدی B با تعریف $B[۱..n+\sum_{i=۱}^{a-۱} ۲(n-i)]$ ذخیره کرده‌ایم آنگاه برای بازیابی مقدار $M[i,j]$ باید یکی از سه حالت زیر را در نظر گرفت:

- اگر $i=j$:

$$M[i,j]=B[i+\sum_{k=۱}^{a-۱}(n-k)]$$

- اگر $i > j$:

$$M[i,j] = B[i + n + \sum_{k=۱}^{a-۱} (n-k) + \sum_{k=۱}^{i-j+۱} (a-k)]$$

- اگر $i < j$:

$$M[i,j] = B[i + \sum_{k=j-i+۱}^{a-۱} (n-k)]$$

فصل ۵

درخت‌های عمومی، دودویی و هیپ

۱.۵ مقدمه

داده‌ساختار درخت یکی از مهمترین و کاربردی‌ترین داده‌ساختارها است. انواع مختلفی از این داده‌ساختار وجود دارد و سوالات مطرح شده در این فصل تنها بخش کوچکی از این انواع را پوشش می‌دهد.

داده‌ساختار درخت کاربردهای فراوانی در علم کامپیوتر دارد که از جمله کاربردهای آن می‌توان به پیاده‌سازی سیستم فایل‌های سلسله مراتبی اشاره کرد مانند آنچه در سیستم عامل ویندوز یا گنو/لینوکس وجود دارد. همچنین شالوده‌ی سامانه‌ی نام دامنه[۱] از ساختار درختی پیروی می‌کند.

با توجه به اینکه برای برخی از مفاهیم داده‌ساختار درخت، مانند سطح، عمق، ارتفاع و غیره، در کتاب‌های مرجع تعاریف مختلفی ارائه شده است در ادامه تعاریفی که از آنها در این فصل استفاده شده است آورده خواهد شد.

سطح[۲] یک گره: سطح گره x برابر است با تعداد یال‌هایی که باید از گره ریشه طی کرد تا به گره x رسید بعلاوه یک. به این ترتیب گره ریشه دارای سطح یک است، فرزندان ریشه دارای سطح دو هستند و به همین ترتیب.

عمق[۳] یک گره: عمق گره x برابر است با تعداد یال‌هایی که باید از گره ریشه طی کرد تا به گره x رسید. در این صورت گره ریشه دارای عمق صفر است، فرزندان ریشه دارای عمق یک هستند و به همین ترتیب. عمق درخت T برابر با عمق عمیق‌ترین گره درخت T است.

ارتفاع[۴] یک گره: ارتفاع گره x برابر است با تعداد یالهای طولانی‌ترین مسیری که گره x را به یک گره برگ متصل می‌کند. با این تعریف می‌توان نتیجه گرفت تمام گره‌های برگ در یک درخت دارای ارتفاع صفر هستند. ارتفاع درخت T برابر با ارتفاع ریشه آن است.

[۱] Domain Name System (DNS)
[۲] Level
[۳] Depth
[۴] Height

درخت kتایی پر: درخت T یک درخت kتایی پر است اگر تمام گره‌های غیر برگ آن دارای دقیقا k فرزند باشند و همچنین تمام گره‌های برگ آن در یک سطح قرار گرفته باشند.

ساختار گره‌ی درخت دودویی: ساختار گره‌ی یک درخت دودویی دارای سه فیلد $data$، $leftChild$ و $rightChild$ است که از اولین فیلد برای نگهداری مقدار داده‌ای و از فیلدهای دوم و سوم به ترتیب برای اشاره به فرزند چپ و راست گره استفاده می‌شود.

ساختار گره‌ی درخت عمومی: ساختار گره‌ی یک درخت عمومی دارای سه فیلد $data$، $next$ و $children$ است که از فیلد اول برای نگهداری مقدار داده‌ای، از فیلد دوم برای اشاره به همنیای سمت راست و از فیلد سوم برای اشاره به لیست فرزندان گره استفاده می‌شود.

۲.۵ منابع مطالعاتی

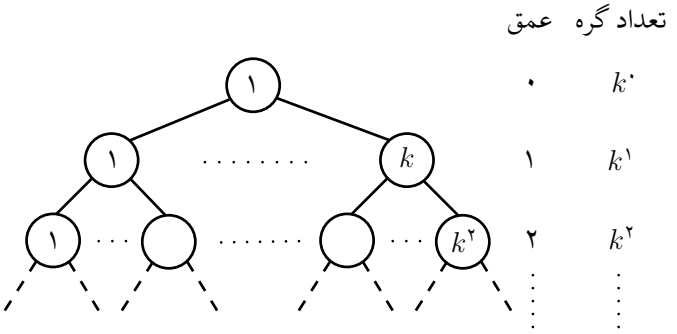
برای مطالعه در مورد تعاریف و فرمول‌های درخت و همچنین درخت‌های عمومی و دودویی می‌توانید به فصل هفتم [۲] و فصل پنجم [۳] مراجعه کنید. برای کسب اطلاعات در مورد درخت‌های هیپ فصل نهم [۲] و فصل ششم [۴] منابع مناسبی هستند.

۳.۵ روابط حاکم بر درخت‌ها

◄ **سوال ۱.** ثابت کنید عمق یک درخت kتایی پر با n گره، برابر با $\lfloor \log_k n \rfloor$ است.

◁ **پاسخ سوال ۱.**

در عمق صفر یک درخت kتایی پر دارای تنها یک گره هستیم که همان گره‌ی ریشه است. در عمق یک چنین درختی دارای k گره خواهیم بود زیرا درخت پر است و گره ریشه باید دارای دقیقاً k فرزند باشد. در عمق دو دارای k^2 گره خواهیم بود و به همین ترتیب. به عبارت بهتر تعداد گره‌ها در عمق d ($d \neq 0$)، k برابر تعداد گره‌های عمق $d-1$ است. برای درک بهتر این موضوع به شکل (۱.۵) توجه کنید.



شکل (۱.۵): بخشی از یک درخت kتایی پر

با در نظر گرفتن شکل (۱.۵) تعداد کل گره‌ها، که آن را با n نمایش می‌دهیم، در یک درخت kتایی پر به عمق m به صورت زیر به دست می‌آید:

$$\begin{aligned} n &= ۱ + k + k^۲ + k^۳ + \cdots + k^m \\ &= \frac{k^{m+۱} - ۱}{k - ۱} \end{aligned} \tag{۱.۵}$$

اگر در عبارت $\lfloor \log_k n \rfloor$ به جای n، مقدار به دست آمده از (۱.۵) را قرار دهیم باید داشته باشیم:

$$\left\lfloor \log_k \frac{k^{m+۱} - ۱}{k - ۱} \right\rfloor = m \tag{۲.۵}$$

به این ترتیب باید ثابت کنیم تساوی (۲.۵) برقرار است. در ادامه، به اثبات برقراری این تساوی می‌پردازیم.

با در نظر گرفتن خاصیت تابع جزء صحیح می‌توان تساوی (۲.۵) را به صورت زیر نوشت:

$$m \leq \log_k \frac{k^{m+۱} - ۱}{k - ۱} < m + ۱ \tag{۳.۵}$$

اگر طرفین نامعادله (۳.۵) را به عنوان توان عدد k در نظر بگیریم خواهیم داشت:

$$k^m \leq k^{\log_k \frac{k^{m+۱}-۱}{k-۱}} < k^{m+۱} \Rightarrow k^m \leq \frac{k^{m+۱} - ۱}{k - ۱} < k^{m+۱} \tag{۴.۵}$$

با ضرب طرفین نامعادله (۴.۵) در $k - ۱$ داریم:

$$k^m(k - ۱) \leq k^{m+۱} - ۱ < k^{m+۱}(k - ۱) \tag{۵.۵}$$

با ساده‌سازی نامعادله (۵.۵) به عبارت زیر می‌رسیم:

$$k^{m+۱} - k^m \leq k^{m+۱} - ۱ < k^{m+۱}(k - ۱) \tag{۶.۵}$$

درستی نامعادله (۶.۵) را می‌توان با استقرا بر روی m نشان داد و اثبات را کامل کرد. انجام استقرا به خواننده واگذار می‌شود.

◄ **سوال ۲.** اگر $n_۰$ تعداد گره‌های برگ و $n_۲$ تعداد گره‌های با دو فرزند در درخت دودویی T باشند آنگاه درستی رابطه‌ی $n_۰ = n_۲ + ۱$ را نشان دهید.

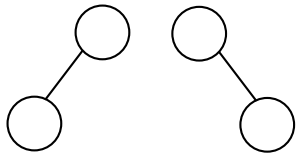
◃ **پاسخ سوال ۲.**

اگر n را برابر با تعداد کل گره‌ها و $n_۱$ را برابر با تعداد گره‌های تک فرزندی در درخت دودویی T در نظر بگیریم چون تمامی گره‌ها در درخت T دارای حداکثر دو فرزند هستند در نتیجه رابطه‌ی (۷.۵) برقرار خواهد بود.

$$n = n_۰ + n_۱ + n_۲ \tag{۷.۵}$$

در هر درخت تعداد کل گره‌ها همواره یک واحد از تعداد کل یال‌های درخت بیشتر است و این یعنی اگر تعداد کل یال‌ها را با e نشان دهیم خواهیم داشت:

$$n = e + ۱ \tag{۸.۵}$$



شکل (۲.۵): درخت‌های دودویی مختلفی که با دو گره می‌توان ساخت

از طرفی تمامی یال‌های درخت یا از یک گره تک فرزندی و یا از یک گره دو فرزندی منشعب شده‌اند و این یعنی $e = n_1 + 2n_2$.

اگر در (۸.۵) به جای e معادل آن یعنی $n_1 + 2n_2$ را قرار دهیم به رابطه‌ی (۹.۵) می‌رسیم.

$$n = n_1 + 2n_2 + 1 \tag{۹.۵}$$

به کم کردن (۹.۵) از (۷.۵) و مرتب کردن نتیجه، به رابطه‌ی $n_0 = n_2 + 1$ می‌رسیم.

◄ **سوال ۳.** تعداد درخت‌های دودویی مختلفی که با صفر گره می‌توان ساخت برابر با یک است (درخت تهی). تعداد درخت‌های دودویی مختلفی که با یک گره می‌توان ساخت نیز برابر با یک است و چنین درختی فقط دارای عنصر ریشه است. درخت‌های دودویی مختلفی که می‌توان با دو گره ساخت در شکل (۲.۵) نشان داده شده است. با n گره چه تعداد درخت دودویی متفاوت می‌توان ساخت؟

◃ **پاسخ سوال ۳.**

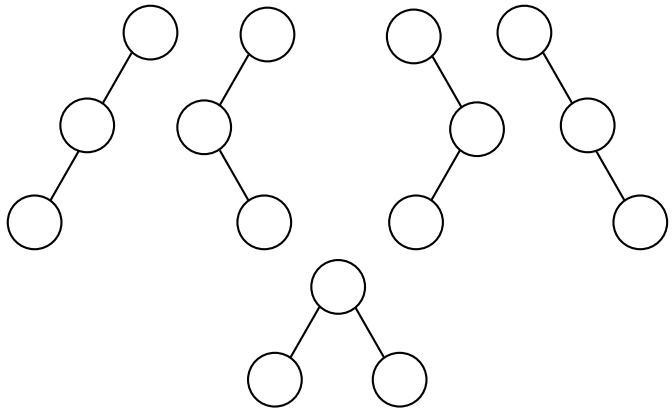
فرض کنید دارای n گره هستیم که آنها را از ۱ تا n شماره‌گذاری کرده‌ایم و این گره‌ها را به صورت زیر در کنار هم قرار داده‌ایم:

$$1, 2, 3, \cdots, n$$

با در نظر گرفتن این شماره‌گذاری، مجدداً فرض کنید که گره شماره‌ی i ($1 \leqslant i \leqslant n$) ریشه‌ی یک درخت دودویی باشد و تمام گره‌های با شماره کمتر از i در زیردرخت چپ i و تمام گره‌های با شماره بیشتر از i در زیردرخت راست i قرار بگیرند. در این‌صورت زیردرخت چپ ریشه دارای $i - 1$ گره و زیردرخت راست دارای $n - i$ گره خواهند بود.

اگر $T(n)$ را برابر با تعداد درختهای دودویی مختلفی که با n گره می‌توان ساخت در نظر بگیریم آنگاه تعداد زیردرخت‌های چپ و راست درخت با ریشه i به ترتیب برابر با $T(i-1)$ و $T(n-i)$ خواهند بود. با توجه به اصل شماره دو شمارش، یعنی اصل ضرب، حاصل عبارت $T(i-1) \cdot T(n-i)$ برابر با تعداد کل درخت‌های دودویی ممکن با ریشه i است. i می‌تواند هر یک از مقادیر ۱ تا n را اختیار کند. در نتیجه به منظور در نظر گرفتن کلیه‌ی حالات گره‌ی ریشه، باید مقدار عبارت $T(i-1) \cdot T(n-i)$ را برای همه‌ی مقادیر i با یکدیگر جمع کنیم. بدین ترتیب به رابطه بازگشتی (۱۰.۵) می‌رسیم.

$$T(n) = \begin{cases} T(0) = T(1) = 1 & i = 0, 1 \\ \sum_{i=1}^{n} T(i-1) \cdot T(n-i) & i > 1 \end{cases} \tag{۱۰.۵}$$



شکل (۳.۵): درخت‌های دودویی مختلفی که با سه گره می‌توان ساخت

با حل رابطه‌ی بازگشتی (۱۰.۵) به رابطه‌ی (۱۱.۵) می‌رسیم.

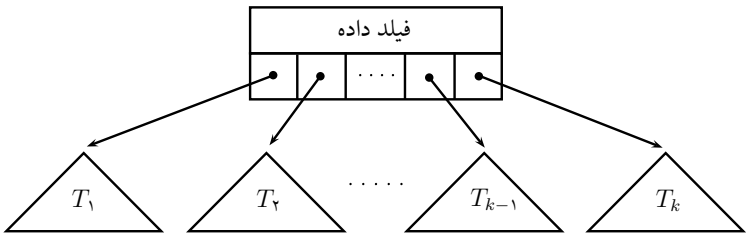
$$T(n) = \frac{\binom{۲n}{n}}{n+۱} \tag{۱۱.۵}$$

به این ترتیب و با استفاده از رابطه‌ی (۱۱.۵) می‌توان تعداد درخت‌های دودویی مختلفی که با n گره می‌توان ساخت را به دست آورد. برای مثال تعداد درخت‌های دودویی مختلفی که با ۳ گره می‌توان ساخت برابر است با:

$$T(۳) = \frac{\binom{۶}{۳}}{۴} = \frac{۲۰}{۴} = ۵$$

این درخت‌ها در شکل (۳.۵) نشان داده شده‌اند. به دنباله‌ی اعداد حاصل از رابطه‌ی (۱۱.۵)، دنباله اعداد کاتالان گفته می‌شود.

◄ **سوال ۴.** فرض کنید برای پیاده‌سازی درخت‌های kتایی از گره‌هایی با قالب نشان داده شده در شکل (۴.۵) استفاده شده است. در چنین ساختاری دارای یک فیلد داده‌ای و k فیلد اشاره‌گر برای اشاره به هر یک از k زیردرخت یک گره هستیم. اگر دارای n گره در چنین درختی باشیم آنگاه تعداد اشاره‌گرهای تهی چه تعداد خواهد بود؟



شکل (۴.۵): ساختار یک گره در یک درخت kتایی

◃ **پاسخ سوال ۴.**

در چنین ساختاری هر گره دارای k اشاره‌گر است. در نتیجه هنگامی که n گره در درخت وجود دارد دارای nk اشاره‌گر هستیم. از این تعداد اشاره‌گر فقط $n - ۱$ اشاره‌گر مورد استفاده قرار می‌گیرند زیرا برای اشاره به هر یک از گره‌ها به جز گره ریشه به یک اشاره‌گر نیاز است. اگر تعداد کل اشاره‌گرها را از تعداد اشاره‌گرهای مورد استفاده کم کنیم خواهیم داشت:

$$nk - (n - ۱) = nk - n + ۱ = n(k - ۱) + ۱$$

به این ترتیب تعداد اشاره‌گرهای تهی برابر با $n(k - ۱) + ۱$ است.

۴.۵ الگوریتم‌های درخت‌های عمومی و دودویی

◄ **سوال ۵.** با فرض پیاده‌سازی درخت عمومی به کمک اشاره‌گرها و اینکه هر گره موظف است آدرس فرزندان خود را در یک لیست پیوندی نگاه دارد، تابعی بازگشتی بنویسید که بررسی کند آیا عنصری با مقدار x در درخت داده شده وجود دارد یا خیر. اگر x موجود بود مقدار TRUE و در غیر اینصورت مقدار FALSE به عنوان خروجی تابع برگردانده شود.

◃ **پاسخ سوال ۵.**

شبه کد تابعی بازگشتی که در درخت $gTree$ به دنبال عنصری با مقدار x می‌گردد در الگوریتم (۱.۵) آمده است. در ادامه به بررسی روش عملکرد این تابع پرداخته می‌شود.

الگوریتم ۱.۵ بررسی وجود عنصری با مقدار مشخص در یک درخت عمومی

```
function FIND(gTree, x)
    if gTree == NULL
        return
    end if
    if gTree.data == x
        return TRUE
    end if
    found = FALSE
    p = gTree.children
    while p ≠ NULL and found == FALSE
        found = FIND(p, x)
        p = p.next
    end while
    return found
end function
```

در ابتدا بررسی می‌شود که آیا درخت تهی است یا خیر. اگر اینگونه بود مقدار FALSE به عنوان خروجی برگشت داده می‌شود. اگر درخت تهی نبود باید بررسی کرد که آیا مقدار عنصر ریشه برابر با x است یا خیر. اگر برابر بود آنگاه مقدار TRUE برگشت داده می‌شود. اگر هیچ یک از دو شرط ابتدایی برقرار نبودند آنگاه باید به دنبال مقدار x در فرزندان گره ریشه و در صورت لزوم در فرزندانِ فرزندان گره ریشه و به همین ترتیب بگردیم تا یا عنصر مورد نظر پیدا شود و یا از عدم وجود آن در درخت اطمینان حاصل نماییم. جستجو در فرزندان ریشه، فرزندانِ فرزندان ریشه و ... توسط حلقه موجود در تابع به صورت بازگشتی انجام می‌شود. برای درک بهتر این الگوریتم آن را بر روی یک درخت عمومی ساده امتحان کنید.

◄ **سوال ۶.** با در نظر گرفتن پیاده‌سازی درخت عمومی به کمک اشاره‌گرها و اینکه هر گره موظف است آدرس فرزندان خود را در یک لیست پیوندی نگاه دارد، تابعی بازگشتی بنویسید که اشاره‌گر به یک گره را دریافت کرده و اشاره‌گر به پدر گره‌ی ورودی را بازگرداند. تابع شما باید دارای تنها دو ورودی باشد. ورودی اول اشاره‌گر به ریشه‌ی درختی است که باید جستجو در آن انجام شود و ورودی دوم اشاره‌گر به گره‌ای است که قصد یافتن پدر آن را داریم.

◃ **پاسخ سوال ۶.**

شبه کد تابعی بازگشتی که در درخت $gTree$ به دنبال پدر گره‌ای می‌گردد که n به آن اشاره دارد در الگوریتم (۲.۵) آورده شده است.

الگوریتم ۲.۵ یافتن پدر یک گره‌ی خاص در یک درخت عمومی

```
function FindParent(gTree, n)
    if gTree == NULL
        return NULL
    end if
    p = gTree.children
    while p ≠ NULL
        if p == n
            return t
        end if
        p = p.next
    end while
    p = gTree.children
    while p ≠ NULL
        q = FindParent(p, n)
        if q ≠ NULL
            return q
        end if
        p = p.next
```

یافتن پدر یک گره‌ی خاص در یک درخت عمومی ــ ادامه

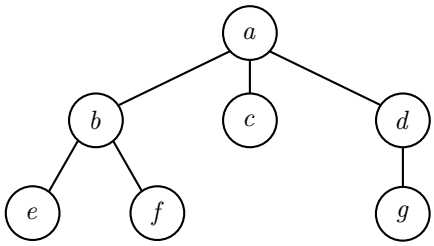
```
    end while
    return NULL
end function
```

اگر درخت $gTree$ تهی باشد آنگاه عمل خاصی لازم نیست و مقدار تهی برگشت داده می‌شود. در صورتی که درخت $gTree$ تهی نباشد آنگاه باید در فرزندان گره ریشه به دنبال عنصری بگردیم که n به آن اشاره دارد. اگر چنین عنصری یافت شد آنگاه مقدار $gTree$ به عنوان خروجی بازگردانده می‌شود و به این معنی است که گره‌ای که n به آن اشاره دارد یکی از فرزندان گره‌ی ریشه است و در نتیجه گره‌ی ریشه پدر آن است (خطوط ۵ تا ۱۱). اگر عنصری که n به آن اشاره دارد یکی از فرزندان گره‌ی ریشه نباشد آنگاه باید به صورت بازگشتی در فرزندان گره‌ی ریشه به دنبال عنصر n بگردیم (خطوط ۱۲ تا ۱۹). اگر به خط انتهایی تابع برسیم بدین معنی است که قادر به یافتن عنصری که n به آن اشاره دارد در درخت $gTree$ نبوده‌ایم و در نتیجه مقدار تهی به عنوان خروجی تابع برگردانده می‌شود.

◄ **سوال ۷.** تابعی بازگشتی بنویسید که یک درخت عمومی، پیاده‌سازی شده توسط لیست عمومی، را به عنوان ورودی دریافت کرده و به عنوان خروجی، درختی دودویی را برگرداند که با استفاده از روش فرزند چپ ــ همنیای راست[۵] از درخت ورودی به دست آمده است.

◃ **پاسخ سوال ۷.**

در ابتدا ببینیم از چه نوع درختی باید به چه نوع درختی برسیم. درخت سه تایی نشان داده شده در شکل (۵.۵) را در نظر بگیرید. اگر از پیاده‌سازی لیست عمومی برای نمایش این درخت استفاده کنیم آنگاه به درخت عمومی نشان داده شده در شکل (۶.۵) می‌رسیم.

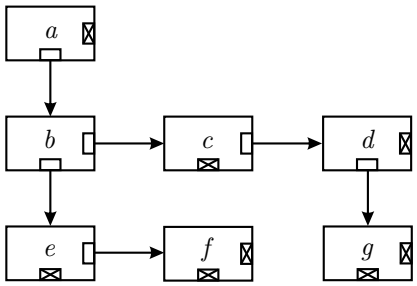


شکل (۵.۵): یک درخت سه تایی

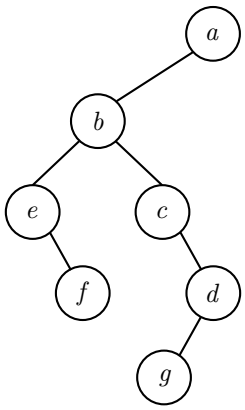
الگوریتم ایجاد درخت دودویی از روی درخت عمومی، باید اشاره‌گر به درختی مانند آنچه در شکل (۶.۵) نشان داده شده است را دریافت کند و با استفاده از روش فرزند چپ ــ همنیای راست درختی دودویی با ساختاری که در شکل (۷.۵) نشان داده شده است را به عنوان خروجی بازگرداند.

تابع بازگشتی ایجاد درخت دودویی از روی درخت عمومی با استفاده از روش فرزند چپ ــ همنیای راست در قالب الگوریتم (۳.۵) نشان داده شده است. در ادامه به بررسی روش کار این الگوریتم پرداخته می‌شود.

[۵] Left child - right sibling



شکل (۶.۵): نمایش درخت شکل (۵.۵) با استفاده از لیست عمومی



شکل (۷.۵): درخت دودویی حاصل از اجرای الگوریتم فرزند چپ ـ همنیای راست بر روی درخت شکل (۵.۵)

الگوریتم ۳.۵ ایجاد درخت دودویی ار روی درخت عمومی به روش فرزند چپ ـ همنیای راست

```
function Gen2Bin(gTree)
    if gTree == NULL
        return NULL
    end if
    New(bTree)
    bTree.data = gTree.data
    q = gTree.children
    if q == NULL
        bTree.leftChild = NULL
        bTree.rightChild = NULL
        return bTree
    end if
    bTree.leftChild = Gen2Bin(q)
    p = bTree.leftChild
    p.rightChild = NULL
    q = q.next
```

ایجاد درخت دودویی از روی درخت عمومی به روش فرزند چپ ـ همنیای راست ـ ادامه

```
17:     while q ≠ NULL
18:         p.rightChild = Gen2Bin(q)
19:         p = p.rightChild
20:         p.rightChild = NULL
21:         q = q.next
22:     end while
23:     return bTree
24: end function
```

در صورتی که درخت عمومی تهی باشد نیاز به انجام کار خاصی نیست و مقدار NULL به عنوان خروجی الگوریتم برگشت داده می‌شود.

اگر درخت عمومی دارای یک گره باشد آنگاه یک گره برای درخت دودویی ایجاد شده و مقدار داده‌ای تنها گره درخت عمومی در فیلد داده‌ای گره‌ی تازه ایجاد شده قرار می‌گیرد و پس از تنظیم اشاره‌گرهای گره‌ی تازه ایجاد شده الگوریتم Gen2Bin اشاره‌گر به گره تازه ایجاد شده، که همان ریشه درخت دودویی است، را به عنوان خروجی باز می‌گرداند (خطوط ۵ تا ۱۲).

در صورتی که درخت عمومی دارای بیش از یک گره باشد آنگاه در مرحله اول (خطوط ۱۳ تا ۱۶) با استفاده از یک فراخوانی بازگشتی، از روی زیردرختی که سمت چپ‌ترین فرزند گره‌ی ریشه‌ی درخت عمومی است، یک درخت دودویی به دست می‌آید و به عنوان فرزند چپ گره‌ی ریشه در درخت دودویی اصلی قرار می‌گیرد. در مرحله دوم (خطوط ۱۷ تا ۲۲) به کمک فراخوانی بازگشتی، به ازای هر یک از فرزندان باقی مانده‌ی گره‌ی ریشه یک درخت دودویی به دست می‌آوریم و طبق روش فرزند چپ ـ همنیای راست در مکان مناسب از درخت دودویی اصلی درج می‌کنیم.

در انتهای اجرای الگوریتم اشاره‌گر $bTree$ که به ریشه‌ی درخت دودویی ایجاد شده از روی درخت عمومی ورودی اشاره دارد به عنوان خروجی الگوریم برگشت داده می‌شود.

شکل (۸.۵) روند ایجاد درخت دودویی پس از مراحل مختلف اجرای الگورریتم Gen2Bin بر روی درخت شکل (۵.۵) را نشان می‌دهد.

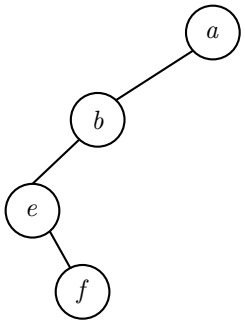
◄ **سوال ۸.** پیاده‌سازی درخت دودویی با استفاده از آرایه را در نظر بگیرید. برای ذخیره‌ی یک درخت دودویی با n گره، در بدترین حالت به آرایه‌ای با چند خانه نیاز داریم؟ چه تعداد از این خانه‌ها بدون استفاده باقی می‌مانند؟

◃ **پاسخ سوال ۸.**

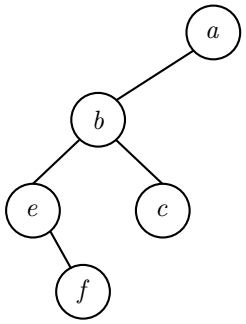
در صورتی که درخت دودویی مورب به راست و هر سطح از درخت دارای تنها یک گره باشد آنگاه بیشترین تعداد خانه مورد نیاز خواهد بود. شکل (۹.۵) نشان دهنده‌ی یک درخت دودویی مورب به راست است.

در چنین درختی، که دارای n سطح است، دارای تنها n گره هستیم اما تعداد خانه‌های لازم برای ذخیره‌سازی چنین درختی برابر با تعداد خانه‌های لازم برای یک درخت دودویی پر با n سطح است. تعداد خانه‌های لازم

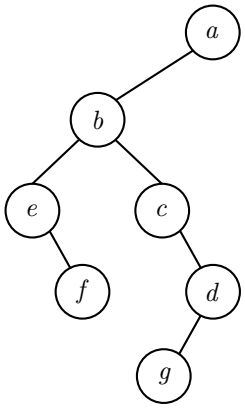
(آ) درخت دودویی پس از اجرای خطوط ۵ تا ۱۲



(ب) درخت دودویی پس از اجرای خطوط ۱۳ تا ۱۶

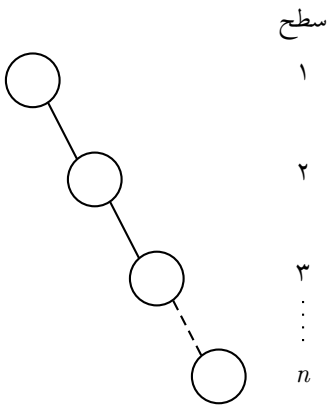


(ج) درخت دودویی پس از اجرای دور اول حلقه



(د) درخت دودویی پس از خاتمه‌ی دور دوم و آخر حلقه

شکل (۸.۵): روند ایجاد درخت دودویی با اجرای الگوریتم GEN2BIN بر روی درخت عمومی شکل (۵.۵)



شکل (۹.۵): درخت دودویی مورب به راست

برای ذخیره‌سازی یک درخت دودویی پر با n سطح برابر است با:

$$۱ + ۲ + ۲^۲ + ۲^۳ + \cdots + ۲^n = \frac{۲^n - ۱}{۲ - ۱} = ۲^n - ۱$$

در نتیجه، به $۲^n - ۱$ خانه احتیاج خواهیم داشت که از این تعداد فقط n خانه مورد استفاده قرار می‌گیرد و $۲^n - ۱ - n$ خانه از آرایه بدون استفاده باقی می‌ماند.

◄ **سوال ۹.** درختی دودویی را در نظر بگیرید که به روش فرزند چپ ـ همنیای راست از یک درخت عمومی به دست آمده است. تابعی بازگشتی بنویسید که چنین درخت دودویی را دریافت کرده و به عنوان خروجی، درخت عمومی معادل با درخت دودویی را بازگرداند. تنها ورودی تابع شما باید اشاره‌گر به ریشه‌ی درخت دودویی باشد.

◃ **پاسخ سوال ۹.**

الگوریتمی که در این سوال به دنبال آن هستیم وارونه‌ی الگوریتمی است که در سوال ۷ به دنبال آن بودیم. در سوال ۷ از روی درخت عمومی باید درخت دودویی به دست می‌آوردیم اما در این سوال باید از روی درخت دودویی به درخت عمومی برسیم. شبه کد تابع ایجاد درخت عمومی از روی درخت دودویی در الگوریتم (۴.۵) آورده شده است. در ادامه به بیان روش کار این الگوریتم می‌پردازیم.

الگوریتم ۴.۵ ایجاد درخت عمومی از روی درخت دودویی

```
function Bin2Gen(bTree)
    if bTree == NULL
        return NULL
    end if
    New(gTree)
    gTree.data = bTree.data
    gTree.next = NULL
    gTree.children = NULL
    q = gTree
    p = bTree.leftChild
    if p ≠ NULL
        q.children = Bin2Gen(p)
        q = q.children
        q.next = NULL
        p = p.rightChild
    end if
    while p ≠ NULL
        q.next = Bin2Gen(p)
```

ایجاد درخت عمومی از روی درخت دودویی ــ ادامه

```
19:         p = p.rightChild
20:         q = q.next
21:     end while
22:     return gTree
23: end function
```

در صورتی که درخت دودویی ورودی تهی باشد کار بسیار ساده است و تنها باید مقدار NULL به عنوان خروجی الگوریتم برگشت داده شود.

اگر درخت دودویی دارای یک گره باشد آنگاه یک گره برای درخت عمومی ایجاد می‌شود و مقدار داده‌ای تنها گره درخت دودویی در فیلد داده‌ای گره‌ی تازه ایجاد شده قرار می‌گیرد. پس از تنظیم اشاره‌گرهای گره‌ی تازه ایجاد شده الگوریتم BIN2GEN اشاره‌گر به گره تازه ایجاد شده، که همان ریشه درخت عمومی است، را به عنوان خروجی باز می‌گرداند (خطوط ۶ تا ۱۰).

اگر درخت دودویی ورودی دارای بیش از یک گره باشد آنگاه در مرحله اول (خطوط ۱۱ تا ۱۶) با استفاده از یک فراخوانی بازگشتی، از روی زیردرخت چپ ریشه‌ی درخت دودویی، یک درخت عمومی به دست می‌آید و در مکان مناسب از درخت عمومی اصلی درج می‌شود. در مرحله دوم (خطوط ۱۷ تا ۲۱) به کمک فراخوانی بازگشتی، با دنبال کردن فرزندان راست زیر دخت چپ ریشه درخت دودویی به ازای هر یک از این گره‌ها، یک درخت عمومی به دست می‌آوریم و در مکان مناسب از درخت عمومی اصلی درج می‌کنیم.

در انتهای اجرای الگوریتم اشاره‌گر $gTree$ که به ریشه‌ی درخت عمومی ایجاد شده از روی درخت دودویی ورودی اشاره دارد به عنوان خروجی الگوریم برگشت داده می‌شود.

◄ **سوال ۱۰.** تابعی بازگشتی بنویسید که اشاره‌گر به ریشه‌ی یک درخت دودویی را دریافت کرده و تعداد سطوح آن را بازگرداند.

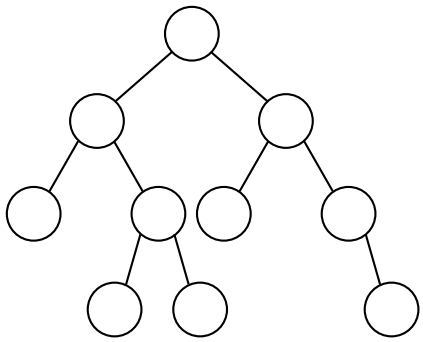
◁ **پاسخ سوال ۱۰.**

شبه کد الگوریتم یافتن تعداد سطوح یک درخت دودویی در الگوریتم (۵.۵) نشان داده شده است. اگر درخت دودویی ورودی تهی باشد مقدار صفر به عنوان تعداد سطوح درخت بازگردانده می‌شود. در غیر این صورت به صورت بازگشتی به ترتیب تعداد سطوح زیردرخت چپ و راست ریشه به دست آورده می‌شود. سپس تعداد سطوح زیردرخت چپ و راست با یکدیگر مقایسه شده و مقدار بزرگتر با عدد یک جمع شده و در متغیر $level$ قرار می‌گیرد. دلیل جمع شدن مقدار بزرگتر با عدد یک این است که با انجام دو فراخوانی بازگشتی تعداد سطوح دو زیردرخت ریشه را به دست آورده‌ایم و با در نظر گرفتن گره ریشه باید یک واحد به تعداد سطوح درخت اضافه شود. در نهایت متغیر $level$ که بیانگر تعداد سطوح درخت است به عنوان خروجی الگوریتم برگشت داده می‌شود.

◄ **سوال ۱۱.** پهنای یک درخت برابر است با تعداد گره‌های سطحی که دارای بیشترین تعداد گره در میان تمام سطوح درخت است. برای مثال درخت نشان داده شده در شکل (۱۰.۵) دارای پهنای چهار است زیرا سطح اول دارای یک گره، سطح دوم دارای دو گره، سطح سوم دارای چهار گره و سطح چهارم دارای یک گره است

الگوریتم ۵.۵ به دست آوردن تعداد سطوح یک درخت دودویی

```
function TreeLevel(t)
    if t == NULL
        return 0
    end if
    leftLevel = TreeLevel(t.leftChild)
    rightLevel = TreeLevel(t.rightChild)
    if leftLevel > rightLevel
        level = leftLevel + 1
    else
        level = rightLevel + 1
    end if
    return level
end function
```



شکل (۱۰.۵): درخت دودویی با پهنای چهار

و در نتیجه حداکثر تعداد گره در یک سطح از این درخت برابر با چهار است و این یعنی پهنای درخت نیز برابر با چهار است. تابعی بازگشتی بنویسید که اشاره‌گر به ریشه‌ی یک درخت دودویی را به عنوان ورودی دریافت کرده و پهنای درخت را به عنوان خروجی بازگرداند (فرض کنید درخت ورودی تهی نیست).

◁ **پاسخ سوال ۱۱.**

شبه کد تابع به دست آوردن پهنای یک درخت دودویی در الگوریتم (۶.۵) نشان داده شده است. این تابع علاوه بر پهنای درخت، شماره سطحی را که تعیین کننده پهنای درخت هست نیز برمی‌گرداند. برای به دست آوردن پهنای درختی دودویی که T به ریشه آن اشاره دارد تابع TreeWidth باید به صورت $\text{TreeWidth}(T, 1)$ فراخوانی شود. روش کار این تابع در ادامه بیان می‌شود.

زمانی که درخت ورودی تهی است مقدار صفر هم برای سطح و هم برای پهنای درخت بازگردانده می‌شود. اگر درخت ورودی دارای تنها یک عنصر باشد، که همان عنصر ریشه است، مقدار یک هم برای سطح و هم برای پهنای درخت بازگردانده می‌شود و این یعنی درخت دارای پهنای یک است و همچنین سطحی که دارای این پهنا

است سطح شماره یک است. زمانی که درخت ورودی دارای بیش از یک عنصر است از فراخوانی بازگشتی استفاده می‌کنیم تا پهنای زیردرخت چپ و راست گره ریشه را به دست بیاوریم. چون پهنای زیر درخت چپ و راست گره ریشه به صورت جداگانه محاسبه می‌شوند باید بررسی کنیم تا در صورتی که $leftLevel$ برابر با $rightLevel$ باشد، مقادیر $leftWidth$ و $rightWidth$ با یکدیگر جمع شوند تا پهنای کلی درخت به دست آید. در صورتی که $leftLevel$ برابر با $rightLevel$ نباشد آنگاه مقدار بزرگتر از میان دو مقدار $leftWidth$ و $rightWidth$ به عنوان پهنای درخت و سطح متناظر با مقدار بزرگتر به عنوان سطحی که تعیین کننده پهنای درخت است به عنوان خروجی تابع بازگردانده می‌شوند.

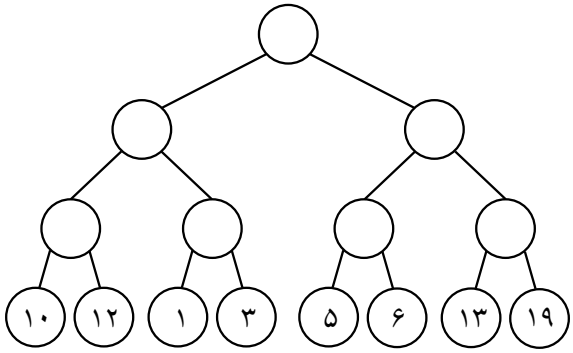
الگوریتم ۶.۵ به دست آوردن پهنای یک درخت دودویی

```
function TreeWidth(T, currentLevel)
    if T.leftChild == NULL and T.rightChild == NULL
        return (1, currentLevel)
    end if
    leftWidth, leftLevel = TreeWidth(T.leftChild, currentLevel + 1)
    rightWidth, rightLevel = TreeWidth(T.rightChild, currentLevel + 1)
    if leftLevel == rightLevel
        width = leftWidth + rightWidth
        level = leftLevel
    else
        width = Max(leftWidth, rightWidth)
        if leftWidth > rightWidth
            width = leftWidth
            level = leftLevel
        else
            width = rightWidth
            level = rightLevel
        end if
    end if
    return (width, level)
end function
```

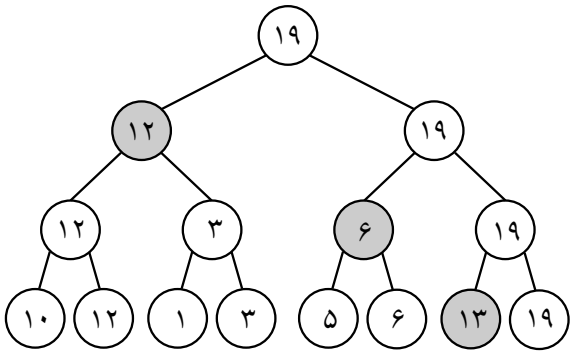
◄ **سوال ۱۲.** ساده‌ترین الگوریتم برای یافتن دومین بزرگترین مقدار در یک آرایه‌ی n خانه‌ای نیاز به تقریباً $۲n$ مقایسه دارد. گام‌های کلی الگوریتمی را شرح دهید که دومین بزرگترین مقدار در یک آرایه را با حداکثر $n + ۲\lceil \lg n \rceil - ۲$ مقایسه پیدا می‌کند (راهنمایی: فرض کنید تعداد عناصر آرایه زوج است و از درخت برنده‌ـ بازنده[۶] استفاده کنید).

◃ **پاسخ سوال ۱۲.**

[۶] Winner-loser tree



شکل (۱۱.۵): درخت برنده‌ـ بازنده در ابتدای مرحله اول از اجرای الگوریتم یافتن دومین بزرگترین مقدار



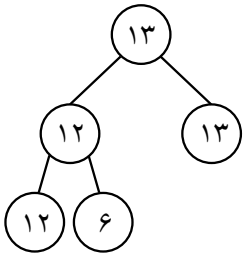
شکل (۱۲.۵): درخت برنده‌ـ بازنده در انتهای مرحله اول از اجرای الگوریتم یافتن دومین بزرگترین مقدار

الگوریتم یافتن دومین بزرگترین مقدار در یک آرایه یک بعدی دارای دو مرحله است و در هر مرحله درختی دودویی با نام درخت برنده‌ـ بازنده ساخته می‌شود.

در یک درخت برنده‌ـ بازنده تمامی عناصر آرایه ورودی به عنوان برگ‌های درخت قرار می‌گیرند. اگر فرض کنیم آرایه ورودی دارای هشت خانه است و حاوی اعداد ۱۹، ۱۳، ۶، ۵، ۳، ۱، ۱۲، ۱۰ است آنگاه درخت برنده‌ـ بازنده‌ی ابتدایی در مرحله اول اجرای الگوریتم به صورت شکل (۱۱.۵) خواهد بود. در هر سطح، عناصری که دارای پدر یکسان هستند با یکدیگر مقایسه شده و مقدار بزرگتر به سطح بالاتر یعنی به گره‌ی پدر آن دو گره‌ای که با یکدیگر مقایسه شده‌اند کپی می‌شود. این روند به همین ترتیب ادامه می‌یابد تا در نهایت گره ریشه بدست آید که همان بزرگترین مقدار آرایه است. درخت نهایی مرحله اول اجرای الگوریتم در شکل (۱۲.۵) به نمایش در آمده است.

در این درخت با هر مقایسه‌ای که انجام می‌شود یک برنده، یعنی عنصر با مقدار بزرگتر، و یک بازنده، یعنی عنصر با مقدار کوچکتر، مشخص می‌شوند و مقدار عنصر برنده به سطح بالاتر منتقل می‌شود. با توجه به اینکه هر عنصری غیر از بزرگترین عنصر، دقیقاً یک بار باید بازنده شده باشد می‌توان گفت که تعداد مقایسات لازم برای ساخت چنین درختی برابر با $n - ۱$ است.

در مرحله دوم اجرای الگوریتم می‌توان گفت عنصر با دومین بزرگترین مقدار، عنصری است که تنها به بزرگترین عنصر، یعنی عنصر ریشه، باخته است. در نتیجه کافی است در مسیر حرکت از گره‌ی ریشه به سمت پایین، در مسیری که بزرگترین عنصر برای رسیدن به گره ریشه طی کرده است حرکت کرده و مقادیری که در مقایسه با گره‌ی ریشه بازنده بوده‌اند را مشخص کنیم (گره‌های خاکستری در شکل (۱۲.۵)).



شکل (۱۳.۵): درخت برنده‌ـ بازنده در انتهای مرحله دوم از اجرای الگوریم یافتن دومین بزرگترین مقدار

با توجه به دودویی بودن درخت و تعداد سطوح چنین درختی، تعداد عناصر بازنده به بزرگترین عنصر حداکثر برابر با $\lfloor \lg n \rfloor$ خواهد بود. لذا $\lfloor \lg n \rfloor$ مقایسه نیاز خواهیم داشت تا تمام عناصر بازنده به بزرگترین عنصر را به دست بیاوریم. حال برای این $\lfloor \lg n \rfloor$ عنصر نیز یک درخت برنده‌ـ بازنده می‌سازیم (شکل (۱۳.۵)) و سپس با $\lfloor \lg n \rfloor - ۱$ مقایسه بزرگترین عنصر این درخت که همان دومین بزرگترین عنصر آرایه است را می‌یابیم.

به این ترتیب می‌توان گفت حداکثر تعداد مقایسات برای یافتن دومین بزرگترین مقدار در آرایه‌ای n خانه‌ای برابر با $n + ۲\lfloor \lg n \rfloor - ۲$ است.

◄ **سوال ۱۳.** ثابت کنید هر الگوریتم مرتب‌سازی که از مقایسه‌ی عناصر برای تعیین ترتیب صحیح عناصر استفاده می‌کند در بهینه‌ترین حالت از مرتبه $\Omega(n \lg n)$ است.

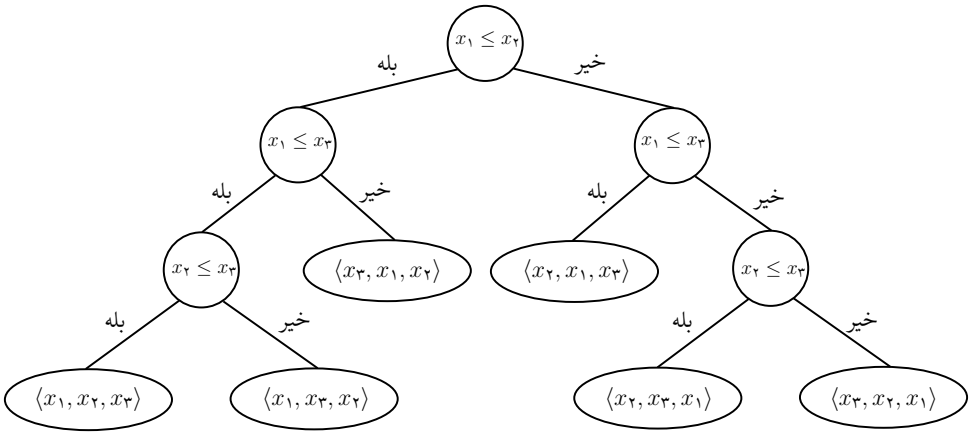
◁ **پاسخ سوال ۱۳.**

در ابتدا یک مثال ساده را بررسی می‌کنیم. فرض کنید قصد مرتب‌سازی آرایه‌ای سه عنصری را داریم و هر سه عنصر متمایز از یکدیگر هستند. برای نمایش روند مرتب‌سازی عناصر می‌توان از یک درخت دودویی استفاده کرد. در چنین درختی برچسب هر گره نمایش‌دهنده یک مقایسه میان دو عنصر آرایه است و برچسب هر یال نشان دهنده‌ی نتیجه مقایسه دو عنصر است. ترتیب مرتب شده عناصر را می‌توان با حرکت از ریشه‌ی درخت تا یکی از برگ‌ها بدست آورد زیرا هر برگ بیانگر یکی از جایگشت‌های عناصر آرایه است. به چنین درختی، یک درخت تصمیم دودویی[۷] گفته می‌شود. شکل (۱۴.۵) نشان دهنده‌ی درخت تصمیم دودویی برای یک آرایه سه عنصری است (در این شکل x_i به معنی مقدار خانه iام آرایه است).

هر الگوریتم مرتب‌سازی مبتنی بر مقایسه، با توجه به مقایساتی که انجام می‌دهد، یک درخت تصمیم دودویی را ایجاد می‌کند. در چنین درختی، طولانی‌ترین مسیر از گره‌ی ریشه به یک گره‌ی برگ بیانگر بدترین حالت ممکن در اجرای الگوریتم است. یعنی در چنین حالتی الگوریتم برای مرتب‌سازی عناصر به بیشترین تعداد مقایسات نیاز دارد. همچنین بهترین حالت در اجرای الگوریتم معادل با کوتاه‌ترین مسیر از گره‌ی ریشه به یک گره‌ی برگ است. حالت متوسط نیز از تقسیم تعداد یال‌های موجود در درخت بر تعداد برگ‌های موجود بدست می‌آید. در حالت متوسط در حقیقت مشخص می‌کنیم به طور متوسط تعداد یال‌هایی که باید طی شوند تا به یک برگ برسیم چه تعداد است.

اگرچه ممکن است در نگاه اول بتوان با رسم درخت تصمیم دودویی برای هر الگوریتم مرتب‌سازی، به تعیین

[۷] Binary decision tree



شکل (۱۴.۵): درخت تصمیم دودویی برای مرتب‌سازی سه عنصر

طول کوتاهترین و طولانی‌ترین مسیر پرداخت امّا حالتی را در نظر بگیرید که قصد مرتب‌سازی آرایه‌ای با ۱۰ عنصر را داریم. درخت تصمیم دودویی چنین آرایه‌ای دارای حداقل !۱۰ برگ است (چون ممکن است برخی از جایگشت‌ها بیش از یکبار ظاهر شوند دارای حداقل این تعداد برگ هستیم) و دارای حداقل ۲۲ سطح است. در نتیجه رسم چنین درختی با این ابعاد منطقی به نظر نمی‌رسد. پس باید دید چگونه می‌توان با استفاده از ایده‌ی درخت تصمیم دودویی به مرتبه زمانی $\Omega(n \lg n)$ رسید.

باید در حالت کلی بررسی کنیم که حداقل عمق یک درخت تصمیم دودویی برای مرتب‌سازی n عنصر چه مقداری است. با تعیین این مقدار در حقیقت حداقل تعداد مقایسات در یک الگوریتم مرتب‌سازی مبتنی بر مقایسه را به دست آورده‌ایم.

می‌دانیم که یک درخت تصمیم دودویی برای آرایه‌ای n عنصری دارای حداقل $n!$ برگ است. در ادامه حداقل عمق ممکن برای درخت تصمیمی با $n!$ برگ را محاسبه می‌کنیم. اگر عمق ریشه را برابر با صفر در نظر بگیریم می‌توان گفت در عمق k تعداد گره‌ها برابر است با ۲^k. لذا برای داشتن درختی با حداقل $n!$ برگ، اگر k را کمترین عمق ممکن برای درخت در نظر بگیریم آنگاه باید داشته باشیم:

$$n! \leq ۲^k \tag{۱۲.۵}$$

با لگاریتم گرفتن از طرفین نامعادله (۱۲.۵) به نامعادله (۱۳.۵) می‌رسیم.

$$\lg n! \leq k \tag{۱۳.۵}$$

با درنظر گرفتن این که $\lg n!$ تقریباً برابر با $n \lg n - ۱/۵$ است (چرا؟) می‌توان گفت حداقل مقدار k تقریباً برابر با $n \lg n$ است. بدین ترتیب حداقل عمق یک درخت تصمیم دودویی با $n!$ برگ برابر با $n \lg n$ است و این یعنی تعداد مقایسات در یک الگوریتم مرتب‌سازی مبتنی بر مقایسه در بهینه‌ترین حالت از مرتبه $\Omega(n \lg n)$ است. در نتیجه می‌توان گفت هیچ الگوریتم مرتب‌سازی مبتنی بر مقایسه نمی‌تواند در زمانی بهتر از $\Omega(n \lg n)$ آرایه‌ای n عنصری را مرتب کند.

۵.۵ درخت‌های هیپ

◂ **سوال ۱۴.** در یک درخت هیپ بیشینه[۸] با n عنصر متمایز، اعمال زیر را با چه مرتبه‌ای می‌توان انجام داد؟ توضیح دهید.

- به دست آوردن مجموع همه‌ی اعداد موجود در هیپ
- به دست آوردن مجموع $\lg n$ بزرگترین اعداد موجود در هیپ
- به دست آوردن مجموع ۱۰ عدد بزرگ موجود در هیپ

◃ **پاسخ سوال ۱۴.**

برای به دست آوردن مجموع اعداد موجود در درخت هیپ کافیست درخت را با استفاده از یک پیمایش دلخواه، مثلا میان‌ترتیب، پیمایش کرده و در حین پیمایش با ملاقات هر گره مقدار آن را به مقدار مجموع اضافه کنیم. با توجه به اینکه مرتبه پیمایش میان‌ترتیب برابر با $O(n)$ است پس مرتبه اجرایی این الگوریتم نیز برابر با $O(n)$ است.

برای به دست آوردن مجموع $\lg n$ بزرگترین اعداد موجود در هیپ می‌توان به تعداد $\lg n$ بار عمل حذف عنصر ریشه را انجام داد و مقدار عدد حذف شده را به مقدار مجموع اضافه کرد. به این ترتیب در بار اول بزرگترین عدد، که در ریشه هیپ است، حذف شده و مقدار آن به مقدار مجموع اضافه می‌شود سپس دومین بزرگترین عنصر حذف شده و به مقدار مجموع اضافه شده و به همین ترتیب. با توجه به اینکه هر عمل حذف از هیپ از مرتبه $O(\lg n)$ است پس مرتبه کلی الگوریتم برابر خواهد بود با $\lg n \times O(\lg n) = O(\lg^{۲} n)$.

به دست آوردن مجموع ۱۰ عدد بزرگتر هیپ حالت خاصی از الگوریتم قبلی است. یعنی کافیست ۱۰ بار عنصر ریشه را حذف کرده و هر بار مقدار عنصر حذف شده را به یک متغیر که حاصل جمع را نگه می‌دارد اضافه کنیم. مرتبه زمانی انجام چنین کاری برابر است با $۱۰ \times O(\lg n) = O(\lg n)$

◂ **سوال ۱۵.** در یک درخت هیپ بیشینه حاوی n عدد متمایز، چهارمین بزرگترین عدد ممکن است در کدامیک از خانه‌های آرایه‌ی حاوی هیپ بیشینه قرار داشته باشد؟

◃ **پاسخ سوال ۱۵.**

برای تعیین اینکه چهارمین بزرگترین مقدار می‌تواند در کدامیک از خانه‌های آرایه قرار بگیرد باید به طور کلی به دنبال این باشیم که kامین بزرگترین مقدار در چه سطوحی از درخت هیپ می‌تواند ظاهر شود. سپس شماره‌ی خانه‌های متناظر این سطوح در آرایه را یافته و به این ترتیب بازه‌ای از خانه‌های آرایه که می‌تواند حاوی kامین بزرگترین مقدار باشد مشخص می‌شود.

بزرگترین مقدار همواره در سطح اول درخت که تنها شامل گره‌ی ریشه است قرار می‌گیرد. دومین بزرگترین مقدار می‌تواند در یکی از دو گره‌ی سطح دوم قرار بگیرد. سومین بزرگترین مقدار می‌تواند در یکی از گره‌های سطوح دوم یا سوم قرار بگیرد. می‌توان اثبات کرد که kامین بزرگترین مقدار می‌تواند در یکی از سطوح ۲ تا k قرار بگیرد (طبق آنچه گفته شد برای حالت خاص $k = ۱$ این رابطه برقرار نیست و بزرگترین مقدار در سطح یک قرار خواهد گرفت).

[۸] Max heap

بزرگترین مقدار در خانه‌ی شماره یک آرایه قرار می‌گیرد. دومین بزرگترین مقدار می‌تواند در سطح دو (خانه‌های دو یا سه) یا سطح سه (خانه‌های چهار تا هفت) قرار بگیرد (در مجموع خانه‌های دو تا هفت). به همین ترتیب می‌توان گفت kامین بزرگترین مقدار می‌تواند در یکی از خانه‌های ۲ تا $۲^k - ۱$ قرار بگیرد.

با توجه به اینکه یک فرمول کلی به دست آوردیم در نتیجه می‌توان گفت چهارمین بزرگترین مقدار می‌تواند در یکی از خانه‌های دو تا پانزده آرایه قرار بگیرد.

◄ **سوال ۱۶.** یک درخت هیپ بیشینه شامل اعداد ۱ تا ۱۰۲۳ را در نظر بگیرید. الف) چه تعداد از اعداد بزرگتر از ۱۰۰۰ می‌توانند در گره‌های برگ چنین درختی قرار بگیرند؟ ب) حداکثر چه تعداد از اعداد بزرگتر از ۱۰۰۰ می‌توانند بطور همزمان برگ باشند؟

◁ **پاسخ سوال ۱۶.**

برای پاسخ به این سوال به دو نکته‌ی زیر توجه می‌کنیم:

۱. در یک درخت هیپ kامین بزرگترین عنصر می‌تواند در یکی از سطوح دوم، سوم، ...، kام قرار بگیرد.

۲. یک درخت هیپ که حاوی اعداد ۱ تا ۱۰۲۳ است دارای ده سطح است.

الف)

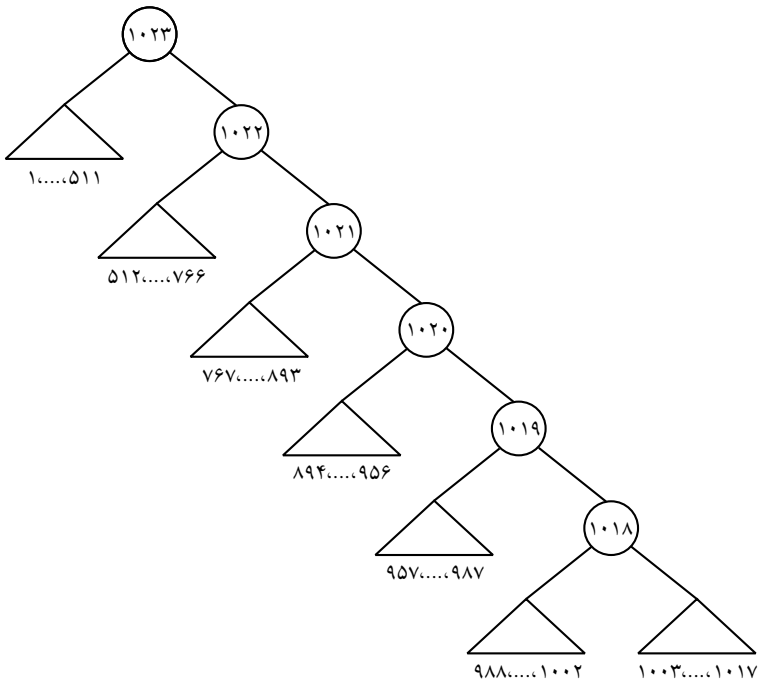
با در نظر گرفتن دو نکته‌ی مطرح شده می‌توان گفت عدد ۱۰۲۳ در سطح اول، عدد ۱۰۲۲ در سطح دوم، عدد ۱۰۲۱ در یکی از سطوح دوم یا سوم و به همین ترتیب تا عدد ۱۰۱۵ که در یکی از سطوح دوم تا نهم قرار می‌گیرند. در نتیجه می‌توان گفت اعداد ۱۰۱۵ تا ۱۰۲۳ نمی‌توانند در سطح دهم قرار بگیرند و این یعنی این اعداد نمی‌توانند به عنوان برگ این درخت هیپ بیشینه ظاهر شوند. به بیان دیگر چهارده عدد بزرگتر از ۱۰۰۰ این امکان را دارند که به عنوان برگ درخت هیپ ظاهر شوند و این اعداد عبارتند از ۱۰۰۱ تا ۱۰۱۴.

ب)

برای پاسخ به این قسمت باید طوری اعداد را در هیپ بیشینه قرار دهیم که تا حد ممکن بیشترین اعداد در بازه ۱۰۰۱ تا ۱۰۲۳ در گره‌های برگ ظاهر شوند.

می‌دانیم که بزرگترین مقدار یعنی ۱۰۲۳ در سطح اول، دومین بزرگترین مقدار یعنی ۱۰۲۲ در سطح دوم، سومین بزرگترین مقدار در یکی از سطوح دوم یا سوم و به همین ترتیب ظاهر می‌شوند. حال برای آنکه بیشترین تعداد از اعداد در بازه ۱۰۰۱ تا ۱۰۲۳ در برگ‌ها ظاهر شوند، سعی می‌کنیم اعداد ۱ تا ۵۱۱ را زیردرخت چپ ریشه و اعداد ۵۱۲ تا ۱۰۲۲ را در زیردرخت راست ریشه قرار دهیم (می‌توانستیم به صورت برعکس نیز عمل کنیم بدین معنی که اعداد ۱ تا ۵۱۱ را زیردرخت راست ریشه و اعداد ۵۱۲ تا ۱۰۲۲ را در زیردرخت چپ ریشه قرار دهیم). با توجه به اینکه اعداد مورد نظر ما یعنی ۱۰۰۱ تا ۱۰۲۲ در زیردرخت راست قرار دارند در نتیجه برای ادامه کار فقط به زیردرخت راست توجه می‌کنیم و زیردرخت چپ ریشه را نادیده می‌گیریم.

در زیردرخت راست، عدد ۱۰۲۲ را به عنوان گره ریشه این زیردرخت در نظر می‌گیریم و در زیردرخت چپ این گره اعداد ۵۱۲ تا ۷۶۶ و در زیردرخت راست آن اعداد ۷۶۷ تا ۱۰۲۱ را قرار می‌دهیم. مجدداً برای زیردرخت با ریشه ۱۰۲۲ سعی می‌کنیم اعداد ۷۶۷ تا ۸۹۳ را در زیردرخت چپ و اعداد ۸۹۴ تا ۱۰۲۱ را در زیردرخت راست قرار دهیم. اگر همین روند را ادامه دهیم در سطح ششم درخت به حالتی می‌رسیم که در



شکل (۱۵.۵): درخت هیپ بیشینه حاوی اعداد ۱ تا ۱۰۲۳

آن باید اعداد ۹۸۸ تا ۱۰۱۸ را در درخت قرار دهیم. در این حالت نیز به این صورت عمل می‌کنیم که عدد ۱۰۱۸ را در ریشه، اعداد ۹۸۸ تا ۱۰۰۲ را در زیردرخت چپ و اعداد ۱۰۰۳ تا ۱۰۱۷ را در زیردرخت راست قرار می‌دهیم. درخت هیپ مورد نظر تا این مرحله در شکل (۱۵.۵) نشان داده شده است.

در این حالت می‌توان گفت زیردرخت حاوی اعداد ۱۰۰۳ تا ۱۰۱۷ دارای چهار سطح است و در نتیجه دارای هشت برگ است. به این ترتیب اعداد ۱۱۶، ۱۱۵ و ۱۱۴ با توجه به اینکه اولین، دومین و سومین بزرگترین مقادیر در این زیردرخت هستند در نتیجه نمی‌توانند در سطح چهارم این زیردرخت (یعنی به عنوان گره‌ی برگ) ظاهر شوند. پس می‌توان گفت در زیردرخت راست گره‌ای که دارای مقدار ۱۰۱۸ در ریشه خود است، هشت عنصر بزرگتر از ۱۰۰۰ و کوچکتر از ۱۰۱۴ می‌توانند به طور همزمان در برگهای درخت هیپ ظاهر شوند.

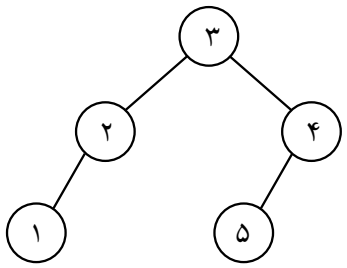
در مورد ۱۰۰۱ و ۱۰۰۲ باید گفت با توجه به اینکه ۱۰۰۱ و ۱۰۰۲ در زیردرخت حاوی این اعداد، به ترتیب اولین و دومین بزرگترین عناصر هستند در نتیجه در زیردرخت خود نمی‌توانند به عنوان برگ ظاهر شوند.

به عنوان نتیجه‌گیری می‌توان گفت اعداد ۱۰۰۱ تا ۱۰۱۴ این امکان را دارند که به عنوان گره‌ی برگ در درخت هیپ بیشینه ظاهر شوند اما فقط هشت عدد از میان اعداد ۱۰۰۳ تا ۱۰۱۳ این امکان را دارند که بطور همزمان در گره‌های برگ ظاهر شوند.

◄ **سوال ۱۷.** داده‌ساختار صف اولویت میانه[۹]، یک داده‌ساختار شامل n عنصر با مقادیر متمایز است که می‌توان اعمال زیر را بر روی آن انجام داد:

- درج یک عنصر در زمان $O(\lg n)$
- حذف عنصر دارای مقدار میانه در زمان $O(\lg n)$

[۹] Median priority queue



شکل (۱۶.۵): داده‌ساختار صف اولویت میانه که در آن عنصر میانه در ریشه درخت دودویی قرار دارد

با استفاده از داده‌ساختار درخت هیپ، داده‌ساختار صف اولویت میانه را طراحی کنید. سپس نحوه‌ی انجام اعمال نامبرده با پیچیدگی زمانی خواسته شده را شرح دهید.

◁ **پاسخ سوال ۱۷.**

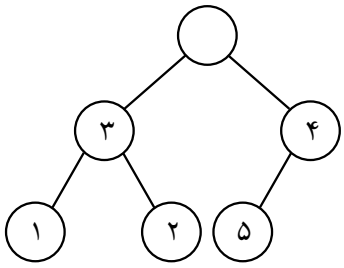
عنصر میانه در یک لیست، عنصری است که از نیمی از داده‌های لیست بزرگتر و از نیمی از داده‌ها کوچکتر است. از همین ویژگی برای طراحی داده‌ساختار صف اولویت میانه استفاده خواهیم کرد.

در لیستی با تعداد عناصر فرد، عنصر میانه عنصری است که پس از مرتب کردن عناصر لیست در وسط لیست قرار می‌گیرد. به طور مثال لیست $L = \langle ۴, ۲, ۳, ۵, ۱ \rangle$ را در نظر بگیرید. پس از مرتب کردن عناصر لیست L، به لیست $L = \langle ۱, ۲, ۳, ۴, ۵ \rangle$ می‌رسیم. بدین ترتیب میانه لیست L عدد ۳ خواهد بود. به عبارت بهتر اگر در لیستی با تعداد عناصر فرد تعداد عناصر را با n نشان دهیم عنصر میانه پس از مرتب‌سازی لیست در مکان $\lfloor n/۲ \rfloor + ۱$ قرار خواهد گرفت. در لیستی n عنصری با زوج عنصر، با توجه به اینکه دارای عنصر وسط نیستیم، می‌توان پس از مرتب کردن لیست، عنصر موجود در مکان $\lfloor n/۲ \rfloor$ یا $\lfloor n/۲ \rfloor + ۱$ را به عنوان عنصر میانه در نظر گرفت.

فرض کنید می‌خواهیم عنصر میانه را از لیست $L = \langle ۴, ۲, ۳, ۵, ۱ \rangle$ حذف کنیم. در این صورت باید عدد ۲ (بزرگترین عددی که کوچکتر از میانه است) یا ۴ (کوچکترین عددی که از میانه بزرگتر است) را به عنوان عنصر میانه‌ی جدید برگزینیم. برای این منظور می‌توان درختی دودویی را در نظر گرفت که در آن زیردرخت چپ ریشه یک درخت هیپ بیشینه و زیردرخت راست ریشه یک درخت هیپ کمینه است. عناصر کمتر از میانه‌ی جاری را در هیپ بیشینه و عناصر بیشتر از میانه‌ی جاری را در هیپ کمینه نگه می‌داریم. شکل این روش ذخیره‌سازی داده‌ساختار صف اولویت میانه برای لیستی مانند $L = \langle ۴, ۲, ۳, ۵, ۱ \rangle$ به صورت نشان داده شده در شکل (۱۶.۵) خواهد بود.

می‌توانیم عنصر میانه را در یکی از دو درخت هیپ نیز قرار دهیم. برای مثال اگر عنصر میانه را در هیپ بیشینه قرار دهیم داده‌ساختار صف اولویت میانه مانند آنچه در شکل (۱۷.۵) نشان داده شده است خواهد بود.

در ادامه فرض می‌کنیم که از حالت ذخیره‌سازی دوم استفاده شده است. با این روش پیاده‌سازی، برای حذف عنصر میانه، در شرایطی که تعداد عناصر هیپ بیشینه و هیپ کمینه با یکدیگر برابر نباشند، همواره عنصر ریشه‌ی درختی که تعداد عناصرش یکی بیشتر از دیگری است را به عنوان میانه جدید انتخاب می‌کنیم. اما اگر تعداد عناصر هر دو هیپ با یکدیگر برابر باشند می‌توان به دلخواه هر یک از دو ریشه‌ی هیپ‌های مذکور را به عنوان میانه‌ی جدید انتخاب کرد.



شکل (۱۷.۵): داده‌ساختار صف اولویت میانه که در آن عنصر میانه در ریشه زیردرخت چپ درخت دودویی قرار دارد

در ادامه بدین صورت قرارداد می‌کنیم که اگر تعداد عناصر هر دو هیپ با یکدیگر برابر بودند آنگاه عنصر ریشه هیپ بیشینه به عنوان میانه جدید در نظر گرفته خواهد شد. در ادامه الگوریتم‌های مربوط به اعمال درج و حذف را شرح داده و نشان می‌دهیم که هر دو از مرتبه $O(\lg n)$ هستند.

برای حذف عنصر میانه در ابتدا باید بررسی کنیم که هر کدام از دو درخت هیپ دارای چه تعداد عنصر هستند. برای این کار می‌توان از دو متغیر استفاده کرد که هر کدام تعداد عناصر یکی از هیپ‌ها را نگه می‌دارد. اگر تعداد عناصر دو هیپ با یکدیگر برابر نبود، کافیست ریشه‌ی درخت هیپی که یک عنصر بیشتر دارد را به عنوان میانه‌ی جدید برگزینیم و آن ریشه را از هیپ حذف کنیم که در این صورت تعداد عناصر در هر دو هیپ با یکدیگر برابر خواهد شد. اما در صورتی که تعداد عناصر در هر دو هیپ با یکدیگر برابر باشد آنگاه عنصر ریشه هیپ بیشینه به عنوان میانه جدید در نظر گرفته می‌شود و این عنصر از درخت هیپ بیشینه حذف می‌شود. همانطور که می‌دانیم مرتبه زمانی عمل حذف یک عنصر از یک هیپ n عنصری برابر با $O(\lg n)$ است. اگر تعداد عناصر هیپ بیشینه را n_1 و تعداد عناصر هیپ کمینه را n_2 در نظر بگیریم آنگاه حذف ریشه از هیپ بیشینه از مرتبه $O(\lg n_1)$ و حذف ریشه از هیپ کمینه از مرتبه $O(\lg n_2)$ است. از طرفی می‌دانیم که $n = n_1 + n_2$ و اختلاف n_1 و n_2 حداکثر یک واحد است. پس می‌توان گفت که حذف ریشه از هر کدام از درخت‌ها از مرتبه $O(\lg(n/2))$ است که این نیز برابر با $O(\lg n)$ است.

در هنگام درج در این داده‌ساختار باید کاری کرد که اختلاف تعداد عناصر هیپ‌های کمینه و بیشینه حداکثر یک باشد. در ادامه مقداری که قرار است درج شود را a، عنصر ریشه هیپ بیشینه را x و عنصر ریشه هیپ کمینه را y می‌نامیم. a را با x و y مقایسه می‌کنیم. سه حالت ممکن به شرح زیر خواهند بود:

۱. $y \leq a$: در این صورت a باید در هیپ کمینه، که شامل عناصر بزرگتر از میانه است، درج شود. اگر تعداد عناصر هیپ کمینه، کمتر یا مساوی تعداد عناصر هیپ بیشینه باشد، عمل درج را به راحتی انجام می‌دهیم. در چنین حالتی تعداد عناصر هیپ کمینه برابر یا یکی بیشتر از عناصر هیپ بیشینه خواهد شد. اما در صورتی که تعداد عناصر هیپ کمینه یکی بیشتر از هیپ بیشینه باشد، درج عنصر جدید در هیپ کمینه سبب خواهد شد که تعداد عناصر این هیپ دو واحد بیشتر از تعداد عناصر هیپ بیشینه شود که چنین حالتی قابل قبول نخواهد بود. در نتیجه پیش از درج عنصر جدید در هیپ کمینه، ابتدا ریشه‌ی هیپ کمینه، یعنی y، را حذف کرده و مقدار حذف شده را در هیپ بیشینه درج می‌کنیم. سپس a را در هیپ کمینه درج می‌کنیم که با انجام این کار دوباره اختلاف حداکثر یک واحدی تعداد عناصر دو هیپ نسبت به یکدیگر حفظ خواهد شد.

۲. $a \leq x$: مراحل کلی عمل درج شبیه به حالت قبل است با این تفاوت که درج در هیپ بیشینه انجام خواهد شد.

۳. $y \leq a \leq x$: در این حالت تعداد عناصر هیپ‌های بیشینه و کمینه را با یکدیگر مقایسه می‌کنیم. اگر تعداد عناصر هر دو هیپ برابر بود، می‌توان عمل درج را در هر یک از هیپ‌ها انجام داد. اما اگر تعداد عناصر دو هیپ یکسان نبود، درج را در هیپ با تعداد عناصر کمتر انجام می‌دهیم.

با توجه به توضیحات ارائه شده، برای درج یک عنصر در صف اولویت میانه در بدترین حالت باید یک عمل حذف و دو عمل درج در هیپ انجام داد. در نتیجه مرتبه زمانی عمل درج در بدترین حالت برابر با $O(۳ \times \lg(n/۲))$ است که این نیز از مرتبه‌ی $O(\lg n)$ است.

◄ **سوال ۱۸.** درستی عبارت زیر را اثبات کنید.

«تعداد گره‌های با ارتفاع h در یک درخت هیپ n عنصری، حداکثر برابر با $\lceil n/۲^{h+۱} \rceil$ است»

◃ **پاسخ سوال ۱۸.**

اثبات را با استقرا بر روی h انجام می‌دهیم.

پایه‌ی استقرا: باید نشان دهیم هنگامی که $h = ۰$ است تعداد گره‌های با ارتفاع h، کمتر یا مساوی $\lceil n/۲^{h+۱} \rceil$ است. به عبارت دیگر باید نشان دهیم تعداد گره‌های با ارتفاع $h = ۰$ کوچکتر یا مساوی $\lceil n/۲ \rceil$ است.

اگر فرض کنیم درخت هیپ دارای عمق D است آنگاه گره‌های با ارتفاع صفر، که همان گره‌های برگ هستند، می‌توانند در عمق D یا $D - ۱$ باشند و هر یک از این گره‌ها در یکی از دو حالت زیر صدق می‌کنند:

- گره در عمق D قرار دارد.
- گره در عمق $D - ۱$ قرار دارد و این گره پدر هیچ گره‌ای در عمق D نیست.

حال فرض می‌کنیم x تعداد گره‌های موجود در عمق D باشد. اگر n را تعداد کل گره‌های درخت در نظر بگیریم آنگاه $n - x$ عددی فرد است زیرا $n - x$ گره‌ی موجود، تشکیل یک درخت دودویی با حداکثر تعداد گره را می‌دهند. به عبارت دیگر با نادیده گرفتن x گره‌ی موجود در عمق D دارای درختی کامل با $n - x$ گره خواهیم بود. به این ترتیب می‌توان تعداد گره‌های درخت را برابر با $n = (۲^D - ۱) + x$ در نظر گرفت. با توجه به فرد بودن مقدار $n - x$، اگر n فرد باشد آنگاه x زوج است و اگر n زوج باشد x فرد است.

برای اثبات پایه‌ی استقرا باید دو حالت زوج و فرد بودن n را به صورت جداگانه درنظر بگیریم. برای ادامه‌ی اثبات توجه داشته باشید که در عمق d به شرطی که $d < D$ باشد دارای ۲^d گره هستیم زیرا درخت در چنین عمقی دارای حداکثر تعداد گره است.

اگر n فرد باشد آنگاه x زوج است. چون x زوج است پس دقیقاً $x/۲$ گره در عمق $D - ۱$ قرار دارند که والدین x گره‌ی موجود در عمق D هستند. این بدین معنی است که تعداد گره‌های عمق $D - ۱$ که پدر هیچ گره‌ای در عمق D نیستند برابر است با:

$$۲^{D-۱} - \frac{x}{۲} \tag{۱۴.۵}$$

این یعنی علاوه بر x گره‌ی برگ موجود در عمق D دارای $۲^{D-۱} - x/۲$ گره برگ در عمق $D - ۱$ نیز هستیم. پس تعداد کل گره‌های برگ برابر خواهد بود با $x+(۲^{D-۱}-x/۲)$ که این مقدار معادل با $۲^{D-۱}+x/۲$ است. این مقدار را می‌توان به صورت $(۲^D + x)/۲$ نیز نوشت. با توجه به اینکه x زوج است مقدار $(۲^D + x)/۲$ معادل با $\lceil(۲^D + x - ۱)/۲\rceil$ است. حال اگر به جای عبارت $۲^D + x - ۱$ مقدار معادل آن یعنی n را جایگزین کنیم آنگاه عبارت $\lceil(۲^D + x - ۱)/۲\rceil$ تبدیل به $\lceil n/۲\rceil$ می‌شود. بدین ترتیب نشان دادیم که وقتی n فرد است تعداد گره‌های با عمق صفر برابر با $\lceil n/۲\rceil$ است.

حال اگر فرض کنیم n زوج است آنگاه x فرد خواهد بود و با استدلالی شبیه به استدلال انجام شده در حالت فرد بودن n، تعداد گره‌های با عمق صفر به صورت زیر به دست می‌آید:

$$\begin{aligned} n_۰ &= x + \left(۲^{D-۱} + \frac{x+۱}{۲}\right) \\ &= ۲^{D-۱} + \frac{x-۱}{۲} \\ &= \frac{۲^{D-۱} - ۱ + x}{۲} \\ &= \frac{n}{۲} \end{aligned} \tag{۱۵.۵}$$

چون n زوج است در نتیجه می‌توان تساوی (۱۵.۵) را معادل با $n_۰ = \lceil n/۲\rceil$ در نظر گرفت. به این ترتیب نشان دادیم وقتی n زوج باشد تعداد گره‌های با ارتفاع صفر برابر با $\lceil n/۲\rceil$ است.

با توجه به اینکه هم برای n فرد و هم برای n زوج نشان دادیم که تعداد گره‌های برگ در ارتفاع صفر حداکثر برابر با $\lceil n/۲\rceil$ است در نتیجه می‌توان گفت پایه‌ی استقرا برقرار است.

فرض استقرا: فرض می‌کنیم در یک درخت هیپ حداکثر تعداد گره‌های برگ با ارتفاع $h - ۱$ برابر با $\lceil n/۲^h\rceil$ است.

حکم استقرا: نشان می‌دهیم در یک درخت هیپ حداکثر تعداد گره‌های برگ با ارتفاع h برابر با $\lceil n/۲^{h+۱}\rceil$ است.

فرض می‌کنیم n_h تعداد گره‌های برگ با عمق h در یک درخت هیپ n عنصری به نام T باشد. T' را درختی در نظر می‌گیریم که با حذف برگ‌های درخت T حاصل شده است. به این ترتیب تعداد گره‌های درخت T'، که آن را با n' نشان می‌دهیم، برابر است با $n' = n - n_۰$. با توجه به اثباتی که برای پایه‌ی استقرا انجام دادیم می‌دانیم که $n_۰ = \lceil n/۲\rceil$ است. پس مقدار n' را می‌توان به صورت زیر نوشت:

$$n' = n - n_۰ = n - \left\lceil \frac{n}{۲} \right\rceil = \left\lfloor \frac{n}{۲} \right\rfloor$$

توجه به این نکته ضروری است که اگر ارتفاع گره‌ای در درخت T برابر با h باشد آنگاه چنین گره‌ای در درخت T' دارای ارتفاع $h - ۱$ است. با توجه به این نکته و تعریف $n'_{h-۱}$ به عنوان تعداد گره‌های با ارتفاع $h - ۱$

در درخت T' می‌توان گفت $n_h = n'_{h-۱}$. حال با استفاده از فرض استقرا داریم:

$$n_h = n'_{h-۱} \leq \left\lceil \frac{n'}{۲^h} \right\rceil = \left\lceil \frac{\lfloor n/۲ \rfloor}{۲^h} \right\rceil \leq \left\lceil \frac{n/۲}{۲^h} \right\rceil = \left\lceil \frac{n}{۲^{h+۱}} \right\rceil$$

بدین ترتیب اثبات کامل است.

کتاب‌نامه

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd ed. , 2009.

[۲] محسن ابراهیمی مقدم، آذرخش کی‌پور و حسین عبدی. *ساختمان‌داده‌ها و الگوریتم‌ها*. انتشارات نصیر، ویرایش اول، ۱۳۹۲.

[3] E. Horowitz, S. Sahni, and S. Anderson-Freed. *Fundamentals of Data Structures in C*. Silicon Press, 2nd ed. , 2007.

[4] M. A. Weiss. *Data Structures and Algorithm Analysis in Java*. Pearson Education, 3rd ed. , 2012.