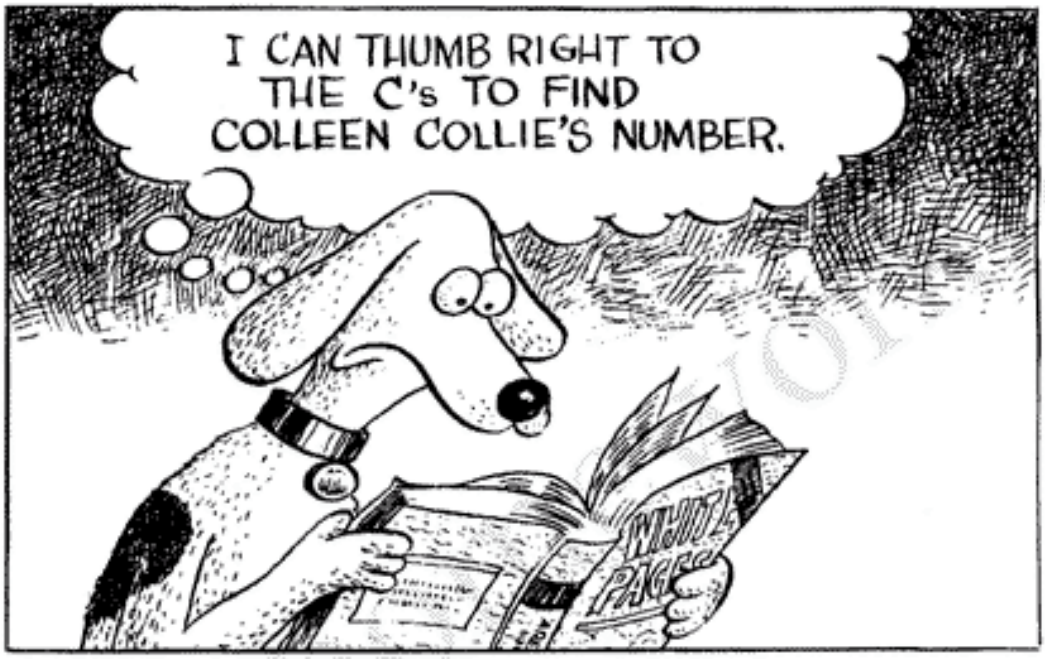
فهرست

فصل ۱

الگوریتم‌ها: کارایی، تجزیه و تحلیل، و ترتیب



این کتاب دربارهٔ تکنیک‌هایی برای حل مسائل به کمک کامپیوتر است. تنها به وسیلهٔ تکنیک نمی‌توانیم به یک سبک برنامه‌سازی یا یک زبان برنامه‌نویسی معنا ببخشیم؛ بلکه بایستی روشهای حل مسئله را نیز ارائه کنیم. به عنوان مثال، فرض کنید Barney می‌خواهد نام Collen را در دفترچه تلفن پیدا کند. یک روش این است که وی همه اسامی را به ترتیب حروف الفبا کنترل کند، یعنی از اولین اسم شروع کرده تا در نهایت Collen را پیدا کند. پرواضح است که هیچکس برای جستجوی یک اسم از چنین روشی استفاده نمی‌کند. Barney از این حقیقت استفاده می‌کند که نامها در دفترچه تلفن طبقه‌بندی هستند. لذا دفترچه را از جایی باز می‌کند که فکر می‌کند می‌تواند نامهایی که با C شروع شده‌اند را پیدا کند. اگر او زیاد به جلو برود، کمی به سمت عقب ورق می‌زند و آنقدر به عقب و جلو ورق می‌زند تا به صفحه‌ای که شامل اسم مورد نظرش است، برسد. شاید شما روش اول را جستجوی ترتیبی و روش دوم را جستجوی دودویی بدانید. به هر حال، ما این دو روش را در بخش ۱-۲ به تفصیل بیان خواهیم کرد.

در فصلهای ۲ الی ۶، تکنیک‌های مختلف حل مسئله و بکارگیری این تکنیک‌ها برای حل مسائل گوناگون مورد بحث و بررسی قرار می‌گیرد. بکارگیری یک تکنیک برای حل یک مسئله، از روش حل

قدم به قدم مسئله ناشی می‌شود. به این روش حل قدم به قدم مسئله، الگوریتم مسئله گفته می‌شود. هدف از مطالعه این تکنیکها و کاربردهایشان این است که هنگام روبرو شدن با یک مسئله جدید، بتوانیم با استفاده از مجموعهٔ تکنیک‌ها، راههای مختلف حل مسئله را ارائه و بررسی نمائیم. اغلب برای حل یک مسئله روشهای مختلفی قابل ارائه است اما آن روشی مورد نظر است که الگوریتمی سریعتر از دیگر الگوریتم‌ها دارد. بنابراین، پس از تعیین توانایی روش ارائه شده در حل مسئله بایستی کارایی الگوریتم حاصل را از نظر زمان و منبع ذخیره‌سازی مورد بررسی قرار دهیم. در تحلیل کارایی یک الگوریتم به هنگام اجرا روی کامپیوتر، زمان *(Time)* به چرخه‌های CPU و منبع ذخیره‌سازی *(Storage)* به حافظه اطلاق می‌گردد.

در این فصل، ابتدا به برخی از مفاهیم بنیادین الگوریتم‌ها پرداخته و در ادامه کارائی الگوریتم‌ها را از لحاظ زمان و حافظه مورد بررسی قرار دهیم.

۱-۱ الگوریتم‌ها

تا بحال از کلماتی نظیر " مسئله "، "راه حل" و "الگوریتم" صحبت کردیم و البته تا حدودی با این کلمات آشنا هستیم؛ اما در عین حال، سعی ما بر این است که به منظور ایجاد یک زیربنای صحیح و اصولی برای بحث، تعریف دقیق و جامعی از این کلمات ارائه دهیم. یک برنامهٔ کامپیوتری، از واحدهای منحصر بفردی تشکیل شده است که این واحدهای قابل فهم توسط کامپیوتر، وظایف مشخصی نظیر جستجوی اعداد یا مرتب‌سازی داده‌ها را به انجام می‌رسانند. هدف ما در این متن، نه طراحی یک برنامهٔ کامل، که طراحی همین واحدها است. به یک وظیفه مشخص، **مسئله** گفته می‌شود. به عبارتی، یک مسئله، یک سؤال است که ما جواب آن را جستجو می‌کنیم. در مثالهای زیر، مسئله‌هایی مطرح می‌گردند:

مثال ۱-۱ لیست S شامل n عدد را به صورت غیرنزولی مرتب کنید.
جواب، اعداد لیست S است که به صورت یک رشته مرتب در آمده است.

با ساختار داده‌ای **لیست**، می‌توانیم مجموعه‌ای از اعداد که با توالی خاصی مرتب شده‌اند را معرفی کنیم. برای مثال $S = [۱۰، ۷، ۱۱، ۵، ۱۳، ۸]$، یک لیست از شش عنصر است که اولین عنصر آن عدد ۱۰، دومین عنصر عدد ۷ و ... می‌باشد. در اینجا از اصطلاح "غیرنزولی" به جای صعودی استفاده کردیم؛ چرا که ممکن است در یک لیست، عناصر تکراری وجود داشته باشد و در اینصورت لفظ صعودی درست نخواهد بود.

مثال ۱-۲ تعیین کنید که آیا عدد x در لیست n عنصری S وجود دارد یا خیر؟
اگر x در لیست وجود داشت، جواب "بله" و در غیر اینصورت، جواب "خیر" است.

ممکن است یک مسئله، شامل متغیرهایی باشد که مقادیر مشخصی به آنها نسبت داده نشده است. به این متغیرها، **پارامترهای مسئله** گفته می‌شود. در مثال ۱-۱ دو پارامتر وجود دارد: یکی S (لیست) و دیگری n (تعداد عناصر لیست)، و در مثال ۱-۲ سه پارامتر وجود دارد: S، n و عدد x البته در این مثال وجود پارامتر n ضروری نیست؛ چراکه خود لیست S، عدد n را مشخص می‌کند. به هر حال، با وجود عدد n به عنوان یک پارامتر، تشریح مسئله آسانتر می‌شود.

وجود پارامترها در یک مسئله موجب خواهد شد که ما نه با یک مسئله، بلکه با دسته‌ای از مسائل روبرو باشیم. هر انتساب مشخص مقادیر به پارامترها، یک **نمونه** از مسئله نامیده می‌شود و یک **راه حل** برای یک نمونه از مسئله، جوابی است به سؤالی که در آن نمونه مشخص مطرح شده است.

مثال ۱-۳ یک نمونه ازمسئله مطرح شده در مثال ۱-۱ چنین است :

$$n = ۶ \quad و \quad S = [۱۰، ۷، ۱۱، ۵، ۱۳، ۸]$$

جواب این نمونه مسئله، $S = [۵، ۷، ۸، ۱۰، ۱۱، ۱۳]$ می‌باشد.

مثال ۱-۴ در یک نمونه از مسئله مطرح شده در مثال ۱-۲ داریم :

$$x = ۵ \quad ، \quad n = ۶ \quad و \quad S = [۱۰، ۷، ۱۱، ۵، ۱۳، ۸]$$

که جواب این نمونه مسئله چنین است: "بله، x در S وجود دارد."

ما می‌توانیم با کمی دقّت در بررسی لیست S، رشته مرتب شده‌ای را به عنوان جواب نمونه مثال ۱-۳ تولید کنیم. این امر امکانپذیر است، چراکه S، یک لیست بسیار کوچک است و ذهن ما می‌تواند به سرعت عمل پویش اعداد و جایگزینی درست آنها را انجام دهد. اما در عین حال، قادر نیستیم مراحل مختلف بدست آوردن جواب را تشریح کنیم. ولی اگر نمونه مسئله به جای ۵ عدد، شامل ۱۰۰۰ عدد برای لیست S بود، آنگاه نه ذهن ما قادر بود از این روش به جواب دست یابد و نه می‌توانست چنین روشی را به صورت یک برنامهٔ کامپیوتری ارائه دهد. برای ایجاد یک برنامه کامپیوتری که قادر باشد همه نمونه‌های یک مسئله را حل کند، می‌بایست یک روال قدم به قدم کلی برای تولید جواب هر نمونه مشخص کنیم. این روال قدم به قدم، **الگوریتم** نامیده می‌شود و می‌گوئیم: "الگوریتم روندی است که مسائل را حل می‌کند."

مثال ۱-۵ یک الگوریتم برای مثال ۱-۲ می‌تواند به صورت زیر باشد:

جستجو را با اولین عنصر S شروع کن. x را به ترتیب با هر یک از عناصر S مقایسه کن تا اینکه x پیدا شود یا S بطور کامل بررسی شود. اگر x پیدا شد، جواب "بله" و در غیر اینصورت، جواب "خیر" است.

ما می‌توانیم هر الگوریتمی را به زبان انگلیسی(فارسی) بنویسیم (نظیر آنچه در مثال ۱-۵ دیدیم)؛ اما در نوشتن الگوریتم به این روش، دو مشکل وجود دارد: اول اینکه، نوشتن یک الگوریتم پیچیده به این

روش بسیار مشکل است. حتی اگر این عمل انجام هم شود، کاربر به سختی می‌تواند این الگوریتم را بفهمد و دوم آنکه، روشن نیست که چگونه می‌توان یک برنامهٔ کامپیوتری را از یک الگوریتم ارائه شده به زبان محاوره‌ای (مثلاً انگلیسی یا فارسی) تولید نمود.

به دلیل اینکه C++، یک زبان آشنا و رایج بین دانشجویان است؛ لذا از آن، به عنوان یک **شبه‌کد** برای بیان الگوریتم‌ها استفاده خواهیم کرد. هر کسی که در زمینه برنامه‌نویسی در زبانهای شبه ALGOL مانند C، پاسکال، یا جاوا اندک تجربه‌ای داشته باشد، با شبه‌کد پیشنهادی ما مشکلی نخواهد داشت. برای شروع، شبه‌کدی برای الگوریتمی که کلیهٔ نمونه‌های مسئله مثال ۲-۱ را حل می‌کند، ارائه می‌دهیم. در حالت کلی، هدف از ذکر مثالهای فوق این است که عناصری را در یک مجموعهٔ مورد نظر جستجو یا مرتب نمائیم. اغلب، هر عنصر منحصراً یک رکورد را می‌شناساند و به همین دلیل به عنصر، **کلید** هم گفته می‌شود. برای مثال، یک رکورد می‌تواند شامل اطلاعات خصوصی یک شخص باشد که در آن، عدد امنیت داده‌ای شخص به عنوان کلید معرفی شده است. ما الگوریتم‌های جستجو و مرتب‌سازی را با تعریف نوع داده‌ای **keytype** برای عناصر ارائه می‌دهیم و این بدین معناست که عناصر می‌توانند از هر مجموعهٔ مورد نظر و دلخواهی انتخاب گردند.

در الگوریتم زیر، لیست S با ساختار داده‌ای آرایه معرفی شده است و به جای جواب "بله" یا "خیر"، محل عنصر x در آرایه (در صورت وجود) و در غیر اینصورت، عدد صفر بازگردانده می‌شود.

الگوریتم ۱-۱ جستجوی ترتیبی (Sequential Search)

مسئله: آیا عدد x در آرایهٔ n کلیدی S وجود دارد؟

ورودی (پارامتر): عدد صحیح مثبت n، آرایه‌ای از کلیدها (S) با شاخصهایی از ۱ تا n، و یک کلید.

خروجی: مکان x در آرایهٔ S (در صورت عدم وجود، عدد صفر).

```
void seqsearch (int n,
                const keytype S[ ],
                keytype x,
                index& location)
{
   location = 1;
   While (location<=n && S[location]!=x)
      location ++;
   if (location > n)
      location = 0;
}
```

یک نکتهٔ جالب توجه در مثال فوق، بکارگیری آرایه‌ها است. C++ فقط به آرایه‌ها امکان می‌دهد با اعداد صحیح از صفر تا n شاخص‌دهی شوند. اغلب، الگوریتم‌ها را با بکارگیری آرایه‌های شاخص‌دهی شده

در محدوده‌های دلخواهی از اعداد صحیح تشریح می‌کنند و گاهی‌اوقات، آنها با شاخصهایی که عدد صحیح نیستند، تبیین می‌شوند. محدودهٔ شاخصها برای الگوریتم‌ها، در مشخصات **ورودی و خروجی** ذکر می‌گردند. برای مثال، آرایهٔ S در الگوریتم ۱-۱ از ۱ تا n شاخص‌دهی شده است. در واقع، برای شمارش عناصر یک لیست، از آرایه‌ای با شاخصهای ۱ تا n استفاده کردیم که این بهترین محدودهٔ شاخص‌دهی برای بکارگیری یک لیست است. البته، این الگوریتم خاص می‌تواند با تعریف عبارت زیر در C++ نیز بکار گرفته شود:

```
keytype S[n+1];
```

از اینجا به بعد، دیگر از بکارگیری الگوریتمها در یک زبان برنامه‌نویسی خاص صحبت نمی‌کنیم و بحث را به ارائه الگوریتم‌هایی می‌کشانیم که به سرعت قابل درک، تجزیه و تحلیل هستند.

ذکر دو نکته در اینجا ضروری است ؛ اول اینکه، این امکان وجود دارد که آرایه‌های دوبعدی با طول متغیر، به عنوان پارامترهای یک روال تعریف شوند (نگاه کنید به الگوریتم ۴-۱) و دوم آنکه، می‌توان در الگوریتم‌ها از آرایه‌های محلی با طول متغیر، استفاده نمود. برای مثال، اگر n یک پارامتر برای روال example بوده و ما به یک آرایه با شاخصهای ۲ تا n نیازمند باشیم، چنین تعریف می‌کنیم:

```
void example (int n)
{
    keytype S[2..n];
    .
    .
    .
}
```

توجه داریم که نماد S[2..n] ، به معنای آرایهٔ S با شاخصهای ۲ تا n از مفاهیم شبه‌کد است نه بخشی از زبان C++. هرگاه بتوانیم مراحل الگوریتم را با استفاده از عبارات ریاضی یا عبارات شبه انگلیسی، صریح‌تر و مفیدتر از دستورات C++ بیان کنیم، می‌بایست این عمل انجام شود. به عنوان مثال، فرض کنید برخی از دستورات، در صورتی اجرا می‌شوند که یک متغیر x بین مقادیر low و high باشد؛ می‌نویسیم:

```
if (low <= x &&  x <= high){
    .
    .
    .
}
```

بهتر است از ←

```
if(low ≤ x ≤ high){
    .
    .
    .
}
```

و فرض کنید که می‌خواهیم متغیر x ، مقدار y و متغیر y ، مقدار x را بگیرد؛ لذا می‌نویسیم:

```
temp = x;
x = y;
y = temp;
```

بهتر است از ←

```
exchange x and y;
```

علاوه بر نوع داده keytype، از سه نوع داده‌ای زیر نیز به عنوان نوع تعریف شده در شبه‌کدها (نه C++) استفاده می‌شود:

نوع داده	کاربرد
index	یک متغیر از نوع صحیح که به عنوان شاخص بکار می‌رود.
number	یک متغیر که می‌تواند بصورت صحیح یا حقیقی تعریف شود.
bool	یک متغیر که می‌تواند مقادیر "true" یا "false" را به خود بگیرد.

* از نوع داده‌ای **number** زمانی استفاده می‌کنیم که صحیح یا اعشاری بودن اعداد برای الگوریتم اهمیتی نداشته باشد.
* برخی اوقات، از ساختار کنترلی غیراستاندارد زیر استفاده می‌کنیم:

```
repeat (n times){
    .
    .
    .
}
```

این بدین معناست که کد برنامه به تعداد n مرتبه تکرار می‌شود. اما برای انجام این کار در C++، حتماً بایستی یک متغیر کنترلی تعریف نموده و یک حلقهٔ **for** بنویسیم.

هنگامی که در معرفی یک الگوریتم مشخص شود که می‌بایست مقداری توسط آن برگردانده شود، آن الگوریتم را به صورت یک تابع می‌نویسیم. درغیراینصورت، آن را به عنوان یک روال معرفی کرده و از پارامترهای ارجاعی، برای بازگردندن مقادیر استفاده می‌کنیم. اگر پارامتر ارجاعی یک آرایه نباشد، آن را با علامت & در انتهای نام نوع داده‌ای معرفی می‌کنیم. آرایه‌ها در C++، به طور پیش‌فرض، به صورت پارامترهای ارجاعی تعریف شده‌اند و دیگر نیازی به نماد & در تعریف این ساختار داده‌ای نیست. برای تغییر این پیش فرض در C++، از کلمه **const** استفاده می‌کنیم. به عبارت دیگر، با استفاده از کلمه **const**، آرایه را طوری به عنوان پارامتر معرفی می‌کنیم که دیگر نتواند مقداری را توسط الگوریتم برگشت دهد.

بطور کلی، سعی می‌کنیم تا حد امکان از خصوصیات C++ اجتناب کنیم. بنابراین، هرکسی که با یکی از زبانهای سطح بالا آشنایی داشته باشد، می‌تواند از این شبه‌کدها استفاده نماید. اگر شما با C++ آشنا نیستید، ممکن است به دنبال عملگرهای منطقی و مقایسه‌ای آن باشید. نیازی نیست؛ چرا که بخشی از آن را در زیر آورده‌ایم:

عملگر	نماد C++
AND	&&
OR	||
NOT	!

عبارت مقایسه‌ای	کد C++
$x = y$	x == y
$x \neq y$	x != y
$x \leq y$	x <= y
$x \geq y$	x >= y

به مثالهای زیر توجه کنید. در اولین مثال، کاربرد یک تابع نشان داده شده است. توجه داریم که قبل از نام روال‌ها در شبه‌کد، از کلمهٔ **void** استفاده می‌شود؛ در حالیکه قبل از نام توابع، از نوع داده‌ایی استفاده می‌شود که مقدار بازگشتی تابع - توسط دستور **return** - از آن نوع می‌باشد.

الگوریتم ۲-۱ جمع عناصر یک آرایه

مسئله: کلیه عناصر آرایهٔ n عنصری S را با هم جمع کنید.

ورودی: عدد صحیح مثبت n، آرایهٔ S با شاخصهایی از ۱ تا n

خروجی: sum، حاصل جمع عناصر آرایهٔ S

```
number sum (int n, const number S[ ])
{
   index i;
   number result;
   result = 0;
   for (i = 1; i <= n; i++)
      result = result + S[i];
   return result;
}
```

ما در این کتاب با بیشتر الگوریتم‌های مرتب‌سازی آشنا می‌شویم. به یک نوع آن توجه کنید.

الگوریتم ۳-۱ مرتب‌سازی تبادلی (حبابی یا Exchange Sort)

مسئله: n کلید را به صورت غیرنزولی مرتب کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S با شاخصهایی از ۱ تا n

خروجی: آرایهٔ S شامل کلیدهای مرتب شده به صورت غیرنزولی.

```
void exchangesort (int n, keytype S[ ])
{
   index i, j;
   for (i = 1; i <= n - 1; i++)
      for (j = i + 1; j <= n; j++)
         if (S[j] < S[i])
            exchange S[j] and S[i];
}
```

دستور exchange S[j] and S[i]، مقادیر S[i] و S[j] را با هم عوض می‌کند؛ یعنی S[i] مقدار S[j] و S[j] مقدار S[i] را به خود می‌گیرد. این دستور شباهتی با دستورات C++ ندارد و همانطوریکه قبلاً گفته شد، اگر بتوانیم عملیاتی را به سادگی و بدون استفاده از دستورات C++ تعریف کنیم، لازم است که این کار را انجام دهیم. در مرتب‌سازی تبادلی، عنصر iام آرایه با عناصر i+۱ام تا nام آرایه مقایسه می‌شود. هرگاه عددی که با عنصر i مقایسه شده، کوچکتر از آن باشد، دو عدد با هم جابجا (مبادله) می‌شوند. در اولین اجرای حلقه i کوچکترین عنصر به بالای آرایه هدایت می‌شود. در دومین اجرا، کوچکترین عنصر بعدی به بالا فرستاده می‌شود و به همین ترتیب، این روند برای n-۱ عنصر باقی‌مانده تکرار می‌شود تا اینکه کل لیست، مرتب شده و بزرگترین مقدار در انتهای آن قرار بگیرد.

الگوریتم بعدی، ضرب ماتریس‌ها است. فرض کنید که دو ماتریس ۲ × ۲ به نامهای A و B داریم که

$$A=\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \text{و} \quad B=\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

حاصل ضرب C = A × B براساس قاعده زیر محاسبه می‌شود:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j}$$

برای مثال،

$$\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} \times \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 2\times5+3\times6 & 2\times7+3\times8 \\ 4\times5+1\times6 & 4\times7+1\times8 \end{bmatrix} = \begin{bmatrix} 28 & 38 \\ 26 & 36 \end{bmatrix}$$

در حالت کلی، اگر دو ماتریس n × n به نامهای A و B داشته باشیم، حاصل ضرب C عبارت است از

$$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj} \qquad \text{برای } 1 \le i, j \le n$$

الگوریتم ۴-۱ ضرب ماتریس‌ها

مسئله: حاصلضرب ماتریس n×n را تعیین کنید.

ورودی: یک عدد صحیح مثبت n، آرایه‌های دوبعدی A و B که سطرها و ستونهای هر دو آرایه از ۱ تا n شاخص‌دهی شده است.

خروجی: آرایهٔ دوبعدی C که سطرها و ستونهای آن از ۱ تا n شاخص‌دهی شده است.

```
void matrixmult (int n,
                 const number A[ ][ ],
                 const number B[ ][ ],
                 number C[ ][ ])
{
  index i, j, k;
  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++){
      C[i][j] = 0;
      for (k = 1; k <= n; k++)
        C[i][j] = C[i][j] + A[i][k] * B[k][j];
  }
}
```

۱-۲ اهمیت توسعهٔ الگوریتم‌های کارا

قبلاً گفته شد که کارایی الگوریتم‌ها - بدون توجه به سریع شدن کامپیوترها- همواره به صورت یک اصل مهم باقی خواهد ماند. اکنون با مقایسه دو الگوریتم برای یک مسئلهٔ خاص، اهمیت این موضوع را بررسی خواهیم کرد.

۱-۲-۱ جستجوی ترتیبی در مقایسه با جستجوی دودویی

در اوایل بحث، اشاره‌ای داشتیم به این نکته که جستجوی یک اسم در دفترچهٔ تلفن، یک جستجوی دودویی تغییریافته است که معمولاً بسیار سریعتر از جستجوی ترتیبی به نتیجه می‌رسد. برای اثبات این مطلب، با مقایسهٔ دو الگوریتم فوق، چگونگی این سرعت عمل را بررسی می‌کنیم.

الگوریتم جستجوی ترتیبی، با عنوان الگوریتم ۱-۱ نوشته شده است. الگوریتمی که جستجوی دودویی را بر روی یک آرایهٔ مرتب غیرنزولی انجام می‌دهد، شباهت زیادی با ورق زدن - به جلو و عقب - یک دفترچه تلفن دارد. فرض کنید عدد x را در آرایهٔ مورد نظر جستجو می‌کنیم. در ابتدا الگوریتم، عنصر x را با عنصر میانی آرایه مقایسه می‌کند. اگر آنها مساوی بودند، الگوریتم به پایان رسیده است و اگر x کوچکتر از عنصر میانی آرایه بود، می‌گوییم که x در صورت وجود بایستی در نیمهٔ اول آرایه باشد. لذا الگوریتم، روند جستجو را برای نیمه اول تکرار می‌کند. (اگر x برابر با عنصر میانی نیمه اول بود، الگوریتم به انجام رسیده است و الی آخر). اما اگر x از عنصر میانی آرایه بزرگتر بود، جستجوی روی نیمه دوم آرایه تکرار می‌گردد. جستجو آنقدر ادامه می‌یابد تا x در آرایه پیدا شود یا مشخص گردد که x در آرایه وجود ندارد. یک الگوریتم برای این روش را در زیر آورده‌ایم:

الگوریتم ۵-۱ جستجوی دودویی

مسئله: تعیین کنید که آیا عدد x در آرایهٔ n کلیدی S وجود دارد یا خیر.

ورودی: یک عدد صحیح مثبت n، آرایه‌ای مرتب (بصورت غیرنزولی) از کلیدها S با شاخصهایی از ۱ تا n، و یک کلید x

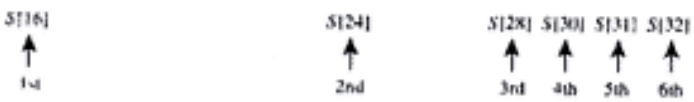
خروجی: مکان کلید x در آرایهٔ S (صفر، اگر x در آرایهٔ S نباشد).

```
void binsearch(int n,
               const keytype S[ ],
               keytype x,
               index& loccation)
{
    index low, high, mid;
    low = 1; high = n;
    location = 0;
```

```
    while (low <= high && location == 0){
        mid = ⌊(low + high) / ۲⌋;
        if (x == S[mid]);
            location = mid;
        else if (x < S[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
}
```

به منظور مقایسهٔ دو الگوریتم جستجوی ترتیبی و دودویی، بایستی تعداد مقایسه‌های انجام شده توسط هر یک از الگوریتم‌ها تعیین شود. اگر آرایهٔ S شامل ۳۲ عنصر بوده و عدد x در آرایه موجود نباشد، الگوریتم ۱-۱ (جستجوی ترتیبی) قبل از اینکه مشخص کند x در آرایه وجود ندارد، x را با تمام ۳۲ عنصر آرایه مقایسه می‌کند. در حالت کلی، جستجوی ترتیبی برای تعیین اینکه عدد x در آرایهٔ n عنصری S وجود ندارد(بدترین حالت)، تعداد n مقایسه انجام می‌دهد. پرواضح است که این بیشترین تعداد مقایسهٔ جستجوی ترتیبی در یک آرایهٔ n عنصری است. اگر x در آرایه موجود باشد، باز هم تعداد مقایسه‌ها از n بزرگتر نخواهد بود.

الگوریتم ۵-۱ (جستجوی دودویی) را در نظر بگیرید. در هر حلقهٔ **while**، دو مرتبه عدد x با S[mid] مقایسه می‌شود (مگر زمانی که x پیدا شود). در یک اجرای کارا از الگوریتم توسط اسمبلر زبان، x در هر گذر از حلقهٔ **while**، تنها یک مرتبه با S[mid] مقایسه می‌شود. نتیجهٔ این مقایسه، کد شرط را مقداردهی نموده و شاخهٔ مناسب شرط، براساس کد شرط تعیین می‌گردد. فرض می‌کنیم که الگوریتم جستجوی دودویی، از این روش استفاده می‌کند. با این فرض، طبق آنچه که در شکل ۱-۱ آمده است، الگوریتم برای جستجوی عدد x که از همهٔ عناصر آرایهٔ ۳۲ عنصری مورد نظر بزرگتر است، بایستی تنها شش عمل مقایسه انجام دهد. توجه دارید که ۶ = ۱ + ۳۲ lg (منظور از lg همان $\log_2$ است). حتماً قانع شده‌اید که این بیشترین تعداد مقایسه‌ها است که با روش دودویی انجام می‌شود و البته این در صورتی است که عنصر x در آرایه موجود نباشد. در صورتی که x در بین عناصر آرایه باشد، باز هم تعداد مقایسه‌ها بیشتر از عدد بدست آمده نخواهد بود. فرض کنید که تعداد عناصر آرایه را به دو برابر یعنی ۶۴ عنصر افزایش می‌دهیم؛ در اینصورت، جستجوی دودویی تنها یک مقایسه بیش از حالت قبل انجام می‌دهد و آن هم در



شکل ۱-۱ عناصر آرایه با اندازه ورودی ۳۲، که جستجوی دودویی برای یافتن x بزرگتر از عناصر آرایه با آنها مقایسه می‌شود. عناصر به ترتیبی که مقایسه می‌شوند، شماره‌گذاری شده‌اند.

جدول ۱-۱ تعداد مقایسات در جستجوهای ترتیبی و دودویی، وقتی که x بزرگتر از همهٔ عناصر آرایه است.

اندازهٔ آرایه	تعداد مقایسات در جستجوی ترتیبی	تعداد مقایسات در جستجوی دودویی
۱۲۸	۱۲۸	۸
۱۰۲۴	۱۰۲۴	۱۱
۱۰۴۸۵۷۶	۱۰۴۸۵۷۶	۲۱
۴۲۹۴۹۶۷۲۹۶	۴۲۹۴۹۶۷۲۹۶	۳۳

اولین مقایسه است که آرایه به دو زیرآرایه تقسیم می‌گردد. لذا در بدترین حالت جستجو، یعنی زمانیکه x در آرایه موجود نباشد، جستجوی دودویی هفت مقایسه انجام می‌دهد (۷ = lg ۶۴ + ۱). در حالت کلی، زمانی که تعداد عناصر آرایه دو برابر می‌شود، تعداد مقایسه‌ها یک مرتبه بیشتر می‌شود. بنابراین، اگر n یکی از توانهای ۲ و x عددی بزرگتر از عناصر موجود در یک آرایهٔ n عنصری باشد، تعداد مقایسه‌ها در جستجوی دودویی برابر است با $\lg n + 1$.

جدول ۱-۱، تعداد مقایسه‌های انجام شده به وسیلهٔ جستجوی دودویی و جستجوی ترتیبی را برای مقادیر مختلف n، وقتی که x بزرگتر از کلیهٔ عناصر آرایه است، نشان می‌دهد. هنگامی که آرایه شامل تقریباً ۴ میلیارد عنصر باشد (در حدود جمعیت جهان)، جستجوی دودویی با ۳۲ مقایسه در برابر جستجوی ترتیبی با حدود ۴ میلیارد مقایسه قرار می‌گیرد. حتی اگر کامپیوتر قادر به انجام یک گذر از حلقهٔ **while** در یک نانوثانیه (یک میلیاردم ثانیه) باشد، جستجوی ترتیبی برای تعیین عدم وجود x در آرایه به چهار ثانیه وقت نیاز دارد. اهمیت این اختلاف در عملیات دسترسی برخط (Online Application) یا جستجو در آرایه‌هایی با تعداد زیاد عناصر مشخص می‌شود.

در بحث فوق، تنها آرایه‌هایی را برای جستجوی دودویی درنظر گرفتیم که تعداد عناصر آنها توانی از ۲ است. در فصل ۲ به جستجوی دودویی، به عنوان یک مثال از بحث تقسیم و غلبه رجوع خواهیم کرد و در آنجا، آرایه‌هایی را برای این جستجو در نظر می‌گیریم که تعداد عناصر آنها می‌تواند هر عدد صحیح و مثبتی باشد.

۱-۲-۲ دنباله فیبوناچی

الگوریتمی که در اینجا بررسی می‌شود، محاسبهٔ عنصر nام فیبوناچی است که به صورت بازگشتی زیر تعریف شده‌است:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \qquad \text{برای } n \geq 2$$

با محاسبهٔ چند عنصر اولیهٔ این دنباله داریم:

$$f_۲ = f_۱ + f_۰ = ۱ + ۰ = ۱$$
$$f_۳ = f_۲ + f_۱ = ۱ + ۱ = ۲$$
$$f_۴ = f_۳ + f_۲ = ۲ + ۱ = ۳$$
$$f_۵ = f_۴ + f_۳ = ۳ + ۲ = ۵, \ldots$$

دنبالهٔ فیبوناچی دارای کاربردهای مختلفی در علوم ریاضیات و کامپیوتر می‌باشد و بدلیل اینکه این دنباله به صورت بازگشتی تعریف شده است، الگوریتم بازگشتی زیر نیز بر همین اساس تعریف می‌شود.

الگوریتم ۶-۱ عنصر nام فیبوناچی (بازگشتی)

مسئله: عنصر nام دنبالهٔ فیبوناچی را تعیین کنید.

ورودی: یک عدد صحیح غیرمنفی n.

خروجی: *fib*، عنصر nام دنباله فیبوناچی.

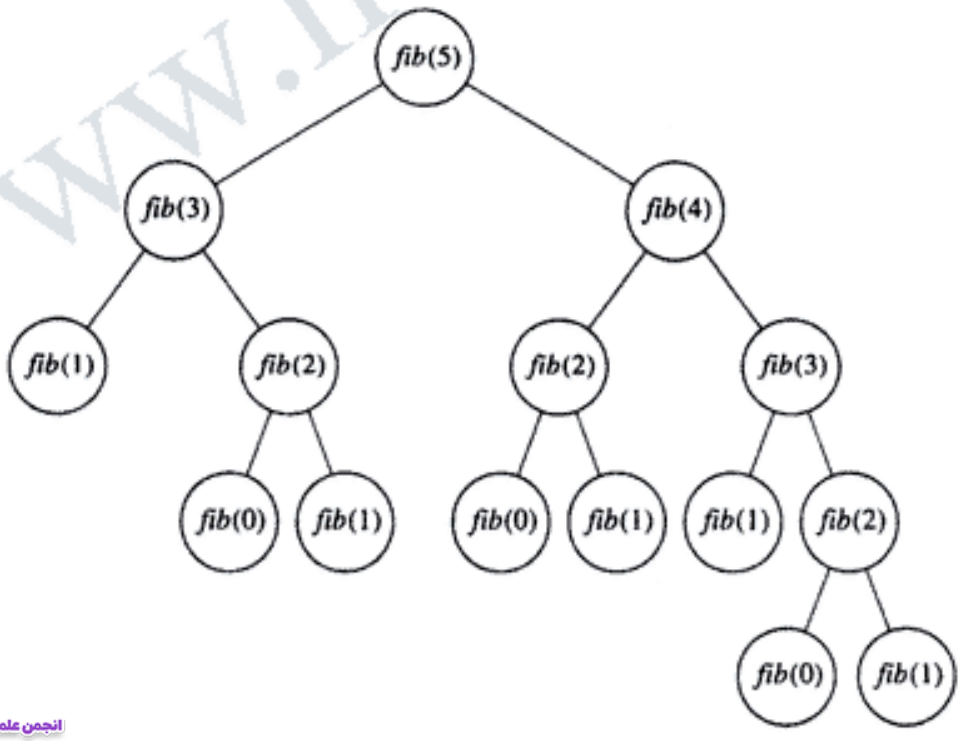
```
int fib (int n)
{
  if (n >= 1)
    retrun n;
  else
    return fib(n - 1) + fib(n - 2);
}
```

"عدد صحیح غیرمنفی" شامل کلیهٔ اعداد صحیح بزرگتر یا مساوی صفر می‌باشد، برخلاف "عدد صحیح مثبت" که فقط اعدادی را شامل می‌شود که بزرگتر از صفر هستند. با اینکه از این عبارت برای وضوح بیشتر مسئله استفاده نموده‌ایم ولی به منظور اجتناب از درهم‌ریختگی در متن، آن را به اختصار به صورت "عدد صحیح" بیان می‌کنیم.

الگوریتم بازگشتی فیبوناچی، در عین سادگی در نوشتن و درک مسئله، از کارایی بسیار پایینی برخوردار است. شکل ۲-۱ درخت بازگشتی الگوریتم را برای محاسبهٔ fib(۵) نشان می‌دهد. فرزندان یک گره در درخت، خود شامل فراخوانی‌های بازگشتی هستند. به عنوان مثال، برای محاسبهٔ fib(۵) در بالاترین سطح به fib(4) و fib(3) نیاز داریم و برای محاسبهٔ fib(۳) بایستی fib(۲) و fib(۱) محاسبه گردد و الی آخر. با کمی دقت، عدم کارایی در درخت الگوریتم مشخص می‌شود زیرا بعضی از مقادیر، مدام در حال محاسبهٔ مجدد هستند. به عنوان مثال، fib(۲) سه بار محاسبه شده است. درخت شکل ۲-۱ نشان می‌دهد که الگوریتم، به منظور تعیین fib(n) برای $۶ \le n \le ۰$ بایستی چه تعداد عنصر را محاسبه کند:

n	تعداد عناصر محاسبه شده
۰	۱
۱	۱
۲	۳
۳	۵
۴	۹
۵	۱۵
۶	۲۵

شش مقدار اول را می‌توان با شمارش گره‌های زیر درختی به ریشهٔ fib(n) برای $1 \leq n \leq 5$ مشخص کرد. تعداد عناصر برای fib(۶) برابر است با مجموع تعداد گره‌های موجود در fib(۵) و fib(۴) به اضافه یک. برخلاف تصور ما، این اعداد همانند ارزیابی جستجوی دودویی به سادگی قابل محاسبه نیستند. توجه دارید که در حالت فوق، وقتی n با عدد ۲ جمع می‌شود، تعداد عناصر درخت از دو برابر هم بیشتر می‌شود. به عنوان مثال، برای $n = ۴$، نه عنصر در درخت وجود دارد و زمانی که $n = ۶$ می‌شود، تعداد عناصر درخت به ۲۵ می‌رسد. T(n) را تعداد عناصر درخت بازگشتی برای n می‌نامیم. اگر افزودن ۲ به n، تعداد عناصر را به بیشتر از دو برابر برساند، دراینصورت می‌توان برای n، توان مثبتی از ۲، چنین داشت:



$$
\begin{aligned}
T(n) &> ۲ \times T(n-۲) \\
&> ۲ \times ۲ \times T(n-۴) \\
&> ۲ \times ۲ \times ۲ \times T(n-۶) \\
&\vdots \\
&> \underbrace{۲ \times ۲ \times ۲ \times \ldots \times ۲}_{n/۲ \text{ عناصر}} \times T(۰)
\end{aligned}
$$

از آنجائیکه $T(۰) = ۱$ است، لذا $T(n) > ۳^{n/۲}$ خواهد بود. با استفاده از استقراء می‌توان ثابت نمود که برای هر $n \geq ۲$ حتی اگر n توانی از ۲ نباشد، این نامساوی برقرار است. برای $n = ۱$ داریم $T(۱) = ۱$ که کمتر از ۲ است؛ لذا این نامساوی برای $n = ۱$ صدق نمی‌کند. مبحث استقراء را در ضمیمه A بخش A-۳ آورده‌ایم.

قضیه ۱-۱ اگر تعداد عناصر درخت بازگشتی مربوط به الگوریتم ۶-۱ را T(n) بنامیم، آنگاه برای هر $n \geq ۲$ داریم:

$$T(n) > ۳^{n/۲}$$

اثبات: اثبات از روش استقراء انجام می‌شود.

پایهٔ استقراء: ما به دو پایه برای استقراء نیازمندیم؛ چرا که در گام استقراء همواره دو حالت قبلی مدنظر می‌باشند. طبق شکل ۲-۱ برای $n = ۲$ و $n = ۳$ داریم:

$$T(۲) = ۳ > ۲ = ۲^{۲/۲}$$

$$T(۳) = ۵ > ۲/۸۳ \approx ۳^{۳/۲}$$

فرض استقراء: یک روش برای فرض استقراء این است که درنظر بگیریم برای هر $m < n$، این عبارت درست است. آنگاه در گام استقراء نشان می‌دهیم که این عبارت برای هر n هم درست می‌باشد. این روشی است که در اثبات این تئوری در نظر گرفتیم. پس فرض ما این است که برای $۲ \leq m < n$ داریم:

$$T(m) > ۳^{m/۲}$$

گام استقراء: بایستی نشان دهیم که $T(n) > ۳^{n/۲}$. مقدار T(n) برابراست با حاصل جمع $T(n-۱)$ و $T(n-۲)$ به اضافه یک (یک گره در ریشه). بنابراین،

$$
\begin{aligned}
T(n) &= T(n-۲) + T(n-۱) + ۱ \\
&> ۲^{(n-۱)/۲} + ۲^{(n-۲)/۲} + ۱ \qquad \text{(طبق فرض استقراء)} \\
&> ۲^{(n-۲)/۲} + ۲^{(n-۲)/۲} = ۲ \times ۲^{(n/۲)-۱} = ۳^{n/۲}
\end{aligned}
$$

ما ثابت نمودیم که تعداد عناصر محاسبه شده توسط الگوریتم ۶-۱ برای تعیین عنصر nام فیبوناچی از $2^{n/2}$ بزرگتر است. برای مشخص نمودن عدم کارایی الگوریتم بازگشتی به این نتیجه رجوع خواهیم کرد. اما نخست یک الگوریتم کارا برای محاسبهٔ عنصر nام دنباله فیبوناچی طرح می‌کنیم. آنچنانکه در شکل ۲-۱ مشاهده می‌کنید، برای تعیین fib(۵)، سه مرتبه fib(۲) محاسبه شده است؛ در صورتی که اگر هنگام محاسبهٔ یک مقدار، آن را در آرایه‌ای ذخیره کنیم، آنگاه لزومی به محاسبهٔ مجدد آن نیست. الگوریتم تکرار زیر از این تکنیک استفاده می‌کند.

الگوریتم ۷-۱ عنصر nام فیبوناچی (تکرار)

مسئله: عنصر nام دنباله فیبوناچی را تعیین کنید.

ورودی: یک عدد صحیح غیرمنفی n.

خروجی: *fib2*، عنصر nام دنبالهٔ فیبوناچی.

```
int fib2 (int n)
{
   index i;
   int f[0..n];
   f[0] = 0;
   if (n > 0){
     f[1] = 1;
     for (i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
   }
   return f[n];
}
```

الگوریتم ۷-۱ می‌تواند بدون استفاده از آرایه نیز نوشته شود، چراکه در هر تکرار، تنها دو عنصر انتهای دنباله مورد نیاز هستند. لیکن استفاده از آرایه، وضوح خاصی به الگوریتم می‌بخشد. این الگوریتم برای تعیین fib2(n) هر یک از n+۱ عنصر اول را فقط یک بار محاسبه می‌کند. در جدول ۲-۱، مدت زمان لازم برای محاسبهٔ فیبوناچی با استفاده از دو الگوریتم فوق و به ازاء مقادیر مختلف n با هم مقایسه شده‌اند. فرض بر این است که یک کامپیوتر فرضی، هر عنصر را در مدت زمان یک نانوثانیه محاسبه می‌کند. هنگامی که n برابر ۸۰ می‌شود، الگوریتم ۶-۱ حداقل ۱۸ دقیقه وقت می‌گیرد و زمانی که n برابر ۱۲۰ می‌شود، محاسبه بیش از ۳۶ سال طول می‌کشد که این مدت زمان برای یک انسان غیرقابل تحمل است. حتی اگر کامپیوتری ساخته شود که یک میلیارد مرتبه از کامپیوتر فرضی ما سریعتر باشد، محاسبهٔ عنصر ۲۰۰ام بیش از ۴۰۰۰۰ سال طول می‌کشد. الگوریتم ۶-۱ در یک مدت زمان غیرقابل تحملی عملیات محاسباتی‌اش را انجام می‌دهد؛ مگر اینکه n عددی کوچکتر باشد. اما از طرف دیگر، الگوریتم ۷-۱، عنصر nام فیبوناچی را در یک لحظه محاسبه می‌کند. این مقایسه می‌تواند اهمیت بررسی کارایی الگوریتم‌ها را به وضوح نشان دهد.

جدول ۲-۱ یک مقایسه بین الگوریتم ۶-۱ و الگوریتم ۷-۱.

n	$n+1$	$2^{n/2}$	Execution Time Using Algorithm 1.7	Lower Bound on Execution Time Using Algorithm 1.6
40	41	1,048,576	41 ns*	1048 μs†
60	61	1.1×10^{9}	61 ns	1 s
80	81	1.1×10^{12}	81 ns	18 min
100	101	1.1×10^{15}	101 ns	13 days
120	121	1.2×10^{18}	121 ns	36 years
160	161	1.2×10^{24}	161 ns	3.8×10^{7} years
200	201	1.3×10^{30}	201 ns	4×10^{13} years

*1 ns = 10^{-9} second.
†1 μs = 10^{-6} second.

الگوریتم ۶-۱، یک الگوریتم تقسیم و غلبه است. به خاطر دارید که تکنیک تقسیم و غلبه، یک الگوریتم بسیار کارا (الگوریتم ۵-۱: جستجوی دودویی) برای جستجو در یک آرایهٔ مرتب ارائه نموده است. آنچنانکه در فصل ۲ نشان خواهیم داد، الگوریتم تقسیم و غلبه برای برخی مسائل بسیار کارا و برای برخی دیگر، غیرقابل تحمل و بسیار ناکارا خواهد بود. الگوریتم کارای ما برای محاسبهٔ عنصر nام فیبوناچی (الگوریتم ۷-۱)، یک مثال از تکنیک برنامه‌نویسی پویا است که در فصل ۳ تشریح خواهد شد. پس همانطور که مشاهده می‌کنیم، *انتخاب* بهترین تکنیک برای نوشتن الگوریتم مسئله می‌تواند امری ضروری و بسیار اساسی باشد.

۳-۱ تحلیل الگوریتم‌ها

تعیین این نکته که یک الگوریتم با چه میزان کارایی مسئله را حل می‌کند، نیاز به تجزیه و تحلیل دارد. ما با مقایسهٔ الگوریتم‌ها در بخش قبل، بررسی کارایی الگوریتم‌ها را مطرح کردیم. به هر ترتیب، تحلیل انجام شده، یک تحلیل غیراصولی بود. اما اکنون قصد داریم درباره اصطلاحات مورد استفاده در تحلیل الگوریتم‌ها و همچنین روشهای استاندارد برای انجام این عمل بحث کنیم.

۱-۳-۱ تحلیل پیچیدگی زمانی

به منظور تجزیه و تحلیل کارایی یک الگوریتم در عنصر زمان، لزومی به تعیین دقیق تعداد چرخه‌های CPU نیست؛ چرا که چرخهٔ CPU از ویژگیهای خاص کامپیوتری است که الگوریتم بر روی آن اجرا می‌شود. همچنین نیازی به شمارش دستوراتی که بر روی سیستم اجرا می‌شوند، وجود ندارد زیرا تعداد دستورات اجراشونده، به زبان برنامه‌نویسی که الگوریتم را پیاده‌سازی کرده و به روش و مهارت برنامه‌نویس بستگی دارد. بطورکلی، در تحلیل الگوریتم ۱-۱ و الگوریتم ۵-۱ برای مقادیر مختلف n

(n تعداد عناصر آرایه است) دریافتیم که الگوریتم ۵-۱، بسیار کراتر از الگوریتم ۱-۱ است. در حالت کلی، زمان اجرای یک الگوریتم با افزایش ورودی، افزایش می‌یابد. به عبارت دیگر، زمان اجرا با تعداد دفعاتی که یک عمل مبنایی - نظیر یک دستور مقایسه- انجام می‌شود، رابطهٔ مستقیم دارد. بنابراین، کارایی الگوریتم را به عنوان تابعی از اندازه ورودی، با تعیین تعداد دفعات انجام برخی عملیات اصلی، تجزیه و تحلیل می‌کنیم. این، یکی از تکنیکهای استاندارد تحلیل سیستم است.

برای بسیاری از الگوریتم‌ها، یافتن یک واحد ورودی - موسوم به اندازه ورودی- مشکل نیست. به عنوان مثال، الگوریتم‌های ۱-۱ (جستجوی ترتیبی)، ۲-۱ (جمع عناصر آرایه)، ۳-۱ (مرتب‌سازی جایگزینی) و ۵-۱ (جستجوی دودویی) را در نظر بگیرید. تعداد عناصر آرایه در این الگوریتم‌ها، یک مقدار ساده برای ورودی است که به همین دلیل به آن اندازه ورودی می‌گوئیم. در الگوریتم ۴-۱ (ضرب ماتریسی)، تعداد سطرها و ستونهای آرایه، به عنوان اندازه ورودی معرفی شده است. در برخی از الگوریتم‌ها، مناسبتر است که دو واحد ورودی به عنوان اندازه ورودی در نظر گرفته شود. برای مثال، زمانی که ورودی الگوریتم، یک گراف باشد، می‌بایست تعداد رئوس گراف و تعداد لبه‌های آن را برای الگوریتم مشخص کنیم. لذا اندازه ورودی شامل هر دو پارامتر می‌باشد.

گاهی اوقات لازم است در معرفی یک پارامتر به عنوان اندازه ورودی احتیاط زیادی به عمل آوریم. برای مثال، شاید فکر می‌کنید که در الگوریتم ۶-۱ (عنصر nام فیبوناچی، بازگشتی) و الگوریتم ۷-۱ (عنصر nام فیبوناچی، تکرار)، مقدار n بایستی به عنوان اندازه ورودی معرفی شود. به هر حال، n یک ورودی است، نه اندازه ورودی. برای این الگوریتم، یک واحد قابل قبول برای اندازه ورودی، تعداد نمادهائی است که n را کددهی کرده‌اند. اگر برای نمایش اعداد از سیستم دودویی استفاده کنیم، اندازه ورودی، تعداد بیت هایی است که برای معرفی n بکار می‌آیند که برابراست با $\lfloor \lg n \rfloor + 1$. برای مثال،

$$n = ۱۳ = \underbrace{۱۱۰۱_۲}_{\text{۴ بیت}}$$

بنابراین اندازه ورودی n = ۱۳ برابر ۴ است. ما با تعیین تعداد عناصری که هر الگوریتم محاسبه می‌کند، بینشی را نسبت به کارایی نسبی دو الگوریتم بدست آورده‌ایم اما هنوز هم n را به عنوان اندازه ورودی نمی‌پذیریم. این نکته، در فصل ۹، وقتی که به جزئیات بیشتری از اندازه ورودی می‌پردازیم، بسیار مهم خواهد بود. تا آن زمان، استفاده از اندازه ورودی ساده‌ای نظیر تعداد عناصر یک آرایه برای ما کافی خواهد بود.

بعد از تعیین اندازه ورودی بایستی دستور یا دستوراتی را انتخاب کنیم که کل عملیات انجام شده توسط الگوریتم با تعداد دفعاتی که این دستور یا دستورات در الگوریتم اجرا می‌شوند، متناسب باشد. این دستور یا دستورات در الگوریتم، **عمل مبنایی** نامیده می‌شوند. به عنوان مثال، در الگوریتم‌های ۱-۱ و ۵-۱ دیدیم که مقدار x در هر گذر از حلقه، با یک عنصر از آرایهٔ S مقایسه می‌شود. بنابراین، دستور مقایسه می‌تواند انتخاب خوبی برای مشخص نمودن عمل مبنایی در هر یک از این دو الگوریتم باشد. با تعیین

تعداد تکرارهای عمل مبنایی در الگوریتم‌ها با مقادیر مختلف n، کارایی‌های نسبی دو الگوریتم به روشنی مشخص می‌شود.

در حالت کلی، **تحلیل پیچیدگی زمانی** یک الگوریتم، تعیین تعداد دفعاتی است که عمل مبنایی به ازاء هر یک از مقادیر اندازه ورودی انجام می‌شود. اگرچه نمی‌خواهیم به جزئیات پیاده‌سازی یک الگوریتم بپردازیم، اما معمولاً فرض می‌کنیم که عمل مبنایی همواره در کاراترین حالت ممکن پیاده‌سازی شده است. برای مثال، فرض کنید که در پیاده سازی الگوریتم ۵-۱، تنها یک مرتبه عمل مقایسه انجام شود. این حالت مبین کاراترین حالت ممکن برای انجام عمل مبنایی است. برای انتخاب عمل مبنایی، هیچ قاعده سریع و فوری وجود ندارد. همانطوریکه قبلاً نیز اشاره شد، ما معمولاً دستورات مربوط به ساختار کنترلی را درنظر نمی‌گیریم. به عنوان مثال، در الگوریتم ۱-۱، دستوراتی که عمل افزایش و مقایسهٔ شاخص را به منظور کنترل حلقهٔ **while** به عهده داشتند، درنظر نگرفتیم. گاهی اوقات کافی است به طور ساده، تنها یک گذر از حلقه را به عنوان یک مرتبه از اجرای عمل مبنایی بررسی کنیم. حتی شخص می‌تواند دستورالعمل ماشین را به عنوان یک مرتبه از اجرای عمل مبنایی در نظر بگیرد اما به دلیل اینکه ما می‌خواهیم تحلیلی مستقل از کامپیوتر داشته باشیم، لذا در این کتاب از آن استفاده نمی‌کنیم.

گاهی اوقات ممکن است بخواهیم دو عمل مبنایی مختلف را در یک الگوریتم بررسی کنیم. به عنوان مثال، در الگوریتمی که به وسیله مقایسهٔ کلیدها عمل مرتب‌سازی را انجام می‌دهد، می‌خواهیم دستورالعمل مقایسه و دستورالعمل انتساب را به عنوان عمل مبنایی در نظر بگیریم. این بدین معنا نیست که دو دستورالعمل با هم عمل مبنایی را تشکیل می‌دهند، بلکه ما دو عمل مبنایی مجزا داریم؛ یکی دستورالعمل مقایسه (قیاس) و دیگری دستورالعمل انتساب. ما به این دلیل اینکار را انجام می‌دهیم که دریک الگوریتم مرتب‌سازی، تعداد مقایسه‌های انجام شده با تعداد اجرای دستورات یکسان نیست. لذا با انتخاب دو عمل مبنایی در یک الگوریتم و تعیین تعداد اجرای هر یک از آنها، می‌توانیم آگاهی بیشتری نسبت به کارایی الگوریتم بدست آوریم.

به خاطر دارید که تحلیل پیچیدگی زمانی یک الگوریتم تعیین می‌کند که برای هر مقدار از اندازه ورودی، چند بار عمل مبنایی انجام می‌شود. در برخی حالات، تعداد دفعات اجرای عمل مبنایی، نه تنها به اندازه ورودی، بلکه به مقادیر ورودی نیز بستگی دارد. الگوریتم ۱-۱ (جستجوی ترتیبی)، از جمله این حالات است. برای مثال، اگر x اولین عنصر آرایه باشد، عمل مبنایی تنها یک مرتبه انجام می‌شود؛ در حالیکه اگر x در آرایه وجود نداشته باشد، عمل مبنایی، n مرتبه اجرا می‌گردد. در حالت دیگر، نظیر الگوریتم ۲-۱ (جمع عناصر آرایه)، عمل مبنایی برای هر نمونه از اندازه ورودی n، به تعداد مشخص و یکسانی انجام می‌شود. در چنین حالتی، **T(n)** مبین تعداد عمل مبنایی است که الگوریتم به ازاء یک نمونه از اندازه ورودی n انجام می‌دهد. T(n) را **پیچیدگی زمانی حالت معمول** الگوریتم و تعیین T(n) را **تحلیل پیچیدگی زمانی حالت معمول** الگوریتم می‌نامیم. مثالی از این تحلیل را در زیر آورده‌ایم:

تحلیل پیچیدگی زمانی حالت معمول الگوریتم ۲-۱ (جمع عناصر آرایه)

غیر از دستورالعمل‌های کنترلی، تنها دستورالعمل موجود در حلقه همان است که یک عنصر آرایه را به sum اضافه می‌کند، لذا این دستور را به عنوان عمل مبنایی در نظر می‌گیریم.

عمل مبنائی: افزودن یک عنصر آرایه به sum

اندازه ورودی: n، تعداد عناصر موجود در آرایه.

بدون توجه به مقادیر n عنصر موجود در آرایه، همواره n گذر از حلقهٔ for وجود خواهد داشت، بنابراین، عمل مبنایی همواره n مرتبه انجام می‌شود و

$$T(n) = n$$

تحلیل پیچیدگی زمانی حالت معمول الگوریتم ۳-۱ (مرتب‌سازی تبادلی)

همانطور که قبلاً نیز اشاره شد، در حالتی که الگوریتم با مقایسه کلیدها عمل مرتب‌سازی را انجام می‌دهد، می‌توانیم دستورالعمل مقایسه یا دستورالعمل انتساب را به عنوان عمل مبنایی در نظر بگیریم. در اینجا، تعداد مقایسات را مورد تجزیه و تحلیل قرار می‌دهیم:

عمل مبنایی: مقایسه S[j] با S[i]

اندازه ورودی: n، تعداد عناصری که باید مرتب شوند.

حال بایستی تعداد گذرهای موجود در حلقهٔ for j را تعیین کنیم. برای هر n معین، همواره تعداد n-۱ گذر از حلقهٔ for i وجود دارد. در اولین گذر از حلقهٔ for i، n-۱ گذر از حلقهٔ for j، در دومین گذر از حلقهٔ for i، n-۲ گذر از حلقهٔ for j، ... و در آخرین گذر، تنها یک گذر از حلقهٔ for j وجود خواهد داشت. بنابراین، تعداد کل گذرهای انجام شده از حلقهٔ for j برابر است با:

$$T(n) = (n-1) + (n-2) + (n-3) + \ldots + 1 = \frac{n(n-1)}{2}$$

حل معادله اخیر در مثال A-۱ از ضمیمه A بررسی شده است.

تحلیل پیچیدگی زمانی حالت معمول الگوریتم ۴-۱ (ضرب ماتریس‌ها)

تنها دستورالعملی که در حلقهٔ for داخلی وجود دارد، همان است که یک عمل ضرب و یک عمل جمع را انجام می‌دهد و این امکان وجود دارد که الگوریتم با روشی بکار گرفته شود که در آن تعداد جمعها کمتر از تعداد ضربها انجام شده باشد، بنابراین ما تنها دستورالعمل ضرب را به عنوان عمل مبنایی در نظر می‌گیریم.

عمل مبنایی: دستورالعمل ضرب در داخلی‌ترین حلقهٔ for.

اندازه ورودی: n، تعداد سطرها و ستونها.

همواره n گذر از حلقهٔ for i وجود دارد که در هر گذر آن، n گذر از حلقهٔ for j و در هر گذر از حلقهٔ for j،

n گذر از حلقهٔ **for k** صورت می‌پذیرد. چون عمل مبنایی در داخلی‌ترین حلقهٔ یعنی حلقهٔ **for k** قرار دارد، لذا داریم:

$$T(n) = n \times n \times n = n^3$$

همانطوریکه قبلاً اشاره کردیم، عمل مبنایی در الگوریتم ۱-۱ برای همه نمونه‌های اندازه ورودی n به تعداد یکسانی انجام نشده است. بنابراین، نمی‌توانیم آن را دارای یک پیچیدگی زمانی حالت معمول بدانیم. این مطلب برای بسیاری از الگوریتم‌ها نیز صادق است. البته منظور ما این نیست که چنین الگوریتم‌های نمی‌توانند تجزیه و تحلیل شوند، چون هنوز سه روش دیگر نیز برای تحلیل الگوریتم‌ها وجود دارد. در این روش، W(n) بعنوان حداکثر دفعات اجرای عمل مبنایی، و برای یک الگوریتم معین، تعیین W(n) بعنوان **پیچیدگی زمانی بدترین حالت** الگوریتم در نظر گرفته می‌شود. واضح است که اگر T(n) وجود داشته باشد، W(n) = T(n) خواهد بود. در زیر تحلیلی از W(n)، در حالتی که T(n) وجود ندارد را آورده‌ایم:

تحلیل پیچیدگی زمانی بدترین حالت الگوریتم ۱-۱ (جستجوی ترتیبی)

عمل مبنایی: مقایسه یک عنصر آرایه با x

اندازه ورودی: n، تعداد عناصر آرایه

عمل مبنایی، حداکثر n مرتبه انجام شده است و این حالتی است که x در آرایه وجود ندارد و یا x آخرین عنصر آرایه است. بنابراین،

$$W(n) = n$$

اگر چه تحلیل بدترین حالت، ما را از حداکثر زمانی که صرف الگوریتم می‌شود، آگاه می‌سازد؛ اما در بعضی حالات، ممکن است به میانگین زمانی الگوریتم نیز علاقه‌مند باشیم. برای یک الگوریتم معین، A(n) به عنوان میانگین (مقدار مورد انتظار) تعداد دفعاتی که الگوریتم، عمل مبنایی را به ازاء هر اندازه ورودی n اجر می‌کند، معرفی شده و تعیین آن را **پیچیدگی زمانی حالت میانی** الگوریتم می‌نامیم. همانند حالت ذکر شده برای W(n)، اگر T(n) وجود داشته باشد، آنگاه A(n) = T(n) خواهد بود.

برای محاسبهٔ A(n)، لازم است که احتمالاتی را به تمامی ورودیهای ممکن به اندازه n نسبت دهیم. این احتمالات باید بر اساس تمامی اطلاعات موجود باشد. به عنوان مثال، تحلیل بعدی ما، یک تحلیل حالت میانی برای الگوریتم ۱-۱ خواهد بود. فرض بر این است که در صورت وجود x در آرایه، x با احتمالهای یکسان در اندیسهای مختلف آرایه جای می‌گیرد. لذا اگر بدانیم که ممکن است x تنها در بعضی از مکانهای آرایه حضور داشته باشد، هیچ دلیلی برای ترجیح یک اندیس آرایه به اندیس دیگر وجود نخواهد داشت. بنابراین، می‌توانیم بپذیریم که احتمالات مساوی به تمام اندیسهای آرایه نسبت داده شود.

این بدین معناست که ما سعی داریم میانگین زمان جستجو را برای زمانی که تعداد یکسانی عنصر مورد جستجو قرار می‌گیرند، تعیین کنیم. اگر ما اطلاعاتی داشته باشیم مبنی بر این که ورودیها، طبق توزیع فرض شده نخواهند رسید، نبایستی چنین توزیعی را در تحلیل الگوریتم به کار بگیریم. برای مثال، اگر آرایه‌ای شامل اسامی افراد باشد و ما در جستجوی نامهایی باشیم که به طور تصادفی از مردم ایالات متحده انتخاب شده‌اند، بالطبع اندیس با محتوای نام متداول john بیشتر از اندیس با محتوای نام غیرمتداول felix جستجو خواهد شد (برای بحث تصادف، به بخش A-۸-۱ از ضمیمه A مراجعه کنید). لذا ما نبایستی از این اطلاعات چشم‌پوشی کرده و فرض کنیم که همه اندیسها با هم یکسانند. همانطوری که مشاهده خواهید کرد، معمولاً تحلیل حالت میانی مشکل‌تر از تحلیل بدترین حالت است.

تحلیل پیچیدگی زمانی حالت میانی الگوریتم ۱-۱ (جستجوی ترتیبی)

عمل مبنایی: مقایسه یک عنصر آرایه با x

اندازه ورودی: n، تعداد عناصر موجود در آرایه.

ابتدا مسئله را در حالتی تحلیل می‌کنیم که می‌دانیم عنصر x حتماً در آرایه وجود دارد. همه عناصر آرایهٔ S به صورت مجزا هستند و هیچ دلیلی وجود ندارد که احتمال x در یک اندیس آرایه را بیشتر از اندیس دیگر بدانیم. براساس این اطلاعات، برای $1 \le k \le n$، احتمال اینکه x در اندیس kام باشد برابر $1/n$ است. اگر x در اندیس kام باشد، تعداد دفعاتی که عمل مبنایی باید اجرا شود تا محل x در آرایه مشخص شود (و از حلقه خارج شود) برابر k است و این موضوع بدین معناست که پیچیدگی زمانی حالت میانی برابراست با:

$$A(n) = \sum_{k=1}^{n} (k \times \frac{1}{n}) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

سومین تساوی در مثال A-۱ از ضمیمه A بررسی شده است. بنابراین انتظار داریم براساس میانگین بدست آمده، حدود نیمی از آرایه جستجو شود.

در ادامه، مسئله را در حالتی تحلیل می‌کنیم که ممکن است x در آرایه وجود نداشته باشد. برای تحلیل این حالت، فرض می‌کنیم که احتمال اینکه x در آرایه موجود باشد، برابر p بوده و در صورت وجود x در آرایه، احتمال اینکه در هر یک از اندیسهای ۱ تا n جای گیرد، یکسان باشد. بدین ترتیب، احتمال اینکه x در اندیس kام آرایه باشد برابراست با $\frac{p}{n}$ و احتمال اینکه x در آرایه نباشد برابراست با $p - 1$. یادآور می‌شویم که بایستی kگذر از حلقه انجام شود تا به عنصر x در اندیس kام دست یابیم یا nگذر از حلقه صورت گیرد تا پی ببریم که x در آرایه موجود نیست. بنابراین پیچیدگی زمانی حالت میانی برابراست با:

$$A(n) = \sum_{k=1}^{n} (k \times \frac{p}{n}) + n(1-p) = \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n(1-\frac{p}{2}) + \frac{p}{2}$$

اگر $p = 1$ باشد، آنگاه $A(n) = (n+1)/2$ و اگر $p = 1/2$ باشد، آنگاه $A(n) = 3n/4 + 1/4$ خواهد شد؛ یعنی به طور میانگین، در حدود ۳/۴ از آرایه باید مورد جستجو قرار گیرد.

اگر چه یک میانگین، اغلب به عنوان یک رخداد خاص مطرح می‌شود. اما در تفسیر میانگین به این صورت بایستی بسیار دقیق باشیم. برای مثال، یک هواشناس ممکن است بگوید که در یک ۲۵ ژانویهٔ خاص، دمای هوا در شیکاگو ۲۲ درجه فارنهایت است چون میانگین دمای هوا در ۸۰ سال گذشته در این تاریخ، ۲۲ درجه فارنهایت بوده است و یا روزنامه‌ای بنویسید که درآمد یک خانوادهٔ خاص در Evanston برابر ۵۰،۰۰۰ دلار است چون میانگین درآمد خانواده‌های این شهر ۵۰،۰۰۰ دلار است. ما تنها زمانی می‌توانیم یک میانگین رابه عنوان یک رخداد خاص در نظر بگیریم که حالتهای حقیقی، خیلی دور از مقدار میانگین نباشند؛ به عبارتی، تنها زمانی که انحراف معیار مقداری کوچک باشد. لذا ممکن است دمای هوای روز ۲۵ ژانویه بیشتر باشد. شهر Evanston، اجتماعی از خانواده‌های ثروتمند و فقیر است و احتمال اینکه درآمد خانواده‌ای ۲۰،۰۰۰ و یا ۱۰۰،۰۰۰ دلار در سال باشد، بیشتر از ۵۰،۰۰۰ دلار در سال است.

با مروری بر تحلیل گذشته، A(n) وقتی برابر $(n+1)/2$ است که x در آرایه وجود داشته باشد. این مورد به عنوان یک نمونهٔ خاص از زمان جستجو نیست زیرا تمامی این موارد با توزیعی یکنواخت بین ۱ و n قرار دارند. چنین نگرشی در مورد الگوریتم‌هایی که با زمان پاسخ سروکار دارند، بسیار حائز اهمیت است. برای مثال، سیستم هشدار دهندهٔ یک رآکتور هسته‌ای را درنظر بگیرید. در صورت وجود حتی یک زمان پاسخ بد، نتایج بدست آمده فاجعه‌آمیز خواهد بود. بنابراین، دانستن این نکته که آیا میانگین زمانهای پاسخ برابر با ۳ ثانیه است، قابل توجّه خواهد بود؛ زیرا به این نکته پی می‌بریم که تمام واکنشها در مدت زمانی حدود ۳ ثانیه انجام شده و یا اینکه بیشتر آنها در یک ثانیه و برخی دیگر در ۶۰ ثانیه انجام می‌شوند.

یک مورد دیگر در تحلیل پیچیدگی زمانی، تعیین حداقل تعداد دفعاتی است که یک عمل مبنایی انجام می‌شود. در یک الگوریتم در حال بررسی، **B(n)** نشان‌دهندهٔ حداقل تعداد دفعات اجرای عمل مبنایی به ازاء ورودی n است. به همین دلیل، تعیین B(n)، **پیچیدگی زمانی بهترین حالت** الگوریتم نامیده می‌شود. همانند حالات W(n) و A(n)، اگر T(n) وجود داشته باشد، آنگاه B(n) = T(n).

تحلیل پیچیدگی زمانی بهترین حالت الگوریتم ۱-۱ (جستجوی ترتیبی)

عمل مبنایی: مقایسهٔ یک عنصر آرایه با x

اندازه ورودی: n، تعداد عناصر آرایه.

از آنجائیکه $n \geq 1$ است، لذا بایستی حداقل یک گذر از حلقه وجود داشته باشد و اگر $x = S[1]$ باشد بدون توجه به مقدار اندازه ورودی n، تنها یک گذر از حلقه وجود خواهد داشت. بنابراین،

$$B(n) = 1$$

برای الگوریتم‌هایی که پیچیدگی زمانی حالت معمول ندارند، اغلب از تحلیل‌های بدترین حالت و حالت میانی استفاده می‌کنیم. یک تحلیل حالت میانی بسیار با ارزش است زیرا به ما اطلاعاتی در مورد مقدار زمان صرف شده برای اجرای الگوریتم با ورودیهای مختلف ارائه می‌دهد. این موضوع،

برای الگوریتم‌هایی نظیر الگوریتم مرتب‌سازی که مکرراً برای همه ورودیهای ممکن بکار گرفته می‌شود، مفید می‌باشد. اغلب، یک مرتب‌سازی نسبتاً کند بشرطی قابل تحمل است که میانگین زمان مرتب‌سازی آن، خوب باشد. در بخش ۴-۲، الگوریتمی را به نام مرتب‌سازی سریع بررسی خواهیم کرد که دقیقاً این مورد درباره آن صدق می‌کند. این الگوریتم، یکی از متداولترین الگوریتمهای مرتب‌سازی است. آنچنانکه قبلاً نیز اشاره شد، یک تحلیل حالت میانی برای سیستم هشداردهندهٔ رآکتور هسته‌ای نمی‌تواند کافی باشد. در این حالت تحلیل بدترین حالت بسیار مفید است زیرا بالاترین حد زمانی که سیستم صرف بکارگیری الگوریتم می‌کند را نشان می‌دهد. برای هر دو مثال فوق، تحلیل بهترین حالت بسیار کم ارزش است.

ما فقط در مورد تحلیل پیچیدگی زمانی یک الگوریتم بحث کرده‌ایم. توجه داریم که کارایی الگوریتمها، به تحلیل پیچیدگی حافظه‌ای نیز بستگی دارد. با وجود اینکه بیشتر مباحث این کتاب در تحلیل پیچیدگی زمانی است، در یک فرصت مناسب تحلیل پیچیدگی حافظه را نیز بررسی خواهیم کرد.

به طور کلی، یک **تابع پیچیدگی** می‌تواند هر تابعی از اعداد صحیح غیرمنفی به اعداد حقیقی غیرمنفی باشد. هر گاه در تحلیل برخی از الگوریتمهای خاص به پیچیدگی زمانی یا پیچیدگی حافظه اشاره نشود، معمولاً از توابع استانداری نظیر f(n) و g(n) به عنوان توابع پیچیدگی استفاده می‌شود.

مثال ۶-۱ توابع

$$f(n) = n$$
$$f(n) = n^2$$
$$f(n) = lg\ n$$
$$f(n) = 3n^2 + 4n$$

همگی مثالهایی از توابع پیچیدگی هستند زیرا تمامی آنها تابعی از اعداد صحیح غیرمنفی به اعداد حقیقی غیرمنفی می‌باشند.

۲-۳-۱ استفاده از تئوری

گاهی لازم است که در هنگام بکارگیری تئوری تحلیل یک الگوریتم، به مواردی چون مدت زمان اجرای عمل مبنایی، دستورات سربار و دستورات کنترلی کامپیوتری که الگوریتم را به کار گرفته است نیز توجه شود. "دستورات سربار" به دستوراتی نظیر مقداردهی اولیهٔ دستورات قبل از یک حلقه اطلاق می‌گردد. تعداد دفعات اجرای این دستورات با بزرگتر شدن اندازه ورودی، افزایش نمی‌یابد. "دستورات کنترلی"، به دستوراتی نظیر افزایش مقدار شاخص به منظور کنترل حلقه اطلاق می‌شود که تعداد دفعات اجرای این دستورات، برخلاف دستورات سربار، با بزرگتر شدن اندازه ورودی افزایش می‌یابد. عمل مبنایی،

دستورات سربار و دستورات کنترلی همگی از خصوصیات یک الگوریتم و پیاده‌سازی آن هستند، نه از خصوصیات مسئله. به عبارت دیگر، این مفاهیم برای هر دو الگوریتمی که برای یک مسئله ارائه می‌شوند، متفاوت خواهند بود.

فرض کنید که برای یک مسئله، دو الگوریتم با پیچیدگی‌های زمانی حالت معمول n و n^2 ارائه شده است (الگوریتم اول کاراتر به نظر می‌رسد) و کامپیوتر اجراکنندهٔ این الگوریتم‌ها، عمل مبنایی الگوریتم اول را ۱۰۰۰ مرتبه طولانی‌تر (کندتر) از عمل مبنایی الگوریتم دوم پردازش می‌کند. وقتی صحبت از پردازش می‌کنیم، بایستی زمان اجرای دستورات کنترلی را نیز در نظر بگیریم. بنابراین، اگر t مدت زمان لازم برای یک بار پردازش عمل مبنایی در الگوریتم دوم باشد، ۱۰۰۰t مدت زمانی است که الگوریتم اول برای یک بار پردازش عمل مبنایی‌اش صرف می‌کند. برای ساده‌تر شدن مسئله، از محاسبهٔ ۲زمان مورد نیاز برای دستورات سربار در هر دو الگوریتم صرف نظر می‌کنیم. بدین ترتیب، مدت زمان لازم برای پردازش الگوریتم اول با اندازه ورودی n برابر ۱۰۰۰t × n و این مدت زمان برای الگوریتم دوم برابر $n^2 \times t$ می‌باشد. تعیین اینکه چه وقت الگوریتم اول کاراتر است، مستلزم حل نامعادله زیر است:

$$n^2 \times t > n \times ۱۰۰۰t$$

با تقسیم طرفین نامساوی به nt داریم:

$$n > ۱۰۰۰$$

بنابراین اگر اندازه ورودی کوچکتر از ۱۰۰۰ باشد بایستی از الگوریتم دوم استفاده کنیم. پیش از ادامه بحث، باید به این نکته توجه کنیم که مشخص نمودن اینکه دقیقاً یک الگوریتم چه وقت از الگوریتمی دیگر سریعتر است، همیشه کار آسانی نیست. گاهی اوقات بایستی از روشهای تقریب‌زنی برای تحلیل نامساوی‌ها جهت مقایسه دو الگوریتم استفاده کنیم.

به خاطر دارید که در مثال فوق، مدت زمان لازم برای پردازش دستورات سربار را درنظر نگرفتیم. اگر این فرض وجود نداشت، می‌بایستی با درنظر گرفتن این دستورات، الگوریتم کاراتر را مشخص می‌کردیم.

۱-۳-۳ تحلیل درستی

در این کتاب، "تحلیل یک الگوریتم" به این معناست که کارایی الگوریتم، از لحاظ زمان و حافظه بررسی شود. انواع دیگری از تجزیه و تحلیل هم وجود دارند. برای مثال، ما می‌توانیم با اثبات این نکته که الگوریتم ما واقعاً همان کاری را انجام می‌دهد که پیش‌بینی می‌شد، صحت و درستی یک الگوریتم را تحلیل کنیم. با اینکه اغلب، بدون استفاده از منطق صوری نشان می‌دهیم که الگوریتم ما درست کار می‌کند و حتی گاهی اوقات آن را به اثبات می‌رسانیم، ولی به منظور آشنایی کامل با مبحث درستی یک الگوریتم، لازم است به مقالات Kingston (1990)، Gries (1981) یا Dijkstra (1976) نیز توجه کنید.

۱-۴ ترتیب (order)

نشان داده‌ایم که یک الگوریتم با پیچیدگی زمانی n برای مقادیر بزرگ n (بدون توجه به مدت زمان اجرای عمل مبنایی)، از الگوریتمی با پیچیدگی زمانی n^2 کاراتر است. فرض کنید برای یک مسئله، دو الگوریتم با پیچیدگی‌های زمانی حالت معمول $100n$ و $0.1n^2$ ارائه شده است. با استفاده از روش قبلی می‌توانیم نشان دهیم که الگوریتم اول، در نهایت از الگوریتم دوم کاراتر است. برای مثال، با فرض اینکه مدت زمان لازم برای پردازش عملیات مبنایی و دستورات سربار در هر الگوریتم یکسان باشد، الگوریتم اول کاراتر خواهد بود اگر

$$0.1n^2 > 100n$$

با تقسیم دو طرف نامساوی به 0.1n داریم:

$$n > 1000$$

اگر مدت زمان لازم برای پردازش عمل مبنایی در الگوریتم اول بیشتر از الگوریتم دوم باشد، آنگاه الگوریتم اول به ازاء برخی مقادیر بزرگتر n، کاراتر می‌گردد.

الگوریتم‌هایی با پیچیدگی زمانی n و 100n، به الگوریتم‌های **زمان- خطی (linear-time)** موسومند. به این دلیل که پیچیدگی زمانی آنها روی اندازه ورودی n به صورت خطی است؛ در حالیکه الگوریتمهایی با پیچیدگی زمانی n^2 و $0.1n^2$، الگوریتم‌های **زمان-مربعی (quadratic-time)** نامیده می‌شوند. زیرا پیچیدگی زمانی آنها مربعی از اندازه ورودی n است. در اینجا یک اصل اساسی مطرح است و آن اینکه یک الگوریتم زمان-خطی، نهایتاً از یک الگوریتم زمان-مربعی کاراتر است. در تحلیل تئوری یک الگوریتم، رفتار نهایی الگوریتم مورد توجه است. در ادامه خواهید دید که چگونه می‌توان الگوریتمها را براساس رفتار نهائی‌شان دسته‌بندی کرد.

۱-۴-۱ مقدمه‌ای بر ترتیب

توابعی نظیر $5n^2$ و $100 + 5n^2$، به توابع **مربعی محض (pure quadratic)** موسومند زیرا در آنها اثری از عناصر خطی دیده نمی‌شود؛ در حالیکه تابعی چون $100 + n + 0.1n^2$، به دلیل وجود یک عنصر خطی، تابع **مربعی کامل (complete quadratic)** نامیده می‌شود. جدول ۳-۱، برتری عنصر درجه دوم را در این تابع نشان می‌دهد، یعنی مقادیر سایر عناصر در مقایسه با عنصر مربعی، نهایتاً (به ازاء مقادیر بزرگ n) کم ارزش و کم اهمیت می‌شود. بنابراین، اگرچه تابع $100 + n + 0.1n^2$ یک تابع مربعی محض نیست، ولی می‌توانیم آن را در این گروه جای داده و اینچنین تعمیم دهیم که هر الگوریتمی که دارای چنین پیچیدگی زمانی باشد، می‌تواند به عنوان یک الگوریتم زمان-مربعی معرفی شود. به نظر می‌رسد که بتوانیم به هنگام طبقه‌بندی توابع پیچیدگی، عناصر با ترتیب پایین را کم اهمیت فرض کرده و آنها را درنظر نگیریم. برای مثال، می‌توانیم تابع $25 + 5n + 10n^2 + 0.1n^3$ را با توابع مکعبی محض، در یک گروه طبقه‌بندی کنیم. به زودی، روشی کامل برای انجام این کار ارائه خواهیم داد؛ اما ابتدا اجازه دهید یک تصویر اولیه برای طبقه‌بندی توابع پیچیدگی ارائه دهیم.

جدول ۳-۱ عنصر مربعی در نهایت برتری می‌یابد.

n	$0.1n^2$	$0.1n^2 + n + 100$
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1000	100,000	101,100

مجموعهٔ تمامی توابع پیچیدگی که می‌توانند با توابع مربعی محض طبقه‌بندی شوند، $\Theta(n^2)$ نامیده می‌شوند [علامت Θ (تتا) یک حرف بزرگ یونانی است]. اگر یک تابع، عنصری از مجموعهٔ $\Theta(n^2)$ باشد، می‌گوئیم که آن تابع، یک ترتیب از n^2 است. برای مثال، چون می‌توانیم از عناصر با ترتیب پایین صرف نظر کنیم، لذا

$$g(n) = 5n^2 + 100n + 20 \in \Theta(n^2)$$

به این معنی که $g(n)$ ترتیبی از n^2 است. برای ارائه یک مثال واقعی‌تر، الگوریتم ۳-۱ (مرتب‌سازی تبادلی) را بادآور می‌شویم که پیچیدگی زمانی آن رابه صورت $T(n) = n(n-1)/2$ بیان کرده‌ایم. از آنجائیکه $n(n-1)/2 = n^2/2 - n/2$ است، لذابا کنار گذاشتن عنصر کم ترتیب $n/2$ می‌توان گفت که

$$T(n) \in \Theta(n^2)$$

هنگامی که پیچیدگی زمانی یک الگوریتم در $\Theta(n^2)$ است، الگوریتم را **الگوریتم زمان-مربعی** یا الگوریتم $\Theta(n^2)$ می‌نامیم. مرتب‌سازی تبادلی، یک الگوریتم زمان-مربعی است. به طور مشابه، سری توابع پیچیدگی که می‌توانند به همراه توابع مکعبی کامل دسته‌بندی شوند، $\Theta(n^3)$ و یا ترتیب n^3 نامیده می‌شوند والی آخر. ما به این سری‌ها، **رده‌های پیچیدگی** می‌گوئیم. رده‌های زیر برخی از رایج‌ترین رده‌های پیچیدگی هستند:

$$\Theta(\lg n) \quad \Theta(n) \quad \Theta(n \lg n) \quad \Theta(n^2) \quad \Theta(n^3) \quad \Theta(2^n)$$

به این ترتیب، اگر $f(n)$ در رده‌ای واقع در سمت چپ ردهٔ شامل $g(n)$ باشد، آنگاه $f(n)$ در روی نمودار، نهایتاً زیر $g(n)$ قرار می‌گیرد. شکل ۳-۱، ساده‌ترین اعضاء این رده را نشان می‌دهد: $n \lg n$ ،$\lg n$ ،n و غیره. جدول ۴-۱، زمان اجرای الگوریتم‌هایی که پیچیدگی‌های زمانی آنها با این توابع بیان می‌شود را نشان می‌دهد. فرض کنید که پردازش عمل مبنایی برای هر الگوریتم، یک نانوثانیه 10^{-9} طول می‌کشد. این جدول نتیجه‌ای را نشان می‌دهد که شاید تعجب آور باشد. با توجه به جدول، احتمالاً به این نتیجه می‌رسیم که، همین که یک الگوریتم از نوع زمان-نمایی نباشد، برای ما کافی است. به هر حال، حتی یک الگوریتم زمان-مربعی برای پردازش یک نمونه با اندازه ورودی یک میلیارد، ۳۱/۷ سال وقت می‌گیرد. در حالیکه الگوریتم $\Theta(n \lg n)$ تنها ۲۹/۹ ثانیه برای پردازش چنین نمونه‌ای زمان صرف می‌کند.

جدول ۴-۱ زمانهای اجرا برای الگوریتم‌هایی با پیچیدگی زمانی معین.

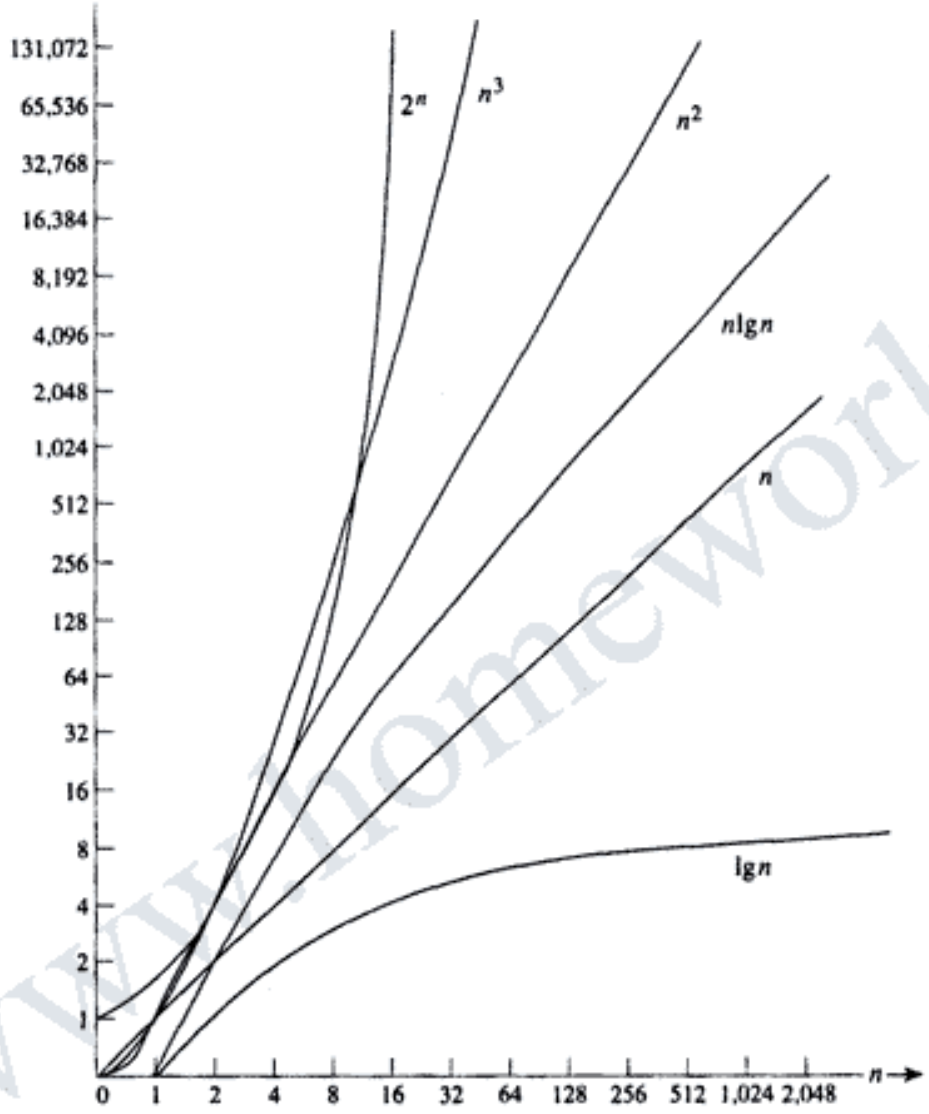
n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs*	0.01 μs	0.033 μs	0.1 μs	1 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	125 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10 μs	1 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1 ms	1 s	
10^4	0.013 μs	10 μs	130 μs	100 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.67 ms	10 s	11.6 days	
10^6	0.020 μs	1 ms	19.93 ms	16.7 min	31.7 years	
10^7	0.023 μs	0.01 s	0.23 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.66 s	115.7 days	3.17×10^7 years	
10^9	0.030 μs	1 s	29.90 s	31.7 years		

*1 μs = 10^{-6} second.
†1 ms = 10^{-3} second.

بطور کلی، یک الگوریتم باید به شکل $\Theta(n \lg n)$ یا بهتر از آن باشد تا بتوانیم فرض کنیم که نمونه‌های بسیار بزرگ ورودی می‌توانند در مدت زمان قابل قبولی حل شوند. البته این بدین معنا نیست که الگوریتم‌هایی که پیچیدگی‌های زمانی آنها در رده‌های بالاتر واقع است قابل استفاده و مفید نیستند؛ بلکه الگوریتم‌های با پیچیدگی‌های زمانی درجه دوم، درجه سوم و حتی بالاتر، اغلب می‌توانند نمونه‌های عملی که در بسیاری از کاربردها به وجود می‌آیند را به خوبی جواب دهند.

قبل از پایان دادن به این بحث تأکید می‌کنیم که برای شناخت دقیق یک پیچیدگی زمانی، اطلاعاتی دقیق‌تر و جامع‌تر از شناخت سادهٔ ترتیب آن نیز وجود دارد. برای مثال، الگوریتم‌های فرضی که قبلاً درباره آنها بحث کردیم و دارای پیچیدگی‌های زمانی $100n$ و $0/01n^2$ بودند را درنظر می‌گیریم. اگر پردازش عملیات مبنایی و اجرای دستورالعمل‌های سربار در هر دو الگوریتم به یک اندازه طول بکشد، آنگاه الگوریتم زمان-مربعی برای نمونه‌های کوچکتر از ۱۰۰۰۰، کاراتر خواهد بود. اگر در عمل، هیچگاه نمونه هایی بزرگتر از این نداشته باشیم، بایستی الگوریتم زمان-مربعی را پیاده سازی کنیم. ضرائب کمکی در این مثال بسیار بزرگ اند؛ درحالیکه در عمل، به این بزرگی نیستند. علاوه بر این، نمونه‌هایی وجود دارد که تعیین دقیق پیچیدگی‌های زمانی آنها بسیار مشکل است. لذا گاهی اوقات، تنها به تعیین ترتیب آنها راضی می‌شویم.

شکل ۳-۱ نرخ رشد برخی از توابع پیچیدگی متداول.



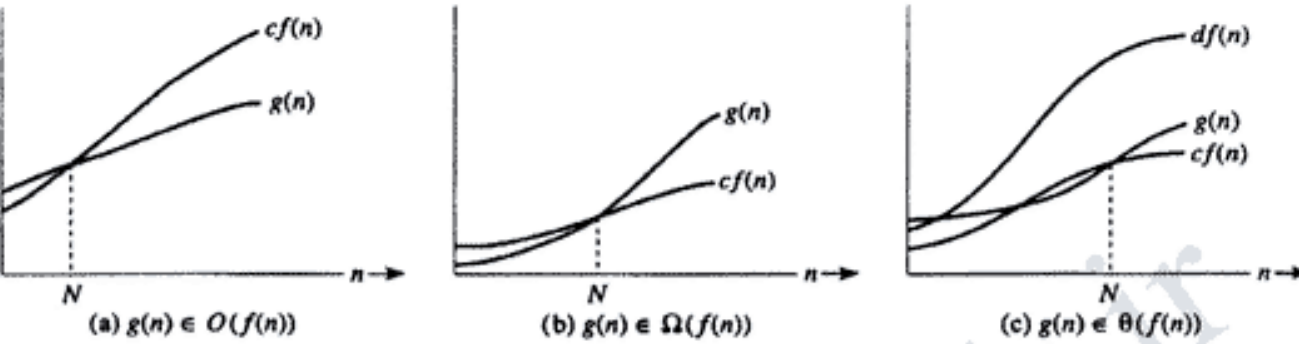
۲-۴-۱ معرفی کامل ترتیب

تا به حال مفاهیم و اشاراتی را درباره ترتیب (Θ) بیان داشتیم، حال سعی می‌کنیم که با طرح دو مفهوم اساسی و بنیادین تعریف دقیق و جامعی از ترتیب ارائه دهیم. اولین مفهوم، big O یا "O بزرگ" نام دارد.

تعریف برای تابع پیچیدگی مفروض $f(n)$، $O(f(n))$ مجموعه‌ای از توابع پیچیدگی $g(n)$ است که برای آن ثابت مثبت و حقیقی C و عدد صحیح غیرمنفی N یافت می‌شود بطوری که به ازای تمامی مقادیر $n \geq N$ داریم:

$$g(n) \leq c \times f(n)$$

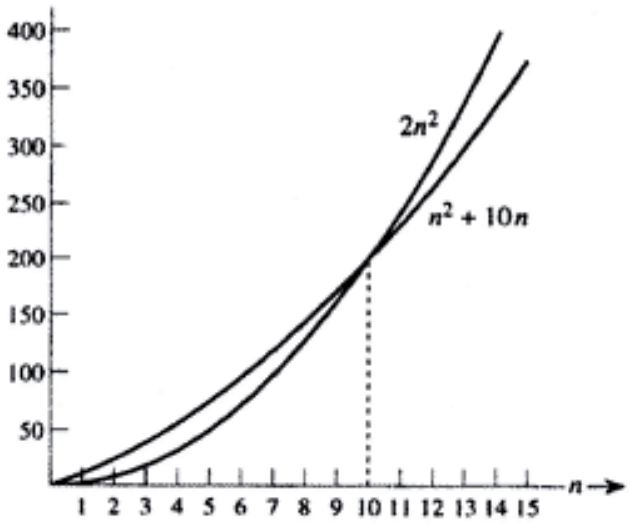
شکل ۴-۱ پیچیدگی‌های زمانی big O، Ω و Θ.



اگر $g(n) \in O(f(n))$ باشد، می‌گوئیم $g(n)$، یک **big O** از $f(n)$ است. شکل ۴-۱(a)، "big O" را نشان می‌دهد. اگرچه در ابتدا $g(n)$ در بالای $cf(n)$ قرار دارد ولی در نهایت به زیر $cf(n)$ نزول کرده و در زیر آن نیز باقی خواهد ماند. شکل ۵-۱، یک مثال حقیقی از آن را نشان می‌دهد. در این شکل، اگرچه $n^2 + 10n$، ابتدا در بالای $2n^2$ قرار دارد، ولی برای $n \geq 10$ داریم:

$$n^2 + 10n \leq 2n^2$$

بنابراین، می‌توانیم در تعریف big O، $c = 2$ و $N = 10$ را در نظر بگیریم تا رابطه $n^2 + 10n \in O(n^2)$ بدست آید. اگر، به عنوان مثال، $g(n)$ روی ترتیب $O(n^2)$ باشد، سرانجام $g(n)$ بر روی نمودار به زیر یک تابع مربعی محض مانند cn^2 نزول خواهد کرد. این بدین معناست که اگر $g(n)$ پیچیدگی زمانی یک الگوریتم باشد، نهایتاً زمان اجرای الگوریتم، حداقل به سرعت یک نمونهٔ درجه دومی



شکل ۵-۱ تابع $n^2 + 10n$ سرانجام در زیر تابع $2n^2$ قرار می‌گیرد.

خواهد بود. از نظر تحلیلی می‌توان گفت که در نهایت $g(n)$ حداقل به خوبی یک تابع مربعی محض می‌باشد. روش big O (و سایر روشهایی که به زودی معرفی می‌شوند) رفتارهای **جانبی** یک تابع را توجیه می‌کنند، چراکه این روشها تنها با رفتارهای نهائی توابع سروکار دارند. در این حالت می‌گوئیم "روش big O یک **حد بالای مجانب** بر یک تابع می‌نهد." در مثالهای زیر، به چند نمونه از توابع big O توجه کنید:

مثال ۱-۷ نشان می‌دهیم که $5n^2 \in O(n^2)$ است. ازآنجائیکه برای $n \geq 0$ داریم

$$5n^2 \leq 5n^2$$

می‌توانیم c را برابر ۵ و N را برابر صفر بگیریم تا نتیجه مورد نظر حاصل می‌شود.

مثال ۱-۸ به خاطر دارید که پیچیدگی زمانی الگوریتم ۳-۱ برابراست با

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

و چون برای $n \geq 0$ داریم:

$$T(n) = \frac{n(n-1)}{2} \leq \frac{n(n)}{2} = \frac{1}{2}n^2$$

می‌توانیم c را برابر ۱/۲ و N را برابر صفر بگیریم تا به نتیجهٔ $T(n) \in O(n^2)$ برسیم.

مشکلی که اغلب دانشجویان با روش big O دارند این است که آنها به اشتباه فکر می‌کنند فقط یک c و یک N منحصر به فرد برای نشان دادن یک تابع به عنوان یک big O از تابع دیگر وجود دارد؛ در حالیکه به هیچ وجه چنین نیست. به خاطر دارید که شکل ۵-۱ (با استفاده از $c = 2$ و $N = 10$) نشان می‌داد که $n^2 + 10n \in O(n^2)$ است.

مثال ۱-۹ نشان می‌دهیم که $n^2 + 10n \in O(n^2)$ است. از آنجائیکه برای $n \geq 1$ داریم

$$n^2 + 10n \leq n^2 + 10n^2 = 11n^2$$

لذا با انتخاب $c = 1$ و $N = 1$ به نتیجه مطلوب خود می‌رسیم.

بطورکلی می‌توان روش big O را با هر دستکاری که لازم باشد تغییر داد تا واضح‌تر و ساده‌تر به نظر آید.

مثال ۱-۱۰ می‌توانیم نشان دهیم که $n^2 \in O(n^2 + 10n)$ است. از آنجائیکه برای $n \geq 0$ داریم

$$n \leq 1 \times (n^2 + 10n)$$

لذا با انتخاب $c = 1$ و $N = 0$، به نتیجهٔ مطلوب می‌رسیم.

هدف از این مثال اینست که توابع درون big O، لزوماً نباید یکی از توابع ساده‌ای باشند که در شکل ۳-۱ نشان داده شده است، بلکه می‌تواند هر تابع پیچیدگی باشد. اگر چه اغلب آنها را به صورت توابع ساده (نظیر توابع شکل ۳-۱) در نظر می‌گیریم.

مثال ۱۱-۱ می‌توانیم نشان دهیم که $n \in O(n^2)$ است. از آنجائیکه برای $n \geq 1$ داریم

$$n \leq 1 \times n^2$$

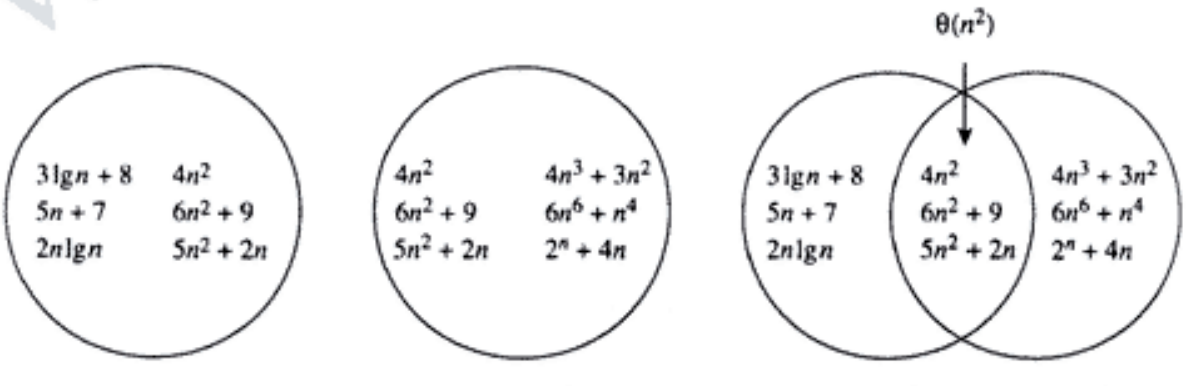
لذا با انتخاب $c = 1$ و $N = 1$ به نتیجه مطلوب خود می‌رسیم.

از مثال اخیر نتیجه می‌گیریم که لزومی ندارد یک تابع پیچیدگی حتماً یک عنصر درجهٔ ۲ داشته باشد تا در $O(n^2)$ جای گیرد، بلکه بایستی نهایتاً روی نمودار در زیر برخی توابع مربعی محض قرار بگیرد. بنابراین، هر تابع پیچیدگی خطی یا لگاریتمی را می‌توان در $O(n^2)$ جای داد. به همین ترتیب، هر تابع پیچیدگی لگاریتمی، خطی یا درجهٔ ۲ را می‌توان در $O(n^3)$ جای داد و همینطور الی آخر. شکل ۶-۱، چند نمونه از اعضاء $O(n^2)$ را نشان می‌دهد. همانطوری که big O یک حدّ بالای مجانب بر یک تابع پیچیدگی قرار می‌دهد، مفهوم زیر یک **حدّ پایین مجانب** بر یک تابع پیچیدگی قرار می‌دهد.

تعریف برای یک تابع پیچیدگی $f(n)$، $\Omega(f(n))$ شامل مجموعه‌ای از توابع پیچیدگی $g(n)$ است که برای آن ثابت حقیقی مثبت c و عدد صحیح غیرمنفی N یافت می‌شود بطوری که به ازاء تمامی مقادیر $n \geq N$ داریم:

$$g(n) \geq c \times f(n)$$

علامت Ω (امگا) یک حرف بزرگ یونانی است. اگر $g(n) \in \Omega(f(n))$ باشد، می‌گوئیم که $g(n)$ امگائی است از $f(n)$. شکل ۴-۱(b)، امگا را نشان می‌دهد. به مثالهای زیر توجه کنید:



(a) $\theta(n^2)$ (b) $\Omega(n^2)$ (c) $\theta(n^2) = O(n^2) \cap \Omega(n^2)$

شکل ۶-۱ مجموعه‌های $O(n^2)$, $\Omega(n^2)$ و $\Theta(n^2)$. چند نمونه از اعضاء آنها مشخص شده است.

مثال ۱۲-۱ نشان می‌دهیم که $5n^2 \in \Omega(n^2)$ است. از آنجائیکه برای $n \geq 0$ داریم

$$n^2 \times 5n^2 \geq 1$$

لذا می‌توانیم c را برابر ۱ و N را برابر صفر بگیریم تا نتیجهٔ مطلوب حاصل گردد.

مثال ۱۳-۱ نشان می‌دهیم که $n^2 + 10n \in \Omega(n^2)$ است. چون برای $n \geq 0$ داریم

$$n^2 + 10n \geq n^2$$

لذا با انتخاب $c = 1$ و $N = 0$ به نتیجهٔ مطلوب می‌رسیم.

مثال ۱۴-۱ پیچیدگی زمانی الگوریتم ۳-۱ (مرتب‌سازی تبادلی) را در نظر بگیرید. نشان می‌دهیم که

$$T(n) = \frac{n(n-1)}{2} \in \Omega(n^2)$$

برای $n \geq 2$ داریم:

$$n - 1 \geq \frac{n}{2}$$

بنابراین، برای $n \geq 2$

$$\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4}n^2$$

به این معنی که ما می‌توانیم با $c = 1/4$ و $N = 2$ به نتیجه مطلوب برسیم.

همانند "big O"، در تعریف Ω نیز مقادیر ثابت منحصر به فردی برای c و N وجود ندارد و ما می‌توانیم هر کدام از آنها که موجب آسانتر شدن کار می‌شود را انتخاب نمائیم.

اگر یک تابع در $\Omega(n^2)$ باشد، سرانجام این تابع در روی نمودار، در بالای برخی توابع درجه دوم محض قرار می‌گیرد. از نظر تحلیلی، بدین معناست که آن تابع، حداقل به بدی یک تابع مربعی محض خواهد شد. به هر حال، آنچنانکه در مثالهای زیر خواهید دید، لزومی ندارد که تابع حتماً از درجه دوم باشد.

مثال ۱۵-۱ نشان می‌دهیم که $n^3 \in \Omega(n^2)$ است.

از آنجائیکه اگر $n \geq 1$ باشد، آنگاه $n^3 \geq 1 \times n^2$ می‌شود، لذا با انتخاب $c = 1$ و $N = 1$ به جواب مسئله می‌رسیم.

شکل ۱-۶(b) چند عضو از $\Omega(n^2)$ را به طور نمونه نشان می‌دهد.

اگر یک تابع، هم در $O(n^2)$ و هم در $\Omega(n^2)$ باشد، می‌توان نتیجه گرفت که تابع در روی نمودار، سرانجام در زیر برخی توابع درجه دوم محض و در بالای برخی توابع درجه دوم محض قرار می‌گیرد. یعنی می‌توان گفت که در نهایت آن تابع، حداقل به خوبی چند تابع درجه دوم محض و حداقل به بدی برخی توابع درجه دوم محض خواهد بود. بنابراین، می‌توان نتیجه گرفت که چنین تابعی مشابه با یک تابع درجه دوم محض رشد می‌کند و این دقیقاً همان نتیجه‌ای است که برای شناخت کامل ترتیب نیاز داریم.

تعریف برای تابع پیچیدگی مفروض f(n)،

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

این بدین معناست که $\Theta(f(n))$ مجموعه‌ای از توابع پیچیدگی $g(n)$ است که ثابت حقیقی c و d، و عدد صحیح غیرمنفی N برای آن یافت می‌شود بطوری که به ازاء همهٔ مقادیر $n \geq N$ داریم:

$$c \times f(n) \leq g(n) \leq d \times f(n)$$

اگر $g(n) \in \Theta(f(n))$ باشد، گوئیم $g(n)$ یک **ترتیب** از $f(n)$ است.

مثال ۱۶-۱ یک بار دیگر پیچیدگی زمانی الگوریتم ۳-۱ را درنظر بگیرید. مثالهای ۳-۱، ۸-۱ و ۱۴-۱ ثابت کردند که

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

هم در $O(n^2)$ و هم در $\Omega(n^2)$ قرار دارد. این بدین معناست که

$$T(n) \in O(n^2) \cap \Omega(n^2) = \Theta(n^2)$$

شکل ۶-۱(c) نشان می‌دهد که $\Theta(n^2)$ از اشتراک $O(n^2)$ و $\Omega(n^2)$ بدست می‌آید. شکل ۴-۱(c) نیز Θ را نشان داده است. در شکل ۶-۱(c) توجه کنید که تابع $5n + 7$ در $\Omega(n^2)$ و تابع $4n^3 + 3n^2$ در $O(n^2)$ نیست، لذا هیچ یک از این توابع در $\Theta(n^2)$ نخواهند بود. اگرچه این مطلب صحیح بنظر می‌رسد، اما هنوز آن را ثابت نکرده‌ایم. می‌خواهیم با استفاده از مثال نقض، این مطلب را ثابت کنیم.

مثال ۱۷-۱ با استفاده از **برهان خلف** نشان می‌دهیم که n در $\Omega(n^2)$ نیست.

در این نوع از برهان فرض می‌کنیم که برخی چیزها - در این حالت $n \in \Omega(n^2)$ - صحیح است، آنگاه با اعمال برخی قوانین به نتیجه‌ای منتهی می‌شویم که درست نیست. یعنی اینکه نتیجه، خلاف آن چیزهایی می‌شود که ما درست فرض کرده بودیم و در نهایت به این نتیجه می‌رسیم که آنچه در ابتدا فرض کردیم، نمی‌تواند درست باشد.

فرض می‌کنیم که $n \in \Omega(n^2)$ است. دراینصورت بایستی مقدار ثابت مثبت c و عدد صحیح غیرمنفی N وجود داشته باشد که برای تمام مقادیر $n \geq N$، $n \geq cn^2$ شود. اگر طرفین این نامساوی را به cn تقسیم کنیم، برای $n \geq N$ خواهیم داشت:

$$\frac{1}{c} \geq n$$

در حالیکه این نامساوی نمی‌تواند برای مقادیر $n > 1/c$ صحیح باشد. به عبارت دیگر، نامساوی برای همهٔ مقادیر $n \geq N$ درست نمی‌باشد. این تناقض ثابت می‌کند که n در $\Omega(n^2)$ نیست.

تعاریف دیگری نیز برای ترتیب وجود دارد که بیانگر روابطی نظیر آنچه که بین تابع n و تابع n^2 وجود دارد، می‌باشد.

تعریف برای تابع پیچیدگی مفروض $f(n)$، o(f(n)) مجموعه‌ای است از توابع پیچیدگی $g(n)$، که برای هر ثابت حقیقی مثبت c، یک عدد صحیح غیرمنفی N یافت می‌شود بطوری که برای هر $n \geq N$،

$$g(n) \leq c \times f(n)$$

اگر $g(n) \in o(f(n))$ باشد، آنگاه گوئیم که $g(n)$ یک small o (o کوچک) از $f(n)$ است. به یاد دارید که big O به این معناست که بایستی ثابت حقیقی مثبت cای برای برقراری نامساوی وجود داشته باشد (یعنی نامساوی، به ازاء برخی مقادیر حقیقی مثبت برقرار است). اما این تعریف می‌گوید که هر مقدار ثابت حقیقی مثبت c باید در نامساوی مذکور صدق کند. برای مثال، اگر $g(n) \in o(f(n))$ باشد، Nای وجود دارد بطوری که برای تمام مقادیر $n > N$ داشته باشیم:

$$g(n) \leq ۰/۰۰۰۱ \times f(n)$$

مشاهده می‌کنیم که با بزرگتر شدن مقدار n، مقدار $g(n)$ نسبت به $f(n)$ ناچیز می‌گردد. از نظر تحلیلی می‌گوئیم که اگر $g(n)$ در $o(f(n))$ باشد، در اینصورت تابع $g(n)$ نهایتاً خیلی بهتر از توابعی نظیر $f(n)$ خواهد بود. مثالهای زیر این مطلب را تشریح می‌کنند.

مثال ۱-۱۸ نشان می‌دهیم که $n \in o(n^2)$ است.

اگر $c > ۰$ باشد، آنگاه نیاز به پیدا کردن یک N داریم بطوری که برای هر $n \geq N$ داشته باشیم $n \leq cn^2$. اگر هر دو طرف نامساوی را به cn تقسیم کنیم، خواهیم داشت:

$$\frac{۱}{c} \leq n$$

بنابراین، کافی است هر $N \geq ۱/c$ را انتخاب کنیم.

توجه داشته باشید که مقدار N به مقدار ثابت c وابسته است. برای مثال، اگر $c = ۰/۰۰۰۱$ باشد، بایستی N را بزرگتر یا مساوی با ۱۰۰،۰۰۰ در نظر بگیریم. یعنی برای $n \geq ۱۰۰,۰۰۰$ خواهیم داشت:

$$n \leq ۰/۰۰۰۰۱n^2$$

مثال ۱-۱۹ می‌خواهیم نشان دهیم که n در $o(۵n)$ نیست.

با استفاده از برهان خلف این مطلب را ثابت می‌کنیم. فرض کنید که $c = ۱/۶$ است. اگر $n \in o(۵n)$ باشد، در اینصورت بایستی مقداری برای N وجود داشته باشد بطوری که برای هر $n \geq N$،

$$n \leq \frac{۱}{۶}۵n = \frac{۵}{۶}n$$

این تناقض، نبودن n در $o(۵n)$ را ثابت می‌کند.

قضیه زیر رابطه small o را با مجانبهای دیگر به خوبی تشریح می‌کند.

قضیه ۲-۱ اگر $g(n) \in o(f(n))$ باشد، در اینصورت

$$g(n) \in O(f(n)) - \Omega(f(n))$$

یعنی $g(n)$ در $O(f(n))$ است ولی در $\Omega(f(n))$ نیست.

اثبات: از آنجائیکه $g(n) \in O(f(n))$ است، لذابرای هر ثابت حقیقی مثبت c، Nای وجود دارد بطوری که برای هر $n \geq N$ داریم:

$$g(n) \leq c \times f(n)$$

بدین معنا که این نامساوی برای برخی مقادیر c صادق است. بنابراین،

$$g(n) \in O(f(n))$$

با استفاده از برهان خلف نشان می‌دهیم که $g(n)$ در $\Omega(f(n))$ نیست. اگر $g(n) \in \Omega(f(n))$ باشد، آنگاه چند ثابت حقیقی $c>0$ و برخی مقادیر N_1ای وجود دارد بطوری که به ازاء هر $n > N_1$ داریم:

$$g(n) \geq c \times f(n)$$

اما چون $g(n) \in O(f(n))$ است، لذا N_2ای وجود دارد به طوری که به ازاء هر $n \geq N_2$ داریم:

$$g(n) \leq \frac{c}{2} \times f(n)$$

هر دو نامساوی فوق بایستی برای nهای بزرگتر از N_1 و N_2 برقرار باشند. این تناقض ثابت می‌کند که $g(n)$ نمی‌تواند در $\Omega(f(n))$ باشد.

ممکن است فکر کنید که $o(f(n))$ و $O(f(n)) - \Omega(f(n))$ بایستی مجموعه یکسانی باشند، اما این درست نیست؛ زیرا توابعی وجود دارند که در $O(f(n)) - \Omega(f(n))$ بوده، ولی در $o(f(n))$ نمی‌باشند. به مثال زیر توجه کنید.

مثال ۲۰-۱ تابع $g(n)$ را درنظر بگیرید:

$$g(n) = \begin{cases} n & \text{اگر n زوج باشد} \\ 1 & \text{اگر n فرد باشد} \end{cases}$$

به عنوان تمرین نشان دهید که $g(n) \in O(n) - \Omega(n)$ است اما $g(n)$ در $o(n)$ وجود ندارد.

زمانی که توابع پیچیدگی، پیچیدگی زمانی الگوریتمهای واقعی را نشان می‌دهند، معمولاً توابعی که در $O(f(n)) - \Omega(f(n))$ هستند، در $o(f(n))$ نیز وجود دارند.

بیائید بیشتر در مورد Θ بحث کنیم. در تمرینات ثابت می‌کنیم که $g(n) \in \Theta(f(n))$ است اگر و فقط اگر $f(n) \in \Theta(g(n))$ باشد. برای مثال،

$$n^2 + 10n \in \Theta(n^2) \quad \text{و} \quad n^2 \in \Theta(n^2 + 10n)$$

این بدین معناست که Θ، توابع پیچیدگی را به مجموعه‌هایی مجزا از هم تبدیل می‌کند. ما این مجموعه‌ها را **رده‌های پیچیدگی** می‌نامیم. برای سهولت، اغلب یک رده را با ساده‌ترین عضوش نشان می‌دهیم. ردهٔ پیچیدگی قبلی توسط $\Theta(n^2)$ نشان داده می‌شود. پیچیدگی زمانی برخی از الگوریتم‌ها، به همراه n افزایش نمی‌یابد. به عنوان مثال، همانطوری که می‌دانید پیچیدگی زمانی بهترین حالت B(n) الگوریتم ۱-۱ برای هر مقدار n برابر یک می‌باشد. ردهٔ پیچیدگی که شامل چنین توابعی باشد را می‌توان به وسیله هر مقدار ثابتی نشان داد که برای سادگی، آن را با $\Theta(1)$ نمایش می‌دهیم.

برخی از ویژگی‌های مهم ترتیب که تعیین ترتیب بسیاری از توابع پیچیدگی را آسان می‌سازد، در زیر آورده شده است. آنها را بدون اثبات بیان می‌کنیم؛ چرا که اثبات برخی از این ویژگی‌ها در تمرینات نهفته شده و اثبات برخی دیگر را می‌توانید از زیربخشهای بعدی کتاب نتیجه‌گیری نمائید.

ویژگیهای ترتیب :

۱- $g(n) \in O(f(n))$ است اگر و فقط اگر $f(n) \in \Omega(g(n))$ باشد.

۲- $g(n) \in \Theta(f(n))$ است اگر و فقط اگر $f(n) \in \Theta(g(n))$ باشد.

۳- اگر $a > 1$ و $b > 1$ باشد، آنگاه $\log_a n \in \Theta(\log_b n)$ است.

این ویژگی نشان می‌دهد که همه توابع پیچیدگی لگاریتمی در یک رده پیچیدگی قرار دارند. ما این رده را با $\Theta(\lg n)$ نشان می‌دهیم.

۴- اگر $b > a > 0$ باشد، آنگاه $a^n \in o(b^n)$ است.

این ویژگی نشان می‌دهد که همه توابع پیچیدگی نمایی در یک رده پیچیدگی قرار ندارند.

۵- برای هر $a > 0$ ، $a^n \in o(n!)$ است.

این ویژگی نشان می‌دهد n! از تمامی توابع پیچیدگی نمایی بدتر می‌باشد.

۶- به رده‌های پیچیدگی زیر توجه کنید:

$$\Theta(\lg n) \quad \Theta(n) \quad \Theta(n \lg n) \quad \Theta(n^2) \quad \Theta(n^j) \quad \Theta(n^k) \quad \Theta(a^n) \quad \Theta(b^n) \quad \Theta(n!)$$

که در آن $k > j > 2$ ، $b > a > 1$ می‌باشد. اگر یک تابع پیچیدگی $g(n)$ در ردهٔ سمت چپ ردهٔ شامل $f(n)$ باشد، آنگاه $g(n) \in o(f(n))$ خواهد بود.

۷- اگر $c \geq 0$ ، $d > 0$ ، $g(n) \in O(f(n))$ و $h(n) \in \Theta(f(n))$ باشد، آنگاه

$$c \times g(n) + d \times h(n) \in \Theta(f(n))$$

مثال ۲۱-۱ ویژگی ۳ بیان می‌کند که تمامی توابع پیچیدگی لگاریتمی، در یک ردهٔ پیچیدگی قرار دارند. برای مثال،

$$\Theta(log_{\mathfrak{r}}n) = \Theta(lgn)$$

این بدین معناست که ارتباط بین lgn و $log_{\mathfrak{r}}n$ مشابه ارتباط بین $n^{\mathfrak{r}}$ و $\mathfrak{r}n^{\mathfrak{r}}+\mathfrak{d}n$ می‌باشد.

مثال ۲۲-۱ ویژگی ۶ بیان می‌کند که نهایتاً هر تابع لگاریتمی بهتر از هر چندجمله‌ای و هر چندجمله‌ای بهتر از هر تابع نمایی و هر تابع نمایی بهتر از هر تابع فاکتوریل خواهد بود. برای مثال،

$$lgn \in o(n) \qquad n^{\mathfrak{r}\circ} \in o(\mathfrak{r}^n) \qquad \mathfrak{r}^n \in o(n!)$$

مثال ۲۳-۱ ویژگیهای ۶ و ۷ می‌توانند مکرراً استفاده شوند. به عنوان مثال، می‌توانیم با بکارگیری مکرر ویژگیهای ۶ و ۷، نشان دهیم که $\mathfrak{d}n + \mathfrak{r}lgn + \mathfrak{r}\circ n\,lgn + \mathfrak{r}n^{\mathfrak{r}} \in \Theta(n^{\mathfrak{r}})$ داریم:

$$\mathfrak{r}n^{\mathfrak{r}} \in \Theta(n^{\mathfrak{r}})$$

که نتیجه می‌دهد

$$\mathfrak{r}\circ n\ lgn + \mathfrak{r}n^{\mathfrak{r}} \in \Theta(n^{\mathfrak{r}})$$

که نتیجه می‌دهد

$$\mathfrak{r}lgn + \mathfrak{r}\circ n\ lgn + \mathfrak{r}n^{\mathfrak{r}} \in \Theta(n^{\mathfrak{r}})$$

که نتیجه می‌دهد

$$\mathfrak{d}n^{\mathfrak{r}} + \mathfrak{r}lgn + \mathfrak{r}\circ n\ lgn + \mathfrak{r}n^{\mathfrak{r}} \in \Theta(n^{\mathfrak{r}})$$

در عمل ما به ویژگی‌های ترتیب مراجعه نمی‌کنیم، بلکه به سادگی می‌توانیم از عناصر با ترتیب پایین صرفنظر کنیم. اگر بتوانیم پیچیدگی زمانی یک الگوریتم را دقیقاً بدست آوریم، می‌توانیم با رد کردن عنصر با ترتیب پایین، به سادگی ترتیب آن را بدست آوریم. هر گاه این روش امکان‌پذیر نباشد، می‌توانیم برای تعیین ترتیب، به تعریف big O و Ω مراجعه کنیم. برای مثال، فرض کنید که برای الگوریتمی نتوانیم T(n) [یا A(n)، یا W(n) و یا B(n)] را به طور دقیق بدست آوریم. اگر ما بتوانیم نشان دهیم که

$$T(n) \in O(f(n)) \quad , T(n) \in \Omega(f(n))$$

آنگاه با توجه به تعاریف، می‌توانیم نتیجه بگیریم که $T(n) \in \Theta(f(n))$ است.

گاهی اوقات، نشان دادن $T(n) \in O(f(n))$ نسبتاً آسان است ولی تعیین اینکه آیا T(n) در $\Omega(f(n))$ وجود دارد یا خیر، کار مشکلی است. در اینگونه موارد ممکن است تنها به نشان دادن $T(n) \in O(f(n))$ بسنده کنیم؛ چراکه این عبارت به طور ضمنی بیان می‌کند که T(n) حداقل به خوبی برخی توابع، نظیر f(n) است. به طور مشابه، ممکن است تنها به تعیین $T(n) \in \Omega(f(n))$ قانع شویم؛ چراکه این عبارت بیان می‌کند که T(n) حداقل به بدی برخی توابع، نظیر f(n) است.

در خاتمه متذکر می‌شویم که بعضی نویسندگان بجای $f(n) \in \Theta(n^{\mathfrak{r}})$ می‌نویسند $f(n) = \Theta(n^{\mathfrak{r}})$، که هر دو به یک معنا است. همچنین مرسوم است که بجای $f(n) \in O(n^{\mathfrak{r}})$ می‌نویسند $f(n) = O(n^{\mathfrak{r}})$.

۳-۴-۱ استفاده از حد برای تعیین ترتیب

می‌خواهیم نشان دهیم که چگونه می‌توان یک ترتیب را با استفاده از حد تعیین کرد. این بخش برای کسانی است که با حد و مشتق آشنائی با این موارد در جاهای دیگر این کتاب ضروری نیست.

قضیه ۳-۱ عبارت زیر را داریم:

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \begin{cases} c \text{ implies } g(n) \in \Theta(f(n)) \text{ if } c > 0 \\ 0 \text{ implies } g(n) \in o(f(n)) \\ \infty \text{ implies } f(n) \in o(g(n)) \end{cases}$$

اثبات: اثبات این قضیه به عنوان یک تمرین آمده است.

مثال ۲۴-۱ قضیه ۳-۱ اشاره دارد به اینکه $n^2/2 \in o(n^3)$

زیرا:

$$\lim_{n \to \infty} \frac{n^2/2}{n^3} = \lim_{n \to \infty} \frac{1}{2n} = 0$$

استفاده از قضیه ۳-۱ در مثال ۲۴-۱ جالب نیست زیرا جواب آن را می‌توان براحتی و بطور مستقیم تعیین کرد.

مثال ۲۵-۱ قضیه ۳-۱ نشان می‌دهد که برای $a > 0$, $b > a$ $a^n \in o(b^n)$

زیرا

$$\lim_{n \to \infty} \frac{a^n}{b^n} = \lim_{n \to \infty} \left[\frac{a}{b}\right]^n = 0$$

حد برابر صفر است زیرا $0 < a/b < 1$ می‌باشد.

این، همان ویژگی چهارم از ویژگیهای تعریف است.

مثال ۲۶-۱ قضیه ۳-۱ اشاره دارد به اینکه برای $a > 0$: $a^n \in o(n!)$

اگر $a \leq 1$ باشد، نتیجه جزئی و ناچیز است. فرض کنید که $a > 1$ باشد. اگر n نیز به اندازه‌ای بزرگ باشد که $n/2 > 2a^4$، آنگاه داریم:

$$\frac{a^n}{n!} < \frac{a^n}{a^4 a^4 \cdots a^4} \leq \frac{a^n}{(a^4)^{n/2}} = \frac{a^n}{a^{2n}} = \left(\frac{1}{a}\right)^n$$

و چون $a > 1$ لذا $\lim_{n \to \infty} \frac{a^n}{n!} = 0$ که همان ویژگی پنجم از ویژگیهای ترتیب است.

قضیه زیر که اثبات آن در بسیاری از متون ریاضی پیدا می‌شود، فوائد قضیه ۳-۱ را زیادتر می‌کند.

قضیه ۴-۱ قانون هوپیتال

اگر $f(x)$ و $g(x)$ دو تابع مختلف با مشتقات $f'(x)$ و $g'(x)$ باشند و اگر

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

آنگاه

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

است در صورتیکه در حد سمت راست وجود داشته باشد.

قضیه ۴-۱ برای توابعی با مقادیر حقیقی می‌باشد، حال آنکه توابع پیچیدگی، توابعی از نوع متغیرهای صحیح هستند. با وجود این، در بین توابع پیچیدگی توابع بسیاری (نظیر $\lg n$، n و...) نیز مشاهده ی شوند که ازنوع متغیرهای حقیقی هستند علاوه بر این اگر تابع $f(x)$ تابعی از متغیر حقیقی x باشد، آنگاه رای عدد صحیح n داریم

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

در صورتیکه حد سمت راست وجود داشته باشد. بنابراین می‌توانیم قضیه ۴-۱ را برای تحلیل پیچیدگی (همانند مثالهای فوق) بکار ببریم.

مثال ۲۷-۱ قضایای ۴-۱ و ۳-۱ نشان می‌دهند که $\lg n \in o(n)$

زیرا:

$$\lim_{x \to \infty} \frac{\lg x}{x} = \lim_{x \to \infty} \frac{d(\lg x)/dx}{dx/dx} = \lim_{x \to \infty} \frac{1/(x \ln 2)}{1} = 0$$

مثال ۲۸-۱ قضایای ۳-۱ و ۴-۱ نشان می‌دهند که برای $a > 1, b > 1$: $\log_a n \in \theta(\log_b n)$

زیرا

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

که همان ویژگی سوم از ویژگیهای ترتیب است.

۵-۱ سازمان کلی کتاب

هم اکنون آماده‌ایم تا الگوریتم‌های مختلفی را مورد تجزیه و تحلیل قرار دهیم. بیشتر بخشها، به جای مباحث کاربردی، بر تکنیک‌ها استوارند. همانطوریکه قبلاً نیز اشاره شد، هدف از این بحث، بررسی مجموعه‌ای از روشهایی است که می‌توانند به عنوان راههای ممکن برای ورود به یک مسئله جدید مطرح شوند. فصل ۲، روشی موسوم به "تقسیم و غلبه" را مورد بررسی قرار می‌دهد. در فصل ۳، روش "برنامه‌نویسی پویا" تشریح می‌شود. در فصل ۴، "الگوریتمهای حریص" را بررسی می‌کنیم.

در فصل ۵، تکنیک "بک‌تراکینگ" (بازگشت به عقب) معرفی شده است و فصل ۶، روشی موسوم به "شاخه و حد" را مورد بررسی قرار می‌دهد. در فصلهای ۷ و ۸ به جای تجزیه و تحلیل الگوریتم‌ها، به تحلیل خود مسائل می‌پردازیم. یک نمونه از آن که تحلیل پیچیدگی محاسباتی نامیده می‌شود، حد پائین پیچیدگیهای زمانی را برای تمامی الگوریتم‌های یک مسئله بررسی می‌کند. فصل ۷ به بررسی مسئله مرتب‌سازی و فصل ۸ به بررسی مسئله جستجو می‌پردازد. فصل ۹ به یک سری مسائل خاص اختصاص داده شده است. این سری، شامل مسائلی است برای توجیه این نکته که تا به حال الگوریتمی ارائه نشده است که پیچیدگی زمانی بدترین حالت آن، بهتر از حالت نمایی باشد والبته هنوز هم کسی ثابت نکرده که ارائه چنین الگوریتمی غیر ممکن است. مطالعهٔ اینگونه مسائل، فضای نسبتاً جدید و مهیجی را در علم کامپیوتر باز کرده است. تمامی الگوریتم‌های مطرح شده در نه فصل اول کتاب، برای کامپیوترهایی ارائه شده‌اند که تنها یک رشته از دستورات را اجرا می‌کنند. با کاهش شدید قیمت سخت‌افزار کامپیوتر، پیشرفت جدیدی در توسعه کامپیوترهای موازی به وجود آمد. این کامپیوترها بیش از یک پردازنده دارند و همهٔ پردازنده‌ها می‌توانند بطور همزمان و موازی، دستورالعملها را اجرا کنند. الگوریتمهایی که برای اینگونه از کامپیوترها طراحی و ارائه می‌شوند، "الگوریتمهای موازی" نامیده می‌شوند. فصل ۱۰، مقدمه ای بر این نوع از الگوریتمها است.

تمرینات

بخش ۱-۱

۱- الگوریتمی بنویسید که بزرگترین عدد را در یک لیست (یک آرایه) n عنصری پیدا کند.

۲- الگوریتمی بنویسید که kامین عدد کوچکتر را در یک لیست عنصری پیدا کند.

۳- الگوریتمی بنویسید که تمامی زیر مجموعه‌های سه عضوی یک مجموعهٔ n عضوی را چاپ کند.اعضای این مجموعه، در لیستی که به عنوان پارامتر ورودی به الگوریتم داده شده است، قرار دارند.

۴- یک الگوریتم مرتب‌سازی درجی بنویسید که از روش جستجوی دودویی برای یافتن محل درج عنصر بعدی استفاده کند.

۵- الگوریتمی بنویسید که بزرگترین مقسوم علیه مشترک دو عدد صحیح را پیدا کند.

۶- الگوریتمی بنویسید که کوچکترین و بزرگترین عناصر یک لیست n عنصری را پیدا کند. سعی کنید که روش جستجوی شما، بیشتر از $1/5\,n$ مقایسه نداشته باشد.

۷- الگوریتمی بنویسید که تعیین کند آیا یک درخت دودویی تقریباً کامل، یک هرم (heap) است یا خیر.

بخش ۲-۱

۸- تحت چه شرایطی، جستجوی ترتیبی (الگوریتم ۱-۱) مناسب نیست ؟

۹- یک مثال عملی ارائه دهید که در آن از روش مرتب‌سازی تبادلی (الگوریتم ۳-۱) استفاده شود.

بخش ۳-۱

۱۰- عملیات مبنایی را برای الگوریتم‌های تمرینات ۱ تا ۷ مشخص نموده، کارایی این الگوریتم را مورد ارزیابی قرار دهید. اگر الگوریتمی پیچیدگی زمانی حالت معمول دارد، آن را مشخص کنید؛ در غیر اینصورت، پیچیدگی زمانی بدترین حالت الگوریتم را تعیین نمائید.

۱۱- پیچیدگی زمانی بدترین حالت، حالت میانی و بهترین حالت مرتب‌سازی درجی اصلی و مرتب‌سازی درجی تمرین ۴، که در آن از جستجوی دودویی استفاده شده است، را مشخص کنید.

۱۲- یک الگوریتم زمان-خطی بنویسید که n عدد صحیح غیرتکراری بین ۱ تا ۵۰۰ را مرتب کند.
راهنمائی: از یک آرایهٔ ۵۰۰ عنصری استفاده کنید.

۱۳- الگوریتم A، $۱۰۰n^۲$ عمل مبنایی و الگوریتم B، $۳۰۰ \ln n$ عمل مبنایی را انجام می‌دهد. به ازاء چه مقداری از n، الگوریتم B کارایی بهتری نسبت به الگوریتم A دارد؟

۱۴- برای مسئله‌ای با اندازه n، دو الگوریتم با نامهای Alg۱ و Alg۲ وجود دارند. Alg۱ در $n^۲$ میکروثانیه و Alg۲ در $۱۰۰n \log n$ میکروثانیه اجرا می‌شود. Alg۱ برای پیاده‌سازی، به ۴ ساعت کار برنامه نویسی و ۲ دقیقه زمان CPU نیاز دارد. از طرف دیگر، Alg۲ برای پیاده‌سازی، به ۱۵ ساعت کار برنامه‌نویسی و ۶ دقیقه زمان CPU نیاز دارد. اگر دستمزد برنامه‌نویسان برای هر ساعت کار، ۲۰ دلار باشد و هر دقیقه از کار CPU، ۵۰ دلار هزینه داشته باشد، یک نمونه مسئله با اندازه ۵۰۰، چند مرتبه بایستی به وسیله Alg۲ حل شود تا هزینهٔ این کار را توجیه کند؟

بخش ۴-۱

۱۵- نشان دهید که $f(n) = n^۲ + ۳n^۳ \in \Theta(n^۳)$ است. به عبارتی با استفاده از تعاریف O و Ω نشان دهید که f(n) هم در $O(n^۳)$ و هم در $\Omega(n^۳)$ قرار دارد.

۱۶- با استفاده از تعاریف O و Ω نشان دهید که $۰٫۰۱n^۲ + ۲۰n \in O(n^۳)$، اما $۰٫۰۱n^۲ + ۲۰n \notin \Omega(n^۳)$.

۱۷- با استفاده از ویژگیهای ترتیب در بخش ۲-۴-۱ نشان دهید که

$$۵n^۵ + ۴n^۴ + ۶n^۳ + ۲۰۱^۲ + n + ۷ \in \Theta(n^۵)$$

۱۸- فرض کنید $P(n) = a_k n^k + a_{k-۱} n^{k-۱} + \ldots + a_۱ n + a_۰$، که $a_k > ۰$ است. با استفاده از ویژگیهای ترتیب در بخش ۲-۴-۱ نشان دهید که

$$p(n) \in \Theta(n^k)$$

۱۹- توابع زیر را در رده‌های پیچیدگی دسته‌بندی کنید:

$$n \ln n \quad (\lg n)^۲ \quad ۵n^۲ + ۷n \quad n^{۵/۲} \quad n! \quad ۲^{n!} \quad ۴^n \quad n^n \quad n^n + \ln n$$
$$۵^{\lg n} \quad (\lg !) \quad (\lg n)! \quad \sqrt{n} \quad e^n \quad ۸n + ۱۲ \quad ۱۰^n + n^{۲۰}$$

۲۰- ویژگیهای ۱، ۲، ۶ و ۷ از ویژگیهای ترتیب بخش ۲-۴-۱ را ثابت کنید.

۲۱- ویژگیهای بازتابی، تقارن و تراگذاری را برای مفاهیم O، Θ، Ω و o بررسی کنید.

۲۲- فرض کنید کامپیوتری دارید که برای حل نمونه مسئله‌ای با اندازهٔ $n = ۱۰۰۰$ به یک دقیقه زمان نیاز دارد. اگر کامپیوتر جدیدی خریدید که ۱۰۰۰ برابر سریعتر از قبلی عمل می‌کند، در یک دقیقه، یک نمونه با چه اندازه ورودی را می‌تواند حل کند؛ با فرض اینکه پیچیدگی‌های زمانی $T(n)$ زیر برای الگوریتم ما وجود دارند:

(a) $T(n) \in \Theta(n)$

(b) $T(n) \in \Theta(n^۲)$

(c) $T(n) \in \Theta(۱.^n)$

۲۳- صحت جملات زیر را بررسی کنید:

(a) $lg\, n \in O(n)$

(b) $۲^۲ \in \Omega(۵^{ln\ n})$

(c) $n \in O(n\ lg\ n)$

(d) $n\, lg\, n \in O(n^۲)$

(e) $lg^۳ n \in o(n^{۰/۵})$

تمرینات اضافی

۲۵- پیچیدگی زمانی $T(n)$ حلقه‌های تودرتوی زیر چیست؟ فرض کنید که n توانی از ۲ است.

```
   ⋮
for(i = 1; i <= n; i++){
   j = n;
   while(j >= 1){
      <while بدنه حلقه>        // به Θ(n) نیاز دارد.
      j = ⌊j / 2⌋;
   }
}
   ⋮
```

۲۶- پیچیدگی زمانی $T(n)$ حلقه‌های تودرتوی زیر چیست؟ فرض کنید که n توانی از ۲ است.

```
   ⋮
i = n;
while(i >= 1){
   j = i;
   while(j <= n){
      <بدنه حلقهٔ while درونی>        // به Θ(n) نیاز دارد.
      j = 2 * j;
   }
   i = ⌊i / 2⌋;
}
   ⋮
```

۲۷- الگوریتمی برای مسئله زیر ارائه دهید و پیچیدگی زمانی آن را تعیین کنید. یک لیست شامل n عنصر مثبت مجزا مفروض است، لیست را به دو زیرلیست تقسیم کنید بطوری که اندازهٔ هر زیرلیست برابر n/۲ باشد و اختلاف بین مجموع اعداد صحیح دو زیرلیست، حداکثر شود. می‌توانید فرض کنید که n مضربی از ۲ است.

۲۸- یک الگوریتم $\Theta(n \lg n)$ ارائه دهید که باقیماندهٔ تقسیم x^n بر P را محاسبه کند. می‌توانید n را توانی از ۲ درنظر بگیرید ($n = ۲^k$).

۲۹- توضیح دهید چه توابعی در مجموعه‌های زیر قرار می‌گیرند؟

(a) $n^{o(1)}$ (b) $O(n^{o(1)})$ (c) $O(O(n^{o(1)})$

۳۰- نشان دهید که تابع $f(n) = \left| n^{۲} \sin n \right|$ نه در O(n) قرار دارد و نه در $\Omega(n)$

۳۱- الگوریتمی برای مسئله زیر بنویسید. یک لیست با n عدد صحیح مثبت مجزا مفروض است. آن را طوری به دو زیر لیست به اندازه‌های n/۲ تقسیم کنید که اختلاف بین مجموعه اعداد صحیح دو زیرلیست، حداقل شود. پیچیدگی زمانی الگوریتم را تعیین کنید. می‌توانید n را مضربی از ۲ درنظر بگیرید.

۳۲- می‌دانیم که الگوریتم ۷-۱ (nامین عنصر فیبوناچی، تکرار) در n، به صورت خطی است. آیا این الگوریتم یک الگوریتم زمان-خطی می‌باشد؟ در این الگوریتم، n یک ورودی است و تعداد بیتهایی که برای کد کردن n بکار می‌روند، اندازه ورودی می‌باشند. نشان دهید که الگوریتم ۷-۱، برحسب اندازه ورودی، به صورت زمان-نمایی است. همچنین نشان دهید که هر الگوریتمی که برای محاسبهٔ nامین عنصر فیبوناچی نوشته شود، باید یک الگوریتم زمان-نمایی باشد زیرا اندازهٔ خروجی به صورت نمایی از اندازه ورودی است. در بخش ۲-۹، پیچیدگی زمانی الگوریتم ۶-۱ (عنصر nام فیبوناچی، بازگشتی) بر اساس اندازه ورودی آن تعیین می‌شود.

فصل ۲

تقسیم و غلبه (Divid-and-Conqure)

اولین روش طراحی الگوریتم‌ها، موسوم به تقسیم و غلبه، از استراتژی شگرف ناپلئون در جنگ استرلیتز در دوم سامبر ۱۸۰۵ الگوبرداری شده است. ارتش متشکل از نیروهای اتریش و روسیه بود که حدود ۱۵۰۰۰ سرباز بیش از سپاه ناپلئون نیرو داشت. نیروهای اتریشی- روسی در تدارک حمله به جناح راست ارتش فرانسه بودند که ناپلئون با پیش‌بینی حمله نیروهای متخاصم، سربازانش را به سمت مرکز نیروهای دشمن هدایت کرده و آنها را به دو قسمت تقسیم نمود. به دلیل عدم توانایی دو نیرو در غلبه بر ناپلئون، هر کدام از جناحهای دشمن متحمل خسارات سنگینی شده و ناگزیر به عقب نشینی شدند. ناپلئون توانست با تقسیم ارتش بزرگ به دو سپاه کوچکتر و غلبه بر هر کدام از این دو سپاه، بر ارتش بزرگ اتریشی- روسی پیروز شود.

روش تقسیم و غلبه، این استراتژی را در حل نمونه‌ای از یک مسئله به خدمت گرفت. بدین صورت که یک نمونه از یک مسئله را به دو یا چند قسمت کوچکتر تقسیم می‌کند. قسمتهای کوچکتر، معمولاً نمونه‌هایی از مسئله اصلی هستند. اگر جواب نمونه‌های کوچکتر به راحتی محاسبه شود، می‌توان جواب نمونه اصلی را با ترکیب این جوابها بدست آورد. اما اگر نمونه‌های کوچکتر هنوز آنقدر بزرگ هستند که

به سادگی حل نشوند، می‌توان آنها را به نمونه‌های کوچکتری تقسیم نمود. این فرآیند تقسیم نمونه‌ها، تا آنجا ادامه می‌یابد که برای هر نمونه کوچک بتوان جوابی را به سهولت بدست آورد.

روش تقسیم و غلبه، یک روش **بالا به پائین** است. بدینصورت که جواب یک نمونه سطح بالا از یک مسئله، با پائین رفتن و بدست آوردن جواب نمونه‌های کوچکتر حاصل می‌شود. شاید شما این روش را همان روشی بدانید که توسط روالهای بازگشتی به کار گرفته می‌شود. به خاطر داشته باشید که هنگام نوشتن روالهای بازگشتی، شخص در سطح حل مسئله فکر می‌کند و به سیستم اجازه می‌دهد که با استفاده از ساختار داده‌ای پشته، به جزئیات بدست آوردن جواب بپردازد. ما نیز هنگام تشریح یک الگوریتم تقسیم و غلبه، در همین سطح فکر کرده و روالها را به صورت بازگشتی می‌نویسیم. بعدها می‌توانیم در صورت امکان، با استفاده از تکرار و بدون بکارگیری روالهای بازگشتی، نسخه کارآمدتری از یک الگوریتم ارائه نماییم. اینک با ارائه چند مثال، به معرفی روش تقسیم و غلبه می‌پردازیم. اولین مثال، جستجوی دودویی است.

۱-۲ جستجوی دودویی (Binary Search)

در بخش ۱-۲، الگوریتم جستجوی دودویی را با استفاده از روش تکرار بیان نمودیم (الگوریتم ۵-۱). در اینجا، شرح بازگشتی آن را ارائه می‌دهیم چرا که بازگشت نمایانگر روش بالا به پائین است که در تقسیم و غلبه بکار گرفته می‌شود. همانطوریکه گفته شد، جستجوی دودویی برای جستجوی کلید x در یک آرایهٔ مرتب غیرنزولی، آن را با عنصر میانی آرایه مقایسه می‌کند. اگر این دو با هم مساوی باشند، الگوریتم پایان می‌یابد. در غیر اینصورت، آرایه به دو آرایهٔ کوچکتر (زیرآرایه) تقسیم می‌شود بطوری که یک زیرآرایه، شامل همه عناصر سمت چپ عنصر میانی و زیرآرایه دیگر، شامل همه عناصر سمت راست آن می‌باشد. اگر x کوچکتر از عنصر میانی باشد، این روند را برای زیرآرایه چپ بکار می‌گیریم. در غیر اینصورت، زیرآرایه راست مورد جستجو قرار خواهد گرفت. در ادامه، x با عنصر میانی زیرآرایه مورد نظر مقایسه می‌شود. اگر این دو مساوی بودند، الگوریتم حل شده است؛ وگرنه زیرآرایه به دو زیرآرایهٔ دیگر تقسیم می‌شود. این روند تا زمانی ادامه می‌یابد که x پیدا شود و یا اینکه مشخص شود x در آرایه موجود نیست. مراحل جستجوی دودویی را می‌توان به صورت زیر خلاصه نمود:

اگر x با عنصر میانی برابر باشد، خارج شوید. در غیر اینصورت،

۱- آرایه را به دو زیرآرایه مساوی تقسیم کنید. اگر x از عنصر میانی کوچکتر باشد، زیرآرایه چپ و در غیر اینصورت زیرآرایهٔ راست را انتخاب نمائید.

۲- زیرآرایه را حل (غلبه) کنید با تعیین این نکته که آیا x در زیرآرایه وجود دارد یا خیر. اگر زیرآرایه به اندازه کافی کوچک نباشد، از بازگشت برای انجام این کار استفاده کنید.

۳- جواب آرایه را از جواب زیرآرایه بدست آورید.

جستجوی دودویی، ساده‌ترین نوع الگوریتم تقسیم و غلبه است زیرا در آن، هر نمونه به یک نمونهٔ کوچکتر تقسیم می‌شود. بنابراین، هیچ ترکیبی از جوابها وجود ندارد. به عبارتی دیگر، جواب نمونهٔ اصلی همان جواب نمونهٔ کوچکتر است. مثال زیر، جستجوی دودویی را نشان می‌دهد.

مثال ۱-۲ فرض کنید x = ۱۸ و آرایه زیر را در اختیار داریم:

۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱	۱۲	۱۳	۱۴
۱۰	۱۲	۱۳	۱۴	۱۸	۲۰	۲۵	۲۷	۳۰	۳۵	۴۰	۴۵	۴۷	

عنصرمیانی

۱- آرایه را تقسیم می‌کنیم : از آنجائیکه x < ۲۵ است، لذا بایستی زیرآرایهٔ چپ را جستجو کنیم. یعنی

۱	۲	۳	۴	۵	۶	۷
۱۰	۱۲	۱۳	۱۴	۱۸	۲۰	

۲- زیرآرایه را با تعیین اینکه آیا x در آن وجود دارد یا خیر، حل می‌کنیم. (این کار توسط تقسیم بازگشتی زیرآرایه انجام می‌شود) :

بله، x در زیرآرایه وجود دارد.

۳- جواب آرایه را از جواب زیرآرایه بدست می‌آوریم :

بله، x در آرایه وجود دارد.

در مرحلهٔ ۲، ما بسادگی فرض کردیم که جواب زیرآرایه قابل دسترسی بوده است و در مورد جزئیات چگونگی بدست آمدن جواب بحث نکردیم چرا که می‌خواستیم جواب را تنها در سطح حل مسئله نشان دهیم. بطور کلی، جهت ارائه الگوریتم بازگشتی بر روی یک مسئله، نیازمند موارد زیر هستیم:

- طرح یک روش، جهت دستیابی به جواب نمونه اصلی توسط جواب یک یا چند نمونه کوچکتر.
- تعیین شرط یا شرایط نهایی که نمونه یا نمونه‌های کوچکتر، به سهولت قابل حل باشند.
- تعیین جواب در شرط یا شرایط نهایی.

در این موارد نیازی نیست که به چگونگی بدست آوردن جواب بپردازیم. در واقع، نگرانی از همین جزئیات است که گاهی طرح و توسعه یک الگوریتم بازگشتی پیچیده را مختل می‌کند. شکل ۱ - ۲، مراحل جستجوی دودویی توسط یک انسان را نشان می‌دهد.

الگوریتم ۱-۲ جستجوی دودویی (بازگشتی)

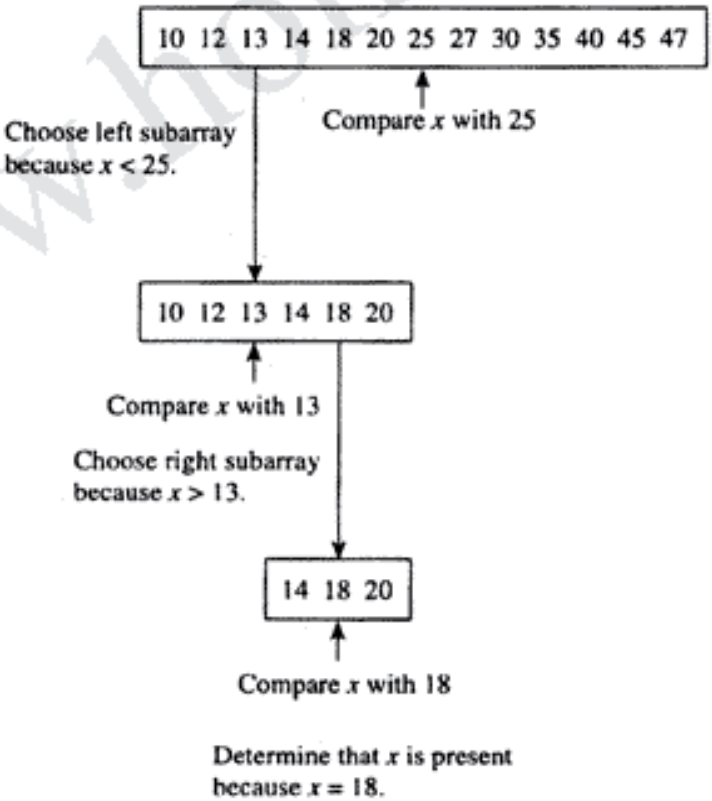
مسئله: تعیین کنید که آیا x در آرایه مرتب S با اندازه ورودی n وجود دارد یا خیر؟

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S با شاخصهایی از ۱ تا n (مرتب شده به صورت غیرنزولی)، کلید x

خروجی: location، موقعیت x در آرایه S (صفر، اگر x در S نباشد).

```
index locatoin (index low, index high)
{
   index mid;
   if (low > high)
      return 0;
   else{
      mid = ⌊(low + high)/۲⌋;
      if (x == S[mid])
         return mid;
     else if (x < S[mid])
         return location(low, mid-1);
     else
         return location (mid+1, high);
   }
}
```

توجه داشته باشید که n، s و x پارامترهای تابع Location نیستند زیرا در پایان تمامی فراخوانی‌های بازگشتی، بدون تغییر می‌مانند. در این کتاب، تنها متغیرهایی را به عنوان پارامترهای بازگشتی معرفی می‌کنیم که مقدارشان توسط فراخوانی‌های بازگشتی تغییر می‌یابد. دو دلیل برای این کار وجود دارد.



شکل ۱-۲ مراحلی که توسط انسان هنگام جستجوی دودویی انجام می‌شود. (توجه: x = ۱۸)

اول اینکه، تشریح روالهای بازگشتی با درهم ریختگی و سردرگمی همراه نخواهدبود و دوم آنکه، در پیاده‌سازی واقعی یک روال بازگشتی، یک کپی جدید از هر متغیر موجود در روال در فراخوانی بازگشتی ایجاد می‌شود. لذا اگر مقدار متغیر تغییر نکند، عمل کپی غیرضروری خواهد بود و اگر متغیر مورد نظر از نوع آرایه باشد، انجام این عمل بسیار پرهزینه می‌گردد. یک راه برای جلوگیری از این امر، استفاده از پارامترهای ارجاعی (ارجاع توسط آدرس) است. توجه دارید که اگر زبان بکارگیرندهٔ الگوریتم C++ باشد، یک آرایه به طور پیش فرض، به صورت پارامترهای ارجاعی تعریف شده است و تنها با استفاده از کلمهٔ کلیدی const می‌توانیم آرایه را طوری تعریف کنیم که محتوایش تغییر نکند. به هر ترتیب، بروز این تغییرات موجب سردرگمی و احتمالاً کاهش وضوح و شفافیت الگوریتم خواهد شد.

الگوریتم‌های بازگشتی بسته به زبان بکارگیرندهٔ آن می‌توانند به روشهای متعددی توسعه یابند. به عنوان مثال، برای بکارگیری آنها در C++ می‌توانیم همه پارامترها را به روال بازگشتی ارسال کنیم یا اینکه می‌توانیم در آنها از کلاس‌ها استفاده کنیم و یا اینکه پارامترهایی که در طی فراخوانی‌های بازگشتی تغییر نمی‌کنند را به صورت سراسری تعریف نمائیم. چگونگی انجام مورد اخیر را به موقع بیان خواهیم کرد. اگر S و x را به صورت سراسری تعریف کنیم و n تعداد عناصر موجود در S باشد، آنگاه فراخوانی سطح بالای تابع Location در الگوریتم ۱-۲ چنین خواهد بود:

```
Locationout = Location(1, n);
```

ازآنجائیکه نمونهٔ بازگشتی جستجوی دودویی، از دنباله-بازگشت (که در آن هیچ عملیاتی بعد از فراخوانی بازگشت انجام نمی‌شود) استفاده می‌کند، لذا تهیه یک نسخه تکرار از الگوریتم، آنچنانکه در بخش ۲-۱ انجام دادیم، آسانتر است. همانطوریکه قبلاً نیز گفته شد، ما در حال نوشتن یک نسخهٔ بازگشتی هستیم؛ چرا که بازگشت، بوضوح نمایانگر فرآیند تقسیم و غلبه با تقسیم یک نمونه به نمونه‌های کوچکتر است. به هر حال، این از مزایای زبانهایی نظیر C++ است که می‌توان دنباله-بازگشت را با تکرار جایگزین نمود و مهمتر از همه اینکه با این کار می‌توانیم با حذف پشته از ساختار الگوریتم، در حجم زیادی از حافظه صرفه جویی کنیم. می‌دانید که وقتی یک روال، روال دیگری را فرا می‌خواند، لازم است که تمامی اطلاعات و نتایج مربوط به روال اول با انجام عمل push، در پشتهٔ رکوردهای فعال‌سازی ذخیره و نگهداری شود. اگر روال دوم نیز روال دیگری را فراخوانی کند، تمامی اطلاعات و نتایج این روال در پشته قرار می‌گیرد و الی آخر. هنگامی که کنترل به روال فراخواننده باز می‌گردد، رکورد فعال‌سازی مربوط به آن روال، با انجام pop از پشته خارج شده و اجرای دستورات بعدی، با نتایج حاصله صورت می‌گیرد. در یک روال بازگشتی، تعداد رکوردهای فعال سازی push شده به پشته، توسط عمق فراخوانی بازگشتی تعیین می‌شود. در جستجوی دودویی، پشته به عمقی می‌رسد که در بدترین حالت تقریباً برابر $\lg n+1$ است. دلیل دیگری که برای جایگزینی دنباله-بازگشت توسط تکرار وجود دارد این است که الگوریتم تکرار، سریعتر (اما فقط با یک ضریب ثابت) از الگوریتم بازگشتی است زیرا از هیچ پشته‌ای جهت ذخیره‌سازی اطلاعات استفاده نمی‌کند. از طرف دیگر چون اکثر زبانهای LISP جدید در مرحله کامپایل،

دنباله-بازگشت را به تکرار تبدیل می‌کنند، لذا در آنجا هیچ دلیلی برای جایگزینی دنباله-بازگشت توسط تکرار وجود ندارد.

جستجوی دودویی دارای پیچیدگی زمانی حالت معمول نیست. بنابراین، الگوریتم را از لحاظ پیچیدگی زمانی بدترین حالت، مورد بررسی قرار می‌دهیم. البته این مورد را در بخش ۲-۱ به طور صوری نشان دادیم. اگرچه تحلیل زیر به الگوریتم ۱-۲ تعلق دارد، ولی با الگوریتم ۵-۱ نیز مرتبط است. اگر با روشهای حل معادلات بازگشتی آشنایی ندارید، قبل از هر اقدام، ضمیمه B را مطالعه کنید.

تحلیل پیچیدگی زمانی بدترین حالت الگوریتم ۱-۲ (جستجوی دودویی، بازگشتی)

در جستجوی یک آرایه، پرهزینه‌ترین عمل مقایسهٔ عنصر مورد جستجو با یک عنصر آرایه است. بنابراین،

عمل مبنایی: مقایسه x با S[mid].

اندازه ورودی: n، تعداد عناصر آرایه.

در ابتدا حالتی را بررسی می‌کنیم که در آن n توانی از ۲ است. در هر فراخوانی تابع Location، دو مقایسه بین x و S[mid] وجود دارد (به استثنای زمانی که این دو با هم مساویند). به هر حال، همانطوری که در تحلیل صوری جستجوی دودویی در بخش ۲-۱ بحث شد، می‌توانیم فرض کنیم که در هر فراخوانی تابع، تنها یک مقایسه انجام می‌شود چرا که با بکارگیری یک زبان اسمبلر کارا، چنین امری امکان‌پذیر است. (طبق اشاره‌ای که در بخش ۳-۱ داشتیم، معمولاً فرض می‌کنیم که عمل مبنایی به صورت کاراترین حالت ممکن پیاده سازی می‌شود.)

در بخش ۲-۱ گفتیم که یکی از بدترین حالتها زمانی رخ می دهد که x از تمامی عناصر آرایه بزرگتر باشد. اگر n توانی از ۲ و x از تمامی عناصر آرایه بزرگتر باشد، آنگاه هر نمونهٔ بازگشتی، نمونه را دقیقاً به نصف کاهش می دهد. برای مثال، اگر n = ۱۶ باشد، آنگاه $mid = \lfloor (1 + 16)/2 \rfloor = 8$ و چون x از تمامی عناصر آرایه بزرگتر است، لذا هشت عنصر بالایی، به عنوان ورودی اولین فراخوانی بازگشتی در نظر گرفته می‌شوند و به همین ترتیب، چهار عنصر بالایی، به عنوان ورودی دومین فراخوانی بازگشتی در نظر گرفته می‌شوند و الی آخر. دراینصورت بازگشت زیر را داریم :

$$W(n) = W(n/2) + 1$$

مقایسه در سطح بالا — تعداد مقایسات در فراخوانی بازگشتی

اگر n = ۱ و x از آرایه تک عنصری بزرگتر باشد، تنها یک مقایسه بین x و عنصر آرایه وجود خواهد داشت. در اینجا شرط نهایی درست است بدین معنا که دیگر مقایسه‌ای انجام نخواهد شد. بنابراین، W(۱) = ۱ است. بازگشت زیر را داریم:

$$W(n) = W\left(\frac{n}{۲}\right) + ۱ \qquad n \text{ توانی از } ۲ \text{ ، } n > ۱$$
$$W(۱) = ۱$$

این بازگشت در مثال ۱-B از ضمیمه B حل شده است، بدینصورت که

$$W(n) = lg\, n + ۱.$$

اگر n را به توانی از ۲ محدود نکنیم، خواهیم داشت:

$$W(n) = \lfloor lg\, n \rfloor + ۱ \in \Theta(lg\, n),$$

که نماد $\lfloor y \rfloor$ به معنای بزرگترین عدد صحیح کوچکتر یا مساوی y است. مثلاً $\lfloor ۳/۲۵ \rfloor = ۳$

۲-۲ مرتب‌سازی ادغامی (MergeSort)

ادغام فیزیکی از فرآیندهای مرتبط با مرتب‌سازی است. با **ادغام دوتایی** می‌توانیم دو آرایهٔ مرتب شده را به یک آرایهٔ مرتب تبدیل کنیم. به عنوان مثال، برای مرتب‌سازی یک آرایه ۱۶ عنصری، ابتدا آن را به دو زیرآرایهٔ ۸ عنصری تقسیم کرده، سپس هر یک از آنها را مرتب می‌کنیم و در نهایت، برای تولید یک آرایهٔ مرتب، آن دو را با هم ادغام می‌کنیم. بطور مشابه، هر زیرآرایه به اندازه ۸ می‌تواند به دو زیرآرایه به اندازه ۴ تقسیم شود. آنگاه این دو زیرآرایه، مرتب شده و با هم ادغام می‌شوند. در نهایت اندازه زیرآرایه‌ها به یک می‌رسد. پرواضح است که آرایهٔ تک عنصری، به خودی خود مرتب است. به این روش، **مرتب‌سازی ادغامی** گوئیم. بطور کلی، مراحلی که مرتب‌سازی ادغامی برای یک آرایه n عنصری (برای سهولت کار، n را توانی از ۲ فرض می‌کنیم) طی می‌کند، به صورت زیر است:

۱- تقسیم آرایه به دو زیرآرایه که هر کدام دارای n/۲ عنصر هستند.

۲- حل (غلبه) هر زیرآرایه با مرتب کردن آن. اگر آرایه به اندازه کافی کوچک نباشد، از بازگشت برای انجام این کار استفاده می‌کنیم.

۳- ادغام زیرآرایه‌های مرتب شده جهت تولید یک آرایه مرتب .

مثال ۲-۲ فرض کنید یک آرایه شامل عناصری به ترتیب زیر باشد:

۲۲ ۱۵ ۱۳ ۲۵ ۲۰ ۱۲ ۱۰ ۲۷

۱- تقسیم آرایه:

۲۲ ۱۵ ۱۳ ۲۵ و ۲۰ ۱۲ ۱۰ ۲۷

۲- مرتب‌سازی هر زیرآرایه:

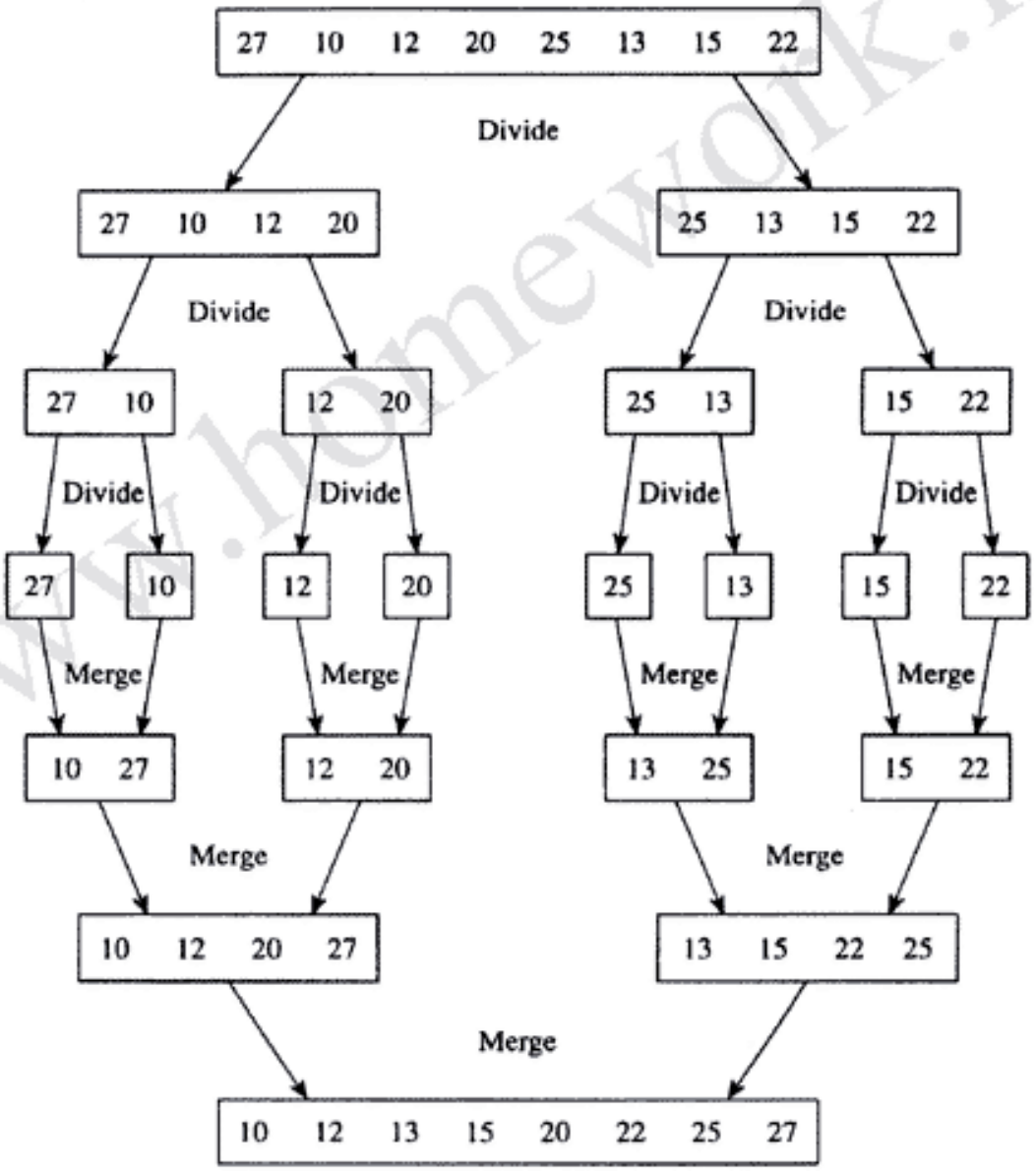
۲۵ ۲۲ ۱۵ ۱۳ و ۲۷ ۲۰ ۱۲ ۱۰

۳- ادغام زیرآرایه‌ها:

۲۷ ۲۵ ۲۲ ۲۰ ۱۵ ۱۳ ۱۲ ۱۰

ما در مرحله ۲، در سطح مسئله فکر می‌کنیم و فرض می‌کنیم که جواب زیرآرایه‌ها در دسترس هستند. برای روشن شدن مطلب، به شکل ۲-۲ که نشانگر مراحل انجام مرتب‌سازی ادغامی توسط انسان است، توجه کنید. شرط پایانی زمانی است که اندازه زیرآرایه به ۱ می‌رسد. در آن هنگام است که ادغام زیرآرایه‌ها آغاز می‌شود.

برای بکارگیری مرتب‌سازی ادغامی، به الگوریتمی نیازمندیم که دو آرایه مرتب شده را با هم ادغام کند. ابتدا الگوریتم مرتب‌سازی ادغامی را بیان می‌کنیم.



شکل ۲-۲ مراحلی که توسط انسان در هنگام مرتب‌سازی ادغامی انجام می‌شود.

الگوریتم ۲-۲ مرتب‌سازی ادغامی (Mergesort)

مسئله: یک آرایه n کلیدی را به صورت غیرنزولی مرتب کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S با شاخصهایی از ۱ تا n.

خروجی: آرایه S شامل کلیدهایی مرتب به صورت غیرنزولی.

```
void mergesort (int n, Keytype S[ ])
{
   const int h= ⌊ n/2 ⌋ ; , m = n – h;
   keytype U[l...h], V[l...m];
   if (n > 1){
      copy S[h+l] through S[h] to U[l] through U[h];
      mergesort (h, U);
      mergesort (m, V);
      mergesort (h, m, U, V, S);
   }
}
```

قبل از تحلیل الگوریتم Morgesort باید الگوریتمی را تحلیل کنیم که دو آرایه مرتب را با هم ادغام می‌کند.

الگوریتم ۲-۳ ادغام(Merge)

مسئله: دو آرایهٔ مرتب شده را به صورت یک آرایه مرتب با هم ادغام کنید.

ورودی: اعداد صحیح مثبت h و m، آرایه‌ای مرتب از کلیدها U با شاخصهایی از ۱ تا h، آرایه‌ای مرتب از کلیدها V با شاخصهایی از ۱ تا m.

خروجی: یک آرایهٔ مرتب S با شاخصهایی از ۱ تا h+m شامل کلیدهای موجود در U و V.

```
void merge (int h, int m, const keytype U[ ],
                          const keytype V[ ],
                          const keytype S[ ])
{
   index i, j, k;
   i=1; j=1; k=1;
   while (i <= h && j <= m){
      if (U[i] < V[j]){
         S[k] = U[i];
         i++;
      }
      else{
        S[k]=V[i];
        j++;
      }
      k++;
```

```
    if (i > h)
        copy V[i] through V[m] to S[k] through S[h + m];
    else
        copy U[i] through U[h] to S[k] through S[h + m];
}
```

جدول ۱-۲، چگونگی انجام عمل merge را برای ادغام دو آرایه ۴ عنصری نشان می‌دهد.

تحلیل پیچیدگی زمانی بدترین حالت الگوریتم ۳-۲ (Merge)

همانطوریکه در بخش ۳-۱ بیان شد، در مورد الگوریتم هایی که با مقایسه کلیدها عمل مرتب‌سازی را انجام می‌دهند، هر یک از دستورالعملهای مقایسه و انتساب می‌توانند بعنوان عمل مبنایی درنظر گرفته شوند. در این فصل، دستورالعمل مقایسه و در فصل ۷، دستورالعمل انتساب را به عنوان عمل مبنایی در نظر می‌گیریم. در این الگوریتم، تعداد مقایسات به h و m بستگی دارد. لذا داریم:

عمل مبنایی: مقایسه U[i] با V[i].

اندازه ورودی: h و m، تعداد عناصر موجود در هر یک از دو آرایه ورودی.

بدترین حالت زمانی اتفاق می‌افتد که از حلقه خارج می‌شویم، چرا که در این حالت، یکی از شاخصها (مثلاً i) به نقطه خروجی‌اش (h) رسیده، در حالیکه شاخص دیگر (j) تنها به m-۱، یعنی یکی کمتر از نقطه‌خروجی‌اش (m) می‌رسد. برای مثال، این حالت می‌تواند زمانی اتفاق بیفتد که ابتدا m-۱ عنصر اول V در S جایگزین شده، سپس تمامی h عنصر U در S قرار گرفته باشد. اینجاست که از حلقه خارج می‌شویم؛ چرا که i مساوی h شده است. بنابراین،

$$W(h,m) = h + m - 1$$

جدول ۱-۲ یک مثال از ادغام دو آرایه U و V به آرایهٔ S *

k	U				V				S (Result)							
1	**10**	12	20	27	**13**	15	22	25	10							
2	10	**12**	20	27	**13**	15	22	25	10	12						
3	10	12	**20**	27	**13**	15	22	25	10	12	13					
4	10	12	**20**	27	13	**15**	22	25	10	12	13	15				
5	10	12	**20**	27	13	15	**22**	25	10	12	13	15	20			
6	10	12	20	**27**	13	15	**22**	25	10	12	13	15	20	22		
7	10	12	20	**27**	13	15	22	**25**	10	12	13	15	20	22	25	
—	10	12	20	27	13	15	22	25	10	12	13	15	20	22	25	27 ← Final values

*The items compared are in boldface.

تحلیل پیچیدگی بدترین حالت الگوریتم ۲-۲ (مرتب‌سازی ادغامی)

عمل مبنایی، مقایسه‌ای است که در روال merge قرار دارد. به دلیل اینکه تعداد مقایسات با h و m افزایش می‌یابد و h و m نیز با n افزایش پیدا می‌کنند، لذا

عمل مبنایی: دستورالعمل مقایسه‌ای که در روال merge قرار دارد.

اندازه ورودی: n، تعداد عناصر آرایه S.

تعداد کل مقایسات برابراست با مجموع تعداد مقایسات در فراخوانی بازگشتی mergesort با ورودی U، تعداد مقایسات در فراخوانی بازگشتی mergesort با ورودی V و تعداد مقایسات در فراخوانی سطح بالای merge. بنابراین،

$$W(n) = \underbrace{W(h)}_{\text{مدت زمان مرتب‌سازی U}} + \underbrace{W(m)}_{\substack{\text{مدت زمان} \\ \text{مرتب‌سازی V}}} + \underbrace{h+m-1}_{\text{مدت زمان ادغام}}$$

ابتدا حالتی را بررسی می‌کنیم که در آن n توانی از ۲ است. در این حالت،

$$h = \lfloor \frac{n}{2} \rfloor = \frac{n}{2}$$

$$m = n - h = n - \frac{n}{2} = \frac{n}{2}$$

$$h + m = \frac{n}{2} + \frac{n}{2} = n$$

بنابراین برای W(n) داریم

$$W(n) = W\left(\frac{n}{2}\right) + W\left(\frac{n}{2}\right) + n - 1$$

$$= 2W\left(\frac{n}{2}\right) + n - 1$$

هرگاه اندازهٔ ورودی یک شود، شرط نهایی برقرار شده و هیچ ادغامی صورت نمی‌گیرد. بنابراین، W(1) برابر صفر خواهد شد. بازگشت زیر را ارائه داده‌ایم:

$$W(n) = 2W\left(\frac{n}{2}\right) + n - 1 \qquad n>1 \text{ ، } n \text{ توانی از ۲ است}$$
$$W(1) = 0$$

این بازگشت، در مثال ۱۹- B، به صورت زیر حل شده است:

$$W(n) = n \lg n - (n-1) \in \Theta(n \lg n)$$

وقتی که n توانی از ۲ نباشد، تابع پیچیدگی برابراست با

$$W(n) = W\left(\lfloor \frac{n}{2} \rfloor\right) + W\left(\lceil \frac{n}{2} \rceil\right) + n - 1$$

که نماد ⌈y⌉ نشانگر کوچکترین عدد صحیح بزرگتر یا مساوی y و نماد ⌊y⌋ نشانگر بزرگترین عدد صحیح کوچکتر یا مساوی y می‌باشند. تحلیل این حالت به دلیل وجود جزء صحیح بالا و پائین، بسیار مشکل است. به هر حال، با استفاده از خاصیت استقراء، نظیر آنچه که در مثال B-۲۵ از ضمیمه B آمده است، می‌توان نشان داد که W(n) غیرنزولی است. بنابراین، براساس قضیه B-۴ داریم:

$$W(n) \in \Theta(n \lg n)$$

یک **مرتب‌سازی درون‌مکانی**، الگوریتمی است که به فضای اضافی جهت ذخیره‌سازی ورودی نیاز ندارد. الگوریتم ۲-۲، یک مرتب‌سازی درون‌مکانی نیست زیرا علاوه بر ورودی آرایه S از آرایه‌های U و V نیز استفاده می‌کند. گر U و V به عنوان پارامترهای متغیر (پارامترهای ارجاعی یا آدرس) در روال merge تعریف شده باشند، دیگر لزومی به تهیه کپی دوم از این آرایه‌ها به هنگام فراخوانی merge وجود نخواهد داشت. با وجود این، هر زمان که mergesort فراخوانی می‌شود، آرایه‌های جدید U و V ایجاد خواهند شد. مجموع تعداد عناصر این دو آرایه در سطح بالا برابر n است. این مجموع در فراخوانی بازگشتی سطح بالا، تقریباً برابر n/۲، در سطح بعدی تقریباً برابر n/۴ و در حالت کلی، در هر سطح بازگشتی در حدود نصف مجموع در سطح قبلی خواهد شد. بنابراین، تعداد عناصر اضافی تولید شده، در حدود n(۱ + ۱/۲ + ۱/۴ + ...) = ۲n می‌باشد.

الگوریتم ۲-۲ به وضوح فرآیند تقسیم نمونه‌ای از یک مسئله به نمونه‌های کوچکتر را نشان می‌دهد؛ چرا که دو آرایهٔ جدید (نمونه‌های کوچکتر) در واقع از آرایه ورودی (نمونه اصلی) تولید می‌شوند. لذا این الگوریتم، انتخاب مناسبی جهت معرفی mergesort و روش تقسیم و غلبه می‌باشد. ذکر یک نکته در مورد mergesort ضروری است و آن اینکه با اندکی دستکاری روی آرایه ورودی S می‌توانیم مقدار فضای اضافی را تنها به یک آرایه n عنصری کاهش دهیم. روش مورد استفاده برای این کار مشابه روش استفاده شده برای الگوریتم ۱-۲ (جستجوی دودویی، بازگشتی) است.

الگوریتم ۴-۲ **مرتب‌سازی ادغامی ۲** (Mergesort2)

مسئله: یک آرایهٔ n کلیدی را به صورت غیرنزولی مرتب کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S با شاخصهایی از ۱ تا n.

خروجی: آرایه S شامل کلیدهایی مرتب به صورت غیرنزولی.

```
void mergesort2 (index low, index hgih).
{
   index mid;
   if (low < high){
      mid = ⌊(low + high) / 2⌋ ; ;
      mergesort2 (low, mid);
      mergesort2 (mid+1, high);
      merge2 (low, mid, high);
   }
}
```

طبق قراردادی که داشتیم، تنها متغیرهایی را به عنوان پارامترهای بازگشتی معرفی می‌کنیم که مقدارشان توسط فراخوانی‌های بازگشتی تغییر یابد، لذا n و s پارامترهای روال mergesort2 نیستند. اگر در الگوریتم، آرایهٔ S به صورت سراسری و n به عنوان تعداد عناصر S تعیین شده باشد، آنگاه فراخوانی سطح بالای mergesort2 به صورت ;(۱,n)mergesort2 خواهد بود. روال ادغام را برای mergesort2 می‌نویسیم.

الگوریتم ۵-۲ ادغام۲ (Merge2)

مسئله: دو زیرآرایه مرتب تولید شده در Mergesort2 را با هم ادغام کنید.

ورودی: شاخصهای low، mid، high و زیرآرایه‌ای از S با شاخصهایی از low تا high. کلیدها از قبل، در اندیس‌های low تا mid و ۱+mid تا high آرایه، به صورت غیرنزولی مرتب شده‌اند.

خروجی: زیرآرایه‌ای از S با شاخصهایی از low تا high شامل کلیدهایی مرتب به صورت غیرنزولی.

```
void merge2 (index low, index mid, index high)
{
   index i, j, k;
   keytype U[low..high];      //A local array needed for the merging
   i = low; j = mid + 1; k = low;
   while (i ≤ mid && j ≤ high){
      if (S[i] < S[j]){
         U[k] = S[i];
         i++;
      }
      else{
         U[k] = S[j];
         j++;
      }
      k++;
   }
   if (i > mid)
      move S[j] through S[high] to U[k] through U[high];
   else
      move S[i] through S[mid] to U[low] through S[high];
   move U[low] through U[high] to S[low] through S[high];
}
```

۲-۳ روش تقسیم و غلبه

با مطالعه جزء به جزء دو الگوریتم تقسیم و غلبه، اکنون آمادگی بهتری برای درک آن دارید. استراتژی **تقسیم و غلبه**، مراحل زیر را در برمی‌گیرد:

۱- تقسیم نمونه‌ای از یک مسئله به یک یا چند نمونه کوچکتر.

۲- حل (غلبه) هر یک از نمونه‌های کوچکتر. (اگر نمونه به اندازه کافی کوچک نباشد، از بازگشت برای انجام این کار استفاده می‌کنیم)

۳- در صورت لزوم، ترکیب جوابهای نمونه‌های کوچکتر جهت بدست آوردن جواب نمونه اصلی.

در مرحله ۳، از عبارت "در صورت لزوم" استفاده کردیم، چراکه در الگوریتم‌هایی نظیر جستجوی دودویی (الگوریتم ۱-۲)، نمونه اصلی تنها به یک نمونه کوچکتر کاهش می‌یابد. بنابراین، نیازی به ترکیب جوابها نیست. در ادامه، مثالهای بیشتری از تقسیم و غلبه ارائه خواهیم داد. در این مثالها، به طور ضمنی از مراحل فوق برای بدست آوردن جواب استفاده می‌کنیم.

۲-۴ مرتب‌سازی سریع (Quicksort)

در اینجا به یک الگوریتم مرتب‌سازی، موسوم به مرتب‌سازی سریع می‌پردازیم که در سال ۱۹۶۲ توسط Hoare ارائه شد. مرتب‌سازی سریع، که به Partition Exchange Sort نیز مشهور است، از این جهت با مرتب‌سازی ادغامی شباهت دارد که عمل مرتب‌سازی در آن، با تقسیم آرایه به دو بخش و سپس مرتب کردن هر بخش به صورت بازگشتی انجام می‌شود. در مرتب‌سازی سریع، آرایه با تعیین یک عنصر محوری و قراردادن تمامی عناصر کوچکتر از عنصر محوری در قبل از آن و قراردادن کلیه عناصر بزرگتر یا مساوی عنصر محوری در بعد از آن، بخش بندی می‌شود. عنصر محوری می‌تواند هر یک از عناصر آرایه باشد. اما برای سهولت کار، اولین عنصر را به عنوان عنصر محوری در نظر می‌گیریم. مثال زیر، چگونگی انجام مرتب‌سازی سریع را نشان می‌دهد.

مثال ۳-۲ فرض کنید آرایه‌ای شامل عناصر زیر باشد:

۱	۲	۳	۴	۵	۶	۷	۸
۲۵	۲۰	۱۰	۱۲	۲۷	۱۳	۲۲	۱۵

عنصرمحوری

۱- آرایه را طوری تقسیم کنید که همهٔ عناصر کوچکتر از عنصر محوری در سمت چپ آن و همه عناصر بزرگتر از عنصر محوری در سمت راست آن قرار گیرند:

۱	۲	۳	۴	۵	۶	۷	۸
۱۰	۱۳	۱۲	۱۵	۲۲	۲۷	۲۰	۲۵

عناصرکوچکتر — عنصرمحوری — عناصربزرگتر

۲- زیرآرایه‌ها را مرتب کنید:

۱	۲	۳	۴	۵	۶	۷	۸
۱۰	۱۳	۱۲	۱۵	۲۰	۲۲	۲۵	۲۷

مرتب شده — عنصرمحوری — مرتب شده

پس از تقسیم آرایه، ترتیب عناصر موجود در زیرآرایه‌ها مشخص نیست. ترتیب آنها از چگونگی انجام بخش بندی آرایه‌ها ناشی می‌شود. اما موضوع مهم این است که همهٔ عناصر کوچکتر از عنصر محوری به سمت چپ آن و همه عناصر بزرگتر از عنصر محوری به سمت راست آن انتقال می‌یابند، آنگاه روال مرتب‌سازی سریع جهت مرتب کردن زیرآرایه به صورت بازگشتی فراخوانی می‌شود. زیرآرایه‌ها تقسیم می‌شوند و این روال تا زمانی ادامه می‌یابد که به آرایه‌هایی تک عنصری برسیم. روشن است که آرایهٔ تک عنصری به خودی خود مرتب است. مثال ۳-۲، جواب را در سطح حل مسئله نشان می‌دهد و شکل ۳-۲، مراحل مختلف مرتب‌سازی آرایه به روش سریع را به تصویر می‌کشد.

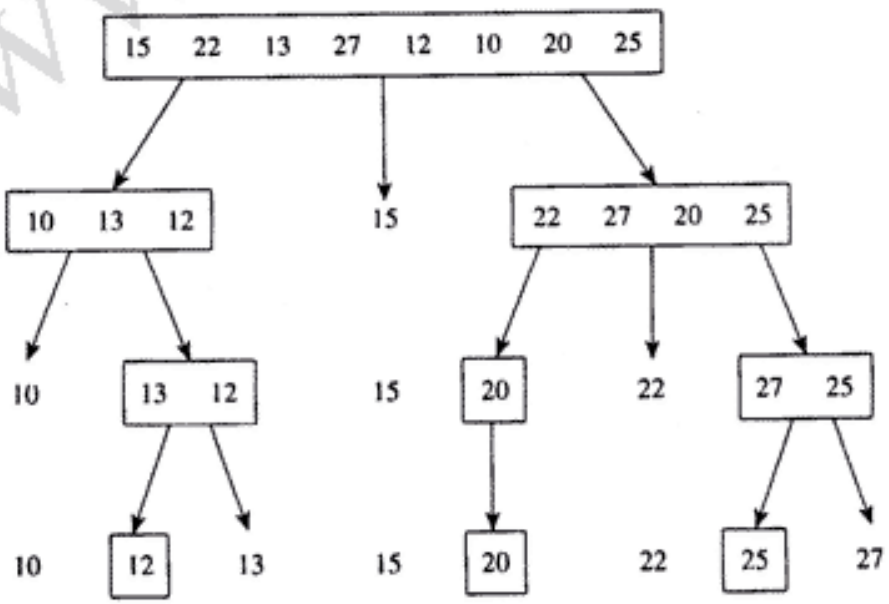
الگوریتم ۶-۲ مرتب‌سازی سریع (Quicksort)

مسئله: n کلید را به صورت غیرنزولی مرتب کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S با شاخصهایی از ۱ تا n.

خروجی: آرایه S شامل کلیدهایی مرتب به صورت غیرنزولی.

```
void quicksort (index low, index high)
{
   index pivotpoint;
   if (high > low){
      partition (low, high, pivotpoint);
      quicksort (low, pivotpoint - 1);
      quicksort (pivotpoint - 1, high);
   }
}
```



شکل ۳-۲ مراحل مرتب‌سازی سریع که توسط انسان انجام می‌شود.

جدول ۲-۲ یک مثال از روال partition *

i	j	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]	
—	—	15	22	13	27	12	10	20	25	←Initial values
2	1	**15**	**22**	13	27	12	10	20	25	
3	2	**15**	22	**13**	27	12	10	20	25	
4	2	**15**	[13]	[22]	**27**	12	10	20	25	
5	3	**15**	13	22	27	**12**	10	20	25	
6	4	**15**	13	[12]	27	[22]	**10**	20	25	
7	4	**15**	13	12	[10]	22	[27]	**20**	25	
8	4	**15**	13	12	10	22	27	20	**25**	
—	4	[10]	13	12	[15]	22	27	20	25	←Final values

*Items compared are in boldface. Items just exchanged appear in squares.

طبق قرارداد قبلی، n و s را به عنوان پارامترهای روال quicksort در نظر نمی‌گیریم. اگر آرایهٔ S را به صورت سراسری تعریف کرده و n تعداد عناصر آرایه S باشد، فراخوانی سطح بالای quicksort بصورت quicksort(۱, n); خواهد بود. بخش‌بندی آرایه توسط روال Partition انجام می‌شود. در زیر، الگوریتمی برای این روال آورده‌ایم.

الگوریتم ۲-۷ بخش‌بندی (Partition)

مسئله: آرایه S را برای مرتب‌سازی سریع بخش‌بندی کنید.

ورودی: دو شاخص low و high، و زیرآرایه‌ای از S با شاخصهایی از low تا high.

خروجی: pivotpoint، نقطه محوری برای زیرآرایه‌ای با شاخصهای low تا high.

```
void partition (index low, index high, index& pivotpoint)
{
  index i, j;
  keytype pivotitem;
  pivotitem = S[low];                    // انتخاب اولین عنصر بعنوان عنصر میانی
  j = low;
  for (i = low + 1; j <= high; i++)
    if (S[i] < pivotitem){
      j++;
      exchange S[i] and S[j];
    }
  pivotpoint = j;
  exchange S[low] and S[pivotpoint];
}
```

روال partition، تک‌تک عناصر آرایه را به ترتیب بررسی می‌کند. هر گاه عنصری کوچکتر از عنصر محوری باشد، آن را به سمت چپ آرایه منتقل می‌کند. جدول ۲-۲، چگونگی عملکرد روال patition بر آرایهٔ مثال ۳-۲ را نشان می‌دهد. اکنون روال‌های partition و quicksort را مورد تجزیه و تحلیل قرار می‌دهیم.

تحلیل پیچیدگی زمانی حالت معمول الگوریتم ۷-۲ (Partition)

عمل مبنایی: مقایسه S[i] با عنصر محوری.

اندازه ورودی: $n = high - low + 1$، تعداد عناصر زیرآرایه.

از آنجائیکه همهٔ عناصر غیر از اولین عنصر مورد مقایسه قرار می‌گیرند، لذا داریم:

$$T(n) = n - 1$$

در اینجا، n اندازهٔ زیرآرایه است، نه اندازهٔ آرایهٔ S. در واقع، n تنها در بالاترین سطح فراخوانی بیانگر اندازهٔ آرایه است.

مرتب‌سازی سریع، پیچیدگی زمانی حالت معمول ندارد. لذا تحلیل بدترین حالت و حالت میانی را برای آن انجام می‌دهیم.

تحلیل پیچیدگی زمانی بدترین حالت الگوریتم ۶-۲ (Quicksort)

عمل مبنایی: مقایسه S[i] با pivotitem در روال partition.

اندازه ورودی: n، تعداد عناصر آرایهٔ S.

بدترین حالت زمانی رخ می‌دهد که آرایه، از قبل به صورت غیرنزولی مرتب شده باشد. دلیل این امر روشن است. اگر آرایه‌ای به صورت غیرنزولی مرتب شده باشد، هیچ عنصری کوچکتر از اولین عنصر آرایه (عنصر محوری) نیست. بنابراین، هنگامی که روال pivotitem در بالاترین سطح فراخوانی می‌شود، هیچ عنصری به سمت چپ عنصر محوری منتقل نمی‌شود و بدین ترتیب، مقدار pivotpoint برابر ۱ می‌گردد. بطور مشابه، در هر فراخوانی بازگشتی، مقدار pivotpoint برابر مقدار low خواهد شد. لذا آرایه مکرراً به یک زیرآرایه خالی (در سمت چپ) و یک زیرآرایه با یک عنصر کمتر (درسمت راست) تقسیم می‌شود. در اینصورت داریم:

$$T(n) = \underbrace{T(0)}_{\text{زمان مرتب‌سازی زیرآرایهٔ سمت‌چپ}} + \underbrace{T(n-1)}_{\text{زمان مرتب‌سازی زیرآرایه سمت‌راست}} + \underbrace{n-1}_{\text{زمان‌بخش‌بندی}}$$

ما از نماد T(n) استفاده کردیم زیرا اکنون در حال تعیین پیچیدگی زمانی حالت معمول، برای دسته‌ای از نمونه‌ها که به صورت غیرنزولی مرتب هستند، می‌باشیم. چون $T(0) = 0$، لذا بازگشت زیر را داریم:

$$T(n) = T(n-1) + n - 1 \qquad ,n>0$$
$$T(0) = 0$$

این بازگشت در مثال ۱۶- B از ضمیمه B حل شده است. جواب چنین است:

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

دریافتیم که بدترین حالت، حداقل معادل $n(n-1)/2$ است. می‌خواهیم با استفاده از استقراء نشان دهیم که برای تمامی مقادیر n داریم:

$$W(n) \le \frac{n(n-1)}{2}$$

پایه استقراء: برای $n = 0$،

$$W(0) = 0 \le \frac{0(0-1)}{2}$$

فرض استقراء: فرض کنید که

$$W(k) \le \frac{k(k-1)}{2} \qquad \text{برای } 0 \le k < n,$$

گام استقراء: بایستی نشان دهیم که

$$W(n) \le \frac{n(n-1)}{2}$$

برای یک n معین، نمونه‌ای با اندازه n وجود دارد که زمان پردازش آن برابر W(n) است. P را مقدار Pivotpoint (نقطه محوری) بازگردانده شده توسط روال Partition در بالاترین سطح پردازش این نمونه درنظر می‌گیریم. به دلیل اینکه زمان پردازش نمونه‌های به اندازه $p - 1$ و $n - p$ نمی‌توانند بیشتر از $W(p-1)$ و $W(n-p)$ باشد، بنابراین

$$W(n) \le W(p-1) + W(n-p) + n - 1$$
$$\le \frac{(p-1)(p-2)}{2} + \frac{(n-p)(n-p-1)}{2} + n - 1$$

نامساوی اخیر از فرض استقراء نتیجه می‌شود. محاسبات جبری نشان می‌دهد که برای $1 \le p \le n$، عبارت اخیر بدینصورت خلاصه می‌شود:

$$W(n) \le \frac{n(n-1)}{2}$$

این مطلب، اثبات استقراء را کامل می‌کند. بنابراین، نشان داده‌ایم که پیچیدگی زمانی بدترین حالت برابراست با

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

همانطوریکه گفته شد، بدترین حالت زمانی اتفاق می‌افتد که آرایه از قبل مرتب شده باشد؛ چرا که ما همیشه اولین عنصر آرایه را به عنوان عنصر محوری انتخاب می‌کنیم. بنابراین، اگر بدانیم که آرایه نزدیک به

مرتب شدن است، آنگاه انتخاب اولین عنصر به عنوان عنصر محوری، انتخاب خوبی نخواهد بود. در فصل هفتم که بیشتر به بحث مرتب‌سازی سریع می‌پردازیم، روشهای دیگری را برای انتخاب عنصر محوری مطرح می‌کنیم که اگر از این روشها استفاده کنیم، در صورتی که آرایه از قبل مرتب نباشد، بدترین حالت رخ نمی‌دهد، ولی پیچیدگی زمانی بدترین حالت همچنان برابر $n(n-1)/2$ خواهد بود.

در بدترین حالت، الگوریتم ۶-۲ سریعتر از الگوریتم ۳-۱ (مرتب‌سازی تبادلی) نیست؛ بلکه در حالت میانی است که مرتب‌سازی سریع، شایستگی این نام را پیدا کرده است.

تحلیل پیچیدگی زمانی حالت میانی الگوریتم ۶-۲ (Quicksort)

عمل مبنایی: مقایسه S[i] با pivotpoint در روال partition.

اندازه ورودی: n، تعداد عناصر آرایهٔ S.

فرض می‌کنیم هیچ دلیلی وجود ندارد که عناصر موجود در آرایه به شکل خاصی مرتب شده باشند. بنابراین، pivotpoint می‌تواند بطور کاملاً مشابه و یکسان، هر یک از مقادیر ۱ تا n را به خود بگیرید. اگر به دلیلی توجیه شویم که عناصر آرایه به شکل خاصی قرار گرفته‌اند، این تحلیل درست نخواهد بود. پیچیدگی زمانی حالت میانی به صورت بازگشت زیر ارائه شده است:

احتمال نقطه محوری p است

$$A(n) = \sum_{p=1}^{n} \frac{1}{n} [A(p-1) + A(n-p)] + \underbrace{n-1}_{\text{زمان بخش‌بندی}} \qquad (۲-۱)$$

زمان میانگین برای مرتب‌سازی زیرآرایه‌ها وقتی که نقطه محوری برابر p است.

در تمرینات نشان می‌دهیم که

$$\sum_{p=1}^{n} [A(p-1) + A(n-p)] = 2 \sum_{p=1}^{n} A(p-1)$$

با ترکیب دو تساوی فوق داریم:

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

و با ضریب طرفین تساوی در n داریم:

$$nA(n) = 2 \sum_{p=1}^{n} A(p-1) + n(n-1) \qquad (۲-۲)$$

به جای n در تساوی فوق، n-1 را قرار می‌دهیم:

$$(n-1)A(n-1) = 2 \sum_{p=1}^{n} A(p-1) + (n-1)(n-2) \qquad (۲-۳)$$

تساوی ۳-۲ را از تساوی ۲-۲ کم می‌کنیم:

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1)$$

که به صورت زیر ساده می‌شود:

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

اگر فرض کنیم که $a_n = \frac{A(n)}{n+1}$ است، آنگاه بازگشت زیر را خواهیم داشت:

$$\begin{aligned} a_n &= a_{n-1} + \frac{2(n-1)}{n(n+1)} \qquad ,n > 0 \\ a_0 &= 0 \end{aligned}$$

همانند بازگشت ارائه شده در مثال B-۲۲، جواب تقریبی این بازگشت چنین است:

$$a_n \approx 2 \ln n$$

که اشاره دارد به اینکه

$$\begin{aligned} A(n) &\approx (n+1)\,2 \ln n = (n+1)\,2(\ln 2)(\lg n) \\ &\approx 1/38(n+1)\lg n \in \Theta(n \lg n) \end{aligned}$$

پیچیدگی زمانی حالت میانی مرتب‌سازی سریع، مشابه پیچیدگی زمانی مرتب‌سازی ادغامی است. این دو مرتب‌سازی در فصل هفتم و در کتاب knuth (۱۹۷۳) بیشتر با هم مقایسه می‌شوند.

۲-۵ الگوریتم ضرب ماتریسی استراسن (STRASSEN)

به خاطر آورید که الگوریتم ۲-۱ (ضرب ماتریسی)، دو ماتریس را دقیقاً براساس تعریف ضرب ماتریسها درهم ضرب می‌کرد و ما نشان دادیم که پیچیدگی زمانی تعداد ضربهای آن برابر است با $T(n) = n^3$، که n تعداد سطرها و ستونهای ماتریسها است. همچنین می‌توان تعداد جمع‌ها را بررسی نمود. پیچیدگی زمانی تعداد جمعها با اندکی تغییرات در الگوریتم، برابراست با $T(n) = n^3 - n^2$. بدلیل آنکه پیچیدگی زمانی هر دو الگوریتم فوق در $\Theta(n^3)$ می‌باشد، به وضوح الگوریتمهایی غیرکارا به نظر می‌رسند. در سال ۱۹۶۹، استراسن الگوریتمی ارائه نمود که پیچیدگی زمانی آن هم در ضرب و هم در جمع/تفریق بهتر از توان سوم عناصر است. مثال زیر بیانگر روش ابداعی اوست.

مثال ۲-۴ فرض کنید می‌خواهیم دو ماتریس ۲×۲ با نامهای A و B را در هم ضرب کرده و ماتریس C، که حاصلضرب آن دو است را بدست آوریم. یعنی:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

استراسن بیان نمود که اگر فرض کنیم

$$m_1 = (a_{11} + a_{22})(b_{11} + b_{22}) \qquad m_5 = (a_{11} + a_{12})b_{22}$$
$$m_2 = (a_{21} - a_{22})b_{11} \qquad m_6 = (a_{21} - a_{11})(b_{11} + b_{12})$$
$$m_3 = a_{11}(b_{11} - b_{22}) \qquad m_7 = (a_{12} - a_{22})(b_{21} + b_{22})$$
$$m_4 = a_{22}(b_{21} - b_{11})$$

آنگاه حاصلضرب C از فرمول زیر محاسبه می‌شود:

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

شما در تمرینات، صحت این موضوع را نشان خواهید داد.

روش استراسن برای ضرب دو ماتریس ۲×۲، نیازمند هفت عمل ضرب و ۱۸ عمل جمع/تفریق می‌باشد؛ در حالیکه در روش قبلی می‌بایست هشت عمل ضرب و چهار عمل جمع/تفریق انجام می‌دادیم. بنابراین، در روش استراسن توانسته‌ایم یک ضرب را با ۱۴ جمع/تفریق عوض کنیم. در واقع، روش استراسن برای ضرب ماتریسهای ۲×۲، آنچنان مؤثر و ارزشمند نیست. از آنجائیکه در فرمول استراسن از جابجاپذیری ضرب استفاده نشده است، لذا آن فرمولها برای ماتریس‌های بزرگتری که هر یک به چهار زیر ماتریس تقسیم شده‌اند، مناسب می‌باشند. ابتدا ماتریسهای A و B را تقسیم‌بندی می‌کنیم (آنچنانکه در شکل ۴-۲ نشان داده‌ایم). با فرض اینکه n توانی از ۲ است، ماتریس A_{11}، برای مثال، نمایانگر زیرماتریسی از A می‌باشد:

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} \cdots a_{1,n/2} \\ a_{21} & a_{22} \cdots a_{2,n/2} \\ & \vdots \\ a_{n/2,1} & \cdots \quad a_{n/2,n/2} \end{bmatrix}$$

با روش استراسن، ابتدا عبارت زیر را محاسبه می‌کنیم:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$\begin{matrix} \\ n/2 \end{matrix} \overset{\leftarrow n/2 \rightarrow}{\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

شکل ۴-۲ پخش‌بندی زیر ماتریسها در الگوریتم استراسن

عملیات ما در اینجا، جمع و ضرب ماتریسی است. به روشی مشابه، M_2 تا M_7 را محاسبه می‌کنیم و سپس به محاسبه C_{11} می‌پردازیم:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

و به همین ترتیب C_{12}، C_{21}، C_{22}. در نهایت، با ترکیب چهار زیر ماتریس C_{ij} می‌توانیم حاصلضرب C از ماتریسهای A و B را بدست آوریم. مثال زیر این مراحل را نشان می‌دهد.

مثال ۵-۲ فرض کنید

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} \qquad B = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

شکل ۵-۲ چگونگی تقسیم بندی ماتریس‌ها به روش استراسن را نشان می‌دهد. پس از تقسیم ماتریس‌ها بایستی عبارت زیر را محاسبه کنیم:

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ &= \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix}\right) \times \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix}\right) \\ &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} \end{aligned}$$

هنگامی که ماتریسها به اندازه کافی کوچک هستند، ضرب را به روش استاندارد انجام می‌دهیم. در این مثال، وقتی که $n = 2$ است، چنین عملی صورت می‌گیرد. بنابراین،

$$\begin{aligned} M_1 &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} \\ &= \begin{bmatrix} 3 \times 17 + 5 \times 7 & 3 \times 10 + 5 \times 9 \\ 11 \times 17 + 13 \times 7 & 11 \times 10 + 13 \times 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix} \end{aligned}$$

سپس M_2 تا M_7 نیز به همین روش محاسبه شده و در نهایت با استفاده از M_2 تا M_7، مقادیر $C_{11}, C_{12}, C_{21}, C_{22}$ را محاسبه می‌کنیم. ترکیب $C_{11}, C_{12}, C_{21}, C_{22}$ حاصلضرب C را تولید می‌کند.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \left[\begin{array}{cc:cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hdashline 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{array}\right] \times \left[\begin{array}{cc:cc} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ \hdashline 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{array}\right]$$

شکل ۵-۲ بخش بندی به روش استراسن با $n = 4$ و مقادیر داده شده در ماتریسها.

الگوریتم ۸-۲ استراسن (Strassen)

مسئله: حاصلضرب دو ماتریس $n \times m$، وقتی که n توانی از ۲ است را تعیین کنید.

ورودی: عدد صحیح n که توانی از ۲ است، و دو ماتریس $n \times n$ مفروض A و B.

خروجی: C، حاصلضرب A و B.

```
void strassen (int n,
               n × n_matrix A,
               n × n_matrix B,
               n × n_matrix& C)
{
  if (n <= threshold)
     compute C = A × B using the standard algorithm;
  else{
     partition A into four submatrices A11, A12, A21, A22;
     partition A into four submatrices B11, B12, B21, B22;
     compute C = A × B using the strassen's Method;
        // strassen (n/2, A11+A12, B11+B22, M1)  یک نمونه از فراخوانی بازگشتی:
  }
}
```

مقدار آستانه، نقطه‌ای است که احساس می‌شود استفاده از الگوریتم استاندارد، سودمندتر از بکارگیری روش استراسن بطور بازگشتی می‌باشد. در بخش ۷-۲، روشی را جهت تعیین مقدار آستانه بررسی می‌کنیم.

تحلیل پیچیدگی زمانی حالت معمول تعداد ضربهای الگوریتم ۸-۲ (استراسن)

عمل مبنایی: یک ضرب ابتدائی.

اندازه ورودی: n، تعداد سطرها و ستونها در ماتریسها.

برای سهولت، حالتی را بررسی می‌کنیم که در آنجا تقسیمات متوالی را تا رسیدن به دو ماتریس ۱×۱ ادامه می‌دهیم. مقدار آستانه واقعی، بر روی ترتیب مؤثر نیست. هنگامی که $n = 1$ باشد، دقیقاً یک عمل ضرب انجام می‌شود و هنگامی که یک ماتریس $n \times n$ $(n<1)$ داشته باشیم، الگوریتم دقیقاً هفت مرتبه و هر بار با ارسال یک ماتریس $(n/2) \times (n/2)$ فراخوانی می‌شود و هیچ ضربی در سطح بالا انجام نمی‌گیرد. بازگشت زیر را ارائه می‌دهیم:

$$T(n) = 7T\left(\frac{n}{2}\right) \quad n > 1,\ n \text{ توانی از ۲ است}$$
$$T(1) = 1$$

این بازگشت در مثال ۲ - B از ضمیمه B به صورت زیر حل شده است:

$$T(n) = n^{\lg 7} \approx n^{2/81} \in \Theta(n^{2/81})$$

تحلیل پیچیدگی زمانی حالت معمول تعداد جمعها/تفریقها در الگوریتم ۸-۲ (استراسن)

عمل مبنایی: یک جمع یا تفریق ابتدائی.

اندازه ورودی: n، تعداد سطرها و ستونها در ماتریسها.

مجدداً فرض می‌کنیم که عمل تقسیم ماتریس‌ها را تا دستیابی به دو ماتریس ۱×۱ ادامه می‌دهیم. هنگامی که $n = 1$ است، هیچ جمع/تفریقی انجام نمی‌شود و هنگامی که دو ماتریس $n \times n$ $(n<1)$ داریم، الگوریتم دقیقاً هفت مرتبه و هر بار با ارسال یک ماتریس $(n/2) \times (n/2)$ فراخوانی می‌شود و ۱۸ عمل جمع/تفریق ماتریسی بر روی ماتریسهای $(n/2) \times (n/2)$ انجام می‌گیرد. هنگامی که دو ماتریس $(n/2) \times (n/2)$ با هم جمع/تفریق می شوند، به تعداد $(n/2)^2$، جمع/تفریق روی عناصر ماتریس انجام می‌شود. لذا بازگشت زیر را داریم:

$$T(n) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2 \quad n>1,\ n \text{ توانی از } 2 \text{ است}$$
$$T(1) = 0$$

این بازگشت در مثال ۲ - B از ضمیمه B به صورت زیر حل شده است:

$$T(n) = 6n^{\lg 7} - 6n^2 \approx 6n^{2/81} - 6n^2 \in \Theta(n^{2/81})$$

اگر n توانی از ۲ نباشد، بایستی الگوریتم قبلی را تغییر دهیم. یک تغییر ساده، افزودن تعداد کافی سطر و ستون صفر به ماتریس اولیه جهت رسیدن به ماتریسی با ابعاد توانی از ۲ است. روش دیگر، این است که در فراخوانی‌های بازگشتی هر جا که تعداد سطرها و ستونهای ماتریس فرد شد، یک سطر و یک ستون صفر به ماتریس اضافه کنیم. استراسن (۱۹۶۹) تغییر پیچیده‌تری را پیشنهاد کرد بدینصورت که ما ماتریسها را در یک ماتریس بزرگتر با سطرها و ستونهایی به اندازهٔ $m2^k$ محاط کنیم بطوری که $k = \lfloor \lg n - 4 \rfloor$ و $m = \lfloor n/2^k \rfloor + 1$ باشد، آنگاه روش استراسن را تا رسیدن به مقدار آستانه m، و روش استاندارد را پس از دستیابی به مقدار آستانه بکار بگیریم. می‌توان نشان داد که در این روش پیشنهادی، تعداد کل عملیات محاسباتی (ضرب، جمع و تفریق) کمتر از $4.7n^{2.81}$ است.

جدول ۳-۲، مقایسه پیچیدگیهای زمانی الگوریتم استاندارد و الگوریتم استراسن را برای وقتی که n توانی از ۲ است، نشان می‌دهد. با چشم پوشی از سربارهای موجود در فراخوانی‌های بازگشتی، الگوریتم استراسن همواره از نظر تعداد ضربها و برای مقادیر بزرگ n، از نظر تعداد جمعها/تفریقها کاراتر است. در بخش ۷-۲، تحلیلی انجام می‌دهیم که مدت زمان مورد استفاده توسط فراخوانی‌های بازگشتی را محاسبه می‌کند.

سامئول وینوگراد نوع دیگری از الگوریتم استراسن را ارائه داد که تنها به ۱۵ عمل جمع/تفریق نیاز داشت. برای این الگوریتم، پیچیدگی زمانی جمع/تفریق از طریق فرمول زیر محاسبه می‌شود:

$$T(n) \approx 5n^{2.81} - 5n^2$$

جدول ۳-۲ مقایسهٔ دو الگوریتم ضرب ماتریسهای n × n		
	Standard Algorithm	Strassen's Algorithm
Multiplications	n^3	$n^{2.81}$
Additions/Subtractions	$n^3 - n^2$	$6n^{2.81} - 6n^2$

کوپر اسمیت و وینوگراد نیز در سال ۱۹۸۷، یک الگوریتم ضرب ماتریسی نوشتند که پیچیدگی زمانی آن برای تعداد ضربها به صورت $O(n^{2/38})$ است. با وجود این، مقدار ثابت آنقدر بزرگ است که معمولاً الگوریتم استراسن کاراتر می‌شود.

می‌توان ثابت نمود که ضرب ماتریس به یک الگوریتم با پیچیدگی زمانی حداقل از درجه دوم نیازمند است. اینکه آیا ضربهای ماتریسی می‌توانند به صورت الگوریتم‌های زمان-مربعی مطرح شوند، سؤالی است که همچنان باقی مانده است زیرا هیچکس تاکنون یک الگوریتم زمان-مربعی برای ضرب ماتریسی بدست نیاورده و البته کسی هم ثابت نکرده است که ارائه چنین الگوریتمی غیرممکن است.

آخرین نکته این است که برخی عملیات ماتریسی نظیر معکوس کردن یک ماتریس و یافتن دترمینان یک ماتریس، مستقیماً به ضرب ماتریسها مربوط می‌شود. بنابراین، می‌توانیم به آسانی الگوریتم‌هایی را برای این عملیات که به اندازه الگوریتم استراسن برای ضرب ماتریسی کارائی دارند، ارائه دهیم.

۲-۶ محاسبه با اعداد صحیح بزرگ

فرض کنید که می‌خواهیم عملیات محاسباتی را بر روی اعداد صحیحی انجام دهیم که اندازهٔ آنها خارج از توانایی سخت‌افزار در نمایش آن اعداد صحیح است. اگر نگهداری تمامی ارقام مهم در نتایج حاصله ضروری باشد، آنگاه تغییر وضعیت نمایشی به اعداد اعشاری نیز بی‌ارزش خواهد بود. در چنین حالاتی، تنها راه چاره، استفاده از نرم‌افزار جهت نمایش و اجرای عملیات بر روی اعداد صحیح بزرگ است. ما می‌توانیم این کار را به کمک روش تقسیم و غلبه انجام دهیم. بحث‌های آتی ما بر روی اعداد صحیح در مبنای ۱۰ است. با این حال، روشهای مورد بحث می‌توانند در مبناهای دیگر نیز بسط داده شوند.

۲-۶-۱ نمایش اعداد صحیح بزرگ:
جمع و دیگر عملیات زمان - خطی

یک روش مشخص برای نمایش یک عدد صحیح بزرگ، استفاده از آرایهٔ اعداد صحیح است. بدینصورت که هر اندیس آرایه، یک رقم را در خود ذخیره می‌کند. برای مثال، عدد صحیح ۵۴۳/۱۲۷ را می‌توان به صورت زیر در آرایه S نشان داد:

۵	۴	۳	۱	۲	۷
$S[۶]$	$S[۵]$	$S[۴]$	$S[۳]$	$S[۲]$	$S[۱]$

برای مشخص نمودن اعداد صحیح در آرایه باید بالاترین اندیس آرایه را به عنوان علامت عدد در نظر بگیریم، بدینصورت که دادن مقدار صفر به اندیس مورد نظر، نشانهٔ مثبت بودن عدد و دادن مقدار یک به آن، نشانه منفی بودن عدد می‌باشد. ما این نحوهٔ نمایش را در نظر می‌گیریم و از نوع داده‌ای large-integer جهت تعریف یک آرایهٔ به اندازه کافی بزرگ جهت نمایش اعداد صحیح استفاده می‌کنیم.

نوشتن یک الگوریتم زمان-خطی جهت انجام عمل جمع یا تفریق اعداد صحیح بزرگ، کار مشکلی نیست. عمل مبنایی، شامل یک عمل روی یک رقم دهدهی است. در تمرینات از شما می‌خواهیم که این الگوریتم‌ها را نوشته و آنها را تحلیل کنید. علاوه بر این، الگوریتم‌های زمان-خطی می‌توانند به راحتی برای انجام عملیات زیر نوشته شوند:

$$U \times ۱۰^m \qquad U \; divide \; ۱۰^m \qquad U \; rem \; ۱۰^m$$

که در آن U بیانگر یک عدد صحیح بزرگ و m بیانگر یک عدد صحیح غیرمنفی، divide خارج قسمت و rem باقیمانده تقسیم عدد صحیح است.

۲-۶-۲ ضرب اعداد صحیح بزرگ

یک الگوریتم زمان-مربعی ساده برای ضرب اعداد صحیح بزرگ، الگوریتمی است که از روش استاندارد ضرب تبعیت می‌کند. حال می‌خواهیم الگوریتمی بهتر از زمان-مربعی ارائه دهیم. الگوریتم‌ها، براساس روش تقسیم و غلبه، یک عدد صحیح n رقمی را به دو عدد صحیح تقریباً n/۲ رقمی تقسیم می‌کنند. به مثال زیر توجه کنید.

$$\underbrace{۵۶۷,۸۳۲}_{۶\text{ رقم}} = \underbrace{۵۶۷}_{۳\text{ رقم}} \times ۱۰^{۳} + \underbrace{۸۳۲}_{۳\text{ رقم}}$$

$$\underbrace{۹,۴۲۳,۷۲۳}_{۷\text{ رقم}} = \underbrace{۹۴۲۳}_{۴\text{ رقم}} \times ۱۰^{۳} + \underbrace{۷۲۳}_{۳\text{ رقم}}$$

در حالت کلی، اگر n تعداد رقم‌های عدد صحیح U باشد، آنگاه عدد U به دو عدد صحیح، یکی با $\lceil n/۲ \rceil$ و دیگری با $\lfloor n/۲ \rfloor$ رقم، به صورت زیر تقسیم می‌شود:

$$\underset{\text{رقم}}{n} \quad U = \underset{\lceil n/۲ \rceil \text{ رقم}}{x} \times ۱۰^{m} + \underset{\lfloor n/۲ \rfloor \text{ رقم}}{y}$$

که با توجه به فرم کلی فوق، توان m عدد ۱۰، برابراست با $m = \lfloor n/۲ \rfloor$

اگر ما دو عدد صحیح n رقمی داشته باشیم:

$$u = x \times ۱۰^{m} + y$$
$$v = w \times ۱۰^{m} + z$$

آنگاه حاصل uv برابراست با

$$uv = (x \times ۱۰^{m} + y)(w \times ۱۰^{m} + z)$$
$$= xW + ۱۰^{۲m} + (xz + wy) \times ۱۰^{m} + yz$$

ما می‌توانیم u و v را در هم ضرب کنیم بطوری که چهار عمل ضرب روی اعداد صحیح که شامل تقریباً نیمی از ارقام می‌باشند، انجام شود و بدین ترتیب، عملیات به صورت زمان-خطی انجام پذیرد. مثال زیر، بیانگر این روش است.

مثال ۶-۲ عبارت زیر را در نظر بگیرید.

$$۵۶۷,۸۳۲ \times ۹,۴۲۳,۷۲۳ = (۵۶۷ \times ۱۰^{۳} + ۸۳۲)(۹۴۲۳ \times ۱۰^{۳} + ۷۲۳)$$
$$= ۵۶۷ \times ۹۴۲۳ \times ۱۰^{۶} + (۵۶۷ \times ۷۲۳ + ۹۴۲۳ \times ۸۳۲) \times ۱۰^{۳} + ۸۳۲ \times ۷۲۳$$

به طور بازگشتی، اعداد صحیح کوچکتر نیز می‌توانند با تقسیم شدن به اعدادی کوچکتر از خود، در هم ضرب شوند. این عمل تقسیم تا آنجا ادامه می‌یابد که به یک مقدار آستانه برسیم که در آن هنگام می‌توانیم عمل ضرب را به روش استاندارد انجام دهیم.

اگرچه این روش را با اعدادی که دارای تعداد ارقام تقریباً مساوی بودند، ارائه کردیم؛ ولی این کار را می‌توان در حالتی که تعداد ارقام دو عدد تقریباً مساوی نباشند نیز انجام داد. با استفاده از $m = \lfloor n/۲ \rfloor$ می‌توانیم آنها را به دوبخش تقسیم کنیم که n، تعداد ارقام عدد صحیح بزرگتر است. فرآیند تقسیم تا آنجا ادامه می‌یابد که یکی از اعداد صحیح برابر صفر شود یا به برخی مقادیر آستانه برای اعداد صحیح بزرگ دست یابیم؛ یعنی تا زمانی که عمل ضرب بتواند با استفاده از سخت‌افزار کامپیوتر انجام شود.

الگوریتم ۹-۲ ضرب اعداد صحیح

مسئله: دو عدد صحیح بزرگ U و V را در هم ضرب کنید.

ورودی: اعداد صحیح بزرگ U و V.

خروجی: Prod، حاصلضرب U و V.

```
large_integer prod (large_integer U, large_integer V)
{
   large_integer x, y, w, z;
   int n, m;
```

```
n = maximum(number of digits in U, number of digists in V);
if (U == O || V == O)
   return O;
else if (n <= threshold)
   return U × V obtained in the usual way;
else{
   m = ⌊ n/2⌋ ;;
   x = U divide 10^m; y = U rem 10^m;
   w = V divide 10^m; z = V rem 10^m;
   return prod(x, w) × 10^2m + (prod(x, z) + prod(w, y)) × 10^m + prod(y, z);
}
}
```

توجه کنید که n، یک ورودی تلویحی برای الگوریتم است زیرا نمایانگر تعداد ارقام عدد صحیح بزرگتر می‌باشد. به خاطر داشته باشید که divide ، rem و × نمایانگر توابعی زمان-خطی هستند.

تحلیل پیچیدگی زمانی بدترین حالت الگوریتم ۹-۲ (ضرب اعداد صحیح بزرگ)

ما بررسی می‌کنیم که چه مدت طول می‌کشد تا دو عدد صحیح n رقمی در هم ضرب شوند.

عمل مبنایی: انجام یک عمل روی یک رقم دهدهی به هنگام عمل جمع، تفریق، یا محاسبهٔ $divide\ ۱۰^m$, $rem\ ۱۰^m$ یا $۱۰^m \times$ می‌باشد. برای انجام هر یک از سه عمل اخیر بایستی m مرتبه عمل مبنایی اجرا شود.

اندازه ورودی: n، تعداد ارقام در هر یک از دو عدد صحیح.

بدترین حالت زمانی رخ می‌دهد که در هر دو عدد صحیح، هیچ رقم صفر یافت نشود و بازگشت، تنها زمانی خاتمه می‌یابد که به مقدار آستانه برسیم. ما این حالت را تحلیل می‌کنیم. فرض کنید که n توانی از ۲ است، آنگاه تعداد رقمهای x ، y ، w و z دقیقاً برابر n/۲ می‌باشد. بدین معنا که اندازه ورودی هر یک از چهار فراخوانی بازگشتی برای تابع Prod برابر n/۲ است. از آنجائیکه m = n/۲ است، لذا عملیات زمان-خطی جمع، تفریق، $divide\ ۱۰^m$, $rem\ ۱۰^m$ و $۱۰^m \times$ همگی پیچیدگی‌های زمانی خطی روی عناصر n دارند. ماکزیمم اندازه ورودی برای تمامی این عملیات زمان-خطی، یکسان نیست. بنابراین، تعیین دقیق پیچیدگی زمانی کار ساده‌ای نخواهد بود. می‌توان کلیه عملیات زمان-خطی را در یک گروه قرار داده و آن را با cn نمایش داد که در آن c، یک مقدار ثابت مثبت است. در این صورت بازگشت ما چنین خواهد بود:

$$W(n) = ۴W(\frac{n}{۲}) + Cn \qquad n > s \text{ و } n \text{ توانی از } ۲$$
$$W(s) = ۰$$

مقدار واقعی S که در آن تقسیم نمونه‌ها پایان می‌یابد، کمتر یا مساوی مقدار آستانه و توانی از ۲ خواهد بود؛

چرا که تمامی ورودی‌ها در این حالت، توانی از ۲ می‌باشند.

در حالتی که n توانی از ۲ نباشد، نوشتن یک بازگشت همانند حالت قبلی امکان‌پذیر است. اما باید از جزء صحیح بالا و پائین برای این منظور استفاده کنیم. با استفاده از استقرایی نظیر مثال B-۲۵ در ضمیمه B می‌توانیم نشان دهیم که نهایتاً W(n) غیرنزولی است. بنابراین، از قضیه B-۶ در ضمیمه B استنباط می‌شود که

$$W(n) \in \Theta(n^{\lg 4}) = \Theta(n^2)$$

الگوریتم ما برای ضرب اعداد صحیح بزرگ، هنوز از درجهٔ دوم است. مشکل اینجاست که الگوریتم، چهار عمل ضرب بر روی اعداد صحیح با تعداد ارقامی برابر نصف تعداد ارقام اعداد صحیح اولیه انجام می‌دهد. اگر ما بتوانیم تعداد این ضربها را کاهش دهیم، آنگاه الگوریتمی بهتر از درجه دوم خواهیم داشت. این کار را به روش زیر انجام می‌دهیم. به خاطر دارید که تابع prod بایستی موارد زیر را تعیین کند:

$$xw,\ xz + yw,\ yz \qquad (۲-۴)$$

و این کار را با چهار مرتبه فراخوانی تابع Prod به طور بازگشتی انجام می‌داد تا موارد زیر را محاسبه کند:

$$xw,\ xz,\ yw,\ yz$$

حال اگر به جای این کار، چنین تعریف کنیم:

$$r = (x + y)(w + z) = xw + (xz + yw) + yz$$

آنگاه خواهیم داشت:

$$xz + yw = r - xw - yz$$

و این بدین معناست که می‌توانیم با تعیین سه مقدار زیر، عبارت ۴-۲ را بدست آوردیم.

$$r = (x + y)(w + z),\ xw,\ yz$$

برای تعیین این سه مقدار کافیست سه عمل ضرب و چند عمل زمان-خطی جمع و تفریق انجام دهیم. الگوریتم زیر مبین این روش است.

الگوریتم ۱۰-۲ ضرب اعداد صحیح بزرگ ۲

مسئله: دو عدد صحیح بزرگ U و V را در هم ضرب کنید.

ورودی: عدد صحیح بزرگ U و V.

خروجی: Prod2، حاصلضرب U و V.

```
large_integer prod2(large_integer U, large_integer V)
{
   large_integer x, y, w, z, r, p, q;
   int n, m;
   n = maximum(number of digits in U, number of digists in V);
   if (U == 0 || V == 0)
```

```
    return 0;
  else if (n <= threshold)
    return U × V obtained in the usual way;
  else{
    m = ⌊ n/2 ⌋;;
    x = U divide 10^m; y = U rem 10^m;
    w = V divide 10^m; z = V rem 10^m;
    r = prod2(x+y, w+z);
    p = prod2(x, w);
    q = prod2(y, z);
    return p × 10^2m + (r - p - g) × 10^m + q;
  }
}
```

تحلیل پیچیدگی زمانی بدترین حالت الگوریتم ۱۰-۲ (ضرب اعداد صحیح بزرگ ۲)

می‌خواهیم بررسی کنیم که چه مدت طول می‌کشد تا دو عدد صحیح n رقمی در هم ضرب شوند.

عمل مبنایی: انجام یک عمل بر روی یک رقم دهدهی به هنگام جمع، تفریق یا محاسبه $divide\ 10^m$, $rem\ 10^m$ یا $10^m \times$ برای انجام هر یک از این سه فراخوانی اخیر، بایستی m مرتبه عمل مبنایی انجام شود.

اندازه ورودی: n، تعداد ارقام در هر یک از دو عدد صحیح.

بدترین حالت زمانی رخ می‌دهد که در هر دو عدد صحیح، هیچ رقم صفر یافت نشود زیرا بازگشت، تنها زمانی خاتمه می‌یابد که به مقدار آستانه برسیم. ما این حالت را بررسی می‌کنیم.

اگر n توانی از ۲ باشد، آنگاه تعداد رقمهای x ، y ، w و z دقیقاً برابر n/۲ است. بنابراین، همانطوریکه در جدول ۴-۲ نشان داده شده است،

$$n/2 \le \text{تعداد رقمها در } x+y \le n/2 + 1$$

$$n/2 \le \text{تعداد رقمها در } w+z \le n/2 + 1$$

یعنی اینکه اندازه ورودیهای زیر را برای فراخوانی‌های تابع داریم:

	اندازه ورودی
$prod2(x+y, w+z)$	$n/2 \le$ اندازه ورودی $\le n/2 + 1$
$prod2(x, w)$	$n/2$
$prod2(y, z)$	$n/2$

چون $m = n/2$ است، لذا عملیات زمان-خطی جمع، تفریق، $divide\ 10^m$, $rem\ 10^m$ و $10^m \times$

جدول ۴-۲ مثالهایی از تعداد ارقام x+y در الگوریتم ۲-۱۰

n	x	y	x + y	Number of Digits in x + y
4	10	10	20	$2 = \frac{n}{2}$
4	99	99	198	$3 = \frac{n}{2} + 1$
8	1000	1000	2,000	$4 = \frac{n}{2}$
8	9999	9999	19,998	$5 = \frac{n}{2} + 1$

پیچیدگیهای زمانی خطی روی عناصر n دارند. بنابراین،

$$3W(\frac{n}{2}) + cn \le W(n) \le 3W(\frac{n}{2} + 1) + cn \qquad n, n>s \text{ توانی از } 2$$
$$W(s) = 0$$

که در آن S کوچکتر یا مساوی مقدار آستانه و توانی از ۲ است؛ چرا که تمامی ورودی‌ها در این حالت، توانی از ۲ می‌باشند. در حالتی که n توانی از ۲ نباشد، نوشتن یک بازگشت همانند حالت قبلی امکانپذیر است اما باید از جزء صحیح بالا و پائین برای این منظور استفاده کنیم. با استفاده از استقرایی نظیر مثال B-۲۵ در ضمیمهٔ B می‌توانیم نشان دهیم که W(n) نهایتاً غیرنزولی است. بنابراین، با توجه به نامساوی سمت چپ در بازگشت فوق و قضیه B-۲۶ استنباط می‌شود که

$$W(n) \in \Omega(n^{\log_2 3})$$

همچنین می‌توانیم نشان دهیم

$$W(n) = W'(n - 2) \in o(n^{\lg_2 3})$$

که با ترکیب دو نتیجه فوق داریم:

$$W(n) \in \Theta(n^{\lg_2 3}) \approx \Theta(n^{1/58})$$

بوروین و مونرو در سال ۱۹۷۵ با استفاده از تبدیل فوریه سریع، یک الگوریتم $\Theta(n(\lg n)^2)$ برای ضرب اعداد صحیح بزرگ ارائه دادند. می‌توان الگوریتمهایی برای سایر عملیات نظیر تقسیم و جذر روی اعداد صحیح بزرگ نوشت که پیچیدگی زمانی آنها مشابه عمل ضرب باشد.

۲-۷ تعیین مقادیر آستانه

همانطوری که در بخش ۱-۲ بحث شد، بازگشت به سربار نیاز دارد. اما اگر بخواهیم، بعنوان مثال، فقط ۸ کلید را مرتب کنیم، آیا واقعاً می‌ارزد که این سربار را به خاطر الگوریتم $\Theta(n \lg n)$ بپذیریم و یا اینکه الگوریتم $\Theta(n^2)$ را ترجیح می‌دهیم؟ یا شاید برای چنین n کوچکی، مرتب‌سازی تبادلی (الگوریتم ۳-۱) سریعتر از مرتب‌سازی ادغامی بازگشتی خواهد بود؟ می‌خواهیم روشی ارائه دهیم که تعیین می‌کند به ازاء چه مقادیری از n، الگوریتمی دیگر حداقل با همان سرعت فراخوانی الگوریتمی که نمونه‌ها را به نمونه‌های کوچکتر تقسیم می‌کند، عمل می‌نماید. این مقادیر، به الگوریتم تقسیم و غلبه (الگوریتم جانشین) و کامپیوتر بکارگیرنده این الگوریتم‌ها بستگی دارند. هدف این است که یک **مقدار آستانهٔ بهینه** از n بدست آوریم. این مقدار، یک اندازهٔ نمونه است بطوری که برای هر نمونه کوچکتر، حداقل با همان سرعت فراخوانی الگوریتمی دیگر، که نمونه را بیشتر تقسیم می‌کند، خواهد بود و برای هر اندازهٔ نمونه بزرگتر، تقسیم نمونه با سرعتی بیشتر انجام خواهد شد. با وجود این، آنچنانکه خواهیم دید، همیشه یک مقدار آستانهٔ بهینه وجود ندارد. حتی اگر تجزیه و تحلیل ما، یک مقدار آستانه بهینه را مشخص نکند، می‌توانیم با بکارگیری نتایج تحلیل، مقدار آستانه‌ای را بدست آوریم. سپس الگوریتم تقسیم و غلبه را طوری تغییر دهیم که وقتی n به مقدار آستانه رسید، دیگر تقسیم نمونه صورت نگیرد؛ در عوض الگوریتمی دیگر فراخوانی شود. تاکنون کاربرد مقادیر آستانه را در الگوریتم‌های ۸-۲، ۹-۲ و ۱۰-۲ دیده‌ایم.

برای تعیین یک مقدار آستانه باید کامپیوتری را درنظر بگیریم که الگوریتم بر روی آن بکار گرفته شده است. این تکنیک، به هنگام استفاده از مرتب‌سازی ادغامی و تبادلی نشان داده شده است. در این تحلیل، از پیچیدگی زمانی بدترین حالت مرتب‌سازی ادغامی استفاده می‌کنیم. در واقع، سعی بر این است که رفتار بدترین حالت الگوریتم را بهینه کنیم. در هنگام تحلیل مرتب‌سازی ادغامی مشخص شد که بدترین حالت با بازگشت زیر بیان می‌شود:

$$W(n) = W(\lfloor \frac{n}{2} \rfloor) + W(\lceil \frac{n}{2} \rceil) + n - 1$$

فرض کنید که الگوریتم مورد استفاده، مرتب‌سازی ادغامی۲ (الگوریتم ۴-۲) و مدت زمان صرف شده برای تقسیم و ترکیب مجدد یک نمونه به اندازه n توسط کامپیوتر بکارگیرنده این الگوریتم، برابر ۳۲n میکروثانیه است. توجه داریم که مدت زمان لازم برای تقسیم یک نمونه و ترکیب مجدد آن شامل مدت زمان محاسبه مقدار mid، مدت زمان انجام عملیات پشته‌ای برای دو فراخوانی بازگشتی و مدت زمان ادغام دو زیرآرایه می‌باشد. از آنجائیکه چندین جزء بر مدت زمان تقسیم و ترکیب مجدد یک نمونه مؤثر هستند، بعید بنظر می‌رسد که زمان کل صرف شده، مقدار ثابتی از n باشد. به هر حال، فرض کنید که شرایط تا حد امکان ساده باشد. به دلیل اینکه عبارت n-۱ در فرمول محاسباتی $W(n)$ همان مدت زمان ترکیب مجدد یک نمونه است، لذا برابر ۳۲ میکروثانیه خواهد بود. بدین ترتیب، برای مرتب‌سازی ادغامی۲ داریم:

$$W(n) = W(\lfloor \frac{n}{۲} \rfloor) + W(\lceil \frac{n}{۲} \rceil) + ۳۲n\ \mu s$$

چون برای اندازه ورودی ۱، تنها یک بررسی در شرط نهایی انجام شده است، لذا فرض می‌کنیم که W(۱)، اساساً برابر صفر است. برای سادگی کار، ابتدا بحث خود را به nای محدود می‌کنیم که توانی از ۲ است. در این حالت بازگشت زیر را داریم:

$$W(n) = ۲W(\frac{n}{۲}) + ۳۲n\ \mu s \qquad \text{n توانی از ۲،}$$
$$W(۱) = ۰\ \mu s$$

با استفاده از تکنیکهای ضمیمه B می‌توان بازگشت فوق را حل نمود. جواب چنین است:

$$W(n) = ۳۲n\ lgn\ \ \mu s$$

فرض کنید بر روی همان کامپیوتر، مرتب‌سازی تبادلی برای یک نمونه به اندازه n، دقیقاً n(n−۱)/۲ میکروثانیه وقت می‌گیرد. گاهی اوقات دانشجویان به اشتباه تصور می‌کنند که نقطه بهینه، یعنی جائیکه مرتب‌سازی ادغامی۲ باید مرتب‌سازی تبادلی را فراخوانی کند، می‌توانند از حل نامساوی زیر بدست آید:

$$n(n-۱)/۲\ \mu s < ۳۲n\ lgn\ \mu s$$

که جواب برابراست با

$$n < ۲۵۷$$

عقیده آنها بر این است که وقتی $n < ۲۵۷$ باشد، بهتر است مرتب‌سازی تبادلی فراخوانی شود. در غیر اینصورت مرتب‌سازی ادغامی۲ فراخوانی گردد. این تحلیل، تقریبی است چرا که مبنا را بر توان ۲ بودن n گزاردیم. نکته مهمتر اینکه این تحلیل نادرست است زیرا نقطه بهینه بدست آمده تنها به ما می‌گوید که اگر از مرتب‌سازی ادغامی۲ استفاده کنیم و تقسیم را تا n = ۱ ادامه دهیم، آنگاه برای $n < ۲۵۷$، مرتب‌سازی تبادلی بهتر خواهد بود. می‌خواهیم از مرتب‌سازی ادغامی ۲ استفاده کنیم و تقسیم را تا آنجاادامه دهیم که فراخوانی مرتب‌سازی تبادلی بهتر از تقسیم بیشتر نمونه‌ها باشد. درک انتزاعی این نکته که نقطه بهینه بایستی کوچکتر از ۲۵۷ باشد کمی دشوار است. مثال واقعی زیر، نقطه‌ای را تعیین می‌کند که در آن، مرتب‌سازی تبادلی کاراتر از تقسیم بیشتر نمونه‌ها می‌باشد. از این پس خود را محدود به nای نمی‌کنیم که توانی از ۲ باشد.

مثال ۷-۲ می‌خواهیم مقدار آستانه بهینه برای الگوریتم ۲-۲ (مرتب‌سازی ادغامی۲) را به هنگام فراخوانی الگوریتم ۳-۱ (مرتب‌سازی تبادلی) مشخص کنیم. فرض کنید مرتب‌سازی ادغامی۲ را به گونه‌ای تغییر داده‌ایم که برای $n \le t$ (t، مقدار آستانه است)، مرتب‌سازی تبادلی فراخوانی می‌شود. با فرض بکارگیری

$$W(n)=\begin{cases}\dfrac{n(n-1)}{2} & n\le t\\ W(\lfloor\frac{n}{2}\rfloor)+W(\lceil\frac{n}{2}\rceil)+32n\ \mu s & n>t\end{cases} \qquad (۲-۵)$$

مقدار بهینه t، مقداری است که از تساوی دو عبارت بالا و پائین در معادله فوق بدست می‌آید؛ چراکه در این نقطه، فراخوانی مرتب‌سازی تبادلی به همان کارایی تقسیم بیشتر نمونه‌ها است. بنابراین، برای تعیین مقدار بهینه t بایستی عبارت زیر را حل کنیم:

$$W(\lfloor\frac{t}{2}\rfloor)+W(\lceil\frac{t}{2}\rceil)+32t=\frac{t(t-1)}{2} \qquad (۲-۶)$$

چون $\lfloor\frac{t}{2}\rfloor$ و $\lceil\frac{t}{2}\rceil$ هر دو کوچکتر یا مساوی t هستند، زمان اجرای نمونه‌ای به اندازه هر یک از اینها، با عبارت بالایی معادله ۵-۲ محاسبه می‌شود. بنابراین،

$$W(\lceil t/2\rceil)=\frac{\lceil t/2\rceil(\lceil t/2\rceil-1)}{2} \quad \text{و} \quad W(\lfloor t/2\rfloor)=\frac{\lfloor t/2\rfloor(\lfloor t/2\rfloor-1)}{2}$$

با جایگزینی این تساویها در تساوی ۶-۲ داریم:

$$\frac{\lfloor t/2\rfloor(\lfloor t/2\rfloor-1)}{2}+\frac{\lceil t/2\rceil(\lceil t/2\rceil-1)}{2}+32t=\frac{t(t-1)}{2} \qquad (۲-۷)$$

بطور کلی، در یک معادله با جزء صحیح بالا و پائین، وقتی که مقدار t را زوج یا فرد در نظر بگیریم، می‌توانیم جوابهای متفاوتی بدست آوریم و به همین دلیل، همیشه یک مقدار آستانه بهینه وجود نخواهد داشت. اگر یک مقدار زوج برای t قرار دهیم، با درنظر گرفتن $t/2=\lceil t/2\rceil=\lfloor t/2\rfloor$ و حل معادله ۷-۲ خواهیم داشت:

$$t=128$$

و اگر یک مقدار فرد برای t قرار دهیم، بادرنظر گرفتن $(t-1)/2=\lfloor\frac{t}{2}\rfloor$ و $(t+1)/2=\lceil\frac{t}{2}\rceil$، و حل معادله ۷-۲ خواهیم داشت:

$$t=128/008$$

بنابراین، ما یک مقدار آستانه بهینه برابر ۱۲۸ داریم.

اکنون مثالی ارائه می‌دهیم که در آن هیچ مقدار آستانه بهینه‌ای وجود ندارد.

مثال ۸-۲ فرض کنید برای یک الگوریتم تقسیم و غلبه که بر روی کامپیوتر بخصوصی در حال اجراست، مقدار T(n) را از رابطه زیر تعیین می‌کنیم:

$$T(n)=3T(\lceil\frac{n}{2}\rceil)+16n\ \mu s$$

که در آن مدت زمان لازم برای تقسیم و ترکیب مجدد یک نمونه به اندازهٔ n، 16n میکروثانیه است.

همچنین فرض کنید که بر روی همان کامپیوتر، یک الگوریتم تکرار برای پردازش نمونه‌ای به اندازه n، به n^2 میکروثانیه زمان نیاز دارد. لذا برای تعیین مقدار t در جائیکه الگوریتم تکرار فراخوانی می‌شود، بایستی رابطه زیر را حل کنیم:

$$3T(\lceil \frac{t}{2} \rceil) + 16t = t^2$$

چون $t \leq (\lceil \frac{t}{2} \rceil)$ است، لذا الگوریتم تکرار زمانی فراخوانی می‌شود که ورودی به همین اندازه باشد و این بدین معناست که

$$T(\lceil \frac{t}{2} \rceil) = \lceil \frac{t}{2} \rceil^2$$

بنابراین بایستی رابطه زیر را حل کنیم:

$$\lceil \frac{t}{2} \rceil^2 + 16t = t^2$$

اگر یک مقدار زوج را به جای t قرار دهیم (با قرار دادن $\lceil \frac{t}{2} \rceil = \frac{t}{2}$) و مسئله را حل کنیم، آنگاه

$$t = ۶۴$$

اگر یک مقدار فرد را به جای t قرار دهیم (با قرار دادن $\lceil \frac{t}{2} \rceil = \frac{t+1}{2}$) و مسئله را حل کنیم، آنگاه

$$t = ۷۰/۰۴$$

از آنجائیکه این دو مقدار t با هم مساوی نیستند، لذا هیچ مقدار آستانه بهینه‌ای وجود ندارد. این بدین معناست که اگر اندازهٔ نمونه، یک عدد صحیح زوج بین ۶۴ و ۷۰ باشد، تقسیم نمونه به نمونه‌های کوچکتر و ترکیب مجدد آنها بهینه‌تر است، در حالیکه اگر اندازه نمونه، یک عدد صحیح فرد بین ۶۴ و ۷۰ باشد، الگوریتم تکرار کاراتر خواهد بود. برای اندازه‌های کوچکتر از ۶۴، الگوریتم تکرار و برای اندازه‌های بزرگتر از ۷۰، روش تقسیم و غلبه همواره کارایی بیشتری خواهند داشت. جدول ۵-۲ اندازه نمونه‌های مختلف را نشان می‌دهد.

جدول ۵-۲ اندازه نمونه‌های مختلف که مقدار آستانه برای n زوج، ۶۴ و برای n فرد، ۷۰ است.

n	n^2	$3\left\lceil \frac{n}{2} \right\rceil^2 + 16n$
62	3844	3875
63	3969	4080
64	4096	4096
65	4225	4307
68	4624	4556
69	4761	4779
70	4900	4795
71	5041	5024

۲-۸ چه وقت از روش تقسیم و غلبه استفاده نکنیم؟

تا حد امکان باید از بکارگیری روش تقسیم و غلبه در دو حالت زیر اجتناب کنیم:

۱- یک نمونه به اندازه n، به دو یا چند نمونه به اندازه تقریبی n تقسیم می‌شود.

۲- یک نمونه به اندازه n، به حدوداً n نمونه به اندازه n/c تقسیم می‌شود که c یک عدد ثابت است.

در اولین مورد، به یک الگوریتم زمان-نمایی و در مورد دوم به الگوریتمی از نوع $\Theta(n^{\log n})$ هدایت می‌شویم که هیچکدام از اینها برای مقادیر بزرگ n قابل قبول نیستند. می‌توان دید که چرا این موارد منجر به کاهش راندمان می‌شوند. برای مثال، مورد اول شبیه به این است که ناپلئون، ارتش ۳۰،۰۰۰ نفری دشمن را به دو ارتش ۲۹،۹۹۹ نفری تقسیم کند(اگر چنین چیزی ممکن بود). وی با این کار تعداد سربازان دشمن را به جای نصف کردن، به دو برابر می‌رساند که در این صورت واترلو را خیلی زودتر ملاقات می‌کرد.

الگوریتم ۶-۱ (عنصر nام فیبوناچی، بازگشتی)، یک الگوریتم تقسیم وغلبه است که نمونه‌ای که عنصر nام را محاسبه می‌کند را به دو نمونه کوچکتر که به ترتیب عنصر n-۱ام و عنصر n-۲ام را محاسبه می‌کند، تقسیم می‌نماید. اگرچه n به عنوان اندازه ورودی آن الگوریتم نیست ولی دقیقاً وضعیتی مشابه اندازه ورودی توصیف شده دارد. یعنی تعداد عناصر محاسبه شده توسط الگوریتم ۶-۱، نمایی از n است؛ در حالیکه تعداد عناصر محاسبه شده توسط الگوریتم ۷-۱ (عنصر nام فیبوناچی، بازگشتی) بر روی n، وضعیت خطی دارد.

از طرف دیگر گاهی اوقات لازم است که یک مسئله به صورت نمایی باشد. در چنین مواقعی دلیلی وجود ندارد که از بکارگیری روش تقسیم و غلبه اجتناب کنیم. به مسئله برجهای هانوی که در تمرین ۱۷ ارائه شده است، توجه کنید. به طور خلاصه، مسئله با حرکت n حلقه از یک میله به دیگری با رعایت برخی ضوابط خاص در چگونگی حرکت حلقه‌ها درگیر است. در تمرینات نشان خواهید داد که توالی حرکتها، که از اعمال روش تقسیم و غلبه استاندارد برای مسئله بدست آمده است، بر روی عنصر n به صورت نمایی است و البته این کاراترین ترتیب برای حل مسئله می‌باشد.

تمرینات

بخش ۱-۲

۱- با استفاده از جستجوی دودویی (الگوریتم ۱-۲)، عدد صحیح ۱۲۰ را در لیست (آرایه) زیر جستجو کنید. مراحل را قدم به قدم نشان دهید.

۱۲ ۳۴ ۳۷ ۴۵ ۵۷ ۸۲ ۹۹ ۱۲۰ ۱۳۴

۲- فرض کنید که می‌خواهیم با استفاده از جستجوی دودویی (الگوریتم ۱-۲)، یک لیست شامل ۷۰۰ میلیون عنصر را مورد جستجو قرار دهیم. مشخص کنید ماکزیمم تعداد مقایساتی که این الگوریتم باید برای جستجوی عنصری که در لیست وجود ندارد انجام دهد، چقدر است؟

۳- فرض کنیم که همواره یک جستجوی با موفقیت را انجام می‌دهیم. بدین معنا که در الگوریتم ۱-۲ می‌توان عنصر x را در لیست S پیدا کرد. در اینصورت الگوریتم را با حذف دستورات غیرضروری اصلاح کنید.

۴- نشان دهید که پیچیدگی زمانی بدترین حالت جستجوی دودویی (الگوریتم ۲-۱) با رابطهٔ زیر محاسبه می‌شود:

$$W(n) = \lfloor lg\, n \rfloor + 1$$

که در آن n به توانی از ۲ محدود نمی‌باشد. *راهنمایی:* ابتدا نشان دهید که معادله بازگشتی برای W(n) برابراست با

$$W(n) = 1 + W\left(\lfloor \frac{n}{2} \rfloor\right) \quad \text{برای } n > 1$$
$$W(1) = 1$$

برای انجام این کار، مقادیر زوج و فرد را به طور جداگانه برای n در نظر بگیرید. سپس از استقراء، برای حل معادله بازگشتی استفاده نمائید.

۵- فرض کنید که در الگوریتم ۱-۲ (سطر ۴)، تابع جداسازی(Split) به عبارت mid=low تبدیل شود. استراتژی جستجوی جدید را تشریح کرده، کارایی آنرا تحلیل نموده و نتایج را به وسیله نمادهای ترتیب نشان دهید.

۶- الگوریتمی بنویسید که یک لیست n عنصری را با تقسیم به سه زیرلیست تقریباً n/۳ عنصری، مورد جستجو قرار می‌دهد. این الگوریتم، زیرلیستی که احتمالاً عنصر مورد جستجو در آن قرار دارد را پیدا کرده و آن را به سه زیر لیست با اندازه‌های تقریباً یکسان تقسیم می‌کند. این فرآیند تا زمانی ادامه می‌یابد که عنصر مورد نظر پیدا شده یا مشخص شود که آن در لیست وجود ندارد. الگوریتم را تحلیل نموده و نتایج را با نمادهای ترتیب نمایش دهید.

۷- با استفاده از روش تقسیم و غلبه، الگوریتمی بنویسید که بزرگترین عنصر یک لیست n عنصری را پیدا کند. الگوریتم ارائه شده را تحلیل نموده و نتایج را با نمادهای ترتیب نشان دهید.

بخش ۲-۲

۸- با بکارگیری مرتب‌سازی ادغامی(الگوریتم ۲-۲ و ۴-۲)، لیست زیر را مرتب کنید. مراحل را قدم به قدم نشان دهید.

۲۴۰ ۹ ۱۲ ۱۵۰ ۵۶ ۱۸۹ ۳۴ ۱۲۳

۹- درخت فراخوانی‌های بازگشتی تمرین ۸ را نشان دهید.

۱۰- برای مسئله زیر، یک الگوریتم بازگشتی بنویسید که پیچیدگی زمانی بدترین حالت آن بدتر از $\Theta(n \lg n)$ نباشد. یک لیست با n عدد صحیح مثبت مجزا داده شده است. لیست را به دو

زیر لیست با اندازه‌های n/۲ تقسیم نمائید بطوری که اختلاف بین مجموع اعداد صحیح در دو زیر لیست، ماکزیمم باشد. می‌توانید n را مضربی از ۲ در نظر بگیرید.

۱۱- یک الگوریتم غیربازگشتی برای مرتب‌سازی ادغامی (الگوریتم ۲-۲ و ۴-۲) بنویسد.

۱۲- نشان دهید که معادله بازگشتی برای پیچیدگی زمانی بدترین حالت مرتب‌سازی ادغامی (الگوریتم‌های ۲-۲ و ۴-۲) به صورت زیراست:

$$W(n) = W(\lfloor \frac{n}{۲} \rfloor) + W(\lceil \frac{n}{۲} \rceil) + n - ۱$$

که در آن n به توانی از ۲ محدود نمی شود.

۱۳- الگوریتمی بنویسید که یک لیست n عنصری را، با تقسیم به سه زیرلیست تقریباً n/۳ عنصری مرتب نماید. این الگوریتم هر یک از زیر لیست‌ها را به طور بازگشتی مرتب نموده و در نهایت آنها را با هم ادغام می‌کند. الگوریتم را تحلیل نموده و نتایج را با نمادهای ترتیب نشان دهید.

بخش ۳-۲

۱۴- رابطه بازگشتی زیر داده شده است:

$$T(n) = ۷T(\frac{n}{۵}) + ۱۰n \qquad \text{برای } n > ۱$$
$$T(۱) = ۱$$

T(۶۲۵) را پیدا کنید.

۱۵- به روال Solve(P, I, O)، که در زیر آمده است، توجه کنید. این الگوریتم، مسئله P را با پیدا کردن خروجی (جواب) O متناظر با هر ورودی I حل می‌کند.

```
Procedure solve(p, I, o)
begin
   If size(I) = 1  then
      find solution O directly
   else
      partition I into 5 inputs I1, I2, I3, I4, I5, where
      size (Ij) := size (I)/3 for j =1,...,5;
      for j = 1 to 5 do
        solve (p, Ij, Oj)
      end;
     combine O1, O2, O3, O4, O5 to get O for P with input 1
   end
end;
```

فرض کنید g(n)، عمل مبنایی برای تقسیم نمونه‌ها و ترکیب مجدد آنها باشد. همچنین هیچ عمل مبنایی برای نمونه‌ای به اندازه ۱ وجود نداشته باشد.

a) یک معادله بازگشتی T(n) برای تعداد عملیات مبنایی مورد نیاز جهت حل مسئله P بنویسید.
b) اگر $g(n) \in \Theta(n)$ باشد، جواب این معاله بازگشتی چیست؟ (اثبات لازم نیست.)
c) با فرض اینکه $g(n) = n^2$، معادله بازگشتی را برای n=۲۷ حل نمائید.
d) یک جواب عمومی برای n، وقتی که توانی از ۳ باشد، ارائه کنید.

۱۶- فرض کنید که در یک الگوریتم تقسیم و غلبه، یک نمونه مسئلهٔ به اندازه n را همیشه به ۱۰ زیر نمونه به اندازهٔ n/۳ تقسیم می‌کنیم و مراحل تقسیم و ترکیب به مدت زمانی که در $\Theta(n^2)$ است، نیازمند می‌باشد. یک معادله بازگشتی برای زمان اجرای T(n) نوشته و آن را حل نمائید.

۱۷- یک الگوریتم تقسیم و غلبه برای مسئله برجهای هانوی بنویسید. مسئله برج هانوی شامل سه میله و n دیسک به اندازه‌های متفاوت است. هدف، جابجاکردن دیسکهایی که به ترتیب اندازه‌شان بر روی میله قرار دارند، از یکی از سه میله به میله دیگر با کمک میله کمکی سوم می‌باشد. مسئله بایستی با توجه به قوانین زیر حل شود: ۱) وقتی دیسکی حرکت داده می‌شود، باید روی یکی از سه میله قرار داده شود. ۲) در هر زمان می‌توان فقط یک دیسک و آنهم دیسک بالایی میله را حرکت داد. ۳) هیچگاه دیسک بزرگتر روی دیسک کوچکتر قرار نمی‌گیرد.

a) نشان دهید که برای الگوریتم شما، $S(n) = 2^n - 1$ است.[در اینجا S(n)، معرف تعداد مراحل (حرکت‌ها) برای ورودی n دیسک می‌باشد.]
b) ثابت کنید که هر الگوریتم دیگر، حداقل به همان تعداد حرکات بخش (a) نیاز دارد.

۱۸- هنگامی که الگوریتم تقسیم و غلبه، یک نمونه مسئلهٔ به اندازه n را به زیر نمونه‌هایی به اندازه n/c تقسیم می‌کند، رابطه بازگشتی چنین است:

$$T(n) = aT(\frac{n}{c}) + g(n) \qquad \text{برای } n > 1$$
$$T(1) = d$$

که در آن g(n)، هزینه فرآیندهای تقسیم و ترکیب مجدد نمونه‌ها و d یک مقدار ثابت است. با فرض $n = c^k$

a) نشان دهید که

$$T(c^k) = d \times a^k + \sum_{j=1}^{k} [a^{k-j} \times g(c^j)]$$

b) رابطه بازگشتی را با درنظرگرفتن $g(n) \in \Theta(n)$ حل کنید.

بخش ۴-۲

۱۹- با بکارگیری مرتب‌سازی سریع (الگوریتم ۶-۲) لیست زیر را مرتب کنید. مراحل را نشان دهید.

۲۴۰ ۹ ۱۲ ۱۵۰ ۵۶ ۱۸۹ ۳۴ ۱۲۳

۲۰- درخت فراخوانی‌های بازگشتی تمرین ۱۹ را نشان دهید.

۲۱- درستی رابطه زیر را مشخص کنید.

$$\sum_{p=1}^{n} [A(p-1) + A(n-p)] = 2 \sum_{p=1}^{n} A(p-1)$$

این نتیجه در تحلیل پیچیدگی زمانی حالت میانی الگوریتم ۶-۲ (مرتب‌سازی سریع) استفاده می‌شود.

۲۲- نشان دهید که اگر

$$W(n) \leq \frac{(p-1)(p-2)}{2} + \frac{(n-p)(n-p-1)}{2}$$

آنگاه

$$W(n) \leq \frac{n(n-1)}{2} \qquad \text{برای } 1 \leq p \leq n$$

این نتیجه در تحلیل پیچیدگی زمانی بدترین حالت الگوریتم ۶-۲ استفاده می‌شود.

۲۳- یک الگوریتم غیربازگشتی برای مرتب‌سازی سریع (الگوریتم ۶-۲) بنویسید. الگوریتم را تحلیل نموده و نتایج را با نمادهای ترتیب نمایش دهید.

۲۴- فرض کنید که مرتب‌سازی سریع، اولین عنصر را به عنوان عنصر محوری درنظرمی گیرد.

a) یک لیست n عنصری (برای مثال، آرایه‌ای از ۱۰ عدد صحیح) بنویسید که نمایانگر بدترین حالت حل مسئله باشد.

b) یک لیست n عنصری (برای مثال، آرایه‌ای از ۱۰ عدد صحیح) بنویسید که نمایانگر بهترین حالت حل مسئله باشد.

بخش ۵-۲

۲۵- نشان دهید که با اعمال یک تغییر جزئی در الگوریتم ۴-۱ (ضرب ماتریس) می‌توان تعداد جمعهای انجام شده را به $n^3 - n^2$ تقلیل داد.

۲۶- در مثال ۴-۲، ضرب دو ماتریس را به روش استراسن انجام دادیم. صحت حاصلضرب را بررسی کنید.

۲۷- با بکارگیری الگوریتم استاندارد ضرب ماتریسی، چه تعداد عمل ضرب بایستی برای محاسبه حاصلضرب دو ماتریس ۶۴×۶۴ باید انجام شود؟

۲۸- چه تعداد عمل ضرب باید برای محاسبه حاصل ضرب دو ماتریس ۶۴×۶۴ با استفاده از استراسن (الگوریتم ۸-۲) انجام شود؟

۲۹- یک معادله بازگشتی برای الگوریتم استراسن تغییر یافته که توسط ساموئل وینوگراد ارائه شده و در آن از ۱۵ جمع/تفریق به جای ۱۸ تا استفاده شده است، بنویسید. معادله بازگشتی را حل نموده و صحت جواب معادله را با بکارگیری پیچیدگی زمانی نشان داده شده در پایان بخش ۵-۲ بررسی کنید.

بخش ۶-۲

۳۰- با استفاده از الگوریتم ۱۰-۲ (ضرب اعداد صحیح بزرگ)، حاصلضرب ۱۲۵۳ و ۲۳۱۰۳ را پیدا کنید.

۳۱- چه تعداد ضرب نیاز است تا حاصلضرب دو عدد صحیح تمرین ۳۰ مشخص شود؟

۳۲- الگوریتم هایی بنویسد که عملیات زیر را انجام دهند.

$$U \times ۱۰^m,\ U\ divide\ ۱۰^m,\ U\ rem\ ۱۰^m$$

که در آن، U یک عدد صحیح بزرگ، m یک عدد صحیح غیرمنفی، divide خارج قسمت و rem باقیمانده تقسیم دو عدد می‌باشد. الگوریتم را تجزیه و تحلیل نموده و نشان دهید که این عملیات می‌توانند به صورت زمان-خطی انجام شوند.

۳۳- الگوریتم ۹-۲ (ضرب اعداد صحیح بزرگ) را به گونه‌ای تغییر دهید که هر عدد صحیح n رقمی را تقسیم کند به

a) سه عدد صحیح کوچکتر، هر یک با n/۳ رقم (فرض کنید $n = ۳^k$)

b) چهار عدد صحیح کوچکتر، هر یک با n/۴ رقم (فرض کنید $n = ۴^k$)

الگوریتم را تحلیل نموده و پیچیدگی‌های زمانی آن را با نمادهای ترتیب نمایش دهید.

بخش ۷-۲

۳۴- الگوریتمهای مرتب‌سازی سریع و مرتب‌سازی تبادلی را برای مرتب نمودن یک لیست n عنصری بر روی کامپیوترتان بکار بگیرید. یک حد پائین برای n مشخص نمائید که بکارگیری Quicksort به همراه سربار آن را توجیه کند.

۳۵- الگوریتم‌های ضرب استاندارد و استراسن برای ضرب دو ماتریس $n \times n$ $(n = ۲^k)$ را بر روی کامپیوترتان پیاده سازی کنید. حد پائینی برای n مشخص کنید که دلیلی برای بکارگیری الگوریتم استراسن به همراه سربارش وجود داشته باشد.

۳۶- فرض کنید که روی کامپیوتر بخصوصی، $۱۲n^۲$ میکروثانیه زمان نیاز است تا یک نمونهٔ به اندازه n در حالت الگوریتم ۸-۲ (استراسن)، تقسیم و مجدداً ترکیب شود. توجه داریم که این زمان شامل مدت زمان لازم جهت انجام همه جمع‌ها و تفریق‌ها است. اگر برای ضرب دو ماتریس n×n با استفاده از الگوریتم استاندارد به $n^۳$ میکروثانیه وقت نیاز داشته باشیم، مقدار آستانه‌ای را تعیین کنید که در آن بایستی الگوریتم استاندارد به جای تقسیم بیشتر نمونه‌ها فراخوانی شود. آیا مقدار آستانه بهینه منحصربفردی یافت می‌شود؟

بخش ۸-۲

۳۷- با استفاده از روش تقسیم و غلبه، الگوریتمی بازگشتی بنویسید که فاکتوریل n را محاسبه کند. اندازهٔ ورودی را تعیین نمایید. آیا تابع شما دارای پیچیدگی زمانی به صورت نمایی است؟

۳۸- فرض کنید که در یک الگوریتم تقسیم و غلبه، یک نمونه مسئله به اندازه n را همیشه به n زیر نمونه به اندازه n/۳ تقسیم می‌کنیم و مراحل تقسیم و غلبه به مدت زمانی که به صورت زمان-خطی است، نیازمند می‌باشند. یک معادله بازگشتی برای زمان اجرای T(n) نوشته و آن را حل کنید.جواب را با نمادهای ترتیب نشان دهید.

تمرینات اضافی

۳۹- یک الگوریتم کارا بنویسید که عنصری را در یک جدول n×m (آرایه دوبعدی) جستجو کند. این جدول در راستای سطرها و ستون‌ها مرتب شده است، بطوری که

$$Table\ [i][j] \le Table\ [i][j+1]$$

$$Table\ [i][j] \le Table\ [i+1][j]$$

۴۰- فرض کنید که در یک تورنمنت حذفی، $n = 2^k$ تیم وجود دارد که در مرحله اول، n/۲ بازی صورت می‌گیرد و $n/2 = 2^{k-1}$ بازیکن پیروز به مرحله دوم راه پیدا می‌کنند و الی آخر.

a) یک معادله بازگشتی برای تعداد مراحل در تورنمنت ارائه دهید.

b) اگر تعداد تیم‌ها به ۶۴ برسد، چند مرحله در تورنمنت وجود خواهد داشت؟

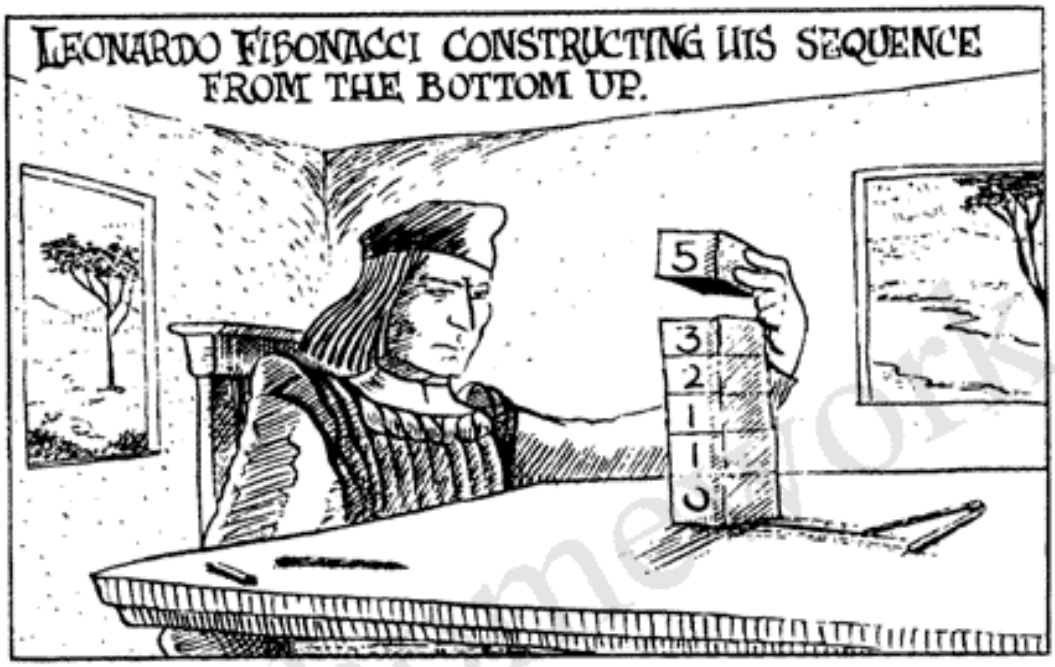
c) معادله بازگشتی بخش (a) را حل نمائید.

۴۱- یک الگوریتم بازگشتی $\Theta(n \lg n)$ بنویسید که باقیماندهٔ تقسیم x^n بر p را محاسبه نماید. سه عدد صحیح x ، n و p پارامترهای این الگوریتم می‌باشند. برای سادگی، می‌توانید فرض کنید که n توانی از ۲ است؛ یعنی $n = 2^k$، که در آن k یک عدد صحیح مثبت است.

۴۲- با بکارگیری روش تقسیم و غلبه، یک الگوریتم بازگشتی بنویسید که ماکزیمم مجموع را در هر زیرلیست از لیست n، که شامل اعداد حقیقی است، پیدا کند. الگوریتم را تحلیل نموده و نتایج را با نمادهای ترتیب نمایش دهید.

فصل ۳

برنامه‌نویسی پویا (Dynamic Programing)



به خاطر آورید که تعداد عناصر محاسبه شده توسط الگوریتم تقسیم و غلبه برای محاسبهٔ عنصر nام دنبالهٔ فیبوناچی (الگوریتم ۶-۱) نمایی از n بود، به این دلیل که روش تقسیم و غلبه، یک نمونه را به نمونه‌های کوچکتر تقسیم نموده و سپس آنها را کورکورانه حل می‌کرد. آنچنانکه در فصل ۲ گفته شد، این روش یک روش بالا به پائین است که در مسائلی نظیر مرتب‌سازی ادغامی (Mergesort) که در آن نمونه‌ها با هم بی‌ارتباط هستند، بکار می‌آید. این نمونه‌ها غیرمرتبط هستند زیرا هر یک شامل آرایه‌ای از کلیدها است که بایستی بطور جداگانه مرتب شود. اما در مسائلی نظیر عنصر nام فیبوناچی، نمونه‌های کوچکتر با هم مرتبطند. برای مثال، برای محاسبهٔ پنجمین عنصر فیبوناچی نیاز داریم که چهارمین و سومین عنصر فیبوناچی را محاسبه کنیم و به همین ترتیب، برای محاسبهٔ چهارمین و سومین عنصر فیبوناچی باید دومین عنصر این دنباله مشخص شده باشد. از آنجائیکه الگوریتم تقسیم و غلبه، محاسبهٔ این دو عنصر را به صورت جداگانه و مستقل از هم انجام می‌دهد، لذا عنصر دوم فیبوناچی بیش از یکبار محاسبه می‌شود. بطور کلی، در مسائلی که نمونه‌های کوچکتر با هم مرتبط هستند، الگوریتم تقسیم و غلبه اغلب نمونه‌های مشترک را به صورت تکراری محاسبه می‌کند که در نتیجه، یک الگوریتم غیر کارا خواهد بود.

برنامه‌نویسی پویا، روشی است که در این فصل مورد بررسی قرار می‌گیرد. این تکنیک از این نظر شبیه به تقسیم و غلبه است که یک نمونه از مسئله را به نمونه‌های کوچکتر تقسیم می‌نماید. در این روش، ابتدا نمونه‌های کوچکتر را حل کرده و نتایج حاصله را ذخیره می‌کنیم؛ سپس، هنگامی که به نتیجه‌ای نیاز داریم، به جای محاسبهٔ مجدد فقط آن را بازیابی می‌کنیم. عبارت "برنامه‌نویسی پویا" از تئوری کنترلی می‌آید و در این تعبیر، لفظ "برنامه‌نویسی" به معنای استفاده از یک آرایه برای تولید یک جواب می‌باشد. همانطوریکه در فصل ۱ اشاره شد، الگوریتم کارای ما برای محاسبهٔ عنصر nام فیبوناچی (الگوریتم ۷-۱)، یک مثال از برنامه‌نویسی پویا است. به خاطر دارید که این الگوریتم، با ایجاد دنباله‌ای از n+1 عنصر اول آرایهٔ f که از ۰ تا n شاخص‌دهی شده است، عنصر nام دنبالهٔ فیبوناچی را محاسبه می‌کند. در یک الگوریتم برنامه‌نویسی پویا، تولید جواب از پائین به بالا انجام می‌شود. به عبارت دیگر، برنامه‌نویسی پویا، یک روش **پائین به بالا** می‌باشد. گاهی اوقات همانند الگوریتم ۷-۱، پس از ارائه الگوریتم با استفاده از یک آرایه (یا دنباله‌ای از آرایه‌ها) می‌توانیم با بازبینی مجدد الگوریتم، بسیاری از فضاهای اشغال شدهٔ غیرضروری را اصلاح کنیم. مراحل تولید یک الگوریتم برنامه‌نویسی پویا به شرح زیر است:

۱- *ارائه* یک خاصیت بازگشتی جهت حل یک نمونه مسئله.

۲- حل یک نمونه مسئله به روش *پائین به بالا* و با شروع از حل نمونه‌های کوچکتر.

برای تشریح این مراحل، مثال ساده‌ای از برنامه‌نویسی پویا را در بخش ۱-۳ مطرح می‌کنیم. بخشهای دیگر، کاربردهای پیشرفته‌تر برنامه‌نویسی پویا را مورد بحث و بررسی قرار می‌دهند.

۱-۳ ضریب دوجمله‌ای

ضریب دوجمله‌ای، که در بخش ۷-A از ضمیمه A معرفی می‌شود، به صورت زیر است :

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad \text{برای } 0 < k < n$$

ما نمی‌توانیم برای مقادیری از n و k که کوچک نیستند، ضریب دوجمله‌ای را مستقیماً با استفاده از این تعریف محاسبه کنیم زیرا مقدار !n بسیار بزرگ می‌شود. حتی برای مقادیر متوسط n نیز چنین کاری ممکن نیست. در تمرینات ثابت می‌کنیم که

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ or } k = n \end{cases} \qquad (۳-۱)$$

ما می‌توانیم به جای محاسبهٔ !n یا !k از این خاصیت بازگشتی استفاده کنیم که این کار می‌تواند با استفاده از الگوریتم تقسیم وغلبه، به صورت زیر انجام شود.

الگوریتم ۱-۳ **ضریب دوجمله‌ای با استفاده از تقسیم و غلبه**

مسئله: ضریب دوجمله‌ای را محاسبه کنید.

ورودی: اعداد صحیح غیر منفی n و kکه در آن k<n است.

خروجی: *bin* ضریب دوجمله‌ای.

```
int bin (int n, int k)
{
  if (k == 0 || n == k)
    return 1;
  else
    return bin(n - 1, k - 1) + bin(n - 1, k);
}
```

این الگوریتم، همانند الگوریتم ۶-۱ (عنصر nام فیبوناچی، بازگشتی) بسیار غیرکارا است. در تمرینات ثابت خواهیم کرد که برای تعیین $\binom{n}{k}$، الگوریتم بایستی $\binom{n}{k}$ - ۱ ۲ عنصر را محاسبه کند. مشکل از آنجا ناشی می‌شود که در هر فراخوانی بازگشتی، نمونه‌های مشابه به صورت تکراری محاسبه می‌شوند. برای مثال $bin\,(n-1,k-1)$ و $bin(n-1,k)$، هر دو به نتیجهٔ $bin\,(n-2,k-1)$ نیازمند می‌باشند و این نمونه به طور مجزا برای هر یک از این دو فراخوانی بازگشتی حل می‌شود. همانطوریکه در بخش ۸-۲ ذکر شد، در روش تقسیم و غلبه، اگر یک نمونه را به دو نمونهٔ کوچکتر که تقریباً به بزرگی نمونهٔ اصلی هستند تقسیم کنیم، به روشی غیرکارا خواهیم رسید.

برنامه‌نویسی پویا، یک الگوریتم کارا برای رفع چنین مشکلاتی است. می‌خواهیم با استفاده از خاصیت بازگشتی ارائه شده در معادله ۱-۳، جوابی را در یک آرایهٔ B ارائه کنیم که $B[i][j]$ شامل $\binom{i}{j}$ باشد. مراحل ایجاد یک الگوریتم برنامه‌نویسی پویا برای این مسئله بصورت زیر است:

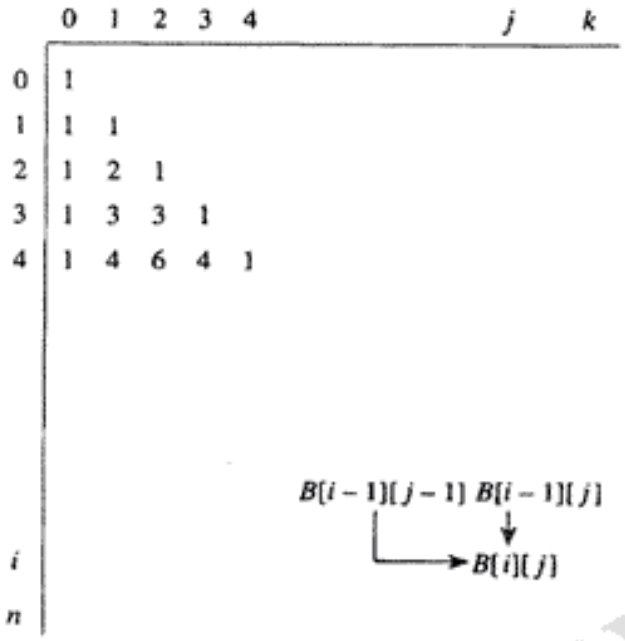
۱- *ارائه* یک خاصیت بازگشتی. عناصر آرایه B به فرم زیر در می‌آید:

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0 \text{ or } j = i \end{cases}$$

۲- حل یک نمونه مسئله به روش پائین به بالا بوسیله محاسبهٔ متوالی سطرهای B با شروع از اولین سطر.

مرحله ۲ در شکل ۱-۳ نشان داده شده است (شاید متوجه شده باشید که آرایهٔ مورد نظر به صورت مثلث پاسکال می‌باشد). در اینجا هر سطر، با استفاده‌از خاصیت بازگشتی مرحلهٔ ۱، از سطر قبلی خود محاسبه می‌شود. مقدار نهایی محاسبه شده $B[n][k]$، برابر $\binom{n}{k}$ باشد. مثال ۱-۳ این مراحل را نشان می‌دهد. توجه داشته باشید که در این مثال، تنها دو ستون اول محاسبه می‌شود زیرا در این مثال k=۲ است. در حالت کلی، به مقادیر سطرهای بالای ستون Kام نیاز داریم. مثال ۱-۳ مقدار $B[0][0]$ را

شکل ۱-۳ آرایه B برای محاسبهٔ ضریب دوجمله‌ای استفاده شده است.



محاسبه می‌کند چون ضریب دوجمله‌ای برای $n = k = ۰$ تعریف شده است، بنابراین، ممکن است یک الگوریتم، این مراحل را انجام دهد؛ حتی اگر به این مقادیر در محاسبهٔ دیگر ضرایب دوجمله‌ای نیازی نباشد.

مثال ۱-۳ $B[n][k] = \binom{n}{k}$ را محاسبه کنید.

محاسبهٔ سطر ۰: {این قسمت، فقط برای رعایت دقیق الگوریتم انجام شده}
{و بعد از این نیازی به محاسبهٔ $B[۰][۰]$ نمی‌باشد.}

$$B[۰][۰] = ۱$$

محاسبهٔ سطر ۱:

$$B[۱][۰] = ۱$$
$$B[۱][۱] = ۱$$

محاسبهٔ سطر ۲:

$$B[۲][۰] = ۱$$
$$B[۲][۱] = B[۱][۰] + B[۱][۱] = ۱ + ۱ = ۲$$
$$B[۲][۲] = ۱$$

محاسبهٔ سطر ۳:

$$B[۳][۰] = ۱$$
$$B[۳][۱] = B[۲][۰] + B[۲][۱] = ۱ + ۲ = ۳$$
$$B[۳][۲] = B[۲][۱] + B[۲][۲] = ۲ + ۱ = ۳$$

محاسبهٔ سطر ۴:

$$B[۴][۰] = ۱$$
$$B[۴][۱] = B[۳][۰] + B[۳][۱] = ۱ + ۳ = ۴$$
$$B[۴][۲] = B[۳][۱] + B[۳][۲] = ۳ + ۳ = ۶$$

مثال ۱-۳ بطور افزایشی، مقادیر بزرگتری از ضرایب دوجمله‌ای رابه ترتیب محاسبه می‌کند. در هر تکرار مقادیر مورد نیاز برای آن تکرار، از قبل محاسبه و ذخیره شده است. این روند، اساس روش برنامه‌نویسی پویا است. الگوریتم زیر از این روش در محاسبهٔ ضریب دوجمله‌ای استفاده می‌نماید.

الگوریتم ۲-۳ محاسبهٔ ضریب دوجمله‌ای با استفاده از برنامه‌نویسی پویا

مسئله: ضریب دوجمله‌ای را محاسبه کنید.

ورودی: اعداد صحیح غیر منفی n و k، بطوری که k<n است.

خروجی: bin2، ضریب دوجمله‌ای.

```
int bin2 (int n, int k)
{
  index i, j;
  int B[0..n][0..k];

  for (i = 0; i <= n; i++)
    for (j = 0; j <= minimum(i, k); j++)
      if (j == 0 || j == i)
        B[i][j] = 1;
      else
        B[i][j] = B[i - 1][j - 1] + B[i - 1][j];
  return B[n][k];
}
```

پارامترهای n و k، اندازه ورودی این الگوریتم نیستند بلکه تنها به عنوان ورودی شناخته می‌شوند. اندازه ورودی این الگوریتم، تعداد نمادهایی است که این پارامترها را کدگذاری می‌کنند. ما وضعیت مشابهی را در بخش ۳-۱، در بحث الگوریتمهای محاسبهٔ عنصر nام فیبوناچی مورد بحث قرار دادیم. به هرحال، می‌توانیم با تعیین میزان عملکرد الگوریتم بعنوان تابعی از n و k، کارایی آن را مشخص کنیم.

برای هر n و k معین، تعداد گذرهای انجام شده در حلقه for-j را محاسبه می‌کنیم. جدول زیر، تعداد گذرها را برای هر مقدار از i نشان می‌دهد

i	۰	۱	۲	۳	....	k	k+1	....	n
تعدادگذرها	۱	۲	۳	۴	....	k+1	k+1	....	k+1

بنابراین، تعداد کل گذرهای انجام شده برابر است با

$$۱+۲+۳+۴+...+k+\underbrace{(k+۱)+(k+۱)+...+(k+۱)}_{n-k+۱ \text{ مرتبه}}$$

با بکارگیری نتیجه مثال ۱-A از ضمیمه A، این عبارت معادل است با

$$\frac{k(k+۱)}{۲}+(n-k+۱)(k+۱)=\frac{(۲n-k+۲)(k+۱)}{۲}\in\theta(nk)$$

با استفاده از برنامه‌نویسی پویا به جای تقسیم و غلبه، در حقیقت یک الگوریتم با کارایی بسیار بالاتری ارائه نموده‌ایم. مطابق آنچه که قبلاً ذکر شد، برنامه‌نویسی پویا از این جهت که دارای یک خاصیت بازگشتی برای تقسیم یک نمونه به نمونه‌های کوچکتر است، به روش تقسیم و غلبه شباهت دارد. اما تفاوت آنها در این که برنامه‌نویسی پویا، بجای استفاده کورکورانه از بازگشت، از خاصیت بازگشتی جهت حل نمونه‌های تکراری به ترتیب و با شروع از کوچکترین نمونه استفاده می‌شود. در این روش، هر نمونهٔ کوچکتر فقط یکبار حل می‌شود. برنامه‌نویسی پویا یک روش خوب برای زمانی است که تقسیم و غلبه موجب کاهش کارایی یک الگوریتم می‌شود.

آسانترین روش برای ارائه الگوریتم ۲-۳ تولید یک آرایهٔ کامل دوبعدی B است. مشاهده می‌کنیم که وقتی یک سطر محاسبه می‌شود، دیگر نیازی به مقادیر سطرهای قبل از آن وجود ندارد. بنابراین، الگوریتم را می‌توان توسط تنها یک آرایهٔ تک‌بعدی که از ۰ تا k شاخص‌دهی شده است، نوشت. این تغییر را در تمرینات بررسی می‌کنیم. یک اصلاح دیگر روی الگوریتم، با استفاده از این واقعیت که $\binom{n}{k}=\binom{n}{n-k}$ است، انجام می‌شود.

۲-۳ الگوریتم فلوید جهت یافتن کوتاهترین مسیرها

یک مسئله متداول در حمل و نقل هوایی، تعیین کوتاهترین مسیر پرواز از یک شهر به شهری دیگر است در صورتیکه بین آن دو شهر، مسیر مستقیمی وجود نداشته باشد. بعداً الگوریتمی برای حل این مسئله و مسائل مشابه ارائه خواهیم داد. اما ابتدا، مروری اجمالی بر تئوری گرافها خواهیم داشت. شکل ۳.۲، یک گراف جهت‌دار و وزن‌دار را نشان می‌دهد. در نمایش تصویری گراف‌ها، دایره‌ها نشانگر **گره‌ها (vertices)** و یک خط از یک دایره به دایره دیگر، نشانگر یک **لبه (edge)** می‌باشد. اگر هر لبهٔ گراف

دارای جهت باشد، به آن **گراف جهت‌دار (digraph)** گوئیم. هنگام رسم لبه در چنین گرافی، از یک پیکان برای مشخص نمودن جهت لبه استفاده می‌کنیم. در یک گراف جهت‌دار، دو لبه می‌تواند بین دو گره وجود داشته باشد که جهت هر یک از لبه‌ها به سمت گره دیگر می‌باشد. برای مثال، در شکل ۲-۳ یک لبه از v_1 به v_2 و یک لبه از v_2 به v_1 وجود دارد. اگر به لبه‌ها مقادیری تخصیص یافته باشد، به آن مقادیر، **وزن** و به آن گراف، **گراف وزن‌دار** می‌گوئیم. ما فرض می‌کنیم که مقادیر وزن‌ها غیر منفی هستند. اگر چه به مقدار یک لبه، وزن گفته می‌شود ولی در بسیاری از کاربردها، این مقادیر معرف فاصله می‌باشند. در یک گراف جهت‌دار، به دنباله‌ای از گره‌ها که بین هر گره و گرهٔ بعدی یک لبه وجود داشته باشد، یک **مسیر (path)** اطلاق می‌شود. برای مثال در شکل ۲-۳، دنباله $[v_1, v_2, v_3]$ یک مسیر است زیرا هیچ لبه‌ای v_1 به v_2 و یک لبه از v_2 به v_3 وجود دارد دنباله $[v_3, v_2, v_1]$ یک مسیر نیست زیرا هیچ لبه‌ای از v_2 به v_1 وجود ندارد. به مسیری که از یک گره شروع شده و به خودش باز می‌گردد، یک **چرخه (cycle)** گوئیم. مسیر $[v_1, v_2, v_5, v_1]$ یک چرخه است. اگر گرافی شامل یک چرخه باشد، به آن **گراف چرخه‌ای** گفته می‌شود؛ در غیر اینصورت به آن **گراف غیرچرخه‌ای** گویند. به مسیری که از یک گره دو بار عبور نکند، **مسیر ساده** اطلاق می‌شود. مسیر $[v_1, v_2, v_3]$ در شکل ۲-۳، یک مسیر ساده و مسیر $[v_1, v_2, v_5, v_1, v_2]$، یک مسیر غیر ساده است. توجه کنید که هرگز در یک مسیر ساده، زیرمسیری که دارای چرخه باشد وجود ندارد. **طول (length)** یک مسیر در یک گراف وزن‌دار، برابر مجموع وزن‌های مسیر است. در یک گراف بی‌وزن، تعداد گره‌های موجود در مسیر بیانگر طول مسیر می‌باشد.

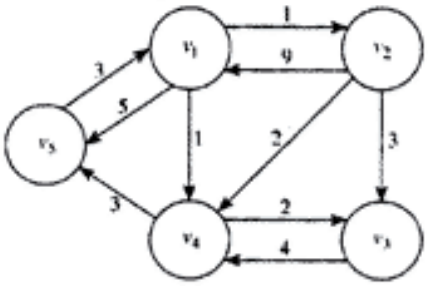
همانطوریکه قبلاً نیز اشاره شد، مسئله پیدا کردن کوتاهترین مسیر از هر گره به گره‌های دیگر، در بسیاری از مسائل کاربرد دارد. واضح است که کوتاهترین مسیر بایستی یک مسیر ساده باشد. در شکل ۲-۳، سه مسیر ساده باشد. در شکل ۳.۲، سه مسیر ساده از v_1 به v_3 به نامهای $[v_1, v_2, v_3]$، $[v_1, v_3]$ و $[v_1, v_2, v_4, v_3]$ وجود دارد. از آنجائیکه

$$\text{length } [v_1, v_2, v_3] = 1 + 3 = 4$$
$$\text{length } [v_1, v_3] = 1 + 2 = 3$$
$$\text{length } [v_1, v_2, v_4, v_3] = 1 + 2 + 2 = 5$$

است، لذا $[v_1, v_3]$ کوتاهترین مسیر از v_1 به v_3 است. یکی از متداولترین کاربردهای کوتاهترین مسیر، تعیین کوتاهترین راه بین شهرها است.

مسئله کوتاهترین مسیر، یک **مسئله بهینه‌سازی** بوده و ممکن است بیش از یک جواب کاندید برای یک نمونه از مسئله بهینه‌سازی وجود داشته باشد. هر جواب کاندید، دارای یک مقدار است و یک جواب برای یک نمونه، جواب کاندیدی است که دارای یک مقدار بهینه می‌باشد. این مقدار، با توجه به نوع مسئله می‌تواند می‌نیمم یا ماکزیمم باشد. در مسئله یافتن کوتاهترین مسیرها، یک جواب کاندید، یک مسیر از یک گره به گرهٔ دیگر است. در این مسئله، طول مسیر مقدار جواب کاندید است و مقدار بهینه، کوچکترین طول مسیر می‌باشد. به دلیل اینکه بیش از یک مسیر با کوتاهترین طول از یک گره به گرهٔ دیگر می‌تواند

شکل ۲-۳ یک گراف وزن‌دار و جهت‌دار.



وجود داشته باشد، لذا مسئله ما بایستی هر یک از این مسیرها را انتخاب کند. بدیهی است که یک الگوریتم برای این مسئله بایستی برای هر گره، طول همه مسیرها از آن گره به هر گرهٔ دیگر را تعیین نموده و سپس حداقل این طولها رامحاسبه کند. با وجود این، الگوریتم بدتر از زمان-نمایی است. برای مثال، فرض کنیدکه از هر گره به گرهٔ دیگر یک لبه وجود داشته باشد، آنگاه زیرمجموعه‌ای از همه مسیرها از یک گره به گرهٔ دیگر، مجموعه‌ای است از همهٔ آن مسیرهایی که از اولین گره شروع و به گرهٔ دیگر ختم می‌شوند و در این میان از تمامی گره‌های دیگر نیز می‌گذرند.

از آنجائیکه دومین گره در این مسیر می‌تواند هر یک از n-۲ گرهٔ دیگر باشد، سومین گره در این مسیر می‌تواند هر یک از n-۲ گرهٔ دیگر باشد،... و دومین گره به انتهای مسیر تنها می‌تواند یک گره باشد، لذا مجموع تعداد مسیرها از یک گره به گرهٔ دیگر که از تمامی گره‌های دیگر نیز عبور کنند، عبارت است از

$$(n-۲)(n-۳)\ldots=(n-۲)!$$

که بدتر از حالت نمایی است. در بسیاری از مسائل بهینه‌سازی با چنین وضعیتی روبرو می‌شویم؛ یعنی الگوریتمی که تمامی حالات ممکنه را در نظر می‌گیرد، زمان-نمایی یا بدتر از آن خواهد بود. هدف ما یافتن الگوریتمی با کارایی بهتر است. با استفاده از برنامه‌نویسی پویا می‌توانیم یک الگوریتم زمان-مربعی برای مسئله کوتاهترین مسیرها ارائه نمائیم. ابتدا یک الگوریتم که تنها طول کوتاهترین مسیرها را تعیین می‌کند، نوشته و سپس آن را جهت تولید کوتاهترین مسیرها تغییر می‌دهیم. یک گراف وزن‌دار شامل n گره را با آرایهٔ w به صورت زیر تعریف می‌کنیم:

$$W[i][j]=\begin{cases}\text{وزن لبه} & \text{اگر یک لبه از } v_i \text{ به } v_j \text{ وجود داشته باشد}\\ \infty & \text{اگر یک لبه از } v_i \text{ به } v_j \text{ وجود داشته باشد}\\ ۰ & \text{اگر } i=j\end{cases}$$

در صورتیکه یک لبه از v_i به v_j وجود داشته باشد، به گرهٔ v_j، **مجاور (adjacent)** گرهٔ v_i گویند. به همین دلیل، این آرایه به **ماتریس مجاور** نمایش گراف موسوم است. گراف شکل ۲-۳ را در شکل ۳-۳ با این روش نشان داده‌ایم. آرایه D در شکل ۳-۳ شامل طول کوتاهترین مسیرها در گراف می‌باشد. برای مثال D[۳][۵] برابر ۷ است چون طول کوتاهترین مسیر از v_3 به v_5 برابر ۷ می‌باشد. اگر ما بتوانیم

شکل ۳-۳ W معرف گراف شکل ۲-۳ و D شامل طول کوتاهترین مسیرها است. الگوریتم ما برای مسئله کوتاهترین مسیرها، مقادیر D را از طریق W محاسبه می‌کند.

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

مقادیر D را به شکلی که در W آمده محاسبه کنیم، دراینصورت الگوریتمی برای مسئله کوتاهترین مسیرها خواهیم داشت. این کار را می‌توان با تولید دنباله‌ای از n+۱ آرایه $D^{(k)}$ که در آن $0 < k < n$ و $D^{(k)}[i][j]$، طول کوتاهترین مسیر از v_i به v_j فقط با استفاده از گره‌های میانی $\{v_1, v_2, ..., v_k\}$ است، انجام داد. قبل از اینکه نشان دهیم «چرا از طریق W، D را محاسبه می‌کنیم؟»، عناصر موجود در این آرایه‌ها را شرح می‌دهیم.

مثال ۲-۳ می‌خواهیم برخی از مقادیر $D^{(k)}[i][j]$ را بطور نمونه برای گراف شکل ۲-۳ محاسبه کنیم.

$D^{(0)}[2][5] = length[V_2, V_5] = \infty$

$D^{(1)}[2][5] = minimum(length[V_2, V_5], lenth[V_2, V_1, V_5])$

$D^{(2)}[2][5] = D^{(1)}[2][5] = 14$ {این مقادیر برای هر گرافی یکسان هستند زیرا
{کوتاهترین مسیری که از v_2 شروع شود
{نمی‌تواند از v_2 بگذرد.}

$D^{(3)}[2][5] = D^{(2)}[2][5] = 14$ {برای این گراف،این مقادیر مساویند زیرا
{با v_2 هیچ مسیر جدیدی از v_2 به v_5 تولید نمی‌شود.}

$D^{(4)}[2][5] = minimum(length[V_2, V_1, V_5], legnth[V_2, V_4, V_5],$
$length[V_2, V_1, V_4, V_5], legnth[V_2, V_3, V_4, V_5])$

$= minimum(14, 5, 13, 10) = 5$

$D^{(5)}[2][5] = D^{(4)}[2][5] = 5$ {برای هر گرافی، این مقادیر مساویند زیرا کوتاهترین
{مسیر منتهی به v_5 نمی‌تواند از عبور کند}

آخرین مقدار محاسبه شده ($D^{(5)}[2][5]$)، طول کوتاهترین مسیر از v_2 به v_5 است که مجاز به عبور از هر گرهٔ دیگر می‌باشد. بدین معنا که آن، طول کوتاهترین مسیر است.

از آنجائیکه $D^{(n)}[i][j]$ طول کوتاهترین مسیر از v_i به v_j است که از تمامی گره‌های دیگر می‌گذرد، طول کوتاهترین مسیر از v_i به v_j نیز به شمار می‌رود. از آنجائیکه طول $D^{(n)}[i][j]$ کوتاهترین مسیری است که از تمامی گره‌های دیگر نمی‌گذرد، لذا مقدار آن، وزن لبه v_i به v_j خواهد بود. گفتیم که

$$D^{(0)} = W \quad , \quad D^{(n)} = D$$

بنابراین، برای تعیین D از W کافی است یک راه برای بدست آوردن $D^{(n)}$ از $D^{(0)}$ ارائه نمائیم. مراحل انجام این کار با استفاده از برنامه‌نویسی پویا به شرح ذیل می‌باشد:

۱- *ارائه* یک خاصیت (فرآیند) بازگشتی که از طریق آن بتوان $D^{(k)}$ را از $D^{(k-1)}$ بدست آورد.

۲- حل یک نمونه مسئله از طریق یک تابع پائین به بالا با تکرار فرآیند مرحله ۱ برای k از ۱ تا n. این عمل دنباله زیر را تولید می‌کند

$$\underset{W}{\uparrow} D^{(0)}, D^{(1)}, D^{(2)}, \ldots, \underset{D}{\uparrow} D^{(n)} \qquad (۳-۲)$$

مرحله ۱ را با درنظر گرفتن دو حالت زیر انجام می‌دهیم:

حالت ۱. حداقل کوتاهترین مسیر از v_i به v_j، فقط با استفاده از گره‌های مجموعه $\{v_1, v_2, \ldots, v_k\}$ به عنوان گره‌های میانی، از گره v_k استفاده نمی‌کند. دراینصورت

$$D^{(K)}[i][j] = D^{(k-1)}[i][j] \qquad (۳-۳)$$

یک مثال از این حالت در شکل ۲-۳ چنین است:

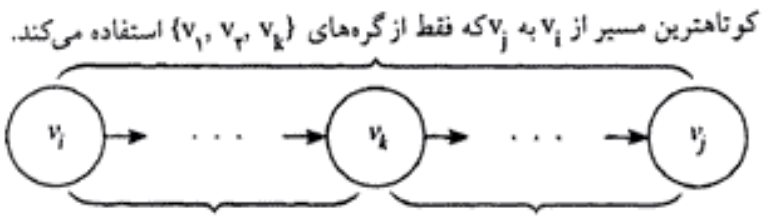
$$D^{(5)}[1][3] = D^{(4)}[1][3]$$

زیرا هنگامی که گرهٔ v_5 را نیز در نظر می‌گیریم، کوتاهترین مسیر از v_1 به v_3 همچنان $\{v_1, v_2, v_3\}$ می‌باشد.

حالت ۲. تمامی کوتاهترین مسیرها از v_i به v_j، فقط با استفاده از گره‌های مجموعه $\{v_1, v_2, v_k\}$ بعنوان گره‌های میانی، از v_k استفاده می‌کنند. در این حالت، هر یک از کوتاهترین مسیرها به صورتی که در شکل ۴-۳ آمده ، ظاهر می‌شوند. از آنجائیکه v_k نمی‌تواند یک گرهٔ میانی در زیر مسیر v_i تا v_k باشد، لذا زیرمسیر فقط گره‌های $\{V_1, V_2, \ldots, V_{k-1}\}$ را بعنوان گره‌های میانی بکار می‌گیرد و این اشاره دارد به اینکه طول زیرمسیر بایستی معادل $D^{(k-1)}[i][k]$ باشد زیرا اولاً، طول زیر مسیر نمی‌تواند کوتاهتر باشد چون $D^{(k-1)}[i][k]$ طول یکی از کوتاهترین مسیرها از v_1 به v_k است که تنها گره‌های $\{V_1, V_2, \ldots, V_{k-1}\}$ را به عنوان گره‌های میانی بکار گرفته است. ثانیاً طول زیر مسیر نمی‌تواند طولانی‌تر باشد؛ زیرا در اینصورت می‌توانستیم آن را با یک مسیر کوتاهتر در شکل ۴-۳ جایگزین نمائیم که این موضوع، با این حقیقت که تمام مسیرها در شکل ۴-۳ خود یکی از کوتاهترین مسیرها می‌باشند، تناقض دارد. بطور مشابه، طول زیرمسیر v_k به v_j در شکل ۴-۳ باید معادل $D^{(k-1)}[i][k]$ باشد. بنابراین، در حالت دوم

$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j] \qquad (۳-۴)$$

شکل ۳-۴ کوتاهترین مسیری که از v_k استفاده می‌کند.



یک مثال از حالت دوم در شکل ۳-۲ چنین است:

$$D^{(2)}[5][3] = 7 = 4 + 3 = D^{(1)}[5][2] + D^{(1)}[2][3]$$

چون باید هر یک از حالتهای اول یا دوم را داشته باشیم، مقدار $D^{(k)}[i][j]$ حداقل مقادیر موجود در سمت راست معادله‌های ۳-۳ و ۴-۳ می‌باشد به این معنی که می‌توان $D^{(k)}$ را توسط $D^{(k-1)}$ به صورت زیر محاسبه نمود:

$$D^{(k)}[i][j] = minimum\ (\ D^{(k-1)}[i][j]\ ,\ D^{(k-1)}[i][k] + D^{(k-1)}[k][j]\ )$$

حالت ۱ — حالت ۲

مرحله اول را با استفاده از الگوریتم برنامه‌نویسی پویا انجام دادیم. برای انجام مرحله دوم، از فرآیند بازگشتی ارائه شده در مرحله ۱ برای تولید دنباله‌ای از آرایهٔ نشان داده شده در عبارت ۲-۳ استفاده می‌کنیم. با یک مثال نشان می‌دهیم که چگونه هر یک از این آرایه‌ها توسط آرایه قبلی‌اش محاسبه می‌شود.

مثال ۳-۳ گراف داده شده در شکل ۲-۳ که به صورت ماتریس مجاور W در شکل ۳-۳ ارائه شده را در نظر بگیرید. برخی محاسبات نمونه برای این گراف در زیر آمده است (می‌دانیم که $D^{(0)} = W$)

$$D^{(1)}[2][4] = minimum\,(D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4])$$
$$= minimum\,(2, 9 + 1) = 2$$

$$D^{(1)}[5][2] = minimum\,(D^{(0)}[5][2], D^{(0)}[5][1] + D^{(0)}[1][2])$$
$$= minimum\,(\infty, 3 + 1) = 4$$

$$D^{(1)}[5][4] = minimum\,(D^{(0)}[5][4], D^{(0)}[5][1] + D^{(0)}[1][4])$$
$$= minimum\,(\infty, 3 + 1) = 4$$

پس از اینکه $D^{(2)}$ به طور کامل محاسبه شد، این توالی ادامه می‌یابد تا اینکه $D^{(5)}$ محاسبه گردد. آرایهٔ نهایی (D) شامل طولهای کوتاهترین مسیرها است که آن را در سمت راست شکل ۳-۳ نشان داده‌ایم.

در ادامه، الگوریتم ارائه شده توسط فلوید (۱۹۶۲) موسوم به الگوریتم فلوید رابیان نموده و توضیح می‌دهیم که چرا آن، علاوه بر آرایهٔ ورودی W از یک آرایهٔ D نیز استفاده می‌کند.

الگوریتم ۳-۳ الگوریتم فلوید برای کوتاهترین مسیرها

مسئله: کوتاهترین مسیرها از هر گره در یک گراف وزن‌دار به گره‌های دیگر را محاسبه نمائید. وزنها،اعدادی غیرمنفی هستند .

ورودی: یک گراف وزن‌دار وجهت‌دار، و n تعداد گره‌های گراف .گراف به صورت یک آرایهٔ دوبعدی W ارائه می‌شود که سطرها و ستونهای آن از ۱ تا n شاخص‌دهی شده‌اند. در این آرایه، W[i][j] وزن یک لبه ازگرهٔ iام به گرهٔ jام است .

خروجی: یک آرایهٔ دوبعدی D که سطرها و ستونهای آن از ۱ تا n شاخص‌دهی شده است .D[i][j]، طول یک کوتاهترین مسیر از گرهٔ iام به گرهٔ jام می‌باشد .

```
void floyd (int n
            const number W[ ][ ],
            number D[ ][ ])
{
  index i, j, k;
  D = W;
  for (k = 1; k <= n; k++)
    for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
        D[i][j] = minimum(D[i][j], D[i][k]+D[k][j]);
}
```

ما می‌توانیم محاسبات خود را تنها با بکاربردن آرایهٔ D انجام دهیم زیرا مقادیر سطر kام و ستون kام در طی kامین تکرار حلقه تغییری نمی‌یابد. یعنی در kامین تکرار، انتساب‌های زیر صورت می‌گیرد:

$$D[i][k] = minimum(D[i][k], D[i][k] + D[k][k])$$

که معادل D[i][k] می‌باشد و

$$D[k][j] = minimum(D[k][j], D[k][k] + D[k][j])$$

که معادل D[k][j] می‌باشد.

D[i][j] در طول تکرار kام، با مقدار خودش و مقادیر موجود در سطر و ستون kام محاسبه می‌شود. چون این مقادیر از ۱-kامین تکرار حلقه باقی مانده‌اند، لذا اینها همان مقادیری هستند که ما می‌خواهیم. گاهی اوقات بعد از ارائه یک الگوریتم برنامه‌نویسی پویا، این امکان وجود دارد که با بازبینی الگوریتم، آن را از لحاظ فضای اشغال شده کاراتر سازیم. حال می‌خواهیم الگوریتم فلوید را تحلیل کنیم.

تحلیل پیچیدگی زمانی حالت معمول الگوریتم ۳-۳ (الگوریتم فلوید برای کوتاهترین مسیرها)

عمل مبنایی: دستور العمل‌های داخل حلقهٔ for-j.

اندازه ورودی: n، تعداد گره‌های گراف.

ما یک حلقه در میان یک حلقه در میان حلقه‌ای دیگر با n گذر از هر حلقه داریم. بنابراین،

$$T(n) = n \times n \times n = n^3 \in \theta(n)$$

در زیر با اعمال تغییراتی در الگوریتم ۳-۳، کوتاهترین مسیرها را تولید می‌کنیم.

الگوریتم ۴-۳ الگوریتم فلوید برای کوتاهترین مسیرها ۲

مسئله: همانند الگوریتم ۳-۳؛ بجز اینکه کوتاهترین مسیرها نیز تولید می‌شوند.

خروجی اضافی: یک آرایهٔ P که سطرها و ستونهایش از ۱ تا n شاخص‌دهی شده، بطوری که p[i][j]، بالاترین شاخص یک گرهٔ میانی در کوتاهترین مسیر از v_i به v_j است اگر حداقل یک گرهٔ میانی وجود داشته باشد؛ صفر است اگر هیچ گرهٔ میانی وجود نداشته باشد.

```
viod floyd2 (int n ,
             const number W[ ][ ],
             number D[ ][ ],
             index P[ ][ ])
{
  index i, j, k;
  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
      p[i][j] = 0 ;
  D = W;
  for (k = 1; k <= n; k++)
   for (k = 1; i <= n; i++)
     for (j = 1; j <= n; j++)
       if (D[i][k]+D[k][j] < D[i][j]) {
         p[i][j] = k;
         D[i][j] = D[i][j] + D[k][j];
       }
}
```

شکل ۵-۳، آرایهٔ تولید شدهٔ P را نشان می‌دهد که این آرایه، هنگام اعمال الگوریتم به گراف شکل ۲-۳ ایجاد می‌گردد. الگوریتم زیر با استفاده از آرایهٔ P، کوتاهترین مسیر از گرهٔ v_1 به گرهٔ v_2 را تولید می‌کند.

الگوریتم ۵-۳ نمایش کوتاهترین مسیر

مسئله: نمایش گره‌های میانی در کوتاهترین مسیر از یک گره به گرهٔ دیگر در یک گراف وزن‌دار.

ورودی: آرایهٔ P که توسط الگوریتم ۴-۳ تولید شده و دو شاخص q و r از گره‌های گراف که به عنوان ورودی الگوریتم ۴-۳ می‌باشند و p[i][j]، بالاترین شاخص یک گرهٔ میانی در کوتاهترین مسیر از v_i به v_j است اگر حداقل یک گرهٔ میانی وجود داشته باشد؛ صفر است اگر هیچ گرهٔ میانی وجود نداشته باشد.

```
void path (index q, r)
{
  if (p[q][r] != 0){
    path(q, p[q][r]);
    cout << "v" << p[q][r];
    path(p[q][r], r);
  }
}
```

به خاطر دارید که طبق قرارداد فصل ۲، فقط از متغیرهائی که مقدارشان می‌توانند در فراخوانی‌های بازگشتی تغییر کنند، به عنوان ورودیهای رودالهای بازگشتی استفاده می‌کنیم. بنابراین، آرایهٔ P یک ورودی برای تابع path نیست. اگر الگوریتم با تعریف P بصورت سراسری بکار گرفته شود و ما در حال تعیین کوتاهترین مسیر از v_q به v_r باشیم، فراخوانی سطح بالای تابع path بصورت path(q,r); خواهد شد. با مقدار معین P در شکل ۵-۳، اگر مقادیر q و r بترتیب برابر ۵ و ۳ باشد، خروجی بصورت $[v_1 \quad v_4]$ خواهد بود. اینها گره‌های میانی روی کوتاهترین مسیر از v_5 هستند. در تمرینات نشان خواهیم داد که برای الگوریتم ۵-۳، $W(n) \in \theta(n)$ خواهد بود.

۳-۳ برنامه‌نویسی پویا و مسائل بهینه‌سازی

به خاطر آورید که الگوریتم ۴-۳ نه تنها طول کوتاهترین مسیرها را معین می‌کند، بلکه کوتاهترین مسیرها را نیز تولید می‌کند. تولید یک جواب بهینه، سومین مرحله در ارائه الگوریتم برنامه‌نویسی پویا برای مسئله بهینه‌سازی است. یعنی مراحل ارائه چنین الگوریتمی به فرم زیر می‌باشد:

۱ - *ارائه* یک خاصیت بازگشتی که جواب بهینهٔ یک نمونه مسئله را ارائه دهد.

۲ - محاسبهٔ مقدار یک جواب بهینه به روش پائین به بالا.

۳ - *تولید* یک جواب بهینه در یک روش پائین به بالا.

مراحل ۲ و ۳، تقریباً در یک نقطه از الگوریتم انجام می‌شوند. به دلیل اینکه الگوریتم ۲-۳ یک مسئلهٔ بهینه‌سازی نیست، لذا مرحله سومی در آن وجود ندارد.

شکل ۵-۳ آرایه P که هنگام اعمال الگوریتم ۴-۳ به گراف شکل ۲-۳ تولید شده است.

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

ممکن است اینطور بنظر برسد که هر مسئلهٔ بهینه‌سازی می‌تواند با استفاده از برنامه‌نویسی پویا حل شود؛ ولی این چنین نیست. ما باید قاعدهٔ بهینگی را در مسئله رعایت کنیم. این قاعده را در زیر آورده‌ایم.

تعریف **قاعدهٔ بهینگی** در مسائلی مطرح می‌شود که یک جواب بهینه برای یک نمونه مسئله همواره شامل جوابهای بهینه برای همه زیرنمونه‌های آن باشد.

بیان قاعدهٔ بهینگی برای مسئله بهینه‌سازی کار مشکلی است. با یک مثال بهتر می‌توانیم آن را درک کنیم در مسئله کوتاهترین مسیرها نشان دادیم که اگر v_k یک گره روی یک مسیر بهینه از v_i به v_j باشد، آنگاه زیرمسیرهای v_i به v_k، و v_k به v_j نیز باید بهینه باشند. بنابراین، جواب بهینه برای نمونه شامل جوابهای بهینه برای همه زیر نمونه‌ها است و در واقع، قاعده بهینگی اعمال می‌شود.

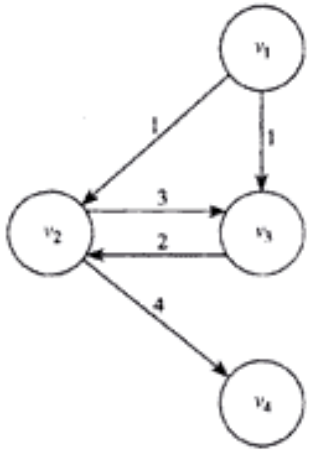
اگر قاعدهٔ بهینگی در یک مسئله بکار رود، می‌توانیم یک خاصیت بازگشتی ارائه نمائیم که یک جواب بهینه برای یک نمونه مسئله، جوابهای بهینه برای تمامی زیر نمونه‌هایش تولید نماید و این یک دلیل مهم و اساسی برای بکارگیری برنامه‌نویسی پویا است. بعنوان مثال، در مسئله کوتاهترین مسیرها، اگر زیرمسیرها خود کوتاهترین مسیر باشند، آنگاه مسیر ترکیب شده از این زیرمسیرها نیز کوتاهترین مسیر خواهد بود. لذا می‌توانیم با استفاده از خاصیت بازگشتی به روش پائین به بالا، جواب بهینه‌ای را برای نمونه‌های بزرگتر ارائه کنیم.

اگرچه قاعدهٔ بهینگی ممکن است واضح و مبرهن باشد ولی در عمل، قبل از اینکه فرض کنیم یک جواب بهینه را می‌توان با استفاده از برنامه‌نویسی پویا پیدا کرد، باید نشان دهیم که این قاعده برای آن صدق می‌کند. مثال زیر نشان می‌دهد که این قاعده در هر مسئله اعمال نمی‌شود

مثال ۳-۴ مسئلهٔ طولانی‌ترین مسیرها برای یافتن طولانی‌ترین مسیرهای ساده از هر گره به تمامی گره‌های دیگر را در نظر بگیرید. در شکل ۶-۳، مسیر سادهٔ بهینه (طولانی‌ترین مسیر) از v_1 به v_4، $[v_1, v_2, v_3, v_4]$ است. باوجود این، زیرمسیر $[v_1, v_2]$ یک مسیر بهینه از v_2 به v_3 نیست زیرا

$$\text{Length } [v_1, v_2] = 1 \qquad \text{Length } [v_1, v_2, v_3] = 2$$

بنابراین، قاعدهٔ بهینگی اعمال نمی‌شود زیرا مسیرهای بهینه از v_1 به v_2 و از v_2 به v_4 را نمی‌توان در کنار هم قرار داد تا یک مسیر بهینه از v_1 به v_4 بدست آید. انجام این کار، بجای آنکه یک مسیر بهینه ایجاد کند، باعث ایجاد یک حلقه می‌شود.



شکل ۶-۳ یک گراف وزن‌دار با یک چرخه.

ادامهٔ این فصل به مسائل بهینه‌سازی اختصاص دارد. هنگام ارائه الگوریتم‌ها، صراحتاً مراحلی که بیان شده است را ذکر نمی‌کنیم؛ اما آنها را انجام می‌دهیم.

۳-۴ ضرب ماتریس زنجیره‌ای

فرض کنید می‌خواهیم یک ماتریس ۲×۳ را در یک ماتریس ۳×۴ به صورت زیر ضرب کنیم:

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix} \times \begin{bmatrix}7 & 8 & 9 & 1\\ 2 & 3 & 4 & 5\\ 6 & 7 & 8 & 9\end{bmatrix} = \begin{bmatrix}29 & 35 & 41 & 38\\ 74 & 89 & 104 & 83\end{bmatrix}$$

ماتریس حاصل، یک ماتریس ۲×۴ خواهد بود. اگر از روش استاندارد ضرب ماتریسها استفاده کنیم، سه عمل ضرب مقدماتی برای هر عنصر در ضرب لازم است. مثلاً اولین عنصر در ستون اول با

$$\underbrace{۱\times ۷ + ۲\times ۲ + ۳\times ۶}_{\text{۳ ضرب}}$$

ضرب بدست می‌آید. از آنجائیکه تعداد ۸ = ۲×۴ ورودی در هر ضرب وجود دارد، تعداد کل ضربهای مقدماتی برابر ۲۴ = ۳ × ۴ × ۲ می‌باشد. بطور کلی، برای ضرب یک ماتریس $i \times j$ در یک ماتریس $j \times k$ به روش استاندارد، نیاز به $i \times j \times k$ ضرب مقدماتی داریم. ضرب چهار ماتریس زیر را در نظر بگیرید:

$$\underset{۲\times ۲}{A} \times \underset{۲\times ۳۰}{B} \times \underset{۳۰\times ۱۲}{C} \times \underset{۱۲\times ۸}{D}$$

ابعاد هر ماتریس در زیر آن آورده شده است. ضرب ماتریسی خاصیت شرکت‌پذیری دارد، یعنی ترتیب عمل

ضرب مهم نیست؛ مثلاً $A(B(CD))$ و $(AB)(CD)$ هر دو به یک جواب می‌رسند. پنج ترتیب متفاوت برای ضرب چهار ماتریس وجود دارد که هر یک از آنها ممکن است منجر به تعداد ضربهای مقدماتی متفاوتی شوند. در مثال قبل، تعداد ضربهای مقدماتی زیر را برای هر ترتیب از آن داریم:

$A(B(CD))$	$۳۰\times۱۲\times۸ + ۲\times۳۰\times۸ + ۲۰\times۲\times۸ = ۳۶۸۰$
$(AB)(CD)$	$۲۰\times۲\times۳۰ + ۳۰\times۱۲\times۸ + ۲۰\times۳۰\times۸ = ۸۸۸۰$
$A((BC)D)$	$۲\times۳۰\times۱۲ + ۲\times۱۲\times۸ + ۲۰\times۲\times۸ = ۱۲۳۲$
$((AB)C)D$	$۲۰\times۲\times۳۰ + ۲۰\times۳۰\times۱۲ + ۲۰\times۱۲\times۸ = ۱۰۳۲۰$
$(A(BC))D$	$۲\times۳۰\times۱۲ + ۲۰\times۲\times۱۲ + ۲۰\times۱۲\times۸ = ۳۱۲۰$

سومین ترتیب، ترتیب بهینه برای ضرب چهار ماتریس فوق است.

هدف ما ارائه الگوریتمی است که ترتیب بهینه‌ای را برای ضرب n ماتریس تعیین کند. ترتیب بهینه، فقط به ابعاد ماتریسها بستگی دارد. لذا علاوه بر n، این ابعاد تنها ورودی الگوریتم می‌باشند. الگوریتم brute-force برای درنظر گرفتن تمامی ترتیبهای ممکن و یافتن حداقل آنها بکار می‌رود. نشان خواهیم داد که این الگوریتم حداقل بصورت زمان-نمایی است. فرض کنید t_n تعداد ترتیبهای مختلفی باشد که می‌توان n ماتریس $A_۱, A_۲, ..., A_n$ را در هم ضرب نمود. زیرمجموعه‌ای از تمامی ترتیبها، مجموعه‌ای از ترتیبها است که $A_۱$، آخرین ماتریس ضرب شدهٔ آن است. همانطوری که در زیر آورده‌ایم، تعداد ترتیبهای مختلف این زیرمجموعه برابر $t_{n-۱}$ است زیرا آن، تعداد ترتیبهای مختلفی است که می‌توانیم $A_۲$ تا A_n را با آن ضرب کنیم.

$$A_۱(A_۲A_۳...A_n)$$
$$t_{n-۱} \text{ ترتیب مختلف}$$

زیرمجموعهٔ دوم از تمامی ترتیبها، مجموعه‌ای از ترتیبها است که A_n، آخرین ماتریس ضرب شدهٔ آن است. واضح است که تعداد ترتیبهای مختلف این زیرمجموعه نیز برابر $t_{n-۱}$ است. لذا

$$t_n \geq t_{n-۱} + t_{n-۱} = ۲t_{n-۱}$$

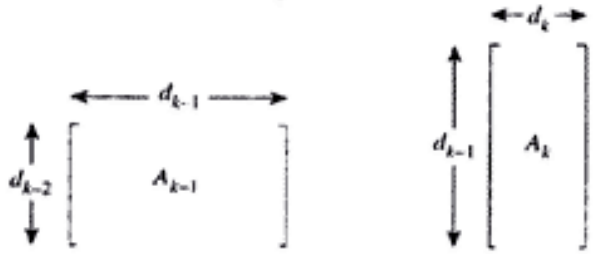
از آنجائیکه تنها یک راه برای ضرب دو ماتریس وجود دارد، لذا $t_۲ = ۱$ می‌باشد. با استفاده از روشی که در ضمیمه B نشان خواهیم داد، می‌توان این بازگشت را حل نمود و نشان داد که $t_n \geq ۳^{n-۲}$ است.

فهم اینکه قاعدهٔ بهینگی در این مسئله بکار رفته است، چندان مشکل نیست. یعنی ترتیب بهینه برای ضرب n ماتریس، شامل ترتیب بهینه برای ضرب هر زیرمجموعه‌ای از n ماتریس است. به عنوان مثال، اگر ترتیب بهینه برای ضرب شش ماتریس به صورت زیر باشد:

$$A_۱((((A_۲A_۳)A_۴)A_۵)A_۶)$$

در اینصورت، $(A_۲A_۳)A_۴$ بایستی ترتیب بهینه برای ضرب ماتریسهای $A_۲$ تا $A_۴$ باشد؛ یعنی اینکه ما می‌توانیم از برنامه‌نویسی پویا برای تولید یک جواب استفاده کنیم.

شکل ۷-۳ تعداد ستونهای A_{k-1} برابر با تعداد سطرهای A_k است.



از آنجائیکه ما در حال ضرب k-۱امین ماتریس A_{k-1} در kامین ماتریس A_k هستیم، لذا تعداد ستونهای A_{k-1} بایستی مساوی تعداد سطرهای A_k باشد. مثلاً در ضربی که قبلاً آورده شد، اولین ماتریس دارای سه ستون و دومین ماتریس دارای سه سطر است. اگر d_0 را به عنوان تعداد سطرهای A_1 و dk را به عنوان تعداد ستونهای A_k در نظر بگیریم بطوری که $1 \le k \le n$، آنگاه ابعاد A_k عبارت است از $d_{k-1} \times d_k$ این موضوع در شکل ۷-۳ نشان داده شده است.

همانند بخش قبل، از توالی آرایه‌ها برای تولید یک جواب استفاده می‌کنیم. فرض کنید برای $1 \le i \le j \le n$ داریم:

M[i][j] = حداقل تعداد ضربهای مورد نیاز برای ضرب A_i تا A_j (برای i<j)

M[i][j] = ۰

قبل از آنکه به چگونگی استفاده از این آرایه‌ها بپردازیم، مفهوم عناصر موجود در آنها را تشرح می‌کنیم.

مثال ۵-۳ فرض کنید شش ماتریس زیر را داریم:

A_1	×	A_2	×	A_3	×	A_4	×	A_5	×	A_6
۵×۲		۲×۳		۳×۴		۴×۶		۶×۷		۷×۸
d_0 d_1		d_1 d_2		d_2 d_3		d_3 d_4		d_4 d_5		d_5 d_6

برای ضرب A_4, A_5, A_6، دو ترتیب و تعداد ضربهای مقدماتی زیر را داریم:

$$(A_4 A_5)A_6 \text{ تعداد ضربهای } = d_3 \times d_4 \times d_5 + d_3 \times d_5 \times d_6$$
$$= ۴ \times ۶ \times ۷ + ۴ \times ۷ \times ۸ = ۳۹۲$$
$$A_4 (A_5 A_6) \text{ تعداد ضربهای } = d_3 \times d_5 \times d_6 + d_3 \times d_4 \times d_6$$
$$= ۶ \times ۷ \times ۸ + ۴ \times ۶ \times ۸ = ۵۲۸$$

بنابراین

$$M[۴][۶] = minimum(۳۹۲, ۵۲۸) = ۳۹۲$$

ترتیب بهینه برای ضرب شش ماتریس باید دارای یکی از فاکتورگیریهای زیر باشد:

۱- $A_1(A_2A_3A_4A_5A_6)$

۲- $(A_1A_2)(A_3A_4A_5A_6)$

۳- $(A_1A_2A_3)(A_4A_5A_6)$

۴- $(A_1A_2A_3A_4)(A_5A_6)$

۵- $(A_1A_2A_3A_4A_5)A_6$

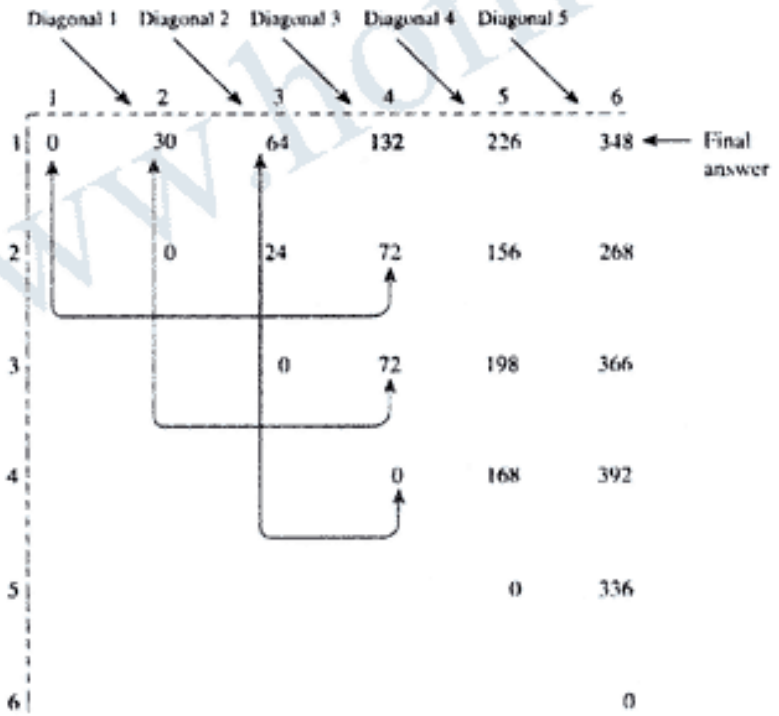
که درون هر پرانتز، حاصلضرب‌ها بر اساس ترتیب بهینه ماتریسها درون پرانتز بدست می‌آیند. از بین این فاکتورگیری‌ها، آن که دارای حداقل تعداد ضرب است باید بهینه باشد. تعداد ضربهای kامین فاکتورگیری برابر است با حداقل تعداد مورد نیاز برای بدست آوردن هر فاکتور بعلاوه تعداد مورد نیاز برای ضرب دو فاکتور، یعنی

$$M[1][k]+M[k+1][6]+d_0d_kd_6$$

به اینجا رسیده‌ایم که،

$$M[1][6]=\underset{1\le k\le 5}{minimum}(M[1][k]+M[k+1][6]+d_0d_kd_6)$$

در عبارت فوق، چیزی وجود ندارد که این محدودیت را بوجود آورد که اولین ماتریس A_1 باشد و یا آخرین ماتریس A_6 برای مثال می‌توانستیم با ضرب A_2 تا A_6 به نتیجه‌ای مشابه دست یابیم. بنابراین، می‌توان این



شکل ۸-۳ آرایه M، ارائه شده در مثال ۶-۳. M[۱][۴] که درون دایره قرار دارد، از زوج ورودی مشخص شده محاسبه می‌گردد.

نتیجه را تعمیم داد تا خاصیت بازگشتی زیر را برای ضرب n ماتریس بدست آورد. برای $1 \le i \le j \le n$،

$$M[i][j] = \underset{i \le k \le j-1}{minimum} (M[i][k] + M[k+1][j] + d_{i-1} d_k d_j) \quad \text{اگر } i < j,$$

$$M[i][i] = 0 \qquad (3-5)$$

یک الگوریتم تقسیم و غلبه براساس این خاصیت، زمان-نمایی است. ما یک الگوریتم کاراتر با استفاده از برنامه‌نویسی پویا ارائه می‌کنیم تا طی مراحلی مقادیر $M[i][j]$ را محاسبه کنیم. از شبکه‌ای همانند مثلث پاسکال استفاده می‌شود (به بخش ۱-۳ نگاه کنید.) محاسبات، که کمی پیچیده‌تر از محاسبات بخش ۱-۳ هستند، براساس خاصیت زیر از معادله ۵-۳ می‌باشند: $M[i][j]$ از تمام ورودیهای همان سطر $M[i][j]$ به طرف چپ و تمام ورودیهای همان ستون $M[i][j]$ به طرف پائین محاسبه می‌گردد. با استفاده از این خاصیت می‌توانیم ورودیهای n را اینطور محاسبه کنیم. سپس تمام ورودیهای قطر بالای آن را، که نامش را قطر ۱ می‌گذاریم، محاسبه می‌کنیم. سپس تمام ورودیهای قطر ۲ را محاسبه می‌کنیم و به همین ترتیب ادامه می‌دهیم. این شیوه را تا محاسبه تنها یک ورودی از قطر ۵ که جواب نهایی ما است (M[۱][۶]) ادامه می‌دهیم. این روال در شکل ۸-۳ برای ماتریسهای مثال ۵-۳ تشریح شده است. مثال زیر، محاسبات را نشان می‌دهد.

مثال ۶-۳ فرض کنید شش ماتریس مثال ۵-۳ را داریم. مراحل الگوریتم برنامه‌نویسی پویا به صورت زیر است. نتایج در شکل ۸-۳ نشان داده شده‌اند.

محاسبهٔ قطر ۰: برای $1 \le i \le 6$، $M[i][i] = 0$

محاسبهٔ قطر ۱:

$$M[1][2] = \underset{1 \le k \le 1}{minimum} (M[1][k] + M[k+1][2] + d_0 d_k d_2)$$

$$= M[1][1] + M[2][2] + d_0 d_1 d_2$$

$$= 0 + 0 + 5 \times 2 \times 3 = 30$$

مقادیر $M[2][3]$، $M[3][4]$، $M[4][5]$، $M[5][6]$ به همان ترتیب محاسبه می‌شوند. آنها را در شکل ۸-۳ نشان داده‌ایم.

محاسبهٔ قطر ۲:

$$M[1][3] = \underset{1 \le k \le 2}{minimum} (M[1][k] + M[k+1][3] + d_0 d_k d_3)$$

$$= minimum\ (M[1][1] + M[2][3] + d_0 d_1 d_3,$$

$$M[1][2] + M[3][3] + d_0 d_2 d_3)$$

$$= minimum(0 + 24 + 5 \times 2 \times 4, 30 + 0 + 5 \times 3 \times 4) = 64$$

مقادیر $M[4][6], M[3][5], M[2][4]$ را به همان ترتیب محاسبه‌می‌شوند. این مقادیر در شکل ۸-۳ نشان داده شده‌اند.

محاسبهٔ قطر ۳:

$$M[1][4] = \underset{1\le k\le 3}{minimum}\,(M[1][k]+M[k+1][4]+d_0 d_k d_4)$$
$$= minimum\,(M[1][1]+M[2][4]+d_0 d_1 d_4,$$
$$M[1][2]+M[3][4]+d_0 d_2 d_4,$$
$$M[1][3]+M[4][4]+d_0 d_3 d_4)$$
$$= minimum\,(0+72+5\times 2\times 6, 30+72+5\times 3\times 6,$$
$$64+0+5\times 4\times 6) = 132$$

مقادیر $M[2][5]$, $M[3][6]$ نیز به همین طریق محاسبه می‌شوند. که در شکل ۸-۳ نشان داده شد.

محاسبهٔ قطر ۴: ورودیهای قطر ۴ نیز به همین طریق محاسبه می‌شوند که در شکل ۸-۳ نشان داده شد.

محاسبهٔ قطر ۵: در نهایت، ورودی قطر ۵ نیز به همان صورت محاسبه می‌شود. این ورودی کوچکترین عدد حاصل از ضربهای مقدماتی(جواب نمونه) و مقدار آن برابر $M[1][6] = 348$ است.

الگوریتم زیر این روش را پیاده‌سازی می‌کند. تنها ورودیهای الگوریتم، ابعاد n ماتریس موسوم به مقادیر d_0 تا d_n می‌باشند. خود ماتریس به عنوان ورودی محسوب نمی‌شوند، زیرا مقادیر ماتریس‌ها مناسبتی با مسئله ندارند. آرایهٔ P حاصله از الگوریتم را می‌توان برای نمایش ترتیب بهینه بکار برد. این موضوع را بعد از تحلیل الگوریتم ۶-۳ بررسی خواهیم کرد.

الگوریتم ۶-۳ حداقل تعداد ضربها

مسئله: کمترین تعداد ضربهای مقدماتی را که برای ضرب n ماتریس ضروری است تعیین نموده، یک ترتیب که کمترین تعداد ضرب را تولید کند مشخص نمائید.

ورودی: تعداد ماتریس‌ها n، و یک آرایه از اعداد صحیح d، که از صفر تا n شاخص‌دهی شده و d[i] × d[i-۱] معرف ابعاد ماتریس iام می‌باشد.

خروجی: minmult، حداقل تعداد ضربهای مقدماتی برای ضرب n ماتریس، یک آرایهٔ دوبعدی P که می‌توان با استفاده از آن، ترتیب بهینهٔ ضرب را بدست آورد. سطرهای P از ۱ تا n-۱ و ستونهایش از ۱ تا n شاخص‌دهی شده است. P[i][j] نقطه‌ای است که ماتریسهای i تا j براساس یک ترتیب مطلوب برای ضرب ماتریسها از هم جدا می‌شوند.

```
int minmult (int n,
             const int d[ ],
             index P[ ][ ])
{
```

```
  index i, j, k, diagonal;
  int M[1..n][1..n];
  for (i = 1 ;i <= n ;i + +)
    M[i][j] = 0;
  for (diagonal = 1; i <= n-1; diagonal+ +)
    for (i = 1; i <= n-diagonal ; i++) {
      j = i + diagonal;
      M[i][j] = minimum(M[i][j] + M[k+1][j] + d[i-l]*d[k]*d[j]);
      P[i][j]= a value of k that gave the minimum;
    }
  return M[1][n];
}
```

تحلیل پیچیدگی زمانی حالت معمول الگوریتم ۶-۳ (حداقل تعداد ضربها)

عمل مبنایی: دستور العملهای اجرا شده برای هر مقدار k که شامل مقایسه‌ای برای بررسی کوچکترین مقدار نیز می‌باشد.

اندازه ورودی: n، تعداد ماتریسهایی که باید در هم ضرب شوند. سه حلقهٔ تودرتو داریم. از آنجائیکه y = i + diagonal است، لذا تعداد گذرها از حلقه k برای مقادیر معین diagonal و i برابر است با

$$j - ۱ - i - ۱ = i + diagonal - ۱ - i + ۱ = diagonal$$

برای یک مقدار معین diagonal، تعداد گذرهای انجام شده از حلقه **for-i** برابر است با n - diagonal و به دلیل اینکه diagonal (قطر) در محدوده ۱ تا n-۱ قرار دارد، لذا تعداد کل دفعات انجام عمل مبنایی برابر است با

$$\sum_{diagonal=۱}^{n-۱} [(n-diagonal)\times diagonal]$$

در تمرینات ثابت خواهیم کرد که این عبارت معادل است با

$$\frac{n(n-۱)(n+۱)}{۶} \in \Theta(n^۳)$$

در ادامه نشان می‌دهیم که چگونه از آرایهٔ P می‌توان یک ترتیب بهینه بدست آورد. مقادیر این آرایه، هنگامی که الگوریتم فوق به ابعاد مثال ۵-۳ اعمال می‌شود، در شکل ۹-۳ نشان داده شده است. به عنوان مثال، P[۲][۵] = ۴ بدین معناست که ترتیب بهینه برای ضرب ماتریسهای $A_۲$ تا $A_۵$ دارای فاکتورگیری $(A_۲ A_۳ A_۴) A_۵$ است که ماتریسهای درون پرانتز بر اساس یک ترتیب بهینه در هم ضرب می‌شوند. یعنی P[۲][۵] که مساوی ۴ است، نقطه‌ای است که ماتریس‌ها باید از هم جدا شوند تا فاکتورها بدست آیند. ما می‌توانیم با مشاهدهٔ P[۱][n]، یک ترتیب بهینه برای تعیین بالاترین سطح فاکتورگیری بدست آوریم. از آنجائیکه n=۶ و P[۱،۶] = ۱ است، لذا فاکتورگیری سطح بالای ترتیب بهینه بصورت

شکل ۹-۳ آرایه P تولید شده از الگوریتم ۶-۲ که به عنوان ابعاد در مثال ۵-۲ بکار می‌رود

	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

A_1 $(A_2\ A_3\ A_4\ A_5\ A_6)$ است. سپس با مشاهدهٔ P[۲][۶]، ترتیب بهینه جهت ضرب A_2 تا A_6 را تعیین می‌کنیم. چون مقدار P[۲][۶] مساوی ۵ است، فاکتورگیری آن برابر است با A_6 $(A_2\ A_3\ A_4\ A_5)$. اکنون می‌دانیم که فاکتورگیری ترتیب بهینه بصورت (A_6) A_1 $(A_2\ A_3\ A_4\ A_5)$ است که هنوز باید فاکتورگیری برای ضرب $A_2\ A_5$ تعیین شود. آنگاه به P[۲][۵] نگاه می‌کنیم و این روش را ادامه می‌دهیم تا اینکه تمام فاکتور گیریها تعیین شوند. جواب بدین صورت است:

$$A_1\ ((((A_2\ A_3)\ A_4)\ A_5)\ A_6)$$

الگوریتم زیر، روش فوق را پیاده‌سازی می‌کند.

الگوریتم ۷-۳ نمایش ترتیب بهینه

مسئله: ترتیب بهینه را برای ضرب n ماتریس نمایش دهید.

ورودی: عدد صحیح ثبت n، و آرایه P که حاصل الگوریتم ۶-۳ است P[i][j] نقطه‌ای است که ماتریس i و j طبق یک ترتیب بهینه برای ضرب ماتریسها، از هم جدا می‌شوند.

خروجی: یک ترتیب بهینه برای ضرب ماتریسها.

```
void order (index i, index j)
{
  if (i == j)
    cout << "A" << i;
  else{
    k = P[i][j];
    cout << "(";
    order(i, k)
    order(k+1, j);
    cout << ")";
  }
}
```

طبق قرارداد روالهای بازگشتی، n و P ورودیهای تابع order نیستند بلکه ورودیهای الگوریتم می‌باشند. اگر الگوریتم را با تعریف n و P به صورت سراسری پیاده‌سازی کنیم، فراخوانی سطح بالای تابع order به صورت ;order(۱, n) خواهد بود. هنگامی که ابعاد مثال ۵-۳ مورد نظر باشند، الگوریتم عبارت زیر را نمایش میدهد.

$$A_1((((A_2\ A_3)\ A_4)\ A_5)\ A_6)$$

دور کل عبارت زیر پرانتز وجود دارد زیرا الگوریتم به دور هر عنصر مرکب پرانتز قرار می‌دهد. در تمرینات نشان می‌دهیم که برای الگوریتم ۷-۳

$$T(n) \in \Theta(n)$$

الگوریتم $\Theta(n^3)$ برای ضرب ماتریس زنجیره‌ای در سال ۱۹۷۳ توسط گادبول مطرح شد. یائو در سال ۱۹۸۲ روشهایی را برای تسریع در برخی راه حلهای برنامه‌نویسی پویا ارائه داد. با استفاده از این روشها، تولید یک الگوریتم $\Theta(n^2)$ برای ضرب ماتریس زنجیره‌ای امکانپذیر شد. در سالهای ۱۹۸۲ و ۱۹۸۴ هیو و شینگ یک الگوریتم $\Theta(n \lg n)$ برای ضرب ماتریس زنجیره‌ای ارائه نمودند.

۵-۳ درختهای جستجوی دودویی بهینه

در ادامه، الگوریتمی را جهت تعیین روش مطلوب سازماندهی مجموعه‌ای از عناصر در یک درخت جستجوی دودویی ارائه می‌کنیم. قبل از بحث در مورد اینکه چه نوع سازماندهی مطلوب و بهینه در نظر گرفته می‌شود، مروری بر این درختها خواهیم داشت. برای هر گروه در یک درخت دودویی، زیردرختی که ریشه‌اش در سمت چپ گره واقع است به نام **زیردرخت چپ** گره خوانده می‌شود. زیردرخت چپ ریشه به زیردرخت چپ درخت موسوم است. **زیردرخت راست**نیز همانند فوق تعریف می‌شود.

تعریف یک درخت جستجوی دودویی، یک درخت دودویی از عناصر(کلید) است که از یک مجموعهٔ مرتب بدست می‌آید بطوری که

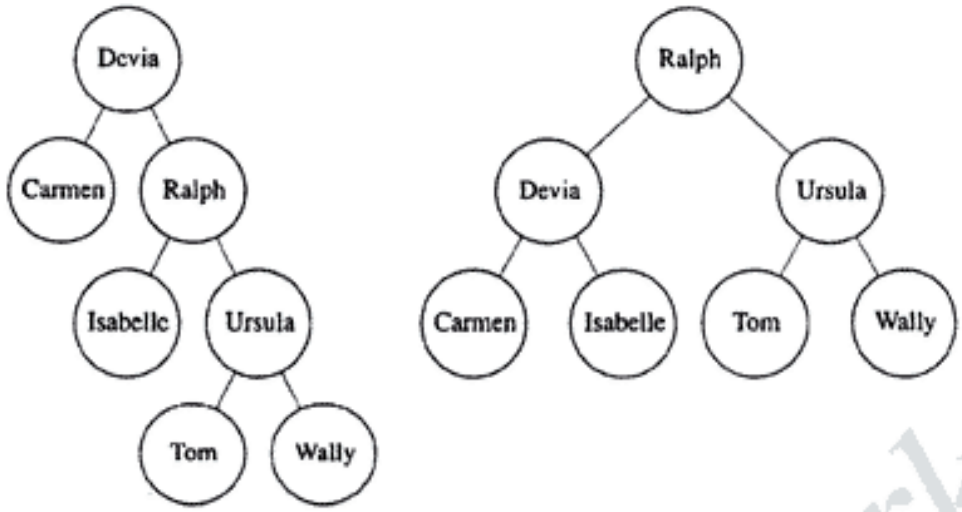
۱- هرگره حاوی یک کلیه است .

۲- کلیه‌های زیردرخت چپ یک گره معین،کوچکتریامساوی کلیه درآن گره هستند.

۳- کلیه‌های زیردرخت راست یک گره معین بزگتریامساوی کلیه درآن گره هستند.

شکل ۱۰- ۳، دو درخت جستجوی دودویی را نشان می‌دهد که هردو دارای کلیدهای مشابه می‌باشند در درخت چپ، به زیر درخت راست گره "ralph" توجه کنید. این زیر درخت شامل "Tom" و "ursula" "wally" است که همگی با توجه به ترتیب حروف الفبا بزرگتر از "ralph" می‌باشند. اگر چه در حالت کلی، یک کلید می‌تواند بیش از یکبار در درخت جستجوی دودویی بکار رود اما به منظور سهولت کار، فرض می‌کنیم که کلیدها، مجزا (منحصر بفرد) هستند.

شکل ۱۰-۳ دو درخت جستجوی دودویی.



عمق یک گره در یک درخت، تعداد لبه‌هایی است که در یک مسیر واحد از ریشه تا گره ادامه دارد. به آن، **سطح** گره در درخت نیز می‌گوئیم. معمولاً که یک گره یک عمق دارد یا یک گره د ریک سطح قرار دارد. برای مثال در درخت سمت چپ شکل ۱۰-۳ عمق گره شامل "ursula" برابر ۲ است. یا اینکه گره در سطح ۲ قرار دارد. ریشه، عمقی برابر صفر دارد یا بعبارتی دیگر، ریشه در سطح صفر قرار دارد. بیشترین عمق تمامی گره‌های درخت، عمق درخت نامیده می‌شود. درخت سمت چپ شکل ۱۰-۳، عمقی برابر ۳ و درخت سمت راست، عمقی برابر ۲ دارد. یک درخت دودویی زمانی متعادل نامیده می‌شود که تفاوت عمق دو زیر درخت زیر درخت چپ ریشه برابر صفر و عمق زیر درخت راست آن برابر ۲ است. درخت سمت راست شکل ۱۰-۳ یک درخت متعادل است. عموماً یک درخت دودویی دارای رکوردهایی است که بر اساس مقادیرکلیدها بازیابی می‌شوند. هدف، سازماندهی کلیدها در یک درخت جستجوی دودویی است بطوری که زمان متوسط یافتن یک کلید در درخت به حداقل ممکن برسد. (برای آشنایی با میانگین، به بخش ۲-۸-A مراجعه نمائید) درختی که بدین صورت سازماندهی می‌شود، بهینه است. اگر احتمال وقوع تمامی کلیدها به عنوان کلید جستجو یکسان باشد، به راحتی می‌توان دریافت که درخت سمت راست در شکل ۱۰-۳، یک درخت بهینه است. مابیشتر به حالتی می‌پردازیم که احتمال وقوع کلیدها بعنوان کلید جستجو، یکسان نیست. برای مثال، به منظور جستجو در درخت شکل ۱۰-۳ می‌توان یک نام از اسامی مردم ایالات متحده را بطور تصادفی انتخاب نمود. از آنجائیکه فراوانی نام "Tom" بیشتر از "ursula" است، لذا احتمال بیشتری را برای "Tom" قائل می‌شویم.

حالتی را در نظر می‌گیریم که می‌دانیم کلید مورد جستجو در درخت وجود دارد. برای به حداقل رساندن متوسط زمان جستجو باید پیچیدگی زمانی یافتن یک کلید را بدانیم. بنابراین، قبل از ادامه بحث الگوریتمی را نوشته و تحلیل می‌کنیم که یک کلید را در یک درخت دودویی جستجو کند. در نوشتن الگوریتم از نوع داده‌ای زیر استفاده می‌کنیم:

```
struct nodetype
{
  keytype key;
  nodetype* left;
  nodetype* righh;
};
typedef nodetype* node_pointer;
```

این تعریف بدین معناست که متغیر node_pointer به یک رکورد از نوع nodetype اشاره می‌کند. به عبارتی دیگر، مقدار آن، آدرس حافظه رکورد می‌باشد.

الگوریتم ۸-۳ درخت جستجوی دودویی

مسئله: گره‌ای که حاوی یک کلید در درخت جستجوی دودویی است را مشخص کنید (فرض می‌کنیم که کلید در درخت وجود دارد).

ورودی: یک اشاره‌گر tree به یک درخت جستجوی دودویی و یک کلید keyin.

خروجی: اشاره‌گر p که به گرهٔ شامل کلید مورد جستجو اشاره می‌کند.

```
void search (node_pointer tree,
             keytype keyin,
             node_ pointer& p)
{
  bool found;
  p = tree;
  found = false;
  while (!found)
    if (p -> key == keyin)
      found = true;
    else if (keyin < p -> key);
      p = p -> left;
    else
      p = p -> right;
}
```

به تعداد مقایسات انجام شده به وسیله روال search جهت یافتن یک کلید، **زمان جستجو** گفته می‌شود. هدف ما تعیین درختی است که متوسط زمان جستجو در آن، می‌بینیم باشد. همانطوریکه در بخش ۲-۱ گفته شد، فرض می‌کنیم که مقایسات با کارایی بالایی پیاده‌سازی می‌شوند. بنابراین، در الگوریتم قبلی تنها یک مقایسه در هر تکرار از حلقهٔ کلی while انجام می‌شود. لذا زمان جستجو برای یک کلید معین

برابر است با depth(key)+۱، که depth(key) عمق گره شامل کلید می‌باشد. برای مثال ، از آنجائیکه عمق گرهٔ شامل "Ursula" در سمت چپ شکل ۱۰-۳ برابر ۲ است، لذا زمان جستجو برابر است با

$$\text{depth(Ursula)} + 1 = 2 + 1 = 3$$

فرض می‌کنیم که key_1، key_2، ... و key_n، n کلید و p_i احتمال کلید k_i به عنوان کلید جستجو باشد. اگر c_i تعداد مقایسات مورد نیاز برای یافتن key_i در یک درخت مفروض باشد، آنگاه متوسط زمان جستجو برای آن درخت برابر است با $\sum_{i=1}^{n} c_i p_i$ و این مقداری است که می‌خواهیم آن را به حداقل برسانیم.

مثال ۷-۳ شکل ۱۱-۳ پنج درخت متفاوت را برای n=۳، نمایش می‌دهد. مقادیر واقعی کلیدها مهم نیستند؛ بلکه ترتیب آنها مورد نیاز است. اگر

p_1=۰/۷ p_2=۰/۲ p_3=۰/۱

باشد، آنگاه متوسط زمان جستجو برای درخت شکل ۱۱-۳ به صورت زیر است:

۱- ۳(۰/۷) + ۲(۰/۲) + ۱ (۰/۱) = ۲/۶

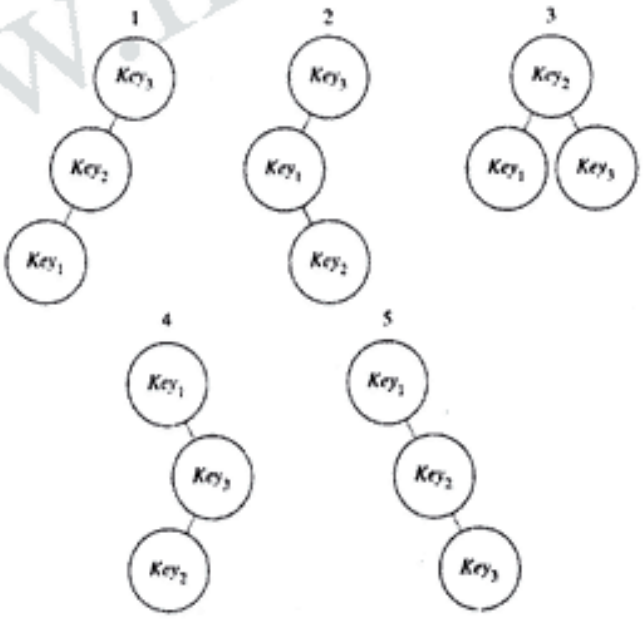
۲- ۲(۰/۷) + ۳(۰/۲) + ۱ (۰/۱) = ۲/۱

۳- ۲(۰/۷) + ۱(۰/۲) + ۲ (۰/۱) = ۱/۸

۴- ۱(۰/۷) + ۳(۰/۲) + ۲ (۰/۱) = ۱/۵

۵- ۱(۰/۷) + ۲(۰/۲) + ۳ (۰/۱) = ۱/۴

درخت پنجم، یک درخت بهینه است.



شکل ۱۱-۳ درختهای جستجوی ممکن، وقتیکه سه کلید وجود داشته باشد.

در حالت کلی، ما نمی‌توانیم با در نظر گرفتن همه درختهای جستجوی دودویی، یک درخت جستجوی دودویی بهینه پیدا کنیم زیرا تعداد چنین درختهایی حداقل به صورت نمایی از n است. ما این موضوع را اینگونه ثابت می‌کنیم که اگر دقیقاً همه درختهای جستجوی دودویی با عمق ۱ - n را در نظر بگیریم، آنگاه تعداد درختها بصورت نمایی خواهد بود. در یک درخت جستجوی دودویی با عمق ۱-n، موقعیت هر گره اطراف ریشه در هر یک از ۱-n سطح می‌تواند در سمت چپ یا راست گره پدرش باشد، یعنی اینکه در هر یک از سطوح مذکور، دو احتمال وجود دارد و این بدین معناست که تعداد درختهای جستجوی دودویی با عمق ۱ - n برابر است با 2^{n-1}.

برای ارائه الگوریتمی کاراتر می‌توان از برنامه‌نویسی پویا استفاده کرد. برای این منظور، فرض کنید کلیدهای key_i تا key_j در درختی مرتب شده‌اند که مقدار $\sum_{m=i}^{j} c_m p_m$ را به حداقل می‌رساند که در آن c_m تعداد مقایسات مورد نیاز برای یافتن key_m در درخت می‌باشد. چنین درختی را یک درخت بهینه برای کلیدهای مذکور می‌نامیم و مقدار بهینه را با A[i][j] نشان می‌دهیم. از آنجائیکه یک مقایسه برای یافتن یک کلید در یک درخت تک کلیدی صورت می‌گیرد، لذا $A[i][i] = p_i$

مثال ۸-۳ سه کلید و احتمالات مثال ۷-۳ مفروض است. یعنی

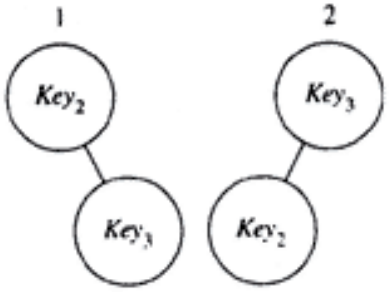
$p_1 = ۰/۷$ $p_2 = ۰/۲$ $p_3 = ۰/۱$

برای تعیین A [۲] [۳] بایستی دو درخت شکل ۱۲-۳ را در نظر بگیریم. برای این درخت داریم:

۱- $۱(p_2) + ۲(p_3) = ۱(۰/۲) + ۲(۰/۱) = ۰/۴$

۲- $۲(p_2) + ۱(p_3) = ۲(۰/۲) + ۱(۰/۱) = ۰/۵$

اولین درخت، بهینه است و A [۲] [۳] = ۰/۴



شکل ۱۲-۳ درختهای جستجوی دودویی متشکل از key۲ و key۳

توجه کنید که درخت بهینه‌ای که در مثال ۸-۳ بدست آمده است، زیردرخت راست ریشهٔ درخت بهینه در مثال ۷-۳ می‌باشد. حتی اگر این درخت دقیقاً مانند آن زیردرخت راست نبود، باز هم متوسط زمان جستجو در آن یکسان بود؛ چراکه درغیراینصورت ما می‌توانستیم آنرا با آن زیردرخت جایگزین نمائیم تا متوسط زمان جستجو در درخت حاصل کمتر شود. بطورکلی، هر زیردرخت از یک درخت بهینه باید برای تمام کلیدهای آن زیر درخت بهینه باشد. لذا، قاعدهٔ بهینگی بکار گرفته می‌شود.

فرض می‌کنیم درخت ۱، یک درخت بهینه برای حالتی که key_1 در ریشه درخت باشد، درخت ۲، یک درخت بهینه برای حالتی که key_2 در ریشه درخت باشد،... و درخت n، یک درخت بهینه برای حالتی که key_n در ریشه درخت باشد. برای هر ،زیر درخت‌های درخت k بایستی بهینه باشند که دراینصورت متوسط زمان جستجو در این زیر درختها همانند شکل ۱۳-۳ خواهد بود. این شکل همچنین نشان می‌دهد که برای هر $m \neq k$ برای یافتن key_m در درخت k، دقیقاً یک مقایسه بیشتر (یکی در ریشه) از زمانی که یافتن آن کلید در زیردرخت مورد نظر باشد انجام می‌گیرد. این مقایسه، مقدار $1 \times p_m$ را به متوسط زمان جستجو برای key_m در درخت k می‌افزاید. در نتیجه، متوسط زمان جستجو برای درخت k به صورت زیر می‌باشد.

$$\underbrace{A[1][k-1]}_{\text{Average time in left subtree}} + \underbrace{p_1 + \cdots + p_{k-1}}_{\text{Additional time comparing at root}} + \underbrace{p_k}_{\text{Average time searching for root}} + \underbrace{A[k+1][n]}_{\text{Average time in right subtree}} + \underbrace{p_{k+1} + \cdots + p_n}_{\text{Additional time comparing at root}}.$$

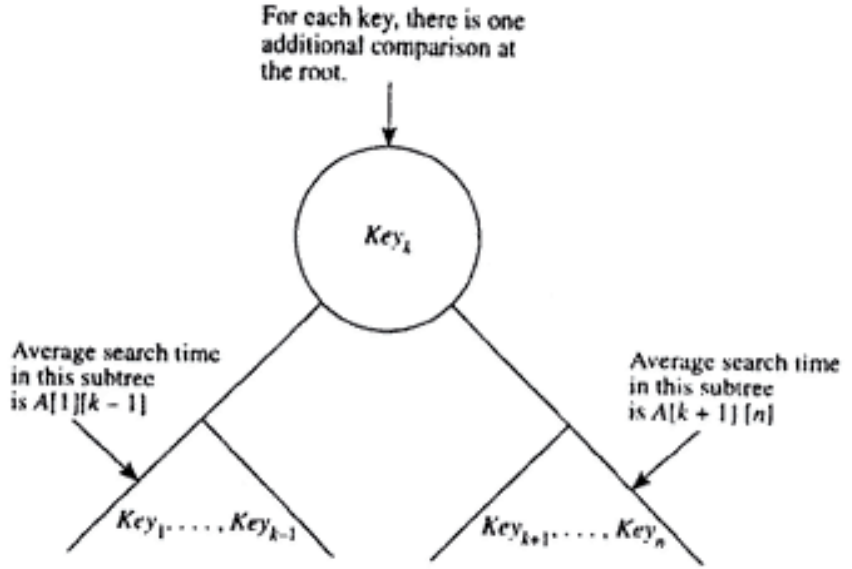
که معادل است با

$$A[i][k-1] + A[k+1][n] + \sum_{m=1}^{n} p_m$$

از آنجائیکه یکی از k درخت بایستی بهینه باشد، لذا متوسط زمان جستجو برای درخت بهینه را به صورت زیر بدست می‌آوریم:

$$A[1][n] = \text{minimum}(A[1][k-1] + A[k+1][n] + \sum_{m=1}^{n} p_m$$

که در آن A[۱][۰] و A[n+۱][n] برابر صفر می‌باشند. اگر چه مجموع احتمالات در این عبارت به وضوح برابر ۱ است، با وجود این، آن را به صورت مجموع نوشتیم زیرا در حال حاضر می‌خواهیم به آن عمومیت ببخشیم. مطلبی از بحث قبلی وجود ندارد که نشان دهد لزوماً کلیدها بایستی از key_1 تا key_n باشند. در حالت کلی، اینحالت میتوانند روی key_i تا key_j صورت گیرد که در آن i<j است بنابراین داریم:



$$A[i][j] = \underset{1 \le k \le n}{minimum} \; (A[i][k-1] + A[k+1][j] + \sum_{m=i}^{j} p_m \quad i<j$$
$$A[i][i] = p_i \qquad (۳-۶)$$
$$A[i][i-1] = ۰ \text{ و } A[j+1][j] = ۰$$

با استفاده از معادله ۶-۳ می‌توانیم الگوریتمی بنویسیم که یک درخت جستجوی دودویی بهینه را تعیین نماید. از آنجائیکه A[i][j] از ورودیهای سطر iام به طرف چپ A[i][j] و ورودیهای ستون jام به طرف پائین A[i][j] محاسبه می‌شوند، لذا ما با محاسبهٔ متوالی مقادیر روی قطر کار را ادامه می‌دهیم (همانند آنچه در الگوریتم ۶-۳ انجام دادیم.) بدلیل اینکه مراحل این الگوریتم خیلی شبیه به اقدامات انجام شده در الگوریتم ۶-۳ است، از بیان مثال برای نشان دادن این مراحل صرفنظر می‌کنیم، اما با ارائه یک الگوریتم ساده و به دنبال آن ذکر یک مثال، نتایج بکارگیری الگوریتم را نشان می‌دهیم. با اجرای الگوریتم، آرایهٔ R بدست می‌آید. این آرایه شامل شاخص کلیدهایی است که در هر مرحله برای ریشه انتخاب میشوند. برای مثال، A[۲][۴] شاخص کلید در ریشهٔ یک درخت بهینه است که حاوی دومین، سومین و چهارمین کلید است. بعد از تحلیل الگوریتم، دربارهٔ چگونگی ایجاد یک درخت بهینه بحث خواهیم کرد.

الگوریتم ۹-۳ درخت جستجوی دودویی بهینه

مسئله: یک درخت جستجوی دودویی بهینه برای مجموعه‌ای از کلیدها تعیین کنید که هر کدام از آنها، از احتمال معینی برای کلید جستجو بودن برخوردارند.

ورودی: n، تعداد کلیدها و یک آرایه از اعداد حقیقی به نام P که از ۱ تا n شاخص‌دهی شده است و P[i] که معرف احتمال جستجوی کلید iام می‌باشد.

خروجی: متغیر minavg که مقدار آن، متوسط زمان جستجو برای یک درخت جستجوی دودویی بهینه است، و یک آرایهٔ دوبعدی R که از آن می‌توان یک درخت بهینه ساخت. سطرهای R از ۱ تا n+۱ و ستونهایش از ۰ تا n شاخص‌دهی شده‌اند و R[i][j] معرف شاخص کلید درریشه یک درخت بهینه‌ای است که دارای کلیدهای iام تا jام می‌باشد.

```
viod optseardtree (int n,
                   const float p[ ],
                   float& minavg,
                   index R[ ][ ])
{
  index i, j, k, diogondal;
  float A[1..n + 1][0..n];
  for (i = 1; i <= n; i++){
    A[i][i-1] = 0;
    R[i][i] = i[i];
    A[i][i] = p[i];
```

```
    R[i][i-1] = 0;
  }
  A[n+1][n] = 0;
  R[n+1][n] = 0;
  for (diogonaol=1; diagonal <= n-1; diagonal++)
    for (i=1 ; i <= n - diagonal; i++){
      j = i + diagoanl;
      A[i][j] = minimum (A[i][k-1] + A[k+1][j] + Σ_{m=i}^{j} p_m)
      R[i][j] = a value of k that gave the minimum;
    }
    minavg = A[1][n];
}
```

تحلیل پیچیدگی زمانی حالت معمول الگوریتم ۹-۳(درخت جستجوی دودویی بهینه)

عمل مبنایی: دستورالعملهای اجراء شده برای هر مقدار از k،که شامل مقایسه برای ارزیابی کوچکترین عدد می‌باشد. برای محاسبهٔ مقدار $\sum_{m=i}^{j} p_m$ ، هربار نیازی به محاسبهٔ مجددنیست. در تمرینات یک روش کارا برای محاسبهٔ این مجموع پیداخواهیدکرد.

اندازه ورودی: n،تعداد کلیدها.

کنترل این الگوریتم تقریباً شبیه به الگوریتم ۶-۳ است بااین تفاوت که برای مقادیر معین diajonad و j، عمل مبنایی به تعداد diagonal+۱ مرتبه انجام می‌شود. تحلیلی شبیه تحلیل الگوریتم ۶-۳ بیان می‌کند که

$$T(n) = \frac{n(n-1)(n+4)}{6} \in \Theta(n^3)$$

الگوریتم زیر، یک درخت دودویی از آرایه R است. بخاطر دارید که R دارای شاخص کلیدهایی است که در هر مرحله به عنوان ریشه انتخاب می‌شوند .

الگوریتم ۱۰-۳ تشکیل درخت جستجوی دودویی بهینه

مسئله: یک درخت جستجوی دودویی بهینه بسازید.

ورودی: تعداد کلیدها n، آرایه‌ای از کلیدها شامل n کلید مرتب، آرایه R حاصل از الگوریتم ۹-۳. R[i][j] شاخص کلید در ریشه یک درخت بهینه است که شامل کلیدهای iام تا jام می‌باشد.

خروجی: یک اشاره‌گر tree، که به یک درخت جستجوی دودویی بهینهٔ حاوی n کلید اشاره می‌کند.

```
node_pointer tree (index i, j)
{
  index k;
  node_pointer p;
  k = R[i][j];
```

```
  if (k == 0)
    return NULL;
  else{
    p = new nodetype;
    P -> key = key[k];
    P -> left = tree (i, k-1);
    P -> right = tree (k+1, j);
    return p;
  }
}
```

دستور العمل p = new nodetype یک گرهٔ جدید را گرفته و آدرس آن را در P قرار می‌دهد. طبق قرارداد ما برای الگوریتم‌های بازگشتی، پارامترهای n ،Key، و R ورودیهای تابع tree نیستند. اگر الگوریتم با تعریف n ،Key، و R بصورت سراسری پیاده‌سازی شود، یک اشاره‌گر root به ریشهٔ یک درخت جستجوی دودویی بهینه، به صورت زیر از فراخوانی تابع tree بدست می‌آید:

root = tree(۱, n);

مثال ۹-۳ فرض کنید آرایهٔ Key دارای مقادیر زیر باشد:

Don	Isabelle	Ralph	Wally
Key[۱]	Key[۲]	Key[۳]	Key[۴]

و $p_1 = 3/8$ $p_2 = 3/8$ $p_3 = 1/8$ $p_4 = 1/8$

آرایه‌های A و R که از الگوریتم ۹-۳ بدست آمده‌اند، در شکل ۱۴-۳، و درخت حاصل از الگوریتم ۱۰-۳ در شکل ۱۵-۳ نشان داده شده‌اند. حداقل متوسط زمان جستجو برابر ۷/۴ می‌باشد.

توجه کنید که R[۱][۲] می‌تواند ۱ یا ۲ باشد زیرا هر یک از این شاخصها می‌توانند شاخص ریشه در درخت بهینه‌ای باشد که تنها شامل دو کلید اول است. لذا این دو شاخص می‌توانند کمترین مقدار A[۱][۲] در الگوریتم ۹-۳ را مشخص کنند، یعنی هر دو می‌توانند برای R[۱][۲] انتخاب شوند.

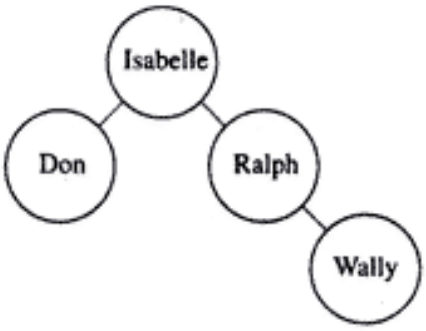
	0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

A

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

R

شکل ۱۴-۳ آرایه‌های A و R که حاصل از اعمال الگوریتم ۹-۳ بر روی مثال ۹-۳ می‌باشند.



شکل ۱۵-۳ درخت حاصل از اعمال الگوریتم‌های ۹-۳ و ۱۰-۳ بر نمونه مثال ۹-۳.

الگوریتم قبلی برای تعیین درخت جستجوی دودویی بهینه در سال ۱۹۵۹ توسط گیلبرت و مور مطرح شد. در سال ۱۹۸۲، یائو اظهار نمود که بااستفاده از روش تسریع برنامه‌نویسی پویا می‌توان یک الگوریتم را $\Theta(n^2)$ بدست آورد.

۶-۳ مسئله فروشندهٔ دوره‌گرد

فرض کنید فروشنده‌ای بخواهد برای فروش کالایش به ۲۰ شهر مسافرت کند. هر شهر بوسیلهٔ یک جاده به چند شهر دیگر متصل است. برای به حداقل رساندن زمان مسافرت ، می‌خواهیم کوتاهترین مسیری را پیدا کنیم که از شهر محل سکونت فروشنده شروع می‌شود و از هر شهر دیگر یکبار عبور می‌کند و مجدداً به شهر محل سکونت فروشنده باز می‌گردد. تعیین کوتاهترین مسیر در این مسئله را مسئله فروشنده دوره گرد می‌نامیم. نمونه‌ای از این مسئله را می‌توان توسط یک گراف وزن‌دار که هر گره آن بیانگر یک شهر است، نشان داد. همانگونه که در بخش ۲-۳ مطرح شد ، در گرافها وزن (فاصله) در یک جهت با وزن (فاصله) در جهت دیگر می‌تواند متفاوت باشد. مجدداً فرض می‌کنیم که فاصله‌ها اعدادی مثبت هستند شکل ۲-۳ و ۱۶-۳ چنین گرافهای وزن‌داری را نشان می‌دهند. یک تور (که به مدارها میتونی نیز موسوم است) در یک گراف جهت‌دار، مسیری است از یک گره به خودش بطوری که از گره‌های دیگر دقیقاً یکبار می‌گذرد. یک تور بهینه در یک گراف وزن‌دار و جهت‌دار، یک چنین مسیری با کمترین طول می‌باشد. مسئله فروشنده دوره گرد ، یافتن یک تور بهینه در یک گراف وزن‌دار و جهت‌دار است از زمان یکه حداقل یک تور وجود دارد. از آنجائیکه نقطه شروع، به تور بهینه ارتباطی ندارد، لذا v_1 را به عنوان گره شروع (گره ابتدایی) در نظر می‌گیریم. در زیر، سه تور و طولهای آنها را برای گراف شکل ۱۶-۳ آورده‌ایم:

$$\text{length}\ [v_1, v_2, v_3, v_4, v_5, v_1] = ۲۲$$
$$\text{length}\ [v_1, v_2, v_4, v_3, v_5, v_1] = ۲۶$$
$$\text{length}\ [v_1, v_2, v_4, v_5, v_3, v_1] = ۲۱$$

آخرین تور، بهینه است. با در نظر گرفتن تمام تورهای ممکن، این نمونه را به سادگی حل نمودیم.

در حالت کلی، از هر گره به هر گرهٔ دیگر یک لبه می‌تواند وجود داشته باشد. لذا اگر ما تمام تورهای ممکن را در نظر بگیریم، دومین گره روی تور می‌تواند هر یک از n-۱ گره دیگر باشد. سومین گره روی تور می‌تواند هر یک از n-۲ گره دیگر باشد،... و nامین گره می‌تواند تنها یک گره باشد. در اینصورت مجموع تعداد تورها برابر است با

$$(n-1)(n-2)\dots(1) = (n-1)!$$

که بدتر از حالت نمایی است. آیا می‌توان از برنامه‌نویسی پویا برای حل این مسئله استفاده نمود؟ توجه کنید که اگر v_k با v_1 روی یک تور بهینه باشد ، زیر مسیر آن تور از v_k به v_1 باید کوتاهترین مسیر از v_k به v_1 باشد که از هر یک از گره‌های دیگر دقیقاً یکبار می‌گذرد. یعنی قاعدهٔ بهینگی بکار گرفته می‌شود و می‌توانیم از برنامه‌نویسی پویا استفاده کنیم. برای این منظور، گراف را با یک ماتریس مجاور w، همانند آنچه در بخش ۲-۳ انجام داده‌ایم، بیان می‌کنیم. شکل ۱۷-۳ ماتریس مجاور ارائه شده برای گراف شکل ۱۶-۳ را نشان می‌دهد.

V = مجموعهٔ تمامی گره‌ها

A = یک زیرمجموعه از V

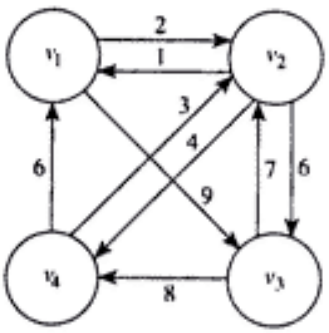
$D[v_i][A]$ = طول کوتاهترین مسیر از v_i به v_1 با تنها یکبار گره‌های مجموعه A

مثال ۱۰-۳ برای گراف شکل ۱۶-۳ داریم $V=\{v_1, v_2, v_3, v_4\}$. توجه کنید که $\{v_1, v_2, v_3, v_4\}$ معرف یک مجموعه و $[v_1, v_2, v_3, v_4]$ نمایانگر یک مسیر است. اگر $A=\{v_3\}$، آنگاه

$$D[v_2][A] = \text{length}[v_2, v_3, v_1] = 9$$

اگر $A=\{v_3, v_4\}$، آنگاه

$$D[v_2][A] = \text{minimum}(\text{length}[v_2, v_3, v_4, v_1], \text{length}[v_2, v_4, v_3, v_1]) =$$
$$= \text{minimum}(20, \infty) = 20$$



شکل ۱۶-۳ تور بهینه، $[v_1, v_2, v_4, v_3, v_1]$

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

شکل ۱۷-۳ ماتریس مجاور گراف شکل ۱۶-۳.

از آنجائیکه $V - \{v_1, v_j\}$ شامل تمامی گره‌ها بجز v_1 و v_j می‌باشد و قاعدهٔ بهینگی بکار رفته است، لذا

$$\text{طول یک تور بهینه} = \min_{2 \le j \le n} (w[1][j] + D[v_j][V - \{v_1 \text{ و } v_j\}])$$

و در حالت کلی، برای $i \neq 1$ که i در A وجود ندارد، داریم

$$D[v_i][A] = \min_{v_j \in A} (w[i][j] + D[v_j][A - \{v_j\}]) \quad \text{اگر } A \neq \phi$$

$$D[v_i][\phi] = w[i][j] \qquad (۳-۷)$$

ما می‌توانیم با استفاده از معادله ۳-۷، یک الگوریتم برنامه‌نویسی پویا برای مسئله فروشنده دوره گرد بنویسم. اما در ابتدا چگونگی عملکرد این الگوریتم را بررسی می‌کنیم.

الگوریتم ۳-۱۱ الگوریتم برنامه‌نویسی پویا برای مسئله فروشندهٔ دوره‌گرد

مسئله: یک تور بهینه برای یک گراف وزن‌دار و جهت‌دار مشخص نمائید. وزنها اعدادی غیرمنفی هستند.

ورودی: یک گراف وزن‌دار و جهت‌دار، و n، تعداد گره‌های گراف. گراف با یک آرایهٔ دوبعدی W مشخص می‌شود که سطرها و ستونهایش از ۱ تا n شاخص‌دهی شده‌اند ودر آن W[i][j] معرف وزن لبه از گره iام به گره jام است.

خروجی: یک متغیر minlength که مقدار آن طول تور بهینه است ، و یک آرایهٔ دوبعدی P که یک تور بهینه را از روی آن می‌توان ساخت. سطرهای P از ۱ تا n و ستونهای آن با تمامی زیرمجموعه‌های $V-\{v_1\}$ شاخص‌دهی شده‌اند. P[i][A] شاخص اولین گره بعد از vi بر روی کوتاهترین مسیر از v_i تا v_j است که از تمام گره های A دقیقاً یکبار می‌گذرد.

```
vide travel (int n,
             const number W[ ][ ],
             index P[ ][ ],
             number& minlength)
{
  index i, j ,k;
  number D[1..n][subset of V-{vi}];
  for (i = 2; i <= n; i++)
    D[i][ø] = w[i][1];
  for (k = 1; k <= n - 2; k++)
    for (all subsets A    V-{v1} containing k vertices
      for (i such that j ≠ 1 and vi is not in A){
        D[i][A] = minimum (W[i][j] + D[vj][A-{vj}]);
        p[i][A] = value of j that gave the minimum
    }
  D[1][V-{vi}] = minimum (W[1][j] + D[vj][V-{v1}];
  p[1][V-{v1}] = value of j that gave the minimum ;
  minlength = D[1][V-{v1}];
}
```

مجموعه‌های A و V و اعضای آنها v_i، در این الگوریتم به عنوان متغیر تعریف نشده‌اند زیرا در پیاده‌سازی الگوریتم تعریف نمی‌شوند. از طرفی در نوشتن الگوریتم، بدون توجه به آنها دچار مشکل می‌شویم. قبل از آنکه نشان دهیم چگونه یک تور بهینه از آرایهٔ P بدست می‌آید، به تحلیل الگوریتم می‌پردازیم. در ابتدا به یک قضیه نیاز داریم.

قضیه ۱-۳ برای هر $n>1$ داریم

$$\sum_{k=1}^{n} k \begin{bmatrix} n \\ k \end{bmatrix} = n\,2^{n-1}$$

اثبات: به عنوان تمرین نشان خواهید داد که

$$k \begin{bmatrix} n \\ k \end{bmatrix} = n \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$$

بنابراین

$$\begin{aligned} \sum_{k=1}^{n} k \begin{bmatrix} n \\ k \end{bmatrix} &= \sum_{k=1}^{n} \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} \\ &= n \sum_{k=0}^{n-1} \begin{bmatrix} n-1 \\ k \end{bmatrix} \\ &= n\,2^{n-1} \end{aligned}$$

تساوی اخیر با استفاده از نتیجهٔ مثال ۱۰-A از ضمیمهٔ A بدست آمده است.

تحلیل پیچیدگی زمانی و حافظه‌ای حالت معمول الگوریتم ۱۱-۳
(الگوریتم برنامه‌نویسی پویا برای مسئله فروشندهٔ دوره‌گرد)

عمل مبنایی: زمان در اولین و آخرین حلقه در مقایسه با زمان در حلقهٔ میانی بی‌اهمیت است، زیرا حلقهٔ میانی شامل سطوح متعددی می‌باشد. بنابراین، ما دستورالعملهای اجرا شده برای هر مقدار از V را بعنوان عمل مبنایی در نظر می‌گیریم که شامل یک دستورالعمل جمع نیز می‌باشد.

اندازه ورودی: n، تعداد گره‌های موجود در گراف.

برای هر مجموعهٔ A که شامل K گره است، باید k-۱-n گره را در نظر بگیریم. برای هر یک از این گره‌ها، k مرتبه عمل مبنایی انجام می‌شود. از آنجائیکه تعداد زیرمجموعه‌های A از (v1)-V که شامل k گره است برابر می‌باشد، لذا مجموع تعداد دفعات اجرای عمل مبنایی برابر است با

$$T(n) = \sum_{k=1}^{n-2} (n-1-k)\,k \begin{bmatrix} n-1 \\ k \end{bmatrix}$$

به راحتی می‌توان دریافت

$$(n-1-k) \begin{bmatrix} n-1 \\ k \end{bmatrix} = (n-1) \begin{bmatrix} n-2 \\ k \end{bmatrix}$$

می‌توانیم با جایگزینی این تساوی در معادله ۸-۳، معادله زیر را بدست آوریم:

$$T(n) = (n-1) \sum_{k=1}^{n-2} k \begin{bmatrix} n-2 \\ k \end{bmatrix}$$

درنهایت، با استفاده از قضیه ۱-۳ داریم :

$$T(n) = (n - ۱)(n - ۲)\,۳^{n-۳} \in \theta(n^{۲}۳^{n})$$

از آنجائیکه حافظهٔ مورد استفاده دراین الگوریتم زیاد است، ما پیچیدگی حافظه‌ای را که M(n) می‌نامیم مورد تحلیل قرار می‌دهیم. حافظهٔ مورد استفاده برای ذخیره آرایه‌های D[vi][A] و P[vi][A]، حجم قابل ملاحظه‌ای می‌باشد. بنابراین، تعیین می‌کنیم که این آرایه‌ها تا چه اندازه باید بزرگ باشند. بدلیل اینکه مجموع V-{vi} شامل n-۱ گره می‌باشد، می‌توانیم با استفاده از نتیجه مثال A-۱۰ در ضمیمهٔ A نشان دهیم که این مجموعه دارای ۲n-۱ زیرمجموعه است. اولین شاخص آرایه‌های D و P در محدوده ۱ تا n قرار دارد. بنابراین

$$M(n) = ۲ \times n\,۳^{n-۱} = n\,۳^{n-۱} \in \theta(n\,۳^{n})$$

ممکن است تعجب کنید که چطور الگوریتم جدید ما هنوز از کارایی $\Theta(n^{۲}۳^{n})$ برخوردار است. در مثال زیر نشان می‌دهیم که حتی یک الگوریتم با این پیچیدگی زمانی نیز می‌تواند گاهی اوقات مفید واقع شود.

مثال ۱۲-۳ رالف و نانسی هر دو برای دستیابی به یک موقعیت شغلی در بخش فروش، رقابت می‌کنند. روز جمعه رئیس به آنها می‌گوید که فروش از روز دوشنبه آغاز می‌شود و هر کس که بتواند تمام مسیر ۲۰ شهر منطقه را زودتر بپیماید، زودتر به این موقعیت نائل می‌شود. مسیر مزبور از محل کارشان شروع وپس از طی ۲۰ شهر به همانجا ختم می‌شود. هر شهر با جاده‌ای به هر شهر دیگر متصل است. رالف یک الگوریتم brute-fource در کامپیوترش اجرا می‌کند تا تمام !(۱-۲۰) حالت ممکن تور را پیداکند. اما نانسی در یافتن مسیرش از الگوریتم روش برنامه‌نویسی پویا استفاده می‌کند. با در نظر گرفتن مزایای این الگوریتم، او الگوریتم را بر روی کامپیوترش اجرا می‌کند. با فرض اینکه زمان پردازش دستورالعمل مبنایی در الگوریتم یک میکروثانیه است و همین زمان نیز برای پردازش عمل مبنایی الگوریتم رالف نیاز باشد، مدت زمان صرف شده در هر الگوریتم به طور تقریبی به قرار زیر است:

الگوریتم brute-force: ۳۸۷۵ = !۱۹ سال

الگوریتم برنامه‌نویسی پویا: ۴۵ = ۳ - ۲۰ (۲ - ۲۰)(۱ - ۲۰) ثانیه

ملاحظه می‌کنید که حتی یک الگوریتم $\Theta(n^{۲}۳^{n})$ نیز مفید است درحالیکه راه‌حل دیگر، یک الگوریتم زمان-فاکتوریل باشد. حافظهٔ مورد استفاده توسط الگوریتم برنامه‌نویسی پویا در این مثال به قرار زیر است:

خانه آرایه ۲۰×۲^{۲۰} = ۲۰۹۷۱۵۲۰

اگر چه این مقدار کاملاً بزرگ است ولی با توجه به استانداردهای امروزی قابل اجرا است.

استفاده از الگوریتم $\Theta(n^{۲}۳^{n})$ برای یافتن تور بهینه، زمانی قابل اجراست که n مقداری کوچک باشد، برای مثال، اگر ۶۰ شهر در مسئله وجود داشت، آنگاه اجرای الگوریتم سالها وقت می‌گرفت.

حال می‌خواهیم بدانیم که چگونه می‌توان از آرایه P، یک تور بهینه را بدست آورد. در اینجا نمی‌خواهیم الگوریتمی را ارائه دهیم؛ بلکه فقط طریقه بدست آوردن آن را نشان می‌دهیم. اعضاء آرایهٔ P که جهت تعیین یک تور بهینه برای گراف شکل ۱۶-۳ مورد نیاز هستند، عبارتند از:

۳	۴	۲
$P[4,\{v_2, v_3, v_4\}]$	$P[4,\{v_2, v_4\}]$	$P[4,\{v_2\}]$

تور بهینه را بدین صورت بدست می‌آوریم:

$$۳ = P[1][\{v_2, v_3, v_4\}] = \text{شاخص اولین گره}$$
$$۴ = P[1][\{v_2, v_4\}] = \text{شاخص دومین گره}$$
$$۲ = P[1][\{v_2\}] = \text{شاخص سومین گره}$$

بنابراین تور بهینه چنین است: $[v_1 v_2, v_3, v_4]$

تاکنون هیچکس نتوانسته برای مسئله فروشندهٔ دوره‌گرد الگوریتمی بنویسد که پیچیدگی زمانی بدترین حالت آن بهتر از حالت نمایی باشد والبته کسی هم ثابت نکرده که نوشتن چنین الگوریتمی ممکن نیست.

تمرینات

بخش ۱-۳

۱- با استفادا از استقراء n، نشان دهید که الگوریتم تقسیم و غلبه برای مسئله ضریب دوجمله‌ای (الگوریتم ۱-۳) بر اساس معادله ۱-۳، تعداد ۱ - $\binom{n}{k}$ ۲ عنصر را برای تعیین $\binom{n}{k}$ محاسبه می‌کند.

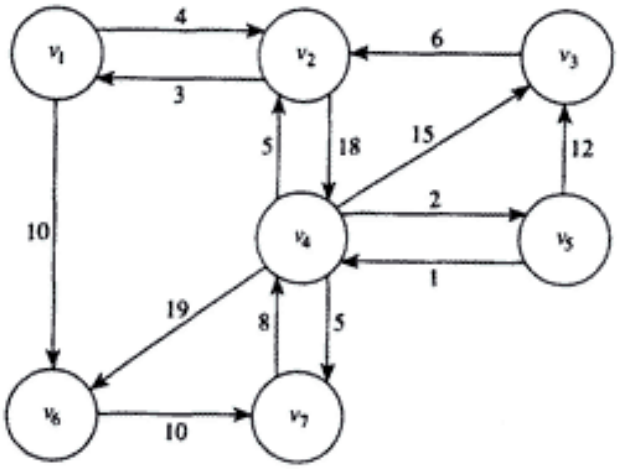
۲- الگوریتم‌های ارائه شده برای مسئله ضریب دوجمله‌ای (الگوریتم‌های ۱-۳ و ۲-۳) را روی کامپیوتر خود پیاده‌سازی نموده و کارایی آنها را با استفاده از نمونه مسئله‌های مختلف مطالعه و بررسی کنید.

۳- الگوریتم ۲-۳ (ضریب دوجمله‌ای با استفاده از برنامه‌نویسی پویا) را بگونه‌ای تغییر دهید که تنها از یک آرایهٔ تک‌بعدی با شاخصهای ۰ تا k استفاده نماید.

۴- معادله ۱-۳، که در این بخش ارائه شده است را ثابت کنید.

بخش ۲-۳

۵- الگوریتم فلوید۲ برای مسئله کوتاهترین مسیرها(الگوریتم ۲-۳)را بکار ببرید تا ماتریس D، شامل طول کوتاهترین مسیرها و ماتریس P، شامل بالاترین شاخصها برای گره‌های میانی روی کوتاهترین مسیرها است را برای گراف زیر ایجاد نماید. عملیات را مرحله به مرحله نشان دهید.



۶- با استفاده از الگوریتم نمایش کوتاهترین مسیر (الگوریتم ۵-۳)، کوتاهترین مسیر از گره v_4 به گره v_4 در گراف تمرین ۵ را با استفاده از ماتریس P، بدست آمده از آن، پیدا کنید. عملیات را به تفکیک مراحل نشان دهید.

۷- الگوریتم نمایش کوتاهترین مسیر (الگوریتم ۵-۳) را تحلیل نموده و نشان دهید که این الگوریتم دارای پیچیدگی زمانی خطی است.

۸- الگوریتم فلوید ۲ برای مسئله کوتاهترین مسیرها(الگوریتم ۴-۳) را برروی کامپیوتر خود پیاده‌سازی کنید و کارایی آن را با استفاده از گرافهای مختلف بررسی کنید.

۹- آیا می‌توانید الگوریتم فلوید۲ برای مسئله کوتاهترین مسیرها (الگوریتم ۴-۳)را بگونه‌ای تغییر دهید که دقیقاً کوتاهترین مسیر از یک گرهٔ معین به گرهٔ معین دیگر در گراف را مشخص کند؟ توضیح دهید.

۱۰- آیا می‌توان الگوریتم فلوید برای مسئله کوتاهترین مسیرها (الگوریتم ۴-۳) را طوری بکار گرفت که کوتاهترین مسیرها در یک گراف با چند وزن منفی را پیدا نماید؟ توضیح دهید.

بخشهای ۳-۳ و ۴-۳

۱۱- یک مسئله بهینه‌سازی بنویسید که قاعدهٔ بهینگی در آن بکار نرود و در نتیجه با استفاده از برنامه‌نویسی پویا نتوان جواب بهینه را بدست آورد.

۱۲- نشان دهید که یک الگوریتم تقسیم و غلبه براساس معادله ۵-۳، دارای پیچیدگی زمانی نمایی است.

۱۳- ترتیب بهینه و هزینه آن را برای ارزیابی حاصل ضرب $A_1 \times A_2 \times A_3 \times A_4 \times A_5$ تعیین کنید که در آن

A_1: ۱۰×۴

A_2: ۴×۵

A_3: ۵×۲۰

A_4: ۲۰×۲

A_5: ۲×۵۰

۱۴- الگوریتم حداقل تعداد ضربها (الگوریتم ۶-۳) و الگوریتم نمایش ترتیب بهینه (الگوریتم ۳.۷) را بر روی کامپیوتر خود پیاده‌سازی کرده، کارایی آنها را استفاده از نمونه مسئله‌های مختلف ارزیابی کنید.

۱۵- معادله زیر را ثابت کنید.

$$\sum_{diagonal=1}^{n-1}[(n-diagonal)\times diagonal]=\frac{n(n-1)(n+1)}{6}$$

۱۶- نشان دهید که برای پرانتزدار کردن کامل یک عبارت شامل n ماتریس، به n-۱ جفت پرانتز نیاز داریم.

۱۷- الگوریتم ۷-۳ را تحلیل نموده و نشان دهید که پیچیدگی زمانی آن، خطی است.

۱۸- یک الگوریتم کارا بنویسید که برای ضرب n ماتریس $A_1 \times A_2 \times ... \times A_n$ یک ترتیب بهینه ارائه نماید. ابعاد ماتریسها ۱×۱، d×۱، ۱×d یا d×d می‌باشد که در آن d یک عدد صحیح مثبت است. الگوریتم را تحلیل نموده و نتایج را با استفاده از نمادهای ترتیب نشان دهید.

بخشهای ۵-۳ و ۶-۳

۱۹- با استفاده از ۶ کلید مجزا، چند درخت جستجوی دودویی مختلف می‌توان ساخت؟

۲۰- یک درخت جستجوی دودویی بهینه برای عناصر زیر تشکیل دهید. احتمال وقوع هر یک از عبارات، داخل پرانتز نوشته شده است.

CASE(۰/۵) ،ELSE(۰/۱۵) ،END(۰/۰۵) ،IF(۰/۳۵) ،OF(۰/۰۵) ،THEN(۰/۳۵)

۲۱- یک روش کارا برای محاسبهٔ $\sum_{m=i}^{j} p_m$ که در الگوریتم درخت جستجوی دودویی (الگوریتم ۹-۳) بکار می‌رود، پیدا کنید.

۲۲- الگوریتم درخت جستجوی دودویی بهینه (الگوریتم ۹-۳) و الگوریتم تشکیل درخت جستجوی دودویی بهینه (الگویتم ۱۰-۳) را تحلیل نموده و پیچیدگی زمانی آن را با استفاده از نمادهای ترتیب نشان دهید.

۲۳- الگوریتم ۱۰-۳ را تحلیل نموده و پیچیدگی زمانی آن را با استفاد از نمادهای ترتیب نشان دهید.

۲۴- الگوریتم درخت جستجوی دودویی بهینه (الگوریتم ۹-۳) را به حالتی که کلید جستجو ممکن است در درخت نباشد تعمیم دهید، بدین صورت که q_i را احتمال اینکه کلید مورد جستجو بین key_i و key_{i+1} قرار گرفته است در نظر بگیرید. الگوریتم را تحلیل نموده و نتایج را بوسیله نمادهای ترتیب نشان دهید.

۲۵- نشان دهید که یک الگوریتم تقسیم و غلبه براساس معادله عدد ۶-۳، یک پیچیدگی زمانی نمایی دارد.

۲۶- بهترین مسیر برای گراف جهت‌دار و وزن‌دار ارائه شده توسط ماتریس W را پیدا کنید. عملیات را مرحله به مرحله نشان دهید.

$$W=\begin{bmatrix}0 & 8 & 13 & 18 & 20\\ 3 & 0 & 7 & 8 & 10\\ 4 & 11 & 0 & 10 & 7\\ 6 & 6 & 7 & 0 & 11\\ 10 & 6 & 2 & 1 & 0\end{bmatrix}$$

تمرینات اضافی

۲۷- اندازه ورودی در الگوریتم ۳.۲ (ضریب دوجمله‌ای با استفاده از برنامه‌نویسی پویا)، همانند الگوریتم محاسبهٔ عنصر nام فیبوناچی، برابر تعداد نمادهایی است که برای کددهی اعداد n و k بکار می‌روند. الگوریتم را از لحاظ اندازه ورودی تحلیل نمائید.

۲۸- تعداد ترتیبهای ممکن برای ضرب n ماتریس A_1، A_2،...، A_n را تعیین کنید.

۲۹- نشان دهید که تعداد درختهای جستجوی دودویی با n کلید از فرمول زیر بدست می‌آید:

$$\frac{1}{(n+1)}\begin{bmatrix} 2n \\ n \end{bmatrix}$$

۳۰- آیا می‌توانید یک الگوریتم زمان-مربعی برای مسئله درخت جستجوی دودویی بهینه (الگوریتم ۳.۹) بدست آورید؟

۳۱- با استفاده از روش برنامه‌نویسی پویا، الگوریتمی بنویسید که ماکزیمم مجموع در هر زیر لیست از یک لیست معین با n مقدار حقیقی را پیدا کند. الگوریتم را تحلیل نموده و نتایج را با استفاده از نمادهای ترتیب نشان دهید.

۳۲- دو رشته کاراکترهای S_1 و S_2 را در نظر می‌گیریم. با فرض اینکه با حذف هر تعدادی کاراکتر از هر قسمت یک رشته بتوان یک زیر رشته ساخت، الگوریتمی بنویسید که بااستفاده از روش برنامه‌نویسی پویا بلندترین زیررشتهٔ مشترک S_1 و S_2 را پیدا کند. این الگوریتم، ماکزیمم طول زیررشتهٔ مشترک از هر رشته را برمی‌گرداند.

فصل ۴

روش حریص (The Greedy Approach)

شاید اسکروج، شخصیت کلاسیک چارلزدیکنز، حریص‌ترین شخصی باشد که تا به حال دیده شده است. اسکروج هرگز گذشته و آینده را در نظر نمی‌گرفت و تنها هدفش این بود که هر روز، با طماعی خاص خودش مقداری طلا به چنگ آورد. یک الگوریتم حریص نیز به همان شیوه اسکروج عمل می‌کند، بدینصورت که عناصر داده‌ای را بطور متوالی گرفته واز بین آنها بدون توجه به انتخاب‌های قبلی یا بعدی، "بهترین" را براساس برخی معیارهای خاص انتخاب می‌کند. البته این عقیده که به علت برخی خصوصیات منفی اسکروج، مانند حرص، نباید از الگوریتم‌های حریص استفاده کرد، اشتباه است؛ چرا که این الگوریتم‌ها، راه حلهایی بسیار ساده و کارا می‌باشند.

الگوریتم‌های حریص همانند برنامه‌نویسی پویا، اغلب برای مسائل بهینه‌سازی بکار می‌روند، با این تفاوت که استفاده از روش حریص، بسیار راحت است. در برنامه نویسی پویا، از خاصیت بازگشتی جهت تقسیم یک نمونه به نمونه‌های کوچکتر استفاده می‌شود؛ در حالیکه در الگوریتم حریص، هیچ تقسیمی به نمونه‌های کوچکتر صورت نمی‌گیرد. یک **الگوریتم حریص (Greedy algorithm)** برای تولید جواب از دنباله‌ای عناصر انتخابی استفاده می‌کند که هر یک از آنها در آن لحظه، بهترین انتخاب

به نظر می‌رسند. یعنی هر انتخاب، به طور محلی بهینه است و انتظار می‌رود که بتوان یک جواب بهینهٔ نهایی بدست آورد. البته جواب نهایی همیشه بهترین جواب نیست. لذا برای یک الگوریتم معین بایستی تحقیق شود که آیا جواب نهایی همیشه بهینه است یا خیر؟

با یک مثال ساده به معرفی الگوریتم حریص می‌پردازیم. جو، کارمند بخش فروش یک مغازه، اغلب با مشکل خردکردن پول مشتریانش روبرو است؛ چرا که مشتریان معمولاً نمی‌خواهند مقدار زیادی سکه دریافت کنند. مثلاً اگر مشتری برای دریافت ۰/۸۷ دلار، ۸۷ سکه یک پنی دریافت کند، ممکن است عصبانی شود. بنابراین، هدف او این است که نه تنها پولهای باقیمانده مشتریان را به طور کامل پرداخت کند، بلکه تعداد سکه‌های داده شده را به حداقل ممکن برساند. یک روش حریصانه برای حل چنین مسئله‌ای به صورت زیر است: جو باید مجموعه‌ای از سکه‌ها را تحویل مشتری دهد. در ابتدا سکه‌ای در این مجموعه وجود ندارد. جو به دنبال بزرگترین سکه می‌گردد؛ با علم به اینکه معیار او برای انتخاب بهترین سکه (بهترین حالت محلی)، ارزش سکه است. این کار در یک الگوریتم حریص به **روال انتخاب** موسوم است. سپس بررسی می‌کند که اگر این سکه را به مجموعه پول خردها اضافه کند، مقدار کل سکه‌ها از میزان بدهی او به مشتری تجاوز می‌کند یا خیر؟ این عمل را **بررسی امکان سنجی** در الگوریتم حریص گویند. اگر افزودن سکه موجب شود که مجموعه پول خردها از میزان بدهی او به مشتری بیشتر نشود، آن سکه را اضافه می‌کند. سپس بررسی می‌کند که آیا ارزش کل سکه‌های مجموعه، برابر میزان بدهی اوست یا خیر؟ به این عمل، **بررسی جواب** در الگوریتم حریص گویند. اگر آنها برابر نبودند، سکهٔ دیگری را با استفاده از روال انتخاب می‌گیرد و روند فوق را تکرار می‌کند. این عمل تا آنجا تکرار می‌شود که ارزش سکه‌ها برابر میزان بدهی او به مشتری شود و یا اینکه سکه‌های صندوق تمام شده باشد که در حالت اخیر، وی قادر نیست پول مشتری را به طور کامل پرداخت کند. در زیر، یک الگوریتم سطح بالا برای این روال آورده‌ایم.

```
while (there are more coins and the instance is not solved){
    Grab the largest remaining coin;                          //روال انتخاب
    if (adding the coin makes the change exceed the amount owed)  //بررسی امکان سنجی
        reject the coin;
    else
        add the coin to the change;
    if (the total value of the change equals the amount owed)     //بررسی جواب
        the instance is solved;
}
```

شکل ۱-۴ یک الگوریتم حریص برای تحویل سکه.

Amount owed: 36 cents

Step	Total Change
1. Grab quarter	
2. Grab first dime	
3. Reject second dime	
4. Reject nickel	
5. Grab penny	

در بررسی امکان‌سنجی، یعنی وقتی که بررسی می‌کنیم که آیا افزودن سکه موجب خواهد شد که مجموعه پول خردها از مقدار پولی که باید به مشتری داده شود، بیشتر شود یا خیر، به این نکته پی می‌بریم که مجموعه‌ای که با افزودن این سکه بدست آمده، نمی‌تواند یک جواب کامل برای آن نمونه مسئله باشد. لذا آن مجموعه، غیرممکن و مردود است. یک مثال کاربردی از این الگوریتم را در شکل ۱-۴ نشان داده‌ایم. این الگوریتم، "حریص" نامیده می‌شود زیرا در روال انتخاب ، بزرگترین سکه بعدی حریصانه و بدون توجه به اشکالات احتمالی چنین انتخابی، برگزیده می‌شود. هیچ فرصتی برای تجدید نظر در یک انتخاب وجود ندارد. وقتی که یک سکه مورد قبول واقع شد، برای همیشه وارد مجموعه جواب می‌شود و وقتی که سکه‌ای رد شد، برای همیشه از مجموعه جواب خارج می‌شود. در عین حال که این روال بسیار ساده است، این سؤال مطرح می‌شود که آیا جواب نهایی که از ترکیب جوابهای محلی بدست می‌آید، بهینه است یا خیر؟ یعنی در مسئله پول خرد، در صورتی که جوابی وجود داشته باشد، آیا جواب ارائه شده توسط الگوریتم شامل کمترین تعداد سکه‌های لازم برای تحویل به مشتری است یا خیر؟ اگر سکه‌ها شامل سکه‌های امریکا (پنی، پنج سنتی، ده سنتی، بیست و پنج سنتی، نیم دلاری) و حداقل یک نوع از هر سکه در دسترس باشد، الگوریتم حریص همواره یک جواب بهینه (در صورت وجود) تولید می‌کند. این موضوع درتمرینات ثابت شده است. البته موارد دیگری نیز وجود دارند که جواب بهینه تولید می‌کنند و ما برخی از آنها را در تمرینات خواهیم دید. توجه کنید که اگر یک سکهٔ ۱۲ سنتی در مجموعهٔ سکه‌های امریکا وجود داشته باشد، الگوریتم حریص همیشه نمی‌تواند یک جواب بهینه تولید نماید. شکل ۲-۴، این موضوع را نشان می‌دهد. در این شکل، جواب نهایی شامل پنج سکه است در حالیکه جواب بهینه می‌تواند شامل ۳ سکه ده سنتی، پنج سنتی و پنی باشد. بنابراین، الگوریتم حریص نمی‌تواند همواره یک جواب بهینه را تضمین کند. در بخشهای ۱-۴، ۲-۴ و ۳-۴ مسائلی مطرح می‌شود که روش حریص می‌تواند همواره یک جواب بهینه برای آنها تولید نماید. در بخش ۴-۴، یک مسئله که مبین چنین حالتی نیست را مطرح می‌کنیم. همچنین در آن بخش، با مقایسه روش حریص با روش برنامه‌نویسی پویا، مشخص می‌کنیم که هر کدام در چه زمانی بکار می آیند.

در یک جمع‌بندی کلی می‌توان گفت که الگوریتم حریص، با یک مجموعهٔ خالی آغاز می‌شود و عناصر، پشت سر هم به این مجموعه اضافه می‌شوند؛ تا اینکه مجموعه، نمایانگر یک جواب برای نمونه مسئله باشد. هر تکرار، شامل اجزاء زیر است:

- یک **روال انتخاب**، عنصر بعدی را جهت افزودن به مجموعه انتخاب می‌کند. این انتخاب، براساس یک معیار حریصانه که به طور محلی بهترین جواب را در هر لحظه برمی‌گزیند، شکل می‌گیرد.
- یک **بررسی امکان‌سنجی**، تعیین می‌کند که آیا با تکمیل مجموعهٔ جدید، امکان دستیابی به جواب برای نمونه مسئله وجود دارد یا خیر.
- یک **بررسی جواب**، تعیین می‌کند که آیا مجموعه جدید، یک جواب برای نمونه مسئله می‌باشد یا خیر.

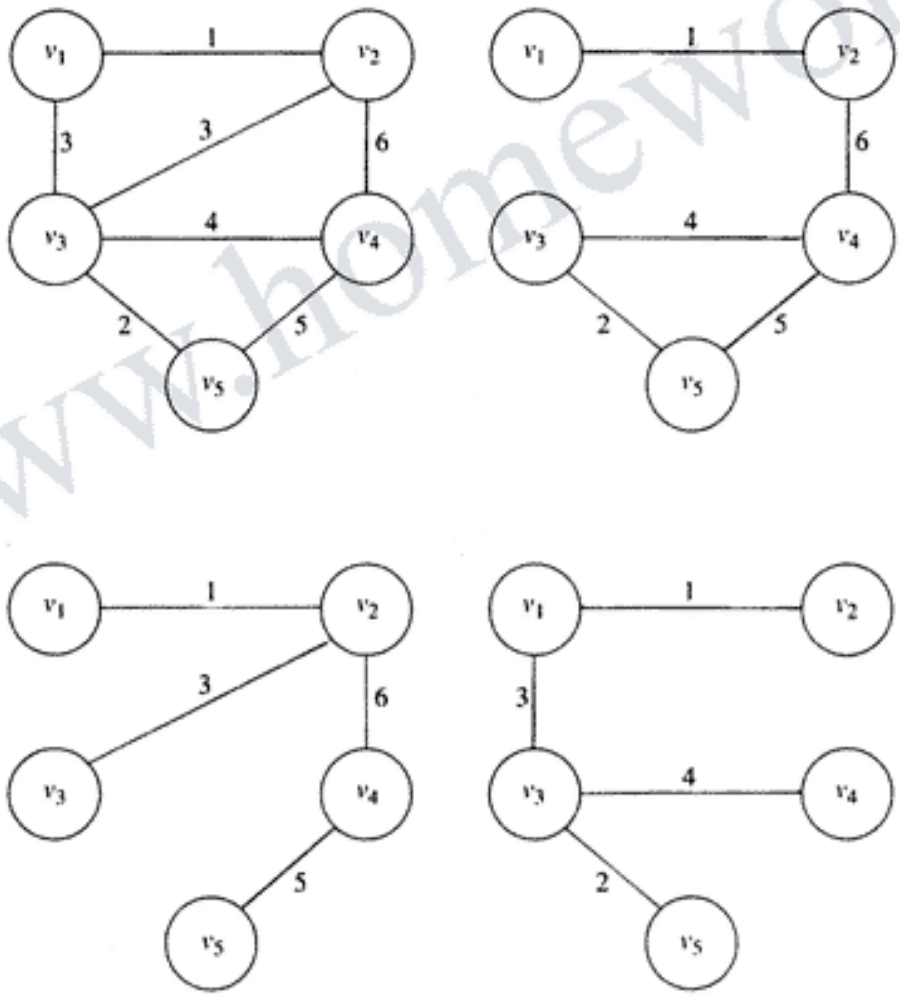
شکل ۲-۴ اگر سکه ۱۲ سنتی وجود داشته باشد، الگوریتم حریص بهینه نیست.

۴-۱ کوچکترین درخت پوشا (درخت پوشای می‌نیمم)

فرض کنید که یک طراح شهری می‌خواهد چند شهر را با جاده‌هایی به هم متصل کند بطوری که امکان حرکت از هر شهری به شهر دیگر وجود داشته باشد. اگر برای طراحی این جاده‌ها بودجه‌ای محدود پیش‌بینی شده باشد، او احتمالاً در طراحی خود، حداقل طول جاده‌ها را درنظر می‌گیرد. می‌خواهیم الگوریتمی ارائه دهیم که بتواند این مسئله و مسائلی از این قبیل را حل نماید. ابتدا مروری گذرا بر تئوری گرافها خواهیم داشت. شکل ۴-۳(a) یک گراف پیوسته، وزن‌دار و بدون‌جهت G را نشان می‌دهد. در اینجا فرض می‌کنیم که وزن لبه‌ها، اعدادی غیرمنفی هستند. گرافی را **بدون‌جهت** گوئیم که لبه‌های آن

(خط ارتباطی بین هر دو گره)، هیچ جهتی نداشته باشند. از آنجائیکه در این حالت لبه‌ها هیچ جهتی ندارند، می‌گوئیم یک لبه بین دو گره قرار دارد. یک **مسیر** در یک گراف بدون‌جهت، دنباله‌ای است از گره‌ها بطوری که یک لبه بین هر گره و گرهٔ بعدیش وجود داشته باشد. چون لبه‌ها بدون جهت هستند، لذا یک مسیر از گره U به گره V وجود خواهد داشت اگر و تنها اگر یک مسیر از گرهٔ V به گرهٔ U وجود داشته باشد. بنابراین، برای گرافهای بدون‌جهت، به سادگی می‌گوئیم که یک مسیر بین دو گره وجود دارد. یک گراف بدون‌جهت، در صورتی **متصل(پیوسته)** است که بین هر دو گرهٔ دلخواه، یک مسیر مشخص وجود داشته باشد. همهٔ گرافهای شکل ۳-۴، از نوع متصل می‌باشند. اگر لبه بین دو گره V_2 و V_4 را در شکل ۳-۴(b) حذف کنیم، دیگر آن گراف، متصل نخواهد بود.

در یک گراف، یک مسیر از یک گره به خودش، **چرخه** نامیده می‌شود و به گرافی که هیچ چرخه‌ای نداشته باشد، گراف **بدون‌چرخه** گوئیم. گراف‌های شکل ۳-۴(c) و ۳-۴(d)، گرافهایی بدون چرخه



شکل ۳-۴ یک گراف وزن‌دار و سه زیرگراف.

و گراف‌های شکل ۳-۴(a) و ۳-۴(b) گراف‌های چرخه دار هستند. یک **درخت**، یک گراف بدون‌جهت، پیوسته و بدون چرخه است. به عبارتی دیگر، یک درخت، یک گراف بدون‌جهت است که در آن دقیقاً یک مسیر بین هر زوج از گره‌ها وجود دارد. گراف‌های شکل ۳-۴(c) و ۳-۴(d)، درخت می‌باشند. با این تعریف، هیچ گره‌ای به عنوان ریشه مشخص نمی‌شود. یک **درخت ریشه‌دار**، درختی است که در آن یک گره به عنوان ریشهٔ درخت مشخص شده باشد. بنابراین، یک درخت ریشه دار، همان چیزی است که ما اغلب آن را به عنوان یک درخت می‌شناسیم (نظیر آنچه که در بخش ۵-۳ انجام شد).

مسئله حذف لبه‌ها از یک گراف بدون‌جهت، وزن‌دار و متصل G و تبدیل آن به زیرگرافی که تمامی گره‌های آن متصل بوده و مجموع وزن‌های لبه‌های باقیمانده، به حداقل ممکن رسیده باشد را درنظر بگیرید. این مسئله می‌تواند کاربردهای متعددی داشته باشد. ممکن است در مسئله طراحی جاده‌ها بخواهیم چند شهر را با حداقل طول جاده‌ها به هم متصل کنیم. بطور مشابه، در لوله کشی یک منطقه می‌خواهیم حداقل اندازهٔ لوله‌ها را بکار بگیریم و در خطوط ارتباطی می‌خواهیم طول کابل مورد استفاده را به حداقل برسانیم. یک زیرگراف با کمترین وزن بایستی حتماً یک درخت باشد زیرا اگر یک زیرگراف، یک درخت نباشد، به این معناست که دارای یک چرخه است و ما می‌توانستیم یک لبه را از روی چرخه حذف کرده و در نتیجه، یک گراف متصل با وزن کمتر داشته باشیم. به شکل ۳-۴ توجه کنید. زیرگراف شکل ۳-۴(b) نمی‌تواند از گراف شکل ۳-۴(a) وزن کمتری داشته باشد زیرا اگر ما هر لبه را از چرخهٔ $[V_3, V_4, V_5, V_3]$ حذف کنیم، باز هم یک زیرگراف متصل باقی می‌ماند. برای مثال، ما می‌توانستیم لبهٔ اتصال‌دهندهٔ گره‌های V_4 و V_5 را حذف کنیم که در اینصورت یک زیرگراف متصل با وزنی کمتر بدست می‌آمد.

یک **درخت پوشا** برای G، یک زیرگراف متصل است که اولاً، شامل همه گره‌های G بوده و ثانیاً، یک درخت باشد. درختهای شکل ۳-۴(c) و ۳-۴(d)، درختهایی پوشا برای G هستند. یک زیرگراف متصل با حداقل وزن باید یک درخت پوشا باشد اما هر درخت پوشایی دارای حداقل وزن نیست. برای مثال، درخت پوشا در شکل ۳-۴(c)، حداقل وزن را دارا نیست زیرا درخت پوشای شکل ۳-۴(d)، وزنی کمتر از آن دارد. یک الگوریتم برای این مسئله بایستی یک درخت پوشا با حداقل وزن را پیدا کند. چنین درختی **کوچکترین درخت پوشا** نامیده می‌شود. درخت شکل ۳-۴(d)، یک درخت پوشای می‌نیمم برای گراف G می‌باشد. لازم به ذکر است که کوچکترین درخت پوشا برای یک گراف، منحصر به فرد نیست.

پیدا کردن یک درخت پوشای می‌نیمم با استفاده از روش brute-force، که در آن تمامی درختهای پوشا درنظر گرفته می‌شود، در بدترین حالت بدتر از حالت نمایی است. ما مسئله را با استفاده از روش حریص، به صورت کاراتری حل خواهیم نمود. البته به تعریف کاملی از یک گراف بدون‌جهت نیاز داریم.

تعریف یک **گراف بدون‌جهت** G، شامل یک مجموعه متناهی V، که اعضای آن گره‌های G نامیده می‌شوند و یک مجموعه E، که شامل جفت گره‌های V است، می‌باشد. این دو مجموعه، لبه‌های G نامیده می‌شوند که آن را به صورت زیر نشان می‌دهیم:

$$G = (V, E)$$

اعضای مجموعهٔ V به صورت V_i و لبهٔ بین دو گره V_i و V_j، به صورت (V_i, V_j) نمایش داده می‌شوند.

مثال ۱-۴ برای گراف شکل ۴-۳(a) داریم

$$V = \{V_1, V_2, V_3, V_4, V_5\}$$
$$V = \{(V_1, V_2), (V_1, V_3), (V_2, V_3), (V_2, V_4), (V_3, V_4), (V_3, V_5), (V_4, V_5)\}$$

ترتیب گره‌های یک جفت گره در معرفی یک لبه در گراف بدون‌جهت، اهمیتی ندارد. برای مثال (V_1, V_2) و (V_2, V_1)، هر دو به یک لبه از گراف اشاره می‌کنند. ما در معرفی لبه‌ها، گرهٔ با اندیس کوچکتر را در ابتدای زوج گره می‌نویسیم.

مجموعه گره‌های یک درخت پوشای T برای G، همان مجموعه گره‌های گراف G یعنی V می‌باشد. اما مجموعهٔ لبه‌های T، یک زیر مجموعهٔ F از مجموعهٔ E است بطوری که T = (V, F)، کوچکترین درخت پوشا برای G می‌باشد. یک الگوریتم حریص سطح بالا برای این مسئله می‌تواند به صورت زیر مطرح شود:

```
F=∅;                                                    // مقداردهی مجموعه لبه ها به تهی
while (the instance is not solved){
    select an edge to some locally optimal consideration;  // روال انتخاب
    if (adding the edge to F does not creat a cycle)       // بررسی امکان سنجی
        add it;
    if (T = (V, F) is a spanning tree)                     // بررسی جواب
        the instance is solved;
}
```

در اینجا دو الگوریتم مختلف حریص را برای این مسئله بررسی خواهیم کرد. الگوریتم Prim و الگوریتم Kruskal، که هر یک از ویژگی بهینهٔ محلی متفاوتی استفاده می‌کنند. یادآور می‌شویم که هیچ تضمینی وجود ندارد که از یک الگوریتم حریص داده شده، یک جواب بهینه بدست آید و این مسئله‌ای است که لازم است وجود یا عدم وجود چنین حالتی در آن اثبات شود. ما ثابت خواهیم کرد که هر دو الگوریتم prim و Kruskal همواره کوچکترین درخت پوشا را تولید می‌کنند.

۱-۱-۴ الگوریتم Prim

الگوریتم Prim با یک زیرمجموعهٔ خالی از لبه‌ها به نام F و یک زیرمجموعه از گره‌ها به نام Y شروع می‌شود که در ابتدا شامل یک گرهٔ دلخواه می‌باشد. زیر مجموعه Y را با $\{V_1\}$ مقداردهی اولیه می‌کنیم. نزدیکترین گره به Y، گره‌ای است در V - Y، که توسط لبه‌ای با کمترین وزن به گره‌ای در Y وصل می‌شود.

(به خاطر دارید که در فصل ۳، اصطلاحات وزن و فاصله برای گرافهای وزن‌دار، معادل هم استفاده می‌شد.) در شکل ۳-۴(a)، V_2 نزدیکترین گره به Y است وقتی که $y = \{V_1\}$ باشد. گره‌ای که به Y نزدیکتر است، به مجموعهٔ Y اضافه شده و لبهٔ آن نیز به مجموعهٔ F افزوده می‌گردد(اتصال‌ها به دلخواه شکسته می‌شوند). در این حالت V_2 به Y اضافه شده و لبهٔ (V_1, V_2) نیز به F افزوده می‌گردد. این فرآیند افزودن نزدیکترین گره‌ها تا وقتی که Y = V شود، تکرار می‌گردد. در زیر یک الگوریتم سطح بالا برای این روال ارائه می‌دهیم:

```
F=∅;                                          // مقداردهی مجموعه لبه ها به تهی
Y={V1};                                       // مقدار دهی مجموعه گره ها به اولین رأس
while(the instance is not solved){
    select a vertex in V - Y that is nearest to Y;   //روال انتخاب و بررسی امکان‌سنجی
    add vertex v to Y;
    add the edge to F;
    if(Y == V)                                // بررسی جواب
      the instance is solved;
}
```

روالهای انتخاب و بررسی امکان‌سنجی با هم اجرا می‌شوند. زیرا گرفتن یک گرهٔ جدید از Y - V تضمین می‌کند که چرخه‌ای به وجود نمی‌آید. شکل ۴-۴، الگوریتم Prim را نشان می‌دهد. در هر مرحله از این شکل، Y شامل گره‌های سایه‌دار و F شامل لبه‌های سایه‌دار می‌باشد.

این الگوریتم سطح بالا در تولید کوچکترین درخت پوشا برای یک گراف کوچک به خوبی کار می‌کند. انسان صرفاً نزدیکترین گره به Y را با یک بررسی دقیق پیدا می‌کند. با وجود این، به منظور نوشتن الگوریتمی که بتواند در یک زبان برنامه‌نویسی پیاده‌سازی شود، نیاز به تعریف یک روال گام به گام داریم. بدین منظور، یک گراف وزن‌دار را به وسیله ماتریس مجاورش نشان می‌دهیم. یعنی آن را به وسیلهٔ یک آرایهٔ W با ابعاد n×n نشان می‌دهیم که در آن

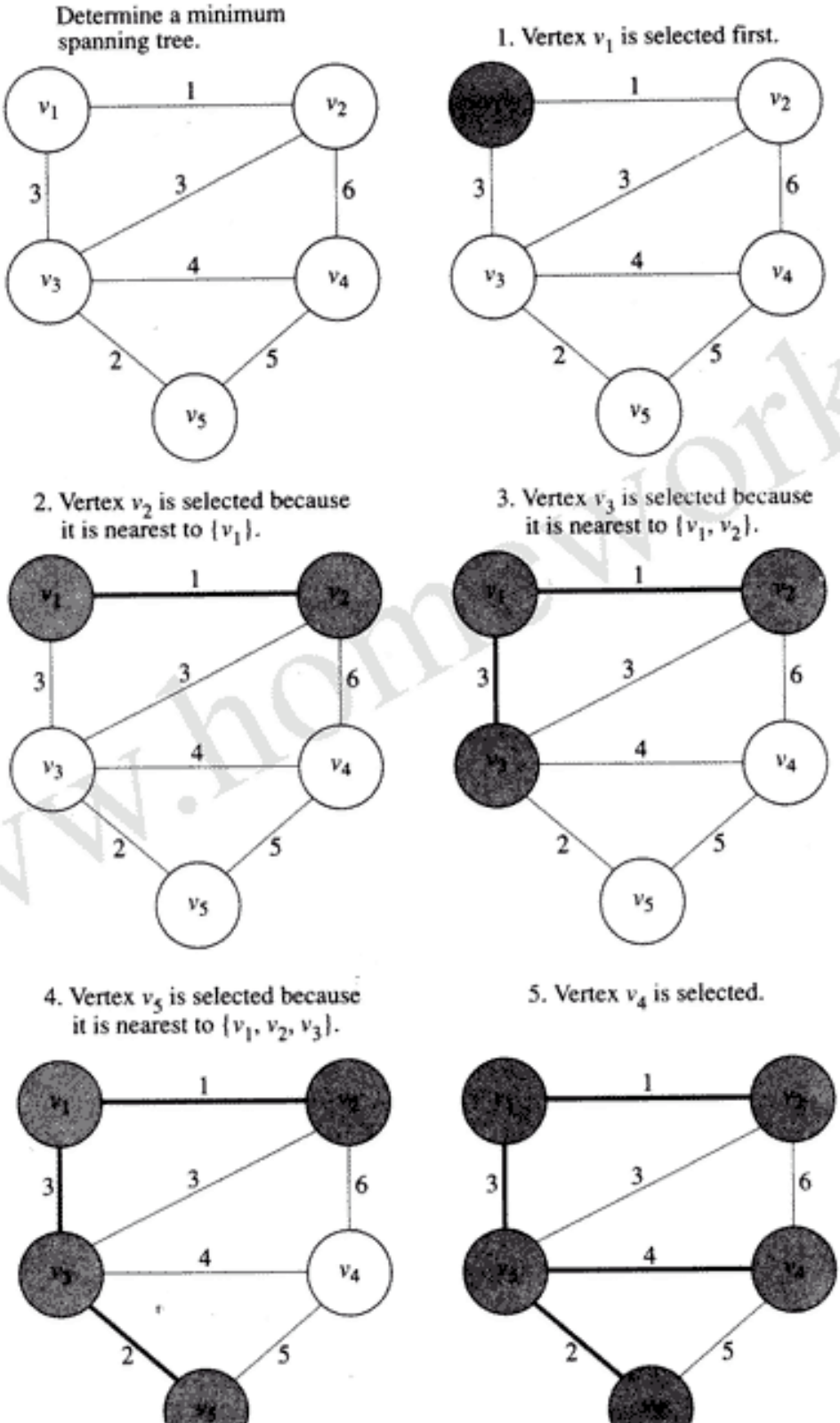
$$W[i][j] = \begin{cases} \text{وزن لبه} & \text{بین } V_i \text{ و } V_j \text{ یک لبه وجود دارد} \\ \infty & \text{بین } V_i \text{ و } V_j \text{ لبه‌ای وجود ندارد} \\ 0 & i = j \end{cases}$$

شکل ۵-۴ نمایانگر گراف شکل ۳-۴(a) براساس این روش می‌باشد. ما به دو آرایه نیاز داریم؛ یکی nearest و دیگری distance که برای $i = 2, \ldots, n$،

nearst[i] = شاخص گره‌ای در Y که نزدیکترین گره به V_i است.

distance[i] = وزن لبهٔ بین V_i و گره‌ای که به وسیله nearest[i] شاخص‌دهی شده است.

شکل ۴-۴ یک گراف وزن‌دار (در گوشهٔ بالای سمت چپ) و مراحل الگوریتم prim برای آن گراف در هر مرحله. گره‌های زیرمجموعه Y و لبه‌های زیرمجموعهٔ F سایه‌دار می‌شوند.



از آنجائیکه در لحظه شروع $y = \{V_1\}$ می‌باشد، لذا nearest[i] با ۱ و distanec[i] با وزن لبهٔ بین V_1 و V_i مقداردهی اولیه می‌شود. همانطوری که گره‌ها به Y اضافه می‌شوند، این دو آرایه برای ارجاع گرهٔ جدید در Y به نزدیکترین گرهٔ خارج از Y، بهنگام(update) می‌شوند. برای معین کردن گره‌ای که باید به Y اضافه شود، در هر تکرار، شاخصی که مقدار distanec[i]l آن می‌نیمم است را محاسبه می‌کنیم. این شاخص را Vnear می‌نامیم. با مقداردهی distance[vnear] به ۱-، گرهٔ با شاخص vnear به Y اضافه می‌گردد. الگوریتم زیر، این روال را پیاده سازی می‌کند.

الگوریتم ۱-۴ الگوریتم Prim

مسئله: یافتن کوچکترین درخت پوشا.

ورودی: عدد صحیح $n \geq 2$ و یک گراف بدون‌جهت، وزن‌دار و پیوسته شامل n گره. گراف توسط یک آرایهٔ دو بعدی W که سطرها و ستونهایش از ۱ تا n شاخص‌دهی شده‌اند نشان داده می‌شود، که در آن W[i][j] معرف وزن لبهٔ بین گره iام و گره jام است.

خروجی: مجموعه‌ای از لبه‌ها F در یک درخت پوشای می‌نیمم برای گراف.

```
void prim (int n,
           const number W[ ][ ],
           set_of_edges& F)
{
  index i, vnear;
  number min;
  edge e;
  index nearest[2..n];
  number distance[2..n];

  F = ∅;
  for (i = 2; i <= n; i++) {
    nearest[i] = 1;                    // For all vertices, initialize v1
    distance[i] = W[1][i];             // to be the nearest vertex in
  }                                    // Y and initialize the distance
                                       // from Y to be the weight
                                       // on the edge to v1.
  repeat (n - 1 times) {               // Add all n - 1 vertices to Y.
    min = ∞;
    for (i = 2; i <= n; i++)           // Check each vertex for
      if (0 ≤ distance[i] < min) {     // being nearest to Y.
        min = distance[i];
        vnear = i;
      }
    e = edge connecting vertices indexed
        by vnear and nearest[vnear];
```

```
    add e to F;
    distance[vnear] = -1;                    // Add vertex indexed by
    for (i = 2; i <= n; i++)                 // vnear to Y.
      if (W[i][vnear] < distance[i]) {       // For each vertex not in Y,
        distance[i] = W[i][vnear];           // update its distance from Y.
        nearest[i] = vnear;
      }
  }
}
```

تحلیل پیچیدگی زمانی حالت معمول الگوریتم ۱-۴ (الگوریتم Prim)

عمل مبنایی: دو حلقه وجود دارد که هر یک با n-۱ تکرار در داخل حلقه repeat قرار دارند. اجرای دستورالعملهای داخل هر یک از آنها می‌تواند به عنوان یک عمل مبنایی درنظر گرفته شود.

اندازه ورودی: n تعداد گره‌ها.

چون حلقه repeat، n-۱ مرتبه تکرار می‌شود، لذا پیچیدگی زمانی آن برابر است با

$$T(n) = ۲(n-۱)(n-۱) \in \Theta(n^۲)$$

واضح است که الگوریتم prim، یک درخت پوشا تولید می‌کند. با وجود این، آیا درخت حاصل همیشه می‌نیمم است؟ چون در هر مرحله، ما نزدیکترین گره به Y را انتخاب می‌کنیم، بنظر می‌رسد که درخت حاصل بایستی می‌نیمم باشد. به هر حال، باید درستی این مطلب ثابت شود. اگرچه الگوریتم‌های حریص اغلب ساده‌تر از الگوریتم‌های برنامه‌نویسی پویا نوشته می‌شوند؛ ولی معمولاً تعیین اینکه آیا یک الگوریتم حریص همیشه یک جواب بهینه تولید می‌کند یا نه، بسیار مشکل است. حتماً بخاطر دارید که برای یک الگوریتم برنامه‌نویسی پویا فقط بایستی نشان دهیم که اصل بهینگی در آن بکار رفته است. اما برای یک الگوریتم حریص به اثبات کامل نیاز داریم. در ادامه، چنین اثباتی را برای الگوریتم prim ارائه می‌دهیم.

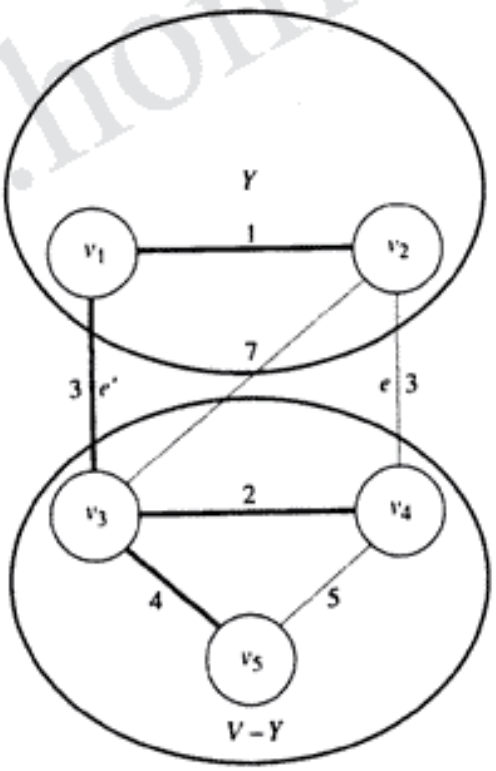
	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

شکل ۵-۴ آرایهٔ متناظر با گراف شکل ۲-۴(a).

گراف بدون‌جهت $G=(V,E)$ را در نظر بگیرید. یک زیرمجموعهٔ F از E را **وعده‌گاه (promising)** گوئیم اگر لبه‌ها بتوانند برای شکل‌گیری یک درخت پوشای می‌نیمم به آن اضافه شوند. زیرمجموعهٔ $\{(V_1,V_2),(V_1,V_3)\}$ در شکل ۴-۳(a) یک وعده‌گاه است، اما زیرمجموعهٔ $\{(V_2,V_4)\}$ یک وعده‌گاه نیست.

پیش‌قضیه ۴-۱ فرض کنید که $G=(V,E)$ یک گراف بدون‌جهت، وزن‌دار و متصل، F یک زیرمجموعه از E به عنوان وعده‌گاه و Y مجموعه‌ای از گره‌های متصل به وسیلهٔ لبه‌های F باشد. اگر e یک لبه با کوچکترین وزن باشد که یک گره در Y را به گره‌ای دیگر در V - Y وصل می‌کند، آنگاه $FU\{e\}$ وعده‌گاه خواهد بود.

اثبات: از آنجائیکه F یک وعده‌گاه است، لذا بایستی مجموعه‌ای از لبه‌ها F' موجود باشد بطوری که $F\subset F'$ بوده و (V,F') یک درخت پوشای می‌نیمم باشد. اگر $e\in F'$ باشد، آنگاه $FU\{e\}\subset F'$ است؛ یعنی اینکه $FU\{e\}$ یک وعده‌گاه است که در اینصورت اثبات انجام شده است. در غیراینصورت، چون (V,F') یک درخت پوشا است، لذا $F'U\{e\}$ بایستی دقیقاً شامل یک چرخه بوده و e نیز باید در این چرخه قرار داشته باشد. شکل ۴-۶، این موضوع را نشان می‌دهد. چرخه شامل $[V_1,V_2,V_4,V_3]$ می‌باشد.



شکل ۴-۶ یک گراف متناظر با پیش قضیهٔ ۴-۱. لبه‌ها در F سایه‌دار شده‌اند.

همانطوریکه درشکل ۶-۴ مشاهده می‌کنید، بایستی یک لبهٔ دیگر $e' \in F'$ نیز در چرخه باشد که یک گره در Y را به گره‌ای دیگر در V - Y وصل کند. اگر ما e' را از $F' U \{e\}$ حذف کنیم، چرخه از بین می‌رود و این بدین معناست که ما یک درخت پوشا داریم. از آنجائیکه e یک لبه با کمترین وزن است که یک گره در Y را به گره‌ای دیگر در V - Y وصل می‌کند، وزن لبه e باید کمتر یا مساوی وزن e' باشد (در واقع، آنها باید مساوی باشند). لذا، $F' \cup \{e\} - \{e'\}$ کوچکترین درخت پوشا است. اکنون $F \cup \{e\} \subset F' \cup \{e\} - \{e'\}$ است زیرا e' نمی‌تواند در F باشد (به خاطر دارید که لبه‌های F فقط به گره‌هایی در V متصل می‌شوند). بنابراین، $F U \{e\}$ یک وعده‌گاه است و بدین ترتیب، اثبات ما کامل می‌شود.

قضیه ۱-۴ الگوریتم prim همواره یک درخت پوشای می‌نیمم تولید می‌کند.

اثبات: با استفاده از استقرا نشان می‌دهیم که مجموعه F بعد از هر تکرار از حلقه repeat، یک وعده‌گاه می‌باشد.

پایه استقراء: واضح است که مجموعه تهی (∅) یک وعده‌گاه است.

فرض استقراء: فرض کنید که بعد از هر تکرار معین از حلقهٔ repeat، مجموعه لبه‌های که تاکنون انتخاب شده‌اند -موسوم به F- وعده‌گاه می‌باشد.

گام استقراء: باید نشان دهیم که مجموعهٔ $F \cup \{e\}$، که درآن e یک لبه انتخاب شده در حلقه بعدی است، وعده‌گاه می‌باشد. از آنجائیکه لبهٔ انتخاب شده e در تکرار بعدی یک لبه با وزن می‌نیمم است که گره‌ای در Y را به گره‌ای دیگر در V - Y متصل می‌کند، لذا با توجه به پیش قضیه ۱-۴، $F \cup \{e\}$ یک وعده‌گاه است. این مطلب، اثبات استقراء را کامل می‌کند.

با توجه به اثبات فوق، مجموعهٔ نهایی از لبه‌ها یک وعده‌گاه است و چون این مجموعه شامل لبه هایی در یک درخت پوشا است، پس درخت آن نیز بایستی یک درخت پوشای می‌نیمم باشد.

۲-۱-۴ الگوریتم Kruskal

الگوریتم Kruskal برای مسئله کوچکترین درخت پوشا، ابتدا زیر مجموعه‌های غیرالحاقی V را برای هر گره و فقط شامل همان گره تولید می‌کند. سپس لبه‌ها را به ترتیب غیرنزولی وزن مورد بررسی قرار می‌دهد (اتصالهابه دلخواه شکسته می‌شوند). اگر لبه‌ای دو گره را در زیر مجموعه‌ای مجزا به هم وصل کند، آن لبه به زیرمجموعه اضافه شده و زیرمجموعه‌ها نیز با هم ادغام می‌شوند تا یک مجموعه پدید آورند. این فرآیند تکرار می‌شود تا اینکه همه زیرمجموعه‌ها با هم ادغام شوند و به یک مجموعه تبدیل گردند. در زیر یک الگوریتم سطح بالا برای این روال ارائه می‌دهیم.

```
F=∅;                                                    // مقداردهی مجموعه لبه ها به تهی
creat disjoint subsets of V, one for
each vertex and containing only that vertex;
sort the edges in E in nondecreasing order;
while(the instance is not solved){
    select next edge;                                   // روال انتخاب
    if(the edge connects two vertices in disjoint subsets){ // بررسی امکان سنجی
      merge the subsets;
      add the edge to F;
    }
    if(all the subsets are merged)                      // بررسی جواب
      the instance is solved;
}
```

شکل ۷-۴، الگوریتم kruskal را نشان می‌دهد. برای نوشتن یک نسخه کامل از این الگوریتم، به یک نوع داده‌ای مجرد برای مجموعهٔ غیرالحاقی نیازمندیم. این نوع داده‌ای در ضمیمه C پیاده‌سازی شده است. از آنجائیکه این پیاده‌سازی برای مجموعه‌های غیرالحاقی شاخص‌ها است، لذا برای استفاده از آن کافیست به وسیلهٔ شاخص به گره‌ها ارجاع کنیم. نوع داده مجرد مجموعه‌های غیرالحاقی، شامل انواع داده‌ای index، Set_Pointer و نیز روالهای initial، find، merge و equal می‌باشد بطوری که اگر تعریفی به صورت زیر داشته باشیم:

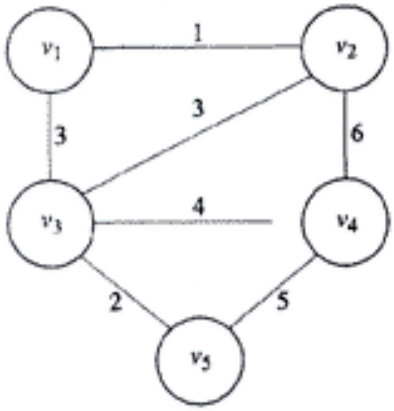
```
index i;
Set_Pointer p, q;
```

آنگاه

- initial(n)، n زیرمجموعهٔ غیرالحاقی را مقداردهی اولیه می‌کند که هر یک شامل دقیقاً یکی از شاخصهای بین ۱ و n می‌باشد.
- p = find(i) سبب می‌شود که p به مجموعهٔ شامل شاخص i اشاره کند.
- merge(p, q)، دو مجموعه‌ای که p و q به آنها اشاره می‌کنند را درهم ادغام نموده و به یک مجموعه تبدیل می‌کند.
- equal(p, q) مقدار True را برمی‌گرداند اگر p و q به یک مجموعه اشاره کنند.

شکل ۷-۴ یک گراف وزن‌دار (در گوشه راست بالای صفحه) و مراحل الگوریتم Kruskal برای گراف.

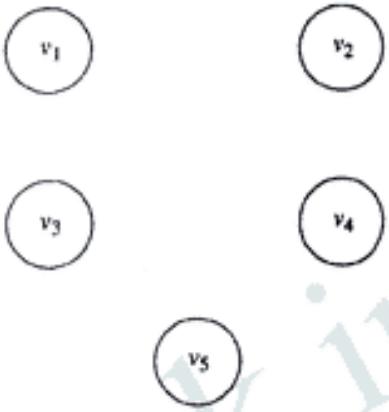
Determine a minimum spanning tree.



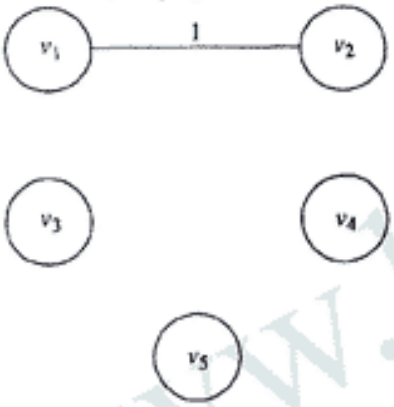
1. Edges are sorted by weight.

(v_1, v_2) 1
(v_3, v_5) 2
(v_1, v_3) 3
(v_2, v_3) 3
(v_3, v_4) 4
(v_4, v_5) 5
(v_2, v_4) 6

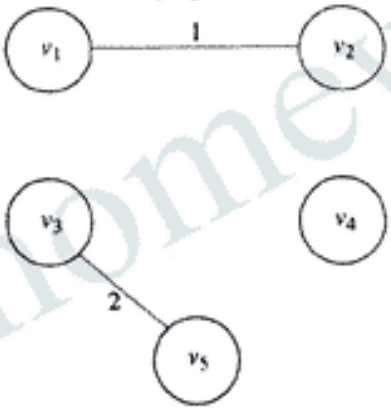
2. Disjoint sets are created.



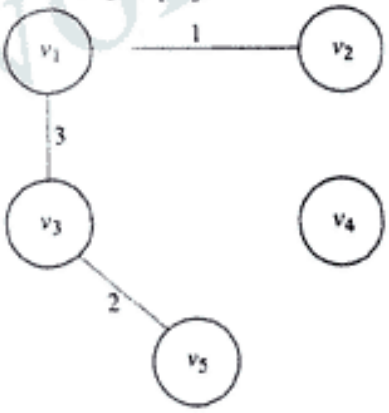
3. Edge (v_1, v_2) is selected.



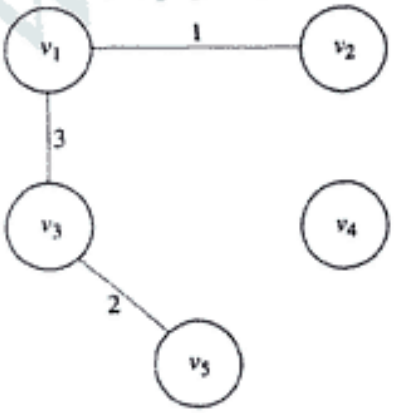
4. Edge (v_3, v_5) is selected.



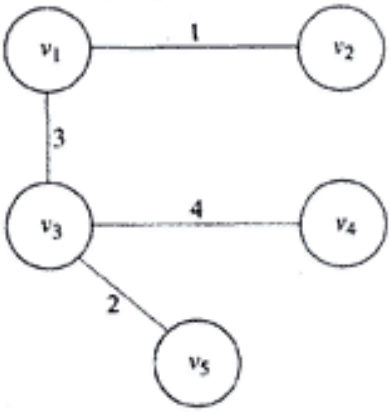
5. Edge (v_1, v_3) is selected.



6. Edge (v_2, v_3) is rejected.



7. Edge (v_3, v_4) is selected.



الگوریتم ۲-۴ الگوریتم Kruskal

مسئله: یک درخت پوشای می‌نیمم، مشخص کنید.

ورودی: عدد صحیح $n \geq 2$، عدد صحیح مثبت m و یک گراف بدون‌جهت، وزن‌دار و متصل شامل n گره و m لبه. گراف با یک مجموعهٔ E که شامل لبه‌های گراف همراه با وزن‌های آنها است، نشان داده می‌شود.

خروجی: مجموعه‌ای از لبه‌ها F در یک درخت پوشای می‌نیمم.

```
void kruskal (int n, int m,
              set_of_edges E,
              set_of_edges& F)
{
  index i, j;
  set_pointer p, q;
  edge e;
  Sort the m edges in E by weight in nondecreasing order;
  F=∅;
  initial(n);                                  // مقداردهی n زیرمجموعه غیرالحاقی
  while(number of edges in F is less than n-1){
    e = edge with least weight not yet considered;
    i, j = indices of vertices connected by e;
    p = find(i);
    q = find(j);
    if(!equal(p, q)){
      merge(p, q);
      add e to F;
    }
  }
}
```

هرگاه n-۱ لبه در F وجود داشته باشد، از حلقهٔ while خارج می‌شویم؛ زیرا دراینصورت، n-۱ لبه در یک درخت پوشا وجود خواهد داشت.

تحلیل پیچیدگی زمانی بدترین حالت الگوریتم ۳-۴ (الگوریتم Kruskal)

عمل مبنایی: یک دستورالعمل مقایسه.

اندازه ورودی: n، تعداد گره‌ها و m، تعداد لبه‌ها.

سه نکته در این الگوریتم وجود دارد.

۱- مدت زمان مرتب سازی لبه‌ها. در فصل ۲، یک الگوریتم مرتب‌سازی (Mergesort) که در بدترین حالت، در $\Theta(m \lg m)$ بود را بدست آوردیم. در فصل ۷ نشان خواهیم داد که بهبود کارایی

الگوریتم‌هایی که به وسیلهٔ مقایسه، کلیدها را مرتب می‌کنند، امکان‌پذیر نیست. لذا پیچیدگی زمانی مرتب‌سازی لبه‌ها برابراست با

$$W(m) \in \Theta(m \lg m)$$

۲- زمان در حلقه while، مدت زمان لازم برای دستکاری مجموعه‌های غیرالحاقی در این حلقه بسیار مهم است (زیرا موارد دیگر ثابت هستند). در بدترین حالت، هر لبه قبل از اینکه از حلقهٔ while خارج شود، در نظر گرفته می‌شود. بدین معنا که m گذر از این حلقه وجود دارد. با استفاده از ساختار داده‌ای مجموعه‌های غیرالحاقی II در ضمیمه C، پیچیدگی زمانی m گذر از یک حلقه، شامل تعداد ثابتی از فراخوانی روالهای find، equal و merge برابر است با

$$W(m) \in \Theta(m \lg m)$$

که در آن عمل مبنایی یک دستورالعمل مقایسه است.

۳- مدت زمان مقداردهی اولیه n مجموعه غیرالحاقی. با استفاده از ساختار داده‌ای مجموعه‌های غیرالحاقی، پیچیدگی زمانی این مقداردهی برابراست با

$$T(n) \in \Theta(n)$$

از آنجائیکه $m \geq n - ۱$ است، لذا مرتب سازی و دستکاری مجموعه‌های غیرالحاقی بر زمان مقداردهی آنها برتری دارد. به این معنی که

$$W(m,n) \in \Theta(m \lg m)$$

بنظر می‌رسد که بدترین حالت، به مقدار n بستگی ندارد. با وجود این، در بدترین حالت، هر گره می‌تواند به هر گرهٔ دیگری وصل شود. یعنی

$$m = \frac{n(n-۱)}{۲} \in \Theta(n^۲)$$

بنابراین می‌توانیم بدترین حالت را به صورت زیر نیز بنویسیم:

$$W(m,n) \in \Theta(n^۲ \lg n^۲) = \Theta(n^۲ ۲ \lg n) = \Theta(n^۲ \lg n)$$

بهتر است که برای مقایسه الگوریتم‌های Kruskal و Prim، از هر دو عبارت فوق برای بدترین حالت استفاده کنیم.

برای اثبات اینکه الگوریتم Kruskal، همیشه یک جواب بهینه تولید می‌کند، به پیش قضیه زیر نیاز داریم.

پیش‌قضیه ۲-۴ فرض کنید که $G(V,E)$ یک گراف بدون‌جهت، وزن‌دار و متصل، F زیرمجموعه‌ای از E به عنوان وعده‌گاه و e یک لبه با کمترین وزن در E - F است بطوری که در $F \cup \{e\}$ هیچ چرخه‌ای وجود ندارد. در اینصورت $F \cup \{e\}$ یک وعده‌گاه است.

اثبات: اثبات این پیش قضیه مشابه اثبات پیش قضیه ۱-۴ است. از آنجائیکه F یک وعده‌گاه است، لذا بایستی مجموعه‌ای از لبه‌ها به نام F' وجود داشته باشد بطوری که $F \subseteq F'$ بوده و (V, F') یک درخت پوشای می‌نیمم باشد. اگر $e \in F'$ باشد، دراینصورت $F' \subseteq F \cup \{e\}$ است، بدین معناکه $F \cup \{e\}$ وعده‌گاه بوده و اثبات کامل می‌شود. در غیراینصورت، چون (V, F') یک درخت پوشا است، لذا $F' \cup \{e\}$ باید شامل دقیقاً یک چرخه باشد و e نیز باید در این چرخه وجود داشته باشد. از آنجائیکه $F \cup \{e\}$ شامل هیچ چرخه‌ای نیست، لذا بایستی تعدادی لبه $e' \in F'$ وجود داشته باشد بطوری که در چرخه باشند اما در F نباشند. یعنی $e' \in E - F$ است. مجموعه $F \cup \{e'\}$، چرخه‌ای ندارد زیرا یک زیرمجموعه از F' است. بنابراین، وزن e بزرگتر از وزن e' نیست. (می‌دانیم که طبق فرض، e یک لبه با کمترین وزن در E - F است بطوری که هیچ چرخه‌ای در $F \cup \{e\}$ وجود ندارد). اگر ما e' را از $F' \cup \{e\}$ حذف نمائیم، چرخهٔ این مجموعه از بین می‌رود؛ یعنی ما یک درخت پوشا داریم. در حقیقت $F' \cup \{e\} - \{e'\}$ یک درخت پوشای می‌نیمم است زیرا همانطوریکه نشان دادیم، وزن e بزرگتر از وزن e' نیست. از آنجائیکه e' در F نیست، لذا $F \cup \{e\} - \{e'\} \subseteq F' \cup \{e\}$ می‌باشد. بنابراین، $F \cup \{e\}$ وعده‌گاه است که این، اثبات ما را کامل می‌کند.

قضیه ۲-۴ الگوریتم Kruskal همواره کوچکترین درخت پوشا را تولید می‌کند.

اثبات: از طریق استقراء و با شروع از یک مجموعه خالی از لبه‌ها، اثبات انجام می‌شود. در تمرینات از شما می‌خواهیم که با استفاده از پیش قضیه ۲-۴، این موضوع را ثابت کنید.

۴-۱-۳ مقایسه الگوریتم Prim با الگوریتم Kruskal

ما پیچیدگیهای زمانی زیر را بدست آورده‌ایم:

الگوریتم Prim: $T(n) \in \Theta(n^2)$

الگوریتم Kruskal: $W(n,m) \in \Theta(m \lg m)$, $W(m,n) \in \Theta(n^2 \lg n)$

همچنین نشان دادیم که در یک گراف متصل،

$$n - 1 \leq m \leq n(n-1)/2$$

برای گرافی که تعداد لبه‌های آن m، بسیار نزدیک به حد پائین عبارت فوق باشد (گراف بسیار پراکنده است)، الگوریتم Kruskal در $\Theta(n \lg n)$ است؛ بدین معنا که این الگوریتم بایستی سریعتر عمل کند. در حالیکه برای گرافی که تعداد لبه‌های آن نزدیک به حد بالا است (گراف بسیار متصل است)، الگوریتم Kruskal در $\Theta(n^2 \lg n)$ است؛ یعنی الگوریتم Prim بایستی سریعتر عمل کند.

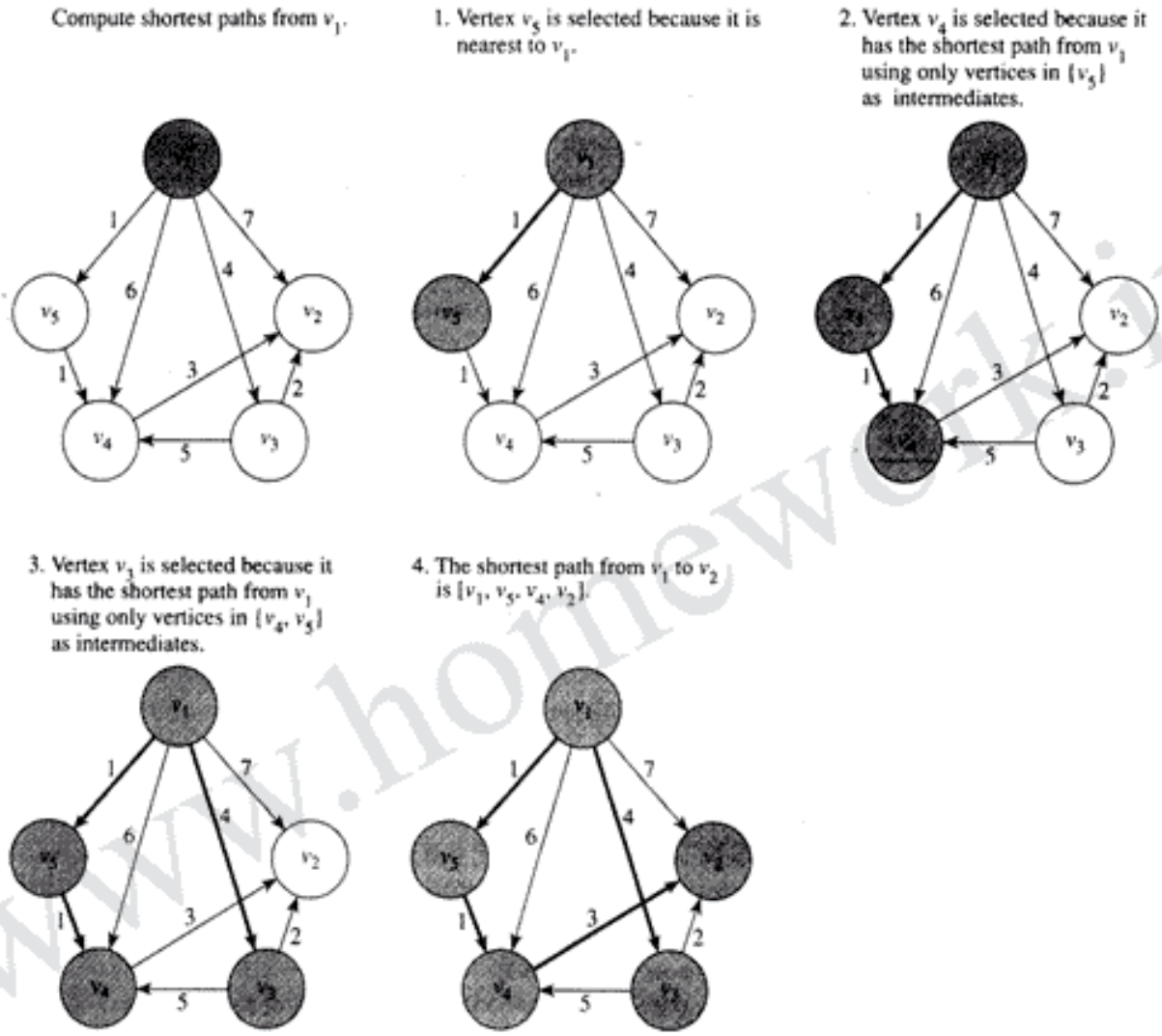
۴-۲ الگوریتم Dijkstra برای مسئله کوتاهترین مسیرهای تک مبدأیی

در بخش ۲-۳، یک الگوریتم $\Theta(n^3)$ برای تعیین کوتاهترین مسیرها از هر گره به گره‌های دیگر در یک گراف جهت دار و وزن‌دار ارائه نمودیم. اگر می‌خواستیم فقط کوتاهترین مسیرها را از یک گره بخصوص به تمام گره‌های دیگر مشخص کنیم، توان الگوریتم مذکور، بیش از حد لازم می‌شد. در ادامه می‌خواهیم با استفاده از روش حریص، یک الگوریتم $\Theta(n^2)$ برای این مسئله (که به مسئله کوتاهترین مسیرهای تک مبدأیی موسوم است) ارائه نمائیم. Dijkstra این الگوریتم را در سال ۱۹۵۹ مطرح نموده است و ما این الگوریتم را با فرض اینکه یک مسیر از گرهٔ مورد نظر به هر یک از گره‌های دیگر وجود دارد، ارائه می‌کنیم. این الگوریتم، شبیه به الگوریتم Prim برای مسئله کوچکترین درخت پوشا است. مجموعهٔ Y را در ابتدا به گره‌ای که قرار است کوتاهترین مسیرهای منتهی به آن تعیین شود، مقداردهی اولیه می‌کنیم. فرض می‌کنیم که گرهٔ مورد نظر، V_1 است. مجموعه لبه‌ها F را تهی درنظر می‌گیریم. در ابتدا گره V، نزدیکترین گره به V_1 را به Y و لبه $<V_1, V>$ را به F اضافه می‌کنیم (منظور از $<V_1, V>$ این است که جهت لبه از گره V_1 به گره V می‌باشد). این لبه، به وضوح کوتاهترین مسیر از V_1 به V است. سپس کلیه مسیرها از گره V_1 به گره‌های مجموعه V - Y را با در نظر گرفتن این نکته که فقط گره‌های مجموعه Y به عنوان گره‌های میانی می‌باشند، بررسی می‌کنیم. هر یک از این کوتاهترین مسیرها به عنوان یک مسیر بهینه است (که بایستی ثابت شود). گرهٔ پایانی چنین مسیری را به Y و لبه‌ای که ما را به این گره می‌رساند به F اضافه می‌کنیم. این روال تا زمانی ادامه می‌یابد که Y معادل V (مجموعه کلیه گره‌ها) شود. اینجاست که F شامل تمامی کوتاهترین مسیرها می‌شود. الگوریتم سطح بالای این روش، به صورت زیر است:

$F = \emptyset$;
$Y = \{V_1\}$;
while(the instance is not solved){
    select a vertex v in V - Y that has a shortest path from V_1, using only vertices in Y as intermediates; // روال انتخاب و بررسی امکان سنجی
    add the new vertex v to Y;
    add the edge (on the shortest path) that touches v to F;
    if(Y == V) // بررسی جواب
      the instance is solved;
}

شکل ۸-۴، الگوریتم Dijkstra را نشان می‌دهد. این الگوریتم سطح بالا، همانند الگوریتم prim، تنها برای حل یک نمونه مسئله با روش بررسی یک گراف کوچک کار می‌کند. برای این الگوریتم، گراف وزن‌دار را با یک آرایهٔ دوبعدی، دقیقاً همانند آنچه که در بخش ۲-۳ انجام دادیم، نشان می‌دهیم. این الگوریتم، بسیار شبیه به الگوریتم ۱-۴ (prim) است؛ با این تفاوت که به جای آرایه‌های nearest و distance، از آرایه‌های

شکل ۸-۴ یک گراف وزن‌دار و جهت‌دار (در گوشهٔ سمت راست بالا) و مراحل الگوریتم dijkstra برای آن گراف. گره‌های مجموعه Y و لبه‌های مجموعه F در هر مرحله سایه دار شده‌اند.



touch و length استفاده می‌کنیم که برای $i = ۲, \ldots, n$،

touch[i] = شاخص گرهٔ V در Y بطوری که لبهٔ $< V_i, V >$، آخرین لبه روی کوتاهترین مسیر جاری از V_1 به V_i است که در آن تنها از گره‌های موجود در Y به عنوان گره‌های میانی استفاده شده است.

length[i] = طول کوتاهترین مسیر جاری از V_1 به V_i است که در آن تنها از گره‌های موجود در Y به عنوان گره‌های میانی استفاده شده است.

الگوریتم Dijkstra به صورت زیر است.

الگوریتم ۳-۴ الگوریتم Dijkstra

مسئله: کوتاهترین مسیرها از V_1 به تمامی گره‌های یک گراف وزن‌دار و جهت دار را تعیین کنید.

ورودی: عدد صحیح $n \geq 2$، و یک گراف جهت‌دار، وزن‌دار و متصل شامل n گرهٔ گراف که با یک آرایهٔ دو بعدی W که سطرها و ستونهایش از ۱ تا n شاخص‌دهی شده و در آن W[i][j] وزن لبه از گره iام به گره jام است، نشان داده می‌شود.

خروجی: مجموعه لبه‌ها F شامل لبه‌های کوتاهترین مسیرها.

```
void dijkstra (int n,
               const number W[ ][ ],
               set_of_edges& F)
{
  index i, vnear;
  edge e;
  index touch[2..n];
  number length[2..n];

  F = ∅;
  for (i = 2; i <= n; i++) {              // For all vertices, initialize v1
    touch[i] = 1;                         // to be the last vertex on the
    length[i] = W[1][i];                  // current shortest path from
  }                                       // v1, and initialize length of
                                          // that path to be the weight
                                          // on the edge from v1.
  repeat (n − 1 times) {                  // Add all n − 1 vertices to Y.
    min = ∞;
    for (i = 2; i <= n; i++)              // Check each vertex for
      if (0 ≤ length[i] < min) {          // having shortest path.
        min = length[i];
        vnear = i;
      }
    e = edge from vertex indexed by touch[vnear]
        to vertex indexed by vnear;
    add e to F;
    for (i = 2; i <= n; i++)
      if (length[vnear] + W[vnear][i] < length[i]) {
        length[i] = length[vnear] + W[vnear][i];
        touch[i] = vnear;                 // For each vertex not in Y,
      }                                   // update its shortest path.
    length[vnear] = −1;                   // Add vertex indexed by vnear
  }                                       // to Y.
}
```

از آنجائیکه فرض کردیم یک مسیر از V_1 به هر گره دیگر وجود دارد، لذا متغیر Vnear در هر تکرار از حلقه repeat، یک مقدار جدید به خود اختصاص می‌دهد که اگر اینچنین نبود، الگوریتم می‌بایست افزودن آخرین لبه را تا انجام n - ۱ تکرار حلقه repeat ادامه می‌داد.

الگوریتم ۳-۴ تنها لبه‌های کوتاهترین مسیرها را تعیین می‌کند و طول این مسیرها را تولید نمی‌نماید. این طولها می‌توانست از لبه‌ها بدست آید. با یک تغییر ساده در الگوریتم می‌توانستیم طولها را نیز محاسبه کرده و آنها را در یک آرایه ذخیره کنیم. کنترل در الگوریتم ۳-۴، عیناً مشابه کنترل در الگوریتم ۱-۴ است. لذا با تحلیل الگوریتم ۱-۴ می‌توان نتیجه زیر را برای الگوریتم ۳-۴ بدست آورد:

$$T(n) = ۲(n - ۱)^{۲} \in \Theta(n^{۲})$$

به راحتی می‌توان ثابت نمود که الگوریتم ۳-۴، همواره کوتاهترین مسیرها را تولید می‌کند. اثبات این مسئله با استفاده از استقراء، مشابه اثبات مسئله تولید درخت پوشای می‌نیمم توسط الگوریتم ۱-۴ (الگوریتم Prim) است.

همانند الگوریتم prim، الگوریتم Dijkstra نیز می‌تواند با استفاده از یک heap یا یک heap فیبوناچی، پیاده‌سازی شود. پیاده‌سازی heap در $\Theta(m \lg n)$ و پیاده سازی heap فیبوناچی در $\Theta(m + n \lg n)$ است که در آن m، تعداد لبه‌ها می‌باشد. برای توضیح بیشتر، به کتاب Tarjan و Fredman (۱۹۸۷) مراجعه کنید.

۴-۳ زمانبندی (Scheduling)

فرض کنید که یک آرایشگر چندین مشتری با تقاضاهای مختلف دارد (مثلاً کوتاه ساده، کوتاه با شامپو، سشوار). مدت زمان انجام تقاضاهای مختلف یکسان نیست ولی آرایشگر مدت زمان اجرای هر یک را به خوبی می‌داند. یک مسئله می‌تواند این باشد که مجموع زمان‌های انتظار و سرویس مشتریان را به حداقل برسانیم. مجموع زمانهای انتظار و سرویس‌دهی به **زمان در سیستم** موسوم است. مسئله به حداقل رساندن مجموع زمان در سیستم، کاربردهای زیادی دارد. برای مثال، شاید بخواهیم دستیابی کاربران به یک دیسک را زمانبندی کنیم تا مجموع زمان انتظار و سرویس‌دهی را به حداقل ممکن برسانیم.

مسئله دیگری که در بحث زمانبندی مطرح می‌شود هنگامی است که هر تقاضا به یک مدت زمان یکسانی برای اجرا نیازمند است، اما هر یک دارای مهلت بخصوصی برای اجرا می‌باشد. هدف از زمانبندی تقاضاها، به حداکثر رساندن بازدهی سیستم است. ما پس از بحث و بررسی مسئله زمانبندی ساده و به حداقل رساندن مجموع زمان در سیستم، به زمانبندی مهلت‌دار نیز خواهیم پرداخت.

۴-۳-۱ به حداقل رساندن مجموع زمان در سیستم

یک راه حل ساده برای به حداقل رساندن مجموع زمان در سیستم، درنظر گرفتن همه زمانبندیهای ممکن و انتخاب کمترین آنها است. این موضوع در مثال زیر به روشنی بیان شده است.

مثال ۲-۴ فرض کنید سه تقاضا وجود دارد که زمان سرویس آنها به صورت زیر داده شده است:

$$t_1 = 5, \qquad t_2 = 10, \qquad t_3 = 4$$

واحدهای حقیقی زمان در این مسئله اهمیتی ندارد. اگر ما آنها را به ترتیب ۱، ۲ و ۳ زمانبندی کنیم، آنگاه زمان سپری شده در سیستم برای سه تقاضا به صورت زیر مطرح می‌شود:

تقاضا (کار)	زمان در سیستم
۱	۵ (زمان سرویس)
۲	۵ (انتظار برای کار ۱) + ۱۰ (زمان سرویس)
۳	۵ (انتظار برای کار ۱) + ۱۰ (انتظار برای کار ۲) + ۴ (زمان سرویس)

مجموع زمان در سیستم برای این زمانبندی برابراست با:

$$\underbrace{5}_{\text{مدت زمان برای تقاضای ۱}} + \underbrace{(5+10)}_{\substack{\text{مدت زمان برای تقاضای ۲}}} + \underbrace{(5+10+4)}_{\text{مدت زمان برای تقاضای ۳}} = 39$$

با این روش محاسبه می‌توان لیست تمام زمانبندیهای ممکن و مجموع زمان در سیستم را پیدا کرد.

زمانبندی	مجموع زمان در سیستم
[۱، ۲، ۳]	$5 + (5 + 10) + (5 + 10 + 4) = 39$
[۱، ۳، ۲]	$5 + (5 + 4) + (5 + 4 + 10) = 33$
[۲، ۱، ۳]	$10 + (10 + 5) + (10 + 5 + 4) = 44$
[۲، ۳، ۱]	$10 + (10 + 4) + (10 + 4 + 5) = 43$
[۳، ۱، ۲]	$4 + (4 + 5) + (4 + 5 + 10) = 32$
[۳، ۲، ۱]	$4 + (4 + 10) + (4 + 10 + 5) = 37$

زمانبندی [۳، ۱، ۲] (با مجموع زمان در سیستم ۳۲)، بهترین جواب مسئله است.

پرواضح است که الگوریتمی که تمامی زمانبندیهای ممکن یک مسئله را درنظر می‌گیرد، یک الگوریتم زمان-فاکتوریل است. توجه دارید که در مثال فوق، زمانبندی بهینه در حالتی رخ داد که تقاضای ۳ با کمترین زمان سرویس (۴) در ابتدا، تقاضای ۱ با کمترین زمان سرویس (۵) در بین تقاضاهای باقیمانده بعد از آن و در نهایت تقاضای ۲ با بزرگترین زمان سرویس (۱۰) در انتها قرار گرفت. در ابتدابه نظر می‌رسد که چنین زمانبندی بهینه است زیرا تقاضاهای کوتاهتر را خارج از نوبت و در ابتدا انجام می‌دهد. یک الگوریتم حریص سطح بالا برای این روش به صورت زیر است.

```
sort the jobs by service time in decreasing order;
while(the instance is not solved){
  shedule the next job;                    // روال انتخاب و بررسی امکان سنجی
  if(there are no more jobs)               // بررسی جواب
    the instance is solved;
}
```

ما این الگوریتم را به فرم کلی روش حریص نوشتیم تا نشان دهیم که آن در واقع، یک الگوریتم حریص است. به هر حال واضح است که الگوریتم مرتب‌سازی تقاضاها، براساس زمان سرویس آنها کار می‌کند. پیچیدگی زمانی این الگوریتم برابراست با:

$$W(n) \in \Theta(n \lg n)$$

اگرچه در ابتدا، بنظر می‌رسد که زمانبندی ارائه شده توسط این الگوریتم بهینه باشد، ولی بایستی این مطلب ثابت شود. قضیه زیر ثابت می‌کند که این نحوهٔ زمانبندی، بهینه‌ترین حالت ممکن است.

قضیه ۳-۴ تنها زمانبندیی که مجموع زمان در سیستم را به حداقل می‌رساند آن است که تقاضاها را به ترتیب غیرنزولی برحسب زمان سرویس، زمانبندی می‌کند.

اثبات: برای $1 \le i \le n-1$، فرض می‌کنیم که t_i زمان سرویس تقاضای iام در یک زمانبندی بهینه باشد. لازم است نشان دهیم که زمانبندی براساس ترتیب غیرنزولی زمانهای سرویس انجام شده است. این کار را بااستفاده از برهان خلف انجام می‌دهیم. اگر آنها براساس ترتیب غیرنزولی زمانبندی نشده باشند، آنگاه برای حداقل یک i که $1 \le i \le n-1$، داریم $t_i \ge t_{i+1}$ ما می‌توانیم زمانبندی اولیه را با جابجایی تقاضاهای iام و i+۱ام دوباره مرتب کنیم. با این کار به میزان t_i واحد از زمانی که تقاضای i+۱ام در سیستم صرف می‌کند، کاسته‌ایم؛ زیرا سیستم منتظر سرویس دهی تقاضای iام در زمانبندی اولیه نمی‌ماند. بطور مشابه، به میزان t_{i+1} واحد به زمانی که تقاضای iام در زمانبندی اولیه صرف کند، افزوده‌ایم. واضح است که زمان مصرفی تقاضاهای دیگر، تغییری نمی‌کند. بنابراین، اگر T مجموع زمان در سیستم زمانبندی اولیه باشد و T' مجموع زمانبندی جدید باشد،

$$T' = T + t_{i+1} - t_i$$

از آنجائیکه $t_i > t_{i+1}$ است، لذا

$$T' < T$$

که با فرض بهینگی زمانبندی اولیه، متناقض است.

به راحتی می‌توانیم الگوریتم فوق را به مسئله زمانبندی با چند سرویس‌دهنده تعمیم دهیم. فرض کنید

m سرویس‌دهنده وجود دارد. سرویس‌دهنده‌ها را به دلخواه و تقاضاها را به ترتیب غیرنزولی زمانهای سرویس مرتب نمائید. فرض کنید اولین سرویس‌دهنده، تقاضای اول را پاسخ می‌دهد؛ دومین سرویس‌دهنده تقاضاهای دوم را پاسخ می‌دهد؛... و mامین سرویس‌دهنده تقاضای mام را پاسخ می‌دهد. اولین سرویس‌دهنده، زودتر از همه به کار خود خاتمه می‌دهد زیرا این سرویس‌دهنده، تقاضایی با کوچکترین زمان سرویس را سرویس‌دهی می‌کند و به همین دلیل، اولین سرویس‌دهنده تقاضای m+۱ام را سرویس می‌دهد. دومین سرویس‌دهنده، تقاضای m+۲ام را سرویس می‌دهد و الی آخر. فرم کلی آن بدین صورت است:

سرویس‌دهنده ۱، تقاضاهای ۱، (m + ۱)، (۲m + ۱)، (۳m + ۱)،... را سرویس می‌دهد.
سرویس‌دهنده ۲، تقاضاهای ۲، (m + ۲)، (۲m + ۲)، (۳m + ۲)،... را سرویس می‌دهد.
⋮
سرویس‌دهنده i، تقاضاهای i، (i + m)، (i + ۲m)، (i + ۳m)،... را سرویس می‌دهد.
⋮
سرویس‌دهنده m، تقاضاهای m، (m + m)، (m + ۲m)، (m + ۳m)،... را سرویس می‌دهد.

پرواضح است که تقاضاها به ترتیب زیر پردازش می‌شوند:

۱، ۲،...،m، ۱ + m، ۲ + m،...،m + m، ۱ + ۲m،...

یعنی تقاضاها براساس ترتیب غیرنزولی زمانهای سرویس، پردازش می‌شوند.

۲-۳-۴ زمانبندی مهلت‌دار

در این مسئله زمانبندی، هر تقاضا به یک واحد زمانی برای انجام کار و یک مهلت زمانی و یک بازده معین نیاز دارد. اگر تقاضایی قبل از مهلت زمانی خود و یا در خلال آن شروع شود، بازده مورد نظر بدست آمده است. هدف از زمانبندی تقاضاها، به حداکثر رساندن مجموع بازدهی است. هیچ لزومی به زمانبندی همه تقاضاها نیست. زمانبندیهایی که در آن تقاضاهایی بعد از مهلت زمانی‌شان وجود دارد را در نظر نمی‌گیریم. به چنین زمانبندیهایی، زمانبندی غیرممکن گوئیم. مثال زیر، این مسئله را نشان می‌دهد.

مثال ۳-۴ فرض کنید تقاضاها، مهلت‌ها و بازده‌های زیر در یک سیستم وجود دارند:

تقاضا	بازده	مهلت
۱	۳۰	۲
۲	۳۵	۱
۳	۲۵	۲
۴	۴۰	۱

وقتی می‌گوئیم کار (تقاضای) ۱، دارای مهلت زمانی ۲ است یعنی اینکه این تقاضا می‌تواند در زمان ۱ یا زمان ۲ شروع شود. توجه داریم که زمان صفر وجود ندارد. از آنجائیکه تقاضای ۲ دارای مهلت زمانی ۱ است، لذا می‌تواند فقط در زمان ۱ اجرا شود. زمانبندیهای ممکن و مجموع بازده آنها به صورت زیر می‌باشند:

زمانبندی	مجموع بازده
[۱، ۳]	۳۰ + ۲۵ = ۵۵
[۲، ۱]	۳۵ + ۳۰ = ۶۵
[۲، ۳]	۳۵ + ۲۵ = ۶۰
[۳، ۱]	۲۵ + ۳۰ = ۵۵
[۴، ۱]	۴۰ + ۳۰ = ۷۰
[۴، ۳]	۴۰ + ۲۵ = ۶۵

زمانبندیهای غیرممکن را در لیست فوق ذکر نکردیم. مثلاً زمانبندی [۲ ،۱] غیرممکن است، زیرا تقاضای ۱ در زمان ۱ آغاز شده و به یک واحد زمانی نیاز دارد تا به طور کامل سرویس‌دهی شود و این موجب می‌شود که تقاضای ۲ در زمان ۲ شروع شود. اما همانطوریکه مشاهده می‌کنید، آخرین مهلت تقاضای ۲، زمان ۱ است. برای مثال، زمانبندی [۳ ،۱] امکان‌پذیر است زیرا تقاضای ۱ قبل از مهلت زمانی خود شروع شده و تقاضای ۳ در مهلت زمانی خود آغاز می‌شود. مشاهده می‌کنید که زمانبندی [۱ ،۴] با مجموع بازدهی ۷۰، زمانبندی بهینه است.

در نظر گرفتن همه زمانبندیها، آنچنانکه در مثال ۳-۴ انجام شد، یک زمان فاکتوریلی صرف می‌کند. توجه کنید که در مثال فوق، تقاضایی که بیشترین بازدهی را دارد (تقاضای ۴) در زمانبندی بهینه منظور شده است، اما تقاضایی که دومین مقدار بازدهی را دارد، نتوانسته است در زمانبندی حضور یابد زیرا هر دو دارای مهلت زمانی ۱ می‌باشند؛ لذا واضح است که هر دو نمی‌توانند در زمانبندی شرکت کنند و البته آن تقاضایی که بازده بیشتری دارد زمانبندی می‌شود. تقاضای دیگری که در زمانبندی وجود دارد، تقاضای ۱ است زیرا بازده آن بیشتر از بازده تقاضای ۳ است. این مثال، نمونه‌ای از بکارگیری روش حریص را به ما نشان می‌دهد و آن اینکه برای حل مسئله، ابتدا باید تقاضاها را براساس ترتیب غیرنزولی مقادیر بازده مرتب کنیم، سپس هر یک از تقاضاها را به ترتیب بررسی کرده و در صورت امکان آن را به زمانبندی اضافه نمائیم. قبل از ارائهٔ حتی یک الگوریتم سطح بالا برای این روش، به چند تعریف نیازمندیم. **رشته (دنباله) ممکن**، رشته‌ای است که همهٔ تقاضاها در آن به ترتیب و با توجه به مهلتشان شروع می‌شوند. در مثال ۳-۴، [۴ ،۱] یک رشته ممکن و [۱ ،۴] یک رشته غیرممکن است. یک مجموعه از تقاضاها را **مجموعهٔ ممکن** گوئیم اگر حداقل یک رشتهٔ ممکن برای تقاضاها در این مجموعه وجود داشته باشد. در مثال ۳-۴، {۴ ،۱}، یک مجموعهٔ ممکن است زیرا زمانبندی رشته [۴ ،۱] امکان‌پذیر است در حالیکه {۲ ،۴}،

یک مجموعهٔ ممکن نیست زیرا امکان زمانبندی هیچ یک از [۲،۴] و [۴،۲] وجود ندارد. هدف، یافتن یک رشتهٔ ممکن با حداکثر مجموع بازدهی است. ما چنین رشته‌ای را **رشته بهینه** نامیده و مجموعه تقاضاهای این رشته را یک **مجموعهٔ تقاضای بهینه** می‌گوئیم. ما می‌توانیم یک الگوریتم حریص سطح بالا برای مسئله زمانبندی مهلت‌دار ارائه دهیم:

```
sort the jobs in nonincreasing order by profit;
S = ∅
while (the instance is not solved){
  select next job;                          // روال انتخاب
  if (S is feasible with this job added)    // بررسی امکان سنجی
    add this job to S;
  if (there are no more jobs)               // بررسی جواب
    the instance is solved;
}
```

مثال ۴-۴ فرض کنید تقاضاها، آخرین مهلت و بازده‌های زیر در یک سیستم وجود دارند:

تقاضا	بازده	مهلت
۱	۴۰	۳
۲	۳۵	۱
۳	۳۰	۱
۴	۲۵	۳
۵	۲۰	۱
۶	۱۵	۳
۷	۱۰	۲

ما تقاضاها را قبل از شماره گذاری، براساس مقدار بازده‌شان مرتب کرده‌ایم. الگوریتم حریص فوق به صورت زیر عمل می‌کند:

۱- مجموعهٔ S در ابتدا ∅ است.

۲- مجموعهٔ S شامل {۱} می‌شود زیرا رشتهٔ [۱] ممکن است.

۳- مجموعهٔ S شامل {۱،۲} می‌شود زیرا رشتهٔ [۲،۱]، یک رشتهٔ ممکن است.

۴- مجموعهٔ {۱،۲،۳} قابل قبول نیست زیرا هیچ رشتهٔ ممکنی در این مجموعه وجود ندارد.

۵- مجموعهٔ S شامل {۱،۲،۴} می‌شود زیرا رشتهٔ [۲،۱،۴]، یک رشتهٔ ممکن است.

۶- مجموعهٔ {۱،۲،۴،۵} قابل قبول نیست زیرا هیچ رشتهٔ ممکنی در این مجموعه وجود ندارد.

۷- مجموعهٔ {۱،۲،۴،۶} قابل قبول نیست زیرا هیچ رشتهٔ ممکنی در این مجموعه وجود ندارد.

۸- مجموعهٔ {۷، ۴، ۲، ۱} قابل قبول نیست زیرا هیچ رشتهٔ ممکنی در این مجموعه وجود ندارد.
مقدار نهایی S برابر {۴، ۲، ۱} و یک رشتهٔ ممکن برای این مجموعه، [۴، ۱، ۲] است. از آنجائیکه تقاضاهای ۱ و ۴ هر دو مهلتی برابر ۳ دارند، لذا می‌توانستیم [۴، ۱، ۲] را نیز به عنوان یک رشتهٔ ممکن درنظر بگیریم.

قبل از اینکه ثابت کنیم این الگوریتم همیشه یک رشتهٔ بهینه تولید می‌کند، می‌خواهیم یک نسخهٔ صوری از آن بنویسیم. برای انجام این کار، به یک روش کارا برای تعیین اینکه آیا یک مجموعه، ممکن است یا خیر، نیاز داریم. این کار را نمی‌توان بادرنظر گرفتن همه رشته‌های ممکن انجام داد زیرا همانطوریکه گفتیم، چنین الگوریتمی از نوع زمان - فاکتوریل خواهد بود. پیش قضیه زیر، روش کاراتری را برای بررسی ممکن بودن یک مجموعه ارائه می‌دهد.

پیش‌قضیه ۳-۴ فرض می‌کنیم که S مجموعه‌ای از تقاضاها است. آنگاه S یک مجموعه ممکن است اگر و تنها اگر رشتهٔ بدست آمده از مرتب سازی تقاضاهای S به صورت غیرنزولی و براساس مهلت تقاضا، یک رشتهٔ ممکن باشد.

اثبات: فرض کنید که S یک مجموعهٔ ممکن باشد، آنگاه حداقل یک رشتهٔ ممکن برای تقاضاهای S وجود دارد. همچنین فرض کنید که در این رشته تقاضای x قبل از تقاضای y زمانبندی شده و مهلت زمانی تقاضای y کوچکتر (زودتر) از تقاضای x باشد. اگر این دو تقاضا را در رشته جابجا کنیم، تقاضای y با توجه به موقعیت مکانی‌اش در رشته، زودتر شروع می‌شود. از آنجائی که مهلت تقاضای x بزرگتر از مهلت تقاضای y است و زمان جدید داده شده به تقاضای x برای y کافی است، لذا تقاضای x بزرگتر از مهلت تقاضای y است و زمان جدید داده شده به تقاضای x برای y کافی است، لذا تقاضای x نیز در مهلت مقررش شروع می‌شود. لذا رشتهٔ جدید هنوز هم ممکن خواهد بود. به هنگام مرتب سازی تبادلی(الگوریتم ۳-۱) بر روی رشتهٔ ممکن اولیه، می‌توان با استفاده مکرر از این واقعیت ثابت نمود که رشتهٔ مرتب شده، یک رشتهٔ ممکن است. به بیان دیگر، S یک مجموعهٔ است اگر رشتهٔ مرتب شده ممکن باشد.

مثال ۵-۴ تقاضاهای مثال ۴-۴ را درنظر بگیرید. برای تعیین اینکه آیا {۷، ۴، ۲، ۱} یک مجموعهٔ ممکن است یا خیر، پیش قضیه ۳-۴ بیان می‌کند که لازم است فقط امکان سنجی رشتهٔ زیر بررسی شود.

[۲، ۷، ۱، ۴]
↑ ↑ ↑ ↑
۱ ۲ ۳ ۳

مهلت هر تقاضا را در زیر آن نوشته‌ایم. از آنجائیکه تقاضای ۴ با توجه به مهلت زمانی‌اش زمانبندی نشده است، لذا رشتهٔ مورد نظر یک رشتهٔ ممکن نیست و در نتیجه با استفاده از پیش قضیه ۳-۴، مجموعه نیز ممکن نخواهد بود.

الگوریتم زمانبندی مهلت‌دار در زیر آمده است. در این الگوریتم فرض شده است که تقاضاها از قبل، براساس مقدار بازده‌شان به صورت غیرنزولی مرتب شده‌اند. از آنجائیکه بازده تقاضا تنها برای مرتب‌سازی تقاضاها بکار می‌آید، لذا تقاضاها را قبل از ورود به الگوریتم، مرتب شده فرض نمودیم. یعنی اینکه آن را به عنوان پارامتر الگوریتم درنظر نمی‌گیریم.

الگوریتم ۴-۴ زمانبندی مهلت‌دار

مسئله: یک زمانبندی با حداکثر مجموع بازدهی را برای تقاضاهایی مشخص کنید که در آن هر تقاضا بازده معینی دارد و در صورتی این بازده درنظر گرفته می‌شود که تقاضا، در مهلت مقرر یا قبل از آن شروع شده باشد.

ورودی: تعداد تقاضاها n، آرایه‌ای از اعداد صحیح deadline که از ۱ تا n شاخص‌دهی شده است بطوری که deadline[i]، آخرین مهلت زمانی تقاضای iام می‌باشد. آرایه به صورت غیرنزولی براساس بازدهٔ تقاضاها مرتب شده است.

خروجی: یک رشتهٔ بهینه J برای تقاضاها.

```
void schedule (int n,
               const int deadline[ ],
               sequence_of_integer& j)
{
  index i;
  sequence_of_integer K;
  J = [1];
  for (i = 2; i <= n; i++){
     K = J with i added according to nondecreasing vlaues of deadline [i];
     if (K is feasible)
        J = K;
  }
}
```

قبل از تحلیل الگوریتم، یک مثال از آن را ارائه می‌دهیم.

مثال ۶-۴ تقاضاهای مثال ۲-۴ را درنظر می‌گیریم.

تقاضا	مهلت
۱	۳
۲	۱
۳	۱
۴	۳
۵	۱
۶	۳
۷	۲

الگوریتم ۴-۴ روی این تقاضاها به صورت زیر عمل می‌کند:

۱- J با [۱] مقداردهی می‌شود.

۲- K با [۲، ۱] مقداردهی می‌شود و امکان‌پذیری آن محرز می‌گردد.

J با [۱] مقداردهی می‌شود زیرا k امکان‌پذیر است.

۳- K با [۳، ۲، ۱] مقداردهی می‌شود و رد می‌گردد زیرا امکان‌پذیر نیست.

۴- K با [۲، ۱، ۴] مقداردهی می‌شود و امکان‌پذیری آن محرز می‌گردد.

J با [۲، ۱، ۴] مقداردهی می‌شود زیرا k امکان‌پذیر است.

۵- K با [۲، ۵، ۱، ۴] مقداردهی می‌شود و رد می‌گردد زیر امکان‌پذیر نیست.

۶- K با [۲، ۱، ۶، ۴] مقداردهی می‌شود و رد می‌گردد زیر امکان‌پذیر نیست.

۷- K با [۲، ۷، ۱، ۴] مقداردهی می‌شود و رد می‌گردد زیر امکان‌پذیر نیست.

مقدار نهایی J برابر [۲، ۱، ۴] است.

تحلیل پیچیدگی زمانی بدترین حالت الگوریتم ۴-۴ (زمانبندی مهلت‌دار)

عمل مبنایی: لازم است مقایساتی برای مرتب سازی تقاضاها صورت بگیرد. همچنین به مقایسات بیشتری برای وقتی که K (که تقاضای iام به آن اضافه شده) را مساوی J قرار می‌دهیم و برای بررسی امکان‌پذیری K، نیاز داریم. بنابراین، یک دستورالعمل مقایسه را به عنوان عمل مبنایی در نظر می‌گیریم.

اندازه ورودی: n، تعداد تقاضاها.

مرتب سازی تقاضاها به زمانی برابر $\Theta(n \lg n)$ نیاز دارد. لذا در هر تکرار از حلقه **for_i** لازم است که حداکثر i - ۱ مقایسه انجام شود تا تقاضای iام به K افزوده شود؛ همچنین به حداکثر i مقایسه نیازمندیم تا امکان‌پذیری K را بررسی کنیم. بنابراین، بدترین حالت برابراست با:

$$\sum_{i=2}^{n} [(i-1)+i] = n^2 - 1 \in \Theta(n^2)$$

اولین تساوی، از مثال ۱ - A در ضمیمه A بدست آمده است. از آنجائیکه این زمان، بر زمان مرتب سازی برتری دارد، لذا

$$W(n) \in \Theta(n^2)$$

در اینجا ثابت می‌کنیم که الگوریتم همواره یک جواب بهینه ارائه می‌نماید.

قضیه ۴-۴ الگوریتم ۴-۴ همواره یک مجموعه بهینه از تقاضاها را تولید می‌کند.

اثبات: اثبات را با استفاده از استقراء روی تعداد تقاضاها (n) انجام می‌دهیم.

پایه استقراء: واضح است که اگر یک تقاضا وجود داشته باشد، قضیه درست است.

فرض استقراء: فرض می‌کنیم که مجموع تقاضاهای بدست آمده از اعمال الگوریتم روی n تقاضای اول، یک مجموعهٔ بهینه باشد.

گام استقراء: لازم است نشان دهیم که مجموعهٔ تقاضاهای بدست آمده از اعمال الگوریتم روی n+۱ تقاضای اول، یک مجموعهٔ بهینه است. برای این منظور فرض می‌کنیم که A، مجموعهٔ تقاضاهای بدست آمده از ۱ + n تقاضای اول و B، یک مجموعهٔ بهینه از تقاضاهای بدست آمده از n+۱ تقاضای اول باشد. دو حالت وجود دارد:

حالت ۱: B شامل تقاضای n+۱ نیست.

در این حالت، B مجموعه‌ای از تقاضاهای بدست آمده از n تقاضای اول است. به هر حال، با توجه به فرضیه استقراء، A شامل یک مجموعه از تقاضاهای بهینه بدست آمده از n تقاضای اول می‌باشد. بنابراین، مجموع بازدهی تقاضای B نمی‌تواند بزرگتر از مجموع بازدهی تقاضاهای A باشد و در نتیجه A یک مجموعهٔ بهینه است.

حالت ۲: B شامل تقاضای n+۱ است.

فرض کنید که A شامل تقاضای n+۱ باشد، آنگاه

$$B = B' \cup \{job\ (n+1)\}\ ,\quad A = A' \cup \{job\ (n+1)\}$$

که در آن B' یک مجموعهٔ بدست آمده از n تقاضای اول و A' مجموعهٔ بدست آمده از اعمال الگوریتم روی n تقاضای اول است. با استفاده از فرض استقراء نتیجه می‌گیریم که A'، یک مجموعهٔ بهینه برای n تقاضای اول است. لذا

$$\begin{aligned} profit\ (B) &= profit\ (B') + profit\ (n+1) \\ &\le profit\ (A') + profit\ (n+1) = profit\ (A) \end{aligned}$$

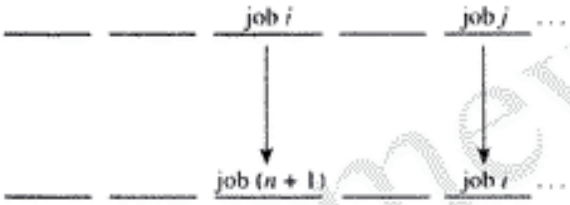
که در آن $profit\ (n+1)$، بازده تقاضای n+۱ و $profit\ (A)$، مجموع بازده تقاضاها در A است. تا زمانیکه B برای n+۱ تقاضای اول بهینه است، می‌توانیم نتیجه بگیریم که A نیز بهینه است.

فرض کنید A شامل تقاضای n+۱ نیست. مدت زمان اشغال شده توسط تقاضای n+۱ در B را در نظر بگیرید. اگر هنگامی که الگوریتم، تقاضای n+۱ را درنظر می‌گرفت، این مدت زمانی قابل دسترسی بود، دراینصورت این تقاضا زمانبندی می‌شد. لذا این مدت زمانی بایستی به تقاضایی در A که آن را تقاضای i می‌نامیم، داده شود. از آنجائیکه تقاضاها به صورت غیرنزولی و براساس مقادیر بازده‌شان مرتب می‌شوند، لذا بازده تقاضای i حداقل به بزرگی بازده تقاضای n+۱ است.

اگر تقاضای i در B نباشد، می‌توانیم تقاضای n+۱ را با تقاضای i در B جایگزین کنیم. نتیجه،

مجموعه‌ای از تقاضاها است که مجموع بازدهی آنها حداقل به بزرگی مجموع بازده تقاضاها در B است. به هر حال، این مجموعه از n تقاضای اول بدست آمده است. لذا، با استفاده از فرض استقراء درمی‌یابیم که مجموع بازدهی آن می‌تواند بزرگتر از مجموع بازدهی تقاضاهای A نباشد؛ یعنی A بهینه است.

اگر تقاضای i در B باشد، آنگاه هر مدت زمانی که در B اشغال می‌کند بایستی توسط تقاضایی در A (که آن را تقاضای j می‌نامیم) اشغال شود. [در غیراینصورت، الگوریتم ما می‌توانست تقاضای i را در آن مهلت زمانی قرار دهد و تقاضای n+۱ را زمانبندی کند.] همانطوریکه در شکل ۹-۴ نشان می‌دهیم، در B می‌توانیم تقاضای n+۱ را با تقاضای j و تقاضای i را با تقاضای j جایگزین کنیم. از آنجائیکه بازدهٔ تقاضای j حداقل به اندازهٔ بازده تقاضای n+۱ است، لذا حاصل مجموعه‌ای از تقاضاهاست که مجموع بازده آن حداقل به بزرگی مجموع بازده تقاضاها در B می‌باشد. کار را همانند گذشته ادامه می‌دهیم تا نتیجه بگیریم که A بهینه است.



شکل ۹-۴ اگر در B، تقاضای n + ۱ را با تقاضای i و تقاضای i را با تقاضای j جایگزین کنیم، بازده کل حداقل به بزرگی نمونهٔ قبلی می‌شود.

با استفاده از ساختار داده‌ای مجموعهٔ غیرالحاقی III که در ضمیمه C ارائه شده است، می‌توانیم یک نسخهٔ $\Theta(n \lg n)$ از روال schedule (در الگوریتم ۴-۴) ارائه کنیم که m می‌نیمم n و بزرگترین مهلت زمانی برای n تقاضا است. از آنجائیکه زمان مرتب سازی در $\Theta(n \lg n)$ است، کل الگوریتم نیز در $\Theta(n \lg n)$ خواهد بود. این بهینه سازی را در تمرینات بررسی خواهیم کرد.

۴-۴ روش حریص در مقایسه با برنامه‌نویسی پویا: مسئله کوله‌پشتی

روش حریص و برنامه‌نویسی پویا، هر دو روشهایی برای حل مسائل بهینه سازی هستند. اغلب، یک مسئله می‌تواند با استفاده از هر دو روش حل شود. به عنوان مثال، مسئله کوتاهترین مسیرهای تک-مبدأیی با استفاده از برنامه‌نویسی پویا در الگوریتم ۳-۳ و با استفاده از روش حریص در الگوریتم ۳-۴ حل شده است. به هر حال، الگوریتم برنامه‌نویسی پویا دارای توانایی اضافی در

کوتاهترین مسیرها از همه مبدأها است و نمی‌توان آن الگوریتم را طوری تغییر داد که تنها کوتاهترین مسیرها را از یک مبدأ پیدا کند و در نتیجه کارایی آن بالا رود زیرا کل آرایه D مورد نیاز است. بنابراین، روش برنامه‌نویسی پویا از یک الگوریتم $\Theta(n^3)$ برای حل مسائل استفاده می‌کند در حالیکه روش حریص، یک روش $\Theta(n^2)$ را بکار می‌گیرد. اغلب، هنگامی که روش حریص مسئله را حل می‌کند، نتیجه، یک الگوریتم ساده‌تر و کاراتر است.

از طرف دیگر، معمولاً تعیین اینکه آیا یک روش حریص همواره یک جواب بهینه تولید می‌کند یا نه، کار مشکل‌تری است. نظیر مسئله پول خرد که نشان می‌دهد همهٔ الگوریتم‌های حریص، جواب بهینه تولید نمی‌کنند. بنابراین، باید ثابت کنیم که یک روش حریص خاص همواره یک جواب بهینه تولید می‌نماید و برای اینکه نشان دهیم این روش، ممکن است چنین جوابی راتولید نکند، کافیست از یک مثال نقض استفاده کنیم. به خاطر دارید که در روش برنامه‌نویسی پویا، تنها لازم است تعیین کنیم که آیا اصل بهینگی در آن بکار رفته است یا خیر؟

برای آنکه بیشتر به اختلاف این دو روش پی ببریم، دو مسئله خیلی شبیه به هم (مسئله کوله‌پشتی ۱-۰ و مسئله کوله‌پشتی جزئی) را ارائه می‌دهیم. ما یک الگوریتم حریص ارائه خواهیم کرد که مسئله کوله‌پشتی ۱-۰ را با موفقیت حل می‌کند اما در مسئله کوله پشتی جزئی با شکست مواجه می‌شود. آنگاه مسئله کوله‌پشتی ۱-۰ را با استفاده از روش برنامه‌نویسی پویا، با موفقیت حل می‌کنیم.

۱-۴-۴ یک روش حریص برای مسئله کوله‌پشتی ۱-۰

یک مثال از این مسئله، دزدی از جواهر فروشی و حمل کالاها به وسیله کوله‌پشتی است. اگر مجموع وزن اشیاء مسروقه بیش از حداکثر وزن W باشد، کوله‌پشتی از هم باز خواهد شد. هر کالا، وزن و ارزش معینی دارد. مشکل دزد این است که مجموع ارزش کالاها را به حداکثر برساند؛ در عین حال که وزن آنها بیش از حد مجاز W نباشد. این مسئله به مسئله کوله‌پشتی ۱-۰ مشهور است که می‌توانیم آن را به صورت زیر در آوریم:

فرض کنید n کالا وجود داشته باشد و

$S = \{item_1, item_2, ...item_n\}$

w_i = وزن کالای i

p_i = وزن کالای i

W = حداکثر وزنی که کوله‌پشتی تحمل می‌کند

که در آن w_i و p_i و W اعداد صحیح و مثبت هستند. یک زیرمجموعهٔ A از S بدست آورید بطوری که

$$\sum_{item_i \in A} w_i \le W \quad \text{حداکثر شود تا اینکه} \quad \sum_{item_i \in A} p_i$$

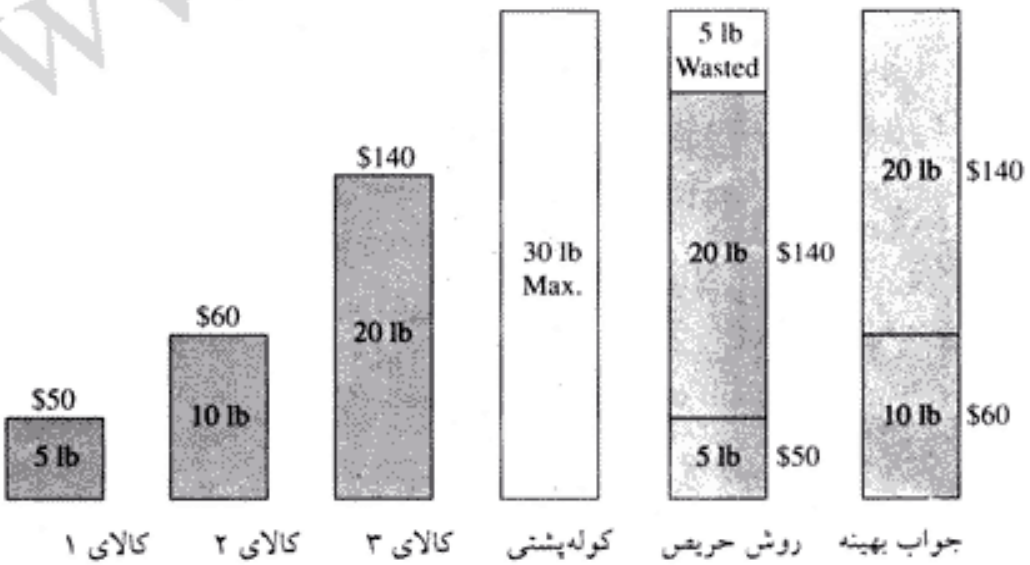
از آنجائیکه brute-force، تمامی زیرمجموعه‌های n کالایی را درنظر می‌گیرد؛ بجز آنهایی که وزنشان

بیش از W وزن است؛ لذا از بین کالاها، آن کالایی را می‌گیرد که بیشترین ارزش را دارد. از آنجائیکه در مثال ۱۰-A در ضمیمه A نشان دادیم که برای هر مجموعهٔ n عنصری، 2^n زیرمجموعه وجود دارد، بنابراین، الگوریتم brute-force یک الگوریتم زمان-نمایی است.

بدیهی است که استراتژی حریص این است که ابتدا کالاهای با ارزش‌تر دزدیده شوند. یعنی اینکه براساس ترتیب غیرنزولی ارزششان دزدیده شوند. به هر حال، اگر با ارزش‌ترین کالا، وزن بیشتری در مقابل ارزشش داشته باشد، دراینصورت، این استراتژی چندان مناسب نخواهد بود. به عنوان مثال، فرض کنید که ما سه کالا داریم که وزن اولی ۲۵ پوند و ارزش آن ۱۰ دلار است و وزن کالاهای دوم و سوم، هر یک ۱۰ پوند و ارزش هر کدام از آنها ۹ دلار است. اگر ظرفیت کوله‌پشتی (W) برابر ۳۰ پوند باشد، این استراتژی حریص تنها ۱۰ دلار سود می‌برد؛ در حالیکه جواب بهینه، ۱۸ دلار است.

استراتژی حریص دیگری که وجود دارد این است که ابتدا سبک‌ترین کالاها دزدیده شوند. اما این استراتژی نیز در صورتی که عناصر سبک‌تر، ارزش کمی در مقایسه با وزنشان داشته باشند با شکست مواجه می‌شود. برای اجتناب از شکست در دو الگوریتم حریص قبلی، یک الگوریتم حریص هوشمند را پیشنهاد می‌کنیم و آن اینکه در ابتدا، کالاهای با ارزش‌تر در واحد وزن دزدیده شوند. بدینصورت که دزد کالاها را براساس ارزش در واحد وزن به صورت غیرنزولی مرتب نموده، سپس آنها را به ترتیب انتخاب نماید. یک کالا در کوله‌پشتی قرار داده می‌شود اگر وزن کالای انتخابی موجب نشود که مجموع وزن کوله‌پشتی بیش از W گردد. این روش را در شکل ۱۰-۴ نشان می‌دهیم. در این شکل، ارزش و وزن هر کالا روی آن نوشته شده و حداکثر وزن کوله‌پشتی(W)، ۳۰ پوند می‌باشد. ارزش هر کالا در واحد وزن به قرار زیر است:

$$item_1: \frac{\$50}{5} = \$10 \quad , item_2 = \frac{\$60}{10} = \$6 \quad , item_3: \frac{\$140}{20} = \$7$$



شکل ۱۰-۴ یک مسئله حریص و یک جواب بهینه برای مسئله کوله‌پشتی ۱-۰.

که با مرتب سازی آنها براساس ارزش در واحد وزن به ترتیب از چپ به راست داریم:

$$item_1, item_3, item_2$$

همانگونه که در شکل می‌بینیم، این روش حریص، کالای ۱ و کالای ۳ را با مجموع ارزش ۱۹۰ دلار انتخاب می‌کند؛ در حالیکه جواب بهینه، انتخاب کالای ۲ و کالای ۳ است که ارزشی معادل ۲۰۰ دلار دارد؛ چراکه با انتخاب کالای ۱ و ۳، به اندازه ۵ پوند جای خالی در کوله‌پشتی هدر می‌رود، چون وزن کالای ۲ برابر ۱۰ پوند است و در کوله‌پشتی جای نمی‌گیرد. حتی این روش حریص هوشمندانه هم نتوانست مسئله کوله‌پشتی ۱-۰ را به خوبی حل کند.

۲-۴-۴ یک روش حریص برای مسئله کوله‌پشتی جزئی

در مسئله کوله‌پشتی جزئی، دزد مجبور نیست که کل یک کالا را در بردارد؛ بلکه او می‌تواند جزئی از آن را انتخاب کند. در مثال جواهرفروشی، کالاهای مسئله کوله‌پشتی ۱-۰ به صورت شمش طلا یا نقره و یا جواهرات کامل می‌باشند؛ در حالیکه کالاهای مسئله کوله‌پشتی می‌تواند به صورت ذرات طلا یا نقره و یا سنگهای قیمتی روی جواهرات نیز باشد. فرض کنید که کالاهای شکل ۱۰-۴ موجود باشد. اگر استراتژی حریص همچنان مبتنی بر انتخاب کالای با ارزش‌تر در واحد وزن باشد، پس از انتخاب تمام کالای ۱ و تمام کالای ۳ می‌توانیم ۵ پوند از کالای ۲ را نیز انتخاب کنیم که هم ظرفیت کوله‌پشتی تکمیل شود و هم به میزان ۵/۱۰ از کالای ۲ را برداریم. مجموع ارزش کالاهای سرقت شده برابراست با:

$$\$۵۰ + \$۱۴۰ + \frac{۵}{۱۰}(\$۶۰) = \$۲۲۰$$

الگوریتم حریص ما در مسئله کوله‌پشتی جزئی، هرگز جائی از کوله‌پشتی را خالی نمی‌گذارد. در نتیجه، همیشه یک جواب بهینه خواهیم داشت. در تمرینات از شما می‌خواهیم که این مطلب را ثابت کنید.

۳-۴-۴ یک روش برنامه‌نویسی پویا برای مسئله کوله‌پشتی ۱-۰

اگر بتوانیم نشان دهیم که در اینجا اصل بهینگی بکار رفته است، می‌توانیم مسئله کوله‌پشتی ۱-۰ را با استفاده از برنامه‌نویسی پویا نیز حل کنیم. برای این منظور، فرض می‌کنیم که A یک زیرمجموعه بهینه از یک مجموعهٔ n کالایی است. دو حالت وجود دارد: یا A شامل کالای n است و یا اینکه کالای n در A موجود نیست. اگر A شامل کالای n نباشد، A معادل است با یک زیرمجموعه بهینه از n-۱ کالای اول، و اگر A شامل کالای n باشد، مجموع ارزش کالاهای مجموعه A برابراست با P_n به اضافهٔ ارزش بهینهٔ بدست آمده از زمانی که کالاها از n-۱ کالای اول، در شرایطی انتخاب شوند که مجموع وزن بیش از $W - w_n$ نشود. بنابراین، اصل بهینگی بکار رفته است. این نتیجه می‌تواند به صورت زیر تعمیم یابد. اگر برای $w>۰, i>۰$، $p[i][w]$، ارزش بهینه از زمانی بدست آمده باشد که کالاها تنها از i کالای اول انتخاب می‌شوند، با این شرط که مجموع وزن بیش از w نباشد. آنگاه

$$p[i][w] = \begin{cases} maximum(p[i-1][w], p_i + p[i-1][w-w_i]) & w_i \le w \\ p[i-1][w] & w_i > w \end{cases}$$

حداکثر ارزش کالاها برابر با p[n][w] است که ما می‌توانیم با استفاده از آرایهٔ دوبعدی p، که سطرهای آن از صفر تا n و ستونهایش از صفر تا w شاخص‌دهی شده است، این ارزش را مشخص کنیم. مقدار سطرهای آرایه را به ترتیب، با استفاده از عبارت قبلی برای p[i][w] محاسبه می‌کنیم. مقادیر p[۰][w] و p[i][۰] برابر صفر هستند. در تمرینات از شما می‌خواهیم که الگوریتم آنرا بنویسید. روشن است که تعداد ورودیهای محاسبه شده آرایه برابراست با:

$$n\,W \in \Theta(n\,W)$$

۴-۴-۴ تصفیه الگوریتم برنامه‌نویسی پویا برای مسئله کوله‌پشتی ۱-۰

این واقعیت که عبارت قبلی برای تعداد ورودیهای محاسبه شدهٔ آرایه، روی n خطی است، می‌تواند ما را به اشتباه بیندازد که الگوریتم برای تمام نمونه‌های n عنصری کارا است؛ در حالیکه چنین نیست. عنصر دیگری در عبارت، به نام W، وجود دارد که هیچ ارتباطی بین آن و n برقرار نیست. بنابراین، برای یک n معین می‌توانیم مقادیر بزرگ دلخواه W را در نظر بگیریم تا نمونه‌های بزرگی ایجاد کنیم. برای مثال، اگر W برابر !n باشد، آنگاه تعداد ورودیهای محاسبه شده، در $\Theta(n \times n!)$ خواهد بود. اگر n = ۲۰ و W = ۲۰! باشد، آنگاه الگوریتم به هزاران سال وقت نیاز دارد تا بر روی یک کامپیوتر مدرن امروزی اجرا شود. هنگامی که W در مقایسه با n بسیار بزرگ می‌شود، این الگوریتم بدتر از الگوریتم brufe-force است که تمامی حالات را درنظر می‌گیرد.

این الگوریتم می‌تواند به صورتی اصلاح شود که بدترین حالت تعداد ورودیهای محاسبه شده در $\Theta(n^{۲})$ باشد. با این اصلاح، هرگز بدتر از الگوریتم brufe-force عمل نمی‌کند و در واقع، خیلی بهتر هم عمل خواهد کرد. در این اصلاح، از این حقیقت استفاده می‌کنیم که لازم نیست ورودیهای سطر iام را برای هر w بین ۱ و W مشخص کنیم؛ بلکه در سطر nام تنها به تعیین p[n][W] نیاز داریم. بنابراین، تنها ورودیهای مورد نیاز در سطر n-۱، ورودیهایی هستند که برای محاسبه p[n][W] به آنها نیاز داریم. از آنجائیکه

$$p[n][w] = \begin{cases} maximum(p[n-۱][w], p_n + p[n-۱][w-w_n]) & w_n \le w \\ p[n-۱][w] & w_n > w \end{cases}$$

لذا تنها ورودیهای مورد نیاز در سطر n-۱ام عبارتند از

$$p[n-۱][W] \quad و \quad p[n-۱][W-w_n]$$

این کار را از n تا ۱ ادامه می‌دهیم تا اینکه تمامی ورودیهای مورد نیاز را مشخص کنیم. یعنی بعد از اینکه ورودیهای لازم در سطر iام را مشخص نمودیم، ورودیهای مورد نیاز در سطر i-۱ام را با استفاده از این حقیقت که p[i][w] از p[i-۱][w] و p[i-۱][w-w_i] محاسبه می‌شود، تعیین می‌کنیم. این کار زمانی

سطر شروع می‌کنیم. مثال زیر این روش را نشان می‌دهد.

مثال ۷-۴ فرض کنید W = ۳۰ و کالاهای شکل ۱۰-۴ موجود است. ورودیهای مورد نیاز هر سطر را تعیین می‌کنیم.

تعیین ورودیهای سطر ۳:

ما نیاز داریم به $p[۳][W] = p[۳][۳۰]$

تعیین ورودیهای سطر ۲:

برای محاسبه P[۳][۳۰] نیاز داریم به

$$p[۳-۱][۳۰] = p[۲][۳۰] \quad و \quad p[۳-۱][۳۰-w_۳] = p[۲][۱۰]$$

تعیین وردیهای سطر ۱:

برای محاسبه P[۲][۳۰] نیاز داریم به

$$p[۲-۱][۳۰] = p[۱][۳۰] \quad و \quad p[۲-۱][۳۰-w_۲] = p[۱][۲۰]$$

برای محاسبه P[۲][۱۰] نیاز داریم به

$$p[۲-۱][۱۰] = p[۱][۱۰] \quad و \quad p[۲-۱][۱۰-w_۲] = p[۱][۰]$$

در ادامه، محاسبات را انجام می‌دهیم.

محاسبه سطر ۱:

$$p[۱][w] = \begin{cases} maximum(p[۰][W],\ \$۵۰ + p[۰][W-۵]) & if\ \ w_۱ = ۵ \le w \\ p[۰][w] & if\ \ w_۱ = ۵ > w \end{cases}$$

$$= \begin{cases} \$۵۰ & if\ \ w_۱ = ۵ \le w \\ \$۰ & if\ \ w_۱ = ۵ > w \end{cases}$$

بنابراین،

$$p[۱][۰] = \$۰$$
$$p[۱][۱۰] = \$۵۰$$
$$p[۱][۲۰] = \$۵۰$$
$$p[۱][۳۰] = \$۵۰$$

محاسبه سطر ۲:

$$p[۲][۱۰] = \begin{cases} maximum(p[۱][۱۰],\ \$۶۰ + p[۱][۰]) & if\ \ w_۲ = ۱۰ \le ۱۰ \\ p[۱][۱۰] & if\ \ w_۲ = ۱۰ > ۱۰ \end{cases}$$

$$= \$۶۰$$

$$p[2][30] = \begin{cases} maximum(p[1][30],\ \$60 + p[1][20]) & if\ w_2 = 10 \le 30 \\ p[1][30] & if\ w_2 = 10 > 30 \end{cases}$$

$$= \$60 + \$50 = \$110$$

محاسبه سطر ۳:

$$p[3][30] = \begin{cases} maximum(p[2][30],\ \$140 + p[2][10]) & if\ w_3 = 20 \le 30 \\ p[2][30] & if\ w_3 = 20 > 30 \end{cases}$$

$$= \$140 + \$60 = \$200$$

این نسخه از الگوریتم تنها هفت ورودی را محاسبه می‌کند؛ در حالیکه نسخه اصلی، تعداد ۹۰ = (۳۰)(۳) ورودی را محاسبه می‌کرد. بیائید کارایی این الگوریتم را در بدترین حالت مشخص کنیم. توجه کنید که حداکثر 2^i ورودی را در سطر i-nام محاسبه می‌کنیم. بنابراین، ماکزیمم مجموع تعداد ورودیهای محاسبه شده برابراست با

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

این تساوی از مثال A-۳ در ضمیمه A بدست آمده است. به عنوان تمرین نشان خواهید داد که مشخصات زیر، نمونه‌ای از مسئله است که در آن تقریباً 2^n ورودی محاسبه می‌شود (ارزشها می‌توانند هر مقداری داشته باشند):

$$w_i = 2^{i-1} \quad \text{برای} \quad 1 \le i < n \quad \text{و} \quad W = 2^n - 2$$

با ترکیب این دو نتیجه مشخص می‌شود که بدترین حالت تعداد ورودیهای محاسبه شده در $\Theta(2^n)$ است. حدودی که تا به حال بدست آمده است، فقط شامل عنصر n می‌باشد. می‌خواهیم حدی را نیز برای ترکیبی از n و W بدست آوریم. می‌دانیم که تعداد ورودیهای محاسبه شده در $\Theta(nW)$ است. این نسخه، از رسیدن به این حد جلوگیری کند. اما ممکن است این چنین نیست. شما در تمرینات نشان خواهید داد که اگر $n = W + 1$ و برای هر i، $w_i = 1$ باشد، آنگاه مجموع تعداد ورودیهای محاسبه شده تقریباً برابراست با

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = \frac{(W+1)(n+1)}{2}$$

اولین تساوی از مثال A-۱ در ضمیمه A و دومین تساوی از این حقیقت که در این نمونه، $n = w + 1$ است، نتیجه می‌شود. لذا این حد برای مقدار بزرگ و دلخواهی از n و W برابراست با $\Theta(nW)$. با ترکیب نتایج فوق مشخص می‌شود که بدترین حالت تعداد ورودیهای محاسبه شده در $O(minimum(2^n, nW))$ است.

برای پیاده سازی الگوریتم، نیازی به تولید یک آرایهٔ کامل نداریم. بلکه فقط می‌توانیم ورودیهای مورد نیاز را ذخیره کنیم. اگر الگوریتم را به این روش اجرا کنیم، بدترین حالت حافظه مورد استفاده، حدودی مشابه حدود فوق خواهد داشت. ما می‌توانستیم با استفاده از عبارت محاسباتی $p[i][w]$ که در

ارائهٔ الگوریتم برنامه‌نویسی پویا بکار رفته، یک الگوریتم تقسیم و غلبه بنویسیم. برای این الگوریتم نیز بدترین حالت تعداد ورودیهای محاسبه شده در $\Theta(2^n)$ است. مزیت اصلی الگوریتم برنامه‌نویسی پویا، حد اضافی این الگوریتم در عناصر nW است. برای الگوریتم تقسیم و غلبه، چنین حدی وجود ندارد. در واقع، این حد از اختلاف اساسی بین برنامه‌نویسی پویا و تقسیم و غلبه بدست آمده است. یعنی برنامه‌نویسی پویا، یک نمونه مشابه را بیش از یکبار پردازش نمی‌کند. حد nW هنگامی که W در مقایسه n بزرگ نباشد، بسیار با اهمیت است.

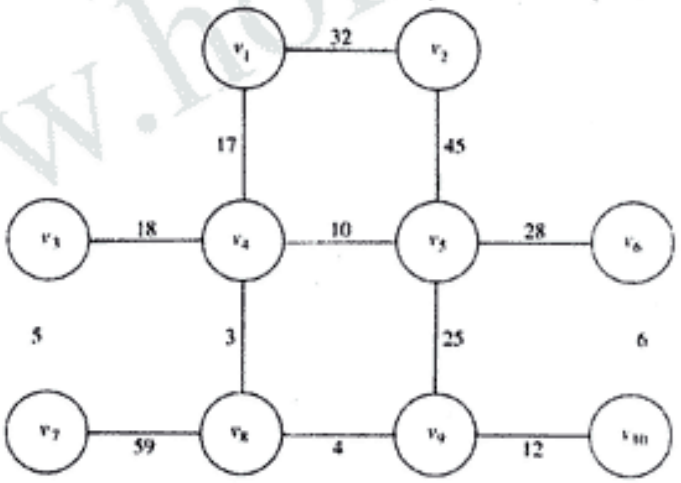
همانند مسئلهٔ فروشندهٔ دوره‌گرد، تا بحال کسی نتوانسته است الگوریتمی برای مسئله کوله‌پشتی ۱-۰ ارائه دهد که بدترین حالت پیچیدگی زمانی آن بهتر از حالت نمایی باشد و البته هنوز کسی ثابت نکرده است که چنین الگوریتمی وجود ندارد. در فصل ۹ بیشتر به این مسائل خواهیم پرداخت.

تمرینات

بخش ۱-۴

۱- نشان دهید که روش حریص همواره یک جواب بهینه برای مسئله پول خرد پیدا می‌کند، وقتی که سکه‌ها در ارزشهای $D^0, D^1, D^2, \ldots, D^i$ (برای مقادیر صحیح و مثبت $i>0$ و $D>0$) قرار داشته باشند.

۲- با استفاده از الگوریتم prim (الگوریتم ۱-۴) یک درخت پوشای می‌نیمم برای گراف زیر پیدا کنید.



۳- یک گراف رسم کنید که بیش از یک درخت پوشای می‌نیمم داشته باشد.

۴- الگوریتم prim (الگوریتم ۱-۴) را روی کامپیوتر خود اجرا کرده، کارایی آن را با استفاده از گرافهای مختلف بررسی نمائید.

۵- الگوریتم prim را به گونه‌ای تغییر دهید که متصل بودن یک گراف وزن‌دار و بدون‌جهت را بررسی کند. الگوریتم را تحلیل نموده و نتایج را با استفاده از نمادهای ترتیب نمایش دهید.

۶- با استفاده از الگوریتم kruskal (الگوریتم ۲-۴) یک درخت پوشای می‌نیمم برای گراف تمرین شماره ۲ پیدا کنید. مراحل را گام به گام نشان دهید.

۷- الگوریتم kruskal (الگوریتم ۲-۴) را روی کامپیوتر خود اجرا کرده، کارایی آن را با استفاده از گرافهای مختلف بررسی نمائید.

۸- آیا فکر می‌کنید که ممکن است یک درخت پوشای می‌نیمم دارای یک چرخه باشد؟ توضیح دهید.

۹- فرض کنید که در یک شبکه کامپیوتری، هر دو کامپیوتر می‌توانند به یکدیگر متصل شوند. با فرض اینکه هزینه هر اتصال مشخص باشد، از کدامیک از الگوریتمهای prim (الگوریتم۱-۴) یاKruskal (الگوریتم ۲-۴) استفاده می‌کنید؟

۱۰- با استفاده از پیش قضیه ۲-۴، اثبات قضیه ۲-۴ را کامل کنید.

بخش ۲-۴

۱۱- با استفاده از الگوریتم Dijkstra (الگوریتم ۳-۴)، کوتاهترین مسیر از گرهٔ V_4 به تمامی گره‌های دیگر در گراف تمرین شماره ۲ را پیدا نمائید. مراحل را گام به گام نشان دهید. فرض کنید که هر لبهٔ بدون‌جهت نمایانگر دو لبهٔ جهت‌دار با همان وزن باشد.

۱۲- الگوریتم Dijkstra (الگوریتم ۳-۴) را روی کامپیوتر خود اجرا نموده، کارایی آن را با استفاده از گرافهای مختلف بررسی کنید.

۱۳- الگوریتم Dijkstra را به گونه‌ای تغییر دهید که طول کوتاهترین مسیرها را محاسبه نماید. الگوریتم تغییر یافته را تحلیل نموده و نتایج را با استفاده از نمادهای ترتیب نشان دهید.

۱۴- الگوریتم Dijkstra را به گونه‌ای تغییر دهید که بررسی کند آیا یک گراف جهت‌دار دارای چرخه است یا خیر؟ الگوریتم را تحلیل نموده، نتایج را با استفاده از نمادهای ترتیب نمایش دهید.

۱۵- آیا الگوریتم Dijkstra می‌تواند برای پیدا کردن کوتاهترین مسیرها در یک گراف با وزنهای منفی بکار رود؟ توضیح دهید.

۱۶- با استفاده از استقراء، درستی الگوریتم Dijkstra (الگوریتم ۳-۴) را ثابت کنید.

بخش ۳-۴

۱۷- تقاضاها و زمانهای سرویس زیر را در نظر بگیرید. با استفاده از الگوریتم بخش ۱-۳-۴، مجموع مقدار زمان صرف شده در سیستم را به حداقل برسانید.

تقاضا	زمان سرویس
۱	۷
۲	۳
۳	۵
۴	۱۰

۱۸- الگوریتم بخش ۱-۳-۴ را روی کامپیوتر خود پیاده‌سازی نموده، آن را روی نمونه تمرین ۱۷ اجرا کنید.

۱۹- یک الگوریتم برای تعمیم مسئله زمانبندی تک‌سرویس‌دهنده به چندسرویس‌دهنده در بخش ۱-۳-۴ نوشته، آن را تحلیل نموده و نتایج را با استفاده از نمادهای ترتیب نشان دهید.

۲۰- تقاضاها، مهلت‌ها و بازده‌های زیر را در نظر بگیرید. با استفاده از الگوریتم زمانبندی مهلت‌دار (الگوریتم ۴-۴)، مجموع بازده را ماکزیمم کنید.

تقاضا	بازده	مهلت
۱	۴۰	۲
۲	۱۵	۴
۳	۶۰	۳
۴	۲۰	۲
۵	۱۰	۳
۶	۴۵	۱
۷	۵۰	۱

۲۱- روال schedule در الگوریتم زمانبندی مهلت‌دار (الگوریتم ۴-۴) را درنظر بگیرید. فرض کنید که d، حداکثر مهلت در n تقاضا باشد. روال را طوری تغییر دهید که یک تقاضای تا حد امکان دیر را به زمانبندی در حال ایجاد اضافه کند اما دیرتر از مهلتش نباشد. این کار را با مقدار دهی اولیهٔ d + ۱ مجموعهٔ غیرالحاقی شامل اعداد صحیح ۰، ۱،... d انجام دهید. فرض کنید small(s)، کوچکترین عضو S باشد. به هنگام زمانبندی یک تقاضا، مجموعه S، شامل می‌نیمم n و مهلت تقاضا را پیدا کنید. اگر small(S) = ۰ باشد، تقاضا را رد کنید وگرنه آن را در زمان small(s) زمانبندی نموده و S را با مجموعه‌ای شامل small(s)-۱ ادغام کنید. با فرض اینکه از ساختار داده‌ای مجموعه‌های غیرالحاقی III در ضمیمه C استفاده می‌کنیم، نشان دهید که این نسخه در $\Theta(n \lg m)$ است که در آن m، می‌نیمم d و n می‌باشد.

۲۲- الگوریتم تمرین ۲۱ را پیاده‌سازی کنید.

۲۳- فرض کنید که ما متوسط زمان ذخیره‌سازی K فایل به طولهای $l_1, l_2, ..., l_n$ بر روی یک نوار را به حداقل می‌رسانیم. اگر احتمال درخواست فایل K برابر P_k باشد، آنگاه زمان دستیابی مورد انتظار برای بارگذاری (load) این n فایل به ترتیب $k_1, k_2, ..., k_n$ به صورت فرمول زیر می‌باشد:

$$T_{average} = C \sum_{f=1}^{n} (p_{kf} \sum_{i=1}^{f} l_{ki})$$

که در آن ثابت C، مبین پارامترهایی نظیر سرعت در درایو و تراکم ذخیره‌سازی است.

a) یک روش حریص به چه ترتیبی باید این فایلها را ذخیره کند تا حداقل متوسط زمان دستیابی تضمین شود؟

b) الگوریتمی بنویسید که فایلها را ذخیره نماید. الگوریتم خود را تحلیل نمائید.

بخش ۴-۴

۲۴- یک الگوریتم برنامه‌نویسی برای مسئله کوله‌پشتی ۱-۰ بنویسید.

۲۵- با استفاده از روش حریص، یک درخت جستجوی دودویی بهینه بسازید که کلید با بیشترین احتمال key_k را به عنوان ریشه در نظر گرفته و زیر درختهای چپ و راست را برای $key_1, key_2, \ldots key_{k-1}$ و $key_{k+1}, key_{k+2}, \ldots, key_n$ به صورت بازگشتی و به همان روش ایجاد نماید.

a) با فرض اینکه کلیدها از قبل مرتب شده‌اند، بدترین حالت پیچیدگی زمانی این روش را تعیین کنید.

b) با استفاده از یک مثال نشان دهید که این روش حریص همواره یک درخت جستجوی دودویی بهینه پیدا نمی‌کند.

۲۶- با استفاده از روش برنامه‌نویسی پویا، یک الگوریتم برای مسئله تمرین ۲۶ بنویسید. الگوریتم را تحلیل نموده و نتایج را با استفاده از نمادهای ترتیب نشان دهید.

۲۷- با استفاده از روش حریص، الگوریتمی بنویسید که تعداد حرکات رکورد در مسئله ادغام n فایل را به حداقل برساند. از الگوی ادغام دوتایی استفاده کنید (دو فایل در هر مرحلهٔ ادغام با هم ترکیب می‌شوند). الگوریتم را تحلیل نموده، نتایج را با استفاده از نمادهای ترتیب نمایش دهید.

۲۸- ثابت کنید که روش حریص برای مسئله کوله‌پشتی جزئی، یک جواب بهینه تولید می‌کند.

۲۹- نشان دهید که بدترین حالت تعداد ورودیهای محاسبه شده توسط الگوریتم برنامه‌نویسی پویای تصفیه شده برای مسئله کوله‌پشتی ۱-۰ در $\Omega(n^2)$ می‌باشد. این کار را با درنظر گرفتن نمونه‌ای که $w = 2^n - 2$ و $w_i = 2^{i-1}$ $(1 \le i \le n)$ است، انجام دهید.

۳۰- نشان دهید در الگوریتم برنامه‌نویسی پویای تصفیه شده برای مسئله کوله‌پشتی ۱-۰، هنگامی که $n = w + 1, w_i = 1$ (برای هر i) است، مجموع تعداد ورودیهای محاسبه شده تقریباً برابراست با

$$(w+1) \times (n+1) / 2$$

تمرینات اضافی

۳۱- با یک مثال نقض نشان دهید که روش حریص همیشه نمی‌تواند یک جواب بهینه برای مسئله پول خرد تولید نماید، وقتی که سکه‌ها متعلق به امریکا بوده و ما حداقل یک سکه از هر نوع نداشته باشیم.

۳۲- ثابت کنید که یک گراف کامل (گرافی که در آن بین هر دو گره یک لبه وجود داشته باشد)، n^{n-2} درخت پوشا دارد. n، تعداد گره‌های گراف است.

۳۳- با استفاده از روش حریص، یک الگوریتم برای مسئله فروشندهٔ دوره‌گرد بنویسید. نشان دهید که الگوریتم شما همواره کوتاهترین تور را پیدا نمی‌کند.

۳۴- ثابت کنید که الگوریتم نوشته شده برای مسئله زمانبندی چندسرویس‌دهنده مثال ۱۹ همواره یک زمانبندی بهینه پیدا می‌کند.

فصل ۵

بازگشت به عقب (Backtracking)

اگر شما برای یافتن مسیرتان از میان مسیرهای پرپیچ وخم معروفی مثل کاخ Hampton Court در انگلستان سعی می‌کردید، مسلماً مسیری که نا امید کننده است را انتخاب نمی‌کردید تا به بن‌بست برسید؛ اما اگر چنین شد، بایستی به یک محل انشعاب برگشته و مسیر دیگری را شروع نمائید. هر کسی که تابحال سعی در حل یک پازل پیچیده داشته، حداقل یکبار بن‌بست را تجربه کرده است. تصور کنید که اگر تابلو یا نشانه‌ای وجود می‌داشت و وضعیت مسیر را اعلان می‌کرد، چقدر کار ما آسانتر می‌شد. اگر تابلویی در ابتدای هر مسیر قرار گیرد، در مدت زمان زیادی صرفه‌جویی کرده‌ایم زیرا از همه انشعاباتی که بعد از تابلو قرار دارند، صرفنظر می‌شود و بدین ترتیب، از بسیاری از بن‌بست‌ها جلوگیری می‌شود. چنین تابلوهایی در هزارتوهای معروف و یا در اکثر پازل‌های پیچیده وجود ندارند. به هر حال، همانطوریکه بعدها خواهیم دید، چنین نشانه‌هایی در الگوریتم‌های بک‌تراکینگ وجود دارند.

بک‌تراکینگ برای مسائلی نظیر مسئله کوله‌پشتی ۱-۰ بسیار کارا است. اگر چه در بخش ۳-۴-۴، الگوریتم کارآمدی از نوع برنامه‌نویسی پویا برای این مسئله، در حالتی که گنجایش کوله‌پشتی زیاد نباشد، پیدا کردیم؛ اما الگوریتم، باز هم در بدترین حالت به صورت زمان-نمایی است. مسئله کوله‌پشتی ۱-۰،

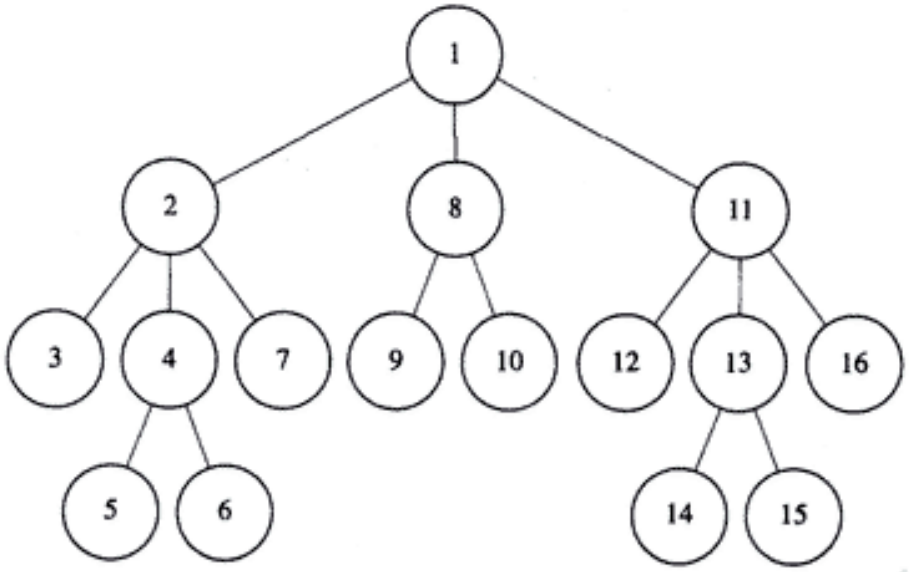
در کلاس مسئله‌های مطرح شده در فصل ۹ می‌باشد. هیچ کسی تاکنون نتوانسته است الگوریتم‌هایی برای آن دسته از مسائلی که پیچیدگی زمانی بدترین حالت آنها بهتر از زمان-نمایی باشد، پیدا کند و البته هیچ کسی نیز ثابت نکرده که ارائه چنین الگوریتمهایی غیرممکن است. یک روش برای حل مسئله کوله‌پشتی ۰-۱، تولید همه زیرمجموعه‌ها است؛ البته این روش شبیه به رفتن در هر مسیر یک هزارتو، تا رسیدن به یک بن‌بست است. از بخش ۴-۲-۱ به خاطر دارید که برای یک مجموعهٔ n عنصری، 2^n زیرمجموعه وجود دارد، بدین معنا که روش brute-force، تنها برای مقادیر پائین n امکان‌پذیر است. به هرحال، اگر به هنگام تولید زیرمجموعه‌ها بتوانیم علائمی را پیدا کنیم که به ما بگوید تعدادی از آنها احتیاج به تولید شدن ندارند، آنگاه می‌توانیم از بسیاری از عملیات غیرضروری اجتناب کنیم و این دقیقاً همان کاری است که یک الگوریتم بک‌تراکینگ انجام می‌دهد. الگوریتم بک تراکینگ برای مسائلی نظیر مسئله کوله‌پشتی ۰-۱، هنوز هم در بدترین حالت به صورت زمان-نمایی (یا حتی بدتر) است. این الگوریتم‌ها مفید هستند زیرا برای نمونه‌های بزرگ، کارا می‌باشند؛ اما نه برای تمام نمونه‌های بزرگ. ما در بخش ۷-۵ به مسئله کوله‌پشتی برمی گردیم. امّا قبل از آن، با یک مثال ساده در بخش ۱-۵، بک‌تراکینگ را معرفی نموده و در بخشهای بعد، چند مسئله را به طور نمونه حل می‌کنیم.

۱-۵ روش بک‌تراکینگ

بک‌تراکینگ برای حل مسائلی بکار می‌رود که در آن از یک مجموعهٔ مشخص، دنباله‌ای از اشیاء انتخاب می‌شوند بطوری که در دنباله، معیارهایی رعایت شده باشد. مسئله n-وزیر، یک مثال کلاسیک از الگوریتم بک‌تراکینگ است. هدف اصلی در این مسئله، تعیین موقعیت n-وزیر بر روی یک صفحه شطرنج $n \times n$ است بطوری که هیچ دو وزیری یکدیگر را تهدید نکنند. یعنی هیچ دو وزیری در یک سطر، ستون و یا قطر قرار نداشته باشند. **دنباله**، n موقعیتی است که وزیرها در آن قرار می‌گیرند. یک **مجموعه** برای هر انتخاب، n^2 موقعیت ممکن بر روی صفحه شطرنج است و **معیار** این است که هیچ دو وزیری یکدیگر را تهدید نکنند. مسئله n-وزیر، تعمیمی از صفحه شطرنج استاندارد با $n = 8$ است. در اینجا برای سهولت کار، بک‌تراکینگ را برای وقتی که $n = 4$ است، توضیح می‌دهیم.

بک‌تراکینگ، یک جستجوی عمقی(depth-first) روی یک درخت است (در اینجا، درخت یعنی درخت ریشه دار). لذا می‌خواهیم قبل از ادامه بحث، نگاهی به جستجوی عمقی داشته باشیم. اگرچه جستجوی عمقی، در حالت کلی، برای گراف‌ها تعریف می‌شود اما ما تنها به جستجوی درختی می‌پردازیم زیرا بک‌تراکینگ فقط با جستجوی درختی سروکار دارد. یک **پیمایش پیشوندی**(Preoder)، یک **جستجوی عمقی** روی درخت است. این بدین معنی است که ابتدا ریشهٔ درخت ملاقات می‌شود و بلافاصله بعد از ملاقات با یک گره، با همه فرزندان گرهٔ موردنظر ملاقات می‌کند. اگرچه جستجوی عمقی نیازی به ملاقات فرزندان با ترتیبی خاص ندارد، اما در این فصل فرزندان یک گره را از چپ به راست پیمایش می‌کنیم. شکل ۱-۵، جستجوی عمقی را بر روی یک درخت نشان می‌دهد. گره‌ها، به ترتیبی که

شکل ۱-۵ یک درخت با گره‌های شماره‌گذاری شده براساس یک جستجوی عمقی.



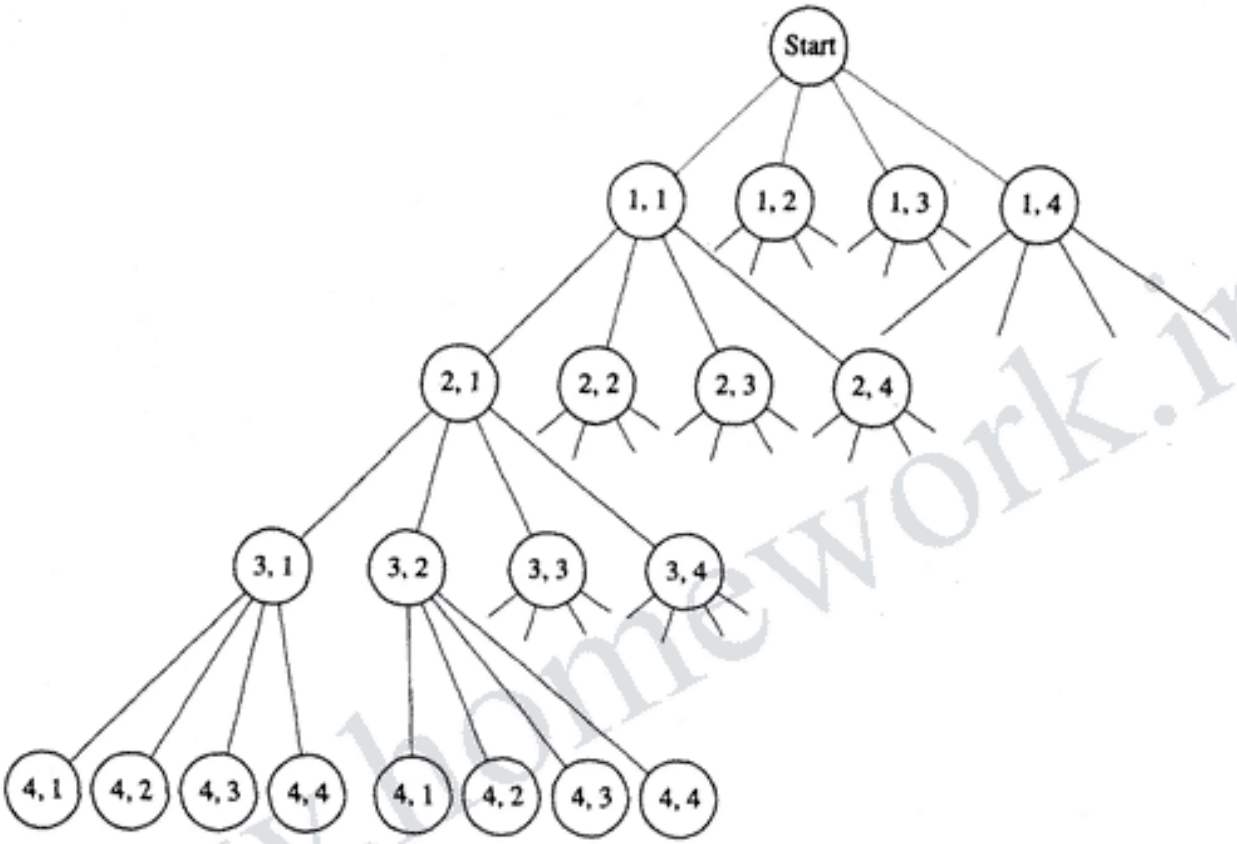
ملاقات می‌شوند، شماره‌گذاری می‌گردند. توجه نمائید که در یک جستجوی عمقی، یک مسیر تا انتهای عمق آن دنبال می‌شود تا به بن‌بست برسد. هنگام رسیدن به یک بن‌بست بایستی برگردد تا به گره‌ای برسد که فرزند ملاقات نشده دارد و به همین ترتیب، تا به عمیق‌ترین گرهٔ ممکن برسد.

در اینجا، یک الگوریتم بازگشتی ساده برای انجام یک جستجوی عمقی ارائه می‌دهیم . چون در حال حاضر، توجه ما به پیمایش پیشوندی است، لذا این نسخه را برای تبیین این مطلب بیان می‌کنیم. این روال، با ارسال ریشه درسطح بالا فراخوانی می‌شود.

```
void depth_first_tree_search (node v)
{
  node u;
  visit v;
  for (each child u of v)
      depth_first_tree_search(u);
}
```

اکنون می‌خواهم روش بک‌تراکینگ را با نمونه‌ای از مسئله n-وزیر با $n = ۴$ توضیح دهیم. وظیفهٔ ما قراردادن ۴ وزیر در یک صفحه شطرنج ۴×۴ است بطوری که هیچ دو وزیری یکدیگر را تهدید نکنند. ما می‌توانیم در ابتدا، موضوع را به این صورت ساده نمائیم که هیچ دو وزیری نمی‌توانند در یک سطر (ردیف) باشند. این نمونه را می‌توان با انتقال هر یک از وزیرها به ردیفی دیگر و بررسی اینکه ترکیبات چه ستونی منجر به جواب می‌شود، حل کرد. از آنجائیکه هر وزیر می‌تواند در هر یک از چهار ستون صفحه جای بگیرد، لذا به تعداد $۲۵۶ = ۴ \times ۴ \times ۴ \times ۴$ جواب کاندید وجود خواهد داشت. ما می‌توانیم جوابهای کاندید را با ساختن یک درخت ایجاد کنیم که در آن، ستون انتخابی برای اولین وزیر (وزیر ردیف ۱)

شکل ۲-۵ بخشی از یک درخت فضای حالات برای نمونه‌ای از مسئله n-وزیر، که ۴ = n می‌باشد. زوج مرتب <i, j> برای هر گره به این معنی است که وزیر، در ردیف i و ستون j قرار دارد. هر مسیر از ریشه به برگ، یک جواب کاندید است.



در گره‌های سطح ۱ درخت ذخیره می‌شود (توجه دارید که ریشه، همان سطح صفر درخت است). ستون انتخابی برای وزیر دوم (وزیر ردیف ۲) در گره‌های سطح ۲ ذخیره می‌شود و الی آخر. یک مسیر از ریشه به برگ، یک جواب کاندید می‌باشد(**برگ** در یک درخت، به گره‌ای اطلاق می‌شود که فرزندی ندارد). این درخت، به **درخت فضای حالات** موسوم است که بخشی از آن را در شکل ۲-۵ مشاهده می‌کنیم. یک درخت کامل، ۲۵۶ برگ دارد که هر کدام از آنها یک جواب کاندید می‌باشند. توجه دارید که در هر گره، یک زوج مرتب <i, j> ذخیره می‌شود. این زوج مرتب، مکان قرارگیری یک وزیر را در ردیف i و ستون j نشان می‌دهد.

برای تعیین جوابها، هر یک از جوابهای کاندید را با شروع از چپ‌ترین مسیر بررسی می‌کنیم (جواب کاندید، هر مسیر از ریشه به برگ است). تعدادی از اولین مسیرهای بررسی شده به صورت زیر است:

[<۱، ۱>، <۲، ۱>، <۳، ۱>، <۴، ۱>]

[<۱، ۱>، <۲، ۱>، <۳، ۱>، <۴، ۲>]

[<۱، ۱>، <۲، ۱>، <۳، ۱>، <۴، ۳>]

[<۱، ۱>، <۲، ۱>، <۳، ۱>، <۴، ۴>]

[<۱، ۱>، <۲، ۱>، <۳، ۲>، <۴، ۱>]

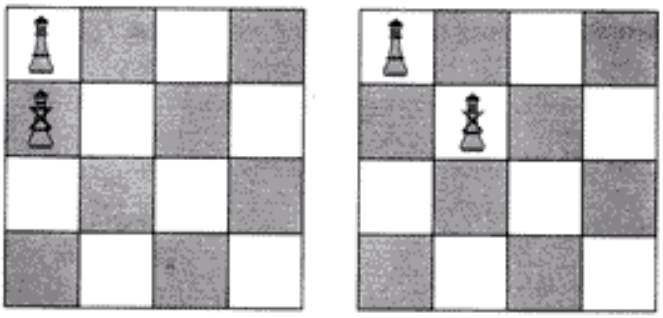
توجه کنید که گره‌ها بر اساس یک جستجوی عمقی که در آن فرزندان یک گره از چپ به راست پیمایش می‌شوند، ملاقات گردیده‌اند. یک جستجوی عمقی ساده روی یک درخت فضای حالات، شبیه به دنبال کردن هر مسیر در هزارتو تا رسیدن به یک بن‌بست است؛ بدون آنکه از علامتهایی در طول مسیر، استفاده شود. ما می‌توانیم با انجام جستجو به دنبال این علامتها، به کارایی بالاتری دست یابیم. به عنوان مثال، همانطوریکه در شکل (a)۳-۵ نشان داده شده است، هیچ دو وزیری نمی‌توانند در یک ستون قرار بگیرند. بنابراین، هیچ نکته‌ای در ایجاد و بررسی مسیرها در کل شاخهٔ منشعب از گرهٔ شامل <۲، ۱> در شکل ۲-۵ وجود ندارد زیرا قبلاً وزیر ۱ در ستون ۱ قرار گرفته و وزیر ۲ نمی‌تواند در آن ستون قرار بگیرد. این علامت به ما می‌گوید که این گره، جز به بن‌بست به جائی منتهی نمی‌شود. بطور مشابه، همانطوریکه در شکل (b) ۳-۵ نشان داده شده است، هیچ دو وزیری نمی‌توانند در یک قطر قرار بگیرند. بنابراین، هیچ نکته‌ای در ایجاد و بررسی شاخهٔ منشعب از گرهٔ شامل <۲، ۲> در شکل ۲-۵ وجود ندارد.

پک‌تراکینگ روالی است که به وسیلهٔ آن، بعد از آنکه فهمیدیم یک گره جز به بن‌بست به جائی ختم نمی‌شود، می‌توانیم به عقب برگردیم تا به پدر گرهٔ جاری برسیم و عملیات جستجو را بر روی فرزند بعدی ادامه دهیم. یک گره را **غیروعده‌گاه(nonpromising)** گوئیم اگر به هنگام ملاقات گره، مشخص شود که احتمالاً آن گره به جواب منتهی نمی‌شود؛ در غیراینصورت، آن را **وعده‌گاه(promising)** می‌نامیم. به طور خلاصه، بک‌تراکینگ شامل یک "جستجوی عمقی" در یک درخت فضای حالات، "بررسی" اینکه آیا یک گره وعده‌گاه است یا خیر و "بازگشت" به پدر گره (در صورت غیروعده‌گاه بودن گره) می‌باشد. به این عمل، **هرس** درخت فضای حالات گوئیم و زیر درخت شامل گره‌های ملاقات شده، **درخت فضای حالات هرس شده** نامیده می‌شود. یک الگوریتم عمومی برای روش بک‌تراکینگ به صورت زیر است:

```
void checknode (node v)
{
  node u;
  if (promising(v))
    if (there is a solution at v)
      write the solution;
    else
      for (each child u of v)
        checknode(u);
}
```

در سطح بالا، ریشهٔ درخت فضای حالات به روال checknode ارسال می‌گردد. پیمایش یک گره، در ابتدا با بررسی اینکه آیا آن گره یک وعده‌گاه است یا خیر، شروع می‌شود. اگر گره یک وعده‌گاه بود و یک جواب در آن گره وجود داشت، آن جواب چاپ می‌شود و اگر جوابی در گرهٔ وعده‌گاه وجود نداشت، فرزندان گره

شکل ۳-۵ اگر وزیر اول در ستون ۱ جای گرفت، وزیر دوم نمی‌تواند (a) در ستون ۱ یا (b) در ستون ۲ قرار گیرد.



(a) (b)

پیمایش می‌شوند. تابع promising در هر کاربرد بک‌تراکینگ متفاوت است. ما این تابع را **تابع وعده‌گاه** الگوریتم‌می‌نامیم. یک الگوریتم بک‌تراکینگ شبیه به یک جستجوی عمقی است به جز آنکه فرزندان یک گره، وقتی ملاقات می‌شوند که گره یک وعده‌گاه باشد ولی جوابی در آن گره وجود نداشته باشد. برخلاف الگوریتم مسئله n-وزیر، در برخی از الگوریتمهای بک‌تراکینگ می‌توان یک جواب را قبل از رسیدن به برگ درخت فضای حالات پیدا کرد. ما روال بک‌تراکینگ را به جای آنکه backtrack بنامیم، checknode نامیدیم زیرا به هنگام فراخوانی روال، عمل بازگشت به عقب صورت نمی‌گیرد؛ بلکه عمل بازگشت وقتی صورت می‌گیرد که گره‌ای را پیدا کنیم که غیروعده‌گاه باشد و بخواهیم عملیات را بر روی فرزند بعدی پدر ادامه دهیم. یک پیاده سازی کامپیوتری الگوریتم بازگشتی، عمل بک‌تراکینگ را با pop کردن رکورد فعال سازی برای یک گرهٔ غیر وعده‌گاه از پشته رکوردهای فعال سازی انجام می‌دهد.

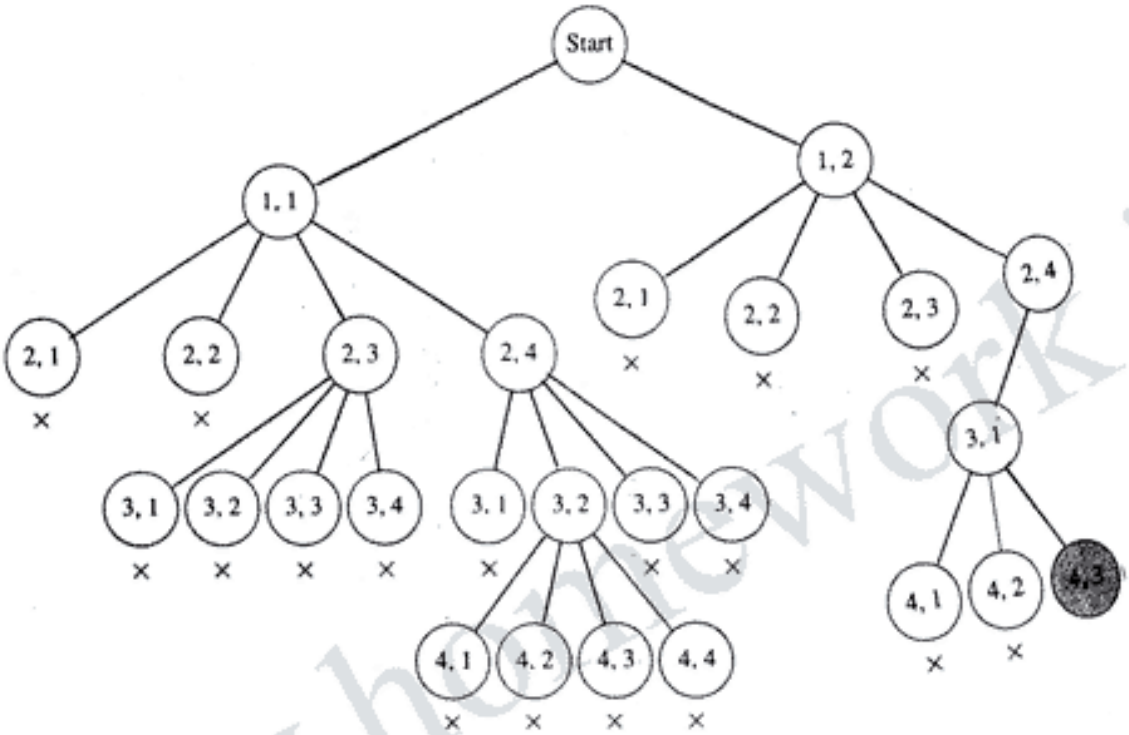
در ادامه با استفاده از بک‌تراکینگ ، نمونه مسئله n-وزیر را برای ۴ = n حل می‌کنیم.

مثال ۱-۵ به خاطر آورید که تابع Promising برای هر کاربرد از بک‌تراکینگ متفاوت است. برای مسئله n-وزیر، اگر یک گره و هر یک از اجدادش، وزیرها را در همان ستون یا قطر قرار دهند، تابع بایستی مقدار False را برگرداند. شکل ۴-۵، بخشی از یک درخت هرس‌شدهٔ فضای حالات را نشان می‌دهد که با استفاده از بک‌تراکینگ، برای حل نمونه‌ای با ۴ = n، تولید شده است. در این شکل، تنها گره‌هایی که برای یافتن اولین جواب بررسی شده‌اند، نمایش داده می‌شوند. شکل ۵-۵، یک صفحه شطرنج واقعی را نشان می‌دهد. در شکل ۴-۵، گره‌های غیروعده‌گاه و در شکل ۵-۵، مکانهای غیروعده‌گاه با علامت × مشخص شده‌اند. در شکل ۴-۵، گره سایه‌دار گره‌ای است که اولین جواب در آن یافت می‌شود. ما به وسیله زوج مرتبی که در یک گره ذخیره شده است، به آن گره رجوع می‌کنیم. برخی از گره‌ها دارای زوج مرتب یکسانی هستند اما شما می‌توانید با پیمایش درخت شکل ۴-۵ بگوئید که منظورمان کدام گره می‌باشد.

(a) <۱، ۱> وعده‌گاه است. {زیرا وزیر ۱، اولین وزیری است که جای می‌گیرد}

(b) <۲، ۱> غیروعده‌گاه است. {زیرا وزیر ۱ در ستون ۱ قرار دارد}

شکل ۴-۵ بخشی از درخت فضای حالات هرس شده که به هنگام بکارگیری بک‌تراکینگ برای حل نمونه‌ای از مسئله n-وزیر با ۴ = n، تولید شده است. فقط گره‌هایی که برای یافتن جواب بررسی شده‌اند، نشان داده می‌شوند. جواب در گرهٔ سایه‌دار یافت می‌شود. هر گرهٔ غیروعده‌گاه با یک علامت × مشخص شده است.



<2، ۲> غیروعده‌گاه است. {زیرا وزیر ۱ روی قطر چپ قرار دارد}

<2، ۳> وعده‌گاه است.

(c) <3، ۱> غیروعده‌گاه است. {زیرا وزیر ۱ در ستون ۱ قرار دارد}

<3، ۲> غیروعده‌گاه است. {زیرا وزیر ۲ روی قطر راست وجود دارد}

<3، ۳> غیروعده‌گاه است. {زیرا وزیر ۲ در ستون ۳ قرار دارد}

<3، ۴> غیروعده‌گاه است. {زیرا وزیر۳ روی قطر چپ قرار دارد}

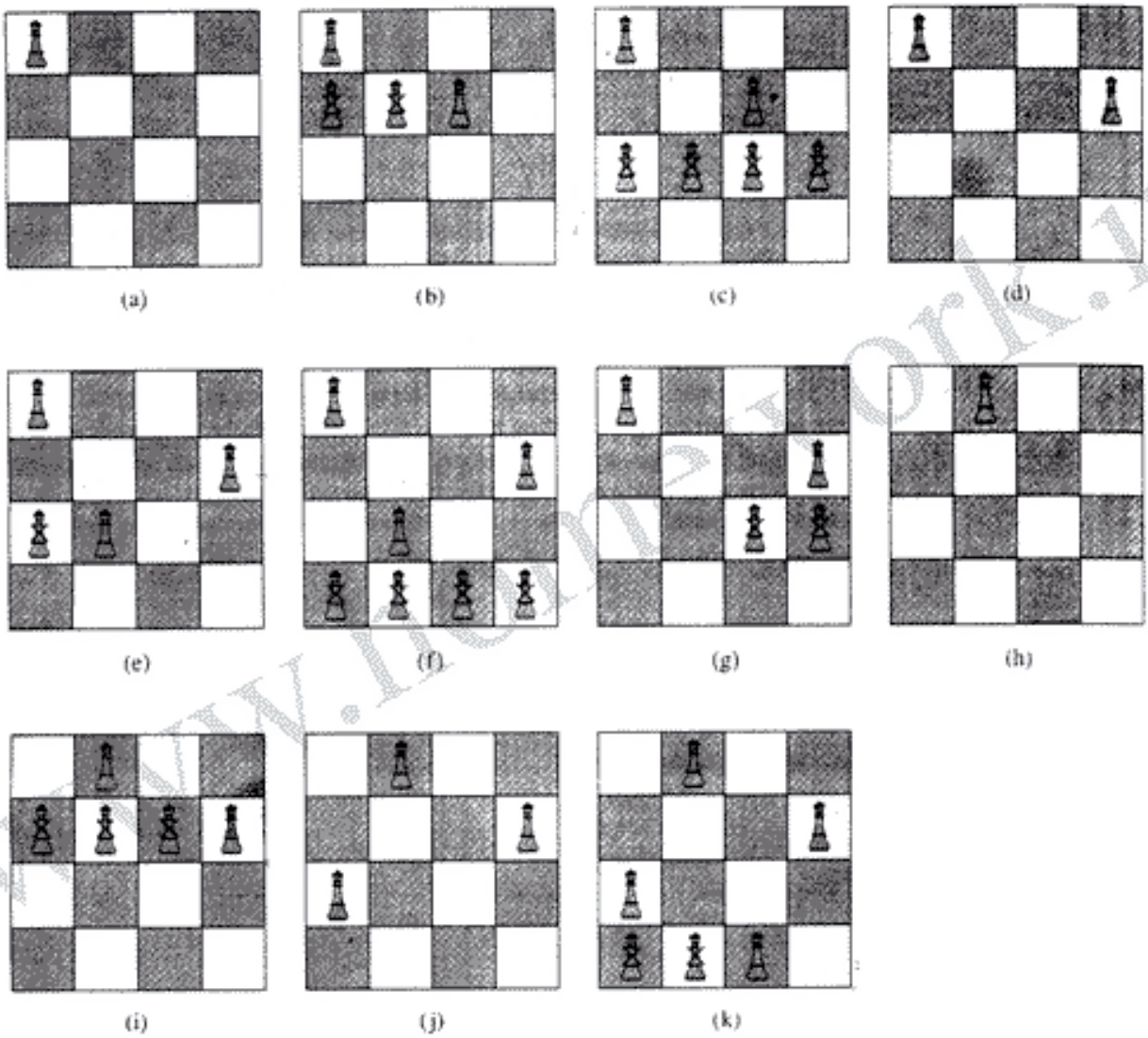
(d) به <1، ۱> برگردید.

<2، ۴> غیروعده‌گاه است.

(e) <3، ۱> غیروعده‌گاه است. {زیرا وزیر ۱ در ستون ۱ قرار دارد}

<3، ۲> وعده‌گاه است. {این دفعه دوم است که <3، ۲> را آزمایش می‌کنیم}

شکل ۵-۵ موقعیتهای صفحه شطرنج واقعی که به هنگام استفاده از بک‌تراکینگ برای حل مسئله n-وزیر با $n = ۴$ آزمایش می‌شوند. هر موقعیت غیروعده‌گاه، توسط یک علامت × مشخص شده است.



(f) <۴، ۱> غیروعده‌گاه است. {زیرا وزیر ۱ در ستون ۱ قرار دارد}

<۴، ۲> غیروعده‌گاه است. {زیرا وزیر ۳ در ستون ۲ قرار دارد.}

<۴، ۳> غیروعده‌گاه است. {زیرا وزیر ۳ روی قطر چپ قرار دارد}

<۳، ۴> غیروعده‌گاه است. {زیرا وزیر ۲ در ستون ۴ قرار دارد}

(g) به <۲،۴> برگردید.
<۳،۳> غیروعده‌گاه است. {زیرا وزیر ۲ روی قطر راست قرار دارد}
<۳،۴> غیروعده‌گاه است. {زیرا وزیر ۲ روی ستون ۴ قرار دارد}

(h) به ریشه برگردید.
<۱،۲> وعده‌گاه است.

(f) <۴،۱> غیروعده‌گاه است. {زیرا وزیر ۱ در ستون ۱ قرار دارد}
<۴،۲> غیروعده‌گاه است. {زیرا وزیر ۳ در ستون ۲ قرار دارد.}

(i) <۲،۱> غیروعده‌گاه است. {زیرا وزیر ۱ بر روی قطرراست قرار دارد}
<۲،۲> غیروعده‌گاه است. {زیرا وزیر ۱ بر روی ستون ۲ قرار دارد.}
<۲،۳> غیروعده‌گاه است. {زیرا وزیر ۱ بر روی قطر چپ قرار دارد}
<۲،۴> وعده‌گاه است.

(j) <۳،۱> وعده‌گاه است. {این دفعه سوم است که <۳،۱> را آزمایش می‌کنیم}

(k) <۴،۱> غیروعده‌گاه است. {زیرا وزیر ۳ در ستون ۱ قرار دارد}
<۴،۲> غیروعده‌گاه است. {زیرا وزیر ۱ در ستون ۲ قرار دارد}
<۴،۳> وعده‌گاه است.

در این لحظه، اولین جواب پیدا شد. این جواب در شکل ۵-۵ (k) نمایان است و گره‌ای که جواب در آن پیدا شده، در شکل ۵-۴ سایه‌دار گشته است.

توجه کنید که الگوریتم بک‌تراکینگ واقعاً نیاز به ساختن درخت ندارد؛ بلکه فقط بایستی ردّ مقادیر شاخهٔ جاری که مورد بررسی قرار می‌گیرد را نگه دارد. این روشی است که در پیاده‌سازی الگوریتم‌های بک‌تراکینگ بکار گرفته می‌شود. ما می‌گوئیم که درخت فضای حالات، به طور ضمنی وجود دارد زیرا واقعاً درختی ساخته نمی‌شود.

شمارش گره‌های شکل ۵-۴ نشان می‌دهد که الگوریتم بک‌تراکینگ قبل از یافتن یک جواب، ۲۷ گره را مورد بررسی قرار می‌دهد. در تمرینات نشان خواهید داد که بدون استفاده از روش بک‌تراکینگ ، یک جستجوی عمقی درخت فضای حالات، ۱۵۵ گره را قبل از یافتن همان جواب بررسی می‌کند.

ممکن است متوجه عدم کارایی در الگوریتم عمومی ما برای بک‌تراکینگ (checknode) شده باشید. بدینصورت که ما پس از ارسال یک گره به روال، وعده‌گاه بودن آن را بررسی می‌کنیم و این بدین معنی است که رکوردهای فعال سازی گره‌های غیروعده‌گاه، بی‌دلیل در پشتهٔ رکوردهای فعال‌سازی گذارده می‌شوند. یک الگوریتم عمومی بک‌تراکینگ که این عمل را انجام می‌دهد به صورت زیر است:

```
void expand (node v)
{
  node u;
  for (each child u of v)
    if (promising(u))
      if (there is a solution at u)
        write the solution;
      else
        expand(u);
}
```

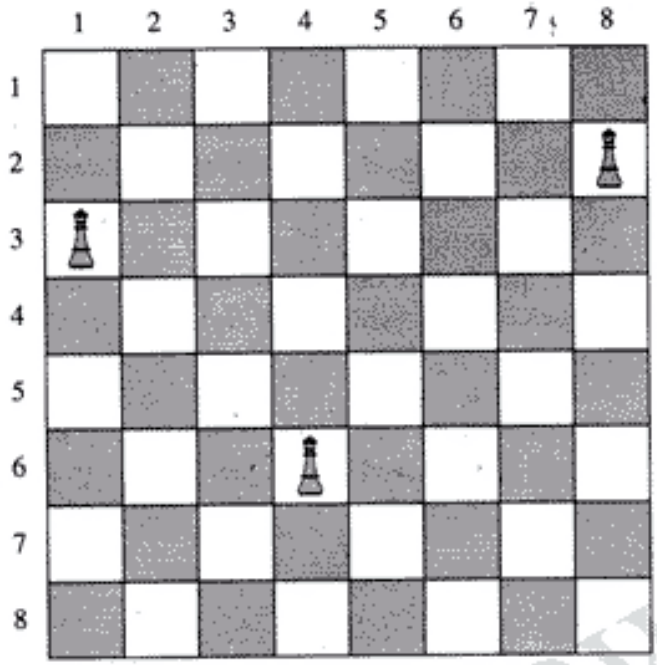
گره‌ای که در سطح بالا به روال ارسال می‌شود، همان ریشهٔ درخت است. این روال را expand نامیدیم زیرا این روال در صورتی فراخوانی می‌شود که یک گرهٔ وعده‌گاه را بسط دهیم. در پیاده‌سازی این الگوریتم بر روی کامپیوتر، بک‌تراکینگ از یک گرهٔ غیروعده‌گاه با push نکردن رکورد فعال‌سازی گره در پشتهٔ رکوردهای فعال‌سازی انجام می‌شود. در تشریح الگوریتم‌های این فصل، از نسخهٔ اول الگوریتم (روال checknode) استفاده می‌کنیم زیرا دریافته‌ایم که این نسخه، معمولاً الگوریتم‌هایی تولید می‌کند که فهم آنها راحت‌تر است؛ بدین علت که یک اجرای checknode، شامل مراحلی است که به هنگام ملاقات یک گرهٔ تنها انجام می‌شود، یعنی شامل مراحل زیر است: تعیین اینکه آیا گره یک وعده‌گاه است یا خیر و اگر وعده‌گاه است، آیا یک جواب در آن گره وجود دارد یا خیر، که در صورت وجود جواب بایستی آن را چاپ کند؛ در غیر اینصورت فرزندان آن را ملاقات نماید. از طرفی دیگر، یک اجرای expand شامل انجام همان مراحل بر روی تمامی فرزندان یک گره است.

در ادامه، الگوریتمهای بک‌تراکینگ را برای چندین مسئله ارائه می‌دهیم و این کار را با مسئله n-وزیر آغاز می‌کنیم. در تمامی این مسائل، درخت فضای حالات شامل تعداد گره‌هایی به صورت نمایی و یا بیشتر است. بک‌تراکینگ، از بررسی غیرضروری گره‌ها جلوگیری می‌کند. اگر دو نمونه با مقدار یکسان n داشته باشیم، یک الگوریتم بک‌تراکینگ ممکن است گره‌های کمی را برای یکی از نمونه‌ها بررسی کند اما برای نمونهٔ دیگر ممکن است تمامی گره‌های درخت فضای حالات را مورد بررسی قرار دهد. این بدان معنی است که ما نمی‌توانیم یک پیچیدگی زمانی کارا برای الگوریتمهای بک‌تراکینگ پیدا کنیم (برخلاف الگوریتم‌های فصلهای گذشته). بنابراین، به جای انواع تحلیلهائی که در فصلهای گذشته انجام می‌شد، الگوریتم‌های بک‌تراکینگ را با استفاده از روش مونت کارلو مورد تجزیه و تحلیل قرار می‌دهیم. این روش بررسی می‌کند که آیا می‌توانیم از یک الگوریتم بک‌تراکینگ انتظار داشته باشیم که برای نمونهٔ خاصی کارا باشد یا خیر. روش مونت کارلو را در بخش ۳-۵ بررسی می‌کنیم.

۲-۵ مسئله n-وزیر

قبلاً راجع به هدف مسئله n-وزیر بحث کردیم. تابع وعده‌گاه بایستی بررسی کند که آیا دو وزیر در یک

شکل ۶-۵ وزیر سطر ۶ با وزیر ۳ در قطر چپ و با وزیر سطر ۲ در قطر راست بررسی می‌شود.



ستون و یا قطر واقع شده‌اند یا خیر. اگر فرض کنیم که $col(i)$ ستونی باشد که وزیر موجود در سطر iام در آن واقع است، در اینصورت برای بررسی اینکه یک وزیر سطر kام در همان ستون قرار دارد یا خیر، بایستی بررسی کنیم که آیا $col(i) = col(k)$ است یا خیر. در ادامه، به چگونگی بررسی قطرها می‌پردازیم. شکل ۶-۵، نمونه‌ای با $n = ۸$ را نشان می‌دهد. در این شکل، وزیر سطر ۶ با وزیر سطر ۳ در قطر سمت چپ و با وزیر سطر ۲ در قطر سمت راست بررسی می‌شود. توجه داشته باشید که

$$col(۶) - col(۳) = ۴ - ۱ = ۳ = ۶ - ۳$$

یعنی برای بررسی وزیری که در قطر چپ قرار دارد، تفاوت ستونها همان تفاوت سطرها است. علاوه براین،

$$col(۶) - col(۲) = ۴ - ۸ = -۴ = ۲ - ۶$$

یعنی برای بررسی وزیری که در قطر راست قرار دارد، تفاوت ستونها برابراست با منفی تفاوت سطرها. اینها مثالهایی از یک نتیجهٔ کلی هستند که اگر وزیر سطر kام، وزیر سطر iام در طول قطرش تهدید کند، دراینصورت داریم

$$col(i) - col(k) = i - k \qquad \text{یا} \qquad col(i) - col(k) = k - i$$

الگوریتم ۱-۵ الگوریتم بک‌تراکینگ برای مسئله n-وزیر

مسئله: n وزیر را بر روی یک صفحه شطرنج طوری قرار دهید که هیچ دو وزیری در یک سطر، ستون و یا قطر قرار نداشته باشند.

ورودی: عدد صحیح مثبت n.

خروجی: تمام حالاتی که n-وزیر می‌توانند بر روی یک صفحه شطرنجی n × n قرار گیرند بطوری که هیچ دو وزیری یکدیگر را تهدید نکنند. هر خروجی شامل آرایه‌ای از اعداد صحیح (col) می‌باشد که از ۱ تا n شاخص دهی شده است و در آن *col* (*i*) ستونی است که وزیر سطر *i*ام در آن واقع است.

```
void queens (index i)
{
  index j;

  if (promising(i))
    if (i == n)
      cout << col[1] through col[n];
    else
      for (j = 1; j <= n; j++) {          // See if queen in (i + 1)st row
        col[i + 1] = j;                   // can be positioned in each of
        queens(i + 1);                    // the n columns.
      }
}

bool promising (index i)
{
  index k;
  bool switch;

  k = 1;
  switch = true;                          // Check if any queen threatens
  while (k < i && switch) {               // queen in the ith row.
    if (col[i] == col[k] || abs(col[i] - col[k]) == i - k)
      switch = false;
    k++;
  }
  return switch;
}
```

هنگامی که در یک الگوریتم، بیش از یک روتین وجود داشته باشد، آنها را بر اساس قوانین زبان برنامه‌نویسی خاصی مرتب نمی‌کنیم؛ بلکه فقط روتین اصلی را در ابتدا معرفی می‌کنیم. این روتین در الگوریتم ۱-۵، queens نام دارد. طبق قراردادی که در بخش ۱-۲ داشتیم، n و col ورودیهای روال بازگشتی queens نیستند. اگر الگوریتم با تعریف n و col به صورت سراسری پیاده‌سازی می‌شد، فراخوانی سطح بالای queens، بدین صورت بود:

```
queens(0);
```

الگوریتم ۱-۵ تمام جوابهای مسئله n-وزیر را تولید می‌کند زیرا در این الگوریتم گفته‌ایم که با یافتن

یک جواب، از الگوریتم خارج نشود. به طور کلی، مسائل این فصل را می‌توان برای یک یا چند و یا تمام جوابها انجام داد. در عمل، بسته به نیاز کاربردی خاص، یک یا چند و یا تمام جوابها را بدست می‌آوریم. الگوریتم‌های ما، اغلب برای تولید تمام جوابها نوشته شده‌اند. متوقف نمودن الگوریتم پس از یافتن یک جواب، با یک تغییر ساده امکانپذیر است.

تحلیل الگوریتم ۱-۵ به صورت تئوری مشکل است. برای انجام این کار بایستی تعداد گره‌های بررسی‌شده را به عنوان تابعی از n مشخص کنیم (n، همان تعداد وزیران است). ما می‌توانیم با شمارش گره‌های درخت فضای حالات، حدّ بالایی را برای تعداد گره‌های درخت فضای حالات هرس‌شده بدست آوریم. این درخت شامل یک گره در سطح صفر، n گره در سطح ۱، n^2 گره در سطح ۲ و... و n^n گره در سطح n می‌باشد. مجموع تعداد گره‌ها برابراست با

$$1+n+n^2+n^3+\ldots+n^n=\frac{n^{n+1}-1}{n-1}$$

این تساوی از مثال ۴-A از ضمیمه A بدست آمده است. برای نمونه‌ای با n = ۸، درخت فضای حالات شامل $(8^{8+1}-1)/(8-1)=19,173,961$ گره می‌باشد.

تحلیل دیگری که می‌توانیم انجام دهیم، یافتن یک حد بالا برای تعداد گره‌های وعده‌گاه است. برای محاسبهٔ چنین حدی از این واقعیت استفاده می‌کنیم که هیچ دو وزیری نمی‌توانند در یک ستون قرار گیرند. به عنوان مثال، نمونه‌ای را در نظر بگیرید که در آن n = ۸ است. اولین وزیر می‌تواند در هر یک از هشت ستون قرار گیرد. پس از وزیر اول، وزیر دوم می‌تواند حداکثر در یکی از هفت ستون باقیمانده قرار گیرد و به همین ترتیب الی آخر. بنابراین، حداکثر

$$1+8+8\times7+8\times7\times6\times5+\ldots+8!=109601$$

گرهٔ وعده‌گاه وجود دارد. با تعمیم این نتیجه به هر مقدار دلخواهی از n به این نتیجه می‌رسیم که حداکثر $n!+\ldots+n(n-1)(n-2)+n(n-1)+n+1$ گرهٔ وعده‌گاه وجود دارد.

این تحلیل نمی‌تواند ایدهٔ خوبی از کارایی الگوریتم را به ما نشان دهد زیرا اولاً، در تابع Promising بررسی قطری انجام نمی‌شود. لذا، تعداد گره‌های وعده‌گاه کمتر از این حدّ بالا نیز می‌توانند وجود داشته باشند. ثانیاً، تعداد کل گره‌های بررسی‌شده شامل تمامی گره‌های وعده‌گاه و غیروعده‌گاه می‌باشد. همانطوریکه بعداً خواهیم دید، تعداد گره‌های غیروعده‌گاه می‌تواند به طور قابل توجهی بیشتر از تعداد گره‌های وعده‌گاه باشد.

یک روش مشخص برای تعیین کارایی الگوریتم این است که آن را به طور واقعی بر روی کامپیوتر اجرا کنیم و تعداد گره‌های بررسی‌شده را بشماریم. جدول ۱-۵، نتایج را با مقادیر متعددی از n نشان می‌دهد. الگوریتم بک‌تراکینگ با دو الگوریتم دیگر برای مسئله n-وزیر مقایسه شده است.الگوریتم ۱، یک جستجوی عمقی درخت فضای حالات بدون بک‌تراکینگ است. الگوریتم ۲، فقط از این واقعیت استفاده می‌کند که هیچ دو وزیری نمی‌توانند در یک سطر یا یک ستون قرار بگیرند. این الگوریتم، تعداد n! جواب کاندید را با روش زیر تولید می‌کند: بررسی وزیر سطر ۱ در هر یک از n ستون، وزیر سطر ۲ در

جدول ۱-۵ یک نمونه از تعداد بررسی‌های انجام شده توسط یک تراکینگ در مسئله n-وزیر

n	Number of Nodes Checked by Algorithm 1†	Number of Candidate Solutions Checked by Algorithm 2‡	Number of Nodes Checked by Backtracking	Number of Nodes Found Promising by Backtracking
4	341	24	61	17
8	19,173,961	40,320	15,721	2057
12	9.73×10^{12}	4.79×10^{8}	1.01×10^{7}	8.56×10^{5}
14	1.20×10^{16}	8.72×10^{10}	3.78×10^{8}	2.74×10^{7}

هر یک از n-۱ ستونی که توسط وزیر اول اشغال نشده است، وزیر سطر ۳ در هر یک از n-۲ ستونی که توسط دو وزیر اول اشغال نشده است و الی آخر. پس از تولید جوابهای کاندید بررسی می‌کند که آیا دو وزیر، یکدیگر را از لحاظ قطری تهدید می‌کنند یا خیر. توجه داشته باشید که مزایای الگوریتم بک‌تراکینگ با افزایش n، افزایش می‌یابد. زمانی که n = ۴ است، الگوریتم ۱، کمتر از ۶ برابر تعداد گره‌ها در الگوریتم بک‌تراکینگ، و الگوریتم بک‌تراکینگ، کمی بدتر از الگوریتم ۲ به نظر می‌رسد؛ اما زمانی که n = ۱۴ است، الگوریتم ۱، تقریباً ۳۲،۰۰۰،۰۰۰ مرتبه بیشتر از الگوریتم بک‌تراکینگ ، عمل بررسی گره‌ها را انجام می‌دهد. همچنین، تعداد جوابهای کاندید الگوریتم ۲، تقریباً ۲۳۰ مرتبه بیشتر از تعداد گره‌های بررسی‌شده در الگوریتم بک‌تراکینگ است. ما تعداد گره‌های وعده‌گاه را در جدول ۱-۵ آورده‌ایم تا نشان دهیم که بسیاری از گره‌های بررسی‌شده، غیروعده‌گاه می‌باشند. این بدان معنی است که روش دوم پیاده‌سازی بک‌تراکینگ (روال expand که در بخش ۱-۵ بررسی شد)، زمان زیادی را صرفه‌جویی می‌کند.

در واقع، اجرای یک الگوریتم به منظور تعیین کارایی آن (نظیر آنچه که در جدول ۱-۵ نشان داده شد)، یک تحلیل واقعی نیست. این کار را صرفاً به این خاطر انجام دادیم که نشان دهیم الگوریتم بک‌تراکینگ تا چه اندازه می‌تواند در زمان صرفه‌جوئی نماید. در بخش بعد، با استفاده از روش مونت کارلو، روشی را برای تخمین میزان کارایی یک الگوریتم بک‌تراکینگ ارائه خواهیم داد.

به خاطر آوردید که در درخت فضای حالات مسئله n-وزیر از این واقعیت استفاده کردیم که هیچ دو وزیری نمی‌توانند در یک سطر قرار بگیرند. روش دیگر این است که می‌توانیم هر وزیر را در هر یک از مکانهای n^2گانه صفحهٔ شطرنج آزمایش کنیم. در اینصورت، هرگاه یک وزیر در یک سطر یا ستون و یا قطری که وزیری دیگر در آن قرار داشت قرار گیرد، می‌تواند به عقب برگردد (بک‌تراک کند). هر گره در این درخت فضای حالات دارای n^2 فرزند و هر کدام از این فرزندان برای یک خانهٔ شطرنج می‌باشد. در اینصورت، $(n^2)^n$ برگ وجود دارد که هر کدام مبین یک جواب کاندید مجزا می‌باشند. الگوریتمی که در این درخت فضای حالات بک‌تراک می‌کند، تعداد گره‌های وعده‌گاه بیشتر از الگوریتم ما پیدا نمی‌کند اما همچنان کندتر از الگوریتم ما عمل می‌نماید.

مدت زمان صرف شده در تابع وعده‌گاه، از جمله مواردی است که در یک الگوریتم بک‌تراکینگ بایستی درنظر گرفته شود. در واقع، هدف ما این نیست که تعداد گره‌های بررسی شده را کم کنیم؛ بلکه هدف اصلی، بهبود کلی کارایی است. یک تابع برنامه‌نویسی که بسیار زمانبر است می‌تواند موجب خنثی شدن مزیت بررسی گره‌های کمتر شود. در الگوریتم ۱-۵ می‌توان تابع وعده‌گاه را با نگهداری ردّ مجموعه‌های ستونها، قطرهای سمت چپ و قطرهای سمت راستی که قبلاً توسط وزیرهای موجود کنترل شده‌اند، بهبود بخشید. در این روش نیازی به بررسی این موضوع نیست که آیا وزیرانی که قبلاً جایگذاری شده‌اند، وزیر کنونی را تهدید می‌کنند یا خیر، بلکه فقط بررسی کنیم که آیا وزیر کنونی می‌خواهد بر روی ستون و یا قطری که قبلاً کنترل شده قرار گیرد یا خیر. این بهبود کارایی در تمرینات بررسی خواهد شد.

۵-۳ استفاده از الگوریتم مونت کارلو برای تخمین میزان کارایی یک الگویتم بک‌تراکینگ

اگر دو نمونه با اندازهٔ یکسان n داشته باشیم، ممکن است یکی از آنها نیاز به بررسی تعداد گره‌های بسیار کمی داشته باشد؛ حال آنکه دیگری، نیاز به بررسی کل درخت فضای حالات داشته باشد. اگر تخمینی از میزان کارایی الگوریتم بک‌تراکینگ برای نمونه‌ای خاص داشته باشیم، در اینصورت می‌توانیم تصمیم بگیریم که آیا استفاده از این الگوریتم برای حل مسئله عاقلانه است یا خیر. ما می‌توانیم این تخمین را به وسیله الگوریتم مونت کارلو بدست آوریم.

الگوریتم‌های مونت کارلو، الگوریتم‌های احتمالی هستند. در یک **الگوریتم احتمالی**، دستورالعمل بعدی، گاهی به صورت اتفاقی اجرا می‌شود؛ در حالیکه در الگوریتم قطعی چنین نیست. تمام الگوریتم‌هایی که تاکنون بررسی کرده‌ایم، از نوع قطعی می‌باشند. **الگوریتم مونت‌کارلو**، مقدار مورد انتظار یک متغیر تصادفی را (که در فضای نمونه تعریف شده) از طریق مقدار میانگینش در نمونهٔ تصادفی فضای نمونه تخمین می‌زند. (بخش ۱-۸-A در ضمیمه A، راجع به فضاهای نمونه، نمونه‌های تصادفی، متغیرهای تصادفی و مقادیر مورد انتظار بحث می‌کند.) تضمینی وجود ندارد که تخمین، به مقدار مورد انتظار واقعی نزدیک باشد اما احتمال این نزدیکی، با افزایش زمان مورد دسترس الگوریتم افزایش می‌یابد.

ما می‌توانیم به ترتیب زیر، میزان کارایی یک الگوریتم بک‌تراکینگ را توسط یک الگوریتم مونت کارلو تخمین بزنیم. یک "مسیر ویژه"، در درخت شامل گره‌هایی که باید در نمونهٔ مفروض بررسی شوند، ایجاد می‌کنیم، و سپس تعداد گره‌ها را در درخت این مسیر تخمین می‌زنیم. این تخمین عبارت است از تخمین تعداد کل گره‌هایی که برای یافتن جواب‌ها بایستی بررسی شوند (به عبارت دیگر، تخمینی است از تعداد گره‌های درخت فضای حالات هرس شده). قبل از بکارگیری این روش بایستی شرایط زیر مهیا باشند:

۱- بایستی از یک تابع وعده‌گاه، برای تمامی گره‌های هم‌سطح در درخت فضای حالات استفاده شود.

۲- گره‌های هم‌سطح در درخت فضای حالات بایستی به تعداد یکسانی فرزند داشته باشند.

لازم به ذکر است که الگوریتم ۱-۵ دارای شرایط فوق می‌باشد.

در روش مونت‌کارلو باید فرزند وعده‌گاه یک گره را بطور تصادفی تولید کنیم. یعنی بایستی از یک فرآیند تصادفی برای تولید فرزند وعده‌گاه استفاده نمائیم. (برای بحث فرآیندهای تصادفی به بخش A-۸-۱ در ضمیمه A مراجعه کنید.) به هنگام پیاده‌سازی این تکنیک بر روی کامپیوتر، تنها می‌توانیم یک فرزند وعده‌گاه شبه‌تصادفی تولید کنیم. این تکنیک به صورت زیر است:

۱- فرض کنید که m_0، تعداد فرزندان وعده‌گاه ریشه باشد.

۲- یک گرهٔ وعده‌گاه را به طور تصادفی در سطح ۱ تولید کنید. فرض کنید که m_1، تعداد فرزندان وعده‌گاه این گره باشد.

۳- یک فرزند وعده‌گاه از گرهٔ بدست آمده از مرحله قبل را به طور تصادفی تولید کنید. فرض کنید که m_2، تعداد فرزندان وعده‌گاه این گره باشد.

۴- یک فرزند وعده‌گاه از گرهٔ بدست آمده از مرحله قبل را به طور تصادفی تولید کنید. فرض کنید که m_i، تعداد فرزندان وعده‌گاه این گره باشد.

این فرآیند تا آنجا ادامه می‌یابد که هیچ فرزند وعده‌گاهی پیدا نشود. از آنجائیکه فرض کردیم گره‌های هم‌سطح در درخت فضای حالات دارای تعداد یکسانی فرزند می‌باشند، لذا m_i، تخمین میانگین تعداد فرزندان وعده‌گاه گره‌های سطح i می‌باشد. فرض کنید که تعداد کل فرزندان یک گره در سطح t_i برابر i باشد. از آنجائیکه تمام فرزندان t_i یک گره بررسی می‌شوند و تنها فرزندان وعده‌گاه m_i دارای فرزندان بررسی‌شده می‌باشند، لذا تخمینی از تعداد کل گره‌های بررسی شده توسط الگوریتم بک‌تراکینگ برای یافتن تمامی جوابها به صورت زیر است:

$$1 + t_0 + m_0 t_1 + m_0 m_1 t_2 + \ldots + m_0 m_1 \ldots m_{i-1} t_i + \ldots$$

در ادامه، یک الگوریتم کلی برای محاسبه این تخمین آورده‌ایم. در این الگوریتم، یک متغیر mprod برای معرفی ضرب $m_0 m_1 \ldots m_{i-1} t_i$ در هر سطح بکار رفته است.

الگوریتم ۲-۵ تخمین مونت کارلو

مسئله: با استفاده از الگوریتم مونت کارلو، میزان کارایی یک الگوریتم بک‌تراکینگ را تخمین بزنید.

ورودی: نمونه‌ای از یک مسئله، که الگوریتم بک‌تراکینگ آن را حل می‌کند.

خروجی: تخمینی از تعداد گره‌های درخت فضای حالات هرس شده، که توسط الگوریتم تولید شده و عبارت است از تعداد گره‌هایی که الگوریتم برای یافتن تمامی جوابهای نمونه بررسی می‌کند.

```
int estimate( )
{
  node v;
  int m, mprod, t, numnodes;
```

```
    v = root of state space tree;
    numnodes = 1;
    m = 1;
    mprod = 1;
    while (m! = 0){
        t = number of children of y;
        mprod = mprod * m;
        numnodes = numnodes + mprod * t;
        m = number of promising children of v;
        if (m! = 0)
            v = randomly selected promising child of v;
    }
    return numnodes;
}
```

در ادامه، نسخهٔ خاصی از الگوریتم ۲-۵ برای الگوریتم ۱-۵ (الگوریتم بک‌تراکینگ برای مسئله n-وزیر) را ارائه می‌دهیم. پارامتر n را به این الگوریتم ارسال می‌کنیم زیرا n به عنوان پارامتر ارسالی به الگوریتم ۱-۵ محسوب می‌شود.

الگوریتم ۳-۵ تخمین مونت کارلو برای الگوریتم ۱-۵ (الگوریتم بک‌تراکینگ برای مسئله n-وزیر)

مسئله: کارایی الگوریتم ۱-۵ را تخمین بزنید.

ورودی: عدد صحیح مثبت n.

خروجی: تخمینی از تعداد گره‌های درخت فضای حالات هرس شده، که توسط الگوریتم ۱-۵ تولید شده و عبارت است از تعداد گره‌هایی که الگوریتم باید قبل از یافتن تمامی راههای ممکن برای جایگذاری n-وزیر، بررسی نماید.

```
int estimate_n_queens (int n)
{
    index i, j, col[1..n];
    int m, mprod, numnodes;
    set_of_index prom_children;
    i = 0;
    numnodes = 1;
    m = 1;
    mprod = 1;
    while (m != 0 && i != n) {
        mprod = mprod * m;
        numnodes = numnodes + mprod * n;        // number of children t is n.
        i++;
        m = 0;
        prom_children = ∅;                      // Initialize set of promising
```

```
                                        // Initialize set of promising
    for (j = 1; j <= n; j++) {          // children to empty.
      col[i] = j;
      if (promising(i)) {               // Determine promising
        m++;                            // children. Function
        prom_children = prom_children ∪ {j}; // promising is the one in
      }                                 // Algorithm 5.1.
    }
    if (m != 0) {
      j = random selection from prom_children;
      col[i] = j;
    }
  }
  return numnodes;
}
```

هنگامی که از الگوریتم مونت کارلو استفاده می‌شود، تخمین بایستی بیش از یکبار اجرا شود تا میانگین نتایج به عنوان تخمین واقعی استفاده شود. با استفاده از روشهای استاندارد آماری می‌توان فاصله مطمئنی را برای تعداد واقعی گره‌های بررسی شده، از بین نتایج آزمایشات، تعیین کرد. هر چند که احتمال بدست آوردن یک تخمین خوب با تکرار بیشتر اجراها افزایش می‌یابد، اما به عنوان یک قاعدهٔ تجربی، حدود ۲۰ آزمایش برای بدست آوردن یک تخمین خوب کافی است.

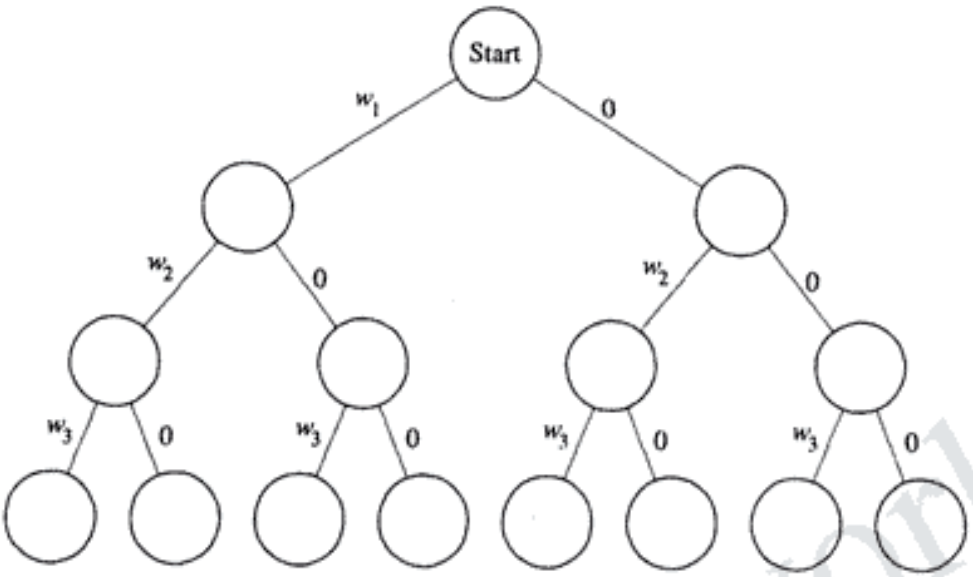
مسئله n-وزیر تنها یک نمونه برای هر مقدار از n است. البته برای بسیاری از مسائل حل‌شده توسط الگوریتم‌های بک‌تراکینگ، این چنین نمی‌باشد. تخمین تولید شده توسط هر اجرای تکنیک مونت کارلو، تنها برای یک نمونهٔ خاص است. همانطوریکه قبلاً نیز گفته شد، اگر دو نمونه با مقدار یکسان n داشته باشیم، یکی از آنها ممکن است گره‌های خیلی کمی را بررسی کند؛ درحالیکه، دیگری نیاز به بررسی کل درخت فضای حالات داشته باشد.

تخمین بدست آمده از روش مونت کارلو، لزوماً معرف خوبی برای تعداد گره‌هایی که باید بررسی شوند تا اولین جواب پیدا شود، نمی باشد. الگوریتم ممکن است برای بدست آوردن فقط یک جواب، بخش کوچکی از گره‌ها را، نسبت به زمانی که می‌خواهیم تمام جوابها را پیدا کنیم، بررسی کند. برای مثال، شکل ۴-۵ نشان می‌دهد که دو شاخه‌ای که اولین وزیر را به ترتیب در ستونهای سوم و چهارم قرار می‌دهند، نبایستی برای یافتن تنها یک جواب پیمایش شوند.

۴-۵ مسئله مجموع زیرمجموعه‌ها

مسئله دزد و کوله‌پشتی ۱-۰ در بخش ۱-۴-۴ را به خاطر آورید. در این مسئله، مجموعه‌ای از کالاها وجود دارد که هر کالا دارای وزن و قیمت خاصی است. کوله‌پشتی دزد فقط تحمل وزن W را دارد. بنابراین، هدف به حداکثر رساندن قیمت کالاهای مسروقه است بشرطی که وزن کل آنها بیش از W نشود.

شکل ۷-۵ درخت فضای حالات برای نمونه‌هایی از مسئله مجموع زیرمجموعه‌ها که در آن ۳ = n است.



در اینجا، فرض می‌کنیم که تمامی کالاها دارای قیمت در واحد وزن یکسانی می‌باشند. جواب بهینه برای دزد، مجموعه‌ای از کالاها است بطوری که وزن کل آنها به حداکثر برسد ولی وزن کل از مقدار W تجاوز نکند.

در ابتدا، ممکن است دزد سعی کند تا مشخص نماید آیا مجموعه‌ای از کالاها وجود دارد که مجموع وزن آنها برابر مقدار W شود یا خیر (زیرا این جواب، بهترین جواب است). مسئله تعیین چنین مجموعه‌هایی، مسئله مجموع زیرمجموعه‌ها نامیده می‌شود. در مسئله مجموع زیرمجموعه‌ها، n عدد صحیح مثبت w_i (وزن‌ها) و یک عدد صحیح مثبت W وجود دارد. هدف، یافتن تمامی زیرمجموعه‌های اعداد صحیح است بطوری که مجموع آنها برابر W باشد. همانطوریکه قبلاً نیز گفته شد، معمولاً مسئله را طوری مطرح می‌کنیم که همه جوابها را پیدا کنیم؛ ولی در مورد مسئله کوله‌پشتی، تنها یک جواب کافی است.

مثال ۲-۵ فرض کنید ۵ = n، ۲۱ = W، و

$$w_1 = 5,\ w_2 = 6,\ w_3 = 10,\ w_4 = 11,\ w_5 = 16$$

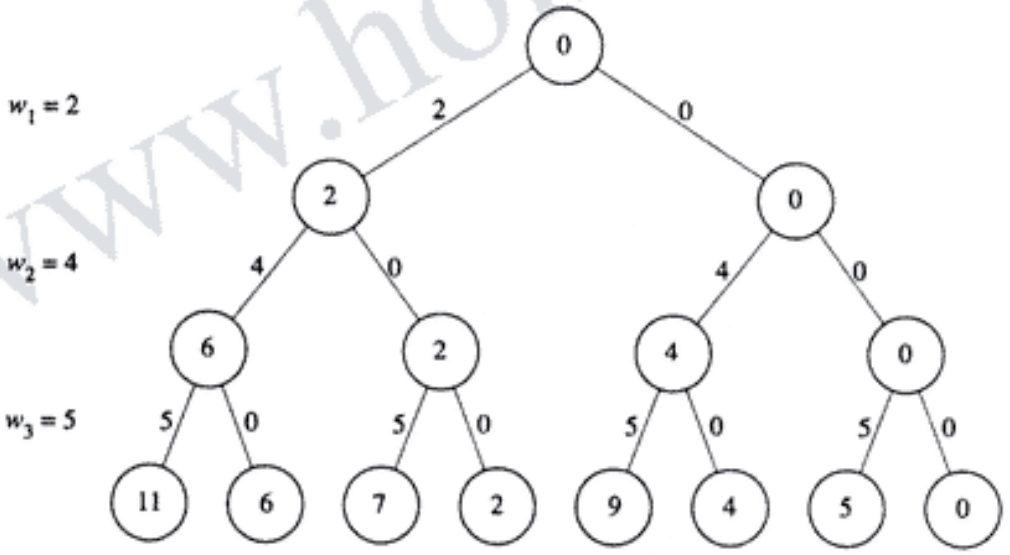
از آنجائیکه

$$w_1 + w_2 + w_3 = 5 + 6 + 10 = 21$$
$$w_1 + w_5 = 5 + 16 = 21$$
$$w_3 + w_4 = 10 + 11 = 21$$

لذا جوابها عبارتند از $\{w_1, w_2, w_3\}$، $\{w_1, w_5\}$ و $\{w_3, w_4\}$.

این نمونه را می‌توان با یک بررسی ساده حل کرد؛ اما برای مقادیر بزرگ n، نیاز به یک روش منظم و اصولی می‌باشد. یک روش، ایجاد درخت فضای حالات است. یک روش برای ساختن درخت را در شکل ۷-۵ نشان داده‌ایم. برای سادگی کار، درخت این شکل برای تنها سه وزن ساخته شده است. برای به حساب آوردن w_1، از سطح ریشه به سمت چپ می‌رویم و برای رد کردن آن، به سمت راست حرکت می‌کنیم. به طور مشابه، برای به حساب آوردن w_2، از سطح ۱ به سمت چپ می‌رویم و برای رد کردن آن، به سمت راست می‌رویم و الی آخر. هر زیرمجموعه، توسط یک مسیر از ریشه به یک برگ مشخص می‌شود. وقتی که می‌خواهیم w_i را به حساب آوریم، w_i را بر روی لبه‌ای که آن را به حساب می‌آوریم، می‌نویسیم و زمانی که نمی‌خواهیم w_i را به حساب آوریم، عدد صفر را برروی لبه می‌نویسیم.

مثال ۳-۵ شکل ۸-۵، درخت فضای حالات برای $n = 3$، $W = 6$، $w_1 = 2$، $w_2 = 4$ و $w_3 = 5$ را نشان می‌دهد. در هر گره، مجموع وزنهایی که تا آن گره به حساب آمده را نوشتیم. بنابراین، هر برگ شامل مجموع وزنهای مجموعهٔ منتهی به آن برگ می‌باشد. دومین گره از سمت چپ، تنها گره‌ای است که شامل ۶ است. از آنجائیکه یک مسیر به این برگ بیانگر زیرمجموعهٔ $\{w_1, w_2\}$ است، لذا این زیرمجموعه تنها جواب ممکن خواهد بود.



شکل ۸-۵ یک درخت فضای حالات برای مسئله مجموع زیرمجموعه‌ها برای نمونه مسئله مثال ۳-۵. مقادیر ذخیره شده در هر گره برابراست با وزن کل گره‌های قبلی (بالایی) به حساب آمده.

اگر ما وزنها را قبل از عمل جستجو به ترتیب غیرنزولی مرتب کنیم، به راحتی می‌توان گفت که کدام گره غیروعده‌گاه است. اگر وزنها براساس این روش مرتب شود، وقتی که در سطح iام هستیم، w_{i+1} کمترین وزن باقیمانده را دارد. فرض کنید weight برابر مجموع اوزان تا گره سطح i باشد. اگر w_{i+1} باعث شود که مقدار weight بیش از W شود، دراینصورت هر وزن دیگری نیز بدینصورت خواهد بود. بنابراین، بجز زمانیکه weight برابر W است (که نشاندهنده یک جواب در گره است)، یک گره در سطح iام به شرطی غیروعده‌گاه است که

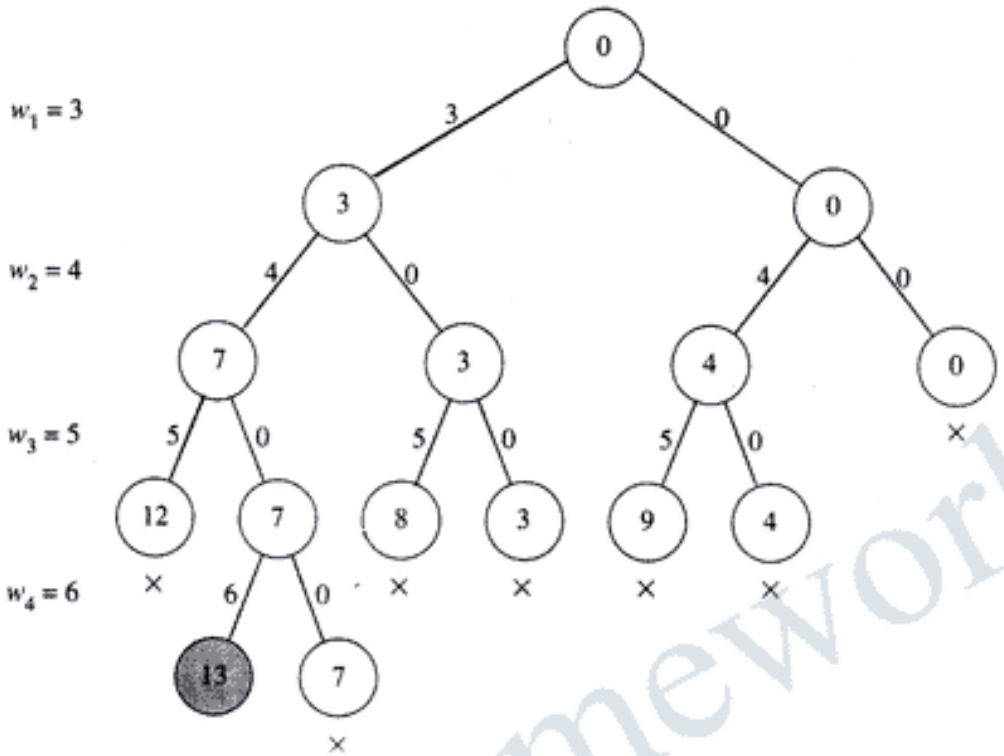
$$weight + w_{i+1} > w$$

علامت دیگری نیز وجود دارد که به ما بگوید کدام گره غیروعده‌گاه است. اگر در یک گرهٔ مفروض، افزودن مجموع وزن کالاهای باقیمانده به weight، آن را حداقل برابر W نکند، در اینصورت weight، هرگز با بسط گره برابر W نخواهد شد. این بدین معنا است که اگر total را مجموع وزن کالاهای باقیمانده فرض کنیم، یک گره به شرطی غیروعده‌گاه است که

$$weight + total > w$$

اگر مجموع وزنها تا یک گره برابر W شود، آنگاه یک جواب در آن گره وجود خواهد داشت. بنابراین، نمی‌توان جواب دیگری را با افزودن کالاهای بیشتر بدست آورد. این بدان معنی است که اگر W = weight شود، آنگاه جواب را چاپ کرده و برمی‌گردیم. این بازگشت به عقب به طور خودکار توسط روال کلی checknode انجام می‌شود زیرا در جائیکه جواب بدست آمده است، هرگز نمی‌توان گرهٔ وعده‌گاهی را بسط داد. به خاطر دارید که هنگامی که دربارهٔ checknode بحث می‌کردیم اشاره داشتیم به این نکته که برخی الگوریتم‌های بک‌تراکینگ، گاهی قبل از رسیدن به یک برگ در درخت فضای حالات، یک جواب را پیدا می‌کنند.

مثال ۴-۵ شکل ۹-۵، درخت فضای حالات هرس شده را به هنگام استفاده از بک‌تراکینگ با n = ۴، W = ۱۳، $w_1 = ۳$، $w_2 = ۴$، $w_3 = ۵$، $w_4 = ۶$ نشان می‌دهد. تنها جواب پیدا شده، در گرهٔ سایه‌دار می‌باشد که برابراست با $\{w_1, w_2, w_4\}$. گره‌های غیروعده‌گاه نیز با یک علامت × مشخص شده‌اند. گره‌های شامل اعداد ۱۲، ۸ و ۹ غیروعده‌گاه هستند زیرا افزودن وزن بعدی (۶) موجب تجاوز مقدار weight نسبت به W می‌شود. گره‌های شامل اعداد ۷، ۳، ۴ و ۰ غیروعده‌گاه هستند زیرا مجموع وزنهای باقیمانده به اندازه‌ای نیست که بتواند مقدار weight را به W برساند. توجه داشته باشید که برگی که در درخت فضای حالات شامل یک جواب نمی‌باشد، به خودی خود غیروعده‌گاه است زیرا وزنهای باقیمانده‌ای وجود ندارد که بتواند weight را به W برساند. گرهٔ شامل ۷، مبین این موضوع است. تنها ۱۵ گره در درخت فضای حالات هرس شده وجود دارد، در حالیکه کل درخت فضای حالات شامل ۳۱ گره می‌باشد.



شکل ۹-۵ درخت فضای حالات هرس شده که توسط بک‌تراکینگ مثال ۴-۵ استفاده شده است. مقدار ذخیره شده در هر گره برابراست با مجموع وزنها تا رسیدن به آن گره، تنها جواب پیدا شده، در گرهٔ سایه‌دار می‌باشد. هر گرهٔ غیروعده‌گاه با یک علامت × مشخص شده است.

در ادامه الگوریتمی را معرفی می‌کنیم که از این استراتژیها استفاده می‌کند. الگوریتم، از یک آرایهٔ include استفاده می‌کند. اگر W[i] بایستی به حساب آید، مقدار include[i] را به yes و درغیراینصورت مقدار آن را به no منتسب می‌کند.

الگوریتم ۴-۵ الگوریتم بک‌تراکینگ برای مسئله مجموع زیرمجموعه‌ها

مسئله: با n عدد صحیح مثبت (وزنها) و عدد صحیح مثبت W، تمام ترکیباتی از اعداد صحیح که مجموعشان برابر W باشد را پیدا کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای مرتب (به ترتیب غیرنزولی) از اعداد صحیح مثبت به نام w، که از ۱ تا n شاخص‌دهی شده است، و عدد صحیح مثبت W.

خروجی: تمام ترکیبات اعداد صحیح که مجموعشان برابر W است.

```
void sum_of_subsets (index i,
                     int weight, int total)
{
  if (promising(i))
    if (weight == W)
      cout << include[1] through include[i];
    else {
      include[i + 1] = "yes";                    // Include w[i + 1].
      sum_of_subsets(i + 1, weight + w[i + 1], total - w[i + 1]);
      include[i - 1] = "no";                     // Do not include w[i + 1].
      sum_of_subsets(i + 1, weight, total - w[i + 1]);
    }
}

bool promising (index i);
{
return (weight + total >= W) && (weight == W || weight + w[i + 1] <= W);
}
```

مطابق قرارداد n، w، W و include ورودیهای روتینهای ما نیستند. اگر این متغیرها به صورت سراسری تعریف می‌شدند، فراخوانی سطح بالای روال sum_of_subsets به صورت زیر بود:

sum_of_subsets(0, 0, total);

که در آن مقداردهی زیر انجام می‌شد:

$$total = \sum_{j=1}^{n} w[j]$$

حتماً به خاطر دارید برگی که در درخت فضای حالات دارای جواب نباشد، غیروعده‌گاه است. زیرا دیگر اوزانی باقی نمی‌مانند تا مقدار weight را به W برسانند. این بدان معنی است که نیازی به بررسی شرط پایانی i = n در الگوریتم نیست. می‌خواهیم صحت این موضوع را در پیاده‌سازی الگوریتم بررسی کنیم. برای i = n، مقدار total برابر صفر است (زیرا وزنی باقی نمی‌ماند). لذا

$$weight + total = weight + 0 = weight$$

یعنی $weight + total \geq w$ بشرطی صحیح است که $weight \geq w$ باشد. از آنجائیکه ما همواره weight را کوچکتر یا مساوی W نگه می‌داریم، بایستی همواره weight = W باشد. بنابراین، برای i = n، تابع promising مقدار true را به شرطی برمی‌گرداند که weight = W شود. اما در این مورد، هیچ فراخوانی بازگشتی وجود ندارد زیرا یک جواب را پیدا کردیم. لذا نیازی به بررسی شرط نهایی i = n نیست. تعداد گره‌های درخت فضای حالات جستجو شده توسط الگوریتم ۴-۵ برابراست با

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

این تساوی از مثال ۳-A در ضمیمه A بدست آمده است. تنها با این نتیجهٔ مشخص، این امکان وجود دارد که بدترین حالت، خیلی بهتر از این باشد. برای هر n، می‌توانیم نمونه‌ای بسازیم که الگوریتم آن، تعداد نمایی بزرگی از گره‌ها را ملاقات کند. این موضوع، حتی زمانی که می‌خواهیم تنها یک جواب را پیدا کنیم،

صحت دارد. برای اثبات این مورد، اگر

$$\sum_{i=1}^{n-1} w_i < W \quad , \quad w_n = W$$

باشد، آنگاه تنها یک جواب $\{w_i\}$ وجود خواهد داشت که تا زمانیکه یک تعداد نمایی از گره‌ها ملاقات نشوند، پیدا نمی‌شود. همانطوریکه قبلاً تأکید شد، حتی اگر بدترین حالت را به صورت نمایی فرض کنیم، الگوریتم برای بسیاری از نمونه‌های بزرگ کارا است. در تمرینات از شما می‌خواهیم که برنامه هایی را با استفاده از روش مونت کارلو بنویسید که کارایی الگوریتم ۴-۵ را بر روی نمونه‌های مختلف تخمین بزنند. حتی اگر مسئله‌ای را مطرح کنیم که تنها به یک جواب نیاز داشته باشد، مسئله مجموع زیرمجموعه‌ها (همانند مسئله کوله‌پشتی ۱-۰) در زمرهٔ مسائل بحث شده در فصل ۹ می‌باشد.

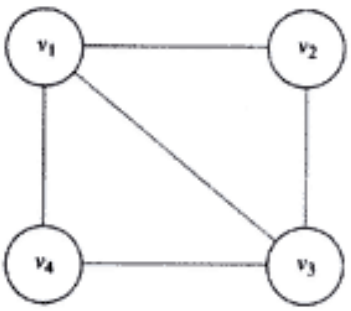
۵-۵ رنگ‌آمیزی گراف

در مسئله m-رنگ می‌خواهیم تمام راههای ممکن برای رنگ‌آمیزی یک گراف بدون‌جهت را با استفاده از حداکثر m رنگ مختلف پیدا کنیم بطوری که هیچ دو گرهٔ مجاور دارای رنگ یکسانی نباشند.

مثال ۵-۵ گراف شکل ۱۰-۵ را در نظر بگیرید. جوابی برای مسئله ۲-رنگ برای این گراف وجود ندارد زیرا اگر بتوانیم حداکثر دو رنگ مختلف را استفاده کنیم، راهی برای رنگ‌آمیزی گره‌ها، طوری که هیچ دو گرهٔ مجاور همرنگ نباشد، وجود ندارد. یک جواب مسئله ۳-رنگ برای این گراف به صورت زیر است:

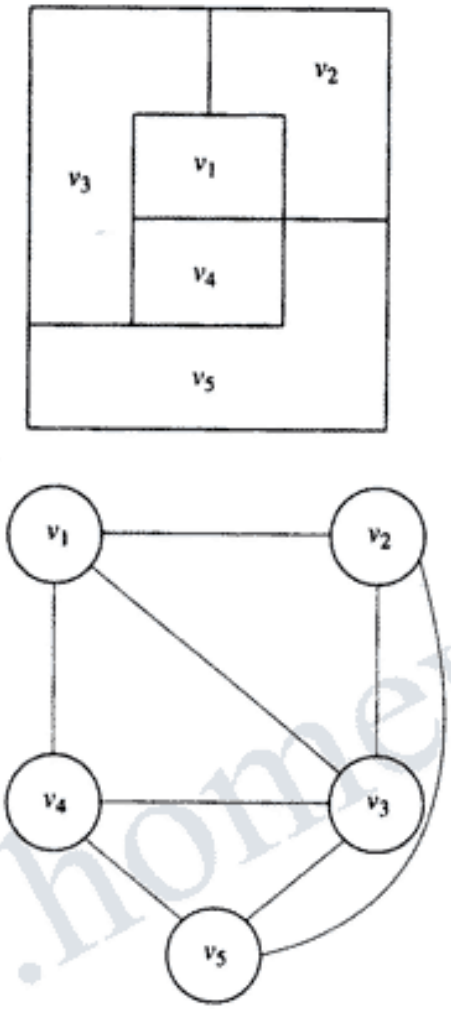
گره	رنگ
v_1	رنگ ۱
v_2	رنگ ۲
v_3	رنگ ۳
v_4	رنگ ۲

شش جواب برای مسئله ۳-رنگ این گراف وجود دارد که تفاوت این جوابها در پس و پیش شدن رنگها است. به عنوان مثال، یک جواب دیگر این است که گرهٔ v_1 را با رنگ ۲، گرهٔ v_2 و v_4 را با رنگ ۱ و گرهٔ v_3 را با رنگ ۳ رنگ‌آمیزی کنیم.



شکل ۱۰-۵ گرافی که برای آن جوابی برای مسئله ۲-رنگ وجود ندارد.

شکل ۱۱-۵ نقشه (بالا) و گراف مسطح متناظر آن (پائین).



یکی از کاربردهای مهم رنگ‌آمیزی گرافها در رنگ‌آمیزی نقشه‌ها است. یک گراف، **مسطح** است اگر بتوان آن را طوری رسم کرد که هیچ دولبه‌ای یکدیگر را قطع نکنند. گراف پائینی شکل ۱۱-۵ مسطح است؛ با وجود این، اگر می‌خواستیم لبه‌های (v_1, v_5) و (v_2, v_4) را اضافه کنیم، دیگر گراف مسطح نبود. برای هر نقشه می‌توان یک گراف مسطح متناظر پیدا کرد. هر ناحیه در نقشه با یک گره معرفی می‌شود. اگر ناحیه‌ای مجاور ناحیه‌ای دیگر باشد، گره‌های متناظر آنها را با یک لبه به هم وصل می‌کنیم. شکل ۱۱-۵، یک نقشه را در بالا و گراف مسطح متناظر با آن را در پائین نمایش می‌دهد. مسئله m-رنگ برای گرافهای مسطح عبارت است از تعیین تعداد راههایی که بتوان نقشه را با m رنگ طوری رنگ‌آمیزی کرد که هیچ دو ناحیهٔ مجاور، همرنگ نباشند.

یک درخت فضای حالات برای مسئله m-رنگ، درختی است که در آن هر یک از رنگها برای گرهٔ v_1 در سطح ۱، هر یک از رنگها برای گرهٔ v_2 در سطح ۲،...، و هر یک از رنگها برای گرهٔ v_n در سطح n

مجاور همرنگ هستند یا خیر، می‌توان جواب بودن یک جواب کاندید را مشخص کرد. در بحث زیر، به خاطر داشته باشید که **گره**، به گره‌ای در درخت فضای حالات و **رأس**، به گره‌ای در گرافی که باید رنگ‌آمیزی شود، اشاره می‌کند.

شکل ۱۲-۵، بخشی از درخت فضای حالات هرس‌شده، که به هنگام بکارگیری روش بک‌تراکینگ به مسئله ۳-رنگ گراف شکل ۱۰-۵ حاصل می‌شود را نشان می‌دهد. عدد موجود در هر گره، شمارهٔ رنگ مورد استفاده در رأسی است که در آن گره رنگ می‌شود. اولین جواب، در گرهٔ سایه‌دار پیدا می‌شود. گره‌های غیروعده‌گاه با یک علامت × مشخص شده‌اند. پس از آنکه گرهٔ v_1 با رنگ ۱ رنگ‌آمیزی شد، انتخاب رنگ ۱ برای v_2، غیروعده‌گاه است زیرا v_1 و v_2 مجاور هم هستند. به طور مشابه، پس از آنکه v_1، v_2 و v_3 به ترتیب با رنگ‌های ۱، ۲ و ۳ رنگ‌آمیزی شدند، انتخاب رنگ ۱ برای v_4 غیروعده‌گاه است زیرا v_1 و v_4 مجاور هم می‌باشند.

در ادامه الگوریتمی را معرفی می‌کنیم که مسئله m-رنگ را برای تمام مقادیر m حل می‌کند. در این الگوریتم، گراف را با یک ماتریس مجاور، همانند بخش ۱-۴، معرفی می‌کنیم. به هر حال، از آنجائیکه گراف بدون وزن است، هر ورودی در ماتریس، بر اساس آنکه لبه‌ای بین دو رأس وجود داشته باشد یا خیر، درست یا نادرست است.

الگوریتم ۵-۵ الگوریتم بک‌تراکینگ برای مسئله m-رنگ

مسئله: تمام راههایی که بتوان رئوس یک گراف بدون جهت را با m رنگ، رنگ‌آمیزی کرد، پیدا کنید. (با این شرط که هیچ دو رأس مجاور همرنگ نباشند.)

ورودی: اعداد صحیح مثبت n و m، یک گراف بدون جهت شامل n رأس. گراف را توسط یک آرایه دوبعدی w نشان داده‌ایم که سطرها و ستونهایش از ۱ تا n شاخص‌دهی شده است و w[i][j] درست است اگر لبه‌ای بین رأس iام و رأس jام وجود داشته باشد، وگرنه نادرست است.

خروجی: تمام رنگ آمیزیهای ممکن گراف با استفاده از m رنگ، بطوری که هیچ دو رأس مجاوری همرنگ نباشند. خروجی هر رنگ‌آمیزی، آرایه‌ای به نام Vcolor است که از ۱ تا n شاخص‌دهی شده و در آن Vcolor[i]، مبین رنگ (یک عدد صحیح بین ۱ تا m) اختصاص داده شده به رأس iام می‌باشد.

```
void m_coloring (index i)
{
   int color;

   if (promising(i))
      if (i == n)
         cout << vcolor[1] through vcolor[n];
      else
         for (color = 1; color <= m; color++) {    // Try every color for
            vcolor[i + 1] = color;                 // next vertex.
```

```
            m_coloring(i + 1);
        }
}

bool promising (index i)
{
   index j;
   bool switch;

   switch = true;
   j = 1;
   while (j < i && switch) {                    // Check if an adjacent
      if (W[i][j] && vcolor[i] == vcolor[j])    // vertex is already this
         switch = false;                        // color.
      j++;
   }
   return switch;
}
```

مطابق قرارداد n، m، W و vcolor ورودیهای روتینهای فوق نمی‌باشند. در یک پیاده سازی از الگوریتم، روتینها بایستی به صورت محلی در روال ساده‌ای که n، m و w را به عنوان ورودی می‌پذیرند و Vcolor نیز به عنوان یک متغیر محلی تعریف می‌شود، نوشته شوند. فراخوانی سطح بالای m_coloring بصورت

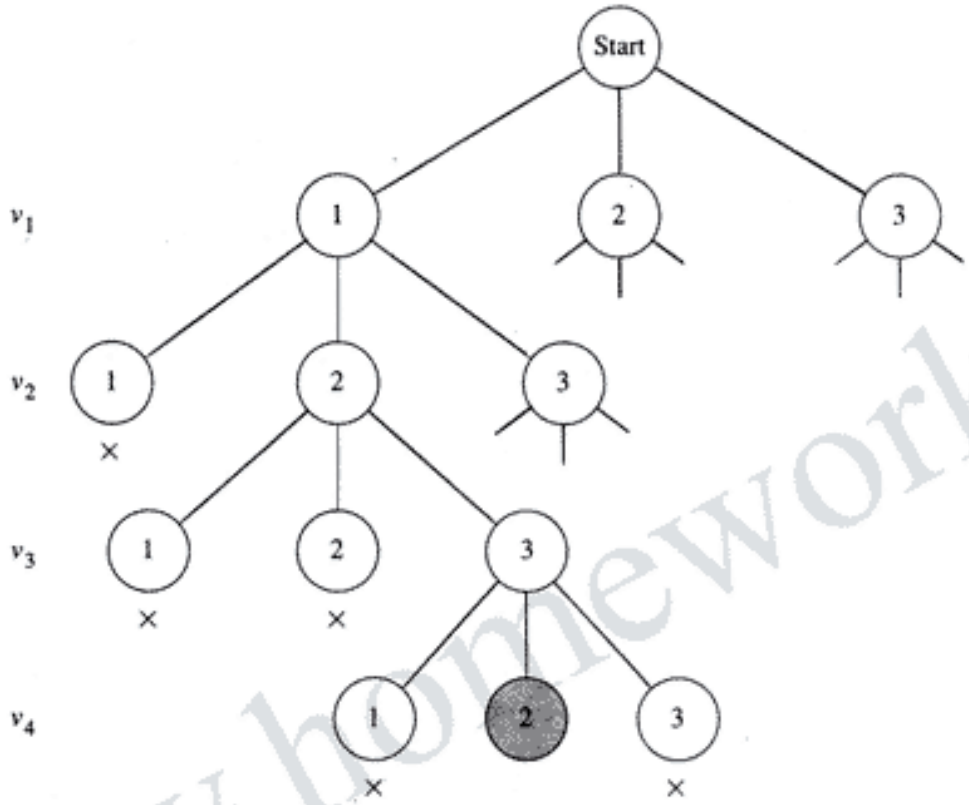
m_coloring(0);

خواهد بود. تعداد گره‌های درخت فضای حالات این الگوریتم برابر است با

$$1 + m + m^2 + \ldots + m^n = \frac{m^{n+1} - 1}{m - 1}$$

این تساوی از مثال A-۴ در ضمیمه A بدست آمده است. برای مقادیر معین n و m، امکان ایجاد نمونه‌ای که حداقل تعداد بزرگ نمایی (بر حسب n) از گره‌ها را بررسی کند، وجود دارد. به عنوان مثال، اگر m فقط برابر ۲ باشد و ما گرافی داشته باشیم که در آن v_n لبه‌ای به هر گرهٔ دیگر داشته باشد و تنها گرهٔ دیگر بین v_{n-2} و v_{n-1} قرار داشته باشد، دراینصورت هیچ جوابی وجود ندارد اما تقریباً تمام گره‌های درخت فضای حالات بایستی برای تعیین این موضوع ملاقات شوند. همانند هر الگوریتم بک‌تراکینگ ، الگوریتم برای یک نمونه بزرگ خاصی کارا است. روش مونت کارلو که در بخش ۳-۵ بحث شد، قابل استفاده برای این الگوریتم است یعنی می‌توان از آن برای تخمین میزان کارایی نمونه‌ای خاص استفاده کرد. در تمرینات از شما می‌خواهیم که مسئله ۲-رنگ را با الگوریتمی حل کنید که پیچیدگی زمانی بدترین حالت آن، نمایی از n نباشد. برای $m \geq 3$، تاکنون کسی نتوانسته الگوریتمی بنویسید که در بدترین حالت، کارا باشد. همانند مسئله مجموع زیرمجموعه‌ها و مسئله کوله‌پشتی ۱-۰، مسئله m-رنگ برای $m \geq 3$ نیز در زمرهٔ مسائلی که در فصل ۹ بحث می‌شوند، قرار دارد.

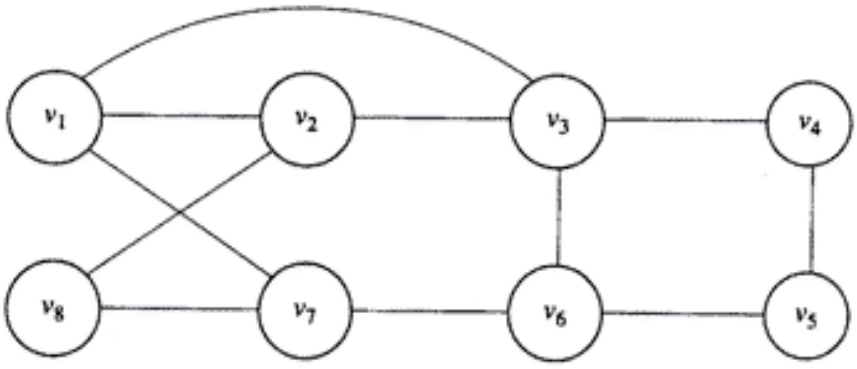
شکل ۱۲-۵ بخشی از درخت فضای حالات هرس شده که از بکارگیری بکتراکینگ برای مسئله ۳-رنگ برای گراف شکل ۱۰-۵ بدست آمده است. اولین جواب، در گرهٔ سایه دار پیدا می‌شود. هر گرهٔ غیروعده‌گاه با یک علامت × مشخص شده است.



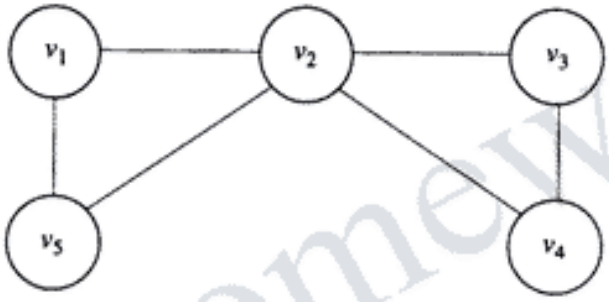
۶-۵ مسئله چرخه‌های هامیلتونی

مثال ۱۲-۳، که در آن نانسی و رالف برای تصاحب شغل فروش با هم رقابت می‌کردند را به خاطر آورید. کسی که می‌توانست تمام ۲۰ شهر در منطقهٔ فروش را سریعتر بپیماید، آن شغل را تصاحب می‌کرد. پیچیدگی زمانی یک الگوریتم برنامه‌نویسی پویا برای آن مسئله، $T(n) = (n-1)(n-2)\,2^{n-3}$ می‌باشد. نانسی توانست کوتاهترین مسیر را در عرض ۴۵ ثانیه پیدا کند، در حالیکه رالف سعی کرد تمام !۱۹ تور ممکن را آزمایش کند. از آنجائیکه الگوریتم رالف بیش از ۳۸۰۰ سال طول میکشد، پس هنوز هم در حال اجرا است و البته نانسی آن شغل را تصاحب کرده است. فرض کنید که او بقدری کارش را خوب انجام می‌دهد که رئیس حوزه کاری او را دو برابر می‌کند و به ۴۰ شهر افزایش می‌دهد. اما در این حوزه، هر شهری به شهر دیگر توسط یک جاده متصل نیست. به خاطر آورید که فرض کرده بودیم الگوریتم برنامه‌نویسی پویای نانسی برای انجام عمل مبنایی‌اش یک میکروثانیه زمان صرف می‌کند. با یک محاسبه سریع می‌توان نشان داد که این الگوریتم به میزان $(40-2)(40-1)\,2^{40-3}$ میکروثانیه یعنی ۶/۴۶ سال زمان

شکل ۱۳-۵ گراف (a) شامل چرخهٔ هامیلتونی $[v_1, v_3, v_4, v_5, v_6, v_7, v_8, v_2, v_1]$ می‌باشد وگراف (b) دارای هیچ چرخهٔ هامیلتونی نیست.



(a)



(b)

نیاز دارد تا بتواند کوتاهترین تور را در منطقه‌ای شامل ۴۰ شهر پیدا کند. روشن است که نانسی باید به دنبال الگوریتمی دیگر باشد. او می‌گوید احتمالاً یافتن یک تور بهینه بسیار مشکل است و به همین دلیل به یافتن هر توری راضی می‌شود. اگر بین هر شهر و شهر دیگر، جاده‌ای وجود می‌داشت، هر پس و پیش کردن بین شهرها موجب تشکیل یک تور می‌شد. اما در منطقه کاری جدید نانسی این چنین نیست. بنابراین، اکنون مسئله او یافتن هر توری در گراف می‌باشد. این مسئله به مسئلهٔ چرخه(مدار) هامیلتونی معروف است که توسط شخصی به نام ویلیام هامیلتون مطرح شد. این مسئله را می‌توان هم برای یک گراف جهت‌دار (که تحت عنوان مسئله فروشندهٔ دوره‌گرد معرفی شد) و هم برای یک گراف بدون جهت بیان کرد. از آنجائیکه این مسئله معمولاً برای گراف بدون جهت بکار برده می‌شود، لذا گرافها را بدون جهت در نظر می‌گیریم. در مورد معمای نانسی، یک گراف بدون‌جهت بدان معنی است که یک جادهٔ دو طرفه بین شهرهایی که به هم متصل هستند، وجود دارد.

در یک گراف متصل بدون‌جهت، یک **چرخهٔ هامیلتونی** (که یک تور نیز نامیده می‌شود) مسیری است که از یک گره شروع شده و تمام گره‌های گراف را دقیقاً یکبار ملاقات می‌کند و در نهایت به گرهٔ اول ختم می‌شود. گراف شکل ۱۳-۵(a)، شامل چرخهٔ هامیلتونی $[v_1, v_3, v_4, v_5, v_6, v_7, v_8, v_2, v_1]$ است اما شکل ۱۳-۵(b) دارای چرخهٔ هامیلتونی نمی‌باشد. یک درخت فضای حالات برای این م

به صورت زیر است

* گرهٔ آغازین را در سطح ۰ درخت قرار داده و آن را گرهٔ صفرم مسیر نامگذاری می‌کنیم.
* در سطح ۱، هر گره‌ای بجز گرهٔ آغازین را به عنوان اولین گره بعد از گرهٔ آغازین در نظر می‌گیریم.
* در سطح ۲، هر یک از این گره‌های مشابه را به عنوان گرهٔ دوم در نظر می‌گیریم و الی آخر.
* سرانجام در سطح ۱ - n، هر یک از این گره‌های مشابه را به عنوان گرهٔ ۱-nام در نظر می‌گیریم.

ملاحظات زیر ما را قادر می‌سازد تا در این درخت فضای حالات به عقب برگردیم :

۱ - iامین گره مسیر بایستی مجاور ۱-iامین گرهٔ مسیر باشد.

۲ - ۱-nامین گره بایستی مجاور گرهٔ صفرم (گرهٔ آغازین) باشد.

۳ - گرهٔ iام نمی‌تواند یکی از ۱-iگرهٔ اول باشد.

الگوریتم ۶-۵ الگوریتم بک‌تراکینگ برای مسئله چرخه‌های هامیلتونی

مسئله: کلیهٔ چرخه‌های هامیلتونی در یک گراف متصل بدون‌جهت را تعیین کنید.

ورودی: عدد صحیح مثبت n و یک گراف بدون جهت با n گره. گراف را توسط یک آرایه دوبعدی W که سطرها و ستونهایش از ۱ تا n شاخص‌دهی شده‌است، نشان می‌دهیم و در آن W[i][j]، در صورتی درست(true) است که لبه‌ای بین گرهٔ iام و گرهٔ jام وجود داشته باشد و در غیراینصورت، نادرست(false) خواهد بود.

خروجی: تمام مسیرهایی که از یک گرهٔ معین شروع می‌شوند و پس از یکبار ملاقات هر گره در گراف، به گرهٔ آغازین منتهی می‌شوند. خروجی هر مسیر، آرایه‌ای از شاخصها به نام vindex است که از ۰ تا ۱-n شاخص‌دهی شده است و vindex[i] مبین شاخص گرهٔ iام در مسیر می‌باشد. شاخص گرهٔ آغازین در vindex[0] وجود دارد.

```
void hamiltonian (index i)
{
  index j;

  if (promising(i)
    if (i == n - 1)
      cout << vindex[0] through vindex[n - 1];
    else
      for (j = 2; j <= n; j++) {                    // Try all vertices as
        vindex[i + 1] = j;                          // next one.
        hamiltonian(i + 1);
      }
}
```

```
bool promising (index i)
{
   index j;
   bool switch;

   if (i == n - 1 && ! W[vindex[n - 1]] [vindex[0]])  // First vertex must
      switch = false;                                 // be adjacent to
   else if (i > 0 && ! W[vindex[i - 1]][vindex[i]])   // last. ith vertex
      switch = false;                                 // must be adjacent
   else {                                             // to (i - 1)st.
      switch = true;
      j = 1;
 while (j < i && switch) {                            // Check if vertex is
         if (vindex[i] == vindex[j])                  // already selected.
            switch = false;
         j++;
      }
   }
   return switch;
}
```

طبق قرارداد، n، W و Vindex ورودیهای روتین نمی‌باشند. اگر این متغیرها به صورت سراسری تعریف شوند، فراخوانی سطح بالای hamiltonian به صورت زیر می‌باشد :

```
vindex[0] = 1;
hamiltonian(0);
```

تعداد گره‌های درخت فضای حالات این الگوریتم برابر است با

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

که بسیار بدتر از نمایی است. این تساوی از مثال A-۴ در ضمیمه A بدست آمده است. هر چند که نمونهٔ زیر کل درخت فضای حالات را بررسی نمی‌کند، اما تعداد گره‌هایی که بررسی می‌کند، بدتر از نمایی است. فرض کنید که تنها لبه متصل به گرهٔ V_1 از طرف V_2 باشد و همه گره‌ها به جز گرهٔ V_1 توسط لبه‌ای بهم وصل باشند. برای این گراف چرخه هامیلتونی وجود ندارد و الگوریتم برای پی بردن به این موضوع بایستی تعداد گره‌هایی بدتر از نمایی را بررسی کند.

به مسئله نانسی برمی‌گردیم. این احتمال وجود دارد که الگوریتم بک‌تراکینگ (برای مسئله چرخه‌های هامیلتونی) حتی طولانی‌تر از الگوریتم برنامه‌نویسی پویا (برای مسئله فروشندهٔ دوره‌گرد) جهت حل نمونه مسئله ۴۰ شهر باشد. چون شرایط استفاده از روش مونت کارلو در این مسئله وجود دارد، لذا نانسی

زمان یافتن تمامی چرخه‌ها را تخمین می‌زند. از آنجائیکه نانسی تنها به یک چرخه نیاز دارد، لذا می‌تواند با یافتن اولین چرخه (اگر وجود داشته باشد) الگوریتم را متوقف کند. به او توصیه می‌کنیم که یک نمونه را برای $n = 40$ ساخته و سرعت الگوریتم را برای یافتن تمامی چرخه‌ها در نمونه تخمین بزند. در نهایت الگوریتم را برای یافتن یک چرخه اجرا کند. حتی اگر تنها یک تور را بخواهیم پیدا کنیم، مسئله چرخه‌های هامیلتونی در زمرهٔ مسائل مطرح شده در فصل ۹ قرار دارد.

۵-۷ مسئله کوله‌پشتی ۱-۰

در بخش ۴-۴، این مسئله را به وسیله برنامه‌نویسی پویا حل کردیم. در اینجا، آن را با استفاده از بک‌تراکینگ حل می‌نمائیم و بعد از آن دو الگوریتم بک‌تراکینگ و برنامه نویسی پویا را با هم مقایسه می‌کنیم.

۵-۷-۱ یک الگوریتم بک‌تراکینگ برای مسئله کوله‌پشتی ۱-۰

به خاطر دارید که در این مسئله، مجموعه‌ای از کالاها داشتیم که هر یک از آنها دارای وزن و ارزشی معین بودند. وزن و ارزش آنها را اعدادی صحیح و مثبت در نظر گرفتیم. دزدی می‌خواهد کالاها را درون کوله‌پشتی قرار دهد؛ با علم به اینکه اگر مجموع وزن کالاهای داخل کوله‌پشتی بیش از عدد صحیح مثبت W شود، کوله‌پشتی باز می‌شود. هدف دزد این است که مجموعه‌ای از کالاها را انتخاب نماید که ارزش آنها ماکزیمم شود، و در عین حال مجموع وزن آنها بیش از مقدار W نشود.

ما می‌توانیم این مسئله را با استفاده از درخت فضای حالات، همانطوریکه برای مسئله مجموع زیرمجموعه‌ها استفاده نمودیم، حل کنیم. یعنی از ریشه به چپ حرکت می‌کنیم تا اولین کالا را به حساب آوریم و به راست حرکت می‌کنیم تا آن را به حساب نیاوریم. به طور مشابه، در سطح ۱ به سمت چپ می‌رویم تا کالای دوم را به حساب آوریم و به سمت راست می‌رویم تا آن را به حساب نیاوریم و به همین ترتیب الی آخر. هر مسیر از ریشه به یک برگ، یک جواب کاندید است.

این مسئله، متفاوت از مسائل دیگری است که در این فصل بررسی شدند. بدین ترتیب که تا زمانیکه عمل جستجو تمام نشده، نمی‌توانیم بفهمیم که آیا یک گره شامل یک جواب است یا خیر. بنابراین، با اندکی تفاوت، عمل بک‌تراکینگ را انجام می‌دهیم. اگر مجموع ارزش گره‌های به حساب آمده بیش از بهترین جوابی باشد که تاکنون بدست آورده‌ایم، آنگاه مقدار بهترین جواب را به این مقدار جدید تغییر می‌دهیم. با وجود این، هنوز هم ممکن است جواب بهتری در فرزندان گره (با دزدیدن کالاهای بیشتر) پیدا کنیم. بنابراین، برای مسائل بهینه‌سازی، همواره فرزندان یک گرهٔ وعده‌گاه را ملاقات می‌کنیم. الگوریتم زیر، یک الگوریتم کلی برای بک‌تراکینگ در مورد مسائل بهینه‌سازی است :

```
void checknode(node v)
{
  node u;
  if (value (v) is better than best)
    best = value(v);
  if (promising (v))
    for (each child u of v)
      checknode(u);
}
```

متغیر best حاوی بهترین جوابی که تاکنون پیدا شده است و value(v) مقدار جواب در گرهٔ v می‌باشد. پس از آنکه best به مقداری که بدتر از مقدار هر جواب کاندید است، مقداردهی اولیه شد، آنگاه ریشه در بالاترین سطح، ارسال می‌شود. توجه داشته باشید که یک گره تنها در صورتی وعده‌گاه است که آن را به فرزندانش گسترش دهیم. همچنین الگوریتمهای ما در صورتی یک گره را وعده‌گاه می‌نامند که یک جواب در گره وجود داشته باشد.

در ادامه ما از این شیوه برای مسئله کوله‌پشتی ۱-۰ استفاده می‌کنیم. ابتدا به علائمی که به ما می‌گویند یک گره غیروعده‌گاه است نگاهی می‌اندازیم. یک علامت آشکار این است که یک گره در صورتی غیروعده‌گاه است که دیگر هیچ ظرفیت باقیمانده‌ای در کوله‌پشتی وجود نداشته باشد تا اقلام بیشتری را در آن قرار دهیم. لذا اگر weight، مجموع وزن کالاهایی باشد که تاکنون به گره‌ای اضافه شده‌اند، دراینصورت آن گره به شرطی غیروعده‌گاه است که $weight \geq w$ باشد. حتی اگر weight مساوی W شود نیز آن گره غیروعده‌گاه است زیرا در مسائل بهینه سازی، «وعده‌گاه» یعنی اینکه ما باید آن گره را به فرزندانش بسط دهیم.

همانطوری که قبلاً نیز دیدیم، روش حریص در ارائه یک جواب بهینه برای چنین مسئله‌ای با شکست مواجه شد (بخش ۴-۴). در اینجا فقط از ملاحظات حریص برای محدود کردن عمل جستجو استفاده می‌کنیم. ولی نمی‌خواهیم یک الگوریتم حریص بنویسیم. برای این منظور ابتدا کالاها را بر اساس مقادیر p_i/w_i و به صورت غیرنزولی مرتب می‌کنیم. p_i، ارزش و w_i، وزن کالای iام می‌باشد. می‌خواهیم تعیین کنیم که آیا یک گرهٔ خاص، وعده‌گاه است یا خیر. به ترتیب زیر می‌توانیم حد بالایی را برای ارزشی که از بسط گره بدست می‌آید، محاسبه کنیم. فرض کنید profit، مجموع ارزش کالاهایی باشد که تا آن گره به حساب آمده‌اند. به خاطر دارید که weight، مجموع اوزان آن کالاها بود. ما متغیرهای bound و totweight را به ترتیب با profit و weight مقداردهی اولیه می‌کنیم. سپس حریصانه کالاها را برمی‌داریم و ارزش آنها را به bound و وزن آنها را به totweight اضافه می‌کنیم تا اینکه به کالایی برسیم که اگر برداشته شود، مقدار totweight از میزان W تجاوز می‌کند. با توجه به وزن مجاز باقیمانده، بخشی از آن کالا را برمی‌داریم و ارزش آن جزء را به bound اضافه می‌کنیم. اگر بتوانیم تنها بخشی از این آخرین وزن را برداریم، گره منجر به برابری ارزش و bound نمی‌شود اما bound، همچنان یک حد بالا بر ارزشی است که می‌توانیم با

است که موجب تجاوز مجموع اوزان از مرز W می‌شود. در اینصورت

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = (profit + \sum_{j=i+1}^{k-1} p_j) + (W - totweight) \times (p_K / w_k)$$

ارزش واحد وزن ظرفیت موجودی ارزش اولین ۱-k
کالای kام برای کالای kام کالای گرفته شده

اگر maxprofit مقدار ارزش بهترین جوابی باشد که تاکنون پیدا شده است، در این صورت یک گره در سطح i به شرطی غیروعده‌گاه است که $bound \leq maxprofit$ باشد. ما از ملاحظات حریص تنها برای این استفاده کردیم که به ما بگوید آیا باید گره‌ای بسط داده شود یا خیر. هدف ما این نیست که به وسیلهٔ آن، کالاها را حریصانه طوری برداریم که دیگر هیچ فرصت تجدیدنظری در آینده وجود نداشته باشد (همانگونه که در روش حریص انجام می‌شد).

مثال ۶-۵ فرض کنید که n = ۴، W = ۱۶ و

i	p_i	w_i	p_i/w_i
۱	$۴۰	۲	$۲۰
۲	$۳۰	۵	$۶
۳	$۵۰	۱۰	$۵
۴	$۱۰	۵	$۲

کالاها را از قبل، بر اساس p_i/w_i مرتب کرده‌ایم. برای سادگی کار، مقادیری از P_i و w_i را انتخاب می‌کنیم که نسبت p_i/w_i، یک مقدار صحیح باشد؛ البته چنین چیزی ضرورت ندارد. شکل ۱۴-۵، یک درخت فضای حالات هرس‌شده که توسط ملاحظات قبلی بک‌تراکینگ تولید شده را نشان می‌دهد. ارزش کل، وزن کل و حد، در هر گره‌ای از بالا به پائین مشخص شده است. اینها همان مقادیر متغیرهای profit، weight و bound، که در بحث قبلی ذکر شدند، می‌باشند. ارزش ماکزیمم، در گرهٔ سایه‌دار پیدا می‌شود. هر گره با سطح آن و موقعیت آن از سمت چپ در درخت مشخص شده است. به عنوان مثال، به گرهٔ سایه‌دار برچسب (۳،۳) زده شده است زیرا در سطح ۳ قرار دارد و سومین گره از سمت چپ در آن سطح می‌باشد. بعداً مراحلی که موجب تولید یک درخت هرس‌شده می‌شوند را معرفی می‌کنیم. در این مراحل، با برچسب یک گره به آن رجوع می‌کنیم.

۱- به maxprofit مقدار $۰ دهید.

۲- گرهٔ (۰،۰) را ملاقات کنید (ریشه).

(a) ارزش و وزن آن را محاسبه کنید.

$$profit = \$۰$$

$$weight = ۰$$

(b) حد آن را محاسبه نمائید. چون ۱۷ = ۱۰ + ۵ + ۲ و ۱۶ < ۱۷ (مقدار W برابر ۱۶ است)،لذاافزودن کالای سوم موجب می‌شود که مجموع اوزان از حد W تجاوز کند. بنابراین، ۳ = k و داریم :

$$totweight = weight + \sum_{j=0+1}^{3-1} w_j = 0+2+5 = 7$$

$$bound = profit + \sum_{j=0+1}^{3-1} p_j + (w - totweight) \times \frac{p_3}{w_3}$$

$$= \$0 + \$40 + \$30 + (16-7) \times \frac{\$50}{10} = \$115$$

(c) آن را به عنوان یک گرهٔ وعده‌گاه تعیین کنید زیرا وزن آن (۰) کمتر از ۱۶ (مقدار W) است و حد آن ۱۱۵$ بزرگتراز ۰$ (مقدار maxprofit) می‌باشد.

۳- گرهٔ (۱، ۱) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید.

$$profit = \$0 + \$40 = \$40$$

$$weight = 0+2 = 2$$

(b) چون وزن آن (۲) کوچکتر یا مساوی ۱۶ (مقدار W) است و ارزش آن ($۴۰) بیشتر از ۰$ است، لذا مقدار maxprofit را برابر ۴۰$ قرار دهید.

(c) حد آن را محاسبه نمائید. چون ۱۷ = ۱۰ + ۵ + ۲ و ۱۶ < ۱۷ (۱۶ مقدار W است)، لذا افزودن کالای سوم باعث می‌شود که مجموع اوزان بیش از مقدار W شود. بنابراین ۳ = k و داریم :

$$totweight = weight + \sum_{j=1+1}^{3-1} w_j = 2+5 = 7$$

$$bound = profit + \sum_{j=1+1}^{3-1} p_j + (w - totweight) \times \frac{p_3}{w_3}$$

$$= \$40 + \$30 + (16-7) \times \frac{\$50}{10} = \$115$$

(d) آن را به عنوان یک گرهٔ وعده‌گاه تعیین کنید زیرا وزن آن (۲) کمتر از ۱۶ (مقدار W) و حد آن ($۱۱۵) بیشتر از ۰$ (مقدار maxprofit) است.

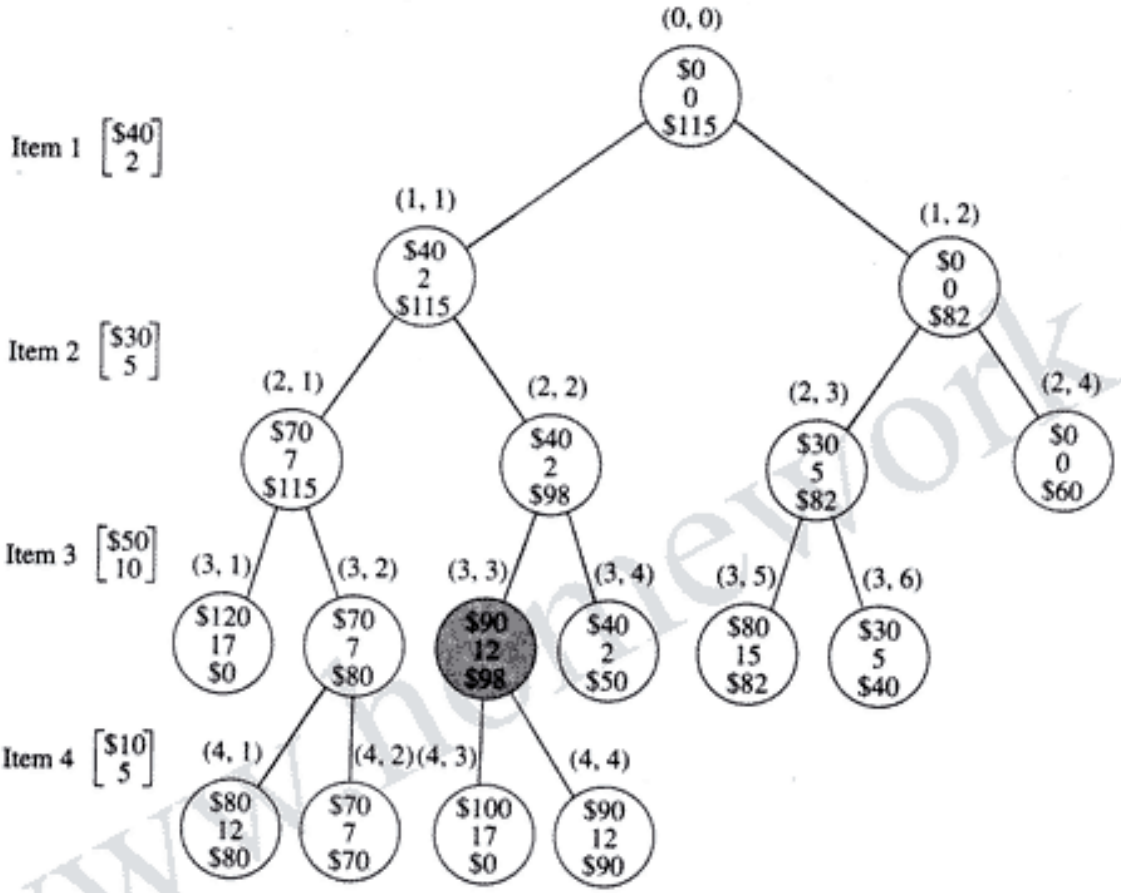
۴- گرهٔ (۱، ۲) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید.

$$totweigth = weight + \sum_{j=2+1}^{3-1} w_j = 7$$

$$bound = \$70 + (16-7) \times \frac{\$50}{10} = \$115$$

شکل ۱۴-۵ درخت فضای حالات هرس شده که با استفاده از بک‌تراکینگ در مثال ۶-۵ تولید شده است. مقادیر ذخیره شده در هر گرهٔ از بالا به پائین بترتیب عبارتند از: مجموع ارزش کالاهای تا آن گره، وزن کل آنها و حد بالای ارزشی که می‌توان با بسط گره بدست آورد. بهترین جواب، در گرهٔ سایه‌دار وجود دارد. هر گرهٔ غیروعده‌گاه با علامت × مشخص شده است.



(b) آن را بعنوان یک گرهٔ وعده‌گاه تعیین کنید زیرا وزن آن (۷) کمتر از ۱۶ (مقدار W) است و حد آن ($۱۱۵) بیش از $۷۰ (مقدار maxprofit) است.

۵- گرهٔ (۱، ۳) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه نمائید.

$$profit = \$۷۰ + \$۵۰ = \$۱۲۰$$

$$weight = ۷ + ۱۰ = ۱۷$$

(b) از آنجائیکه وزن آن (۱۷) بیش از ۱۶ (مقدار W) است، لذا مقدار maxprofit تغییر نمی‌کند.

(c) آن را بعنوان یک گرهٔ غیروعده‌گاه تعیین کنید زیرا وزن آن (۱۷) بیشتر یا مساوی ۱۶ (وزن W) است.

(d) حدی برای این گره محاسبه نمی‌شود زیرا وزن گره، آن را بعنوان یک گرهٔ غیروعده‌گاه تعیین نموده است.

۶- به گرهٔ (۲، ۱) برگردید.

۷- گرهٔ (۳، ۲) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید. چون کالای ۳ را به حساب نمی‌آوریم، لذا

$$profit = \$۷۰$$

$$weight = ۷$$

(b) چون ارزش آن ($۷۰) کوچکتر یا مساوی $۷۰ (مقدار maxprofit) است، maxprofit تغییری نمی‌کند.

(c) حد آن را محاسبه نمائید. وزن کالای چهارم باعث تجاوز مجموع اوزان از مرز W نمی‌شود و تنها ۴ کالا موجود می‌باشد. لذا k = ۵ و

$$bound = profit + \sum_{j=۳+۱}^{۵-۱} p_j = \$۷۰ + \$۱۰ = \$۸۰$$

(d) آن را بعنوان یک گرهٔ وعده‌گاه در نظر بگیرید زیرا وزن آن (۷) کوچکتر از ۱۶ (مقدار W) است و حد آن ($۸۰) بیش از $۷۰ (مقدار maxprofit) است.

(از این پس محاسبهٔ ارزشها، اوزان و حدود را بعنوان تمرین به شما وامی‌گذاریم. علاوه بر این، مقدار maxprofit هیچ تغییری نمی‌کند؛ لذا از بیان مجدد آن خودداری می‌کنیم.)

۸- گرهٔ (۴، ۱) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب برابر $۸۰ و ۱۲ شوند.

(b) چون وزن آن کوچکتر یا مساوی ۱۶ (مقدار W) و ارزش آن ($۸۰) بزرگتر از $۷۰ (مقدار maxprofit) است، لذا مقدار $۸۰ را به maxprofit تخصیص دهید.

(c) حد آن را محاسبه نمائید تا برابر $۸۰ شود.

(d) آن را به عنوان یک گرهٔ غیروعده‌گاه تعیین کنید زیرا حد آن ($۸۰) کوچکتر یا مساوی $۸۰ (مقدار maxprofit) است. برگهای درخت فضای حالات بخودی خود غیروعده‌گاه هستند زیرا حدود آنها همواره کوچکتر یا مساوی maxprofit است.

۹- به گرهٔ (۳، ۲) برگردید.

۱۰- گرهٔ (۴، ۲) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب برابر $۷۰ و ۷ شوند

(b) حد آن را محاسبه نمائید تا برابر $۷۰ شود.

(c) آن را بعنوان یک گرهٔ غیروعده‌گاه مشخص کنید زیرا حد آن ($۷۰) کوچکتر یا مساوی $۸۰ (مقدار maxprofit) است.

۱۱- به گرهٔ (۱، ۱) برگردید.

۱۲- گرهٔ (۲، ۲) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب برابر $۴۰ و ۲ شوند.

(b) حد آن را محاسبه نمائید تا برابر $۹۸ شود.

(c) آن را بعنوان یک گرهٔ وعده‌گاه تعیین کنید زیرا وزن آن (۲) کمتر از ۱۶ (مقدار W) و حد آن ($۹۸) بیشتر از $۸۰ (مقدار maxprofit) می‌باشد.

۱۳- گرهٔ (۳، ۳) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب برابر $۹۰ و ۱۲ شوند.

(b) چون وزن آن (۱۲) کوچکتر یا مساوی ۱۶ (مقدار W) و ارزش آن ($۹۰) بزرگتر از $۸۰ (مقدار maxprofit) است، لذا مقدار $۹۰ را به maxprofit تخصیص دهید.

(c) حد آن را محاسبه نمائید تا برابر $۹۸ شود.

(d) آن را بعنوان یک گرهٔ وعده‌گاه مشخص کنید زیرا وزن آن (۱۲) کمتر از ۱۲ (مقدار W) و حد آن ($۹۸) بیشتر از $۹۰ (مقدار maxprofit) است.

۱۴- گرهٔ (۴، ۳) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب مقادیر $۱۰۰ و ۱۷ بدست آیند.

(b) آن را به عنوان غیروعده‌گاه تعیین کنید زیرا وزن آن (۱۷) بیش از ۱۶ (مقدار W) است.

(c) حدی برای این گره محاسبه نمی‌شود زیرا وزن گره، آنرا بعنوان یک گرهٔ غیروعده‌گاه تعیین نموده است.

۱۵- به گرهٔ (۳، ۳) برگردید.

۱۶- گرهٔ (۴، ۴) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب مقادیر $۹۰ و ۱۲ بدست آیند.

(b) حد آن را محاسبه نمائید تا مقدار $۹۰ بدست آید.

(c) آن را بعنوان یک گرهٔ غیروعده‌گاه تعیین کنید زیرا حد آن ($۹۰) کوچکتر یا مساوی $۹۰ (مقدار maxprofit) است.

۱۷- به گرهٔ (۲، ۲) برگردید.

۱۸- گرهٔ (۳، ۴) را ملاقات کنید.

(a) ارزش و وزن آن را حساب کنید تا بترتیب برابر $۴۰ و ۲ شوند.

(b) حد آن را محاسبه نمائید تا برابر $۵۰ شود.

(c) آن را بعنوان یک گرهٔ غیروعده‌گاه تعیین کنید زیرا حد آن ($۵۰) کوچکتر یا مساوی $۹۰ (مقدار maxprofit) است.

۱۹- به ریشه برگردید.

۲۰- گرهٔ (۲، ۱) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا برابر $۵۰ و ۰ شوند.

(b) حد آن را محاسبه نمائید تا برابر $۸۲ شود.

(c) آن را بعنوان یک گرهٔ غیروعده‌گاه تعیین کنید زیرا حد آن ($۸۲) کوچکتر یا مساوی $۹۰ (مقدار maxprofit) است.

۲۱- به ریشه برگردید.

(a) ریشه فرزند دیگری ندارد. کار ما تمام شده است.

تنها ۱۳ گره در درخت فضای حالات وجود دارد؛ در حالیکه کل درخت فضای حالات شامل ۳۱ گره می‌باشد.

از آنجائیکه این مسئله، یک مسئله بهینه سازی است، ما ردّ بهترین مجموعه کنونی کالاها و مجموع ارزش آنها را نگه می‌داریم که اینکار توسط آرایه bestset و یک متغیر maxprofit انجام می‌شود. بر خلاف مسائل دیگر این فصل، این مسئله را برای یافتن اولین جواب بهینه مطرح می‌کنیم.

الگوریتم ۷-۵ الگوریتم بک‌تراکینگ برای مسئله کوله‌پشتی ۱-۰

مسئله: n کالا داریم که هر کدام از آنها دارای ارزشی و وزنی دارند. ارزش‌ها و وزنها، اعدادی صحیح و مثبت هستند. مجموعه‌ای از کالاها با حداکثر مجموع ارزش را طوری تعیین کنید که مجموع اوزان آنها بیش از عدد صحیح مثبت W نشود.

ورودی: اعداد صحیح مثبت n و W، آرایه‌های w و p که از ۱ تا n شاخص‌دهی شده و هر یک شامل اعداد صحیح و مثبتی هستند که بترتیب غیرنزولی بر اساس مقادیر $p[i]/w[i]$ مرتب‌شده‌اند.

خروجی: مقدار صحیح maxprofit که ماکزیمم ارزش است، آرایه bestset که از ۱ تا n شاخص‌دهی شده و در آن مقادیر bestset، "yes" است اگر کالای iام در مجموعه بهینه قرار داشته باشد؛ در غیراینصورت مقدار آن "no" می‌باشد.

```
void knapsack (index i,
               int profit,int weight)
{
  if (weight <= W && profit > maxprofit) {      // This set is best so far.
    maxprofit = profit;
    numbest = i;                                // Set numbest to number
    bestset = include;                          // of items considered. Set
  }                                             // bestset to this solution.
  if (promising(i)) {
```

```
      include[i + 1] = "yes";                          // Include w[i + 1].
      knapsack(i + 1, profit + p[i + 1], weight + w[i + 1]);
      include[i + 1] = "no";                           // Do not include w[i + 1].
      knapsack(i + 1, profit, weight);
   }
}

bool promising (index i)
{
   index j, k;
   int totweight;
   float bound;

   if (weight >= W)                                    // Node is promising only
      return false;                                    // if we should expand to
   else {                                              // its children. There must
      j = i + 1;                                       // be some capacity left for
      bound = profit;                                  // the children.
      totweight = weight;
      while (j <= n && totweight + w[j] <= W) {        // Grab as many items as
         totweight = totweight + w[j];                 // possible.
         bound = bound + p[j];
         j++;
      }
      k = j;                                           // Use k for consistency
      if (k <= n)                                      // with formula in text.
         bound = bound + (W - totweight) * p[k]/w[k];  // Grab fraction of kth
      return bound > maxprofit;                        // item.
   }
}
```

طبق قرارداد n، w، p، W، maxprofit، include، bestset و numbest ورودیهای روتین نیستند. اگر این متغیرها را به صورت سراسری تعریف می‌کردیم، شبه‌کد زیر ماکزیمم ارزش و مجموعهٔ دارای این ارزش را تولید می‌کرد:

```
numbest = 0;
maxprofit = 0;
Knapsack(0, 0, 0);
cout << maxprofit;                                    // نوشتن ماکزیمم ارزش
for (j = i; j <= numbest; j++)                        // نمایش مجموعه بهینه کالاها
  cout << bestset[i];
```

به خاطر دارید که برگهای درخت فضای حالات به‌خودی‌خود غیروعده‌گاه هستند زیرا حدود آنهانمی‌تواند بیش از مقدار maxprofit باشد. بنابراین، نباید شرط نهایی i = n را در تابع promising بررسی کرد. می‌خواهیم مطمئن شویم که الگوریتم ما نیازی به این بررسی ندارد. اگر i = n باشد، آنگاه bound همان مقدار اولیه‌اش (profit) را دارد و تغییری نمی‌کند. از آنجائیکه profit، کوچکتر یا مساوی maxprofit است، لذا عبارت bound > maxprofit نادرست است و این بدان معنی است که تابع Promising مقدار false را برمی‌گرداند.

حدّ بالای ما، هنگامی که مکرراً به سمت چپ درخت فضای حالات می‌رویم تغییری نمی‌کند تا اینکه به گره‌ای در سطح kام برسیم. (این موضوع با نگاه مجدد به مراحل اولیه مثال ۶-۵ قابل رؤیت است.) بنابراین، هر زمانیکه یک مقدار k تولید می‌شود، می‌توانیم مقدار آن را ذخیره کنیم و بدون فراخوانی تابع promising به جلو برویم تا اینکه به گره‌ای در سطح k-۱ام برسیم. می‌دانیم که فرزند چپ این گره غیروعده‌گاه است زیرا اگر کالای kام را به حساب آوریم، مقدار weight بیش از W می‌شود. بنابراین، فقط به سمت راست گره پیش می‌رویم. تنها پس از یک حرکت به راست است که به فراخوانی تابع promising و تعیین مقدار جدیدی از k نیازمندیم. در تمرینات از شما می‌خواهیم که این بهینه سازی و اصلاح را انجام دهید.

درخت فضای حالات در الگوریتم کوله‌پشتی ۱-۰ همانند مسئله مجموع زیرمجموعه‌ها است. همانطوریکه در بخش ۴-۵ نشان داده شده، تعداد گره‌های آن درخت برابر است با $2^{n+1}-1$. الگوریتم ۷-۵، تمام گره‌های درخت فضای حالات برای نمونه‌های زیر را بررسی می‌کند. برای مقدار مفروض n، فرض کنید W = n و

$$\text{برای } 1 \le i \le n-1 \qquad p_i = 1 \qquad w_i = 1$$

$$p_n = n \qquad w_n = n$$

جواب بهینه این است که تنها کالای nام برداشته شود و این جواب تا زمانیکه تمامی راههای سمت راست گره تا عمق n - ۱ و سپس سمت چپ گره را طی نکرده باشیم، بدست نمی‌آید. قبل از آنکه جواب بهینه پیدا شود، هر گرهٔ غیربرگ به عنوان وعده‌گاه مشخص می‌شود که بدین معنی است که تمامی گره‌های درخت فضای حالات بررسی می‌شوند. می‌توان از روش مونت کارلو برای تخمین میزان کارایی الگوریتم برای نمونه‌ای خاص استفاده کرد.

۵-۷-۲ مقایسه الگوریتم برنامه نویسی پویا و بک تراکینگ برای مسئله کوله‌پشتی ۱-۰

از بخش ۴-۴، به خاطر دارید که بدترین حالت تعداد ورودیهایی که توسط الگوریتم برنامه‌نویسی پویا برای مسئله کوله‌پشتی ۱-۰ محاسبه می‌شود در $(minimum(2^n, nw))$ است. الگوریتم بک‌تراکینگ در بدترین حالت، $\theta(2^n)$ گره را بررسی می‌کند. با داشتن حد اضای nW، ممکن است بنظر برسد که الگوریتم

برنامه‌نویسی پویا بهتر عمل می‌کند. با وجود این، الگوریتمهای بک‌تراکینگ ، آگاهی کمی راجع به تعداد بررسی‌های انجام شده در بدترین حالت توسط بک‌تراکینگ ارائه می‌دهند. تحلیل کارایی‌های نسبی دو الگوریتم فوق از لحاظ تئوری مشکل است. به هر حال می‌توان الگوریتمها را با بکارگیری نمونه‌های زیاد و توجه به عملکرد آنها، با هم مقایسه کرد. Horowitz و Sahni (۱۹۷۸)، با انجام این کار مشاهده کردند که الگوریتم بک‌تراکینگ معمولاً کاراتر از الگوریتم برنامه‌نویسی پویا است.

Horowitz و Sahni (۱۹۷۴)، دو روش تقسیم‌وغلبه و برنامه‌نویسی پویا را با هم ادغام کردند تا الگوریتمی برای مسئله کوله‌پشتی ۱-۰ ارائه نمایند که در بدترین حالت، $O(3^{n/2})$ می‌باشد. آنها نشان دادند که الگوریتمشان معمولاً کاراتر از الگوریتم بک‌تراکینگ است.

تمرینات

بخش ۱-۵ و ۲-۵

۱- الگوریتم بک‌تراکینگ را برای نمونه‌ای از مسئله n-وزیر (الگوریتم ۱-۵) با $n = ۸$ بکار بگیرید و عملیات را مرحله به مرحله نشان دهید. یک درخت فضای حالات هرس شده برای این الگوریتم را تا رسیدن به اولین جواب ترسیم کنید.

۲- یک الگوریتم بک‌تراکینگ برای مسئله n-وزیر بنویسید که به جای نسخه‌ای از روال checknode ، از نسخه‌ای از روال expand استفاده کند.

۳- نشان دهید که بدون استفاده از بک‌تراکینگ بایستی ۱۵۵ گره قبل از رسیدن به اولین جواب در نمونه‌ای از مسئله ۲-وزیر با $n = ۴$ بررسی شوند.

۴- الگوریتم بک‌تراکینگ برای مسئله n-وزیر (الگوریتم ۱-۵) را بر روی کامپیوتر خود پیاده‌سازی کنید و آن را برای نمونه‌هایی با $n = ۴$، $n = ۸$، $n = ۱۰$ و $n = ۱۲$ بکار بگیرید.

۵- با نگهداری ردّی از مجموعه ستونها، قطرهای چپ و قطرهای راست که توسط وزیرهای جایگذاری شده کنترل می‌شوند، الگوریتم بک‌تراکینگ را برای مسئله n-وزیر اصلاح کنید.

۶- الگوریتم بک‌تراکینگ برای مسئله n-وزیر (الگوریتم ۱-۵) را طوری اصلاح کنید که به جای آنکه تمامی جوابهای ممکن را تولید کند، فقط یک جواب را پیدا نماید.

۷- فرض کنید که جوابی برای نمونه مسئله n-وزیر با $n = ۴$ داریم. آیا می‌توانیم با تعمیم این جواب به جوابی برای نمونه مسئله‌ای با $n = ۵$ دست یابیم و سپس با استفاده از جوابهای $n = ۴$ و $n = ۵$، یک جواب برای نمونه مسئله‌ای با $n = ۶$ پیدا کنیم و با ادامه این روش برنامه‌نویسی پویا به جوابی برای هر نمونهٔ $n>۴$ برسیم؟ توضیح دهید.

۸- حداقل دو نمونه از مسئله n-وزیر را پیدا کنید که هیچ جوابی نداشته باشند.

بخش ۳-۵

۹- الگوریتم ۳-۵ (تخمین مونت کارلو برای الگوریتم بک‌تراکینگ مسئله n-وزیر) را بر روی کامپیوتر خود پیاده‌سازی کنید، آن را ۲۰ مرتبه بر روی نمونه‌ای با $n = ۸$ اجرا نمائید و میانگین این ۲۰ تخمین را مشخص نمائید.

۱۰- الگوریتم بک‌تراکینگ برای مسئله n-وزیر (الگوریتم ۱-۵) را طوری تغییر کنید که تعداد گره‌های بررسی شده برای هر نمونه مسئله را پیدا کند، آن را بر روی نمونه‌ای با $n = ۸$ اجرا نموده و نتیجه را با میانگین تمرین ۹ مقایسه کنید.

بخش ۴-۵

۱۱- از الگوریتم مسئله مجموع زیرمجموعه‌ها (الگوریتم ۴-۵) برای یافتن تمامی ترکیبات اعداد زیر، استفاده کنید بطوریکه مجموعشان $W = ۵۲$ شود.

$w_۱=۲$ $w_۲=۱۰$ $w_۳=۱۳$ $w_۴=۱۷$ $w_۵=۲۲$ $w_۶=۴۲$

عملیات را مرحله به مرحله نشان دهید.

۱۲- الگوریتم بک‌تراکینگ برای مسئله مجموع زیرمجموعه‌ها را بر روی کامپیوتر خود پیاده‌سازی کرده و آن را برای نمونه مسئله تمرین ۱۱ اجرا کنید.

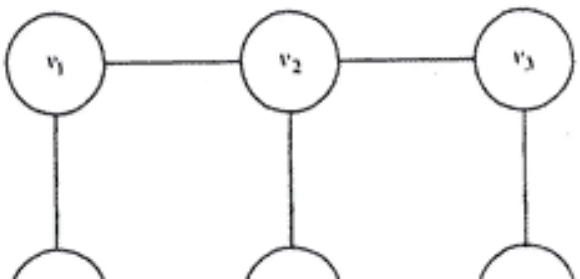
۱۳- یک الگوریتم بک‌تراکینگ برای مسئله مجموع زیر مجموعه‌ها، که در آن وزنها از قبل مرتب شده نیستند، بنویسید. کارایی این الگوریتم را با الگوریتم ۴-۵ مقایسه نمائید.

۱۴- الگوریتم بک‌تراکینگ برای مسئله مجموع زیرمجموعه‌ها (الگوریتم ۴-۵) را طوری تغییر دهید که به جای تولید تمام جوابهای ممکن، تنها یک جواب را مشخص کند.

۱۵- از روش مونت کارلو برای تخمین میزان کارایی الگوریتم بک‌تراکینگ برای مسئله مجموع زیرمجموعه‌ها (الگوریتم ۴-۵) استفاده کنید.

بخش ۵-۵

۱۶- از الگوریتم بک‌تراکینگ برای مسئله m-رنگ جهت یافتن تمام رنگ‌آمیزیهای ممکن گراف زیر، با استفاده از سه رنگ قرمز، سبز و سفید استفاده کنید. عملیات را مرحله به مرحله نشان دهید.



۱۷- برای رنگ‌آمیزی یک گراف با انتخاب یک گرهٔ شروع و یک رنگ، شروع به رنگ‌آمیزی می‌کنیم. سپس یک رنگ جدید و یک گرهٔ رنگ نشده را انتخاب می‌کنیم که تا حد امکان گره‌های بیشتری را رنگ کنیم. این روند را تا آنجا ادامه می‌دهیم که تمامی گره‌های گراف رنگ شوند و تمام رنگها مصرف شوند. الگوریتمی برای این روش حریص، جهت رنگ‌آمیزی گرافی متشکل از n گره بنویسید. الگوریتم خود را تحلیل نموده و نتایج را با استفاده از نمادهای ترتیب نشان دهید.

۱۸- از الگوریتم تمرین ۱۷ برای رنگ‌آمیزی گراف تمرین ۱۶ استفاده کنید.

۱۹- می‌خواهیم تعداد رنگهای مصرفی در رنگ‌آمیزی یک گراف را به حداقل برسانیم. آیا روش حریص تمرین ۱۷، یک جواب بهینه را تضمین می‌کند یا خیر؟ توضیح دهید.

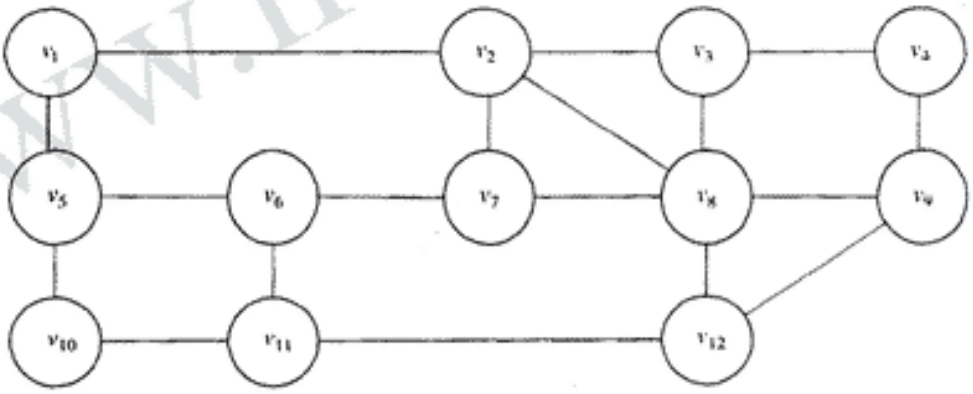
۲۰- کارایی دو الگوریتم بک‌تراکینگ و حریص برای مسئله m-رنگ در تمرین ۱۷ را با هم مقایسه کنید. با توجه به نتایج مقایسه و جواب شما به تمرین ۱۹، چطور ممکن است که شخصی بخواهد از روش حریص استفاده کند؟

۲۱- الگوریتمی برای مسئله ۲-رنگ بنویسید که پیچیدگی زمانی آن در بدترین حالت، نمایی از n نباشد.

۲۲- چند نمونه از کاربردهای علمی مسئله m-رنگ را نام ببرید.

بخش ۶-۵

۲۳- با استفاده از الگوریتم بک‌تراکینگ برای مسئله چرخه‌های هامیلتونی (الگوریتم ۶-۵)، تمام چرخه‌های هامیلتونی ممکن گراف زیر را پیدا کنید.



۲۴- الگوریتم بک‌تراکینگ برای مسئله چرخه‌های هامیلتونی (الگوریتم ۶-۵) را بر روی کامپیوتر خود پیاده سازی نموده و آن را بر روی نمونه تمرین ۲۳ اجرا نمائید.

۲۵- گرهٔ آغازین الگوریتم بک‌تراکینگ برای مسئله چرخه‌های هامیلتونی (الگوریتم ۶-۵) در تمرین ۲۴ را تغییر دهید و کارایی آن را با الگوریتم ۶-۵ مقایسه نمائید.

۲۶- الگوریتم بک‌تراکینگ برای مسئله چرخه‌های هامیلتونی را طوری تغییر دهید که به جای تولید تمام جوابهای ممکن، تنها یک جواب را پیدا کند.

۲۷- الگوریتم بک‌تراکینگ برای مسئله چرخه‌های هامیلتونی (الگوریتم ۶-۵) را تحلیل نموده و پیچیدگی زمانی بدترین حالت آن را با نماد ترتیب نشان دهید.

۲۸- از روش مونت کارلو برای تخمین میزان کارایی الگوریتم بک‌تراکینگ برای مسئله چرخه‌های هامیلتونی (الگوریتم ۶-۵) استفاده کنید.

بخش ۷-۵

۲۹- مقادیر باقیمانده و حدود را، پس از ملاقات گرهٔ (۴،۱) در مثال ۶-۵ (بخش ۱-۷-۵)، محاسبه نمائید.

۳۰- با استفاده از الگوریتم بک‌تراکینگ برای مسئله کوله‌پشتی ۱-۰ (الگوریتم ۷-۵)، ارزش نمونه مسئله زیر را بیشینه کنید. عملیات را مرحله به مرحله نشان دهید.

	p_i/w_i	w_i	p_i	i
	$۱۰	۲	$۲۰	۱
	$۶	۵	$۳۰	۲
W = ۱۹	$۵	۷	$۳۵	۳
	$۴	۳	$۱۲	۴
	$۳	۱	$۲	۵

۳۱- الگوریتم بک‌تراکینگ برای مسئله کوله‌پشتی ۱-۰ (الگوریتم ۷-۵) را بر روی کامپیوتر خود پیاده‌سازی نموده و آن را برای نمونه تمرین ۳۰ اجرا نمائید.

۳۲- الگوریتم برنامه‌نویسی پویا برای مسئله کوله‌پشتی ۱-۰ (در بخش ۳-۴-۴) را پیاده‌سازی نموده و کارایی آن را با الگوریتم بک‌تراکینگ برای مسئله کوله‌پشتی ۱-۰ (الگوریتم ۷-۵) مقایسه نمائید. (از نمونه مسئله‌های بزرگ استفاده کنید.)

۳۳- الگوریتم بک‌تراکینگ برای مسئله کوله‌پشتی ۱-۰ (الگوریتم ۷-۵) را با فراخوانی تابع وعده‌گاه، پس از تنها یک حرکت به راست، اصلاح کنید. سپس آن را طوری تغییر دهید که یک جواب در لیستی با طول متغیر تولید کند.

۳۴- با استفاده از روش مونت کارلو، میزان کارایی الگوریتم بک‌تراکینگ برای مسئله کوله‌پشتی ۱-۰ را تخمین بزنید.

تمرینات اضافی

۳۵- سه کاربرد دیگر روش بک‌تراکینگ را نام ببرید.

۳۶- الگوریتم بک‌تراکینگ برای مسئله n-وزیر را طوری تغییر دهید که تنها جوابهایی که از لحاظ تقارن و یا چرخش با هم یکسان هستند را تولید کند.

۳۷- الگوریتم بک‌تراکینگ برای مسئله چرخه‌های هامیلتونی (الگوریتم ۶-۵) را طوری تغییر دهید که یک چرخهٔ هامیلتونی با حداقل هزینه برای یک گراف وزن‌دار را مشخص نماید.

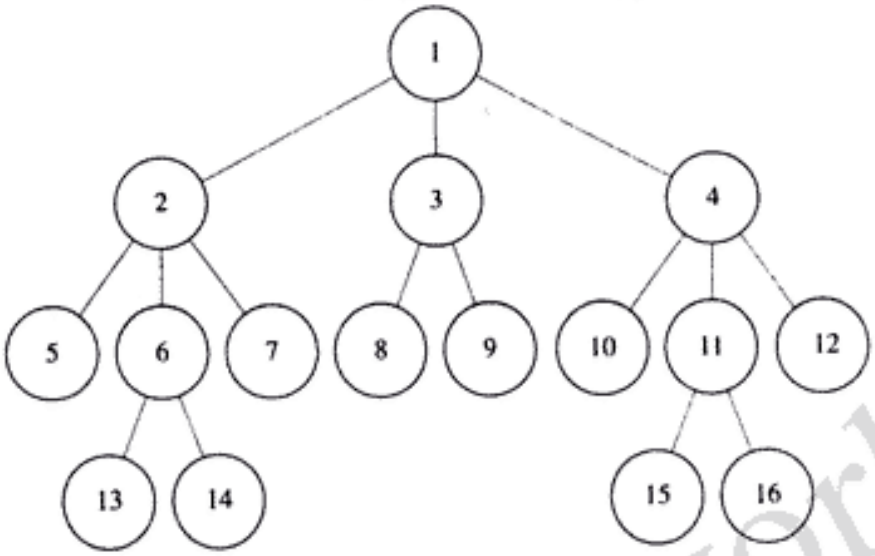
فصل ۶

شاخه و حد (Branch-and-Bound)

تاکنون دو الگوریتم برای مسئله کوله‌پشتی ۱-۰ ارائه داده‌ایم: الگوریتم برنامه نویسی‌پویا در بخش ۴-۴ و الگوریتم پک‌تراکینگ در بخش ۷-۵. از آنجائیکه هر دو الگوریتم مذکور در بدترین حالت به صورت زمان-نمایی عمل می‌کنند، لذا برای حل نمونه مسئله دزد و کوله‌پشتی، به چندین سال وقت نیاز خواهیم داشت. در این فصل روش دیگری را به دزد مسئله‌مان پیشنهاد می‌کنیم. همانطوریکه خواهید دید، این روش که به الگوریتم شاخه و حد موسوم است، در واقع یک اصلاح و بهینه‌سازی روی الگوریتم پک‌تراکینگ می‌باشد.

استراتژی طراحی شاخه و حد بسیار شبیه به تکنیک پک‌تراکینگ در حل یک مسئله با استفاده از درخت فضای حالات است؛ با این تفاوت که روش شاخه و حد، اولاً ما را به روش خاصی از پیمایش درخت محدود نمی‌کند و ثانیاً، تنها برای مسائل بهینه سازی بکار می‌رود. الگوریتم شاخه و حد، حدی را در یک گره محاسبه می‌کند تا مشخص نماید که آیا گرهٔ مورد نظر، یک وعده‌گاه است یا خیر؟ اگر این حد بهتر از مقدار بهترین جوابی که تاکنون پیدا شده است نباشد، یعنی اینکه آن گره یک وعده‌گاه نیست؛ در غیر اینصورت، گره می‌تواند یک وعده‌گاه باشد. از آنجائیکه مقدار بهینه در برخی مسائل، می‌نیمم و

شکل ۱-۶ جستجوی سطحی یک درخت. گره‌ها به ترتیبی که ملاقات می‌شوند، شماره‌گذاری می‌گردند. فرزندان یک گره از چپ به راست ملاقات می‌شوند.



در برخی مسائل ماکزیمم است، لذا ما از لفظ "بهتر" استفاده کردیم تا بتوانیم با توجه به مسئله، آن را تعریف نمائیم. همانند الگوریتم‌های بک‌تراکینگ، الگوریتم‌های شاخه و حد نیز در بدترین حالت به صورت زمان-نمایی (یا بدتر از آن) عمل می‌کنند. به هر حال، آنها می‌توانند برای نمونه‌های خیلی بزرگ، بسیار کارا باشند.

الگوریتم بک‌تراکینگ برای مسئله کوله‌پشتی ۱-۰ در بخش ۷-۵، در واقع یک الگوریتم شاخه و حد است. در این الگوریتم، در صورتی که مقدار bound (حد) بزرگتر از مقدار جاری maxprofit نباشد، تابع وعده‌گاه، یک مقدار false را برمی‌گرداند. با این حال، یک الگوریتم بک‌تراکینگ از مزایای واقعی شاخه و حد استفاده نمی‌کند. علاوه بر بکارگیری حد جهت تعیین وعده‌گاه بودن یک گره، می‌توانیم حدود گره‌های وعده‌گاه را با هم مقایسه نموده و تنها فرزندان گرهٔ با بهترین حد را ملاقات کنیم. در این روش، ما بسیار سریعتر از حالتی که گره‌ها با یک ترتیب از پیش تعیین شده (نظیر جستجوی عمقی) ملاقات می‌شوند، به یک جواب بهینه می‌رسیم. این روش را **جستجوی اول-بهترین با هرس شاخه و حد** می‌نامیم. پیاده سازی این روش با یک تغییر ساده در روشی دیگر، موسوم به **جستجوی سطحی با هرس شاخه و حد** بدست می‌آید. بنابراین، با وجود اینکه روش اخیر مزایا و کارایی جستجوی عمقی را ندارد، ولی در بخش ۱-۶، ابتدا مسئله کوله‌پشتی ۱-۰ را با استفاده از جستجوی اول-بهترین تشریح نموده و آن را برای حل مسئله کوله‌پشتی ۱-۰ بکار می‌گیریم. سپس در بخشهای ۲-۶ و ۳-۶، از روش جستجوی اول-بهترین برای حل دو مسئله دیگر استفاده خواهیم نمود.

قبل از ادامه بحث، مروری بر جستجوی سطحی خواهیم داشت. **جستجوی سطحی** یک درخت، شامل ملاقات ریشه درخت در ابتدا، سپس تمامی گره‌های سطح ۱ و بعد تمامی گره‌های سطح ۲ و الی آخر می‌باشد. شکل ۱-۶، جستجوی سطحی یک درخت را نشان می‌دهد. گره‌ها به ترتیبی که ملاقات می‌شوند، شماره‌گذاری شده‌اند.

برخلاف جستجوی عمقی، هیچ الگوریتم بازگشتی ساده‌ای برای جستجوی سطحی وجود ندارد. برای پیاده‌سازی آن می‌توانیم از یک صف استفاده کنیم که الگوریتم آن در زیر آمده است. روال enqueue، یک عنصر را در انتهای صف درج کرده و روال dequeue عنصری را از جلوی صف حذف می‌کند.

```
void breath_first_tree_search(tree T)
{
   queue_of_node Q;
   node u,v;
   initialize(Q);
   v = root of T;
   visit v;
   enqueue(Q, v);
   while (!empty(Q)){
      denqueue(Q, v);
      for (each child u of v){
        visit u;
        enqueue(Q, u);
      }
   }
}
```

اگر متقاعد نشده‌اید که این روال، یک جستجوی سطحی انجام می‌دهد، کافی است آن را روی درخت شکل ۱-۶ پیاده‌سازی نمائید. در این درخت، همانطوری که قبلاً نیز اشاره شد، گره‌ها از چپ به راست ملاقات می‌شوند.

۱-۶ حل مسئله کوله‌پشتی ۱-۰ با استفاده از شاخه و حد

با بکارگیری استراتژی شاخه و حد بر روی مسئله کوله‌پشتی ۱-۰، چگونگی استفاده از آنرا نشان می‌دهیم. ابتدا روی یک نسخهٔ ساده موسوم به جستجوی سطحی با هرس شاخه و حد بحث نموده، سپس یک اصلاح ساده موسوم به جستجوی اول-بهترین را با هرس شاخه و حد انجام می‌دهیم.

۱-۱-۶ جستجوی سطحی با هرس شاخه و حد

این روش را با یک مثال بیان می‌کنیم.

مثال ۱-۶ نمونه‌ای از مسئله کوله‌پشتی ۱-۰ که در مثال ۶-۵ آمده، مفروض است. یعنی n = ۱۴ و w = ۱۶، و داریم:

i	p_i	w_i	p_i/w_i
۱	$۴۰	۲	$۲۰
۲	$۳۰	۵	$۶
۳	$۵۰	۱۰	$۵
۴	$۱۰	۵	$۲

همانند مثال ۶-۵، عناصر بر اساس مقادیر p_i/w_i مرتب شده‌اند. در این مثال، جستجوی سطحی با هرس شاخه و حد، دقیقاً همانند بک‌تراکینگ برای مثال ۶-۵ عمل می‌کند؛ با این تفاوت که به جای جستجوی عمقی، یک جستجوی سطحی انجام می‌دهد. در اینصورت، weight را برای مجموع وزن و profit را بعنوان مجموع ارزش کالاهایی که تا یک گره به حساب آمده‌اند، درنظر می‌گیریم. برای تعیین اینکه آیا یک گرهٔ وعده‌گاه است یا خیر، در ابتدا به totwight مقدار weight و به bound مقدار profit را تخصیص می‌دهیم، سپس به روش حریص، کالاها را برداشته و اوزان و ارزشهای آنها را بترتیب به totwight و bound اضافه می‌کنیم. این عمل را تا زمانی ادامه می‌دهیم که به کالایی برسیم که افزودن وزن آن موجب تجاوز از حد W شود. در اینصورت بخشی از آن کالا را برمی‌داریم و وزن آن بخش را به totwight اضافه می‌کنیم. در این روش bound یک حد بالا روی مقدار ارزشی می‌شود که ما می‌توانستیم با بسط گره بدست آوریم .اگر گره‌ای در سطح i باشد و گرهٔ سطح k گره‌ای باشد که افزودن وزنش موجب تجاوز از حد W شود، در اینصورت

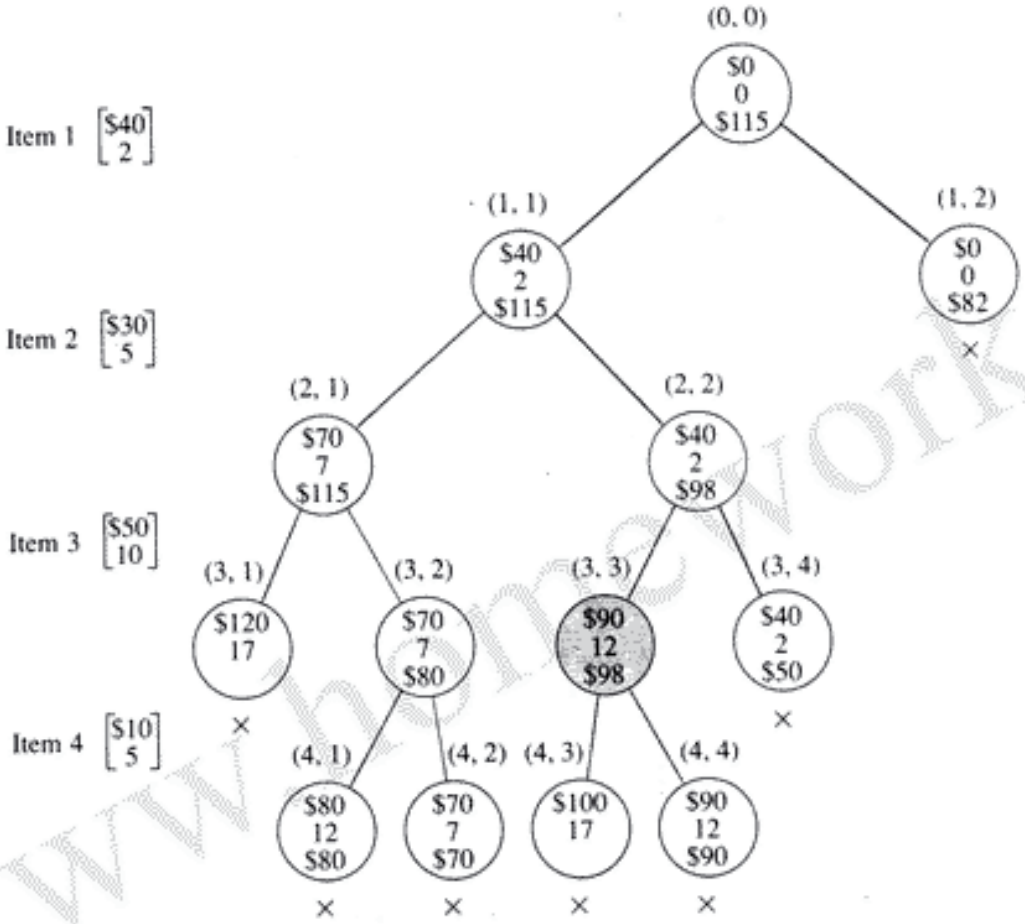
$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

و

$$bound = (profit + \sum_{j=i+1}^{k-1} p_j) + (w - totweight) \times \frac{p_k}{w_k}$$

یک گره در صورتی غیروعده‌گاه است که این حد کوچکتر یا مساوی maxprofit (مقدار بهترین جواب پیدا شده تا این نقطه) باشد. همچنین قبلاً داشتیم که یک گره، در صورتی غیروعده‌گاه است که $weight \geq w$ شود. درخت فضای حالات هرس شده که با بکارگیری جستجوی سطحی بر روی نمونهٔ این مثال تولید شده است، به همراه شاخه‌هایی که با بکارگیری حدود فوق هرس شده‌اند، در شکل ۲-۶ آمده است. مقادیر profit و weight در هر گره از بالا به پائین آمده‌اند. گرهٔ سایه‌دار، جایی است که ماکزیمم ارزش در آن پیدا شده است. گره‌ها براساس سطوح و موقعیتشان از چپ به راست شماره‌گذاری شده‌اند. از آنجائیکه این مراحل بسیار شبیه به مراحل انجام شده در مثال ۶-۵ می‌باشند، لذا از تشریح مجدد آنها خودداری کرده و تنها به چند نکته مهم اشاره می‌کنیم. رجوع ما به یک گره، توسط سطح و موقعیت آن از سمت چپ درخت انجام می‌شود. اولاً، توجه کنید که گره‌های (۱، ۳) و (۱، ۴)، حدی برابر صفر دلار دارند. یک الگوریتم شاخه و حد تعیین می‌کند که آیا بایستی یک گره بسط پیدا کند یا خیر، که این کار با بررسی اینکه «آیا حد آن گره از مقدار بهترین جواب پیدا شده (تاکنون) بهتر است یا خیر؟».

شکل ۲-۶ درخت فضای حالات هرس شده که با بکارگیری جستجوی سطحی با هرس شاخه و حد در مثال ۱-۶ تولید شده است. مقادیر موجود در هر گره از بالا به پائین عبارتند از: ارزش کل کالاهای مسروقه تا آن گره، وزن کل آنها و حدی برای مجموع ارزش که می‌توانست با بسط گره بدست آورد. گرهٔ سایه‌دار، گره‌ای است که جواب بهینه در آن پیدا شده است.



صورت می‌گیرد. بنابراین، هنگامی که یک گرهٔ غیروعده‌گاه است، چون وزن آن کمتر از W نیست، لذا ما حد آن را با $۰ مقداردهی می‌کنیم. در این روش، ما مطمئنیم که حد آن نمی‌تواند بهتر از مقدار بهترین جوابی که تاکنون بدست آمده است، باشد. ثانیاً، به خاطر آورید وقتی که بک‌تراکینگ (جستجوی عمقی) بر روی این نمونه بکار گرفته شد، گرهٔ (۲، ۱) به عنوان گرهٔ غیروعده‌گاه مشخص شد و به همین دلیل، دیگر آن گره را بسط ندادیم. اما در مورد جستجوی سطحی، این گره سومین گره‌ای است که ملاقات می‌شود. در زمانی که این ملاقات صورت می‌گیرد، مقدار maxprofit تنها برابر $۴۰ است. از آنجائیکه حد $۸۲ بیشتر از مقدار maxprofit در این نقطه است، لذا گره را بسط می‌دهیم. بالاخره در یک جستجوی سادهٔ سطحی با هرس شاخه و حد، تعیین اینکه «آیا فرزندان یک گره بایستی ملاقات شوند یا خیر؟»، در زمان ملاقات گره انجام می‌شود. بنابراین، هنگامی که گرهٔ (۲، ۳) را بررسی می‌کنیم، در مورد ملاقات فرزندان آن تصمیم می‌گیریم زیرا مقدار maxprofit در اینجا برخلاف حد گره که $۸۲ بود، برابر $۷۰ است.

در جستجوی سطحی، برخلاف جستجوی عمقی، مقدار maxprofit در زمانی که فرزندان گرهٔ (۲، ۳) را ملاقات می‌کنیم، مقداری برابر ۹۰$ دارد. در اینصورت، با بررسی این فرزندان، زمان را بیهوده تلف می‌کنیم. ما در جستجوی اول-بهترین، از این عمل جلوگیری می‌کنیم.

اکنون که به تشریح تکنیک سطحی می‌پردازیم، می‌خواهیم یک الگوریتم عمومی برای جستجوی سطحی با هرس شاخه و حد ارائه دهیم. اگرچه درخت فضای حالات T را به عنوان ورودی این الگوریتم همه-منظوره معرفی می‌کنیم، اما در عمل، درخت فضای حالت تنها به صورت تلویحی وجود دارد. پارامترهای مسئله، ورودیهای حقیقی الگوریتم هستند که تعیین کنندهٔ درخت فضای حالات T هستند.

```
void breath_first_branch_and_bound (state_space_tree T,
                                    number& best)
{
   queue_of_node Q;
   node u,v;
   initialize(Q);
   v = root of T;
   visit v;
   enqueue(Q, v);
   while (! empty(Q)){
      denqueue(Q, v);
      for (each child u of v){
        if (value(u) is better than best)
          best = value(u);
        if (bound(u) is better than best)
          enqueue(Q, u);
      }
   }
}
```

این الگوریتم از تغییر الگوریتم جستجوی سطحی، که در آغاز این فصل ارائه شد، بدست آمده است. در این الگوریتم، تنها گره‌ای بسط می‌یابد (فرزندانش ملاقات می‌شوند) که حد آن بهتر از مقدار بهترین جواب جاری باشد. مقدار بهترین جواب جاری (متغیر best) با مقدار جواب ریشه، مقداردهی می‌شود. در برخی از کاربردها، هیچ جوابی در ریشه وجود ندارد زیرا ما بایستی در یک برگ از درخت فضای حالات باشیم تا یک جواب داشته باشیم. توابع bound و value در هر کاربردی از breadth_first_branch_and_bound متفاوت هستند. همچنانکه خواهیم دید، هیچگاه واقعاً تابع value را نمی‌نویسیم؛ بلکه مقدار موردنظر را به طور مستقیم محاسبه می‌کنیم.

در ادامه الگوریتمی خاص برای مسئله کوله‌پشتی ۱-۰ ارائه می‌کنیم. از آنجائیکه از امکانات و مزایای بازگشت نمی‌توانیم استفاده کنیم (یعنی متغیر جدیدی نداریم که در هر فراخوانی بازگشتی ایجاد شود)، لذا بایستی تمامی اطلاعات مورد نیاز یک گره را در همان گره ذخیره کنیم. بنابراین، گره‌های الگوریتم ما به صورت زیر تعریف می‌شوند:

```
struct node
{
  int level;          // سطح گره در درخت
  int profit;
  int weight;
};
```

الگوریتم ۱-۶ الگوریتم جستجوی سطحی با هرس شاخه و حد برای مسئله کوله‌پشتی ۱-۰

مسئله: فرض کنید n کالا داریم که هر یک از آنها دارای وزن و ارزشی معین می‌باشند. وزنها و ارزشها، اعدادی صحیح و مثبت هستند. مجموعه‌ای از کالاها با ماکزیمم مجموع ارزشها را مشخص کنید بطوری که مجموع اوزان آنها بیشتر از عدد صحیح مثبت W نباشد.

ورودی: اعداد صحیح مثبت n و W، آرایه‌هایی از اعداد صحیح مثبت w و p که از ۱ تا n شاخص‌دهی شده و بترتیب غیرصعودی براساس مقادیر $p[i]/w[i]$ مرتب شده‌اند.

خروجی: یک عدد صحیح maxprofit که عبارت است از مجموع ارزشهای یک مجموعه بهینه.

```
void knapsack2 (int n,
                const int p[ ], const int w[ ],
                int W,
                int& maxprofit)
{
  queue_of_node Q;
  node u, v;
  initialize(Q);                          // Initialize Q to be empty.
  v.level = 0; v.profit = 0; v.weight = 0;  // Initialize v to be the root.
  maxprofit = 0;
  enqueue(Q, v);
  while (! empty(Q)) {
    dequeue(Q, v);
    u.level = v.level + 1;                // Set u to a child of v.
    u.weight = v.weight + w[u.level];     // Set u to the child that
    u.profit = v.profit + p[u.level];     // includes the next item.
    if (u.weight <= W && u.profit > maxprofit)
      maxprofit = u.profit;
    if (bound(u) > maxprofit)
      enqueue(Q, u);
```

```
    u.weight = v.weight;                    // Set u to the child that
    u.profit = v.profit;                    // does not include the
    if (bound(u) > maxprofit)               // next item.
      enqueue(Q, u);
  }
}

float bound (node u)
{
  index j, k;
  int totweight;
  float result;

  if (u.weight > = W)
    return 0;
  else {
    result = u.profit;
    j = u.level + 1;
    totweight = u.weight;
    while (j <= n && totweight + w[j] <= W){   // Grab as many items
      totweight = totweight + w[j];            // as possible.
      result = result + p[j];
      j++;
    }
    k = j;                                     // Use k for consistency
    if (k <= n)                                // with formula in text.
      result = result + (W - totweight) * p[k]/w[k];  // Grab fracton of kth
    return result;                             // item.
  }
}
```

هنگامی که گرهٔ جاری به حساب نمی‌آید، نیازی به بررسی اینکه «آیا u.profit از maxprofit تجاوز می‌کند یا خیر؟» نداریم. زیرا در این حالت، u.profit برابراست با ارزش متناظر پدر u و این بدین معناست که مقدار آن نمی‌تواند از maxprofit تجاوز کند. هیچ لزومی به ذخیره‌سازی حد در یک گره وجود ندارد زیرا پس از مقایسه آن با maxprofit، دیگر به آن نیازی نداریم. تابع bound، اساساً مشابه تابع promising در الگوریتم ۷-۵ عمل می‌کند؛ با این تفاوت که تابع bound براساس قوانین ایجاد الگوریتم‌های شاخه و حد نوشته شده است و در نتیجه، یک عدد صحیح را برمی‌گرداند؛ در حالیکه تابع promising، به دلیل اینکه براساس قوانین بک‌تراکینگ نوشته شده است، حتماً یک مقدار بولی را ارجاع می‌دهد. در الگوریتم شاخه و حد ما، مقایسهٔ با maxprofit در روال فراخوانی صورت می‌گیرد و هیچ لزومی به بررسی شرط i = n در تابع bound نیست زیرا در این حالت، مقدار ارجاعی توسط تابع bound، کمتر یا مساوی maxprofit است و این بدین معناست که گره در صف قرار نگرفته است.

الگوریتم ۱-۶، یک مجموعهٔ بهینه از عناصر را تولید نمی‌کند؛ بلکه فقط مجموع ارزشهای یک مجموعهٔ بهینه را تعیین می‌کند. الگوریتم می‌تواند به شکلی تغییر یابد که یک مجموعهٔ بهینه را نیز تولید نماید، بدینصورت که در هر گره، یک متغیر items را نیز ذخیره کنیم که آن شامل مجموعه عناصری است که در بالای آن قرار دارند. همچنین متغیر bestitems را به منظور مشخص نمودن بهترین مجموعهٔ جاری از عناصر، در نظر گرفته و هنگامی که maxprofit معادل u.profit می‌شود، bestitems را نیز مساوی u.items قرار می‌دهیم.

۶-۱-۲ جستجوی اول-بهترین با هرس شاخه و حد

بطور کلی، استراتژی جستجوی سطحی هیچ مزیتی نسبت به جستجوی عمقی (یک‌تراکینگ) ندارد. با وجود این، جستجویمان را با استفاده از حدّ بدست آمده بهبود می‌بخشیم تا بتواند بیش از تعیین وعده‌گاه بودن یا نبودن یک گره برای ما مفید باشد. پس از ملاقات تمامی فرزندان یک گرهٔ معین، می‌توانیم به تمامی گره‌های وعده‌گاه (گره‌های توسعه نیافته) نظری بیاندازیم و گرهٔ با بهترین حد را بسط دهیم. به خاطر دارید که یک گره در صورتی وعده‌گاه است که حد آن، بهتر از مقدار جوابی باشد که تاکنون پیدا شده است. در این روش، اغلب بسیار سریعتر از حالتی که بدون توجه و فقط براساس یک ترتیب از پیش تعیین شده حرکت می‌کردیم، به یک جواب بهینه دست می‌یابیم. مثال زیر این روش را نشان می‌دهد.

مثال ۲-۶ نمونه مسئلهٔ کوله‌پشتی ۱-۰ در مثال ۱-۶ را در نظر بگیرید. یک جستجوی اول-بهترین، منجر به تولید درخت فضای حالات هرس شدهٔ شکل ۳-۶ می‌شود. مقادیر profit، weight، و bound در هر گره از درخت، بترتیب از بالا به پائین مشخص شده‌اند. گرهٔ سایه‌دار، جایی است که در آن ارزش ماکزیمم پیدا شده است. در ادامه، مراحل تشکیل این درخت را بررسی می‌کنیم. یادآور می‌شویم که گره‌ها بر اساس سطح و موقعیتشان از چپ به راست شماره‌گذاری می‌شوند. مقادیر و حدود، مشابه روشی که در مثال‌های ۶-۵ و ۱-۶ انجام داده‌ایم، مقایسه می‌شوند. لازم به ذکر است که در هر مرحله، محاسبات را نمی‌نویسیم؛ بلکه تنها نشان می‌دهیم که کدام گره، غیروعده‌گاه و کدامیک، وعده‌گاه است. مراحل به صورت زیر می‌باشند:

۱- گرهٔ (۰، ۰) را ملاقات کنید. (ریشه)

(a) ارزش آن را برابر $۰ و وزن آن را برابر صفر قرار دهید.

(b) حد آن را محاسبه کنید تا برابر $۱۱۵ شود. (برای انجام به مثال ۶-۵ مراجعه کنید.)

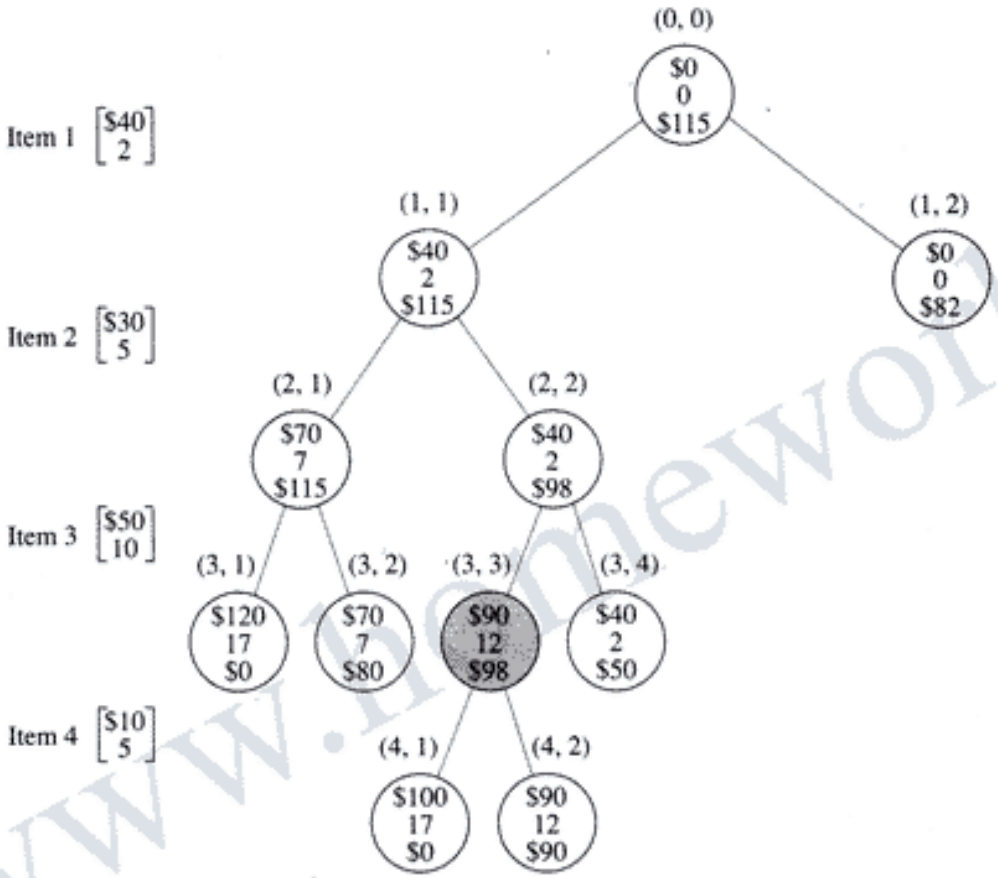
(c) مقدار maxprofit را برابر صفر قرار دهید.

۲- گرهٔ (۱، ۱) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب برابر $۴۰ و ۲ شوند.

(b) چون وزن آن (۲) کوچکتر یا مساوی ۱۶ (مقدار W) است و ارزش آن ($۴۰) بزرگتر از $۰

شکل ۳-۶ درخت فضای حالات هرس شده که با بکارگیری جستجوی اول-بهترین با هرس شاخه و حد در مثال ۲-۶ تولید شده است. مقادیر موجود در هر گره از بالا به پائین عبارتند از: ارزش کل کالاهای مسروقه تا آن گره، وزن کل آنها، و حدی برای مجموع ارزشی که می‌توانست با بسط گره بدست آورد. گرهٔ سایه‌دار، گره‌ای است که جواب بهینه در آن پیدا شده است.



(مقدار maxprofit) است، maxprofit را برابر ۴۰$ قرار دهید.

(c) حد آن را محاسبه نمائید تا برابر ۱۱۵$ شود.

۳- گرهٔ (۲، ۱) را ملاقات کنید.

(a) ارزش و وزن آنرا محاسبه کنید تا برابر ۵۰$ و ۵ شوند.

(b) حد آن را محاسبه نمائید تا برابر ۸۲$ شود.

۴- گرهٔ وعده‌گاه توسعه نیافته با بزرگترین حد را تعیین کنید.

(a) از آنجائیکه گرهٔ (۱، ۱) حدی برابر ۱۱۵$ و گرهٔ (۲، ۱) حدی برابر ۸۲$ دارد، لذا گرهٔ (۱، ۱)، گرهٔ وعده‌گاه توسعه نیافته با بزرگترین حد است. فرزندان این گره را بعداً ملاقات خواهیم کرد.

۵- گرهٔ (۲، ۱) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب $۷۰ و ۷ شوند.

(b) چون وزن آن (۷) کوچکتر یا مساوی ۱۶ (مقدار W) است و ارزش آن ($۷۰) بزرگتر از $۴۰ (مقدار maxprofit) است، maxprofit را برابر $۷۰ قرار دهید.

(c) حد آن را محاسبه نمائید تا برابر $۱۱۵ شود.

۶- گرهٔ (۲، ۲) را ملاقات کنید.

(a) ارزش و وزن آنرا محاسبه کنید تابترتیب برابر $۴۰ و ۲ شوند.

(b) حد آن را محاسبه نمائید تا برابر $۹۸ شود.

۷- گرهٔ وعده‌گاه توسعه‌نیافته با بزرگترین حد را تعیین کنید.

(a) آن گره، گرهٔ (۱، ۲) است. فرزندان آن را بعداً ملاقات خواهیم کرد.

۸- گرهٔ (۱، ۳) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب برابر $۱۲۰ و ۱۷ شوند.

(b) آن را به عنوان یک گرهٔ غیروعده‌گاه تعیین کنید زیرا وزن آن (۱۷) بیش از ۱۶ (مقدار W) است. با مقدار دهی bound با $۰، آنرا بعنوان غیروعده‌گاه مشخص می‌کنیم.

۹- گرهٔ (۲، ۳) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب برابر $۷۰ و ۷ شود.

(b) حد آن را محاسبه نمائید تا برابر $۸۰ شود.

۱۰- گرهٔ وعده‌گاه توسعه‌نیافته با بزرگترین حد را تعیین کنید.

(a) آن گره، گرهٔ (۲، ۲) است فرزندان آن را بعداً ملاقات خواهیم کرد.

(b) چون وزن آن (۱۲) کوچکتر از ۱۶ (مقدار W) است و ارزش آن ($۹۰) بیش از $۷۰ (مقدار maxprofit) است، مقدار $۹۰ را به maxprofit تخصیص دهید.

(c) در این نقطه، گره‌های (۱، ۲) و (۲، ۳) غیروعده‌گاه می‌شوند زیرا حد آنها (بترتیب $۸۲ و $۸۰) کوچکتر یا مساوی $۹۰ (مقدار جدید maxprofit) است.

(d) حد آن را محاسبه نمائید تا برابر $۹۸ شود.

۱۲- گرهٔ (۴، ۳) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب برابر $۴۰ و ۲ شوند

(b) حد آن را محاسبه نمائید تا برابر $۵۰ شود.

(c) آن را به عنوان یک گرهٔ غیروعده‌گاه مشخص کنید زیرا حد آن ($۵۰) کوچکتر یا مساوی $۹۰ (مقدار maxprofit) است.

۱۳- گرهٔ وعده‌گاه توسعه نیافته با بزرگترین حد را تعیین کنید.

(a) تنها گرهٔ وعده‌گاه توسعه نیافته، گرهٔ (۳، ۳) است. فرزندان آنرا بعداً ملاقات خواهیم کرد.

۱۴- گرهٔ (۱، ۴) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب برابر $۱۰۰ و ۱۷ شود.

(b) آن را به عنوان یک گرهٔ غیروعده‌گاه تعیین کنید زیرا وزن آن (۱۷) بزرگتر یا مساوی ۱۶ (مقدار W) است. حد آنرا برابر $۰ قرار می‌دهیم.

۱۵- گرهٔ (۲، ۴) را ملاقات کنید.

(a) ارزش و وزن آن را محاسبه کنید تا بترتیب برابر $۹۰ و ۱۲ شود.

(b) حد آن را محاسبه نمائید تا برابر $۹۰ شود.

(c) آن را به عنوان یک گرهٔ غیروعده‌گاه مشخص کنید زیرا حد آن ($۹۰) کوچکتر یامساوی $۹۰ (مقدار maxprofit) است. برگهای درخت فضای حالت به خودی خود غیروعده‌گاه هستند زیرا حدود آنها نمی‌توانند از مقدار maxfrofit تجاوز کنند.

از آنجائیکه هیچ گرهٔ وعده‌گاه توسعه نیافته‌ای وجود ندارد، لذا عملیات خاتمه می‌یابد.

با استفاده از جستجوی اول-بهترین، تنها ۱۱ گره را مورد بررسی قرار داده‌ایم که ۶ گره کمتر از جستجوی سطحی (شکل ۲-۶) و ۲ گره کمتر از جستجوی عمقی (شکل ۱۴-۵) است. البته اجتناب از بررسی ۲ گره، چندان مؤثر نیست ولی در یک درخت فضای حالات بزرگ، کاهش بررسی‌ها بسیار با ارزش است. در اینجا به این نکته اشاره می‌کنیم که هیچ تضمینی وجود ندارد که آنچه که به عنوان بهترین گره بنظر می‌رسد، واقعاً ما را به یک جواب بهینه هدایت کند. در مثال ۲-۶، بنظر می‌رسد که گرهٔ (۱، ۲) بهتر از گرهٔ (۲، ۲) باشد اما گرهٔ (۲، ۲) ما را به یک جواب بهینه هدایت می‌کند. در کل، جستجوی اول-بهترین هنوز هم می‌تواند منجر به تولید بخش اعظم و یا تمام درخت فضای حالت برای برخی نمونه‌ها شود.

پیاده‌سازی جستجوی اول-بهترین، شامل یک تغییر ساده بر روی جستجوی سطحی است. به جای استفاده از یک صف، از یک صف اولویت استفاده می‌کنیم. در یک **صف اولویت**، عنصر با بالاترین اولویت حذف می‌شود. در جستجوی اول-بهترین، عنصر با اولویت بالاتر، همان گره‌ای است که دارای بهترین حد است. یک صف اولویت می‌تواند به عنوان یک لیست پیوندی پیاده‌سازی شود؛ اما اغلب با کارایی بالاتری در heap مورد استفاده قرار می‌گیرد. (برای آشنایی با heap به بخش ۶-۷ مراجعه کنید.) یک الگوریتم کلی برای روش جستجوی اول-بهترین را در ادامه می‌آوریم. همچون گذشته، درخت T تنها به صورت تلویحی وجود دارد. در این الگوریتم، insert(PQ, v) روالی است که V را به صف اولویت PQ اضافه می‌کند و remore(PQ, v)، روالی است که گرهٔ با بهترین حد را از صف اولویت حذف و مقدار آن را مساوی v قرار می‌دهد.

```
void best_first_branch_and_bound(state_space_tree T,
                                 number& best)
{
   priority_queue_of_node Q;
   node u,v;
```

```
    initialize(PQ);
    v = root of T;
    best = value(v);
    insert(PQ, v);
    while (! empty(Q)){
      remove(PQ, v);
      for (each child u of v){
        if (value(u) is better than best)
          best = value(u);
        if (bound(u) is better than best)
          insert(PQ, u);
      }
    }
}
```

علاوه بر استفاده از صف اولویت، یک بررسی هم به دنبال حذف یک گره از صف اولویت به الگوریتم اضافه نمودیم. این بررسی مشخص می‌کند که آیا هنوز هم حد گره بهتر از best است یا خیر، و این روشی است که می‌فهمیم آیا یک گره پس از ملاقات، غیروعدگاه می‌شود یا خیر؟ به عنوان مثال، گرهٔ (۲، ۱) در شکل ۳-۶ در زمانی که آن را ملاقات می‌کنیم، وعده‌گاه است.در پیاده‌سازی، این موضوع زمانی اتفاق می‌افتد که آن را داخل صف PQ قرار می‌دهیم. اما هنگامی که maxprofit بیش از $۹۰ می‌شود، غیروعده‌گاه می‌شود و این موضوع در پیاده‌سازی، قبل از حذف گره از PQ انجام می‌شود. در این روش، از بررسی فرزندان گره‌ای که بعد از بررسی غیروعده‌گاه شده است، خودداری می‌کنیم.

در ادامه، الگوریتم خاصی را برای مسئله کوله‌پشتی ۱-۰ ارائه می‌دهیم. از آنجائیکه در هنگام عملیات درج، حذف و مرتب‌سازی گره‌ها در صف اولویت به حدّ آن گره نیاز داریم، لذا آن را در گره ذخیره می‌کنیم. تعریف نوع گره به صورت زیر است:

```
struct node              // سطح گره در درخت
{
  int level;
  int profit;
  int weight;
  float bound;
};
```

الگوریتم ۲-۶ **الگوریتم جستجوی اول-بهترین با هرس شاخه و حد برای مسئله کوله‌پشتی ۱-۰**

مسئله: فرض کنید n کالا داریم که هر کدام دارای وزن و ارزشی معین می‌باشند. وزنها و ارزشها، اعدادی صحیح و مثبت هستند. مجموعه‌ای از کالاها با ماکزیمم مجموع ارزشها را مشخص کنید بطوری که مجموع اوزان آنها بیش از عدد صحیح مثبت W نشود.

ورودی: اعداد صحیح مثبت n و W، آرایه‌هایی از اعداد صحیح مثبت p و w که از ۱ تا n شاخص‌دهی شده و بترتیب غیرصعودی براساس مقادیر $p[i]/w[i]$ مرتب شده‌اند.

خروجی: یک عدد صحیح maxprofit که عبارت است از مجموع ارزشهای یک مجموعه بهینه.

```
void knapsack3 (int n,
                const int p[ ], const int w[ ],
                int W,
                int& maxprofit)
{
  priority_queue_of_node PQ;
  node u, v;

  initialize(PQ);                                  // Initialize PQ to be empty.
  v.level = 0; v.profit = 0; v.weight = 0;         Initialize v to be the root.
  maxprofit = 0;
  v.bound = bound(v);
  insert(PQ, v);
  while (! empty(PQ)) {                            // Remove node with
    remove(PQ, v);                                 // best bound.
    if (v.bound > maxprofit) {                     // Check if node is still
      u.level = v.level + 1;                       // promising.
      u.weight = v.weight + w[u.level];            // Set u to the child
      u.profit = v.profit + p[u.level];            // that includes the
      if (u.weight <= W && u.profit > maxprofit)   // next item.
        maxprofit = u.profit;
      u.bound = bound(u);
      if (u.bound > maxprofit)
        insert(PQ, u);
      u.weight = v.weight;                         // Set u to the child
      u.profit = v.profit;                         // that does not include
      u.bound = bound(u);                          // the next item.
      if (u.bound > maxprofit)
        insert(PQ, u);
    }
  }
}
```

* تابع bound همان تابعی است که در الگوریتم ۱-۶ آمده است.

۲-۶ مسئله فروشندهٔ دوره‌گرد

در مثال ۱۲-۳، نانسی توانست با ارائه یک تور بهینه برای فعالیت فروش در ۲۰ شهر با استفاده از یک الگوریتم برنامه‌نویسی پویای $\theta(n^2 2^n)$ در زمان ۴۵ ثانیه بر رالف پیروز شود. رالف از یک الگوریتم

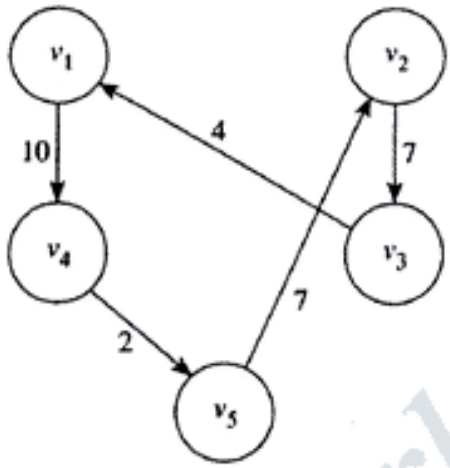
brute-force که تعداد !۱۹ تور را تولید می‌کرد، استفاده نمود که اجرای آن ۳۸۰۰ سال طول می‌کشد و هنوز هم در حال اجراست! در بخش ۶-۵ دیدیم که وقتی تعداد شهرها به ۴۰ شهر افزایش یافت، الگوریتم نانسی برای پیدا کردن یک تور بهینه به بیش از ۶ سال وقت نیاز داشت. در اینجا، او از یک الگوریتم یک‌تراکینگ برای مسئله چرخه‌های هامیلتونی استفاده کرد. اگرچه این الگوریتم، با کارایی بالایی یک تور را مشخص می‌کند، ولی این تور می‌توانست از یک تور بهینه، بسیار دور باشد. به عنوان مثال، اگر یک جادهٔ پرپیچ‌وخم طولانی به طول ۱۰۰ مایل بین دو شهری که فقط ۲۰ مایل با هم فاصله داشتند، وجود می‌داشت این الگوریتم می‌توانست یک تور شامل این جاده را تولید نماید، حتی اگر این امکان وجود داشت که این دو شهر بتوانند به وسیله شهری که فاصله آن با هر کدام از آن دو شهر یک مایل است، به هم مرتبط شوند. لذا نانسی پی‌برد که با استفاده از الگوریتم یک‌تراکینگ نمی‌تواند به صورت کارا به یک تور بهینه دست یابد. او تصمیم می‌گیرد تکنیک شاخه و حد را در مورد مسئله فروشنده دوره‌گرد بکار بگیرد که این تکنیک احتمالاً به صورت زیر عمل خواهد کرد.

به خاطر دارید که هدف ما در این مسئله، یافتن کوتاهترین مسیر در یک گراف جهت‌دار است که از یک گرهٔ معین آغاز شده و هر رأس (گره) گراف را دقیقاً یکبار ملاقات نموده و به همان گرهٔ آغازین ختم می‌شود. به چنین مسیری **تور بهینه** اطلاق میشود. از آنجائیکه مهم نیست که مسیرها از کدام رأس آغاز می‌شوند، ما می‌توانیم اولین رأس را به عنوان رأس آغازین در نظر بگیریم. شکل ۴-۶، ماتریس مجاور برای نمایش گرافی شامل ۵ رأس را نشان می‌دهد که در این گراف از هر رأس به رأس دیگر یک لبه وجود دارد و همچنین یک تور بهینه را برای آن گراف نمایش می‌دهد.

یک درخت فضای حالات برای این مسئله می‌تواند بدینصورت باشد که هر رأس، غیر از رأس آغازین، به عنوان اولین رأس (پس از رأس آغازین) در سطح ۱ آزمایش می‌شود؛ هر رأس، غیر از رأس آغازین و رأسی که در سطح ۱ انتخاب شده، به عنوان رأس دوم در سطح ۲ آزمایش می‌شود و الی آخر. بخشی از این درخت فضای حالت که در آن ۵ رأس وجود دارد و از هر رأسی به رأس دیگر یک لبه وجود دارد، در شکل ۵-۶ نمایش داده شده است. در ادامه، لفظ "گره"، به گره‌ای در درخت فضای حالت و لفظ "رأس"، به رأسی در گراف اطلاق میشود. در هر گره از شکل ۵-۶، ما مسیر انتخاب شده تا رسیدن به آن گره را در نظر گرفته‌ایم. برای سادگی کار، هر رأس گراف را با اندیس آن مشخص می‌کنیم. گره‌ای که برگ نباشد نمایانگر همه تورهایی است که با مسیر ذخیره شده در آن گره شروع شده است. به عنوان مثال، گرهٔ شامل [۱، ۲، ۳] مبین تمامی تورهایی است که با مسیر [۱، ۲، ۳] آغاز می‌شوند. یعنی اینکه آن معرف تورهای [۱، ۲، ۳، ۴، ۵، ۱] و [۱، ۲، ۳، ۵، ۴، ۱] می‌باشد. هر برگ، معرف یک تور است. ما بایستی برگی را پیدا کنیم که شامل یک تور بهینه باشد. وقتی که چهار رأس در مسیر ذخیره شده در یک گره وجود داشته باشد، توسعهٔ درخت را متوقف می‌کنیم زیرا در آن زمان پنجمین رأس به‌خودی‌خود مشخص می‌شود. مثلاً چپ‌ترین برگ، معرف تور [۱، ۲، ۳، ۴، ۵، ۱] است زیرا وقتی که مسیر [۱، ۲، ۳، ۴] را مشخص کردیم، رأس بعدی لزوماً پنجمین رأس گراف است.

شکل ۴-۶ ماتریس مجاور معرف گرافی که در آن از هر رأس، لبه‌ای به هر رأس دیگر وجود دارد (سمت چپ)، و گره‌های گراف و لبه‌های یک تور بهینه در آن (سمت راست)

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$



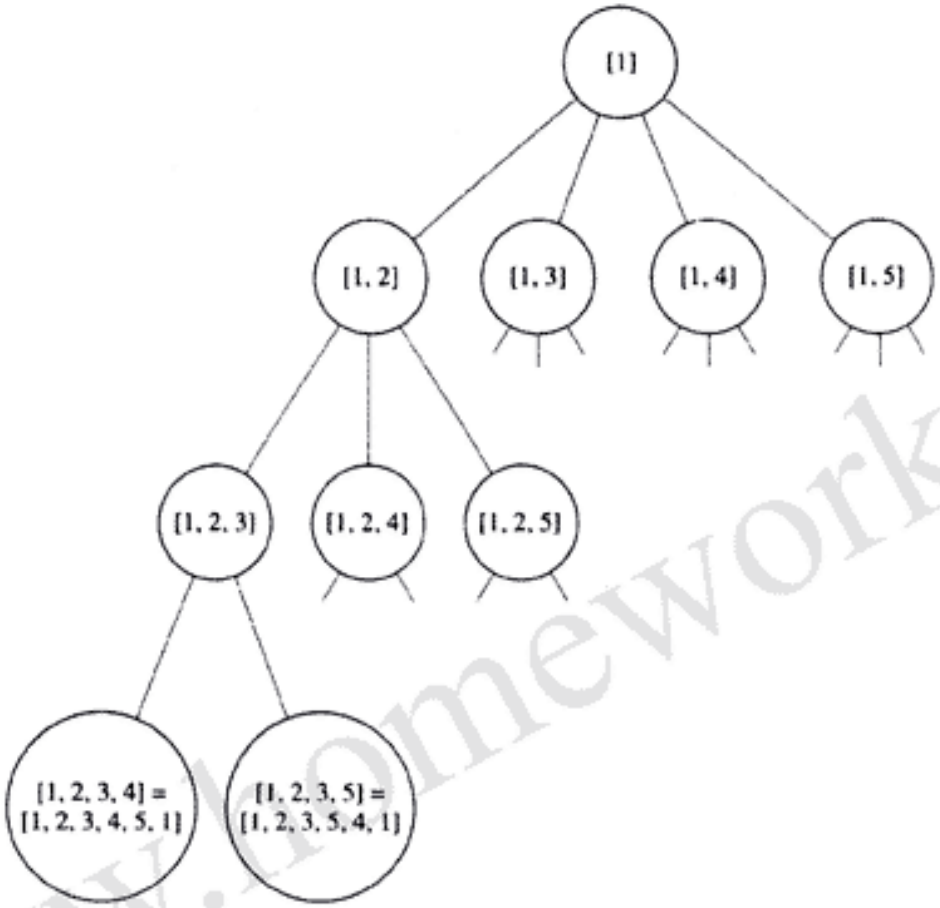
برای آنکه از جستجوی اول-بهترین استفاده کنیم، باید حدی را برای هر گره پیدا کنیم. به خاطر هدف مسئله کوله‌پشتی ۱-۰ (به حداکثر رساندن ارزش، در حالیکه مجموع وزن بیش از W نشود)، حد بالایی را برای مقدار ارزشی که می‌توانست از بسط گره بدست آید، محاسبه نمودیم و یک گره را در صورتی وعده‌گاه نامیدیم که حد آن بیش از ارزش ماکزیمم کنونی باشد. در این مسئله، بایستی یک حد پائین برای طول هر توری که از بسط یک گره بدست می‌آید، پیدا کنیم و گره را در صورتی وعده‌گاه می‌نامیم که حد آن کمتر از طول تور می‌نیمم کنونی باشد. یک حد را می‌توانیم به روش زیر پیدا کنیم. در هر تور، طول لبهٔ گرفته شده به هنگام گذر از یک رأس بایستی حداقل به بزرگی طول کوتاهترین لبه‌ای که از آن رأس ناشی می‌شود، باشد. بنابراین، یک حد پائین روی هزینهٔ (طول لبه گرفته شده) رأس V_1، توسط می‌نیمم تمام ورودیهای غیرصفر در سطر ۱ ماتریس مجاور بدست می‌آید، یک حد پائین روی هزینهٔ رأس V_2 توسط می‌نیمم تمام ورودیهای غیرصفر در سطر ۲ بدست می‌آید و به همین ترتیب، الی آخر. حدود پائین روی هزینه‌های پنج رأس در گراف شکل ۴-۶ به صورت زیر می‌باشند:

$$\begin{aligned} V_1 \quad & minimum(۱۴، ۴، ۱۰، ۲۰) = ۴ \\ V_2 \quad & minimum(۱۴، ۷، ۸، ۷) = ۷ \\ V_3 \quad & minimum(۴، ۵، ۷، ۱۶) = ۴ \\ V_4 \quad & minimum(۱۱، ۷، ۹، ۲) = ۲ \\ V_5 \quad & minimum(۱۸، ۷، ۱۷، ۴) = ۴ \end{aligned}$$

از آنجائیکه یک تور بایستی از هر رأس دقیقاً یکبار عبور کند، یک حد پائین روی طول تور عبارت است از مجموع این می‌نیمم‌ها. بنابراین، یک حد پائین بر روی طول یک تور عبارتست از

$$۴ + ۷ + ۴ + ۲ + ۴ = ۲۱$$

شکل ۵-۶ یک درخت فضای حالت برای نمونه‌ای از مسئله فروشنده دوره گرد که در آن پنج رأس وجود دارد. شاخصهای رئوس در تور جزئی در هر گره ذخیره شده است.



این بدین معنا نیست که توری با این طول وجود دارد؛ بلکه مفهوم آن این است که نمی‌توان توری با طول کمتر پیدا کرد. فرض کنید گرهٔ شامل [۱، ۲] در شکل ۵-۶ را ملاقات کرده‌ایم. در چنین حالتی، قبلاً متعهد شدیم که V_2 را به عنوان دومین رأس تور معرفی کنیم. هزینه حرکت به V_2 برابراست با وزن لبه‌ای که از V_1 به V_2 کشیده شده است، یعنی ۱۴. هر توری که با بسط و توسعهٔ این گره حاصل شود، حدود پائین زیر را روی هزینهٔ عبور از رئوس دیگر خواهد داشت:

V_1 ۱۴

V_2 $minimum(۷، ۸، ۷) = ۷$

V_3 $minimum(۴، ۷، ۱۶) = ۴$

V_4 $minimum(۱۱، ۹، ۲) = ۲$

V_5 $minimum(۱۸، ۱۷، ۴) = ۴$

برای بدست آوردن می‌نیمم برای V_2، دیگر لبهٔ V_1 را درنظر نمی‌گیریم زیرا V_2 نمی‌تواند به V_1 برگردد. برای بدست آوردن می‌نیمم رئوس دیگر، لبهٔ منتهی به V_2 را درنظر نمی‌گیریم زیرا قبلاً در V_2 بوده‌ایم. یک حد پائین برای طول هر تور (که با بسط گرهٔ شامل [۲، ۱۰] بدست آمده)، برابر است با مجموع این می‌نیمم‌ها، یعنی

$$۱۴ + ۷ + ۴ + ۲ + ۴ = ۳۱$$

برای توضیح بیشتر روش تعیین یک حد، فرض کنید گرهٔ شامل [۳، ۲، ۱۰] در شکل ۵-۶ را ملاقات کرده‌ایم. ما متعهد شده‌ایم که V_2 را به عنوان دومین رأس و V_3 را به عنوان سومین رأس درنظر بگیریم. هر تور بدست آمده از بسط این گره دارای حدود پائین زیر بر روی هزینه‌های گذر از رئوس است:

V_1	۱۴
V_2	۷
V_3	$minimum(۷، ۱۶) = ۲$
V_4	$minimum(۱۱، ۲) = ۲$
V_5	$minimum(۱۸، ۴) = ۴$

برای یافتن می‌نیمم‌هایی برای V_2 و V_5، لبه‌های منتهی به V_2 و V_3 را درنظر نمی‌گیریم زیرا قبلاً در این رئوس قرار داشتیم. حد پائین بر روی طول هر تور که با بسط گره‌ای شامل [۳، ۲، ۱۰] بدست می‌آید، بدین صورت است:

$$۱۴ + ۷ + ۷ + ۲ + ۴ = ۳۴$$

به طریق مشابه، می‌توانیم حد پائینی برای طول تور که می‌تواند با بسط یک گره دیگر درخت فضای حالت بوجود آید را محاسبه نموده و این حدود را در جستجوی اول-بهترین بکار می‌گیریم. مثال زیر این روش را نشان می‌دهد.

مثال ۳-۶ گراف شکل ۴-۶ را درنظر بگیرید. با استفاده از بحث‌های گذشته، جستجوی اول-بهترین با هرس شاخه و حد، درخت شکل ۶-۶ را تولید می‌کند. در هر گرهٔ غیر برگ، حد آن گره و در هر برگ، طول یک تور ذخیره می‌شود. می‌خواهیم مراحل ایجاد این درخت را نشان دهیم. ابتدا مقدار بهترین جواب را ∞ (بی‌نهایت) درنظر می‌گیریم زیرا هیچ جوابی در ریشه وجود ندارد (جوابها در درخت فضای حالت، فقط در برگها هستند). لازم به ذکر است که ما حدود برگهای درخت فضای حالت را محاسبه نمی‌کنیم زیرا الگوریتم به صورتی نوشته شده است که برگها را بسط نمی‌دهد. هنگامی که به یک گره رجوع می‌کنیم، در واقع به بخشی از تور که در آن گره ذخیره شده است، اشاره می‌کنیم و این متفاوت است با روشی که در مسئله کوله‌پشتی ۱-۰ به یک گره اشاره می‌نمودیم. این مراحل به صورت زیر است :

۱- گرهٔ شامل [۱] (ریشه) را ملاقات کنید.
(a) حد آن را محاسبه نمائید تا برابر ۲۱ شود. {این یک حد پائین برای طول یک تور است}
(b) مقدار ∞ را به minlength تخصیص دهید.

۲- گرهٔ شامل [۲، ۱] را ملاقات کنید.
(a) حد آن را محاسبه نمائید تا برابر ۳۱ شود.

۳- گرهٔ شامل [۳، ۱] را ملاقات کنید.
(a) حد آن را محاسبه نمائید تا برابر ۲۲ شود.

۴- گرهٔ شامل [۴، ۱] را ملاقات کنید.
(a) حد آن را محاسبه نمائید تا برابر ۳۰ شود.

۵- گرهٔ شامل [۵، ۱] را ملاقات کنید.
(a) حد آن را محاسبه نمائید تا برابر ۴۲ شود.

۶- گرهٔ توسعه‌نیافته وعده‌گاه با کمترین حد را تعیین کنید.
(a) آن گره، گرهٔ شامل [۳، ۱] است. فرزندان آن را بعداً ملاقات خواهیم کرد.

۷- گرهٔ شامل [۲، ۳، ۱] را ملاقات کنید.
(a) حد آن را محاسبه نمائید تا برابر ۲۲ شود.

۸- گرهٔ شامل [۴، ۳، ۱] را ملاقات کنید.
(a) حد آن را محاسبه نمائید تا برابر ۲۷ شود.

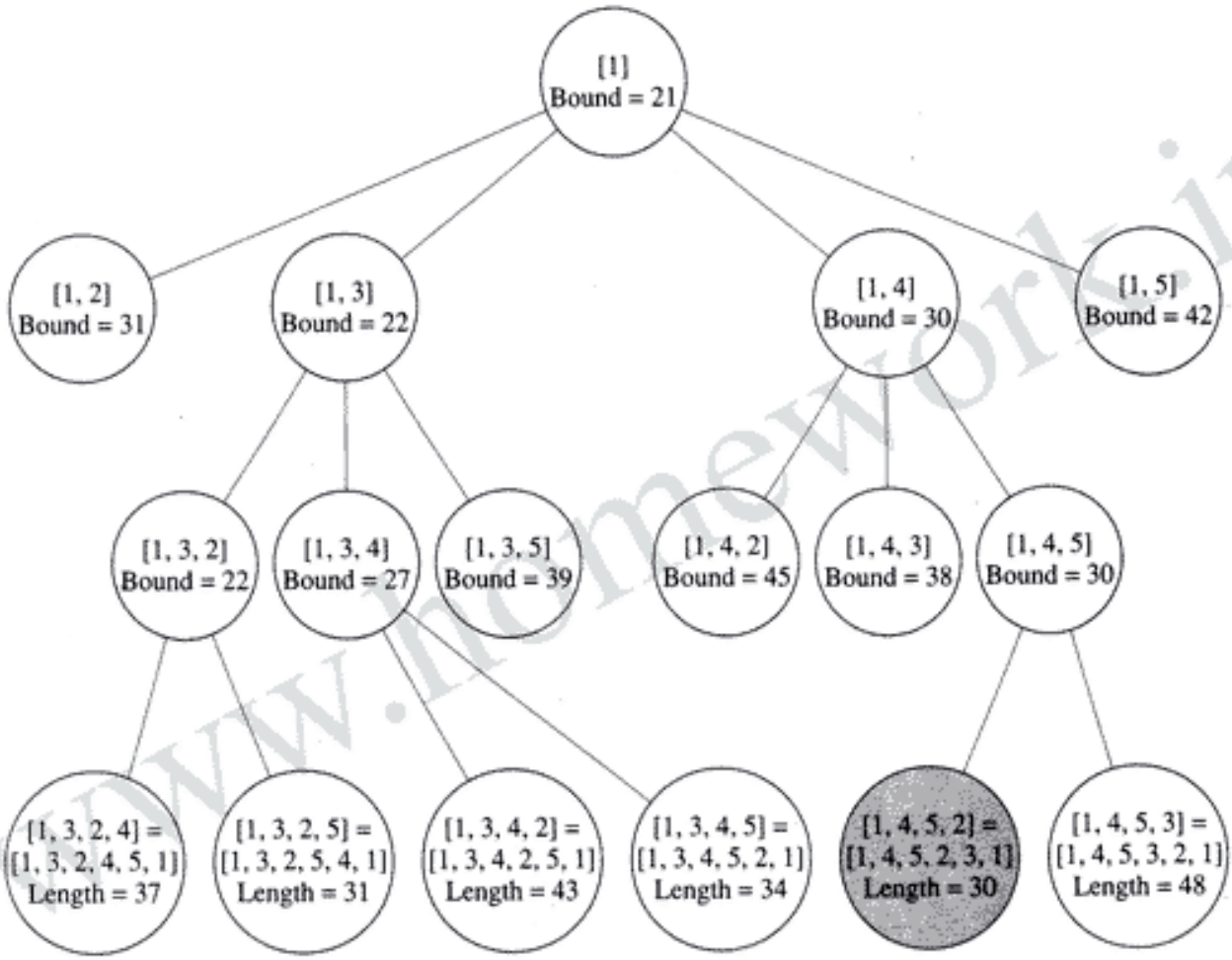
۹- گرهٔ شامل [۵، ۳، ۱] را ملاقات کنید.
(a) حد آن را محاسبه نمائید تا برابر ۳۹ شود.

۱۰- گرهٔ توسعه‌نیافتهٔ وعده‌گاه با کمترین حد را تعیین کنید.
(a) آن گره، گرهٔ شامل [۲، ۳، ۱] است. فرزندان آن را بعداً ملاقات خواهیم کرد.

۱۱- گرهٔ شامل [۴، ۲، ۳، ۱] را ملاقات کنید.
(a) چون این گره یک برگ است، طول تور را محاسبه نمائید تا ۳۷ حاصل شود.
(b) چون طول آن (۳۷) کمتر از ∞ (مقدار minlength) است، مقدار ۳۷ را به minlength تخصیص دهید.
(c) گره‌های شامل [۵، ۱] و [۵، ۳، ۱] غیروعده‌گاه می‌شوند زیرا حدود آنها ۴۲ و ۳۹ بزرگتر یا مساوی ۳۷ (مقدار جدید minlength) است.

۱۲- گرهٔ شامل [۵، ۲، ۳، ۱] را ملاقات کنید.

شکل ۶-۶ درخت فضای حالت هرس شده که حاصل از استفاده جستجوی اول-بهترین با هرس شاخه و حد برای مثال ۲-۶ می‌باشد. در هر گرهٔ غیر برگ در درخت فضای حالت، قسمتی از تور در بالای گره و حد طول هر توری که می‌توانست با بسط گره بدست آید در پائین گره ذخیره شده است. در هر برگ درخت فضای حالت، تور در بالا و طول آن در پائین نشان داده شده است. گرهٔ سایه‌دار، گره‌ای است که تور بهینه در آن پیدا شده است.



(a) چون این گره یک برگ است، طول تور را محاسبه نمائید تا برابر ۳۱ شود.

(b) چون طول آن (۳۱) کمتر از ۳۷ (مقدار minlength) است، مقدار ۳۱ را به minlength تخصیص دهید.

(c) گرهٔ شامل [۲ ،۱] غیروعده‌گاه می‌شود زیرا حد آن (۳۱) بزرگتر یا مساوی ۳۱ (مقدار جدید minlength) است.

۱۳- گرهٔ توسعه‌نیافتهٔ وعده‌گاه با کمترین حد را تعیین کنید.

(a) آن گره، گرهٔ شامل [۴ ،۳ ،۱] است. بعداً فرزندش را ملاقات می‌کنیم.

۱۴- گرهٔ شامل [۱، ۳، ۴، ۲] را ملاقات کنید.

(a) چون این گره یک برگ است، طول تور را محاسبه نمائید تا برابر ۴۳ شود.

۱۵- گرهٔ شامل [۱، ۳، ۴، ۵] را ملاقات کنید.

(a) چون این گره یک برگ است، طول تور را محاسبه نمائید تا برابر ۳۴ شود.

۱۶- گرهٔ توسعه‌نیافتهٔ وعده‌گاه با کوچکترین حد را تعیین کنید.

(a) تنها گرهٔ توسعه‌نیافته وعده‌گاه، گرهٔ شامل [۱، ۴] است. فرزندانش را بعداً ملاقات می‌کنیم.

۱۷- گرهٔ شامل [۱، ۴، ۲] را ملاقات کنید.

(a) حد آن را محاسبه نمائید تا برابر ۴۵ شود.

(b) آن را به عنوان یک گرهٔ غیروعده‌گاه مشخص کنید زیرا حد آن (۴۵) بزرگتر یا مساوی ۳۱ (مقدار minlength) است.

۱۸- گرهٔ شامل [۱، ۴، ۳] را ملاقات کنید.

(a) حد آن را محاسبه نمائید تا برابر ۳۸ شود.

(b) آن را به عنوان یک گرهٔ غیروعده‌گاه تعیین کنید زیرا حد آن (۳۸) بزرگتر یا مساوی ۳۱ (مقدار minlength) است.

۱۹- گرهٔ شامل [۱، ۴، ۵] را ملاقات کنید.

(a) حد آن را محاسبه نمائید تا برابر ۳۰ شود.

۲۰- گرهٔ وعده‌گاه توسعه‌نیافته با کوچکترین حد را تعیین کنید.

(a) تنها گرهٔ وعده‌گاه توسعه‌نیافته، گرهٔ شامل [۱، ۴، ۵، ۲] است. بعداً فرزندش را ملاقات می‌کنیم.

۲۱- گرهٔ شامل [۱، ۴، ۵، ۲] را ملاقات کنید.

(a) چون این گره یک برگ است، طول تور را محاسبه نمائید تا برابر ۳۰ شود.

(b) چون طول آن (۳۰) کوچکتر از ۳۱ (مقدار minlength) است، مقدار ۳۰ را به minlength تخصیص دهید.

۲۲- گرهٔ شامل [۱، ۴، ۵، ۳] را ملاقات کنید.

(a) چون این گره یک برگ است، طول تور را محاسبه نمائید تا برابر ۴۸ شود.

۲۳- گرهٔ وعده‌گاه توسعه‌نیافته با کوچکترین حد را تعیین کنید.

(a) دیگر هیچ گرهٔ وعده‌گاه توسعه‌نیافته‌ای وجود ندارد، لذا عملیات خاتمه می‌یابد.

بدین ترتیب تعیین کرده‌ایم که گرهٔ شامل [۱، ۴، ۵، ۲] (که معرف مسیر [۱، ۳، ۲، ۵، ۴، ۱] است)، شامل یک تور بهینه است که طول این تور بهینه برابر ۳۰ می‌باشد.

تعداد ۱۷ گره در درخت شکل ۶-۶ وجود دارد در حالیکه تعداد گره‌ها در درخت فضای حالت کامل برابر است با ۴۱ = ۲×۳×۴ +۳×۴ + ۴ +۱. ما از نوع داده‌ای زیر در الگوریتمی که استراتژی مورد استفاده در مثال قبل را پیاده‌سازی می‌کند، استفاده می‌کنیم :

```
struct node
{
   int level;              // سطح گره در درخت
   ordered_set path;
   number bound;
};
```

فیلد path شامل تور ناتمام ذخیره شده در آن گره است. برای مثال، در شکل ۶-۶ مقدار path برای فرزند چپ ریشه، [۱، ۲] است. الگوریتم به صورت زیر است.

الگوریتم ۶-۳ الگوریتم جستجوی اول-بهترین با هرس شاخه و حد برای مسئله فروشندهٔ دوره‌گرد

مسئله: یک تور بهینه را در یک گراف وزن‌دار و جهت‌دار مشخص کنید. وزنها، اعدادی غیرمنفی هستند.

ورودی: یک گراف وزن‌دار و جهت‌دار، n تعداد رأسهای گراف. گراف به وسیله یک آرایهٔ دوبعدی W ارائه می‌شود که سطرها و ستونهایش از ۱ تا n شاخص‌دهی شده و w[i][j]، معرف وزن لبه رأس iام به رأس jام است.

خروجی: متغیر minlength که مقدار آن، طول یک تور بهینه است و متغیر opttour که مقدار آن تور بهینه است.

```
void travel2 (int n,
              const number W[ ] [ ],
              ordered-set& opttour,
              number& minlength)
{
  priority_queue_of_node PQ; node u, v;

  initialize(PQ);                    // InitializePQ to be empty.
  v.level = 0;
  v.path = [1];                      // Make first vertex the starting
  v.bound = bound(v);                // one.
  minlength = ∞;
  insert(PQ, v);
  while (! empty(PQ)) {
    remove(PQ, v);                   // Remove node with best bound.
```

```
                                        // Remove node with best bound.
    if (v.bound < minlength) {
      u.level = v.level + 1;            // Set u to a child of v.
      if (u.level == n - 1) {           // Check if u completes a tour.
        u.path = v.path;
        put 1 at the end of u.path;     // Make first vertex the last one.
        if (length(u) < minlength) {    // Function length computes the
          minlength = length(u);        // length of the tour.
          opttour = u.path;
        }
      }
      else
        for (all i such that 2 ≤ i ≤ n && i is not in v.path) {
          u.path = v.path;
          put i at the end of u.path;
          u.bound = bound(u);
          if (u.bound < minlength)
            insert(PQ, u);
        }
    }
  }
```

در تمرینات از شما می‌خواهیم که توابع length و bound را بنویسید. تابع length، طول تور u.path و تابع bound، حد یک گره را برمی‌گرداند. یک مسئله، لزوماً یک تابع حد منحصر به فرد نیست. به عنوان مثال، در مسئله فروشندهٔ دوره‌گرد دیدیم که هر رأس بایستی دقیقاً یکبار ملاقات شود، در اینصورت می‌توانیم از می‌نیمم‌های مقادیر ستونها در ماتریس مجاور به جای می‌نیمم‌های مقادیر رئوس استفاده کنیم. با توجه به اینکه هر رأس بایستی دقیقاً یکبار وارد و خارج شود، می‌توانستیم هم از مزایای سطرها و هم از مزایای ستونها استفاده کنیم. برای یک لبهٔ معین، می‌توانستیم نیمی از وزن آن را برای رأسی که آن را ترک می‌کنیم و نیمی دیگر را برای رأسی که وارد آن می‌شویم در نظر بگیریم که در اینصورت، هزینهٔ ملاقات یک رأس برابر است با مجموع اوزان ورودی و خروجی آن رأس. بعنوان مثال، فرض کنید که می‌خواهیم حدی را برای طول یک تور تعیین کنیم. کمترین هزینه ورودی $V_۲$، با در نظر گرفتن ۱/۲ می‌نیمم مقادیر ستون دوم قابل محاسبه است. حداقل هزینهٔ خروج از $V_۲$ برابر است با ۱/۲ می‌نیمم مقادیر دومین سطر. بدین ترتیب، حداقل هزینهٔ ملاقات $V_۲$ برابر است با

$$\frac{minimum(۱۴, ۵, ۷, ۷) + minimum(۱۴, ۷, ۸, ۷)}{۲} = ۶$$

با استفاده از این تابع حدی، یک الگوریتم شاخه و حد تنها ۱۵ رأس را در نمونه مثال ۳-۶ بررسی می‌کند. هنگامی که دو یا چند تابع حدی موجود باشد، یک تابع حدی ممکن است حدّ بهتری را برای یک گره تولید کند؛ در حالیکه تابع دیگر، حدّ بهتری را برای گرهٔ دیگر تولید نماید. در واقع، همانطوریکه در تمرینات بررسی خواهیم کرد، این حالت برای توابع حدی مسئله فروشندهٔ دوره‌گرد برقرار است. هنگامی که این حالت اتفاق می‌افتد، الگوریتم می‌تواند با استفاده از همهٔ توابع حدی موجود، حدود را محاسبه نموده و

سپس بهترین حد را بکار بگیرد. به هر حال، همانطوریکه در فصل ۵ بحث شد، هدف ما این نیست که تا حد امکان تعداد گره‌های کمتری را ملاقات کنیم؛ بلکه هدف، افزایش کارایی الگوریتم می‌باشد. محاسبات اضافی انجام شده به هنگام استفاده از دو یا چند تابع حدی ممکن است موجب صرفه‌جویی، نسبت به زمانی که گره‌های کمتری را ملاقات می‌کنیم، نشود.

به خاطر دارید که یک الگوریتم شاخه و حد ممکن بود یک نمونه بزرگ را به صورت کارایی حل کند؛ اما یک تعداد نمایی (یا بدتر) از گره‌ها را برای نمونه‌های بزرگ دیگر بررسی می‌کرد. حال شما فکر می‌کنید که نانسی چه کاری باید انجام دهد اگر حتی شاخه و حد هم نتواند نمونه مسئله ۴۰-شهری او را حل نماید؟ یک روش دیگر برای حل مسائلی نظیر فروشندهٔ دوره‌گرد، استفاده از الگوریتم‌های تقریبی است. **الگوریتم‌های تقریبی**، ارائه جوابهای بهینه را تضمین نمی‌کنند اما جوابهایی تولید می‌کنند که نسبتاً نزدیک به جواب بهینه است. این الگوریتم‌ها در بخش ۵-۹ مورد بررسی قرار گرفته‌اند.

۶-۳ استنتاج ربایشی یا مسئله تشخیص بیماری

این بخش به اطلاعاتی راجع به نظریهٔ احتمالات گسسته و قضیه Bayes نیاز دارد. یک مسئله مهم در هوش مصنوعی و سیستم‌های خبره، تعیین محتمل‌ترین تعریف دربارهٔ برخی یافته‌ها است. برای مثال، در پزشکی می‌خواهیم محتمل‌ترین مجموعهٔ بیماریها را برای یک مجموعه از علائم بیماری مشخص کنیم یا در یک مدار الکترونیکی می‌خواهیم محتمل‌ترین بیان را در مورد شکست در یک نقطه از مدار پیدا کنیم. مثال دیگر، تعیین محتمل‌ترین دلیل برای درست کار نکردن یک اتومبیل است. این فرآیند تعیین محتمل‌ترین بیان برای یک مجموعه از یافته‌ها را **استنتاج ربایشی (abductive inference)** گوئیم.

برای سهولت کار، از اصطلاحات پزشکی استفاده می‌کنیم. فرض کنید n بیماری $d_1, d_2, \ldots, d_n$ داریم که هر یک ممکن است در یک بیمار وجود داشته باشند. ما می‌دانیم که بیمار دارای مجموعه‌ای از علائم بیماری (S) است. هدف ما، یافتن مجموعه‌ای از بیماریها است که به احتمال زیاد در این بیمار وجود دارد. از لحاظ تکنیکی، می‌توانست احتمال دو یا چند بیماری وجود داشته باشد ولی اغلب مسئله را بصورتی که یک مجموعهٔ منحصر به فرد دارای بیشتری احتمال است، مطرح می‌کنیم.

شبکهٔ فرضی، به یک استاندارد برای معرفی ارتباطات احتمالی نظیر ارتباطات بین بیماریها و علائم تبدیل شده است. در اینجا به میزانی که از بحث خارج نشویم با این شبکه‌ها آشنا خواهیم شد. در بسیاری از کاربردهای شبکه فرضی، الگوریتم‌های کارایی وجود دارند که می‌توانند محتمل‌ترین بیماری در مجموعه‌ای شامل تنها بیماریهای موجود در مریض را مشخص کنند (قبل از بروز علائم بیماری). در اینجا، به طور ساده فرض می‌کنیم که نتایج الگوریتم‌ها قابل دسترس هستند. به عنوان مثال، این الگوریتم‌ها می‌توانند محتمل‌ترین مورد را در بین بیماریهای d_1, d_3, d_6 (تنها بیماریهای موجود در بیمار) مشخص کنند که این احتمال را به صورت $P(d_1, d_3, d_6)$ یا به وسیلهٔ p(D) که در آن $D = (d_1, d_3, d_6)$ است، نشان می‌دهیم.

این الگوریتم‌ها می‌توانند احتمال مناسبی برای d_1, d_2, d_3 به عنوان تنها بیماریهای موجود در بیمار که دارای مجموعه علائم S است، را مشخص کنند. این احتمال را به صورت $P(d_1, d_2, d_3 \mid S)$ یا $P(D \mid S)$ نشان می‌دهیم.

با فرض اینکه بتوانیم این احتمالات را (با استفاده از الگوریتمهایی که قبلاً ذکر شد) محاسبه کنیم. می‌توانیم مسئله تعیین محتمل‌ترین مجموعه بیماریها را با استفاده از درخت فضای حالت، نظیر آنچه که برای مسئله کوله‌پشتی ۱-۰ آمده است، حل کنیم. برای به حساب آوردن d_1 به سمت چپ ریشه و برای به حساب نیاوردن آن، به سمت راست ریشه حرکت می‌کنیم. به طور مشابه، برای به حساب آوردن d_2 به سمت چپ گرهٔ واقع در سطح ۱ و برای به حساب نیاوردن آن به سمت راست گره می‌رویم و به همین ترتیب، الی آخر. هر برگ درخت فضای حالت، معرف یک جواب محتمل (یعنی مجموعه‌ای از بیماریها که تا آن برگ در نظر گرفته شده‌اند) می‌باشد. برای حل مسئله، احتمال شرطی مجموعه بیماریها در هر برگ را محاسبه می‌کنیم و تعیین می‌کنیم که کدامیک دارای بزرگترین احتمال شرطی است.

برای هرس کردن با استفاده از جستجوی اول-بهترین، نیاز به یافتن یک تابع حد داریم. قضیه زیر، این کار را برای گروه بزرگی از نمونه‌ها انجام می‌دهد.

قضیه ۱-۶ اگر D و D' دو مجموعه از بیماریها باشند بطوری که $P(D') \le P(D)$، آنگاه

$$P(D' \mid S) \le P(D)/P(S)$$

اثبات: طبق قضیه Bayes،

$$P(D' \mid S) = \frac{P(S \mid D')P(D')}{P(S)} \le \frac{P(S \mid D')P(D)}{P(S)} \le \frac{P(D)}{P(S)}$$

نامساوی اول، طبق فرض قضیه و نامساوی دوم، بر اساس این واقعیت که هر احتمالی کوچکتر یا مساوی ۱ است، بدست آمده‌اند.

برای یک گرهٔ معین، فرض می‌کنیم که D مجموعه بیماریهایی باشد که بالای این گره قرار دارند و برای برخی از آیندگان (اخلاف) گره، فرض کنید که D' مجموعه‌ای از بیماریهایی باشد که تا آن خَلَف به شمار می‌آیند. در اینصورت $D \subseteq D'$ است. روشن است که وقتی $D \subseteq D'$ است، $P(D') \le P(D)$ می‌باشد. و از آنجائیکه طبق قضیه ۱-۶، $P(D' \mid S) \le P(D)/P(S)$ است، لذا $P(D)/P(S)$ یک حد بالا برای احتمال شرطی مجموعهٔ بیماریها در هر گرهٔ خلف این گره می‌باشد. مثال زیر نشان می‌دهد که چگونه این حد برای هرس کردن شاخه‌ها بکار می‌رود.

مثال ۴-۶ فرض کنید چهار بیماری ممکن d_1 d_2 d_3 و d_4 و یک مجموعه علائم S وجود دارد. ورودی این مثال، یک شبکه فرضی شامل ارتباطات احتمالی میان بیماریها و علائم می‌باشد. احتمالات مورد استفاده در این مثال با استفاده از روشی که در اوایل بحث گفته شد، از این شبکه فرضی محاسبه خواهند شد. ما احتمالات دلخواهی را منتسب می‌کنیم تا بتوانیم الگوریتم جستجوی اول-بهترین را نشان دهیم. هنگام استفاده از نتایج یک الگوریتم (در این مورد، الگوریتمی که عمل استنتاج از شبکهٔ فرضی را انجام می‌دهد) در الگوریتمی دیگر (در این مورد، الگوریتم جستجوی اول-بهترین)، تشخیص این موضوع که الگوریتم اول نتایج را در کجا قرار می‌دهد تا الگوریتم دوم از آنها استفاده کند، مهم است.

شکل ۷-۶، یک درخت فضای حالت هرس شده که به وسیله جستجوی اول-بهترین تولید شده است را نشان می‌دهد. احتمالات داده شده، مقادیری دلخواه در درخت هستند. در هر گره، احتمالات شرطی در بالا و حد آن در پائین قرار دارد. گرهٔ سایه‌دار، گره‌ای است که بهترین جواب در آن پیدا شده است. همانند بخش ۱-۶، گره‌ها بر اساس عمق و موقعیتشان از چپ به راست شماره‌گذاری شده‌اند. مراحل تشکیل درخت به صورت زیر است. متغیر best، بهترین جواب کنونی است و $p(best \mid S)$ احتمال شرطی آن است. هدف ما، تعیین یک مقدار best است که این احتمال شرطی را به حداکثر برساند. همچنین به دلخواه فرض کردیم که $p(s) = ۰/۰۱$ باشد.

۱- گرهٔ (۰،۰)را ملاقات کنید. (ریشه)

(a) احتمال شرطی آن را محاسبه نمائید. {$\emptyset$ مبین مجموعهٔ تهی است؛ یعنی هیچ بیماری وجود ندارد}

$P(\emptyset \mid S) = ۰/۱$ {محاسبات توسط الگوریتمی دیگر انجام می‌شود.}

{ما مقادیر دلخواهی را منتسب کنیم.}

(b) مقداردهی‌های زیر را انجام دهید.

$P(best \mid S) = P(\emptyset \mid S) = ۰/۱$, $best = \emptyset$

(c) احتمال و حد قبلی آن را محاسبه کنید.

$$P(\emptyset) = ۰/۹$$

$$bound = \frac{p(\emptyset)}{P(S)} = \frac{۰/۹}{۰/۰۱} = ۹۰$$

۲- گرهٔ (۱،۱) را ملاقات کنید.

(a) احتمال شرطی آن را محاسبه نمائید. $P(d_1 \mid S) = ۰/۴$

(b) چون $۰/۴ > P(best \mid S)$ است، لذا مقداردهی‌های زیر را انجام دهید.

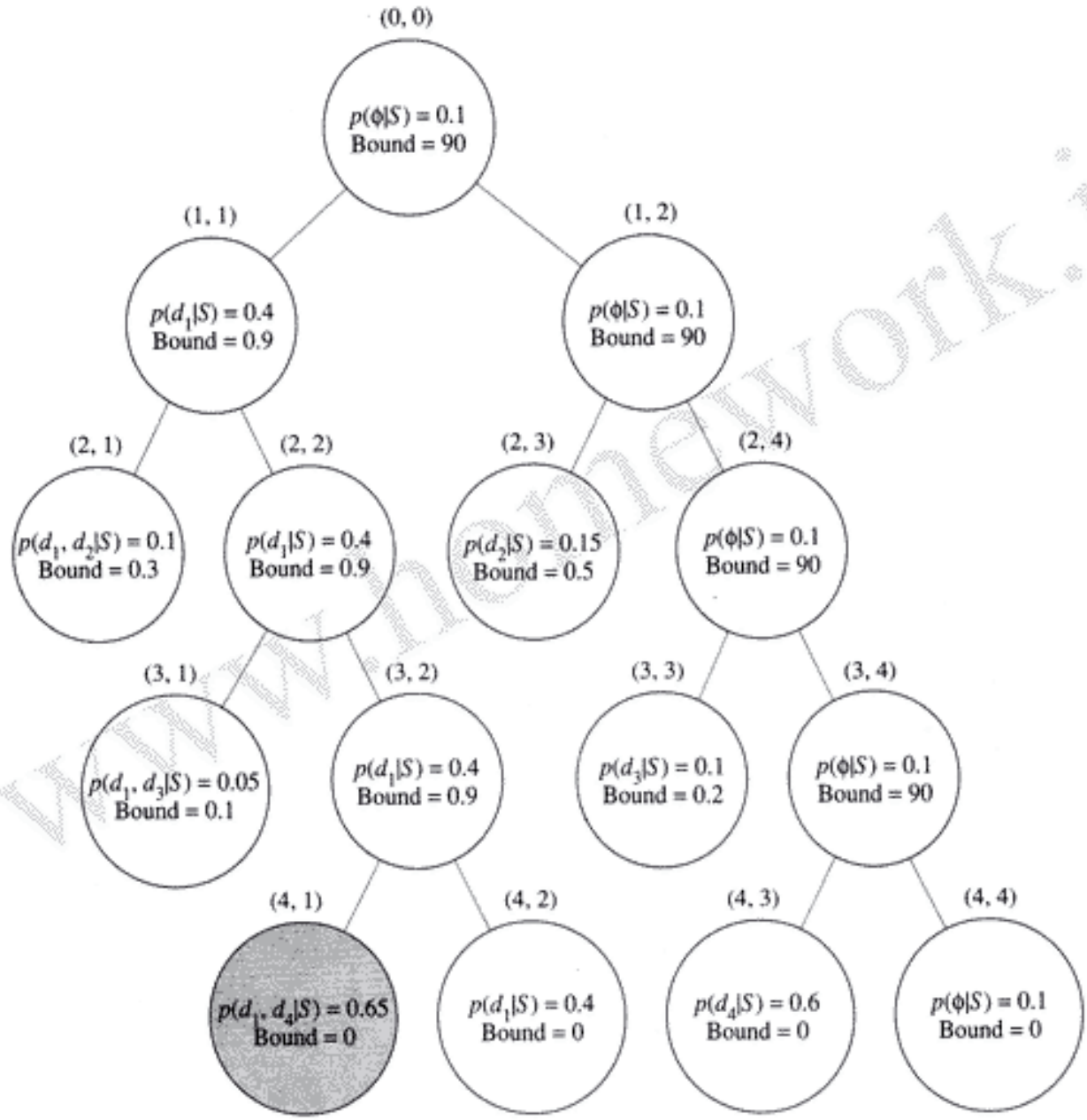
$best = \{d_1\}$, $P(best \mid S) = ۰/۴$

(c) احتمال و حد قبلی آن را محاسبه کنید.

$$p(d_1) = ۰/۰۰۹$$

$$bound = \frac{p(d_1)}{P(s)} = \frac{۰/۰۰۹}{۰/۰۱} = ۰/۹$$

شکل ۷-۶ درخت فضای حالت هرس شده که به وسیله جستجوی اول-بهترین با هرس شاخه و حد برای مثال ۴-۶ تولید شده است. در هر گره احتمال شرطی بیماریهایی که تا آن گره به حساب آمده‌اند، در بالا و حد احتمال شرطی که می‌توانست از بسط آن گره بدست آید در پائین قرار دارد. گرهٔ سایه‌دار، گره‌ای است که یک مجموعه بهینه در آن پیدا شده است.



۳- گرهٔ (۲، ۱) را ملاقات کنید.

(a) احتمال شرطی آن همان احتمال شرطی پدرش (یعنی ۰/۱) است.

(b) احتمال و حد قبلی آن مشابه احتمال و حد پدرش (یعنی ۰/۹ و ۹۰) است.

۴- گرهٔ وعده‌گاه توسعه‌نیافته با بزرگترین حد را تعیین کنید.

(a) آن گره، گرهٔ (۲، ۱) است. فرزندانش را بعداً ملاقات خواهیم کرد.

۵- گرهٔ (۳، ۲) را ملاقات کنید.

(a) احتمال شرطی آن را محاسبه نمائید. $P(d_7 \mid S) = ۰/۱۵$

(b) حد و احتمال قبلی آن را محاسبه نمائید.

$$p(d_7) = ۰/۰۰۵$$

$$bound = \frac{p(d_7)}{P(s)} = \frac{۰/۰۰۵}{۰/۰۱} = ۰/۵$$

۶- گرهٔ (۴، ۲) را ملاقات کنید.

(a) احتمال شرطی آن مشابه احتمال شرطی پدرش (یعنی ۰/۱) است.

(b) احتمال و حد قبلی آن مشابه احتمال و حد پدرش (یعنی ۰/۹ و ۹۰) است.

۷- گرهٔ وعده‌گاه توسعه نیافته با بزرگترین حد را تعیین کنید.

(a) آن گره، گرهٔ (۴، ۲) است. فرزندانش را بعداً ملاقات می‌کنیم.

۸- گرهٔ (۳، ۳) را ملاقات کنید.

(a) احتمال شرطی آن را محاسبه نمائید. $P(d_8 \mid S) = ۰/۱$

(b) احتمال و حد قبلی آن را محاسبه نمائید.

$$P(d_8) = ۰/۰۰۲$$

$$bound = \frac{p(d_8)}{P(s)} = \frac{۰/۰۰۲}{۰/۰۱} = ۰/۲$$

(c) آن را به عنوان یک گرهٔ غیروعده‌گاه مشخص کنید زیرا حد آن ۰/۲ کوچکتر یا مساوی ۰/۴ (مقدار $P(best \mid S)$) است.

۹- گرهٔ (۴، ۳) را ملاقات کنید.

(a) احتمال شرطی آن شبیه پدرش است یعنی ۰/۱

(b) احتمال و حد قبلی آن مشابه احتمال و حد پدرش (یعنی ۰/۹ و ۹۰) است.

۱۰- گرهٔ وعده‌گاه توسعه نیافته با بزرگترین حد را تعیین کنید.

(a) آن گره، گرهٔ (۴، ۳) است. فرزندانش را بعداً ملاقات خواهیم کرد.

۱۱- گرهٔ (۴، ۳) را ملاقات کنید.

(a) احتمال شرطی آن را محاسبه نمائید. $P(d_9 \mid S) = ۰/۶$

(b) چون $P(best \mid S) < ۰/۶$ است، لذا مقداردهی‌های زیر را انجام دهید :

$$best = \{d_۴\} \quad , \quad P(best \mid S) = ۰/۶$$

(c) حد آن را مساوی صفر قرار دهید زیرا آن یک برگ در درخت فضای حالات است.

(d) در این نقطه، گرهٔ (۲، ۳) غیروعده‌گاه می‌شود زیرا حد آن (۰/۵) کوچکتر یا مساوی ۰/۶ (مقدار جدید $P(best \mid S)$) است.

۱۲- گرهٔ (۴، ۴) را ملاقات کنید.

(a) احتمال شرطی آن، همانند پدرش (یعنی ۰/۱) است.

(b) حد آن را مساوی صفر قرار دهید زیرا آن یک برگ در درخت فضای حالت است.

۱۳- گرهٔ وعده‌گاه توسعه نیافته با بزرگترین حد را تعیین کنید.

(a) آن گره، گرهٔ (۱، ۱) است. فرزندانش را بعداً ملاقات خواهیم کرد.

۱۴- گرهٔ (۱، ۲) را ملاقات کنید.

(a) احتمال شرطی آن را محاسبه نمائید.

$$P(d_۱, d_۲ \mid S) = ۰/۱$$

(b) احتمال و حد قبلی آن را محاسبه نمائید.

$$p(d_۱, d_۲) = ۰/۰۰۳$$

$$bound = \frac{p(d_۱, d_۲)}{P(s)} = \frac{۰/۰۰۳}{۰/۰۱} = ۰/۳$$

(c) آن را به عنوان یک گرهٔ غیروعده‌گاه مشخص کنید زیرا حد آن (۰/۳) کوچکتر یا مساوی ۰/۶ (مقدار $P(best \mid S)$) است.

۱۵- گرهٔ (۲، ۲) را ملاقات کنید.

(a) احتمال شرطی آن همانند پدرش (یعنی ۰/۴) است.

(b) احتمال و حد قبلی آن همانند پدرش (یعنی ۰/۰۰۹ و ۰/۹) می‌باشد.

۱۶- گرهٔ وعده‌گاه توسعه‌نیافته با بزرگترین حد را تعیین کنید.

(a) تنها گرهٔ وعده‌گاه توسعه‌نیافته، گرهٔ (۲، ۲) می‌باشد. فرزندانش را بعداً ملاقات خواهیم کرد.

۱۷- گرهٔ (۱، ۳) را ملاقات کنید.

(a) احتمال شرطی آن را محاسبه نمائید.

$$p(d_۱, d_۲ \mid S) = ۰/۰۵$$

(b) احتمال و حد قبلی آن را محاسبه نمائید.

$$p(d_۱, d_۲) = ۰/۰۰۱$$

$$bound = \frac{p(d_۱, d_۲)}{P(s)} = \frac{۰/۰۰۱}{۰/۰۱} = ۰/۱$$

(c) آن را به عنوان یک گرهٔ غیروعده‌گاه مشخص کنید زیرا حد آن (۰/۱) کوچکتر یا مساوی ۰/۶ (مقدار $P(best \mid S)$) است.

۱۸- گرهٔ (۳، ۲) را ملاقات کنید.

(a) احتمال شرطی آن همانند پدرش (یعنی ۰/۴) است.

(b) احتمال و حد قبلی آن همانند پدرش (یعنی ۰/۰۰۹ و ۰/۹) است.

۱۹- گرهٔ وعده‌گاه توسعه‌نیافته با بزرگترین حد را تعیین کنید.

(a) تنها گرهٔ وعده‌گاه توسعه نیافته، گرهٔ (۳، ۲) است. فرزندانش را بعداً ملاقات خواهیم کرد.

۲۰- گرهٔ (۴، ۱) را ملاقات کنید.

(a) احتمال شرطی آن را محاسبه نمائید.

$$p(d_1, d_4 \mid S) = ۰/۶۵$$

(b) چون $P(best \mid S) < ۰/۶۵$ است، بنابراین مقداردهی‌های زیر را انجام دهید :

$$best = \{d_1, d_4\} \quad , \quad P(best \mid S) = ۰/۶۵$$

(c) حد آن را برابر صفر قرار دهید زیرا آن یک برگ در درخت فضای حالت است.

۲۱- گرهٔ (۴، ۲) را ملاقات کنید.

(a) احتمال شرطی آن همانند پدرش (یعنی ۰/۴) است.

(b) حد آن را برابر صفر قرار دهید زیرا آن یک برگ در درخت فضای حالت است.

۲۲- گرهٔ وعده‌گاه توسعه‌نیافته با بزرگترین حد را تعیین کنید.

(a) دیگر گرهٔ وعده‌گاه توسعه نیافته‌ای وجود ندارد. عملیات خاتمه یافته است.

بدین ترتیب، تعیین نموده‌ایم که محتمل‌ترین مجموعه بیماریها $\{d_1, d_4\}$ است که $P\{d_1, d_4 \mid S\} = ۰/۶۵$ می‌باشد.

یک استراتژی معقول در این مسئله این است که بیماریها را بر اساس احتمالات شرطی‌شان به صورت غیرصعودی مرتب کنیم. البته هیچ تضمینی وجود ندارد که این استراتژی، زمان جستجو را به مینیمم برساند. ما این کار را در مثال ۴-۶ انجام ندادیم و در نتیجه ۱۵ گره را بررسی نمودیم. در تمرینات از شما می‌خواهیم نشان دهید که اگر بیماریها به ترتیب فوق مرتب شده بودند، آنگاه ۲۳ گره مورد بررسی قرار می‌گرفت. در ادامه، الگوریتم را ارائه می‌دهیم. این الگوریتم از تعریف زیر استفاده می‌کند :

```
Struct node
{
    int level;              // سطح گره در درخت
    set_of_indices D;
    float bound;
};
```

فیلد D شامل اندیس بیماریها تا بالای گره می‌باشد. یکی از ورودیهای این الگوریتم، شبکه فرضی BN است. همانطوریکه گفته شد، یک شبکه فرضی، ارتباطات احتمالی بین بیماریها و علایم را نشان می‌دهد. الگوریتم ارائه شده در آغاز این بخش می‌تواند احتمالات لازم را از چنین شبکه‌ای محاسبه نماید.

الگوریتم زیر در سال ۱۹۸۶ توسط کوپر ارائه شده و به الگوریتم کوپر (Cooper) نیز مشهور است.

الگوریتم ۴-۶ الگوریتم جستجوی اول-بهترین کوپر با هرس شاخه و حد برای استنتاج ربایشی

مسئله: محتمل‌ترین مجموعه از بیماریها را برای یک مجموعه از علائم مشخص کنید.

فرض می‌شود که اگر مجموعه بیماریهای D، زیر مجموعه‌ای از مجموعه بیماریهای D' باشد آنگاه $P(D') \leq P(D)$ شود.

ورودی: عدد صحیح مثبت n، یک شبکه فرضی BN که نمایانگر ارتباطات احتمالی بین n بیماری و علائم آنها است ، و مجموعه علائم S

خروجی: یک مجموعه به‌نام best که شامل اندیسهای بیماریها در محتمل‌ترین مجموعه (در شرایط S) است و یک متغیر Pbest که احتمال best در S می‌باشد.

```
void cooper (int n,
             belief_network_of_n_diseases BN,
             set_of_symptoms S,
             set_of_indices& best,
             float& pbest)
{
  priority_queue_of_node PQ;
  node u, v;
  v.level = 0;                          // Set v to the root.
  v.D = ∅;                              // Store empty set at root.
  best = ∅;
  pbest = p(∅|S);
  v.bound = bound(v);
  insert(PQ, v);
  while (! empty(PQ)) {
    remove(PQ, v);                      // Remove node with best bound.
    if (v.bound > pbest) {
      u.level = v.level + 1;            // Set u to a child of v.
      u.D = v.D;                        // Set u to the child that includes the
      put u.level in u.D;               // next disease.
      if (p(u.D|S) > pbest) {
        best = u.D;
        pbest = p(u.D|S);
      }
```

```
        u.bound = bound(u);
        if (u.bound > pbest)
          insert(PQ, u);
        u.D = v.D;                    // Set u to the child that does not
        u.bound = bound(u);           // include the next disease.
        if (u.bound > pbest)
          insert(PQ, u);
      }
    }
}

int bound (node u)
{
  if (u.level == n)                   // A leaf is non-promising.
    return 0;
  else
    return p(u.D|p(S);
}
```

p(D) معرف احتمال قبلی D، p(S) معرف احتمال قبلی S و P(D|S) مبین احتمال D در شرایط S است. این مقادیر با توجه به الگوریتم‌های ارائه شده در آغاز این بخش، از شبکه فرضی محاسبه می‌شوند.

ما الگوریتم را کاملاً بر اساس توضیحاتی که برای نوشتن الگوریتم‌های اول-بهترین ارائه شده است، نوشتیم؛ اما امکان اصلاح آن نیز وجود دارد. در این الگوریتم هیچ لزومی به فراخوانی تابع bound برای فرزند راست یک گره نیست زیرا فرزند راست شامل مجموعه مشابهی از بیماریهای خود آن گره است و این بدین معناست که حد آنها نیز با هم برابر است. بنابراین، تنها در صورتی فرزند راست هرس می‌شود که pbest را به حد موجود در فرزند چپ تغییر دهیم. می‌توانیم الگوریتم‌مان را به گونه‌ای تغییر دهیم که در صورت وقوع چنین اتفاقی، فرزند راست را هرس نماید و در غیر اینصورت، فرزند راست را بسط دهد. همانند سایر مسائلی که در این فصل بحث شد، مسئله استنتاج ربایشی در زمرهٔ مسائل مورد بحث در فصل ۹ قرار می‌گیرد.

اگر بیش از یک جواب برای مسئله وجود داشته باشد، الگوریتم قبلی تنها یکی از آنها را تولید می‌کند. روشن است که می‌توانیم الگوریتم را طوری تغییر دهیم که کلیه بهترین جوابها را تولید کند. همچنین می‌توانیم با انجام یک تغییر ساده در الگوریتم، m تا از محتمل‌ترین جوابها را تولید نمائیم که m، هر عدد صحیح مثبت می‌باشد.

تمرینات

بخش ۱-۶

۱- با استفاده از الگوریتم ۱-۶ (جستجوی سطحی با الگوریتم هرس شاخه و حد برای مسئله کوله‌پشتی ۱-۰) ارزش را در نمونه مسئله زیر به حداکثر برسانید. مراحل را قدم به قدم نشان دهید.

i	p_i	w_i	p_i/w_i
۱	$۲۰	۲	$۱۰
۲	$۳۰	۵	$۶
۳	$۳۵	۷	$۵
۴	$۱۲	۳	$۴
۵	$۳	۱	$۳

۲- الگوریتم ۱-۶ را بر روی کامپیوتر خود پیاده‌سازی نموده،آن را بر روی نمونه مسئله تمرین ۱ اجرا کنید.

۳- الگوریتم ۱-۶ را به گونه‌ای تغییر دهید که یک مجموعهٔ بهینه از کالاها را تولید کند. کارایی این الگوریتم را با الگوریتم ۱-۶ مقایسه کنید.

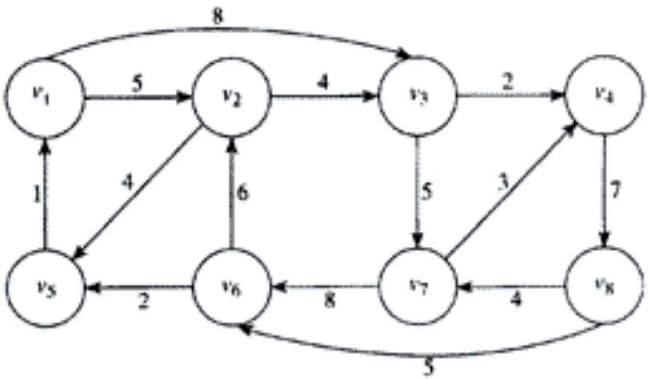
۴- با استفاده از الگوریتم ۲-۶ (جستجوی اول-بهترین با الگوریتم هرس شاخه و حد برای مسئله کوله‌پشتی ۱-۰) ارزش را برای نمونه مسئله تمرین ۱ به حداکثر برسانید. مراحل را قدم به قدم نشان دهید.

۵- الگوریتم ۲-۶ را بر روی کامپیوتر خود پیاده‌سازی نموده،آن را بر روی نمونه مسئله تمرین ۱ اجرا کنید.

۶- کارایی الگوریتم ۱-۶ را با کارایی الگوریتم ۲-۶ برای نمونه مسئله‌های بزرگ مقایسه کنید.

بخش ۲-۶

۷- با استفاده از الگوریتم ۳-۶ (جستجوی اول-بهترین با الگوریتم هرس شاخه و حد برای مسئله فروشنده دوره‌گرد) یک تور بهینه و طول تور بهینه را برای گراف زیر پیدا کنید.



مراحل را قدم به قدم نشان دهید.

۸- توابع length و bound که در الگوریتم ۳-۶ استفاده شده‌اند را بنویسید.
۹- الگوریتم ۳-۶ را بر روی کامپیوتر خود پیاده‌سازی نموده،آن را بر روی نمونه مسئله تمرین ۷ اجرا کنید. با استفاده از توابع حدی مختلف، نتایج را بررسی کنید.
۱۰- کارایی الگوریتم برنامه‌نویسی پویا (به تمرین ۲۷ بخش ۶-۳ مراجعه کنید) برای مسئله فروشندهٔ دوره‌گرد را با الگوریتم ۳-۶، با استفاده از نمونه مسئله‌های بزرگ، مقایسه کنید.

بخش ۳-۶

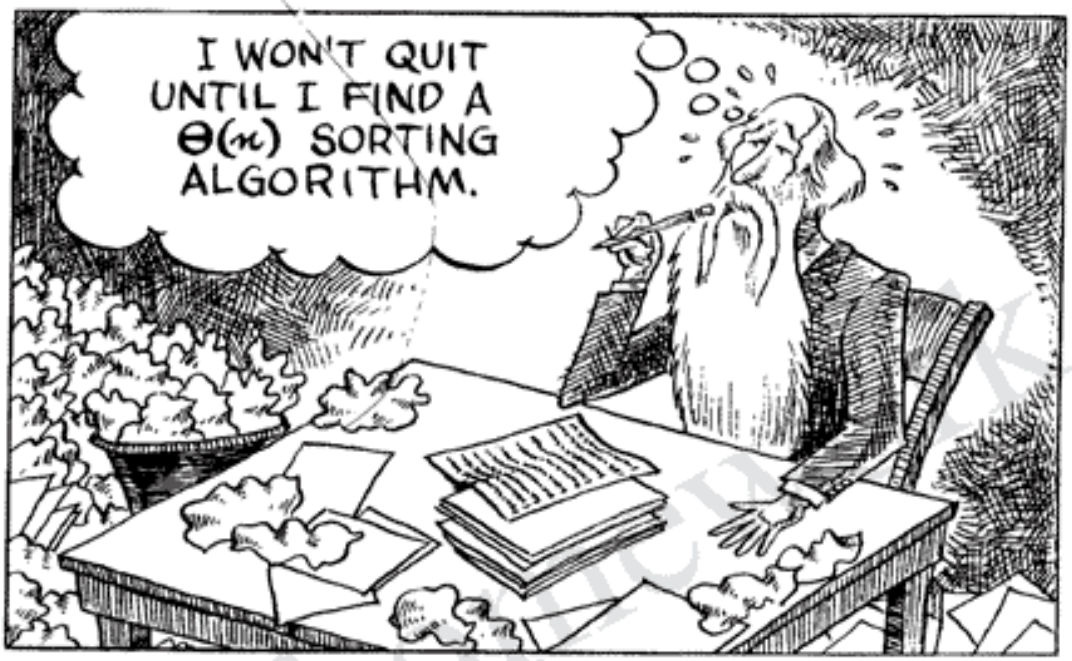
۱۱- الگوریتم ۴-۶ (جستجوی اول-بهترین کوپر با الگوریتم هرس شاخه و حد برای استنتاج ربایشی) را اصلاح کنید تا m تا از محتمل‌ترین جوابها را تولید کند. m می‌تواند هر عدد صحیح مثبت باشد.
۱۲- نشان دهید که اگر بیماریهای مثال ۴-۶ بر اساس احتمالات شرطی به صورت غیرصعودی مرتب شده بودند، تعداد گره‌های مورد بررسی از ۱۵ گره به ۲۳ گره می‌رسید. فرض کنید که $p(d_4) = ۰/۰۰۸$ و $p(d_4 , d_1) = ۰/۰۰۷$ می‌باشد.
۱۳- الگوریتم ۴-۶ را بر روی کامپیوتر خود پیاده‌سازی کنید. کاربر بایستی بتواند یک عدد صحیح m را (همانگونه که در تمرین ۱۱ بیان شد) وارد کند و یا اینکه به راحتی یک P را (همانطور که در تمرین ۱۳ گفته شد) وارد نماید.

تمرینات اضافی

۱۴- آیا می‌توان از روش شاخه و حد برای حل مسئله مطرح شده در تمرین ۳۴ فصل ۳ استفاده کرد؟ توضیح دهید.
۱۵- یک الگوریتم شاخه و حد برای زمانبندی مهلت‌دار بخش ۲-۳-۴ بنویسید.
۱۶- آیا می‌توان از روش شاخه و حد برای حل مسئله مطرح شده در تمرین ۲۶ فصل ۴ استفاده کرد؟ توضیح دهید.
۱۷- آیا می‌توان از روش شاخه و حد برای حل مسئله ضرب ماتریس زنجیره‌ای، مطرح شده در بخش ۴-۳ استفاده کرد؟ توضیح دهید.
۱۸- سه کاربرد دیگر استراتژی شاخه و حد را نام ببرید.

فصل ۷

مقدمه‌ای بر پیچیدگی محاسباتی: مسئله مرتب‌سازی



در بخش ۱-۱، یک الگوریتم مرتب‌سازی از نوع زمان مربعی (مرتب‌سازی تبادلی) ارائه نمودیم. اگرمتخصصان علم کامپیوتر به این الگوریتم مرتب‌سازی اکتفا می‌کردند، امروزه اجرای بسیاری از برنامه‌های کاربردی، بطور قابل ملاحظه‌ای کندتر و بقیه نیز غیرقابل اجرا بودند. به یاد دارید که جهت مرتب‌سازی یک میلیون کلید با استفاده از الگوریتم زمان-مربعی، به سالها وقت نیاز داشتیم (به جدول ۴-۱ رجوع کنید). بعدها الگوریتم‌های مرتب‌سازی بسیار کاراتری ارائه شدند که از آن جمله می‌توان به الگوریتم مرتب‌سازی ادغامی (Mergesort) اشاره نمود. کارایی این الگوریتم، همانگونه که در بخش ۲-۲ بررسی شد، در بدترین حالت برابر $\theta(n \lg n)$ می‌باشد. اگرچه این الگوریتم قادر به مرتب‌سازی یک میلیون عنصر در یک زمان اندک و قابل قبولی نمی‌باشد، ولی جدول ۴-۱ نشان می‌دهد که در کاربردهای off-line از توانایی بالایی برخوردار است. فرض کنید که شخصی می‌خواهد یک میلیون عنصر را در یک سیستم on-line، به طور فوری مرتب نماید. وی ممکن است ساعتها و یا حتی سالهای زیادی را صرف ارائهٔ یک الگوریتم مرتب‌سازی زمان-خطی و یا بهتر از آن نماید. نکته در اینجاست که اگر پس از صرف سالهای طولانی از عمر خویش به غیر ممکن بودن ارائه این الگوریتم پی ببرد، موضوع برای او بسیار ناراحت کننده

خواهد بود. دو روش برای مقابله با این مشکل وجود دارد. روش اول، سعی در ارائه یک الگوریتم کاراتر برای این مسئله است و روش دوم، سعی در اثبات غیرممکن بودن ارائه یک الگوریتم کاراتر برای مسئله می‌باشد. هر گاه چنین موضوعی را ثابت نمودیم از تلاش برای یافتن الگوریتمی سریعتر و کاراتر دست بر خواهیم داشت. بعدها ثابت خواهیم کرد که برای بسیاری از الگوریتم‌های مرتب‌سازی، ارائه یک الگوریتم $\theta(n \lg n)$ غیرممکن است.

۱-۷ پیچیدگی محاسباتی

فصلهای گذشته، در ارتباط با ارائه الگوریتم‌هایی برای مسائل و تحلیل آنها بوده است. ما اغلب روشهای مختلفی را برای حل یک مسئله بکار می‌بریم؛ به این امید که به کارایی بیشتر دست یابیم. هنگامی که ما الگوریتم مشخصی را تحلیل می‌کنیم، در واقع پیچیدگی زمانی (یا حافظه‌ای) و یا ترتیب پیچیدگی آن را مشخص می‌کنیم. در اینجا خود مسئله، تجزیه و تحلیل نمی‌شود. برای مثال، هنگامی که الگوریتم ۴-۱ (ضرب ماتریسی) را تحلیل نمودیم، پیچیدگی زمانی n^3 برای الگوریتم مشخص شد. این بدین معنا نیست که مسئله ضرب ماتریسها، نیاز به یک الگوریتم $\theta(n^3)$ دارد؛ بلکه تابع n^3 یک خاصیت از آن الگوریتم است و لزوماً یک خاصیت از مسئله ضرب ماتریسها نیست. در بخش ۵-۲ الگوریتم ضرب ماتریس استراسن را با پیچیدگی زمانی $\theta(n^{2/81})$ ارائه دادیم. یک سؤال مهم این است که آیا می‌توان الگوریتمی کاراتر برای این مسئله ارائه نمود؟

پیچیدگی محاسباتی که با طراحی و تحلیل الگوریتم‌ها سروکار دارد، بررسی همهٔ الگوریتم‌هایی است که می‌توانند یک مسئله معین را حل نمایند. یک تحلیل پیچیدگی محاسباتی سعی می‌کند حد پائینی را برای کارایی تمام الگوریتم‌های یک مسئله معین، مشخص نماید. در پایان بخش ۵-۲ اشاره کردیم به اینکه ثابت شده است مسئله ضرب ماتریسها به الگوریتمی با پیچیدگی زمانی $\Omega(n^3)$ نیاز دارد. در واقع، تحلیل پیچیدگی محاسباتی مشخص کنندهٔ این مطلب بوده است و ما این نتیجه را با این جمله که "**حد پائین‌تر** برای مسئله ضرب ماتریسی برابر $\Omega(n^3)$ است." بیان می‌کنیم. البته این به معنای اینکه باید یک الگوریتم $\theta(n^3)$ برای ضرب ماتریس ارائه شود نیست، بلکه بدین معناست که فقط ارائه یک الگوریتم بهتر از $\theta(n^3)$ برای ضرب ماتریسی امکانپذیر است. از آنجائیکه بهترین الگوریتم ما، در $\theta(n^{2/38})$ و حد پائین‌تر آن $\Omega(n^2)$ است، جستجوی الگوریتمی کاراتر با ارزش به نظر می‌رسد. این بررسی می‌تواند در دو جهت ادامه یابد. از طرفی می‌توانیم با استفاده از متدولوژی طراحی الگوریتم، الگوریتمی کاراتر برای مسئله پیدا کنیم و از طرف دیگر می‌توانیم با استفاده از تحلیل پیچیدگی محاسباتی، حد پائین بزرگتری را بدست آوریم. شاید روزی یک الگوریتم بهتر از $\theta(n^{2/38})$ پیدا کنیم و یا شاید روزی ثابت کنیم که یک حد پائین بزرگتر از $\Omega(n^2)$ وجود دارد. در کل، هدف ما برای یک مسئله معین، تعیین یک حد پائین‌تر از $\Omega(f(n))$ و ارائه یک الگوریتم $\theta(f(n))$ برای مسئله است. هنگامی که این کار انجام شد، می‌دانیم که بجز اصلاح مقادیر ثابت، هیچگونه اصلاح و بهینه‌سازی روی الگوریتم انجام نخواهد شد.

برخی نویسندگان عبارت "تحلیل پیچیدگی محاسباتی" را شامل تحلیل مسئله و الگوریتم می‌دانند. در این کتاب، هر گاه به تحلیل پیچیدگی محاسباتی اشاره می‌کنیم، منظور همان تحلیل مسئله است.

ما تحلیل پیچیدگی محاسباتی را با مطالعه و بررسی مسئله مرتب‌سازی معرفی می‌کنیم. دو دلیل برای انتخاب این مسئله وجود دارد. اول اینکه، تا بحال چندین الگوریتم برای حل این مسئله ارائه شده است و پرواضح است که با مطالعه و مقایسه این الگوریتم‌ها می‌توان به دیدگاه روشنی نسبت به انتخاب یک الگوریتم از بین آنها و چگونگی اصلاح و بهبود آن دست یافت و دوم آنکه، مسئله مرتب‌سازی از جمله مسائلی است که در ارائه الگوریتم‌هایی با پیچیدگی زمانی نزدیک به حد پائین آنها، موفق بوده‌ایم. بدینصورت که برای بسیاری از الگوریتم‌های مرتب‌سازی، یک حد پائین‌تر از $\Omega(n \lg n)$ تعیین نموده و الگوریتم‌هایی از درجهٔ $\theta(n \lg n)$ ارائه نموده‌ایم. بنابراین، می‌توانیم ادعا کنیم که مسئله مرتب‌سازی را تا آنجایی که به این گره از الگوریتم‌ها مربوط است، حل نموده‌ایم.

این گروه از الگوریتم‌های مرتب‌سازی که برای آنها الگوریتم‌هایی با حد پائین‌تر یافته‌ایم، شامل الگوریتم‌هایی هستند که عمل مرتب‌سازی را تنها با مقاسه کلیدها انجام می‌دهند. همانطوری که در آغاز فصل ۱ بحث شد، کلمه "کلید" به این علت استفاده می‌شود که رکوردها، اغلب شامل یک شناسهٔ منحصر به فرد، موسم به **کلید** هستند که هر کدام عضوی از یک مجموعه مرتب می‌باشند. برای رکوردهایی که با یک توالی اختیاری مرتب شده‌اند، عمل مرتب‌سازی، مجدداً آنها را بر اساس مقادیر کلیدها مرتب می‌نماید. در الگوریتم‌های ما، کلیدها در یک آرایه ذخیره می‌شوند و ما به فیلدهای غیرکلیدی مراجعه نخواهیم کرد. به هر حال فرض می‌کنیم که این فیلدها، همراه با کلید مرتب می‌شوند. الگوریتم‌هایی که عمل مرتب‌سازی را تنها با مقایسه کلیدها انجام می‌دهند، می‌توانند دو کلید را با هم مقایسه کنند تا مقدار بزرگتر مشخص شود و نیز می‌توانند کلیدها را کپی نمایند، اما هیچگونه عملیات دیگری روی کلیدها امکان‌پذیر نیست. الگوریتم‌های مرتب‌سازی که تا به حال برشمردیم (الگوریتم‌های ۳-۱، ۴-۲ و ۶-۲)، در این گروه قرار می‌گیرند.

در بخش‌های ۲-۷ تا ۸-۷، در مورد الگوریتم‌هایی که تنها با مقایسه کلیدها، عمل مرتب‌سازی را انجام می‌دهند، بحث می‌کنیم. بخش ۲-۷، به طور مشخص الگوریتم‌های مرتب‌سازی درجی (Insertion sort) و مرتب‌سازی انتخابی (Selection Sort) که دو تا از کاراترین الگوریتم‌های زمان-مربعی هستند را بررسی می‌کند. در بخش ۳-۷، نشان می‌دهیم که تا زمانی که خود را به الگوریتم‌های فوق محدود می‌کنیم، نمی‌توانیم پیچیدگی زمان-مربعی الگوریتم‌های مرتب‌سازی را بهبود ببخشیم. بخشهای ۴-۷ و ۵-۷، مروری بر الگوریتم‌های مرتب‌سازی $\theta(n \lg n)$ یعنی Mergesort و Quicksort خواهد داشت. بخش ۶-۷، الگوریتم مرتب‌سازی دیگری از نوع $\theta(n \lg n)$، موسوم به Heapsort را ارائه خواهد داد. در بخش ۷-۷، سه نوع مرتب‌سازی با پیچیدگی $\theta(n \lg n)$ را مقایسه می‌کنیم. در بخش ۸-۷، ثابت می‌کنیم که $\Omega(n \lg n)$، پائین‌ترین حد برای الگوریتم‌های مرتب‌سازی است که تنها با مقایسه کلیدها این عمل را انجام می‌دهند. در بخش ۹-۷، الگوریتم مرتب‌سازی پایه‌ای (Radix sort) را معرفی می‌کنیم که با مقایسه کلیدها عمل مرتب‌سازی را انجام نمی‌دهد.

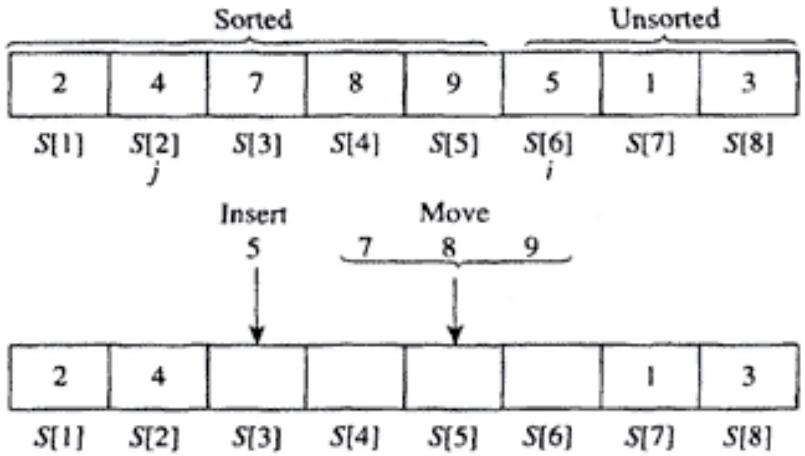
ما الگوریتم‌ها را براساس تعداد مقایسهٔ کلیدها و تعداد انتساب رکوردها تحلیل می‌کنیم. برای مثال، در الگوریتم ۳-۱ (Exchange Sort)، جابجایی s[i] و s[j] می‌تواند به طریقه زیر انجام شود:

```
temp = s[i];
s[i] = s[j];
s[i] = temp;
```

این بدین معناست که برای انجام یک جابجایی، سه مرتبه عمل انتساب رکورد صورت گرفته است. ما تعداد انتسابها را بررسی می‌کنیم زیرا وقتی که رکوردها بزرگ باشند، زمانی که جهت انتساب آنها صرف می‌شود قابل توجه است. ما همچنین مقدار فضای اضافی که الگوریتم‌ها، علاوه بر فضای مورد نیاز جهت ذخیره ورودی به آن نیازمندند را تجزیه و تحلیل می‌کنیم. هرگاه این فضای اضافی یک مقدار ثابت باشد(بدینصورت که مقدار آن با بزرگ‌تر شدن n یعنی تعداد کلیدهای ذخیره شده، افزایش نیابد)، الگوریتم را **مرتب‌سازی درجا(in-place sort)** می‌گوئیم. در نهایت، فرض می‌کنیم که همیشه عمل مرتب‌سازی را به ترتیب غیرنزولی انجام می‌دهیم.

۷-۲ مرتب‌سازی درجی و مرتب‌سازی انتخابی

مرتب‌سازی درجی، یک الگوریتم مرتب‌سازی است که با درج رکوردها در یک آرایهٔ مرتب شده، عمل مرتب‌سازی را انجام می‌دهد. یک نمونهٔ ساده از مرتب‌سازی درجی به صورت زیر عمل می‌کند. فرض کنید که کلیدها در ۱-i اندیس ابتدای آرایه مرتب شده‌اند و x مقدار کلید در اندیس iام باشد. x را با کلید اندیس ۱-iام، یا کلید اندیس ۲-iام و بهمین ترتیب مقایسه می‌کنیم تا زمانیکه یک کلید کوچکتر از x پیدا شود. فرض کنید j اندیسی باشد که کلید کوچکتر در آن قرار دارد. کلیدهای موجود در اندیسهای ۱+j تا ۱-i را به



شکل ۷-۱ یک مثال از مرتب‌سازی درجی، وقتی که ۶ = i و ۲ = j باشد. شکل بالا، آرایه را قبل از مرحلهٔ درج و شکل پائین، آرایه را بعد از مرحله درج نشان می‌دهد.

اندیسهای j+۲ تا i منتقل نموده و x را در اندیس j+۱ام درج می‌کنیم. این روال را برای i=۲ تا i=n تکرار می‌کنیم. شکل ۱-۷، این مرتب‌سازی را نشان می‌دهد. الگوریتم مرتب‌سازی درجی را در زیر آورده‌ایم.

الگوریتم ۱-۷ مرتب‌سازی درجی

مسئله: n کلید را به صورت غیرنزولی مرتب کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S با شاخصهایی از ۱ تا n.

خروجی: آرایه S شامل کلیدهایی که به صورت غیرنزولی مرتب شده‌اند.

```
void insertionsort(int n, keytype S[])
{
    index i,j;
    keytype x;
    for(i = 2 ;i <= n; i++)
       x = s[i];
       j = i - 1;
       while( j > 0 && s[j] > x)
          s[j+1] = s[j];
          j--;
       }
       s[j+1] = x;
    }
}
```

تحلیل پیچیدگی زمانی بدترین حالت تعداد مقایسهٔ کلیدها در الگوریتم ۱-۷ (مرتب‌سازی درجی)

عمل مبنایی: مقایسه s[i] با x

اندازه ورودی: n، تعداد کلیدهایی که باید مرتب شوند.

برای یک i معین، مقایسه s[j] با x زمانی انجام می‌شود که از حلقهٔ while خارج شده باشیم زیرا مقدار j برابر صفر می‌شود. با فرض عدم برقراری شرط اول در یک عبارت &&، شرط دوم مورد ارزیابی قرار نمی‌گیرد. لذا مقایسه s[i] با x، هنگامی که j=۰ است، انجام نمی‌شود. در نتیجه، در حدود i-۱ مقایسه برای هر i معین خواهیم داشت. از آنجائیکه i به مقادیر ۲ تا n محدود می‌شود، لذا مجموع تعداد مقایسه‌های انجام شده تقریباً برابراست با

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

به عنوان تمرین نشان دهید که اگر کلیدها به صورت غیرصعودی در یک آرایه قرار داشته باشند، دقیقاً این تعداد مقایسه انجام می‌پذیرد. بنابراین،

$$w(n) = \frac{n(n-۱)}{۲}$$

تحلیل پیچیدگی زمانی حالت میانی تعداد مقایسه کلیدها در الگوریتم ۱-۷ (مرتب‌سازی درجی)

برای یک i معین، i اندیس وجود دارد که x می‌تواند در آنها جایگزین شود. بدین ترتیب که x می‌تواند در اندیس ۱-iام، یا در اندیس ۲-iام و غیره قرار گیرد. از آنجائیکه هیچگونه اطلاعی از مقدار x نداشته‌ایم و همچنین دلیلی برای اولویت دادن به یک اندیس یا اندیسهایی خاص جهت جایگزینی x نداریم، لذا احتمال یکسانی را به هر یک از i اندیس ابتدای آرایه نسبت می‌دهیم. این بدین معناست که هر اندیس، احتمالی برابر ۱/i خواهد داشت. لیست زیر، تعداد مقایسات انجام شده برای درج x در هر اندیس را نشان می‌دهد:

تعداد مقایسه‌ها	اندیس
۱	i
۲	i - ۱
⋮	⋮
i - ۱	۲
i - ۱	۱

دلیل این که تعداد مقایسه‌ها، برای حالتی که x در اولین اندیس آرایه درج می‌شود، برابر i-۱ است و نه i، این است که اولین شرط در عبارت کنترلی حلقه while، هنگامی که j=۰ است، نادرست بوده و در اینصورت، شرط دوم ارزیابی نمی‌شود. برای یک i معین، میانگین تعداد مقایسه‌های مورد نیاز برای درج x برابراست با

$$۱(\frac{۱}{i}) + ۲(\frac{۱}{i}) + \ldots + (i-۱)(\frac{۱}{i}) + (i-۱)(\frac{۱}{i}) = \frac{۱}{i}\sum_{k=۱}^{i-۱} + \frac{i-۱}{i}$$

$$= \frac{(i-۱)(i)}{۲i} + \frac{i-۱}{i}$$

$$= \frac{i+۱}{۲} - \frac{۱}{i}$$

بنابراین میانگین تعداد مقایسه‌های لازم برای مرتب‌سازی آرایه برابراست با

$$\sum_{i=۲}^{n}(\frac{i+۱}{۲} - \frac{۱}{i}) = \sum_{i=۲}^{n}\frac{i+۱}{۲} - \sum_{i=۲}^{n}\frac{۱}{i} \approx \frac{(n+۴)(n-۱)}{۴} - ln\ n.$$

معادله اخیر با استفاده از نتایج مثالهای A-۱ و A-۹ در ضمیمه A و انجام برخی محاسبات جبری بدست آمده است. ما نشان داده‌ایم که

$$A(n) \approx \frac{(n+۴)(n-۱)}{۴} - - ln\ n \approx \frac{n^۲}{۴}$$

در ادامه، مورد استفادهٔ فضای اضافی را بررسی می‌کنیم.

تحلیل کاربرد فضای اضافی الگوریتم ۱-۷ (مرتب‌سازی درجی)

تنها فضای مورد استفاده که با n افزایش می‌یابد، اندازه آرایه ورودی S است. بنابراین الگوریتم، یک مرتب‌سازی درجا است و فضای اضافی در $\theta(1)$ می‌باشد.

در تمرینات از شما می‌خواهیم که نشان دهید پیچیدگی‌های *زمانی بدترین حالت و حالت میانی برای تعداد انتسابهای انجام شده توسط مرتب‌سازی درجی* برابراست با

$$W(n)=\frac{(n+۴)(n-۱)}{۲}\approx\frac{n^۲}{۲}\quad,\quad A(n)=\frac{n(n+۷)}{۴}-۱\approx\frac{n^۲}{۴}$$

در ادامه، ما مرتب‌سازی درجی را با دیگر الگوریتم زمان-مربعی معرفی شده در این کتاب، موسوم به مرتب‌سازی تبادلی (الگوریتم ۳-۱) مقایسه می‌کنیم. به خاطر آورید که پیچیدگی زمانی حالت معمول تعداد مقایسه کلیدها در مرتب‌سازی تبادلی برابراست با

$$T(n)=\frac{n(n-۱)}{۲}$$

در تمرینات از شما می‌خواهیم که نشان دهید پیچیدگی‌های *زمانی بدترین حالت حالت میانی برای تعداد انتسابهای انجام شده توسط مرتب‌سازی تبادلی* برابراست با

$$T(n)=\frac{n(n-۱)}{۲}$$

در تمرینات از شما می‌خواهیم که نشان دهید پیچیدگی‌های زمانی بدترین حالت و حالت میانی برای تعداد انتسابهای انجام شده توسط مرتب‌سازی تبادلی برابراست با:

$$W(n)=\frac{۳n(n-۱)}{۲}\quad,\quad A(n)=\frac{۳n(n-۱)}{۴}$$

پرواضح است که *مرتب‌سازی تبادلی، یک مرتب‌سازی درجا است.*

جدول ۱-۷، نتایج مربوط به مرتب‌سازی تبادلی و مرتب‌سازی درجی را به طور خلاصه نشان می‌دهد. در اینجا مشاهده می‌کنیم که مرتب‌سازی درجی، از لحاظ مقایسه کلیدها همواره به خوبی مرتب‌سازی تبادلی و در حالت میانی بهتر از آن می‌باشد. از لحاظ انتساب رکوردها، مرتب‌سازی درجی هم در حالت میانی و هم در بدترین حالت، بهتر عمل می‌کند. بدلیل اینکه هر دو الگوریتم از نوع مرتب‌سازی درجا هستند، لذا مرتب‌سازی درجی بهتر از مرتب‌سازی تبادلی است. توجه کنید که الگوریتم دیگری موسوم به مرتب‌سازی انتخابی نیز در جدول آمده است .این الگوریتم، یک مرتب‌سازی تبادلی اصلاح شده است که یکی از نواقص آن رفع شده است. حال به معرفی این الگوریتم می‌پردازیم:

جدول ۱-۷ خلاصه از تحلیل مرتب سازیهای تبادلی، درجی و انتخابی *

Algorithm	Comparisons of Keys	Assignments of Records	Extra Space Usage
Exchange Sort	$T(n) = \frac{n^2}{2}$	$W(n) = \frac{3n^2}{2}$	In-place
		$A(n) = \frac{3n^2}{4}$	
Insertion Sort	$W(n) = \frac{n^2}{2}$	$W(n) = \frac{n^2}{2}$	In-place
	$A(n) = \frac{n^2}{4}$	$A(n) = \frac{n^2}{4}$	
Selection Sort	$T(n) = \frac{n^2}{2}$	$T(n) = 3n$	In-place

*Entries are approximate.

الگوریتم ۲-۷ مرتب‌سازی انتخابی (Selection Sort)

مسئله: n کلید را به صورت غیرنزولی مرتب کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S با شاخصهایی از ۱ تا n.

خروجی: آرایه S شامل کلیدهایی که به صورت غیرنزولی مرتب شده‌اند.

```
void selectionsort (int n, keytype S[ ])
{
   index i, j, smalldest;
   for (i = 1; i <= n-1; i++){
      smallest=i;
      for (j = i+1; j <= n; j++)
         if (S[j] < S[smallest])
            smallest = j;
   exchange S[i] and S[smallest];
   }
}
```

همانطوریکه در جدول ۱-۷ مشاهده می‌شود این الگوریتم از نظر مقایسه کلیدها دارای پیچیدگی زمانی مشابه مرتب‌سازی تبادلی است، اگر چه از نظر تعداد انتساب رکوردها، به طور قابل ملاحظه‌ای با هم متفاوتند. به جای جابجایی s[i] و s[j] در هر موقعیتی که s[j] کوچکتر از s[i] است، نظیر آنچه در مرتب‌سازی تبادلی (الگوریتم ۳-۱) انجام می‌شود، مرتب‌سازی انتخابی به سادگی مسیر شاخص کوچکترین کلید جاری از کلید iام تا nام را در خود نگه می‌دارد، آنگاه پس از تعیین رکورد موردنظر، آن را با رکورد اندیس iام جابجا می‌کند. در این روش، کوچکترین کلید پس از اولین گذر از حلقهٔ **for**-i در اولین

اندیس، دومین کلید کوچکتر پس از دومین گذر از حلقهٔ **for-i** در دومین اندیس و به همین ترتیب قرار می‌گیرند. نتیجهٔ آن مشابه مرتب‌سازی تبادلی خواهد بود. به هر حال، با انجام تنها یک جابجایی در انتهای حلقهٔ **for-i**، دقیقاً تعداد n-۱ جابه جایی انجام خواهد شد. از آنجائیکه برای هر جابجائی لازم است که سه انتساب صورت بگیرد، لذا *پیچیدگی زمانی حالت معمول تعداد انتساب‌های انجام شده توسط مرتب‌سازی انتخابی* برابراست با

$$T(n) = ۳(n - ۱)$$

به خاطر دارید که حالت میانی تعداد انتسابهای لازم برای مرتب‌سازی تبادلی در حدود $۳n^{۲}/۴$ می‌باشد. بنابراین، ما در حالت میانی، الگوریتم زمان-مربعی را با یک الگوریتم زمان-خطی جایگزین نموده‌ایم. گاهی اوقات مرتب‌سازی تبادلی بهتر از مرتب‌سازی انتخابی عمل می‌کند. برای مثال، اگر رکوردها از قبل مرتب شده باشند، در مرتب‌سازی تبادلی هیچ انتسابی صورت نمی‌گیرد.

چگونه مرتب‌سازی انتخابی را با مرتب‌سازی درجی مقایسه کنیم؟ با رجوع مجدد به جدول ۱-۷ مشاهده می‌کنیم که مرتب‌سازی درجی، از لحاظ مقایسه کلیدها همواره به خوبی مرتب‌سازی انتخابی و در حالت میانی، بهتر از آن عمل می‌کند. اگرچه از نظر تعداد انتساب رکوردها، پیچیدگی زمانی مرتب‌سازی انتخابی به صورت خطی و پیچیدگی زمانی مرتب‌سازی درجی به صورت مربعی است. به خاطر دارید که برای مقادیر بزرگ n، زمان-خطی بسیار سریعتر از زمان-مربعی عمل می‌کند. لذا اگر n، مقداری بزرگ و رکوردها نیز به اندازهٔ کافی بزرگ باشند (بطوری که انتساب یک رکورد با ارزش تلقی شود)، مرتب‌سازی انتخابی بایستی بهتر عمل نماید.

هر الگوریتم مرتب‌سازی که رکوردها را به ترتیب، انتخاب و آنها را در موقعیتهای مناسبی قرار دهد، **مرتب‌سازی انتخابی (Selection Sort)** نامیده می‌شود. این بدین معناست که مرتب‌سازی تبادلی نیز یک الگوریتم مرتب‌سازی انتخابی است. در بخش ۶-۷، الگوریتم مرتب‌سازی دیگری، موسوم به مرتب سازی هرمی (heapsort)، را ارائه می‌دهیم. به هر حال، ما الگوریتم ۲-۷، را به عنوان یک الگوریتم مرتب‌سازی انتخابی می‌شناسیم. هدف از مقایسهٔ مرتب‌سازی تبادلی، درجی و انتخابی، ارائه یک مقایسهٔ تا حد امکان ساده از الگوریتم‌های مرتب‌سازی بوده است. در عمل، هیچکدام از این الگوریتم‌ها برای نمونه‌های بسیار بزرگ، عملی نخواهند بود زیرا همهٔ آنها در بدترین حالت به صورت زمان-مربعی عمل می‌کنند. در ادامه، نشان می‌دهیم که اگر خود را به این الگوریتم‌ها و نمونه‌های مشابه محدود سازیم، امکان اصلاح و بهینه سازی الگوریتم‌های زمان-مربعی را از دست خواهیم داد.

۳-۷ حد پائین برای الگوریتم‌هایی که حداکثر یک وارونگی را بعد از هر مقایسه حذف می‌کنند

مرتب‌سازی درجی بعد از هر مقایسه یا کاری انجام نمی‌دهد و یا کلید اندیس j‌ام را به اندیس j+۱ام انتقال می‌دهد. با انتقال کلید اندیس j‌ام به یک مکان بالاتر، به این حقیقت می‌رسیم که x بایستی قبل از آن کلید واقع شود. این تمام کاری است که ما انجام داده‌ایم. ما نشان می‌دهیم که تمام الگوریتم‌هایی که تنها با مقایسه کلیدها عمل مرتب‌سازی را انجام می‌دهند و به یک میزان محدودی پس از هر مقایسه عمل مرتب‌سازی مجدد را اجرا می‌کنند، حداقل به صورت زمان-مربعی می‌باشند. تمام این نتایج، با این فرض که کلیدها از هم مجزا (غیرمشابه) می‌باشند، بدست آمده است.

به طور کلی، ما با مرتب‌سازی n کلید مجزا که از هر مجموعهٔ مرتب شده‌ای می‌آیند، سروکار داریم. بدون درنظر گرفتن این مطلب، می‌توانیم فرض کنیم که اعداد صحیح ۱، ۲،... و n، کلیدهایی هستند که بایستی مرتب شوند. زیرا می‌توانیم ۱ را به عنوان کوچکترین کلید، ۲ را به عنوان دومین کلید کوچکتر و الی آخر جایگزین نمائیم. برای مثال، فرض کنید ورودی ما به صورت الفبایی [Raplh, Clyde, Dave] باشد، آنگاه می‌توانیم Clyde را با ۱، Dave را با ۲ و Ralph را با ۳ جایگزین نمائیم تا به ورودی معادل [۳، ۱، ۲] دست یابیم. هر الگوریتمی که سه عدد صحیح را تنها با مقایسه کلیدها مرتب می‌کند، سه نام را نیز با همان تعداد مقایسات مرتب خواهد کرد.

یک **جایگشت (Permutation)** از n عدد صحیح مثبت ابتدایی می‌تواند به عنوان ترتیبی از این اعداد تعبیر شود. از آنجائیکه n! جایگشت، برای n عد صحیح مثبت ابتدایی وجود دارد (به بخش A-۷ رجوع شود)، لذا به تعداد n!، ترتیبهای متفاوت برای این اعداد می‌توان پیدا نمود. برای مثال، شش جایگشت زیر تمامی ترتیب‌های ممکن سه عدد صحیح مثبت ابتدایی را نشان می‌دهند:

[۳، ۲، ۱] [۳، ۱، ۲] [۲، ۳، ۱] [۲، ۱، ۳] [۱، ۳، ۲] [۱، ۲، ۳]

این بدین معناست که برای n کلید مجزا، n! ورودی مختلف برای الگوریتم مرتب‌سازی وجود دارد. این شش جایگشت، ورودی‌های مختلف اندازه ۳ می‌باشند.

یک جایگشت را به صورت $[k_1, k_2, \ldots, k_n]$ نشان می‌دهیم که در آن k_i، عدد صحیح موقعیت i‌ام می‌باشد. به عنوان مثال، برای جایگشت [۳، ۱، ۲] داریم:

$$k_1 = ۳ \quad , k_2 = ۱, \quad k_3 = ۲$$

یک وارونگی در جایگشت عبارتست از یک زوج (k_i, k_j) بطوری که $i < j$ و $k_i > k_j$ باشد. برای مثال، جایگشت [۵، ۶، ۴، ۲، ۳] شامل وارونگی‌های (۳، ۲)، (۳، ۱)، (۲، ۱)، (۴، ۱) و (۵، ۶) می‌باشد. به عبارت بهتر، یک جایگشت شامل هیچ وارونگی نیست اگرو تنها اگر به صورت [۱، ۲،...، n] مرتب شده باشد. بدین معنا که وظیفه مرتب‌سازی n کلید مجزا، حذف همه وارونگی‌ها در یک جایگشت می‌باشد. در اینجا به تشریح نتیجه اصلی این بخش می‌پردازیم.

قضیه ۱-۷ هر الگوریتمی که n کلید مجزا را تنها با مقایسهٔ کلیدها مرتب می‌کند و حداکثر یک وارنگی را بعد از هر مقایسه حذف می‌کند، بایستی در بدترین حالت، حداقل $\frac{n(n-1)}{2}$ کلید و در حالت میانی، حداقل $\frac{n(n-1)}{4}$ کلید را با هم مقایسه کند.

اثبات: برای اثبات وضعیت بدترین حالت کافی است نشان دهیم که یک جایگشت با تعداد $\frac{n(n-1)}{2}$ وارونگی وجود دارد، زیرا هنگامی که جایگشت به عنوان ورودی است، الگوریتم بایستی آن تعداد وارونگی را حذف نموده و در اینصورت، حداقل آن تعداد مقایسه را انجام دهد. نشان دادن این مطلب که [n، n-۱،...، ۲، ۱]، یک جایگشت است، به عنوان یک تمرین خواهد بود.

برای اثبات وضعیت حالت میانی، ما جایگشت $[k_1, k_2, ..., k_n]$ را با جایگشت $[k_n, k_{n-1}, ..., k_1]$ جفت می‌کنیم. این جایگشت، **ترانهاده(transpose)** جایگشت اصلی نامیده می‌شود. برای مثال، ترانهادهٔ [۳، ۲، ۴، ۱، ۵] عبارتست از [۵، ۱، ۴، ۲، ۳]. اگر n>۱ باشد، آنگاه هر جایگشت دارای یک ترانهاده است که از خود جایگشت مجزا می‌باشد. فرض کنید r و s دو عدد صحیح بین ۱ و n باشند بطوری که s>r. در یک جایگشت معین، زوج (s، r) یا یک وارونگی در جایگشت است و یا یک وارونگی در ترانهادهٔ آن و البته در هر دوی آنها نیست. نشان دادن این نکته که به تعداد $\frac{n(n-1)}{2}$ از چنین زوجهایی از اعداد صحیح بین ۱ تا n وجود دارد، به عنوان یک تمرین خواهد بود. این بدین معناست که در یک جایگشت و ترانهاده‌اش دقیقاً $\frac{n(n-1)}{2}$ وارونگی وجود دارد. بنابراین، میانگین تعداد وارونگی‌ها در یک جایگشت و ترانهادهٔ آن برابراست با

$$\frac{1}{2} \times \frac{n(n-1)}{2} = \frac{n(n-1)}{4}$$

بنابراین، اگر ما همه جایگشت‌ها را با احتمال یکسان به عنوان ورودی درنظر بگیریم، میانگین تعداد وارونگی‌ها در ورودی نیز برابر $\frac{n(n-1)}{4}$ خواهد بود. از آنجائیکه فرض نمودیم الگوریتم حداکثر یک وارونگی را بعد از هر مقایسه حذف می‌کند، لذا در حالت میانی می‌بایست حداقل این تعداد مقایسه را انجام دهد تا همهٔ وارونگی‌ها حذف شوند و در نتیجه ورودی مرتب شود.

مرتب‌سازی درجی حداکثر وارونگی شامل s[i] و x را بعد از هر مقایسه حذف می‌کند و در نتیجه، این الگوریتم در زمرهٔ الگوریتم‌های مطرح شده در قضیه ۱-۷ قرار می‌گیرد. مرتب‌سازی انتخابی و مرتب‌سازی تبادلی نیز در این گروه قرار دارند. برای مشاهدهٔ این حالت، یک مثال از مرتب‌سازی تبادلی ارائه می‌دهیم. ابتدا به خاطر می‌آوریم که الگوریتم مرتب‌سازی انتخابی به صورت زیر می‌باشد:

```
void exchangesort(int n, keytype s[ ])
{
  index i,j;
  for (i = 1; i <= n - 1; i++)
    for (j = i+1; j <= n; j++)
      if (s[j] < s[i])
        exchange s[i] and s[j];
}
```

فرض کنید آرایه S شامل جایگشت [۱، ۳، ۲، ۴]، و ما در حال مقایسه ۲ با ۱ باشیم. بعد از این مقایسه ۲ و ۱ جابجا خواهند شد و در نتیجه وارونگی‌های (۲، ۱)، (۴، ۱) و (۳، ۱) حذف می‌شوند؛ در حالیکه وارونگی‌های (۴، ۲) و (۳، ۲) به آن اضافه خواهند شد و در واقع کاهش شبکه‌ای در وارونگی، تنها یک مورد است. یک نتیجه کلی از این مثال اینست که مرتب‌سازی تبادلی همواره دارای یک کاهش شبکه‌ای از حذف حداکثر یک وارونگی بعد از هر مقایسه است.

از آنجائیکه پیچیدگی زمانی مرتب‌سازی درجی در بدترین حالت برابر $\frac{n(n-1)}{2}$ و در حالت میانی تقریباً برابر $\frac{n(n-1)}{4}$ است، لذا آن تقریباً به خوبی همان الگوریتم‌هایی است که تنها با مقایسه کلیدها عمل مرتب‌سازی را انجام داده و بعد از هر مقایسه حداکثر یک وارونگی را حذف می‌کنند. بخاطر دارید که Mergesort (الگوریتم‌های ۲-۲ و ۴-۲) و Quicksort (الگوریتم ۶-۲) دارای پیچیدگی زمانی بهتر از این می‌باشند. بیائید مروری مختصر بر این الگوریتم‌ها داشته باشیم.

۷-۴ مروری بر Mergesort

الگوریتم مرتب‌سازی ادغامی (Mergesort) در بخش ۲-۲ معرفی شد. در اینجا نشان می‌دهیم که این الگوریتم، گاهی اوقات بیش از یک وارونگی را بعد از یک مقایسه حذف می‌کند. آنگاه چگونگی اصلاح آن را نشان خواهیم داد.

همانطوریکه در اثبات قضیه ۱-۷ مشاهده نمودید، هرگاه ورودی به صورت معکوس مرتب شده باشد، الگوریتم‌هایی که حداکثر یک وارونگی را بعد از هر مقایسه حذف می‌کنند، حداقل $\frac{n(n-1)}{2}$ مقایسه را انجام می‌دهند. شکل ۲-۷ نشان می‌دهد که چگونه مرتب‌سازی ادغامی ۲ (الگوریتم ۴-۲) روی چنین ورودی عمل می‌کند. هنگامی که زیر آرایه‌های [۲ ۱] و [۴ ۳] با هم ادغام می‌شوند، مقایسه‌ها بیش از یک وارونگی را حذف می‌کنند. بعد از اینکه ۳ و ۱ با هم مقایسه شدند، ۱ در اولین اندیس آرایه قرار می‌گیرد و در نتیجه وارونگی‌های (۳، ۱) و (۴، ۱) حذف می‌شوند. بعد از هر مقایسهٔ ۲ و ۳، عدد ۲ در دومین اندیس آرایه قرار می‌گیرد و در نتیجه وارونگی‌های (۳، ۲) و (۴، ۲) حذف می‌شوند.

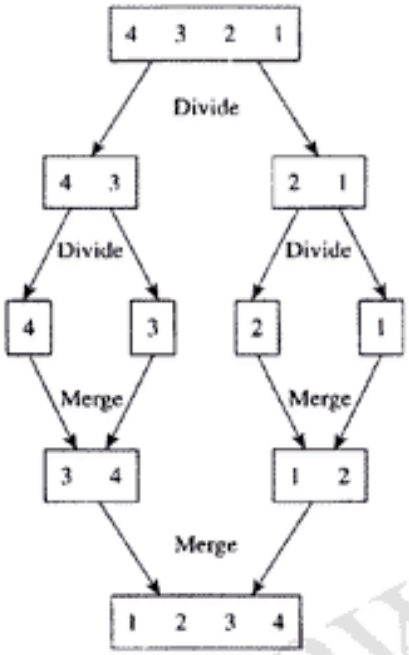
به خاطر دارید که پیچیدگی زمانی بدترین حالت برای تعداد مقایسه کلیدها در مرتب‌سازی ادغامی برابراست با

$$w(n) = n \lg n - (n - 1)$$

که در آن n توانی از ۲، و در حالت کلی، در $\theta(n \lg n)$ است.

ما با ارائه یک الگوریتم مرتب‌سازی، که گاهی اوقات بیش از یک وارونگی را بعد از یک مقایسه حذف می‌کند، توانستیم به این نتیجه دست یابیم. از بخش ۱-۴-۱، به یاد دارید که الگوریتم‌های $\theta(n \lg n)$ می‌توانند برخلاف الگوریتم‌های زمان-مربعی، بر روی ورودیهای بسیار بزرگ نیز عمل نمایند. با استفاده از روش «توابع مولّد» برای حل معادلات بازگشتی، می‌توانیم نشان دهیم که پیچیدگی زمانی حالت میانی برای تعداد مقایسات کلیدها در مرتب‌سازی ادغامی برابراست با

شکل ۲-۷ مرتب‌سازی ادغامی در حالتی که ورودی آن به صورت معکوس مرتب شده است.



$$A(n) = n \lg n - ۲n \sum_{i=۱}^{\lg n} \frac{۱}{۲^i + ۲} \approx n \lg n - ۱/۲۶n$$

که در آن n توانی از ۲ است.

شما می‌توانید این روش را در کتاب sahni (۱۹۸۸) مطالعه نمائید. حالت میانی، خیلی بهتر از بدترین حالت نیست. در تمرینات نشان خواهیم داد که پیچیدگی زمانی *حالت معمول برای تعداد مقایسه کلیدها* در*مرتب‌سازی ادغامی* برابراست با

$$T(n) \approx ۲n \lg n$$

در ادامه، فضای مورد استفاده برای مرتب‌سازی ادغامی را مورد بررسی قرار می‌دهیم.

تحلیل فضای اضافی مورد استفاده برای الگوریتم ۴-۲ (مرتب‌سازی ادغاامی ۲)

همانطور که در بخش ۲-۲ بحث شد، حتی نسخهٔ اصلاح شدهٔ مرتب‌سازی ادغامی (الگوریتم ۴-۲) نیز به یک آرایهٔ اضافی به اندازه n نیاز دارد. به عبارت بهتر، هنگامی که الگوریتم در حال مرتب‌سازی اولین زیرآرایه است، باید مقادیر mid، mid+۱ و high در پشتهٔ رکوردهای فعال سازی ذخیره شوند و از آنجائیکه ارایه همواره در نقطه میانی شکافته می‌شود، لذا این پشته به عمق $\lceil \lg n \rceil$ خواهد رسید. فضای لازم برای آرایه‌ای اضافی از رکوردها محدود می‌شود؛ یعنی اینکه در حالت معمول، بکارگیری فضای اضافی در $\theta(n)$ رکورد صورت می‌گیرد و یا اینکه تعداد رکوردها در $\theta(n)$ خواهد بود.

اصلاح مرتب‌سازی ادغامی

ما می‌توانیم الگوریتم مرتب‌سازی ادغامی را به سه روش اصلاح نمائیم. روش اول، ارائهٔ یک نسخهٔ برنامه‌نویسی پویا، روش دوم نوشتن یک نسخهٔ پیوندی و آخرین روش، یک الگوریتم ادغام بسیار پیچیده است.

به منظور ارائهٔ یک نسخهٔ برنامه‌نویسی پویا از مرتب‌سازی ادغامی به شکل ۲-۲ توجه نمائید. اگر بخواهید مرتب‌سازی ادغامی را به صورت دستی انجام دهید نیازی به تقسیم آرایه تا رسیدن به آرایه‌های تک عنصری نخواهید داشت. به سادگی می‌توانید از آرایه‌های تک عنصری شروع نموده، آنها را به گروههای دوتایی، بعد به گروههای چهارتایی و الی آخر ادغام نمائید تا اینکه به یک آرایهٔ مرتب شده دست یابید. ما نیز می‌توانیم با تقلید از این روش، یک نسخهٔ تکرار برای مرتب‌سازی ادغامی بنویسیم که در اینصورت از سه عمل پشته‌ای مورد نیاز در بازگشت‌ها اجتناب خواهیم کرد. حتماً توجه دارید که این، یک روش برنامه نویسی پویا برای مرتب‌سازی ادغامی است. الگوریتم زیر این روش را بکار می‌گیرد. حلقه در آرایه، اندازهٔ آرایه را به عنوان توانی از ۲ درنظر می‌گیرد. مقادیر n که توانی از ۲ نیستند، $2^{\lceil lg\, n \rceil}$ مرتبه از حلقه می‌گذرد؛ بدون اینکه ادغامی را صورت دهند.

الگوریتم ۳-۷ مرتب‌سازی ادغامی۳ (نسخهٔ برنامه‌نویسی پویا)

مسئله: n کلید را به صورت غیرنزولی مرتب کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S با شاخصهایی از ۱ تا n

خروجی: آرایه S شامل کلیدهایی که به صورت غیرنزولی مرتب شده‌اند.

```
void mergesort3 (int n, keytype S[ ])
{
  int m;
  index low,mid,high,size;                    // Treat array size as a
  m=2^[ lg  n];                               // power of 2.
  size = 1;                                   // size is the size of the
  repeat (lgm times){                         // Subarrays bing ,merged.
   for(low=l; low<=m-2*size-1;low=low+2*size){
     mid=low + size - 1 ;
     high=minimum (low + 2*size - 1,n);       // Dont merge beyond n.
     merge3(low,mid,high,S);
   }
  size=2*size;
 }
}
```

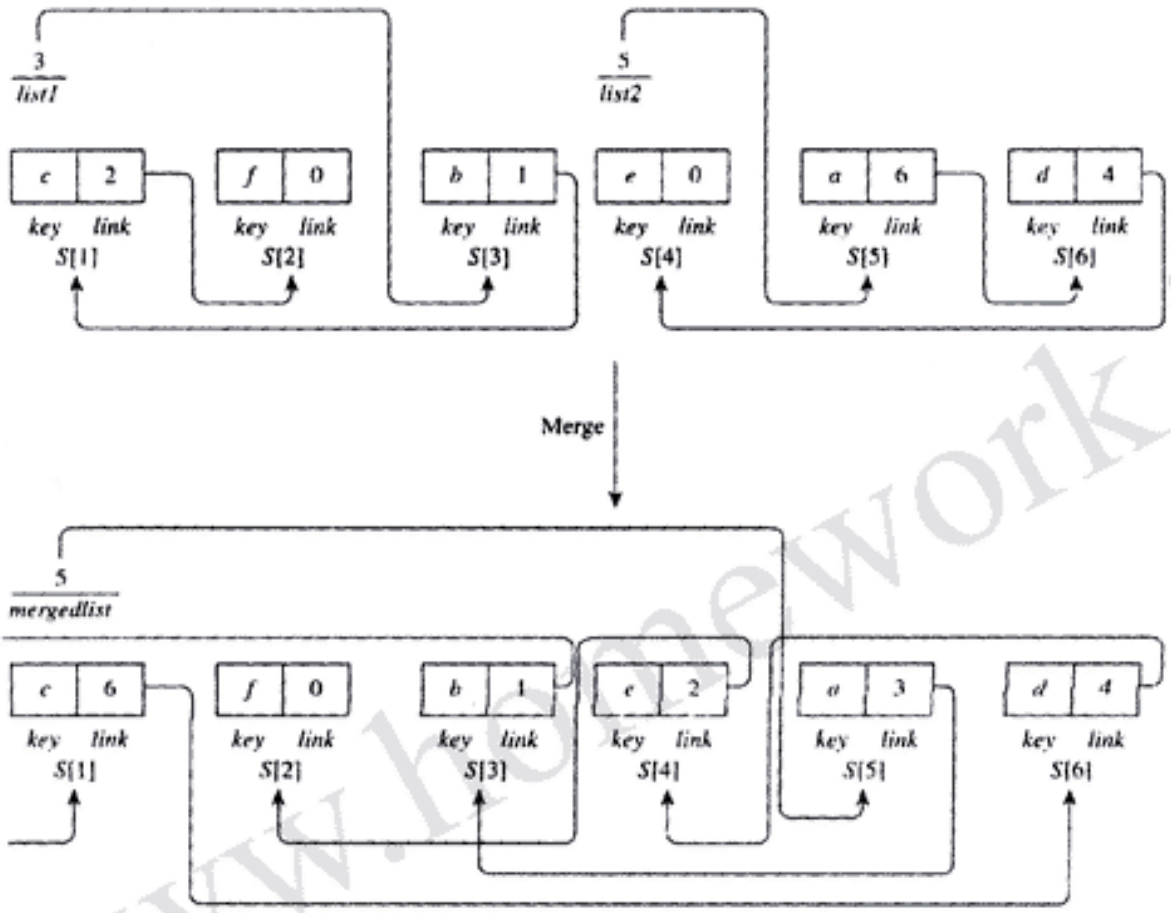
با این اصلاح می‌توانیم تعداد انتساب رکوردها را نیز کاهش دهیم. آرایه U، که در روال merge۲ (الگوریتم ۵-۲) به صورت محلی تعریف شده است، می‌تواند در روال merge۳ نیز به عنوان یک آرایه با شاخصهایی از ۱ تا n به صورت محلی تعریف شود. بعد از اولین گذر از حلقه repeat، U شامل عناصر آرایهٔ S که به صورت جفتهایی از آرایه‌های تک عنصری ادغام شده‌اند، می‌باشد. هیچ نیازی به کپی مجدد این عناصر به S، آنچنانکه در انتهای merge۲ انجام شده، وجود ندارد. بجای این، در گذر بعدی از حلقهٔ repeat، می‌توانیم به سادگی عناصر U را با S ادغام کنیم. اینجاست که نقش آرایه‌های S و U تعویض می‌شود. این جابجایی نقش‌ها می‌تواند در هر گذر از حلقه صورت پذیرد. نسخه‌های mergesort۲ و mergesort۳ را به عنوان تمرین باقی می‌گذارد. در این روش، تعداد انتساب رکوردها از $2n\,lgn$ به حدود $n\,lgn$ کاهش می‌یابد. در اینصورت می‌گوئیم پیچیدگی زمانی *حالت معمول برای تعداد انتساب* رکوردها در الگوریتم ۳-۷ تقریباً برابر است با

$$T(n) \approx n\,lgn$$

دومین روش اصلاح مرتب‌سازی ادغامی، ارائهٔ یک نسخهٔ پیوندی از الگوریتم است. همانطوریکه در بخش ۱-۷ بحث شد، مسئله مرتب‌سازی معمولاً با مرتب‌سازی رکوردها براساس مقادیر کلیدها سروکار دارد. اگر رکوردها بزرگ باشند، مقدار فضای اضافی مورد استفاده توسط مرتب‌سازی ادغامی بایستی درنظر گرفته شود. ما می‌توانیم با افزودن یک فیلد پیوند به هر رکورد، مقدار فضای اضافی را کاهش دهیم. آنگاه با تبدیل رکوردها به لیست پیوندی و الحاق پیوندها به جای انتقال رکوردها به یک مجموعهٔ مرتب شده از رکوردها دست یابیم و این بدین معناست که دیگر نیازی به ایجاد آرایهٔ اضافی از رکوردها نخواهد بود. از آنجائیکه فضای اشغال شده توسط یک پیوند بطور قابل ملاحظه‌ای کمتر از یک رکورد بزرگ است، بنابراین، فضای ذخیره شده قابل توجه خواهد بود. بعلاوه، با توجه به اینکه مدت زمان الحاق پیوندها، بسیار کمتر از انتقال رکوردهای بزرگ است، لذا زمان ذخیره‌سازی نیز از اهمیت خاصی برخوردار خواهد بود. شکل ۳-۷، نشان می‌دهد که چگونه با استفاده از پیوندها، عمل ادغام انجام می‌شود. الگوریتم ۴-۷، شامل این تغییرات خواهد بود. ما مرتب‌سازی ادغامی و ادغام را به صورت یک الگوریتم ارائه می‌دهیم زیرا نیازی به تجزیه و تحلیل بیشتر آنها نداریم. مرتب‌سازی ادغامی، به خاطر خوانایی بیشتر به صورت بازگشتی نوشته شده است. البته نسخهٔ اصلاح شده تکرار این الگوریتم را نیز می‌توان در راستای این اصلاح و بهینه سازی بکار گرفت. اگر ما از هر دو روش تکرار و پیوندها استفاده کنیم، این اصلاح، بهبود تعویض نقش U و S را دربرنمی گیرد زیرا وقتی لیستهای پیوندی با هم ادغام می‌شوند هیچ آرایهٔ اضافی مورد نیاز نمی‌باشد. نوع داده‌ای برای عناصر آرایه S در این الگوریتم به صورت زیر می‌باشد:

```
struct node
{
  keytype key;
  index link;
};
```

شکل ۳-۷ عمل ادغام با استفاده از پیوندها. فلشها چگونگی انجام پیوند را نشان می‌دهند. کلیدها به صورت حرفی هستند تا از سردرگمی و تداخل با شاخص‌ها جلوگیری شود.



الگوریتم ۴-۷ مرتب‌سازی ادغامی۴ (نسخه پیوندی)

مسئله: n کلید را به صورت غیرنزولی مرتب کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از رکوردها S از نوع داده شده با شاخصهای ۱ تا n.

خروجی: آرایه s با مقادیر ذخیره شده در فیلد key که به صورت غیرنزولی مرتب شده‌اند. رکوردها به همراه link به صورت یک لیست پیوندی مرتب می‌باشند.

```
void mergesort4 (index low, index high, index& mergedlist)
{
```

```
    index mid, list1, list2;

    if (low == high) {
        mergedlist = low;
        S[mergedlist].link = 0;
    }
    else {
        mid = ⌊(low + high) / 2⌋;
        mergesort4(low, mid, list1);
        mergesort4(mid + 1, high, list2);
        merge4(list1, list2, mergedlist);
    }
}

void merge4 (index list1, index list2, index& mergedlist)
{
    index lastsorted;

    if (S[list1].key < S[list2].key) {          // Find the start of the merged
        mergedlist = list1;                     // list.
        list1 = S[list1].link;
    }
    else {
        mergedlist = list2;
        list2 = S[list2].link;
    }
    lastsorted = mergedlist;
    while (list1 != 0 && list2 != 0)
        if S[list1].key < S[list2].key {        // Attach smaller key to merged
            S[lastsorted].link = list1;         // list.
            lastsorted = list1;
            list1 = S[list1].link;
        }
        else {
            S[lastsorted].link = list2;
            lastsorted = list2;
            list2 = S[list2].link;
        }


    if (list1 == 0)                             // After one list ends, attach
        S[lastsorted].link = list2;             // remainder of the other.
    else
        S[lastsorted].link = list1;
}
```

در اینجا نیازی به بررسی اینکه آیا list۱ و list۲ به هنگام ورود به mergesort۴، برابر صفر هستند یا خیر نمی‌باشد زیرا mergesort۴ هرگز یک لیست خالی را به merge۴ ارسال نمی‌کند. همانطوریکه برای الگوریتم ۴-۲ (مرتب‌سازی ادغامی۲) عنوان نمودیم مقادیر n و S ورودیهای mergesort۴ نیستند. فراخوانی سطح بالای این الگوریتم به صورت زیر است:

```
mergesort4(1, n, listfront);
```

بعد از اجرا، listfornt شامل شاخص‌های اولین رکورد در لیست مرتب خواهد بود.

بعد از مرتب‌سازی، اغلب می‌خواهیم که رکوردها به صورت یک دنبالهٔ مرتب در اندیسهایی به هم پیوسته قرار بگیرند بطوری که بتوانیم با استفاده از فیلد کلید و با بکارگیری جستجوی دودویی (الگوریتم ۱-۲) بسرعت به آنها دسترسی داشته باشیم. البته ممکن است رکوردهایی که براساس پیوندها مرتب شده‌اند تنها یک بار مجدداً مرتب‌سازی شوند تا همگی با استفاده از یک الگوریتم $\theta(n)$ و درجا به صورت یک دنبالهٔ مرتب در اندیسهایی به هم پیوسته قرار بگیرند. در تمرینات از شما می‌خواهیم که چنین الگوریتمی را بنویسید. با اصلاح الگوریتم مرتب‌سازی ادغامی، ذکر دو نکته ضروری به نظر می‌رسد. اول اینکه، نیاز به n رکورد اضافی با نیاز به تنها n پیوند جایگزین می‌شود. از اینرو داریم:

تحلیل کاربرد فضای اضافی در الگوریتم ۴-۷ (مرتب‌سازی ادغامی ۴)

در حالت معمول، فضای اضافی در $\theta(n)$ پیوند استفاده می‌گردد و یا به عبارت بهتر، تعداد پیوندها برای بکارگیری فضای اضافی در $\theta(n)$ خواهد بود.

دوم آنکه، پیچیدگی زمانی تعداد انتساب رکوردها به صفر کاهش می‌یابد اگر ما به رکوردهای مرتب شده در اندیسهایی به هم پیوسته نیازی نداشته باشیم، و این پیچیدگی زمانی به $\theta(n)$ کاهش می‌یابد اگر این نیاز وجود داشته باشد.

سومین روش اصلاح مرتب‌سازی ادغامی، یک الگوریتم ادغام بسیار پیچیده است که در کتاب Huang و Langston (۱۹۸۸) معرفی شده است.

۷-۵ مروری بر Quicksort

به یاد دارید که الگوریتم Quicksort به صورت زیر می‌باشد:

```
void quicksort(index low, index high)
{
    index pivotpoint;
    if( low > high)
      partition(low, high, pivotpoint);
      partition(low, high, pivotpoint);
      partition(low, high, pivotpoint);
    }
}
```

اگرچه پیچیدگی زمانی بدترین حالت این الگوریتم به صورت مربعی است، در بخش ۴-۲ مشاهده کردیم که پیچیدگی زمانی حالت میانی آن برای تعداد مقایسه کلیدها برابراست با

$$A(n) \approx 1/38(n+1)\lg n$$

که البته خیلی بدتر از مرتب‌سازی ادغامی نیست. Quicksort نسبت به Mergesort این مزیت را داراست که به آرایهٔ اضافی نیازمند نمی‌باشد. به هر حال این الگوریتم، هنوز یک مرتب‌سازی درجا نیست زیرا هنگامی که الگوریتم در حال مرتب‌سازی اولین زیرآرایه است، اولین و آخرین شاخص زیرآرایه‌های دیگر بایستی در پشتهٔ رکوردهای فعال‌سازی ذخیره شوند. برخلاف مرتب‌سازی ادغامی، ما هیچ تعهدی مبنی بر اینکه آرایه، همواره از نقطهٔ میانی تقسیم شود نداریم. در بدترین حالت، partition ممکن است مکرراً آرایه را به یک زیرآرایهٔ خالی در سمت چپ (یا راست) و یک زیرآرایه با یک عنصر کمتر در سمت چپ (یا راست) تقسیم نماید. در این روش، n-۱ جفت از شاخصها در پایان کار در پشته ذخیره خواهند شد و این بدین معنا است که بدترین حالت فضای اضافی مورد استفاده در $\theta(n)$ خواهد بود. ممکن است Quicksort به صورتی تغییر کند که فضای اضافی مورد استفاده حداکثر در حدود $\lg n$ باشد. قبل از مشاهدهٔ این حالت و دیگر مسائل مربوط به Quicksort، از پیچیدگی زمانی تعداد انتساب رکوردها در این الگوریتم بحث می‌کنیم.

در تمرینات از شما می‌خواهیم که نشان دهید میانگین تعداد تبادلات انجام شده توسط Quicksort در حدود $0/69(n+1)\lg n$ است. با فرض اینکه برای هر تبادل(exchange) به سه انتساب نیازمند می‌باشیم، پیچیدگی زمانی حالت میانی برای تعداد انتساب رکوردها در الگوریتم *Quicdsort* برابراست با

$$A(n) \approx 2/07(n+1)\lg n$$

اصلاح الگوریتم اصلی Quicksort

ما می‌توانیم فضای اضافی مورد استفاده توسط الگوریتم Quicksort را به پنج روش کاهش دهیم. اول اینکه در روال Quicksort تعیین کنیم کدام زیرآرایه کوچکتر است و در حالیکه زیرآرایهٔ دیگر در حال مرتب شدن است، این زیرآرایه را در پشته ذخیره نمائیم. در زیر، تحلیلی از فضای اضافی مورد استفاده توسط این نسخه از Quicksort را آورده‌ایم.

تحلیل فضای اضافی مورد استفاده برای Quicksort اصلاح شده

در این نسخه، بدترین حالت فضای مورد استفاده زمانی رخ می‌دهد که partition هر بار آرایه را دقیقاً نصف نماید که در نتیجه به پشته‌ای با عمق تقریباً $\lg n$ خواهیم رسید. بنابراین، بدترین حالت فضای اضافی مورد استفاده، در شاخص‌های $\theta(\lg n)$ خواهد بود.

دومین روش، همانطوریکه در تمرینات بحث شد، یک نسخه از partition است که میانگین تعداد

انتساب رکوردها را به طور قابل ملاحظه‌ای کاهش می‌دهد. برای این نسخه، پیچیدگی زمانی حالت میانی تعداد انتساب رکوردها در *Quicksort* برابراست با

$$A(n) \approx ۰/۶۹(n+۱)lgn$$

هر یک از فراخوانی‌های بازگشتی در روال Quicksort سبب می‌شود مقادیر low، high و pivotpoint در پشته ذخیره شوند. در روش سوم، این بحث مطرح است که بسیاری از اعمال push و pop در این نسخه غیرضروری است. هنگامی که اولین فراخوانی Quicksort انجام می‌شود، تنها مقادیر pivotpoint و high بایستی در پشته ذخیره شوند. وقتی دومین فراخوانی بازگشتی Quicksort انجام می‌شود، نیازی به ذخیره‌سازی هیچ یک از این مقادیر نیست. ما می‌توانیم با نوشتن الگوریتم Quicksort به صورت تکرار و دستکاری پشته در این روال، از عملیات غیرضروری اجتناب کنیم. در تمرینات، این الگوریتم را از شما می‌خواهیم.

چهارم، همانطوری که در بخش ۷-۲ بحث شد، الگوریتم‌های بازگشتی نظیر Quicksort می‌توانند با تعیین یک مقدار آستانه که در آن الگوریتم، یک الگوریتم تکرار را به جای تقسیم بیشتر نمونه فراخوانی می‌کند، اصلاح شوند.

در نهایت، همانطوری که در تحلیل بدترین حالت الگوریتم ۶-۲(Quicksort) مشاهده شد، هنگامی که ورودی از قبل مرتب شده باشد، الگوریتم دارای حداقل کارایی است. بنابراین، اگر به دلیلی مطمئن شویم که آرایه ممکن است از قبل مرتب شده باشد، می‌توانیم با انتخاب اولین عنصر به عنوان عنصر محوری، کارایی آن را بهبود بخشیم. یک استراتژی خوب برای استفاده در این حالت، انتخاب میانه در میان $S[low]$، $S[\lfloor (low+high)/۲ \rfloor]$ و $S[high]$ به عنوان نقطه محوری است. البته اگر ساختار بخصوصی برای آرایهٔ ورودی مد نظر نباشد، انتخاب هر عنصر به عنوان عنصر محوری، در میانگین به خوبی انتخاب هر عنصر دیگر خواهد بود. در این حالت، با انتخاب میانه تضمین می‌کنیم که هیچ یک از زیرآرایه‌ها خالی نباشد.

نکته آخر در مورد Quicksort این است که این الگوریتم از جمله الگوریتم‌هایی است که با انتقال و جابجایی، مرتب‌سازی می‌کنند. به این معنا که مرتب‌سازی با تبادل (انتقال) رکوردهای مجاور صورت می‌پذیرد. یک الگوریتم زمان-مربعی در این گروه الگوریتم‌ها، Bubblesort است که در تمرینات بررسی خواهد شد.

۶-۷ مرتب‌سازی هرمی (Heapsort)

heapsort، برخلاف Mergesort و Quicksort، یک الگوریتم $\theta(n \lg n)$ و درجا می‌باشد. ابتدا مروری بر heap و روالهای مورد نیاز برای heapsort خواهیم داشت. سپس چگونگی بکارگیری این روالها را بررسی خواهیم کرد.

۷-۶-۱ Heap و روالهای اصلی آن

به خاطر دارید که عمق یک گره در یک درخت، تعداد لبه‌ها در یک مسیر منحصربفرد از ریشه به آن گره می‌باشد. عمق d از یک درخت، حداکثر عمق همهٔ گره‌های درخت است و یک برگ، یک گره در درخت است که هیچ فرزندی نداشته باشد(به بخش ۵-۳ مراجعه کنید). یک **گرهٔ داخلی** در یک درخت، گره‌ای است که حداقل یک فرزند دارد. به عبارت دیگر، هر گره‌ای که یک برگ نباشد، یک گره داخلی است.
یک **درخت دودویی کامل**، درخت دودویی است که شرایط زیر را دارا باشد:

- همه گره‌های داخلی آن دو فرزند داشته باشند.
- همه برگها در عمق d باشند.

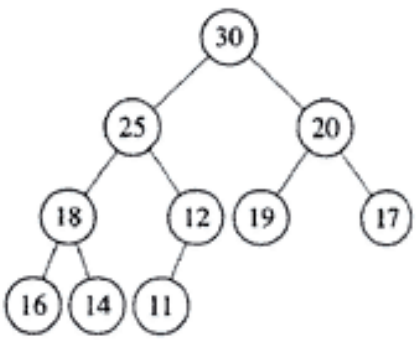
یک **درخت دودویی کامل اصلی**، درخت دودویی است که شرایط زیر را دارا باشد:

- یک درخت دودویی کامل تا عمق d-۱ باشد.
- گره‌های عمق d، حداالامکان در سمت چپ درخت باشند.

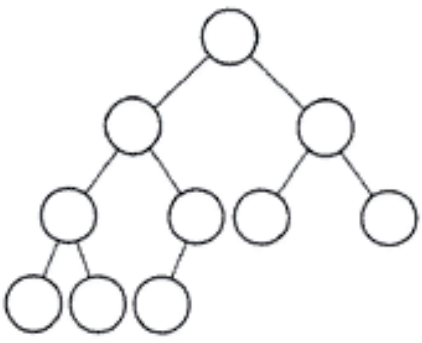
اگرچه تعریف درختهای دودویی کامل اصلی کمی مشکل است ولی از روی شکل، به آسانی می‌توان به خصوصیات آن پی برد. شکل ۷-۴، یک درخت دودویی کامل اصلی را نشان میدهد.
حال می‌توانیم یک heap را تعریف کنیم. یک heap، یک درخت دودویی کامل اصلی است بطوری که

- مقادیر ذخیره شده در گره‌ها از یک مجموعه مرتب آمده باشند.
- مقدار ذخیره شده در هر گره بزرگتر یا مساوی مقادیر ذخیره شده در فرزندانش باشد. به این مورد، **ویژگی heap** گوئیم.

شکل ۷-۵، یک heap را نشان می‌دهد. از آنجائیکه ما در بحث مرتب‌سازی قرار داریم، لذا به عناصر ذخیره شده در heap، به عنوان کلید اشاره می‌کنیم.
اگر ما کلید ذخیره شده در ریشه را مکرراً برای حفظ ویژگی heap برداریم، آنگاه کلیدها در یک توالی غیرافزایشی برداشته خواهند شد. اگر در حال برداشتن کلیدها بتوانیم آنها را در یک آرایه که با اندیس n۱م



شکل ۷-۴ یک درخت دودویی کامل اصلی.



شکل ۷-۵ یک heap.

آغاز شده و به اولین اندیس ختم می‌شود جایگزین کنیم، توانسته‌ایم آنها را در یک توالی غیرنزولی در آرایه مرتب نمائیم. بعد از برداشتن کلید ریشه، با جایگزینی گره زیرین و فراخوانی روال siftdown که کلید موجود در ریشه تا پائین را به منظور رسیدن به یک heap جابجا می‌کند، می‌توانیم ویژگی heap را حفظ کنیم. این جابجایی، با مقایسه اولیه کلید موجود در ریشه با کلید بزرگتر در فرزندان ریشه آغاز می‌شود. اگر کلید ریشه کوچکتر بود، آنگاه کلیدها جابجا می‌شوند. این فرآیند رو به پائین ادامه می‌یابد تا اینکه کلید موجود در یک گره، کوچکتر از کلید بزرگتر در فرزندانش نباشد. شکل ۶-۷، این روال را نشان می‌دهد. شبه‌کد سطح بالا برای این روال به صورت زیر است:

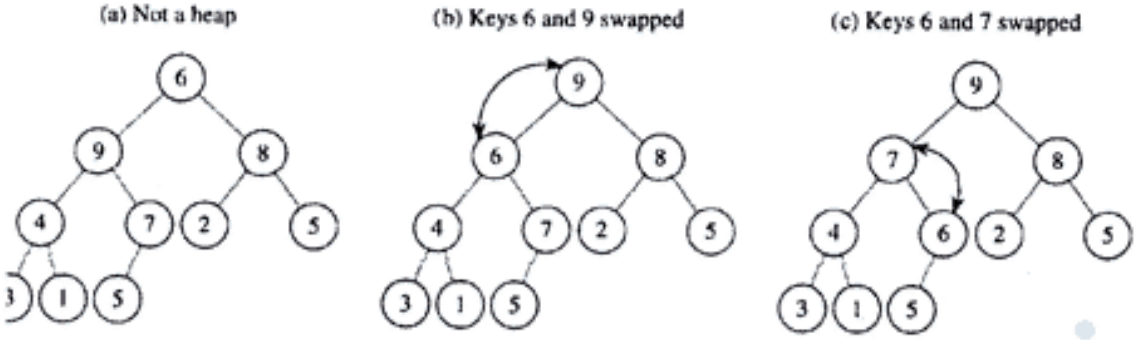
```
void siftdown(heap& H)
{
    node parent, largerchild;
    parent = root of H;
    largerchild = parent's child containing larger key;
    while (key at parent is smaller than key at largerchild){
       exchange key  at parent and key at largerchild;
       parent = largerchild;
       largerchild = parent's child containing larger key;
    }
}
```

شبه‌کد سطح بالا برای یک تابع که کلید موجود در ریشه را جابجا کند و خاصیت بازگشتی را حفظ نماید، به صورت زیر است:

```
keytype root(heap& H)
{
    keytype keyout;
    keyout = key at the root;
    move the key at the bottom mode to the root;
    delete the bottom node;
    siftrdown(H):
    return keyout;
}
```

برای یک heap معین با n کلید، شبه‌کد سطح بالای زیر برای روالی است که کلیدهای یک دنبالهٔ مرتب شده را در یک آرایه S جایگزین می‌کند.

شکل ۶-۷ روال siftdown کلید ۶ را جابجا می‌کند تا ویژگی heap حفظ شود.




```
void removekeys (int n, heap H, keytype S[ ])
{
  index i;
  for (i = n; i < 1; i--)
    s[i] = root(H);
}
```

تنها وظیفه‌ای که باقی مانده است، مرتب‌سازی کلیدها در یک heap در اولین مکان است. فرض کنید که آنها در یک درخت دودویی کامل اصلی به صورتی مرتب شده‌اند که لزوماً خاصیت heap را حفظ نمی‌کنند. چگونگی این حالت را در زیربخش بعدی خواهیم دید. ما می‌توانیم با فراخوانی مکرر siftdown، درخت را به یک heap انتقال دهیم تا عملیات زیر شکل گیرد: اول، کلیهٔ زیر درختهایی که ریشهٔ آنها عمق d-۱ دارند به heapها انتقال یابند. دوم، کلیهٔ زیر درختهایی که ریشه آنها عمق d-۲ دارند به heapها انتقال یابند.... سرانجام، درخت کامل (تنها زیر درختی که عمق صفر دارد) به یک heap انتقال یابد.

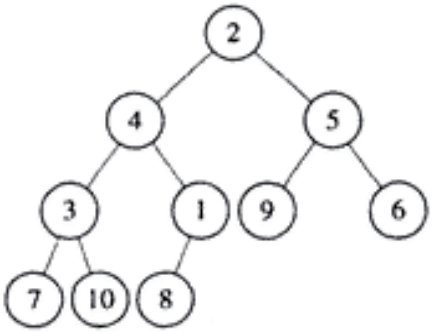
این فرآیند را در شکل ۷-۷ نشان داده و آن را با روال خارجی شبه‌کد سطح بالای زیر بکار گرفته‌ایم:

```
void makeheap(int n, heap& H)
{
  index i;
  heap Hsub;
  for (i = d - 1; i >= 0; i--)
  for (all subtrees Hsub whose roots have depth i)
    siftdown(Hsub);
}
```

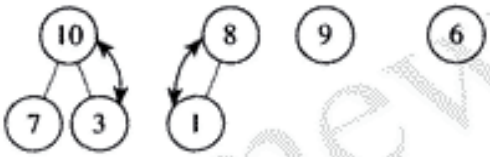
و بالاخره، شبه‌کد سطح بالا برای heapsort را ارائه می‌دهیم (فرض کرده‌ایم که کلیدها از قبل، از یک درخت دودویی کامل اصلی H مرتب شده‌اند).

شکل ۷-۷ استفاده از siftdown برای ایجاد یک heap از درخت اصلی. بعد از نمایش مراحل، زیردرخت راست که ریشه آن عمق ۲-d دارد، به یک heap تبدیل شده و سرانجام، کل درخت به یک heap تبدیل می‌شود.

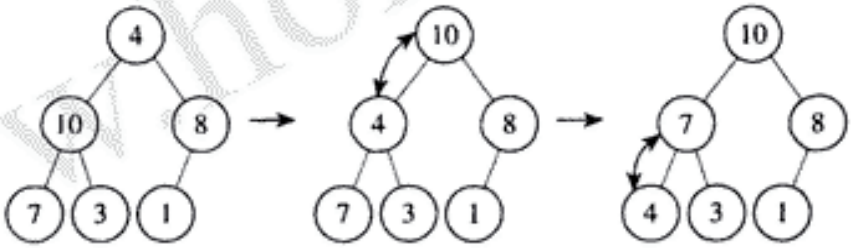
(a) The initial structure



(b) The subtrees, whose roots have depth $d - 1$, are made into heaps



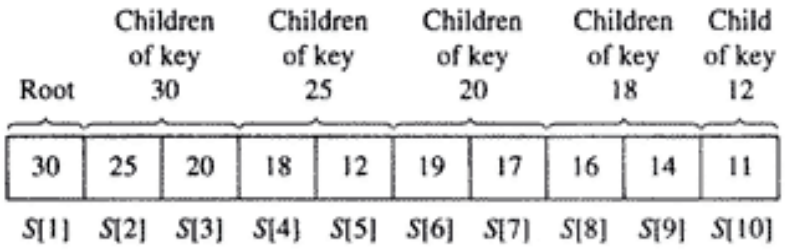
(c) The left subtree, whose root has depth $d - 2$, are made into a heap




```
void heapsort(int n, heap H, keytype S[ ])
{
   makeheap(n, H);
   removekeys(n, H, S):
}
```

شاید تصور شود که ما حقیقت را نگفته‌ایم زیرا این الگوریتم heapsort، یک مرتب‌سازی درجا به نظر نمی‌رسد. در اینصورت، ما به یک فضای اضافی برای heap نیاز داریم. به هر حال، یک heap را با استفاده از پشته به اجرا درآورده و نشان می‌دهیم که آرایهٔ مشابهی که ورودی (کلیدهایی که باید مرتب شوند) را ذخیره می‌کند، می‌تواند برای اجرای heap بکار گرفته شود و البته هرگز به طور همزمان، به یک اندیس آرایه مشابه برای بیش از یک منظور نیازمند نخواهیم شد.

شکل ۸-۷ آرایه متناظر با heap در شکل ۵-۷.



۲-۶-۷ یک اجرا از Heapsort

ما می‌توانیم یک درخت دودویی کامل اصلی را در یک آرایه، با ذخیره کردن ریشه در اولین اندیس آرایه، فرزندان چپ و راست در دومین و سومین اندیس، به همین ترتیب فرزندان چپ و راست فرزند چپ ریشه در چهارمین و پنجمین اندیس و الی آخر ارائه دهیم. شکل ۸-۷، آرایهٔ متناظر با heap در شکل ۵-۷ است. توجه کنید که شاخص فرزند چپ یک گره، دو برابر شاخص آن گره و شاخص فرزند راست یک گره، یکی بزرگتر از دو برابر شاخص آن گره می‌باشد. به خاطر دارید که در شبه‌کد سطح بالای heapsort، نیاز داشتیم به اینکه در ابتدا کلیدها در یک درخت دودویی کامل اصلی باشند. اگر ما کلیدها را با نظمی دلخواه در یک آرایه جای دهیم، آنها با توجه به مطالب فوق در چند درخت دودویی کامل اصلی سازماندهی خواهند شد. شبه‌کد سطح بالای زیر، از این خاصیت استفاده می‌کند.

ساختار داده‌ای Heap

```
struct heap
{
   keytype S[1..n];                    // S is indexed from 1 to n.
   int heapsize;                       // heapsize only takes
                                       // the values 0 through n.
};

void siftdown (heap& H, index i)       // To minimize the number
{                                      // of assignment of records.
   index parent, largerchild; keytype siftkey;  // the key initially at the roo
   bool spotfound;                     // (siftkey) is not assigned to
                                       // node until its final positio
   siftkey = H.S[i];                   // has been determined.
   parent = i; spotfound = false;
   while (2*parent <= H.heapsize && ! spotfound) {
      if (2*parent < H.heapsize && H.S[2*parent] < H.S.[2*parent+1])
         largerchild = 2*parent + 1;   // Index of right child is 1
      else                             // more than twice that of
         largerchild = 2*parent;       // parent. Index of left child
```

```
    if (siftkey < H.S[largerchild]) {          // is twice that of parent
      H.S[parent] = H.S[largerchild];
      parent = largerchild;
    }
    else
      spotfound = true;
  }
  H.S[parent] = siftkey;
}

keytype root (heap& H)
{
  keytype keyout;

  keyout = H.S[1];                             // Get key at the root.
  H.S[1] = H.S[heapsize];                      // Move bottom key to root.
  H.heapsize = H.heapsize - 1;                 // Delete bottom node.
  Siftdown(H, 1);                              // Restore heap property.
  return keyout;
}

void removekeys (int n,                        // H is passed by address
                 heap& H,                      // to save memory.
                 keytype S[ ])                 // S is indexed from 1 to n.
{
  index i;

  for (i = n; i >= 1; i- -)
    S[i] = root(H);
}

void makeheap (int n,
               heap& H)                        // H ends up a heap.
{
  index i;                                     // It is assumed that n keys
                                               // are in the array H.S.

  H.heapsize = n;
  for (i = ⌊n / 2⌋; i >= 1; i- -)              // Last node with depth
    siftdown (H, i);                           // d - 1, that has children, is
}                                              // in slot ⌊n / 2⌋ in the array.
```

حال می‌توانیم یک الگوریتم برای heapsort ارائه دهیم. در الگوریتم فرض شده است کلیدهایی که باید مرتب شوند در H.S قرار دارند. این امر می‌تواند آنها را به طور خودکار و بر اساس فرم نمایشی شکل ۸-۷، در یک درخت دودویی کامل اصلی سازماندهی کند. بعد از اینکه این درخت دودویی به یک heap

شد، کلیدها با شروع از اندیس nام آرایه و ادامه آن تا اولین اندیس، از heap حذف می‌شوند. از آنجائیکه کلیدها در یک دنباله مرتب شده و به ترتیب مشابهی در آرایه خروجی جایگزین می‌شوند، لذا می‌توانیم از H.S به عنوان آرایهٔ خروجی استفاده نمائیم؛ بدون آنکه امکان دوباره‌نویسی یک کلید را در heap فراهم کنیم. این استراتژی، یک الگوریتم درجا به صورت زیر ارائه می‌دهد.

الگوریتم ۵-۷ Heapsort

مسئله: n کلید را به ترتیب غیرنزولی مرتب کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از n کلید مرتب‌شده در یک آرایه H، برگرفته از یک heap.

خروجی: کلیدهایی مرتب‌شده به ترتیب غیرنزولی در آرایه H.S.

```
void heapsort(int n, heap& H)
{
   makeheap(n, H);
   removekeys(n, H, H.S);
}
```

تحلیل پیچیدگی زمانی بدترین حالت تعداد مقایسه کلیدها در الگوریتم ۵-۷ (Heapsort)

عمل مبنایی: مقایسه کلیدها در روال Sifrdown.

اندازه ورودی: n، تعداد کلیدهایی که باید مرتب شوند.

هر دو روال makeheap و removekeys، روال siftdown را فراخوانی می‌کنند. ما این دو روال را به صورت مجزا تحلیل می‌کنیم. تحلیل ما بر اساس n توانی از ۲ انجام می‌گیرد، آنگاه می‌توانیم با استفاده از قضیه B-۴ در ضمیمه B، n را به یک حالت کلّی تعمیم دهیم.

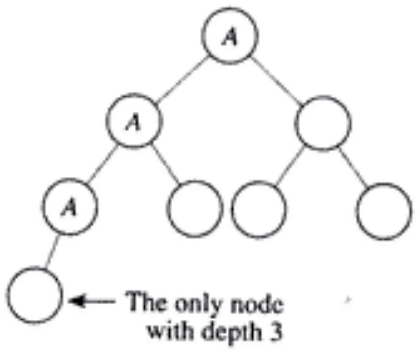
تحلیل Makeheap

فرض می‌کنیم که d عمق یک درخت دودویی کامل اصلی (ورودی) باشد. شکل ۹-۷ نشان می‌دهد که وقتی n توانی از ۲ است، عمق درخت (d) برابر $\lg n$ است و دقیقاً یک گره با این عمق وجود دارد، همچنین این گره دارای d پدربزرگ در درخت می‌باشد. اگر گره‌ای با سطح d موجود نباشد، تعداد گره‌هایی که از طریق کلیهٔ کلیدهای غربال شده بدست می‌آیند حداکثر برابر است با

$$\sum_{j=0}^{d-1} 2^j (d-j-1) = (d-1)\sum_{j=0}^{d-1} 2^j - \sum_{j=0}^{d-1} j(2^j) = 2^d - d - 1$$

تساوی اخیر با استفاده از نتیجه مثال A-۳ و A-۵ در ضمیمه A و انجام چند محاسبه جبری حاصل می‌شود. به خاطر دارید که ما برای بدست آوردن حد بالای حقیقی مجموع تعداد گره‌ها از طریق تمامی

شکل ۹-۷ یک مثال با n=۸ که نشان می‌دهد اگر یک درخت دودویی کامل اصلی دارای n گره(n توانی از ۲) باشد، آنگاه عمق d درخت برابر lg n بوده و یک گره با عمق d وجود خواهد داشت، همچنین این گره دارای d جد است. سه تا از اجداد این گره با حرف "A" مشخص شده‌اند.



کلیدهای غربال شده، به حاصل‌جمع d با این حد نیازمند بودیم. بنابراین، حد بالای حقیقی برابر است با

$$۲^d - d - ۱ + d = ۲^d - ۱ = n - ۱$$

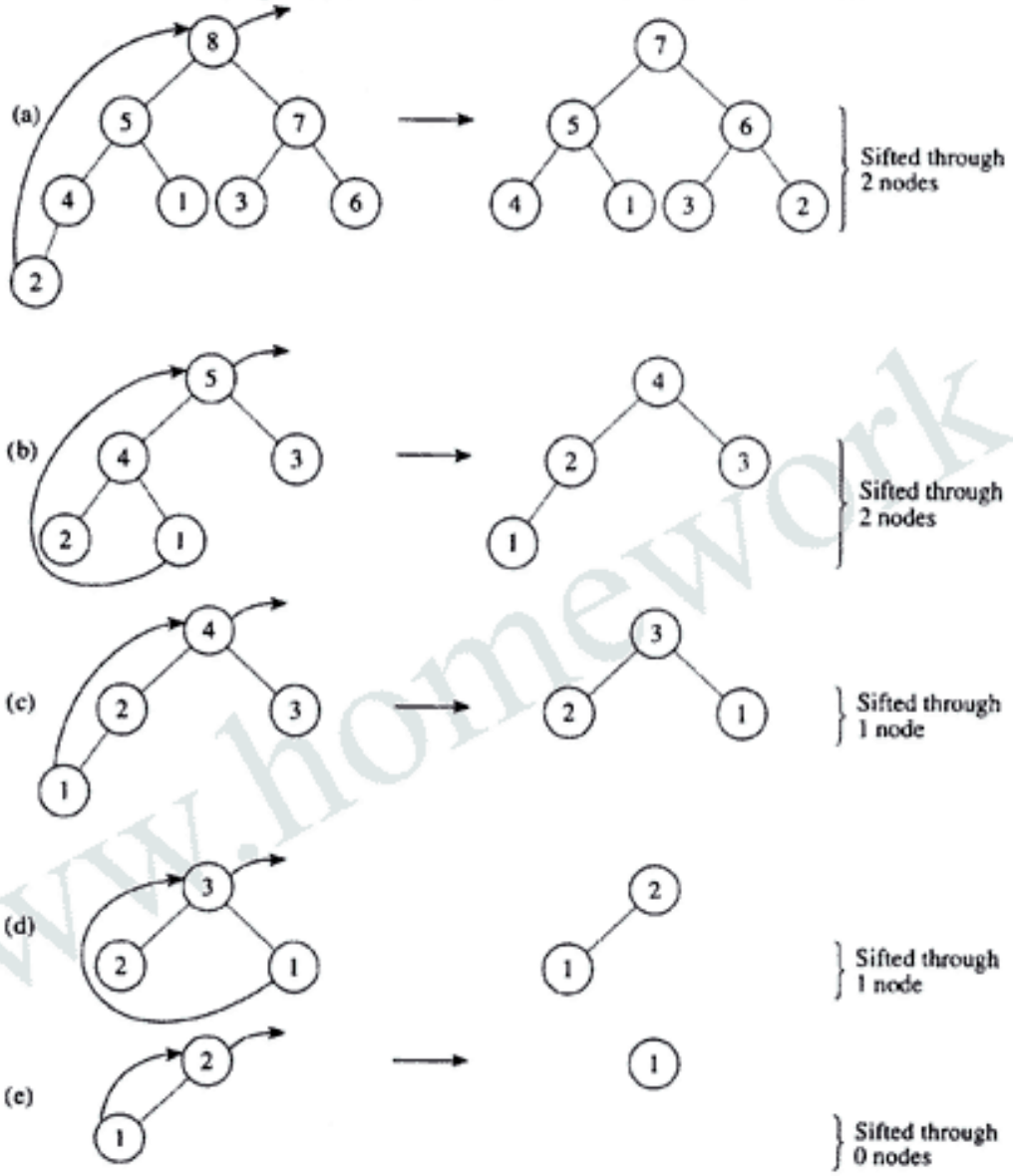
تساوی دوم از آنجا نتیجه می‌شود که وقتی n توانی از ۲ است، آنگاه $d = lg\ n$ خواهد بود. هر زمانی که یک کلید از طریق یک گره جابجا می‌شود یعنی یک گذر از حلقه While در روال siftdown وجود دارد. از آنجائیکه در هر گذر از این حلقه، دو مرتبه مقایسه کلید صورت می‌پذیرد، لذا حداکثر تعداد مقایسه‌ای که توسط Makeheap روی کلیدها انجام می‌شود برابر است با $۲(n - ۱)$. این نتیجه بسیار جالب توجه است زیرا توانستیم به یک ساختار زمان-خطی برای heapsort دست یابیم. اگر می‌توانستیم کلیدها را نیز بر اساس یک تابع زمان-خطی حذف کنیم، آنگاه یک الگوریتم مرتب‌سازی خطی داشتیم. همانطوریکه خواهیم دید، چنین اتفاقی نمی‌افتد.

تحلیل removekeys

شکل ۷-۱۰ حالتی را نشان می‌دهد که در آن n = ۸ و d = lg ۸ = ۳ است. همانطوری که در شکل ۷-۱۰(a) و (b) مشاهده می‌کنید، هنگامی که اولین و چهارمین کلید حذف می‌شوند برای حرکت دادن کلید به ریشه، حداکثر $d - ۱ = ۲$ گره بایستی جابجا شوند. به عبارتی، برای حذف اولین و چهارمین کلید دقیقاً اتفاق مشابهی می‌افتد. بنابراین، هنگامی که چهار کلید اول حذف می‌شوند، کلید حرکت داده شده به ریشه حداکثر از طریق ۲ گره جابه‌جا می‌شود. همانطوریکه در شکل ۷-۱۰(c) و (d) مشاهده می‌کنید، هنگامی که هر یک از دو کلید بعدی حذف می‌شوند، کلید حرکت داده شده به ریشه از طریق $d-۲=۱$ گره جابه‌جا می‌شود و سرانجام مطابق شکل ۷-۱۰(e)، هنگامی که کلید بعدی حذف می‌شود، کلید حرکت داده شده به ریشه از طریق صفر گره جابه‌جا می‌شود. به عبارتی، وقتی آخرین کلید حذف می‌شود، هیچ جابجایی صورت نمی‌گیرد. مجموع تعداد گره‌هایی که از طریق کلیهٔ گره‌ها جابه‌جا می‌شوند برابراست با

$$۱(۲) + ۲(۴) = \sum_{j=۱}^{۳-۱} j\ ۲^j$$

شکل ۱۰-۷ حذف کلیدها از یک heap با هشت گره. (a) حذف گرهٔ اول، (b) حذف گرهٔ چهارم (c) حذف گره پنجم، (d) حذف گره ششم و (e) حذف گره هفتم. کلید حرکت داده شده به ریشه، از طریق تعداد گره‌های نشان داده شده در سمت راست جابجا می‌شود.



تعمیم این مسئله به n (توان دلخواهی از ۲) کار مشکلی نیست. از آنجائیکه هر گاه یک کلید از طریق یک گره جابه‌جا می‌شود، یک گذر از حلقهٔ while در روال siftdown انجام می‌شود و از آنجائیکه در هر گذر از این حلقه، دو مرتبه مقایسهٔ کلیدها صورت می‌گیرد، لذا تعداد مقایسه‌هایی که توسط removekeys روی کلیدها انجام می‌شود برابراست با

$$۲\sum_{j=۱}^{d-۱} j۲^j = ۲(d۲^{۲} - ۳^{d+۱} + ۲) = ۲n \lg n - ۴n + ۴$$

اولین تساوی با استفاده از نتیجه مثال A-۵ در ضمیمه A و انجام چند محاسبه جبری بدست می‌آید و تساوی دوم از این حقیقت ناشی می‌شود که وقتی n توانی از ۲ است، آنگاه $d = lg\ n$ خواهد بود.

ترکیب دو تحلیل

ترکیب تحلیل‌های makeheap و removekeys نشان می‌دهد که تعداد مقایسهٔ کلیدها در heapsort حداکثر برابر است با

$$۲(n-۱) + ۲n\ lg n - ۴n + ۴ = ۲(n\ lgn - n + ۱) \approx ۲n\ lgn$$

که در آن n توانی از ۲ است. بنابراین،

$$w(n) \approx ۲n\ lg n \in \theta(n\ lg\ n)$$

می‌توانیم نشان دهیم که W(n)، احتمالاً غیرنزولی نهایی خواهد بود و در این صورت بر اساس قضیه B-۴ در ضمیمه B، برای n در حالت کلی خواهیم داشت :

$$w(n) \in \theta(n\ lg\ n)$$

به نظر می‌رسد که تحلیل پیچیدگی زمانی حالت میانی Heapsort، کار آسانی نباشد. به هر حال، مطالعات تجربی نشان می‌دهد که حالت میانی آن خیلی بهتر از بدترین حالت آن نیست. این بدین معناست که پیچیدگی زمانی *حالت میانی تعداد مقایسه کلیدها برای Heapsort* برابر است با

$$A(n) \approx ۲n\ lg\ n$$

در تمرینات از شما می‌خواهیم که نشان دهید پیچیدگی زمانی *بدترین حالت تعداد انتساب رکوردها برای heapsort* برابر است با

$$W(n) \approx n\ lg\ n$$

و در پایان، همانطوریکه تاکنون بحث شد، کارایی فضای اضافی را تعیین می‌کنیم.

تحلیل کاربرد فضای اضافی الگوریتم ۵-۷ (Heapsort)

heapsort یک مرتب‌سازی در جا است. بدین معنا که فضای اضافی آن در $\theta(۱)$ می‌باشد.

همانطوریکه در بخش ۲-۷ نشان دادیم، heapsort یک مثال از مرتب‌سازی انتخابی است زیرا با انتخاب رکوردها و جایگزینی آنها در موقعیت مناسب، عمل مرتب‌سازی را انجام می‌دهد.

۷-۷ مقایسهٔ Mergesort، Quicksort و Heapsort

جدول ۲-۷، نتایج مربوط به این سه الگوریتم را به طور خلاصه نشان می‌دهد. از آنجائیکه Heapsort، از لحاظ تعداد مقایسه کلیدها و انتساب رکوردها در حالت میانی بدتر از Quicksort است و از آنجائیکه

جدول ۲-۷ خلاصه‌ای از تحلیل الگوریتمهای مرتب سازی O(n lgn)

Algorithm	Comparisons of Keys	Assignments of Records	Extra Space Usage
Mergesort (Algorithm 2.4)	$W(n) = n \lg n$ $A(n) = n \lg n$	$T(n) = 2n \lg n$	$\Theta(n)$ records
Mergesort (Algorithm 7.4)	$W(n) = n \lg n$ $A(n) = n \lg n$	$T(n) = 0^{\dagger}$	$\Theta(n)$ links
Quicksort (with improvements)	$W(n) = n^2/2$ $A(n) = 1.38n \lg n$	$A(n) = 0.69n \lg n$	$O(\lg n)$ indices
Heapsort	$W(n) = 2n \lg n$ $A(n) = 2n \lg n$	$W(n) = n \lg n$ $A(n) = n \lg n$	In-place

*Entries are approximate; the average cases for Mergesort and Heapsort are slightly better than the worst cases.
†If it is required that the records be in sorted sequence in contiguous array slots, the worst case is in O(n).

فضای اضافی مورد استفاده توسط Quicksort، حداقل است، لذا معمولاً Quicksort نسبت به Heapsort ارجح‌تر است. از آنجائیکه در اجرای Mrgesort (الگوریتم ۲-۲ و ۴-۲) از یک آرایهٔ اضافی از رکوردها استفاده می‌شود و از آنجائیکه Mergesort، همیشه در حدود سه برابر تعداد انتساب رکوردهایی که توسط Quicksort صورت می‌گیرد را در حالت میانی انجام می‌دهد، لذا Quicksort معمولاً نسبت به Mergesort ارجح‌تر است، با وجود اینکه Quicksort به وضوح مقایسات بیشتری را در حالت میانی انجام می‌دهد. به هر حال، نسخهٔ پیوندی Mergesort (الگوریتم ۴-۷) تقریباً تمام مزایای Mergesort را دارا است، الّا اینکه فضای اضافی مورد استفاده در آن برابر $\theta(n)$ پیوند اضافی است.

۸-۷ حدود پائین برای مرتب‌سازی با مقایسه کلیدها

ما توانستیم الگوریتم‌های مرتب‌سازی را با درجه $\theta(n \lg n)$ ارائه کنیم که هر یک پیشرفت قابل توجهی را روی الگوریتم‌های زمان-مربعی نشان می‌دهند. یک سؤال مهم اینست که آیا باز هم می‌توانیم الگوریتم‌های مرتب‌سازی ارائه نمائیم که پیچیدگی زمانی آنها بهتر از این باشد؟ ما نشان می‌دهیم که تا زمانی که خود را به مرتب‌سازی با مقایسه کلیدها محدود می‌کنیم، ارائه چنین الگوریتم‌هایی امکان‌پذیر نیست.

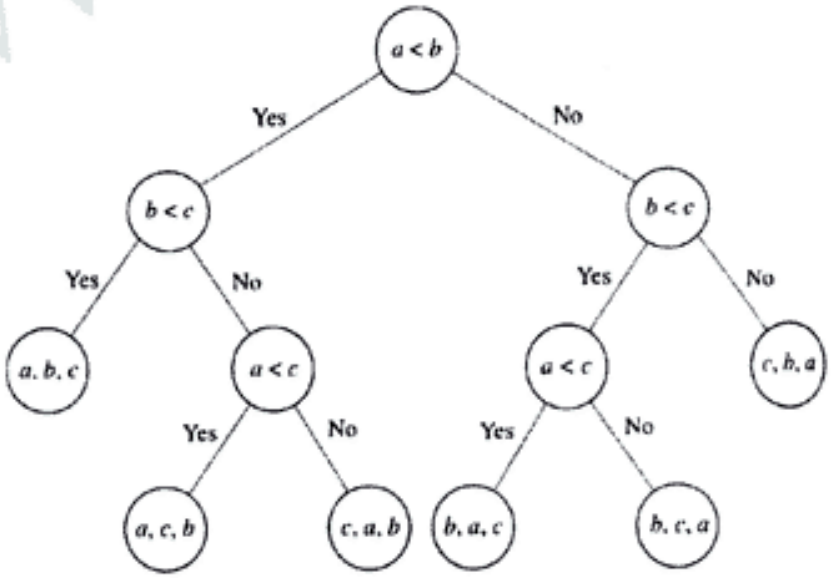
اگر ما بتوانیم الگوریتم مرتب‌سازی احتمالی را در نظر بگیریم، می‌توانیم نتایجی را برای الگوریتم‌های مرتب‌سازی قطعی بدست آوریم (برای آشنائی با بحث الگوریتم‌های احتمالی و قطعی، به بخش ۳-۵ مراجعه کنید). همانند بخش ۳-۷، فرض می‌کنیم که n کلید مجزا از هم (بطور ساده، اعداد صحیح مثبت ۱، ۲،... و n) را در اختیار داریم زیرا می‌توانیم ۱ را برای کوچکترین کلید، ۲ را برای دومین کلید کوچکتر و الی آخر در نظر بگیریم.

۱-۸-۷ درخت‌های تصمیم برای الگوریتم‌های مرتب‌سازی

الگوریتم زیر را برای مرتب‌سازی سه کلید در نظر بگیرید :

```
void sortthree(keytype S[])                    // S از ۱ تا ۳ شاخص‌دهی شده است. //
{
    keytype a, b, c;
    a = S[1];  b = S[2]; c = S[3];
    if (a < b)
      if (b < c)
        S = a, b, c;                           // s[1]=a;  s[2]=b;  s[3]=c;  یعنی
      else if (a < c)
        S = a, c, b;
      else
        S = c, a, b;
    else if (b < c)
      if (a < c)
        S = b, a, c;
      else
        S = b, c, a;
    else
      S = c, b, a;
}
```

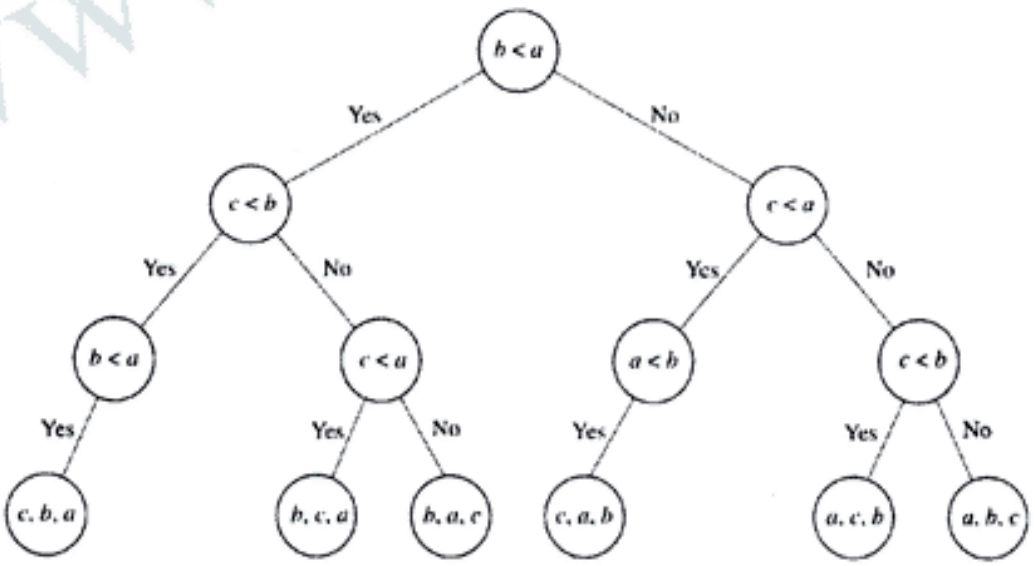
ما می‌توانیم یک درخت دودویی را به صورت زیر با روال Sortthree متناظر کنیم. مقایسه a و b در ریشه قرار می‌گیرد. فرزند چپ ریشه شامل مقایسه‌ای است که در صورت برقراری $a<b$ انجام می‌شود. در غیراینصورت، یعنی در صورتی که $a \geq b$ باشد، فرزند راست را در نظر می‌گیریم. این روند تولید گره‌ها را ادامه می‌دهیم تا اینکه تمامی مقایسه‌های ممکن انجام شده توسط الگوریتم، به گره‌هایی منتسب شود. کلیدهای مرتب‌شده، در برگها ذخیره می‌شوند. شکل ۱۱-۷، یک درخت کامل را نشان می‌دهد. به این درخت، **درخت تصمیم (decision tree)** گوئیم، زیرا در هر گره بایستی یک تصمیم گرفته شود تا



شکل ۱۱-۷ درخت تصمیم متناظر با روال Sortthree

بتوانیم به گرهٔ بعدی دست یابیم. نقش روال Sortthree برای یک ورودی خاص، متناظر است با یک مسیر منحصربفرد از ریشه به یک برگ روی درخت تصمیم، که این مسیر توسط ورودی تعیین می‌شود. برای هر ترتیبی از این سه کلید، یک برگ وجود دارد، زیرا الگوریتم می‌تواند هر ورودی ممکن به اندازه ۳ را مرتب نماید.

یک درخت تصمیم برای مرتب‌سازی n کلید **معتبر** است اگر برای هر خروج ممکن، یک مسیر از ریشه به یک برگ وجود داشته باشد که آن خروجی را مرتب نماید. این بدین معناست که آن می‌تواند هر ورودی به اندازهٔ n را مرتب نماید. برای مثال، درخت تصمیم شکل ۱۱-۷ برای مرتب‌سازی سه کلید معتبر است اما اگر حتی یک شاخه از درخت حذف شود، دیگر اعتباری نخواهد داشت. با هر الگوریتم قطعی برای مرتب‌سازی n کلید، حداقل یک درخت تصمیم معتبر متناظر خواهد بود. درخت تصمیم در شکل ۱۱-۷ با روال Sortthree و درخت تصمیم در شکل ۱۲-۷ با مرتب‌سازی تبادلی (جهت مرتب‌سازی سه کلید) متناظر هستند. در این درخت a، b و c مجدداً مقادیر اولیه $S[3], S[2], S[1]$ را به خود می‌گیرند. وقتی یک گره، به عنوان مثال، شامل مقایسه $c<b$ شود، بدین معنا نیست که مرتب‌سازی تبادلی مقدار $S[3]$ را با $S[2]$ مقایسه می‌کند، بلکه عنصری از آرایه که مقدار آن برابر C است با عنصری که مقدار آن b است، مقایسه می‌شود. توجه کنید که در درخت شکل ۱۲-۷، گره سطح ۲ شامل مقایسه $b<a$ فرزند راست ندارد، زیرا پاسخ "نه" به این مقایسه با پاسخهای بدست آمده از مسیری که به آن گره می‌رسد تناقض دارد. به این معنا که این گره در شاخه چپ $b<a$ واقع در ریشه قرار دارد که در این صورت حتماً $b<a$ خواهد بود. مرتب‌سازی تبادلی در این نقطه، یک مقایسهٔ غیرضروری انجام می‌دهد. مرتب‌سازی تبادلی "نمی‌داند" که پاسخ این سؤال "بله" می‌باشد. این اتفاق، اغلب در الگوریتم‌های مرتب‌سازی زیربهینه رخ می‌دهد. یک درخت تصمیم را **هرس‌شده** گوئیم اگر هر برگ بتواند با



شکل ۱۲-۷ درخت تصمیم متناظر با مرتب‌سازی تبادلی در حین مرتب‌سازی سه کلید.

ایجاد یک رشته تصمیمات متوالی به ریشه دست یابد. درخت تصمیم در شکل ۱۲-۷ هرس‌شده است. با هر الگوریتم قطعی برای مرتب‌سازی n کلید، یک درخت تصمیم معتبر هرس‌شده متناظر می‌باشد. لذا پیش‌قضیه زیر را مطرح می‌کنیم.

پیش‌قضیه ۱-۷ با هر الگوریتم قطعی برای مرتب‌سازی n کلید، یک درخت تصمیم دودویی معتبر هرس‌شده دقیقاً شامل !n برگ متناظر است.

اثبات: همانطوری که مشاهده کردید، یک درخت تصمیم معتبر هرس‌شده به یک الگوریتم مرتب‌سازی n کلیدی مرتبط گردید. هنگامی که همه کلیدها از هم مجزا هستند، نتیجه یک مقایسهٔ "<" یا ">" می‌باشد. بنابراین، هر گره در این درخت، حداکثر دو فرزند دارد و این بدین معناست که این درخت، یک درخت دودویی است. حال می‌خواهیم نشان دهیم که درخت به تعداد !n برگ دارد. از آنجائیکه n کلید مختلف می‌توانند !n ورودی مختلف بسازند و از آنجائیکه یک درخت تصمیم برای مرتب‌سازی n کلید مجزا تنها زمانی معتبر است که برای هر ورودی، یک برگ وجود داشته باشد، لذا درخت دارای حداقل !n برگ خواهد بود. بدلیل اینکه برای هر یک از !n ورودی مختلف در درخت، یک مسیر منحصربفرد وجود دارد و بدلیل اینکه هر برگ در یک درخت تصمیم هرس‌شده بایستی قابل دسترسی باشد، لذا درخت نمی‌تواند بیش از !n برگ داشته باشد. بنابراین، درخت دقیقاً دارای !n برگ است.

با بکارگیری پیش‌قضیه ۱-۷ و با استفاده از درختهای دودویی با !n برگ می‌توانیم حدودی را برای مرتب‌سازی n کلید مجزا تعیین کنیم. این عمل را در زیر انجام می‌دهیم.

۲-۸-۷ حدود پائین برای بدترین حالت

برای بدست آوردن یک حد برای بدترین حالت تعداد مقایسه کلیدها، به پیش‌قضیه زیر نیاز داریم:

پیش‌قضیه ۲-۷ بدترین حالت تعداد مقایسات انجام شده توسط یک درخت تصمیم برابر عمق آن است.

اثبات: برای یک ورودی معین، تعداد مقایسات انجام شده توسط یک درخت تصمیم برابر تعداد گره‌های داخلی روی مسیر است که به آن ورودی ختم می‌شود. تعداد گره‌های داخلی، همان طول مسیر است. بنابراین، بدترین حالت تعداد مقایسه‌های انجام شده توسط یک درخت تصمیم، طول بلندترین مسیر به یک برگ، یعنی عمق درخت تصمیم است.

بر اساس پیش‌قضیه‌های ۱-۷ و ۲-۷، تنها به یک حد پائین روی عمق یک درخت دودویی شامل !n برگ نیازمندیم تا بتوانیم حد پائینی را برای بدترین حالت بدست آوریم. حد پائین مورد نیاز روی عمق درخت با استفاده از پیش قضایا و قضایای زیر مشخص می‌شود:

پیش‌قضیه ۳-۷ اگر m تعداد برگهای یک درخت دودویی و d عمق آن باشد، آنگاه $d \geq \lceil lg\ m \rceil$ است.

اثبات: ابتدا با استفاده از استقراء d نشان می‌دهیم که $m \leq 2^d$ است.

پایه استقراء: یک درخت دودویی با عمق صفر، یک گره دارد که هم ریشه وهم تنها برگ درخت محسوب می‌شود. بنابراین برای چنین درختی، تعداد برگها (m) برابر یک و $m \leq 2^0$ است.

فرض استقراء: فرض می‌کنیم که برای درخت دودویی با عمق d، $m \leq 2^d$ (m، تعداد برگها) است.

گام استقراء: بایستی نشان دهیم که برای هر درخت دودویی با عمق d+1 داریم:

$$2^{d+1} \geq m'$$

که در آن m'، تعداد برگها است. اگر ما تمامی برگها را از چنین درختی حذف کنیم، درختی با عمق d خواهیم داشت که برگهای آن والدین برگهای درخت اصلی (درخت با عمق d+1) هستند. اگر m تعداد این والدین باشد، آنگاه با توجه به فرض استقراء داریم :

$$2^d \geq m$$

از آنجائیکه هر والدین می‌تواند حداکثر دو فرزند داشته باشد، لذا

$$2m \geq m'$$

با ترکیب این دو نامساوی داریم :

$$2^{d+1} \geq 2m \geq m'$$

که این اثبات استقراء را کامل می‌کند. با گرفتن lg از دو طرف نامساوی خواهیم داشت :

$$d \geq lg\ m$$

و از آنجائیکه d یک عدد صحیح است، بنابراین

$$d \geq \lceil lg\ m \rceil$$

قضیه ۲-۷ هر الگوریتم قطعی که تنها با استفاده از مقایسه کلیدها، n کلید مجزا را مرتب می‌کند بایستی در بدترین حالت، حداقل $\lceil lg(n!) \rceil$ مقایسه انجام دهد.

اثبات: با پیش‌قضیه ۱-۷ می‌توان به چنین الگوریتمی، یک درخت تصمیم دودویی معتبر هرس‌شده را متناظر کرد. با پیش‌قضیه ۳-۷ درمی‌یابیم که عمق این درخت بزرگتر یا مساوی $\lceil lg(n!) \rceil$ است و از آنجائیکه پیش‌قضیه ۲-۷ بیان می‌کند که بدترین حالت تعداد مقایسات در هر درخت تصمیم برابر عمق آن است، لذا به اثبات تئوری دست می‌یابیم.

پیش قضیه ۴-۷ برای هر عدد صحیح مثبت n داریم :

$$lg(n!) \geq n\ lg\ n - 1/45n$$

اثبات: برای اثبات، آشنایی با انتگرال‌ها لازم است. داریم

$$lg(n!) \geq lg[n(n-۱)(n-۲)....(۲)۱]$$
$$= \sum_{i=۲}^{n} lg\, i \qquad (lg\, ۱ = ۰ \text{ زیرا})$$
$$\geq \int_{۱}^{n} lg\, x\, dx = \frac{۱}{ln\, ۲}(n\, ln\, n - n + ۱) \geq n\, lg\, n - ۱/۴۵n$$

قضیه ۳-۷ هر الگوریتم قطعی که تنها با استفاده از مقایسه کلیدها، n کلید مجزا را مرتب می‌کند بایستی در بدترین حالت حداقل $\lceil n\, lg\, n - ۱/۴۵n \rceil$ مقایسه انجام دهد.

اثبات: اثبات این قضیه، با استفاده از قضیه ۴-۷ و پیش‌قضیه ۴-۷ به سادگی انجام می‌شود.

مشاهده می‌کنیم که کارایی بدترین حالت مرتب‌سازی ادغامی برابر $n\, lg\, n - (n-۱)$، نزدیک به حالت بهینه است. در ادامه، نشان می‌دهیم که این کارایی برای حالت میانی آن نیز برقرار است.

۳-۸-۷ حدود پائین برای حالت میانی

ما نتایج فوق را، با این فرض که تمامی ترتیبهای ممکن با احتمالاتی یکسان و مشابه به عنوان ورودی در نظر گرفته شده‌اند، بدست آورده‌ایم. اگر درخت تصمیم دودویی معتبر هرس‌شده با یک الگوریتم مرتب‌سازی قطعی که n کلید مجزا را مرتب می‌کند متناظر باشد بطوری که تمامی گره‌های مقایسه‌ای تنها با یک فرزند (همانند درخت شکل ۱۲ - ۷) مرتبط باشند، آنگاه می‌توانیم هر کدام از این گره‌ها را با فرزندش جایگزین نموده و فرزند را حذف کنیم تا یک درخت تصمیم که با استفاده از همان تعداد مقایسات درخت اصلی، عمل مرتب‌سازی را انجام می‌دهد، بدست آوریم. هر گرهٔ غیربرگ در درخت جدید، دقیقاً شامل دو فرزند است. یک درخت دودویی که هر گرهٔ غیربرگ آن، دو فرزند داشته باشد، به **درخت سطح ۲ (2-tree)** موسوم است. ما این نتیجه را در قاعده زیر خلاصه می‌کنیم.

پیش‌قضیه ۵-۷ با هر درخت تصمیم دودویی معتبر هرس‌شده برای مرتب‌سازی n کلید مجزا، یک درخت تصمیم معتبر هرس‌شدهٔ سطح ۲ با حداقل همان کارایی درخت اصلی وجود دارد.

اثبات: اثبات این پیش‌قضیه از بحث قبلی تبعیت می‌کند.

طول مسیر خارجی (EPL) یک درخت، مجموع طول همه مسیرها از ریشه به برگها است. بعنوان مثال، برای درخت شکل ۱۱-۷ داریم:

$$EPL = ۲ + ۳ + ۳ + ۳ + ۳ + ۲ = ۱۶$$

به خاطر دارید که تعداد مقایسات انجام شده توسط یک درخت تصمیم از ریشه تا یک برگ، برابر طول مسیر تا آن برگ است. بنابراین، EPL یک درخت تصمیم برابر مجموع تعداد مقایسات انجام شده توسط

درخت تصمیم برای مرتب‌سازی همهٔ ورودیهای ممکن است. از آنجائیکه n! ورودی مختلف برای n کلید مجزا وجود دارد و از آنجائیکه فرض نمودیم همه ورودیها از احتمال یکسانی برخوردارند، لذا میانگین تعداد مقایسات انجام شده توسط یک درخت تصمیم برای مرتب‌سازی n کلید مجزا برابر است با $EPL/n!$. با استفاده از این نتیجه می‌توانیم یک پیش‌قضیهٔ مهم را ثابت کنیم. ابتدا minEPL(m) را به عنوان کوچکترین EPL از درخت سطح۲ شامل m برگ تعریف نموده، سپس پیش‌قضیه زیر را بیان می‌کنیم.

پیش‌قضیه ۶-۷ هر الگوریتم قطعی که تنها با استفاده از مقایسه کلیدها، n کلید مجزا را مرتب می‌کند بایستی در حالت میانی، حداقل $minEPL(n!)/n!$ مقایسه انجام دهد.

اثبات: پیش‌قضیه ۱-۷ بیان می‌کند که هر الگوریتم قطعی برای مرتب‌سازی n کلید مجزا با یک درخت تصمیم دودویی معتبر هرس‌شده شامل n! برگ متناظر است و پیش‌قضیه ۵-۷ می‌گوید که ما می‌توانیم درخت تصمیم را به یک درخت سطح۲ با همان کارایی درخت اصلی تبدیل کنیم. از آنجائی که درخت اصلی n! برگ دارد، لذا بایستی درخت سطح ۲ را از آن بدست آوریم. اکنون پیش‌قضیه از بحثهای قبلی تبعیت می‌کند.

برای بدست آوردن یک حد پائین برای حالت میانی با استفاده از پیش‌قضیه ۶-۷، تنها به تبعیت یک حد پائین برای minEPL(m) نیازمندیم که این کار با بیان چهار پیش‌قضیه زیر انجام می‌شود.

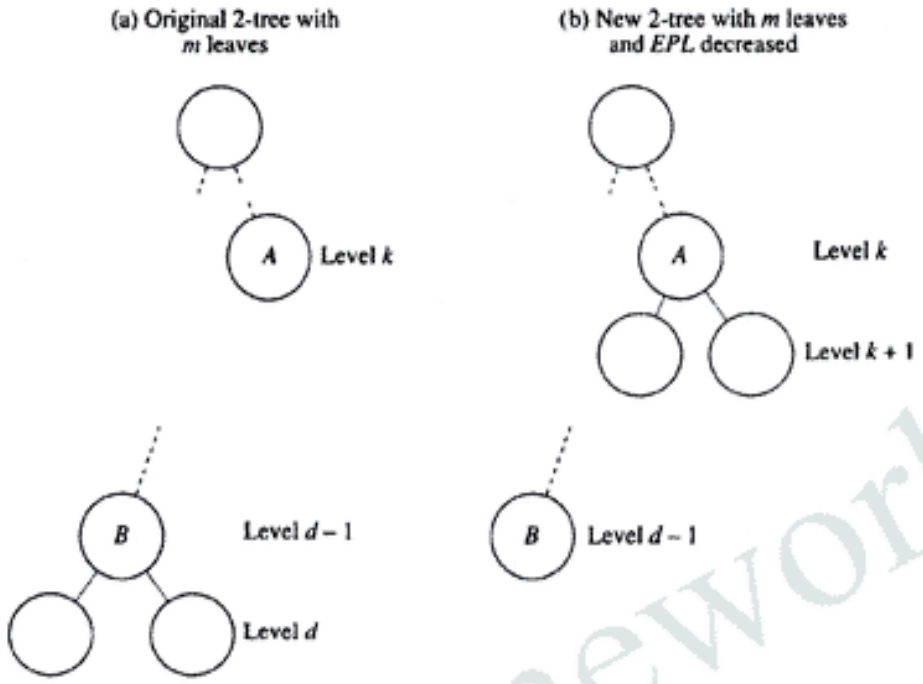
پیش‌قضیه ۷-۷ هر درخت سطح۲ که m برگ دارد و EPL آن معادل minEPL(m) است بایستی همهٔ برگهایش حداکثر در دو سطح زیرین قرار گرفته باشند.

اثبات: فرض کنید که تمامی برگهای یک درخت سطح۲ در دو سطح زیرین قرار ندارند. d را عمق درخت، A را یک برگ در درخت که در دو سطح زیرین نیست، و k را عمق A در نظر می‌گیریم. از آنجائیکه عمق گره‌های سطح زیرین برابر d می‌باشد، لذا $K \leq d-2$ است. نشان می‌دهیم که این درخت نمی‌تواند با ارائهٔ یک درخت سطح۲ با تعداد برگهای یکسان ولی با EPL پائین‌تر، مقدار EPL را کاهش می‌دهد. ما می‌توانیم با انتخاب یک گرهٔ غیربرگ B در سطح d-۱ در درخت اصلی، حذف فرزندان آن و دادن این فرزندان به A (مطابق شکل ۱۵-۷) این کار را انجام دهیم. در درخت جدید، نه A و نه فرزندان B، هیچکدام برگ نیستند، در حالیکه در درخت قدیم به عنوان برگ محسوب می‌شدند. بنابراین، با استفاده از طول مسیر A و طولهای مسیرهای فرزندان B، EPL را کاهش داده‌ایم. به عبارتی، ما EPL را به وسیلهٔ

$$k+d+d=k+2d$$

کاهش داده‌ایم. اما اگر در درخت جدید هم A و هم فرزندان B برگ بودند، ولی در درخت قدیم، آنها به عنوان گره‌های غیربرگ محسوب می‌شدند، در این صورت نیز می‌توانستیم با استفاده از طول مسیر A و طولهای مسیرهای فرزندان B، EPL را کاهش دهیم. به عبارتی، ما EPL را به وسیلهٔ

شکل ۱۳-۷ درختهای (a) و (b) تعداد یکسانی برگ دارند، اما درخت (b)، EPL کوچکتری دارد.



$$d-1+k+1+k+1=d+2k+1$$

کاهش می‌دادیم. تغییر شبکه‌ای در EPL برابر است با

$$(d+2k+1)-(k+2d)=k-d+1\le d-2-d+1=-1$$

این نامساوی اتفاق می‌افتد، زیرا $k\le d-2$ است. از آنجائیکه تغییر شبکه‌ای عددی منفی است، لذا درخت کوچکتری دارد.

پیش‌قضیه ۸-۷ هر درخت سطح ۲ با عمق d که m برگ دارد و EPL آن مساوی minEPL(m) است باید $2m-2^d$ برگ در سطح $d-1$ و $3\cdot 2^d-2m$ برگ در سطح d داشته و هیچ برگی در سطوح دیگر نداشته باشد.

اثبات: بر اساس پیش‌قضیه ۷-۷ که بیان می‌کند همهٔ برگها در دو سطح زیرین قرار دارند و این نکته که گره‌های غیربرگ در درخت سطح ۲ بایستی دو فرزند داشته باشند، مشاهدهٔ این نکته که باید 2^{d-1} گره در سطح $d-1$ باشد کار مشکلی نیست. لذا اگر r تعداد برگها در سطح $d-1$ باشد، آنگاه تعداد گره‌های غیربرگ در این سطح برابر است با $r-2^{d-1}$. از آنجائیکه گره‌های غیربرگ در یک درخت سطح ۲ دقیقاً دو فرزند دارند، لذا برای هر گره غیر برگ در سطح $d-1$ دو برگ در سطح d وجود دارد و از آنجائیکه اینها تنها برگهای سطح d محسوب می‌شوند، لذا تعداد برگها در سطح d برابر است با $2(r-2^{d-1})$. بر اساس پیش‌قضیه ۷-۷ که بیان می‌کند همهٔ برگها در سطح d یا $d-1$ هستند، داریم

$$r+2(2^{d-1}-r)=m$$

که با ساده کردن عبارت فوق خواهیم داشت :

$$r = 2^d - m$$

بنابراین، تعداد برگهای سطح d نیز برابر است با

$$m - (2^d - m) = 2m - 2^d$$

پیش‌قضیه ۹-۷ برای هر درخت سطح ۲ که m برگ دارد و EPL آن مساوی minEPL(m) است، عمق d برابر است با $d = \lceil \lg m \rceil$

اثبات: ما حالتی را ثابت می‌کنیم که m توانی از ۲ است. اثبات حالت کلی بعنوان تمرین ارائه خواهد شد. اگر m توانی از ۲ باشد ، آنگاه برای برخی از اعداد صحیح k داریم :

$$m = 2^k$$

فرض کنیم که d، عمق یک درخت می‌نیمم و r، تعداد برگها در سطح $d-1$ باشد، آنگاه بر اساس پیش قضیه ۸-۷ داریم :

$$r = 2^d - m = 2^d - 2^k$$

از آنجائیکه $r \geq 0$ است، لذا $d \geq k$ می‌باشد. نشان می‌دهیم که پذیرفتن $d > k$ ما را به یک تناقض می‌رساند. اگر $d > k$ باشد، آنگاه

$$r = 2^d - 2^k \geq 2^{k+1} - 2^k = 2^k = m$$

و از آنجائیکه $r \leq m$ است، این بدین معناست که $r = m$ بوده و همه برگها در سطح $d-1$ قرار دارند. اما بایستی چند برگ نیز در سطح d وجود داشته باشد. این تناقض اشاره دارد به اینکه $d = k$ که دراینصورت $r = 0$ خواهد بود. چون $r = 0$ است، لذا

$$2^d - m = 0$$

یعنی $d = \lg m$ و چون m توانی از ۲ است، پس $\lceil \lg m \rceil = \lg m$ که این اثبات ما را کامل می‌کند.

پیش‌قضیه ۱۰-۷ برای همه اعداد صحیح $m \geq 1$ داریم:

$$min\,EPL(m) \geq m \lfloor \lg m \rfloor$$

اثبات: بر اساس پیش‌قضیه ۸-۷، هر درخت سطح ۲ که EPL را به حداقل می‌رساند، بایستی فقط $2^d - m$ برگ در سطح $d-1$ و $2m - 2^d$ برگ در سطح d داشته و هیچ برگی در سطوح دیگر نداشته باشد. بنابراین،

$$min\,EPL(m) = (2^d - m)(d-1) + (2m - 2^d)d = md + m - 2^d$$

لذا، بر اساس پیش‌قضیه ۹-۷ داریم :

$$min\,EPL(m) = m(\lceil \lg m \rceil) + m - 2^{\lceil \lg m \rceil}$$

اگر m توانی از ۲ باشد ،این عبارت معادل m lg m خواهد بود که در این حالت معادل $m \lfloor \lg m \rfloor$ است. اما اگر m توانی از ۲ نباشد ، آنگاه $\lceil \lg m \rceil = \lfloor \lg m \rfloor + 1$ است که در اینصورت

$$\min EPL(m) = m(\lfloor \lg m \rfloor + 1) + m - 2^{\lceil \lg m \rceil}$$
$$= m(\lfloor \lg m \rfloor + 1) + 2m - 2^{\lceil \lg m \rceil} > m \lfloor \lg m \rfloor$$

این نامساوی رخ می‌دهد زیرا در حالت کلی $2^{\lceil \lg m \rceil} < 2m$ است و این اثبات ما را کامل می‌کند.

اکنون که یک حد پائین برای $\min EPL(m)$ داریم می‌توانیم به هدف اصلی خود دست یابیم.

قضیه ۷-۴ هر الگوریتم قطعی که تنها با مقایسه کلیدها، n کلید مجزا را مرتب می‌کند بایستی در حالت میانی، حداقل $\lfloor n \lg n - 1/45n \rfloor$ مقایسه را انجام دهد.

اثبات: بر اساس پیش‌قضیه ۷-۶، این الگوریتم‌ها بایستی در حالت میانی $\min EPL(n!)/n!$ مقایسه را انجام دهند. بر اساس پیش‌قضیه ۷-۱۰، این عبارت بزرگتر یا مساوی

$$\frac{n! \lfloor \lg (n!) \rfloor}{n!} = \lfloor \lg (n!) \rfloor$$

می‌باشد. با استفاده از پیش‌قضیه ۷-۴ می‌توان اثبات قضیه را کامل نمود.

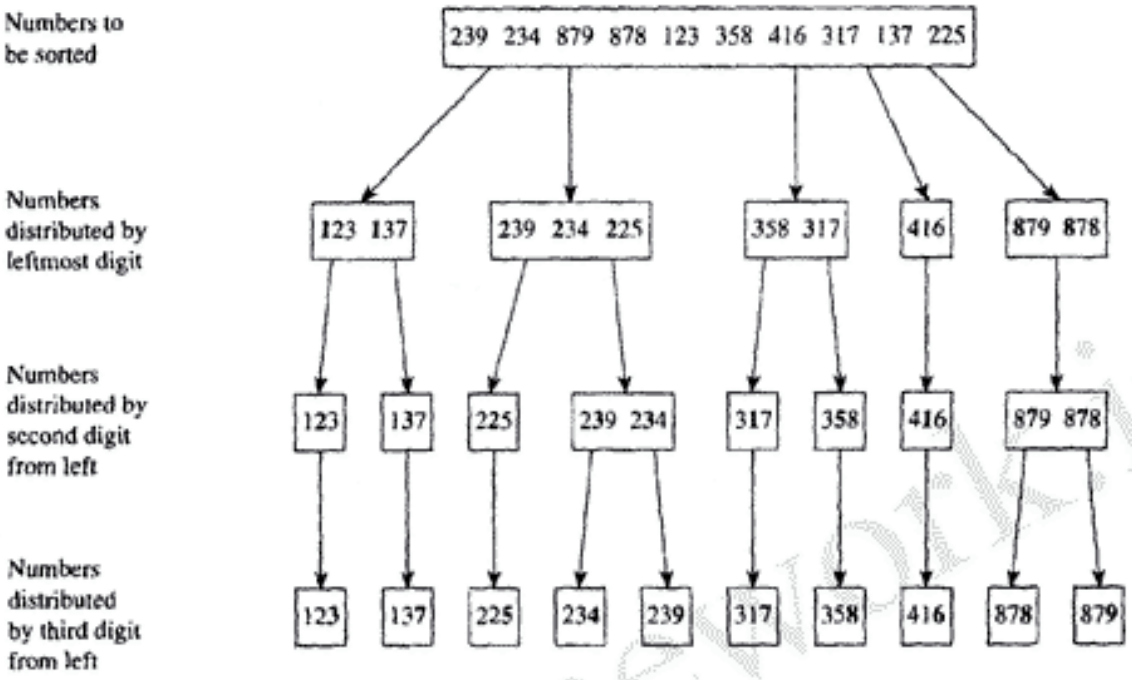
مشاهده می‌کنیم که کارایی حالت میانی مرتب‌سازی ادغامی در حدود $n \lg n - 1/26n$ است که این کارایی برای الگوریتم‌هایی که تنها با مقایسهٔ کلیدها عمل مرتب‌سازی را انجام می‌دهند، نزدیک به حالت بهینه است.

۷-۹ مرتب‌سازی توزیعی (مرتب‌سازی پایه‌ای)

در بخش قبل نشان دادیم که هر الگوریتمی که تنها با مقایسهٔ کلیدها عمل مرتب‌سازی را انجام می‌دهد، نمی‌تواند بهتر از $\theta(n \lg n)$ باشد. اگر ما چیزی در مورد کلیدها ندانیم جز اینکه آنها از یک مجموعهٔ مرتب آمده باشند، هیچ انتخاب خاصی برای مرتب‌سازی به وسیلهٔ مقایسهٔ کلیدها نداریم. اما اگر از نوع کلیدها آگاهی بیشتری داشته باشیم، می‌توانیم الگوریتم‌های مرتب‌سازی دیگری را نیز در نظر بگیریم. در ادامه، می‌خواهیم با استفاده از اطلاعات اضافی در مورد کلیدها، چنین الگوریتمی را ارائه نمائیم.

فرض کنید که می‌دانیم تمامی کلیدها، اعداد صحیح غیرمنفی در مبنای ۱۰ هستند. با فرض اینکه این کلیدها تعداد رقمهای یکسانی دارند، می‌توانیم در ابتدا، آنها را بر اساس مقادیر چپ‌ترین رقمها به گروههایی مجزا توزیع کنیم؛ بدینصورت که کلیدهایی که چپ‌ترین رقم مشابه دارند، در یک گروه قرار داده شوند. آنگاه هر گروه می‌تواند بر اساس مقادیر دومین رقم سمت چپ، به گروههایی توزیع شود. گروههای جدید می‌توانند بر اساس مقادیر سومین رقم سمت چپ به گروههایی توزیع شود و الی آخر. بعد از اینکه تمامی رقمها بررسی گردید، کلیدها مرتب خواهند شد. این روال به "مرتب‌سازی توزیعی" موسوم است. شکل ۷-۱۴، این روال را نشان می‌دهد.

شکل ۱۴-۷ مرتب‌سازی توزیعی، وقتی که بررسی رقمها از چپ به راست صورت می‌گیرد.

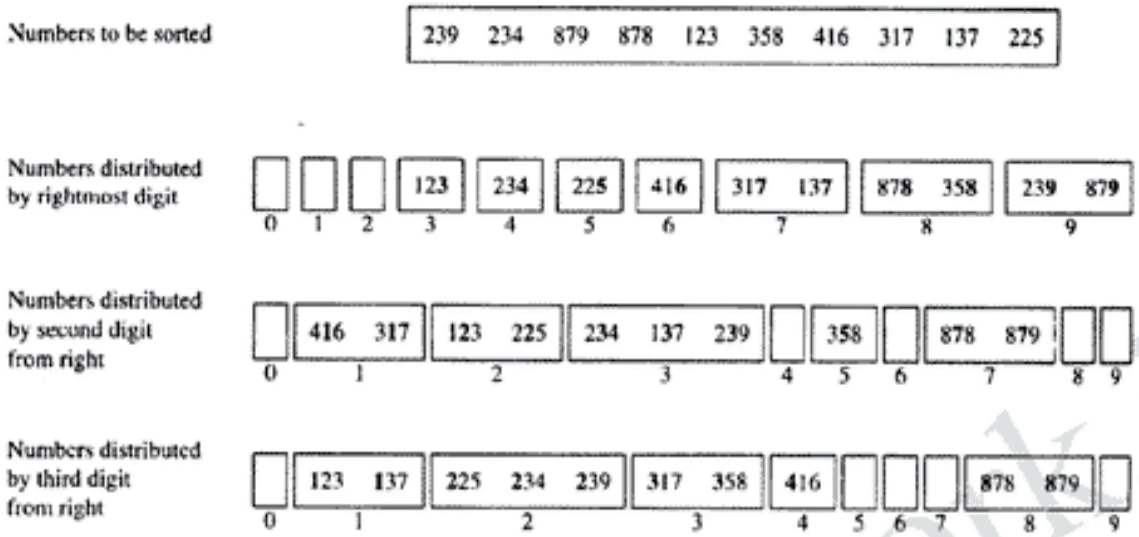


یک مشکل اینجاست که در این روال به تعداد متغیری از گروهها نیازمندیم. فرض کنید به جای اینکه دقیقاً یک گروه را برای هر رقم دهدهی اختصاص دهیم، رقمها را از راست به چپ بررسی کرده و همیشه یک کلید را در گروه مربوط به رقم بررسی شده جایگزین کنیم. اگر این کار انجام شود، در نهایت کلیدهایی مرتب خواهیم داشت. البته با رعایت این نکته که اگر در گذری باید دو کلید در یک گروه قرار گیرند، آنگاه کلیدی که از چپ‌ترین گروه آمده است (در گذر قبلی) در سمت چپ کلید دیگر قرار بگیرد. این روال در شکل ۱۵-۷ نشان داده شده است. به عنوان مثال، بعد از گذر اول کلید ۴۱۶ در یک گروه در سمت چپ کلید ۳۱۷ قرار دارد. بنابراین، بعد از گذر دوم، وقتی که این دو کلید در یک گروه جایگزین می‌شوند، کلید ۴۱۶ در سمت چپ کلید ۳۱۷ قرار می‌گیرد. در سومین گذر، کلید ۴۱۶ درست راست کلید ۳۱۷ واقع می‌شود چون کلید ۴۱۶ در گروه چهارم جایگزین می‌شود.

این روش مرتب‌سازی، از کامپیوترهایی که روش قدیمی مرتب‌سازی کارتی را انجام می‌دادند، الگوبرداری شده است. به این روش، **مرتب‌سازی پایه‌ای (Radix Sort)** گوئیم زیرا اطلاعات بکار رفته برای مرتب‌سازی کلیدها، پایهٔ (مبنا) کلیدها است. این پایه می‌تواند رقمی یا الفبایی باشد. تعداد گروهها، دقیقاً به پایه انتخابی بستگی دارد. برای مثال، اگر در حال مرتب‌سازی اعداد هگزا دسیمال باشیم، تعداد گروهها برابر است با ۱۶ و اگر در حال مرتب‌سازی کلیدهای حرفی بر اساس الفبای انگلیسی باشیم، تعداد گروهها برابر است با ۲۶.

از آنجائیکه تعداد کلیدها در یک گروه خاص در هر گذر تغییر می‌کند، لذا یک روش خوب در پیاده‌سازی الگوریتم‌ها، استفاده از لیستهای پیوندی است. هر گروه با یک لیست‌پیوندی نشان داده می‌شود.

شکل ۱۵-۷ مرتب‌سازی توزیعی، وقتی که بررسی رقمها از راست به چپ صورت می‌گیرد.



بعد از هر گذر، کلیدها با پیوستن به یک لیست پیوندی بالاتر از لیست‌ها (گروهها) حذف شده و بر اساس لیستهای حذف شده، در لیست جدید مرتب می‌شوند. در گذر بعد، لیست بالاتر از ابتدا پیمایش می‌شود و هر کلید در انتهای لیست (گروه) جایگزین می‌شود تا جائی که متعلق به آن گذر باشد. در این روش، قانون فوق رعایت شده است. برای خوانایی بیشتر الگوریتم، کلیدها را اعدادی صحیح و غیرمنفی در مبنای ۱۰ فرض نمودیم. می‌توانیم الگوریتم را به گونه‌ای تغییر دهیم که بدون تأثیر در پیچیدگی زمانی، در هر مبنایی عمل کند. به اعلان زیر در الگوریتم نیاز داریم:

```
struct nodetype;
{
   keytype key;
   nodetype* link;
};
typedef nodetype* node_Pointer;
```

الگوریتم ۶-۷ مرتب‌سازی پایه‌ای (Radix Sort)

مسئله: n عدد صحیح غیرمنفی در مبنای ۱۰ را به ترتیب غیرنزولی مرتب نمائید.

ورودی: لیست پیوندی masterlist شامل n عدد صحیح غیرمنفی، و یک عدد صحیح numdigits که ماکزیمم تعداد ارقام دهدهی در هر عدد صحیح می‌باشد.

خروجی: لیست پیوندی masterlist شامل اعداد صحیح به ترتیب غیرنزولی.

```
void radixsort (node_pointer& masterlist,
                int numdigits)
{
  index i;
  node_pointer list[0..9];
  for (i = 1; i <= numdigits; i++) {
    distribute(i);
    coalesce;
  }
}

void distribute (index i)                          // i is index of current
{                                                  // digit being inspected
  index j;
  node_pointer p;
  for (j = 0; j <= 9; j++)                         // Empty current piles.
    list[j] = NULL;
  p = masterlist;                                  // Traverse masterlist.
  while (p != NULL) {
    j = value of ith digit (from the right) in p-> key;
    link p to the end of list[j];
    p = p-> link;
  }
}

void coalesce ( )
{
  index j;
  masterlist = NULL;                               // Empty masterlist.
  for (j=O; j <= 9;j++)
    link the nodes in list [j] to the end of masterlist;
}
```

تحلیل پیچیدگی زمانی حالت معمول الگوریتم ۶-۷ (Radix Sort)

عمل مبنایی: از آنجائیکه هیچ عمل مقایسه‌ای در این الگوریتم وجود ندارد، لذا باید یک عمل مبنایی متفاوت پیدا کنیم. به منظور کارایی بیشتر روال Coalesce، لیستهایی که شامل گره‌ها هستند اشاره‌گرهایی دارند که به ابتدا و انتهای لیست‌ها اشاره می‌کنند بطوری که به راحتی و بدون پیمایش لیست می‌توان لیستی را به انتهای لیست اصلی اضافه نمود. بنابراین، در هر گذر از حلقه for در این روال، با انتساب سادهٔ یک آدرس به یک متغیر اشاره‌گر می‌توان لیستی را به انتهای masterlist اضافه نمود و در نتیجه می‌توانیم این عمل انتساب را به عنوان عمل مبنایی در نظر بگیریم. در روال distribute می‌توانیم هر یک از دستورالعملهای حلقه While یا تمامی آنها را به عنوان عمل مبنایی در نظر بگیریم. بنابراین، برای ایجاد هماهنگی با Coalesce دستورالعملی را انتخاب می‌کنیم که با انتساب یک آدرس به یک متغیر اشاره‌گر،

یک کلید را به انتهای یک لیست اضافه می‌کند.

اندازه ورودی: n، تعداد اعداد صحیح در masterlist و numdigits، ماکزیمم تعداد ارقام دهدهی در هر عدد صحیح.

پیمایش کامل masterlist همواره به n گذر از حلقه while در distribute نیازمند است. افزودن تمامی لیستها به masterlist همواره به ۱۰ گذر از حلقه for در coalesce نیازمند است. هر یک از این روالها به تعداد numdigits از radix sort فراخوانی می‌شوند. بنابراین،

$$T(n) = numdigits\,(n + ۱۰) \in \theta\,(numdigits\,, n)$$

این الگوریتم، یک الگوریتم $\theta(n)$ نیست زیرا حد مشخص شده، در عناصر numdigits و n می‌باشد. ما می‌توانیم با تولید مقدار بزرگ و اختیاری numdigits، یک پیچیدگی زمانی بزرگ و اختیاری را روی n ایجاد نمائیم. به عنوان مثال، برای مرتب‌سازی ده عدد که بزرگترین آنها عدد ۱،۰۰۰،۰۰۰،۰۰۰ (شامل ده رقم) است به زمانی معادل $\theta(n^{۲})$ نیاز داریم. در عمل، تعداد رقمها خیلی کوچکتر از تعداد اعداد است. برای مثال، اگر در حال مرتب‌سازی ۱،۰۰۰،۰۰۰ شمارهٔ امنیت ملی باشیم، n برابر ۱،۰۰۰،۰۰۰ است؛ درحالیکه numdigits، معادل ۹ می‌باشد. به راحتی می‌توان ثابت نمود هنگامی که کلیدها از هم مجزا هستند، پیچیدگی زمانی بهترین حالت مرتب‌سازی پایه‌ای در $\theta(n \lg n)$است و ما معمولاً به بهترین حالت آن دسترسی داریم.

تحلیل فضای اضافی مورد استفاده توسط الگوریتم ۶-۷ (Radix Sort)

تا به حال گره جدیدی را به الگوریتم اختصاص نداده‌ایم. زیرا یک کلید هرگز به طور همزمان در masterlist و لیست اشاره‌گر به یک گروه وجود ندارد. این بدین معناست که تنها فضای اضافی، فضای لازم جهت ارائه کلیدها در یک لیست پیوندی در اولین مکان است. لذا فضای اضافی مورد استفاده معادل تعداد پیوندها یعنی $\theta(n)$ می‌باشد.

تمرینات

بخش‌های ۱-۷ و ۲-۷

۱- الگوریتم مرتب‌سازی درجی (الگوریتم ۱-۷) را روی کامپیوتر خود اجرا نموده و با استفاده از چند نمونه مسئله، پیچیدگی‌های زمانی بدترین حالت، حالت میانی و بهترین حالت آن را بررسی نمائید.

۲- نشان دهید که ماکزیمم تعداد مقایسات انجام شده به وسیلهٔ مرتب‌سازی درجی (الگوریتم ۱-۷) زمانی حاصل می‌شود که کلیدها به صورت غیرصعودی مرتب شده باشند.

۳- نشان دهید که پیچیدگی‌های زمانی بدترین حالت و حالت میانی برای تعداد انتسابهای انجام شده توسط الگوریتم مرتب‌سازی درجی (الگوریتم ۱-۷) برابر است با

$$W(n)=\frac{(n+۴)(n-۱)}{۲}\approx\frac{n^۲}{۲} \quad , \quad A(n)=\frac{n(n+۷)}{۴}-۱\approx\frac{n^۲}{۴}$$

۴- نشان دهید که پیچیدگی‌های زمانی بهترین حالت و حالت میانی برای تعداد انتسابهای انجام شده توسط الگوریتم مرتب‌سازی تبادلی (الگوریتم ۳-۱) برابر است با

$$W(n)=\frac{۳n(n-۱)}{۲} \quad , \quad A(n)=\frac{۳n(n-۱)}{۴}$$

۵- پیچیدگی‌های زمانی بهترین حالت مرتب‌سازی تبادلی (الگوریتم ۳-۱) و مرتب‌سازی درجی (الگوریتم ۱-۷) را با هم مقایسه کنید.

۶- کدامیک از الگوریتم‌های مرتب‌سازی تبادلی (الگوریتم ۳-۱) و مرتب‌سازی درجی (۱-۷) مناسب‌تر هستند. هرگاه نیاز به پیداکردن ترتیب غیرصعودی K کلید بزرگتر (یا ترتیب غیرنزولی K کلید کوچکتر) در یک لیست n کلید داشته باشیم؟

بخش ۳-۷

۷- نشان دهید که جایگشت $[n, n-۱, \ldots, ۲, ۱]$، به تعداد $n(n-۱)/۲$ وارونگی دارد.

۸- ترانهادهٔ جایگشت [۲،۵،۱،۶،۳،۴] را مشخص نموده و تعداد وارونگی‌ها را در هر دو تعیین نمائید.

۹- نشان دهید که مجموع تعداد وارونگی‌ها در یک جایگشت و ترانهادهٔ آن برابر $n(n-۱)/۲$ است.

بخش ۴-۷

۱۰- الگوریتم‌های مرتب‌سازی معرفی شده در بخشهای ۲-۲ و ۴-۷ را روی کامپیوتر خود اجرا نموده و کارایی بهترین حالت، حالت میانی و بدترین حالت آنها را با استفاده از چند نمونه مسئله بررسی کنید.

۱۱- نشان دهید که پیچیدگی زمانی تعداد انتساب رکوردها برای الگوریتم مرتب‌سازی ادغامی (الگوریتم‌های ۲-۲ و ۴-۲) تقریباً برابر است با $t(n)=۲n\ lgn$

۱۲- یک الگوریتم زمان-خطی درجا بنویسید که لیست پیوندی بدست آمده از الگوریتم مرتب‌سازی ادغامی ۴ (الگوریتم ۴-۷) را به عنوان ورودی پذیرفته و رکوردها را در یک آرایه، به ترتیب غیرنزولی ذخیره نماید.

۱۳- با استفاده از روش تقسیم و غلبه، یک الگوریتم مرتب‌سازی ادغامی غیربازگشتی بنویسید. الگوریتم را تحلیل نموده و نتایج را با استفاده از سمبلهای ترتیب نشان دهید.

بخش ۵-۷

۱۴- نشان دهید که پیچیدگی زمانی میانگین تعداد تبادلات (جابجایی‌ها) انجام شده توسط الگوریتم QuickSort تقریباً برابر است با $۰/۶۹\ (n+۱)\ lgn$ ۰.

۱۵- یک الگوریتم QuickSort غیربازگشتی بنویسید. الگوریتم را تحلیل نموده و نتایج را با استفاده از سمبلهای ترتیب نشان دهید.

۱۶- در زیر نسخه‌ای با سرعت اجرایی بالاتر از روال partition، موسوم به روال QuickSort آمده است:

```
void partition (index low, index high, index& pivotpoint)
{
  index i, j;
  keytype pivotitem;
  pivotitem = S[low];
  i = low;
  j = high;
  while (i < j){
    exchange S[i] and S[j];
    while (S[i] < pivotitem)
      i++;
    while (S[j] < pivotitem)
      j--;
  }
  pivotpoint = i;
  exchange S[high] and S[pivotpoint];
}
```

نشان دهید که با این روال Partition، پیچیدگی زمانی تعداد انتساب رکوردها برای QuickSort برابر است با $A(n) \approx ۰/۶۹ (n+۱) lg n$. همچنین نشان دهید که پیچیدگی زمانی حالت میانی تعداد مقایسه کلیدها نیز در همین حدود می‌باشد.

بخش ۶-۷

۱۷- الگوریتمی بنویسید که بررسی کند آیا یک درخت دودویی کامل اصلی، یک heap هست یا خیر. الگوریتم را تحلیل نموده و نتایج را با استفاده از سمبلهای ترتیب نشان دهید.

۱۸- نشان دهید که 2^j گره با عمق j برای $j<d$ در یک heap شامل n گره (n توانی از ۲ است) وجود دارد. در اینجا، d عمق heap می‌باشد.

۱۹- نشان دهید که یک heap شامل n گره، به تعداد $\lceil n/2 \rceil$ برگ دارد.

۲۰- نشان دهید که پیچیدگی زمانی بدترین حالت تعداد انتساب رکوردها برای heapsort تقریباً برابر است با $W(n) \approx n lg n$

۲۱- الگوریتم Heapsort را به گونه‌ای تغییر دهید که بعد از یافتن Kامین عنصر بزرگتر در ترتیب غیرنزولی متوقف شود. الگوریتم را تحلیل نموده و نتایج را با استفاده از سمبلهای ترتیب نشان دهید.

بخش ۷-۷

۲۲- مزایا و معایب الگوریتم‌های مرتب‌سازی مطرح شده در این فصل را بر اساس مقایسه کلیدها وانتساب رکودها بررسی نمائید.

۲۳- کدام الگوریتمهای مرتب‌سازی انتخابی، درجی، ادغامی، سریع و هرمی را برای هر یک از موقعیتهای زیر در نظر می‌گیرید.

(a) در لیست چند صد رکورد وجود دارد. رکوردها، کاملاً بزرگ ولی کلیدها بسیار کوچک هستند.

(b) لیست در حدود ۴۵،۰۰۰ دارد. به دلایلی مرتب‌سازی باید در همه حالات خیلی سریع انجام شود. حافظه نسبتاً کافی برای ۴۵،۰۰۰ رکورد وجود دارد.

(c) لیست در حدود ۲۵،۰۰۰ رکورد دارد. مرتب‌سازی باید در حالت میانی، به سرعت اجرا شود، اما لزومی ندارد که مرتب‌سازی در هر حالت نیز از سرعت بالایی برخوردار باشد.

بخشهای ۸-۷ و ۹-۷

۲۴- یک الگوریتم زمان-خطی بنویسید که لیستی از مقادیر عددی را به ترتیبی که کاربر مشخص می‌کند، مرتب نماید.

۲۵- نشان دهید هنگامی که همه کلیدها مجزا هستند، پیچیدگی زمانی بهترین حالت مرتب‌سازی پایه‌ای (الگوریتم ۶-۷) برابر است با $o(n lgn)$

۲۶- در فرآیند دوباره سازی لیست اصلی، مرتب‌سازی پایه‌ای (الگوریتم ۶-۷) زمان زیادی را صرف بررسی زیرلیستهای خالی می‌نماید، به ویژه زمانی که تعداد گروهها زیاد باشد. آیا می‌توان با ایجاد تغییراتی در الگوریتم از بررسی زیر لیستهای خالی جلوگیری کرد؟

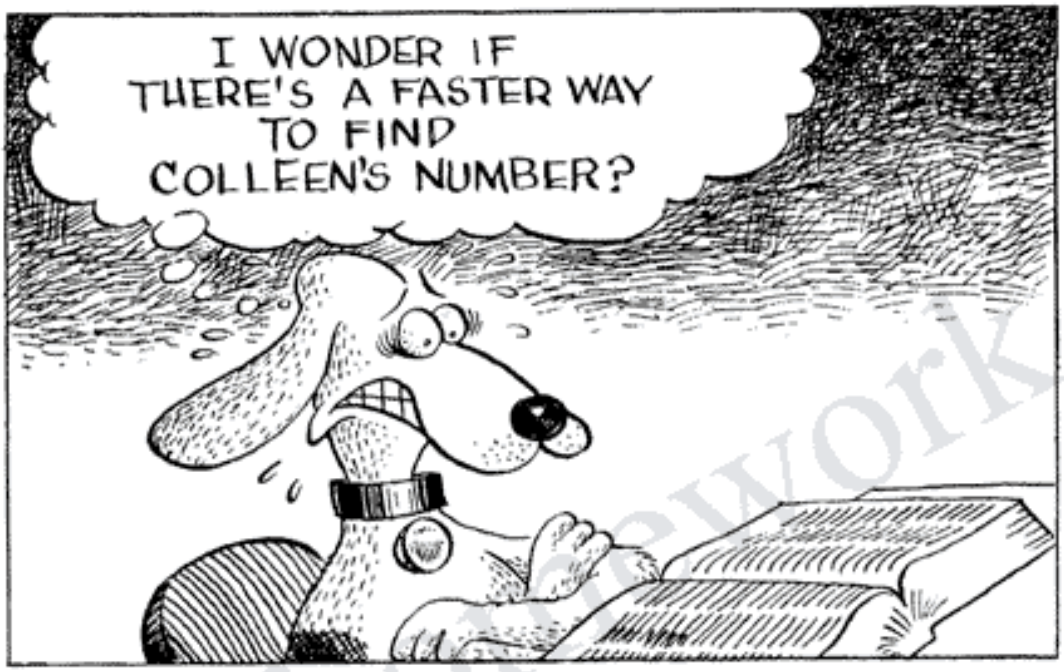
تمرینات اضافی

۲۷- ایدهٔ طراحی یک الگوریتم مرتب‌سازی روی heap سه تایی (ternary heap) را بررسی نمائید. heap سه تایی، دقیقاً مشابه heap معمولی است با این تفاوت که هر گرهٔ داخلی سه فرزند دارد.

۲۸- فرض کنید که یک لیست خیلی بزرگ در حافظه خارجی ذخیره شده است که بایستی مرتب شود. با فرض اینکه این لیست برای حافظه داخلی بسیار بزرگ است، تعیین کنید چه فاکتورهایی باید به هنگام طراحی یک الگوریتم مرتب‌سازی داخلی در نظر گرفته شود؟

فصل ۸

پیچیدگی محاسباتی تکمیلی : مسئله جستجو



از آغاز فصل ۱ به خاطر دارید که بارنی بیگل توانست شماره تلفن کالین کالی را با استفاده از جستجوی دودویی تغییریافته، به سرعت پیدا کند. اکنون بارنی فکر می‌کند که آیا او می‌توانست با روش سریعتری به شماره تلفن کالین دست یابد یا خیر. ما در ابتدا، مسئله جستجو را مورد بررسی قرار داده و سپس به امکانپذیربودن این روش خواهیم پرداخت.

جستجو همانند مرتب‌سازی، یکی از کاراترین عملیات در علم کامپیوتر است. این مسئله معمولاً یک رکورد را براساس مقدار یک یا چند کلید بازیابی می‌کند. یک رکورد، به عنوان مثال، می‌تواند شامل اطلاعات لازم در مورد یک شخص، و کلید می‌تواند کد شناسایی ملی او باشد. هدف ما در اینجا، بررسی مسئله جستجو و دستیابی به الگوریتم‌های جستجویی است که پیچیدگی زمانی آنها تقریباً به خوبی حدّ پائین پیچیدگی زمانی الگوریتم‌های موجود باشد. همچنین می‌خواهیم در مورد ساختارهای داده‌ای مورد استفاده در این الگوریتم‌ها و زمان بکارگیری هر یک از آنها بحث کنیم.

در بخش ۱-۸، همانند مرتب‌سازی در فصل قبل، حدود پائینی را برای جستجوی یک کلید در یک آرایه که فقط با مقایسهٔ کلیدها انجام می‌شود، ارائه نموده و نشان می‌دهیم که پیچیدگی زمانی جستجوی

دودویی (الگوریتم‌های ۵-۱ و ۱-۲) به خوبی همان محدوده‌ها است. در مسئله جستجوی شماره تلفن، بارنی بیگل از یک جستجوی دودویی تغییر یافته موسوم به "جستجوی تخمینی" استفاده می‌کند؛ بدین معنا که بارنی هنگام جستجوی شماره کالین کالی، از وسط دفترچه تلفن شروع نمی‌کند زیرا او می‌داند اسامی که با حرف C شروع می‌شوند جلوتر می‌باشند، لذا او عمل تخمین را به درستی انجام داده و از اوایل دفترچه شروع به جستجو می‌نماید. ما روش جستجوی تخمینی را در بخش ۲-۸ ارائه می‌دهیم. در بخش ۳-۸ نشان می‌دهیم که یک آرایه نمی‌تواند نیازهای برخی از کاربردها را برآورده کند. لذا اگرچه جستجوی دودویی مطلوب است، اما جوابگوی برخی نیازهای خاص نیست. ما نشان می‌دهیم که ساختار داده‌ای درخت می‌تواند این نیازها را برآورده کند و از اینرو درباره جستجوی درختی بحث خواهیم کرد. بخش ۴-۸، مربوط به یک مسئله جستجوی متفاوت یعنی مسئله انتخاب است. این مسئله، kامین کلید کوچکتر یا بزرگتر را در یک لیست nکلیدی جستجو می‌کند. همچنین در این بخش، بحث‌های دیگری را جهت بدست آوردن محدودهٔ کارایی الگوریتم‌ها مطرح خواهیم نمود.

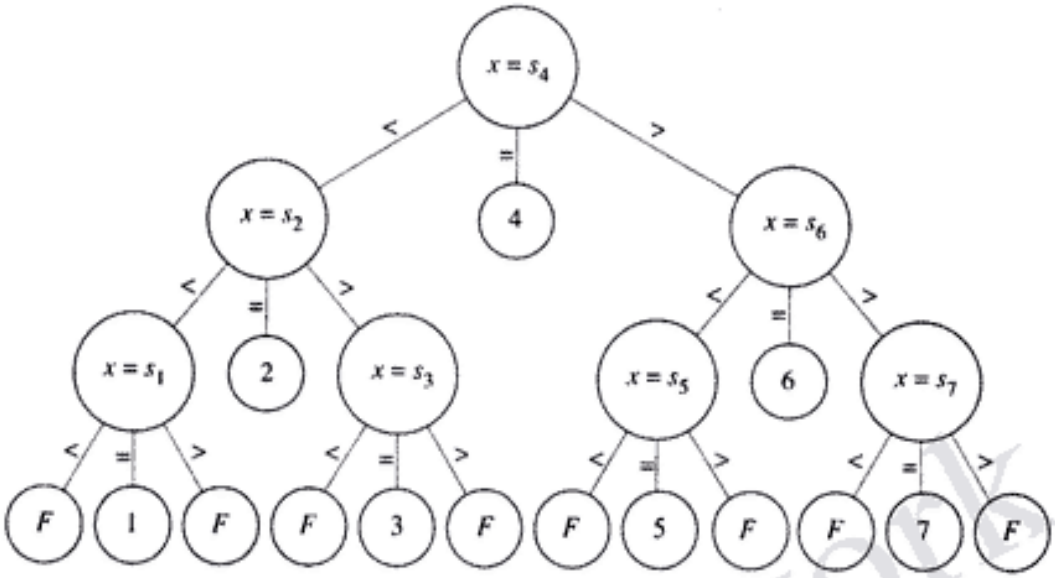
۱-۸ حدود پائین برای جستجوهایی که فقط با مقایسهٔ کلیدها انجام می‌شوند

مسئله جستجو برای یک کلید می‌تواند چنین توصیف شود: یک آرایهٔ S شامل n و یک کلید x موجود است. اندیس i را پیدا کنید طوری که x = S[i] شود اگر x با یکی از کلیدها معادل باشد و درغیراینصورت، عبارت «وجود ندارد» گزارش شود.

جستجوی دودویی (الگوریتم‌های ۵-۱ و ۱-۲) برای حل مسائلی که در آنها آرایه از قبل مرتب شده باشد، بسیار کارا است. به خاطر دارید که پیچیدگی زمانی بدترین حالت این الگوریتم، $\lfloor lgn \rfloor + 1$ است. سؤال اینست که آیا می‌توانیم این کارایی را بهبود ببخشیم؟ خواهیم دید مادامی که خود را به الگوریتم‌هایی که فقط با مقایسهٔ کلیدها جستجو می‌کنند محدود می‌سازیم، چنین پیشرفتی ممکن نخواهد بود. الگوریتم‌هایی که یک کلید x را در یک آرایه، فقط با مقایسه کلیدها جستجو می‌کنند، می‌توانند کلیدها را با یکدیگر یا با کلید مورد جستجوی x مقایسه کرده، همچنین می‌توانند کلیدها را کپی نمایند؛ اما نمی‌توانند عمل دیگری را روی آنها انجام دهند. برای کمک به اینگونه الگوریتم‌ها، آنها می‌توانند از آرایه‌هایی مرتب شده برای جستجو استفاده کنند. همانطوریکه در فصل ۷ مشاهده شد، ما می‌توانستیم حدودی را برای الگوریتم‌های قطعی بدست آوریم. نتایج حاصله برای الگوریتم‌های احتمالی (غیرقطعی) نیز به همان صورت خواهد بود. همانند فصل ۷، فرض می‌کنیم که کلیدها از هم مجزا هستند.

همانطوریکه برای الگوریتم‌های مرتب‌سازی قطعی عمل نمودیم، می‌توانیم با هر الگوریتم قطعی که کلید x را در یک آرایهٔ nکلیدی جستجو می‌کند، یک درخت تصمیم را مرتبط سازیم. شکل ۱-۸، یک درخت تصمیم متناظر با جستجوی دودویی (به هنگام جستجوی هفت کلید) و شکل ۲-۸، یک درخت

شکل ۱-۸ درخت تصمیم متناظر با جستجوی دودویی به هنگام جستجوی هفت کلید.



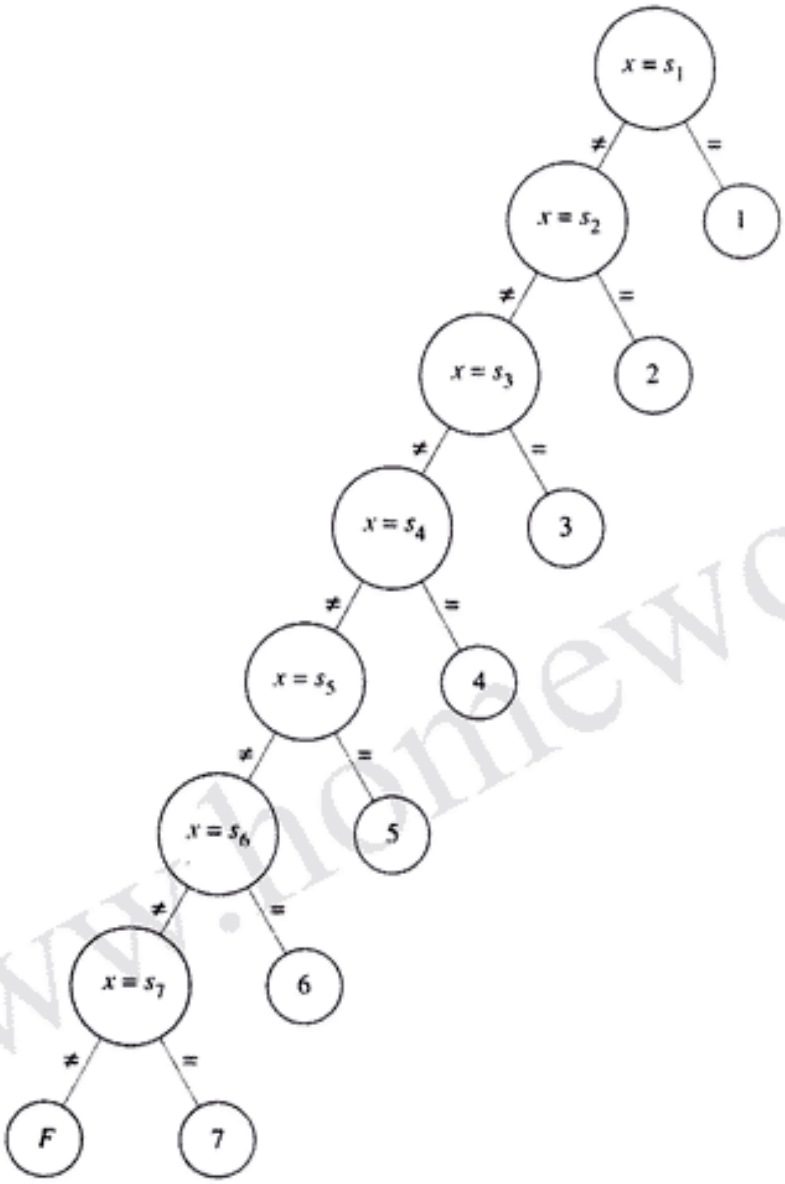
تصمیم متناظر با جستجوی ترتیبی (الگوریتم ۱-۱) را نشان می‌دهد. در این درخت‌ها، هر گرهٔ بزرگ معرف مقایسه یک عنصر آرایه با کلید جستجوی x بوده و هر گرهٔ کوچک، شامل نتیجه‌ای است که باید گزارش شود. زمانی که x در این آرایه وجود دارد، اندیس عنصر x در آرایه را گزارش می‌کنیم و زمانی که x در آرایه وجود ندارد 'F' را به منزلهٔ "نادرست" گزارش می‌کنیم. در شکل‌های ۱-۸ و ۲-۸، مقادیر S_1 تا S_7 به صورت زیر می‌باشند:

$$S[1]=s_1,\ S[2]=s_2,\ \ldots,\ S[7]=s_7$$

ما فرض می‌کنیم که یک الگوریتم جستجو هرگز مقادیر آرایه را تغییر نمی‌دهد. بنابراین، پس از کامل شدن عمل جستجو نیز مقادیر آرایه به همین صورت باقی می‌مانند.

هر برگ در یک درخت تصمیم جستجوی کلید x در میان n کلید، نمایانگر نقطه‌ای است که الگوریتم در آنجا متوقف شده است که یا یک اندیس i را مشخص می‌کند بطوری که $x=s_i$ است، و یا "عدم وجود x در آرایه" را گزارش می‌کند. هر گرهٔ داخلی معرف یک مقایسه است. یک درخت تصمیم برای جستجوی کلید x در میان n کلید را **معتبر** گوئیم اگر برای هر خروجی ممکن، یک مسیر از ریشه به یک برگ وجود داشته باشد که آن خروجی را گزارش نماید. این بدین معناست که باید برگهایی برای هر حالت $x=s_i$ $(1\le i\le n)$ و همچنین یک برگ برای حالت "عدم وجود x" وجود داشته باشد. یک درخت تصمیم را **هرس شده** گوئیم اگر هر برگ آن قابل دسترسی باشد. هر الگوریتمی که یک کلید x را در یک آرایهٔ n کلیدی جستجو می‌کند، با یک درخت تصمیم معتبر و هرس‌شده متناظر است. در حالت کلی، یک الگوریتم جستجو همیشه به مقایسهٔ x با یک عنصر آرایه نیازی ندارد؛ چرا که آن می‌توانست دو عنصر آرایه را با هم مقایسه کند. به هر حال، از آنجائیکه فرض نمودیم تمامی کلیدها مجا از هم می‌باشند، زمانی

شکل ۲-۸ درخت متناظر با جستجوی ترتیبی به هنگام جستجوی هفت کلید.



خروجی یک تساوی خواهد بود که مقایسهٔ x صورت گرفته باشد. چه در جستجوی دودویی و چه در جستجوی ترتیبی، هنگامی که الگوریتم مشخص می‌کند که x با یک عنصر آرایه برابر است، کار جستجو متوقف شده و اندیس عنصر آرایه بازگردانده می‌شود. برخی الگوریتم‌های غیرکارا، بعد از یافتن عنصر مورد نظر نیز کار مقایسه را ادامه داده و اندیس این عنصر را بعد از پایان کار باز می‌گردانند. به هر حال، ما می‌توانیم به جای این الگوریتم، از الگوریتمی استفاده کنیم که بعد از یافتن عنصر مورد نظر متوقف شده و اندیس آن را بازمی‌گرداند. الگوریتم جدید، حداقل به کارایی الگوریتم اول خواهد رسید. از آنجائیکه سه نتیجهٔ ممکن برای یک مقایسه وجود دارد، لذا حداکثر سه مسیر مختلف نیز بعد از هر مقایسه وجود

خواهد داشت. این بدین معناست که هر گرهٔ مقایسه در درخت تصمیم می‌تواند حداکثر سه فرزند داشته باشد. از آنجائیکه این گره بایستی به سمت برگی هدایت شود که این اندیس را برگرداند، لذا حداکثر دو تا از فرزندان می‌توانند گرهٔ مقایسه باشند. بنابراین، مجموعهٔ گره‌های مقایسه در درخت، یک درخت دودویی را تشکیل می‌دهد. به عنوان نمونه، به مجموعهٔ گره‌های بزرگ در شکل‌های ۱-۸ و ۲-۸ توجه کنید.

۸-۱-۱ حدود پائین برای بدترین حالت

از آنجائیکه هر برگ در یک درخت تصمیم معتبر هرس‌شده بایستی قابل دسترسی باشد، لذا بدترین حالت تعداد مقایسات انجام شده برابر تعداد گره‌ها در طولانی‌ترین مسیر از ریشه به یک برگ در درخت دودویی شامل گره‌های مقایسه است که این تعداد برابر عمق درخت دودویی به اضافه یک می‌باشد. بنابراین، برای ارائه یک حد پائین برای تعداد مقایسات، لازم است که حد پائینی را برای عمق درخت دودویی شامل گره‌های مقایسه مشخص کنیم. به منظور ارائه چنین حدی به پیش‌قضایا و قضیه زیر نیاز داریم:

پیش‌قضیه ۱-۸ اگر n تعداد گره‌ها در یک درخت دودویی و d عمق درخت باشد، آنگاه

$$d \geq \lfloor lg(n) \rfloor$$

اثبات: می‌دانیم که

$$n \leq ۱ + ۲ + ۲^۲ + ۲^۳ + \ldots + ۲^d$$

زیرا درخت، تنها یک ریشه، حداکثر دو گره در عمق ۱، حداکثر $۲^۲$ گره در عمق ۲،... و حداکثر ۲^d گره در عمق d دارد. با استفاده از مثال ۳-A در ضمیمه A داریم:

$$n \leq ۲^{d+۱} - ۱$$

بدین معنا که

$$n < ۲^{d+۱}$$
$$lg\, n < d + ۱$$
$$\lfloor lg\, n \rfloor \leq d + ۱$$

اگرچه پیش‌قضیهٔ بعدی واضح به نظر می‌رسد ولی اثبات آن چندان ساده نیست.

پیش قضیه ۲-۸ درخت تصمیم معتبر و هرس‌شده برای جستجوی کلید x در میان n کلید، یک درخت دودویی شامل گره‌های مقایسه است که حداقل n گره داشته باشد.

اثبات: فرض کنیم که s_i $(۱ \leq i \leq n)$ مقادیر n کلید مجزا باشد. ابتدا نشان می‌دهیم که هر s_i بایستی حداقل در یک گرهٔ مقایسه وجود داشته باشد. فرض می‌کنیم که به ازاء برخی مقادیر i چنین حالتی برقرار نیست. دو ورودی که برای همه کلیدها به جز کلید iام مشخص هستند را درنظر می‌گیریم و فرض می‌کنیم که x

مقدار s_i در یکی از ورودیها باشد. از آنجائیکه s_i در هیچ مقایسه‌ای شرکت نکرده و کلیدهای دیگر، رفتار مشابهی با هر دو ورودی دارند، لذا درخت تصمیم نیز برای هر دو ورودی به یک صورت عمل خواهد کرد. اما می‌بینیم که درخت بایستی i را برای یکی از ورودیها گزارش کند و در عین حال نبایستی i را برای ورودی دیگر گزارش نماید. این تناقض نشان می‌دهد که هر s_i بایستی حداقل در یک گره مقایسه قرار داشته باشند. از آنجائیکه هر s_i بایستی حداقل در یک گرهٔ مقایسه باشد، لذا تنها راهی که ما می‌توانستیم کمتر از n گرهٔ مقایسه داشته باشیم این بود که حداقل یک کلید s_i، فقط در مقایسه با کلیدهای دیگر قرار گیرد و هرگز با x مقایسه نشود. فرض کنید که ما چنین کلیدی داشته باشیم. دو ورودی که در همه جا مساوی هستند به جز s_i، که s_i کوچکترین کلید در هر دو ورودی است را در نظر بگیرید. فرض کنید که x کلید iام در یکی از ورودی‌ها باشد. یک مسیر از یک گرهٔ مقایسه شامل s_i بایستی در یک جهت یکسان برای هر دو ورودی باشد و کلیدهای دیگر هم رفتار مشابهی روی هر دو ورودی داشته باشند. بنابراین، درخت تصمیم نیز بایستی برای هر دو ورودی به یک صورت عمل نماید. اما می‌بینیم که درخت بایستی i را برای یکی از ورودیها گزارش کند و در عین حال نبایستی i را برای ورودی دیگر گزارش نماید. این تناقض ما را به اثبات پیش‌قضیه می‌رساند.

قضیه ۸-۱ هر الگوریتم قطعی که برای جستجوی کلید x در میان n کلید مجزا، فقط با مقایسهٔ کلیدها عمل می‌کند بایستی در بدترین حالت، حداقل $\lfloor lg n \rfloor + 1$ مقایسه را انجام دهد.

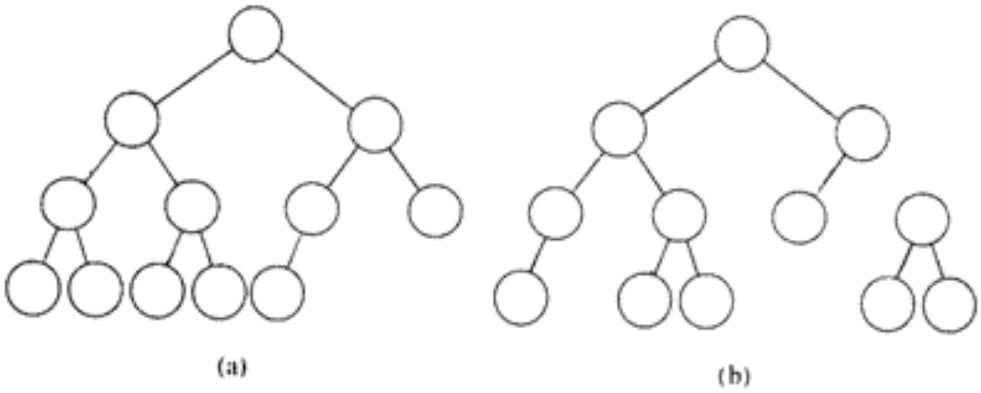
اثبات: با هر الگوریتم، یک درخت تصمیم معتبر هرس‌شده برای جستجوی کلید x در میان n کلید مجزا متناظر است. بدترین حالت تعداد مقایسهٔ کلیدها، تعداد گره‌ها در طولانی‌ترین مسیر از ریشه به یک برگ در درخت جستجوی دودویی شامل گره‌های مقایسه در آن درخت تصمیم است که این تعداد برابر عمق درخت دودویی به اضافهٔ یک می‌باشد. پیش قضیه ۸-۲ بیان می‌کند که این درخت دودویی، حداقل n گره دارد. بنابراین، براساس پیش قضیه ۸-۱، عمق آن بزرگتر یا مساوی $\lfloor lg n \rfloor$ است که این، قضیه ما را ثابت می‌کند.

۸-۱-۲ حدود پائین برای حالت میانی

علاوه بر بحث حدود برای حالت میانی، می‌خواهیم تحلیل حالت میانی را نیز برای جستجوی دودویی انجام دهیم. ما تاکنون برای این تحلیل منتظر مانده‌ایم زیرا استفاده از درخت تصمیم، تحلیل حالت میانی را تسهیل می‌کند. بدین منظور، به یک تعریف و یک پیش‌قضیه نیاز داریم.

یک درخت دودویی را **درخت دودویی کامل نسبی** گوئیم اگر تا عمق d-۱ آن کامل شده باشد. همانطوریکه در شکل ۸-۳ ملاحظه می‌کنید، یک درخت دودویی کامل اصلی می‌تواند یک درخت دودویی نسبی باشد اما عکس آن صادق نیست. (برای تعریف کامل و کامل اصلی به بخش ۶-۷ مراجعه کنید.)

شکل ۳-۸ (a) یک درخت دودویی کامل اصلی (b) یک درخت دودویی کامل نسبی که کامل اصلی نیست.



پیش قضیه ۳-۸ درخت شامل گره‌های مقایسه در درخت تصمیم معتبر هرس‌شدهٔ متناظر با جستجوی دودویی، یک درخت دودویی کامل نسبی است.

اثبات: اثبات به وسیله استقراء n (تعداد کلیدها) انجام می‌شود. روشن است که درخت شامل گره‌های مقایسه، یک درخت دودویی شامل n گره (هر گره برای یک کلید) می‌باشد. بنابراین، می‌توانیم با استقراء روی تعداد گره‌ها در این درخت دودویی نیز این کار را انجام دهیم.

پایهٔ استقراء: یک درخت دودویی شامل یک گره، کامل نسبی است.

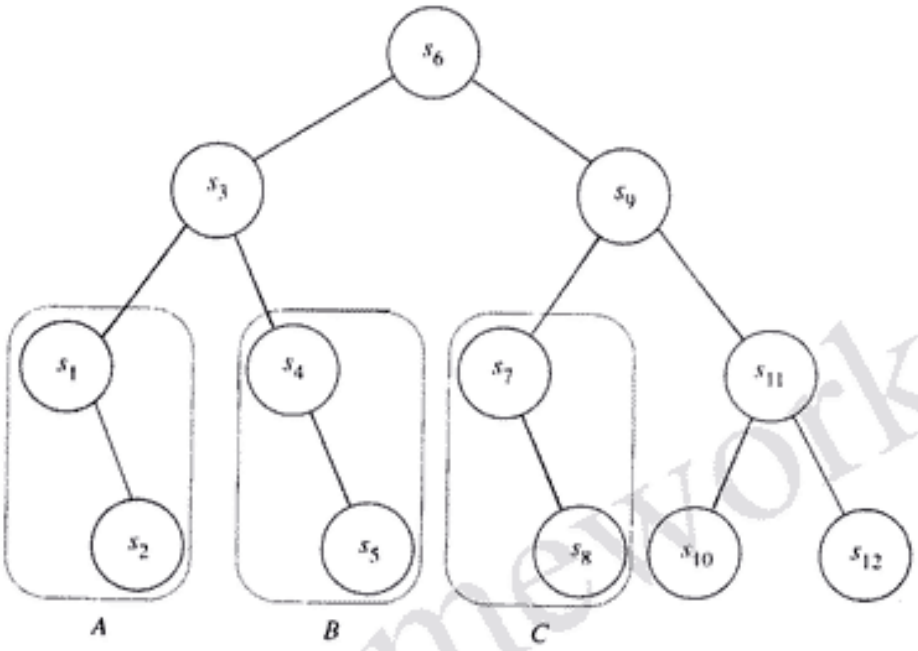
فرض استقراء: فرض کنید که برای هر k<n، درخت دودویی شامل n گره، کامل نسبی است.

گام استقراء: بایستی نشان دهیم که درخت دودویی شامل n گره نیز کامل نسبی است. ما این کار را به طور مجزا برای مقادیر فرد و زوج n انجام می‌دهیم.

اگر n فرد باشد، آنگاه اولین تقسیم در جستجوی دودویی، آرایه را به دو زیرآرایه به اندازهٔ $(n-1)/2$ تقسیم می‌کند. بنابراین، هر دو زیردرخت چپ و راست، درختهای دودویی متناظر با جستجوی دودویی هستند. وقتی که $(n-1)/2$ کلید مورد جستجو باشند. این درختها، با توجه به فرض استقراء، کامل نسبی می‌باشند. از آنجائیکه آنها درخت‌های کامل نسبی مشابهی هستند، لذا درختی که آنها زیردرخت چپ و راست آن هستند نیز کامل نسبی است.

اگر n زوج باشد، آنگاه اولین تقسیم در جستجوی دودویی، درخت را به یک زیرآرایه به اندازه $n/2$ در راست و یک زیرآرایه به اندازهٔ $(n-1)/2$ در چپ تقسیم می‌کند. برای واقعی‌تر شدن بحث حالتی را در نظر گرفتیم که تعداد فرد کلیدها در سمت چپ قرار بگیرد. زیرا هنگامی که تعداد کلیدها در سمت راست فرد است، اثبات پیش‌قضیه مشابه خواهد بود. در اینصورت زیردرختهای چپ و راست زیردرخت چپ، همانند بحث قبلی، درخت مشابهی دارند. یک زیردرخت از زیردرخت راست نیز از همان درخت مشابه است (شما باید آن را تعیین کنید.) از آنجائیکه زیردرخت راست، براساس فرض استقراء، کامل نسبی است و از آنجائیکه یکی از زیردرختهای آن، درخت مشابهی نظیر زیردرختهای چپ و راست زیردرخت چپ دارد، لذا کل درخت نیز بایستی کامل نسبی باشد. به شکل ۴-۸ توجه کنید.

شکل ۴-۸ درخت دودویی شامل گره‌های مقایسه در درخت تصمیم متناظر با جستجوی دودویی هنگامی که n=۱۲ است. فقط مقادیر کلیدها در گره‌ها نشان داده شده‌اند. زیردرختهای A، B و C ساختار مشابهی دارند.



تحلیل پیچیدگی زمانی حالت میانی الگوریتم ۱-۲ (جستجوی دودویی،بازگشتی)

عمل میانی: مقایسهٔ x با s[mid].

اندازه ورودی: n، تعداد عناصر آرایه.

ابتدا حالتی را درنظر می‌گیریم که می‌دانیم x در آرایه هست. با این فرض که x می‌تواند با یک احتمال یکسان در هر یک از اندیسهای آرایه باشد، تحلیل را آغاز می‌کنیم. به مجموع تعداد گره‌ها در مسیر از ریشه به یک گره، **فاصلهٔ گره** و به مجموع فواصل گره به همه گره‌ها، **مجموع فاصلهٔ گره (TND)** گوئیم. توجه داشته باشید که فاصلهٔ یک گره، یک واحد بزرگتر از طول مسیر از ریشه به آن گره است. برای درخت دودویی شامل گره‌های مقایسه در شکل ۱-۸ داریم

$$TND = ۱ + ۲ + ۲ + ۳ + ۳ + ۳ + ۳ = ۱۷$$

مشاهدهٔ این نکته که مجموع تعداد مقایسات انجام شده توسط درخت تصمیم جستجوی دودویی برای جایگزینی x در همه اندیسهای ممکن آرایه برابر TND درخت دودویی شامل گره‌های مقایسه است، کار چندان مشکلی نیست. با فرض اینکه همه اندیسها از احتمال یکسانی برخوردارند، میانگین تعداد مقایسات مورد نیاز برای جایگزینی x برابراست با TND/n. مقداردهی اولیهٔ $n = ۲^k - ۱$، که k یک مقدار صحیح است، را انجام می‌دهیم. پیش قضیه ۳-۸ بیان می‌کند که درخت دودویی شامل گره‌های مقایسه در درخت

تصمیم جستجوی دودویی، یک درخت کامل نسبی است. روشن است که اگر یک درخت دودویی نسبی شامل $3^k - 1$ گره باشد، آنگاه یک درخت دودویی کامل خواهد بود. درخت دودویی در شکل ۱-۸ حالتی را نشان می‌دهد که در آن $k = 3$ است. در یک درخت دودویی کامل، TND بدین صورت تعیین می‌شود که

$$TND = 1 + 2(2) + 3(2^2) + \ldots + k(3^{k-1})$$
$$= \frac{1}{2}[(k-1)3^{k+1} + 2] = (k-1)3^k + 1$$

تساوی اخیر از مثال ۵-A در ضمیمه A بدست آمده است. از آنجائیکه $3^k = n + 1$ است، لذا میانگین تعداد مقایسات برابراست با

$$A(n) = \frac{TND}{n} = \frac{(k-1)(n+1)+1}{n} \approx k - 1 = \lfloor \lg n \rfloor$$

برای n در حالت کلی، میانگین تعداد مقایسات تقریباً به صورت زیر است:

$$\lfloor \lg n \rfloor - 1 \leq A(n) \leq \lfloor \lg n \rfloor$$

این میانگین به حد بالا نزدیکتر است، اگر n توانی از ۲ بوده یا کمی بزرگتر از توانی از ۲ باشد. از آنطرف میانگین به حد پائین نزدیکتر است اگر n کمی کوچکتر از توانی از ۲ باشد. شکل ۱-۸، علت آن را نشان می‌دهد. اگر ما دقیقاً یک گره را به درخت اضافه کنیم بطوری که مجموع تعداد گره‌ها برابر ۸ شود، $\lfloor \lg n \rfloor$ از ۲ به ۳ پرش می‌کند؛ اما میانگین تعداد مقایسات به سختی تغییر می‌کند.

در ادامه، حالتی را درنظر می‌گیریم که ممکن است x در آرایه نباشد. ۲n+۱ امکان مختلف وجود دارد: x می‌توانست کوچکتر از همه عناصر باشد، میان هر دو عنصر باشد، و یا بزرگتر از همه عناصر باشد. بدین ترتیب ما می‌توانستیم داشته باشیم:

$$x = s_i \qquad 1 \leq i \leq n$$
$$x < s_1$$
$$s_i < x < s_{i+1} \qquad 1 \leq i \leq n-1$$
$$x > s_n$$

سپس حالتی را تحلیل می‌کنیم که هر یک از این امکان‌ها از احتمال یکسانی برخوردار باشند. مقداردهی اولیهٔ $n = 3^k - 1$، که k یک مقدار صحیح است، را انجام می‌دهیم. مجموع تعداد مقایسات برای جستجوهای موفق برابر TND درخت دودویی شامل گره‌های مقایسه است. به خاطر دارید که این تعداد معادل $1 + 3^k(k-1)$ است. k مقایسه برای هر یک از n+۱ جستجوی موفق وجود دارد (به شکل ۱-۸ توجه کنید). لذا میانگین تعداد مقایسات برابراست با

$$A(n) = \frac{TND + k(n+1)}{2n+1} = \frac{(k-1)3^k + 1 + k(n+1)}{2n+1}$$

از آنجائیکه $3^k = n + 1$ است، لذا داریم:

$$A(n) = \frac{(k-1)(n+1)+1+k(n+1)}{2n+1}$$
$$= \frac{2k(n+1)+1-(n+1)}{2n+1}$$
$$\approx k-\frac{1}{2} = \lfloor lg\ n \rfloor + 1 - \frac{1}{2} = \lfloor lg\ n \rfloor + \frac{1}{2}$$

برای n در حالت کلی، میانگین تعداد مقایسات تقریباً بصورت زیر است:

$$\lfloor lg\ n \rfloor - \frac{1}{2} \le A(n) \le \lfloor lg\ n \rfloor + \frac{1}{2}$$

میانگین به حد بالاتر نزدیکتر است اگر n توانی از ۲ بوده یا کمی بزرگتر از توانی از ۲ باشد. از طرف دیگر، میانگین به حد پائین نزدیکتر است اگر n کمی کوچکتر از توانی از ۲ باشد.

کارایی حالت میانی جستجوی دودویی خیلی بهتر از بدترین حالت آن نیست. ما می‌توانیم این مطلب در شکل ۱-۸ مشاهده کنیم. این نکته، حتی زمانی که جستجوهای ناموفق را نیز کنار می‌گذاریم نیز صحیح است. (توجه کنید که تمامی جستجوهای ناموفق در زیر درخت قرار دارند.)

پیش قضیه ۴-۸ TND یک درخت دودویی شامل n گره معادل minTND(n) است، اگر و فقط اگر یک درخت کامل نسبی باشد.

اثبات: فرض کنید درختی وجود دارد که کامل نسبی نیست. از اینرو بایستی گره‌ای (نه در یکی از دو سطح زیرین درخت) باشد که حداکثر یک فرزند داشته باشد. ما می‌توانیم هر گرهٔ A را از سطح زیرین حذف نموده و آن را به عنوان فرزند آن گرهٔ تک فرزند قرار دهیم. درخت حاصل، یک درخت دودویی شامل n گره است. تعداد گره‌ها در مسیر منتهی به A در درخت جدید، حداقل یک واحد کمتر از تعداد گره‌ها در مسیر منتهی به A در درخت اصلی خواهد بود. تعداد گره‌ها در تمامی مسیرهای منتهی به گره‌های دیگر یکسان خواهد بود. بنابراین، ما یک درخت دودویی شامل n گره ایجاد کردیم که TND آن کوچکتر از TND درخت اصلی است. بدین معنا که درخت اصلی ما، یک TND می‌نیمم ندارد. مشاهده این نکته که TND برای همهٔ درختهای دودویی کامل نسبی شامل n گره مقداری مشابه است، کار چندان مشکلی نیست. بنابراین، چنین درختهایی بایستی TND می‌نیمم داشته باشند.

پیش قضیه ۵-۸ پیچیدگی زمانی حالت میانی جستجوی دودویی برای جستجوی کلید x در میان n کلید برابراست با minTND(n)/n.

اثبات: همانطوریکه در تحلیل حالت میانی جستجوی دودویی بحث شد، پیچیدگی زمانی حالت میانی با تقسیم TND/n معادل است. با استفاده از پیش‌قضایای ۳-۸ و ۴-۸ می‌توانید این پیش‌قضیه را اثبات کنید.

پیش‌قضیه ۸-۶ اگر فرض کنیم که x با احتمال مشابهی برای هر یک از اندیسها در آرایه موجود است، آنگاه پیچیدگی زمانی حالت میانی هر الگوریتم قطعی که کلید x را در آرایهٔ n کلیدی مجزا جستجو می‌کند به حد پائین $minTND(n)/n$ محدود می‌شود.

اثبات: همانطوریکه در پیش‌قضیه ۸-۲ نشان دادیم. هر عنصر آرایه s_i بایستی حداقل یک مرتبه با x در درخت تصمیم متناظر با الگوریتم مقایسه شود. فرض کنید که c_i، تعداد گره‌ها در کوتاهترین مسیر منتهی به گرهٔ شامل مقایسهٔ s_i با x باشد. از آنجائیکه هر کلید از احتمال مشابه $1/n$ برای جستجو برخوردار است، لذا حد پائین برای پیچیدگی زمانی حالت میانی برابراست با

$$c_1(\frac{1}{n}) + c_2(\frac{1}{n}) + \ldots + c_n(\frac{1}{n}) = \frac{\sum_{i=1}^{n} c_i}{n}$$

به عنوان تمرین نشان دهید که این مقدار بزرگتر یا مساوی minTND(n) است.

قضیه ۸-۲ در میان الگوریتم‌های قطعی که کلید x را در آرایهٔ n کلیدی مجزا، فقط با مقایسهٔ کلیدها جستجو می‌کنند، جستجوی دودویی در حالت میانی بهینه است اگر ما فرض کنیم که x در آرایه موجود بوده و همه اندیسهای آرایه از احتمال یکسانی برخوردار باشند. بنابراین، چنین الگوریتمی باید به طور متوسط تقریباً $\lfloor \lg n \rfloor - 1$ مقایسه را انجام دهد.

اثبات: اثبات این قضیه با استفاده از پیش‌قضایای ۸-۵ و ۸-۶ و تحلیل پیچیدگی زمانی حالت میانی جستجوی دودویی انجام می‌شود.

گفته شد که جستجوی دودویی در حالت میانی تحت شرایط خاصی از توزیع احتمالات، بهینه است و برای توزیع‌های احتمالی دیگر، ممکن است چنین نباشد. برای مثال، اگر احتمال وجود x در S[1] برابر ۰/۹۹۹۹ باشد، بهینه‌تر این است که ابتدا x را با S[1] مقایسه کنیم. این بحث به دلیل نادیده انگاشتن برخی موارد چندان کاربردی به نظر می‌رسد. برای آشنایی بیشتر می‌توانید به بخش ۵-۳ مراجعه کنید.

۸-۲ جستجوی درون‌یابی (Interpolation Search)

حدودی را که تاکنون بدست آورده‌ایم، مربوط به الگوریتم‌هایی هستند که تنها براساس مقایسهٔ کلیدها عمل می‌کنند. اگر در جستجوی کلید از اطلاعات دیگری نیز استفاده کنیم، آنگاه می‌توانیم حدود بدست آمده را اصلاح کنیم. به خاطر دارید که بارنی برای پیداکردن شمارهٔ تلفن کالین از وسط دفترچه شروع به جستجو نکرد، زیرا او می‌داند اسامی که با حرف C شروع می‌شوند نزدیک به ابتدای دفترچه هستند.

فرض کنید که ما در حال جستجوی ۱۰ عدد صحیح هستیم و می‌دانیم که اولین عدد صحیح در محدودهٔ ۰ تا ۹، دومین عدد صحیح از ۱۰ تا ۱۹، سومین عدد صحیح از ۲۰ تا ۲۹،...، و دهمین عدد

صحیح از ۹۰ تا ۹۹ قرار دارد. آنگاه اگر کلید xکوچکتر از ۰ یا بزرگتر از ۹۹ باشد، فوراً "عدم وجود کلید" را گزارش می‌کنیم و در غیر اینصورت کلید x را با $S[1 + \lfloor x/10 \rfloor]$ مقایسه می‌کنیم. برای مثال، x=۲۵ را با $S[3] = S[1 + \lfloor 25/10 \rfloor]$ مقایسه می‌کنیم. اگر آن دو مساوی نبودند، "عدم وجود کلید" راگزارش می‌کنیم. الگوریتمی که از این استراتژی استفاده نماید به **جستجوی درون‌یابی** موسوم است. در جستجوی دودویی، low را با ۱ و high را با n مقداردهی می‌کنیم و سپس با استفاده از درون‌یابی خطی، تعیین می‌کنیم که کلید x تقریباً در کدام نقطه جای گرفته است. بدینصورت که

$$mid = low + \lfloor \frac{x - S[low]}{S[high] - s[low]} \times (high - low) \rfloor$$

الگوریتم ۸-۱ جستجوی درون‌یابی

مسئله: تعیین کنید آیا x در آرایهٔ مرتب‌شده S با اندازهٔ n وجود دارد یا خیر.

ورودی: عدد صحیح مثبت n، آرایهٔ مرتب‌شده (به ترتیب غیرنزولی) S با شاخصهایی از ۱ تا n.

خروجی: اندیس i محل x در آرایه؛ صفر، اگر x در آرایه نباشد.

```
void interpsrch (int n,
                 const number S[ ],
                 number x,
                 index& i)
{
  index low, high, mid;
  number denominator;

  low = 1; high = n;
  i = 0;
  if (S[low] ≤ x ≤ S[high])
    while (low <= high && i == 0) {
      denominator = S[high] - S[low];
      if (denominator == 0)
        mid = low;
      else
        mid = low + ⌊((x - S[low]) * (high - low)) / denominator⌋;
      if (x == S[mid])
        i = mid;
      else if (x < S[mid])
        high = mid - 1;
      else
        low = mid + 1;
    }
}
```

فرض کنید که کلیدها با توزیع غیریکنواخت بین S[1] و S[n] قرار گرفته‌اند. به این معنا که احتمال انتخاب تصادفی یک کلید در یک محدودهٔ خاص با احتمال انتخاب آن در محدوده‌ای دیگر با همان طول، یکسان است. در چنین حالتی انتظار داریم که x را تقریباً در اندیسی که توسط جستجوی درون‌یابی تعیین می‌شود پیدا کنیم. با این فرض که کلیدها به طور غیریکنواخت توزیع شده‌اند و کلید x می‌تواند با احتمال مشابه در هر یک از اندیسها باشد، می‌توانیم نشان دهیم که پیچیدگی زمانی حالت میانی جستجوی درون‌یابی برابراست با

$$A(n) \approx lg\,(lg\,(n))$$

اگر مقدار n برابر یک میلیون باشد، آنگاه مقدار $lg\,n$ در حدود ۳۰ و مقدار $lg\,(lg\,n)$ در حدود ۵ خواهد شد.

در ادامه، به بررسی بدترین حالت الگوریتم جستجوی درون‌یابی می‌پردازیم. فرض کنید که ده کلید با مقادیر ۱، ۲، ۳، ۴، ۵، ۶، ۷، ۸، ۹ و ۱۰۰ داریم. اگر کلید مورد جستجوی x، عدد ۱۰ باشد، آنگاه mid مکرراً با low مقداردهی شده و x با تمامی کلیدها مقایسه می‌شود. در بدترین حالت، جستجوی درون‌یابی به یک جستجوی ترتیبی تنزل پیدا می‌کند. توجه داشته باشید که زمانی این اتفاق می‌افتد که متغیر mid مکرراً با low مقداردهی شود. یک جستجوی درون یابی تغییر یافته موسوم به **جستجوی درون‌یابی توانمند**، با ارائه یک متغیر جدید gap، این مشکل را رفع نمود. متغیر gap که همیشه کوچکتر یا مساوی mid-low و high-mid است به صورت زیر مقداردهی می‌شود:

$$gap = \lfloor (high - low + 1)^{1/2} \rfloor$$

و mid با همان فرمول قبلی برای درون‌یابی خطی محاسبه می‌شود. بعد از این محاسبه، مقدار mid احتمالاً با رابطه زیر تغییر می‌یابد:

$$mid = minimum\,(high - gap\,,\, maximum\,(mid\,,\, low + gap))$$

در مثال قبل که x=۱۰ و ده کلید ۱، ۲، ۳، ۴، ۵، ۶، ۷، ۸، ۹ و ۱۰۰ موردنظر بودند، gap با $\lfloor (۱۰ - ۱ + ۱)^{1/2} \rfloor = ۳$ و mid با ۱ مقداردهی شده است و داریم

$$mid = minimum\,(۱۰ - ۳, maximum\,(۱, ۱ + ۳)) = ۴$$

در این روش تضمین می‌کنیم که شاخص مورد استفاده برای مقایسه، حداقل موقعیت‌های gap دور از low و high می‌باشد. هنگامی که جستجوی x در یک زیرآرایه، شامل تعداد بیشتری از عناصر آرایه دنبال می‌شود، مقدار gap دو برابر می‌گردد، اما هرگز بزرگتر از نصف تعداد عناصر آرایه در آن زیرآرایه نمی‌شود.

با این فرض که کلیدها به طور غیر یکنواخت توزیع شده‌اند و کلیدها x می‌تواند با احتمال مشابه در هر یک از اندیسهای آرایه قرار داشته باشد، پیچیدگی زمانی حالت میانی جستجوی درون‌یابی توانمند برابراست با $\theta\,(lg\,(lg\,n))$ و پیچیدگی زمانی بدترین حالت آن برابراست با $\theta((lg\,n)^2)$ که بدتر از جستجوی دودویی و بسیار بهتر از جستجوی درون‌یابی است.

۸-۳ جستجو در درخت‌ها

با وجود اینکه الگوریتم جستجوی درون‌یابی و الگوریتم اصلاح شدهٔ آن بسیار کارا هستند، معذلک نمی‌توانند در کاربردهای متعدد مورد استفاده قرار گیرند زیرا آرایه‌ها، ساختار داده‌ای مناسبی برای مرتب‌سازی داده‌ها در این کاربردها نمی‌باشند. نشان می‌دهیم که به جای آرایه می‌توانیم از ساختار داده‌ای درخت استفاده کنیم. همچنین نشان می‌دهیم که الگوریتم‌هایی ($O(\lg n)$) برای جستجوی درختی وجود دارند.

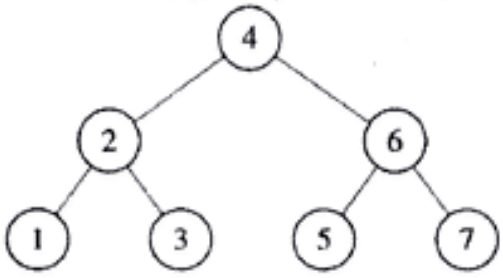
جستجوی ایستا، پردازشی است که در آن تمامی رکوردها در یک زمان به فایل اضافه می‌شوند و هیچ نیازی به جمع یا حذف بعدی رکوردها نیست. یک مثال از جستجوی ایستا را می‌توان در جستجوی انجام شده توسط دستورات سیستم عامل دید. بسیاری از کاربردها به **جستجوی پویا** نیاز دارند؛ پردازشی که در آن رکوردها، مکرراً به فایل اضافه یا از آن حذف شوند. یک سیستم رزواسیون هواپیمایی نمونه‌ای از این نوع جستجو است، زیرا مشتری‌ها مکرراً می‌توانند زمانبند را فراخوانی کرده و از پرواز موردنظر خود انصراف دهند.

ساختار آرایه برای جستجوی پویا بسیار نامناسب است، زیرا هنگامی که ما یک رکورد را به آرایهٔ مرتب‌شده اضافه می‌کنیم بایستی تمامی رکوردهای زیرین آن را جابجا کنیم. جستجوی دودویی به ساختار آرایه نیازمند است چرا که باید روش کارایی برای یافتن عنصر میانی داشته باشد. این بدین معنا است که جستجوی دودویی نمی‌تواند برای جستجوی پویا بکار گرفته شود. اگرچه ما می‌توانیم رکوردها را به راحتی به یک لیست پیوندی اضافه و یا از آن حذف نمائیم، اما هیچ روش کارایی برای جستجوی یک لیست پیوندی وجود ندارد. بنابراین، لیست پیوندی نیز نمی‌تواند نیازهای یک جستجوی پویا را برآورده سازد. جستجوی پویا می‌تواند به صورت کارایی با استفاده از ساختار درخت بکار گرفته شود. ابتدا، دربارهٔ درختهای جستجوی دودویی بحث کرده، سپس به معرفی درختهای B (B-tree) که اصلاح شدهٔ درختهای جستجوی دودویی هستند می‌پردازیم.

۸-۳-۱ درختهای جستجوی دودویی

درختهای جستجوی دودویی در بخش ۵-۳ معرفی شدند. با وجود این، هدف ما بحث در مورد کاربرد آن در جستجوی ایستا است. بدین صورت که می‌خواهیم درخت بهینه‌ای را براساس احتمالات جستجوی کلیدها ایجاد نمائیم. الگوریتم سازندهٔ درخت (الگوریتم ۹-۳) نیاز دارد که تمامی کلیدها در یک زمان اضافه شوند و این بدین معناست که این کاربرد به جستجوی ایستا نیازمند است. درختهای جستجوی دودویی برای جستجوی پویا نیز مناسب هستند. با استفاده از درخت جستجوی دودیی معمولاً می‌توانیم میانگین زمان جستجو را پائین نگه داریم؛ در حالیکه کلیدها را به سرعت اضافه و یا حذف می‌کنیم. به عبارت دیگر، ما می‌توانیم با انجام پیمایش سطحی روی درخت، کلیدها را به یک ترتیب خاص بازیابی کنیم. بخاطر دارید که در یک **پیمایش سطحی (in-order traversal)** روی یک درخت دودویی، ابتدا گره‌های زیردرخت

شکل ۵-۸ یک درخت جستجوی دودویی شامل هفت عدد صحیح ابتدایی.



چپ پیمایش شده، سپس با ملاقات ریشه به پیمایش گره‌های زیردرخت راست می‌پردازیم.

شکل ۵-۸، یک درخت جستجوی دودویی شامل هفت عدد صحیح ابتدایی را نشان می‌دهد. الگوریتم جستجو (الگوریتم ۸-۳) به وسیلهٔ مقایسهٔ کلید x با مقدار موجود در ریشه، جستجو در درخت را انجام می‌دهد. اگر آن دو مساوی باشند، کار انجام شده است. اگر x کوچکتر باشد، استراتژی جستجو برای فرزند چپ، تکرار می‌شود و اگر x بزرگتر باشد، استراتژی جستجو برای فرزند راست بکار می‌رود. این عمل آنقدر ادامه می‌یابد تا x پیدا شود یا عدم وجود x در درخت مشخص شود.

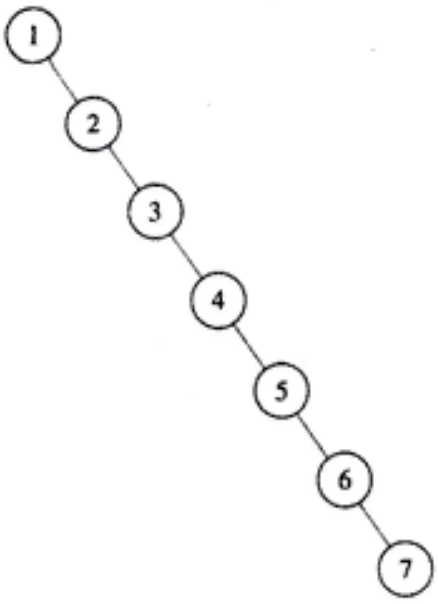
ما می‌توانیم کلیدها را با کارایی بالایی به درخت شکل ۵-۸ اضافه و یا از آن حذف نمائیم. بعنوان مثال، برای اضافه کردن کلید ۵/۵، ابتدا به اولین گره توجه می‌کنیم. اگر ۵/۵ بزرگتر از کلید موجود در گره باشد، به سمت راست و در غیراینصورت به سمت چپ حرکت می‌کنیم تا اینکه برگ شامل کلید ۵ را پیدا کنیم. آنگاه کلید ۵/۵ را به عنوان فرزند راست برگ ۵ اضافه می‌کنیم.

یک نکته مهم در بحث درختهای جستجوی دودویی این است که وقتی کلیدها به صورت پویا اضافه یا حذف می‌شوند، تضمینی وجود ندارد که درخت حاصل متوازن باقی بماند. برای مثال، اگر تمامی کلیدها به ترتیب صعودی حذف شوند، درخت شکل ۶-۸ بدست می‌آید. این درخت که به **درخت انحرافی** موسوم است، یک لیست پیوندی می‌باشد. یکی از کاربردهای الگوریتم ۸-۳ با این درخت در جستجوی ترتیبی است که در آن از یک درخت جستجوی دودویی به جای لیست پیوندی استفاده می‌شود.

اگر کلیدها به طور تصادفی اضافه شوند، به نظر می‌رسد که درخت حاصل بسیار متوازن‌تر از حالت قبل باشد (برای بحث ارقام تصادفی به بحث A-۸-۱ در ضمیمه A مراجعه کنید). بنابراین، در حالت میانی، زمان جستجوی کارایی را انتظار داریم. در واقع، از این نتیجه می‌توانیم به یک قضیه دست یابیم. این قضیه، متوسط زمان جستجو برای ورودیهایی شامل n کلید مجزا با احتمال مشابه را تعیین می‌کند. "ورودیهای با احتمال مشابه" بدین معناست که هر ترتیب ممکنی از n کلید می‌تواند با یک احتمال یکسان به عنوان ورودی الگوریتم درنظر گرفته شود. برای مثال، اگر $n = ۳$ و $s_۱ < s_۲ < s_۳$ سه کلید مجزا باشند، آنگاه ورودیهای زیر از احتمال مشابهی برخوردارند:

$[s_۱, s_۲, s_۳]$ $[s_۱, s_۳, s_۲]$ $[s_۲, s_۱, s_۳]$

شکل ۶-۸ یک درخت جستجوی دودویی انحرافی شامل هفت عدد صحیح ابتدایی.



توجه داشته باشید که گاهی اوقات، دو ورودی می‌توانند درخت مشابهی را ایجاد کنند. برای مثال، ورودیهای $[s_1, s_2, s_3]$ و $[s_2, s_3, s_1]$ از درختی با ریشهٔ s_2، s_1 در چپ و s_3 در راست نتیجه می‌شوند. گاهی اوقات نیز یک درخت به وسیلهٔ تنها یک ورودی تولید می‌شود.

قضیه ۳-۸ با این فرض که همه ورودیها از احتمال مشابهی برخوردارند و کلید جستجوی x می‌تواند با احتمال مشابهی در میان n کلید مجزا باشد، متوسط زمان جستجو برای تمام وردیهای شامل n کلید مجزا با استفاده از درختهای جستجوی دودویی تقریباً برابراست با

$$A(n) \approx 1/38 \lg n$$

اثبات: اثبات قضیه را با این فرض که کلید x در درخت وجود دارد، انجام می‌دهیم. تمام درختهای جستجوی دودویی شامل n کلید مجزا که k امین کلید کوچکتر آنها در ریشه قرار دارد را درنظر می‌گیریم. هر یک از این درختها، $k-1$ گره در زیردرخت چپ و $n-k$ گره در زیردرخت راست خود دارد. متوسط زمان جستجو برای ورودیهایی که این درختها را تولید می‌کنند، با مجموع سه کمیت زیر ارائه می‌شود:

- متوسط زمان جستجو در زیردرخت چپ
- متوسط زمان جستجو در زیردرخت راست
- یک مقایسهٔ در ریشه

متوسط زمان جستجو در زیردرخت چپ از چنین درختهایی برابر $A(k-1)$ و متوسط زمان جستجو در زیردرخت راست برابر $A(n-k)$ است. از آنجائیکه فرض کردیم x با احتمال مشابهی می‌تواند هر یک از کلیدها باشد، لذا احتمال وجود x در زیردرختهای چپ و راست به ترتیب برابراست با

$$\frac{n-k}{n} \quad , \quad \frac{k-1}{n}$$

اگر فرض کنیم که $A(n \mid k)$ به متوسط زمان جستجو روی تمام ورودیهای به اندازهٔ n که درختهای جستجوی دودویی با k امین کلید کوچکتر در ریشه را تولید می‌کند، اشاره می‌نماید، آنگاه خواهیم داشت:

$$A(n \mid k) = A(k-1)\frac{k-1}{n} + A(n-k)\frac{n-k}{n} + 1$$

از آنجائیکه تمامی ورودیها از احتمال مشابهی برخوردارند، لذا هر کلید نیز می‌تواند با احتمال مشابهی به عنوان اولین کلید (کلید ریشه) درنظر گرفته شود. بنابراین، متوسط زمان جستجو روی تمام وردیهای به اندازه n، برای میانگین $A(n \mid k)$ برای می‌باشد. یعنی

$$A(n) = \frac{1}{n}\sum_{k=1}^{n}\left[\frac{k-1}{n}A(k-1) + \frac{n-k}{n}A(n-k) + 1\right]$$

اگر C(n) را با A(n) مقداردهی کرده و به جای A(n) در رابطه $C(n)/n$ را قرار دهیم، آنگاه

$$\frac{C(n)}{n} = \frac{1}{n}\sum_{k=1}^{n}\left[\frac{k-1}{n}\,\frac{C(k-1)}{k-1} + \frac{n-k}{n}\,\frac{C(n-k)}{n-k} + 1\right]$$

که به صورت زیر ساده می‌شود:

$$C(n) = \sum_{k=1}^{n}\left[\frac{C(k-1)}{n} + \frac{C(n-k)}{n} + 1\right]$$

$$= \sum_{k=1}^{n}\frac{1}{n}\left[C(k-1) + C(n-k)\right] + n$$

وضعیت ابتدایی به صورت زیر است:

$$C(1) = 1A(1) = 1$$

این معادلهٔ بازگشتی، تقریباً مشابه حالت میانی Quicksort (الگوریتم ۶-۲) است. در تحلیل حالت میانی Quicksort داشتیم:

$$C(n) \approx 1/38(n+1)\lg n$$

بنابراین،

$$A(n) \approx 1/38 \lg n$$

توجه داشته باشید که این قضیه بدین معنا نیست که متوسط زمان جستجو برای یک ورودی خاص شامل n کلید مجزا در حدود $1/38 \lg n$ است. یک ورودی خاص، درختی نظیر شکل ۶-۸ را تولید می‌کند که به وضوح متوسط زمان جستجوی آن در $\theta(n)$ خواهد بود. قضیه ۳-۸، متوسط زمان جستجو برای تمام ورودیهای شامل n کلید مجزا را تعیین می‌کند.

۸-۳-۲ درختهای-B

در بسیاری از کاربردها، کارایی ممکن است تا حد زیادی به یک زمان جستجوی خطی تنزل پیدا کند. به عنوان مثال، کلیدهای اشاره‌گر به رکوردها در یک بانک اطلاعاتی بزرگ، اغلب نمی‌توانند همگی در یک زمان در حافظه‌ای با سرعت بالا (RAM) جای بگیرند. بنابراین، برای انجام جستجو به دستیابی چنددیسکی نیاز داریم. (به چنین جستجویی، **جستجوی خارجی** گوئیم؛ در حالیکه اگر تمامی کلیدها به طور همزمان در حافظه باشند، **جستجوی داخلی** انجام می‌شود.) از آنجائیکه دستیابی دیسکی، با حرکت مکانیکی هدهای خواندن/نوشتن و دستیابی RAM با انتقال داده‌های الکترونیکی سروکار دارد، لذا دستیابی دیسکی، کندتر از دستیابی RAM عمل می‌کند. یک روش برای رفع این مشکل، نوشتن برنامه‌ای است که درخت جستجوی دودویی موجود را به عنوان ورودی گرفته و یک درخت جستجوی دودویی متوازن شامل همان کلیدهای درخت اصلی را به عنوان خروجی ارائه می‌نماید. این برنامه به صورت دوره‌ای اجرا می‌شود. الگوریتم ۹-۳، یک الگوریتم برای این برنامه است که بسیار قویتر از یک الگوریتم متوازن‌ساز ساده می‌باشد، زیرا قادر است احتمال هر کلید را به عنوان کلید جستجو درنظر بگیرد.

در سال ۱۹۷۲، Bayer و E.McCreight، اصلاحی را روی درخت‌های جستجوی دودویی به نام **درختهای-B** انجام دادند. هنگامی که کلیدها به یک، درخت-B اضافه و یا از آن حذف می‌شوند، همهٔ برگها در همان سطح باقی می‌مانند که این امر حتی بهتر از مسئله متوازن‌سازی درخت‌ها است. الگوریتم‌های ارائه شده برای درخت‌های-B را می‌توانید در کتاب kruse (۱۹۹۴) پیدا کنید. در اینجا فقط به تشریح چگونگی حفظ کلیدها در یک سطح، به هنگام اضافه کردن کلیدها می‌پردازیم.

یکی از ساده‌ترین انواع درخت-B، درخت ۲-۳ است. ویژگیهای این درخت به شرح ذیل است:

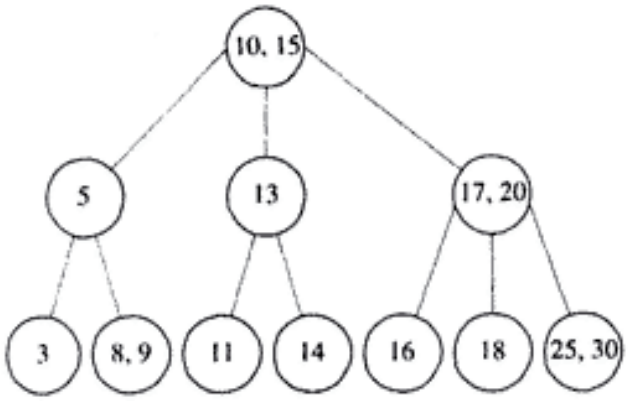
- هر گره شامل یک یا دو کلید است.
- اگر یک گرهٔ غیربرگ شامل یک کلید باشد، حتماً دو فرزند دارد و اگر آن شامل سه کلید باشد، در اینصورت سه فرزند خواهد داشت.
- کلیدهای زیردرخت چپ یک گرهٔ خاص، کوچکتر یا مساوی کلید ذخیره‌شده در آن گره هستند.
- اگر یک گره شامل دو کلید باشد، کلیدهای زیردرخت میانی گره، بزرگتر یا مساوی کلید چپ و کوچکتر یا مساوی کلید راست می‌باشند.
- همه برگها در یک سطح قرار دارند.

شکل ۷-۸، یک درخت ۲-۳ و نحوهٔ اضافه کردن یک کلید جدید به آن را نشان می‌دهد. توجه داشته باشید که درخت، متوازن باقی می‌ماند زیرا از ریشه به صورت عمقی رشد می‌یابد، نه از راه برگها. به طور مشابه، هنگامی که گره‌ای را حذف می‌کنیم، درخت از ریشه به طور عمقی عقب می‌کشد. در این روش، تمام برگها در یک سطح باقی می‌مانند و زمان‌های جستجو، اضافه و حذف در $\theta(lg\ n)$ قرار می‌گیرند. روشن است که یک پیمایش سطحی روی درخت، کلیدها را به شکل مرتب‌شده‌ای بازیابی می‌کند و به همین دلیل، درختهای-B در اکثر سیستمهای مدیریت بانکهای اطلاعاتی بکار می‌روند.

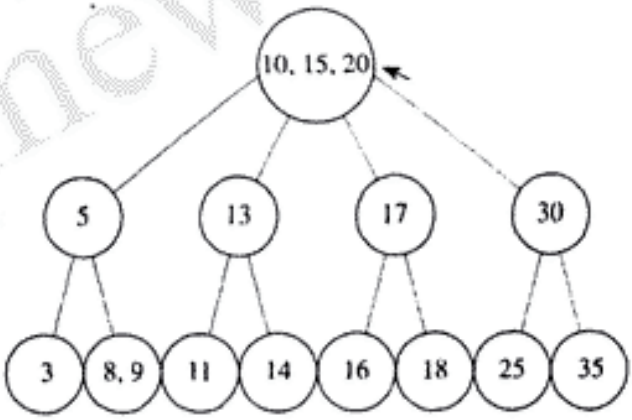
شکل ۷-۸ روش اضافه کردن یک کلید جدید به یک درخت ۲-۳.

(a) یک درخت ۲-۳

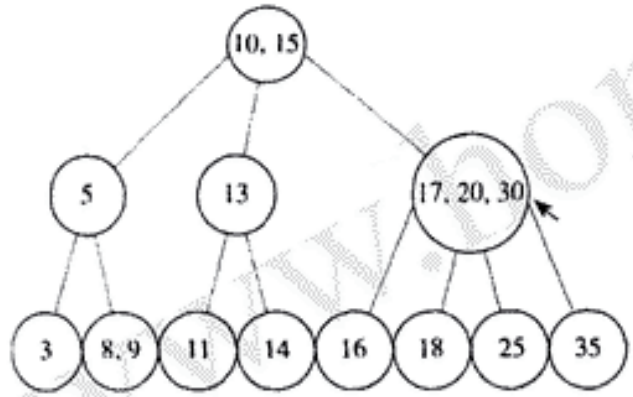
(b) ۳۵ در یک دنباله مرتب‌شده در یک برگ به درخت اضافه می‌شود.



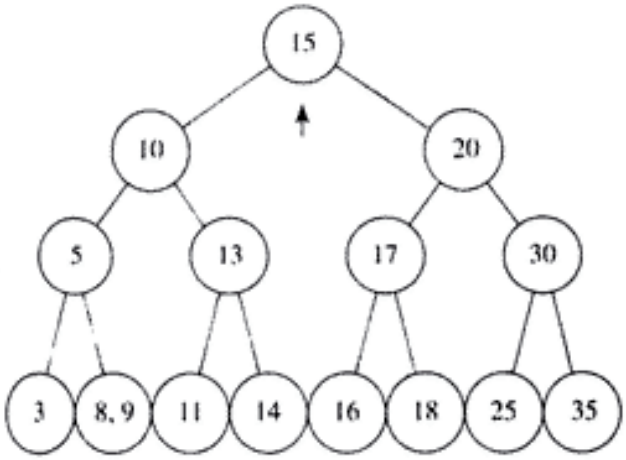
(c) اگر برگ شامل سه کلید باشد، به دو گره شکسته شده و گره میانی را به والدین خود ارسال می‌کند



(d) اگر والدین جدید شامل سه گره باشد، فرآیند شکستن و ارسال گره میانی تکرار می‌شود



(e) اگر ریشه شامل سه کلید باشد، شکسته شده و گره میانی به عنوان ریشه جدید ارسال می‌شود.



۸-۴ درهم‌سازی (Hashing)

فرض می‌کنیم کلیدها، اعدای صحیح از ۱ تا ۱۰۰ و تعداد رکوردها در حدود ۱۰۰ می‌باشد. یک روش مؤثر برای ذخیره‌کردن رکوردها، ایجاد یک آرایهٔ S از ۱۰۰ عنصر و ذخیره‌کردن رکورد با کلید i در S[i] است. بازیابی، فوراً و بدون انجام هیچ مقایسه‌ای انجام می‌شود. این استراتژی می‌تواند برای ۱۰۰ رکورد که کلید آنها عدد نه رقمی "امنیت ملی" است نیز استفاده شود؛ اگرچه در این حالت، استراتژی از لحاظ میزان حافظه بسیار غیرکارا می‌شود، زیرا یک آرایه با یک میلیون عنصر برای ذخیرهٔ ۱۰۰ رکورد مورد نیاز است. ما می‌توانیم یک آرایه ۱۰۰ عنصری با شاخصهای ۰ تا ۹۹ ایجاد نموده و هر کلید را با مقداری از ۰ تا ۹۹ "درهم‌سازی" کنیم. یک **تابع درهم‌ساز**، تابعی است که یک کلید را به یک شاخص تبدیل می‌کند. یک تابع درهم ساز برای عدد امنیت ملی می‌تواند به فرم زیر باشد:

$$h(key) = key \,\%\, ۱۰۰$$

(علامت % نمایانگر باقیمانده تقسیم مقدار کلید بر ۱۰۰ می‌باشد.) به عبارت بهتر، این تابع دو رقم آخر کلید را بازمی‌گرداند. اگر یک کلید خاص با i درهم شود، آنگاه کلید و رکورد آن در S[i] ذخیره می‌شود. لازم به ذکر است که در این روش کلیدها به شکل مرتب شده‌ای ذخیره نمی‌شوند.

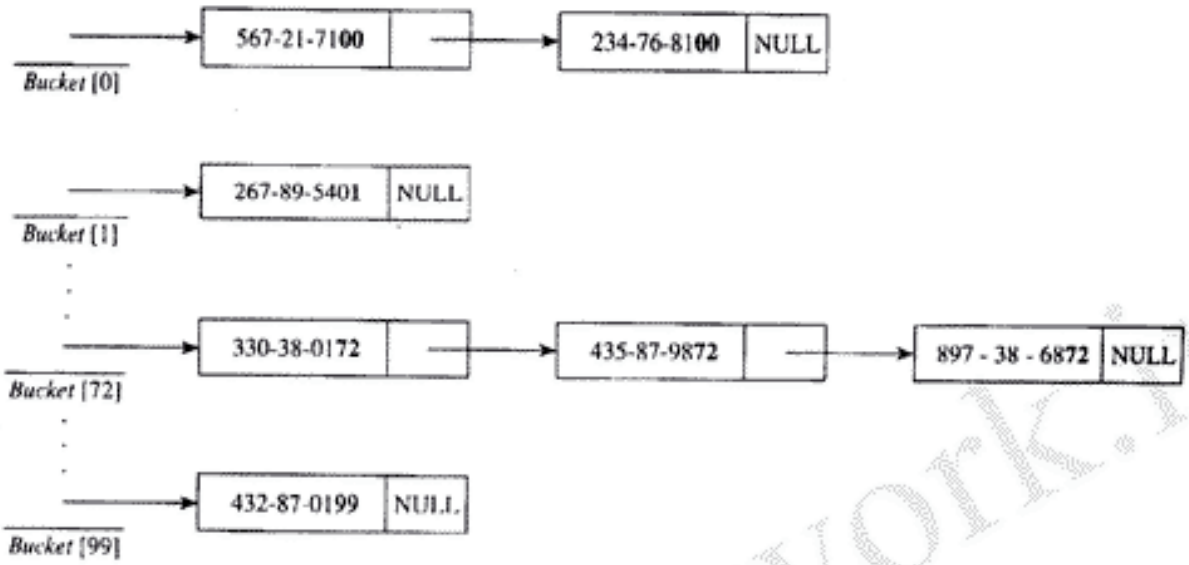
اگر هیچ دو کلیدی با یک شاخص درهم نشوند، این روش خوب عمل می‌کند. اگر چه این حالت، زمانی رخ می‌دهد که تعداد کلیدها قابل توجه باشد. به عنوان مثال، اگر ۱۰۰ کلید داشته باشیم و هر کلید از احتمال یکسانی برای درهم شدن با ۱۰۰ شاخص برخوردار باشد، احتمال اینکه هیچ دو کلیدی با یک شاخص درهم نشوند برابراست با

$$\frac{۱۰۰!}{۱۰۰^{۱۰۰}} \approx ۹/۳ \times ۱۰^{-۴۳}$$

یعنی اینکه تقریباً مطمئن هستیم که دو کلید با یک شاخص درهم می‌شوند. به چنین رخدادی، **تصادم** یا **برخورد درهم** گوئیم. روشهای مختلفی برای حل این مشکل وجود دارد که یکی از بهترین این روش‌ها، استفاده از **درهم‌سازی باز** یا **آدرس‌دهی باز** است. با درهم‌سازی باز، یک باکت برای هر مقدار درهم ممکن ایجاد شده و تمامی کلیدهایی که با یک مقدار درهم شده‌اند، به همراه آن مقدار در باکت قرار می‌گیرند. درهم‌سازی باز معمولاً به همراه لیست‌های پیوندی بکار گرفته می‌شود. برای مثال، اگر عمل درهم‌سازی را با دو رقم آخر یک عدد انجام دهیم، در واقع آرایه‌ای از اشاره‌گرها(Bucket) که از ۰ تا ۹۹ شاخص‌دهی شده‌اند، ایجاد کرده‌ایم. تمامی کلیدهایی که با i درهم شده‌اند، در لیست پیوندی که از Bucket[i] شروع می‌شود، قرار می‌گیرند. این روند را در شکل ۸-۸ نشان می‌دهیم.

تعداد باکتها، لزوماً با تعداد کلیدها برابر نیست. برای مثال، اگر دو رقم آخر عددی را درهم کنیم، آنگاه تعداد باکتها بایستی برابر ۱۰۰ باشد؛ اگرچه می‌توانستیم ۱۰۰، ۲۰۰، ۱۰۰۰ یا هر تعداد دلخواهی کلید را ذخیره نمائیم. اگر تعداد کلیدها بیشتر از تعداد باکتها باشد، آنگاه مطمئن خواهیم شد که تصادم رخ خواهد داد.

شکل ۸-۸ یک نمونه از درهم‌سازی باز. کلیدها براساس دو رقم آخر خود در باکت مشابه قرار می‌گیرند.



هنگام جستجوی یک کلید لازم است که یک جستجوی ترتیبی روی باکت (لیست پیوندی) شامل آن کلید انجام دهیم. اگر همه کلیدها در همان باکت درهم شده باشند، جستجو به یک جستجوی ترتیبی تنزل پیدا می‌کند. سؤال این است که چطور این اتفاق می‌افتد؟ اگر ۱۰۰ کلید و ۱۰۰ باکت وجود داشته باشد و یک کلید با احتمال مشابه بتواند با هر یک از باکتها درهم شود، آنگاه احتمال اینکه تمامی کلیدها در باکت مشابه قرار بگیرند، برابراست با

$$۱۰۰ \times (\frac{۱}{۱۰۰})^{۱۰۰} = ۱۰^{-۱۹۸}$$

بنابراین، وقوع چنین امری تقریباً غیرممکن است. احتمال اینکه ۹۰، ۸۰، ۷۰ یا هر تعداد بزرگ دیگری از کلیدها در باکت مشابه قرار بگیرند در چه حدودی است؟ هدف اصلی ما اینست که با استفاده از درهم‌سازی به جستجویی بهتر از جستجوی دودویی دست یابیم. ما نشان می‌دهیم که اگر فایل به اندازهٔ معقولی بزرگ باشد، این امر تقریباً تحقق خواهد یافت.

پرواضح است که بهترین چیزی که می‌تواند برای کلیدها رخ دهد اینست که آنها را به طور یکنواخت در باکتها توزیع کنیم؛ بدینصورت که اگر n کلید و m کلید باکت داشته باشیم، آنگاه هر باکت دارای n/m کلید باشد. به طور واقعی، زمانی هر باکت دارای n/m کلید است که n مضربی از m باشد؛ در غیر اینصورت، تقریباً توزیع یکنواختی خواهیم داشت. قضیهٔ زیر نشان می‌دهد که با توزیع یکنواخت کلیدها چه اتفاقی می‌افتد. برای سادگی، حالتی را درنظر می‌گیریم که توزیع دقیقاً یکنواخت می‌باشد. (n مضربی از m است).

قضیه ۴-۸ اگر n کلید به طور یکنواخت در m باکت توزیع شده باشند، آنگاه تعداد مقایسات در یک جستجوی ناموفق برابر n/m است.

اثبات: از آنجائیکه کلیدها به طور یکنواخت توزیع شده‌اند، لذا تعداد کلیدها در هر باکت برابر n/m است؛ بدین معنا که هر جستجوی ناموفق به n/m مقایسه نیاز دارد.

قضیه ۵-۸ اگر n کلید به طور یکنواخت در m باکت توزیع شده باشند و هر کلید با احتمال مشابهی به عنوان کلید جستجو درنظر گرفته شود، آنگاه میانگین تعداد مقایسات در یک جستجوی موفق برابراست با

$$\frac{n}{2m}+\frac{1}{2}$$

اثبات: متوسط زمان جستجو در هر باکت معادل متوسط زمان جستجو برای انجام یک جستجوی ترتیبی روی n/m کلید است. تحلیل حالت میانی الگوریتم ۱-۱ در بخش ۳-۱ نشان می‌دهد که این مقدار برابر است با $(n/2m)+(1/2)$.

مثال ۱-۸ اگر کلیدها به طور یکنواخت توزیع شوند و $n=2m$ باشد، آنگاه هر جستجوی ناموفق به تنها $2m/m=2$ مقایسه و هر جستجوی موفق به طور متوسط به $(2m/2m)+(1/2)=3/2$ مقایسه نیاز دارد.

باید توجه داشت زمانی که کلیدها به طور یکنواخت توزیع شده‌اند، زمان جستجو بسیار کوچک است. قضیه زیر نشان می‌دهد که اگر فایل به اندازه کافی بزرگ باشد، احتمال اینکه جستجوی درهم‌سازی به بدی جستجوی دودویی باشد، بسیار کم است.

قضیه ۶-۸ اگر n کلید و m باکت داشته باشیم، احتمال اینکه حداقل یک باکت شامل حداقل k کلید باشد، کوچکتر یا مساوی $\binom{n}{k}\left(\frac{1}{m}\right)^{k-1}$ است با فرض اینکه هر کلید با احتمال مشابهی بتواند در هر باکت قرار بگیرد.

اثبات: برای یک باکت معین، احتمال اینکه هر ترکیب مشخصی از k کلید در آن باکت جای بگیرند برابراست با $(1/m)^k$. به عبارت دیگر، احتمال اینکه یک باکت حداقل شامل کلیدهایی با ترکیب k کلید مشخص باشد برابراست با $(1/m)^k$. به طور کلی، برای دو پیشامد S و T داریم

$$p(S \text{ یا } T) \le p(S)+p(T) \qquad (۸-۱)$$

بنابراین، احتمال اینکه یک باکت حداقل شامل k کلید باشد، کوچکتر یا مساوی مجموع احتمالاتی است که باکت شامل حداقل کلیدها در هر ترکیب مجزا از k کلید می‌باشد. از آنجائیکه $\binom{n}{k}$ ترکیب مجزای k کلیدی می‌تواند از n کلید بدست آید، لذا احتمال اینکه یک باکت خاص شامل حداقل k باشد، کوچکتر یا

مساوی $\binom{n}{k}\left(\frac{1}{m}\right)^k$ خواهد بود. ادامهٔ اثبات قضیه با استفاده از عبارت ۱-۸ و این حقیقت که m باکت وجود دارد، انجام می‌شود.

به خاطر دارید که متوسط زمان جستجو برای جستجوی دودویی در حدود $\lg n$ است. جدول ۱-۸، حدود مختلف را برای احتمالات حداقل $\lg n$ کلید و $2\lg n$ کلید در یک باکت به ازاء مقادیر مختلف n نشان می‌دهد. در این جدول، n=m فرض شده است.

جدول ۱-۸ حدود بالا برای احتمال اینکه یک باکت حداقل شامل k کلید باشد.

n	Bound when $k = \lg n$	Bound when $k = 2\lg n$
128	.021	7.02×10^{-10}
1,024	.00027	3.49×10^{-16}
8,192	.0000013	1.95×10^{-23}
65,536	3.1×10^{-9}	2.47×10^{-31}

*It is assumed that the number of keys n equals the number of buckets.

۸-۵ مسئلهٔ انتخاب: مقدمه‌ای بر آرگومانهای مخالف

تاکنون در مورد جستجوی یک کلید x در یک لیست n کلیدی بحث کرده‌ایم. در ادامه، می‌خواهیم از یک مسئله جستجوی متفاوت به نام **مسئلهٔ انتخاب** صحبت کنیم. این مسئله با یافتن k امین کلید بزرگتر (یا کوچکتر) سروکار دارد. فرض می‌کنیم که کلیدها در یک آرایهٔ نامرتب قرار دارند. ابتدا در مورد حالتی بحث می‌کنیم که در آن k=۱ است؛ یعنی بزرگترین (یا کوچکترین) کلید را جستجو می‌کنیم. سپس موضوع را در حالت k=۲ بررسی نموده و در نهایت، به یک حالت کلی از مسئله دست می‌یابیم.

۸-۵-۱ یافتن بزرگترین کلید

الگوریتم زیر، یک الگوریتم سریع برای یافتن بزرگترین کلید است.

الگوریتم ۲-۸ یافتن بزرگترین کلید

مسئله: بزرگترین کلید در آرایهٔ n عنصری S را پیدا کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S با شاخصهای ۱ تا n

خروجی: متغیر large که مقدار آن بزرگترین کلید موجود در S است.

```
void find_largest (int n,
                   const keytype S[ ],
                   keytype& large)
{
  index i;
  large = S[1];
  for ( i = 2; i <= n; i++)
    if (S[i] > large)
      large = S[i];
}
```

پرواضح است که تعداد مقایسهٔ کلیدها در این الگوریتم برابراست با

$$T(n) = n - ۱$$

بنظر می‌رسد که اصلاح این کارایی غیرممکن باشد. قضیه بعد، تأئیدی بر این مطلب است. الگوریتم یافتن بزرگترین کلید همانند یک تورنمنت از کلیدها است. هر مقایسه، یک مسابقه است که در آن کلید بزرگتر، **برنده** و کلید کوچکتر، **بازنده** خواهد بود. بزرگترین کلید، برنده تورنمنت می‌گردد. ما از این اصطلاحات در این بخش استفاده خواهیم کرد.

قضیه ۷-۸ هر الگوریتم قطعی که بتواند بزرگترین کلید را از میان n کلید در هر ورودی ممکن، تنها با مقایسهٔ کلیدها پیدا کند بایستی حداقل n-۱ مقایسه را در هر حالت انجام دهد.

اثبات: اثبات این قضیه با برهان خلف انجام می‌شود. بدینصورت که نشان می‌دهیم اگر الگوریتم کمتر از n-۱ مقایسه را برای برخی از ورودیها به اندازهٔ n انجام دهد، آنگاه الگوریتم بایستی برای برخی از ورودیهای دیگر جواب اشتباهی ارائه نماید. در نهایت، اگر الگوریتم بتواند با انجام حداکثر n-۲ مقایسه برای برخی از ورودیها، بزرگترین کلید را پیدا کند، آنگاه حداقل دو کلید در آن ورودی هرگز در مقایسه‌ای بازنده نمی‌شوند و حداقل یکی از آن دو کلید نمی‌تواند به عنوان بزرگترین کلید گزارش شود. ما می‌توانیم با جایگزینی یک کلید که از همه کلیدها در ورودی اصلی بزرگتر است به جای آن کلید، یک ورودی جدید تولید کنیم. از آنجائیکه نتایج تمامی مقایسات مشابه ورودی اصلی خواهد بود، لذا کلید جدید به عنوان بزرگترین کلید گزارش نخواهد شد. بدین معنا که الگوریتم پاسخ اشتباهی را برای ورودی جدید ارائه خواهد داد. این تناقض ثابت می‌کند که الگوریتم بایستی حداقل n-۱ مقایسه برای هر ورودی به اندازهٔ n انجام دهد.

باید دقت کنید که قضیهٔ فوق را درست تفسیر کنید. این قضیه بدان معنا نیست که هر الگوریتمی که تنها با مقایسهٔ کلیدها جستجو می‌کند بایستی حداقل n-۱ مقایسه را برای پیدا کردن بزرگترین کلید انجام دهد. برای مثال، اگر یک آرایه مرتب شده باشد، می‌توانیم بدون انجام هیچ مقایسه‌ای، تنها با ارسال آخرین عنصر

آرایه، بزرگترین کلید را پیدا کنیم. پرواضح است که آرایه باید به صورت غیرنزولی مرتب شده باشد. به هر حال، یافتن بزرگترین کلید به این صورت تحت هر شرایطی امکان‌پذیر نیست. قضیه ۷-۸، مربوط به الگوریتم‌هایی است که می‌توانند بزرگترین کلید را در هر ورودی ممکن پیدا کنند.

۸-۵-۲ یافتن کوچکترین و بزرگترین کلید

یک روش سریع برای بدست آوردن همزمان کوچکترین و بزرگترین کلید، تغییر الگوریتم ۲-۸ به صورت زیر است.

الگوریتم ۳-۸ یافتن کوچکترین و بزرگترین کلید

مسئله: کوچکترین و بزرگترین کلید را در آرایهٔ n عنصری S پیدا کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S با شاخصهای ۱ تا n

خروجی: متغیر small و large که مقدار آنها کوچکترین و بزرگترین کلید در S است.

```
void find_both (int n,
                const keytype S[ ],
                keytype& small,
                keytype& large)
{
  index i;
  small = S[1];
  large = S[1];
  for (i = 2; i <= n; i++)
    if (S[i] < small)
      small = S[i];
    else if (S[i] > large)
      large = S[i];
}
```

استفاده از الگوریتم ۳-۸ بهتر از یافتن کوچکترین و بزرگترین کلید به طور مجزا است، زیرا برای برخی از ورودی‌ها، مقایسهٔ S[i] با large برای هر i انجام نمی‌شود؛ بدین ترتیب کارایی حالت میانی بهبود می‌یابد. اما هنگامی که S[۱] کوچکترین کلید آرایه است، آنگاه مقایسه به ازاء تمام i ها انجام می‌شود. بنابراین، بدترین حالت تعداد مقایسهٔ کلیدها برابراست با

$$W(n) = ۲(n - ۱)$$

که دقیقاً معادل تعداد مقایسات انجام شده در حالتی است که کوچکترین کلید و بزرگترین کلید به طور مجزا پیدا می‌شوند. به نظر می‌رسد که ما نمی‌توانیم این کارایی را اصلاح کنیم؛ اما چنین نیست. ما می‌توانیم با استفاده از زوج کلیدها و یافتن کلید کوچکتر در هر زوج این کار را انجام دهیم. این عمل می‌تواند با حدود n/۲ مقایسه و بزرگترین کلید را با حدود n/۲ مقایسه پیدا کنیم. در این روش، کوچکترین و بزرگترین کلید با

حدود ۳n/۲ مقایسه پیدا می‌شود. الگوریتم این روش به صورت زیر است. در این الگوریتم،n را عدد زوج فرض کرده‌ایم.

الگوریتم ۴-۸ یافتن کوچکترین و بزرگترین کلید با زوج کلیدها

مسئله: کوچکترین و بزرگترین کلید را در آرایهٔ n عنصری S پیدا کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S با شاخصهای ۱ تا n.

خروجی: متغیر small و large که مقدار آنها کوچکترین و بزرگترین کلید در S است.

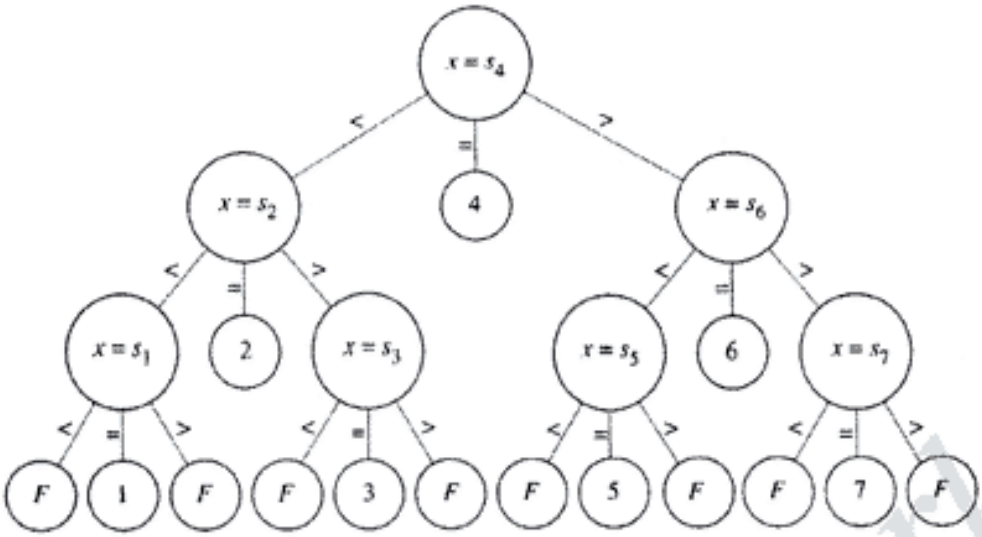
```
void find_both2 (int n,
                 const keytype S[ ],
                 keytype& small,
                 keytype& large)
{
   index i;
   if (S[1] < S[2]){
     small = S[1];
     large = S[2];
   }
   else{
     small = S[2];
     large = S[1];
   }
   for (i = 3; i <= n - 1; i++)
     if (S[i] > S[i+1])
       exchange S[i] and S[i+1];
     if (S[i] < small)
       small = S[i];
     if (S[i+1] > large)
       large = S[i+1];
}
```

تغییرالگوریتم برای درنظر گرفتن حالت فرد n را به عنوان تمرین به شما واگذار می‌کنیم. باید نشان دهید که تعداد مقایسهٔ کلیدها برابراست با

$$T(n) = \begin{cases} \frac{3n}{2} - 2 & \text{اگر n زوج باشد} \\ \frac{3n}{2} - \frac{3}{2} & \text{اگر n فرد باشد} \end{cases}$$

آیا می‌توانیم این کارایی را بهبود ببخشیم؟ نشان می‌دهیم که این امر ممکن نیست. برای این کار از درخت تصمیم استفاده نمی‌کنیم، زیرا درختهای تصمیم برای مسئله انتخاب به خوبی عمل نمی‌کنند و آنهم به این دلیل که درخت تصمیم برای مسئله انتخاب بایستی شامل حداقل n برگ برای n خروجی ممکن باشد. پیش‌قضیه ۳-۷ بیان می‌کند که اگر یک درخت دودویی n برگ داشته باشد، عمق آن بزرگتر یا مساوی

شکل ۹-۸ درخت تصمیم متناظر با الگوریتم ۲-۸، وقتی که n=۴ است.



$\lceil lg\, n \rceil$ خواهد بود. بنابراین، حد پائین ما روی تعداد برگها، حد پائین $\lceil lg\, n \rceil$ را برای تعداد مقایسات در بدترین حالت مشخص می‌کند. این یک حد پائین خوبی نیست زیرا تا بحال با حداقل n-۱ مقایسه می‌توانستیم بزرگترین کلید را پیدا کنیم (قضیه ۷-۸). پیش‌قضیه ۱-۸ نیز مفید نیست زیرا بیان می‌کند که بایستی n-۱ گرهٔ مقایسه در درخت تصمیم وجود داشته باشد. درخت تصمیم برای مسئله انتخاب به خوبی عمل نمی‌کند زیرا یک نتیجه می‌تواند در بیش از یک برگ قرار داشته باشد. شکل ۹-۸، درخت تصمیم را برای الگوریتم ۲-۸ (یافتن بزرگترین کلید)، وقتی که n=۴ است نشان می‌دهد. چهار برگ، کلید ۴ و دو برگ، کلید ۳ را گزارش می‌کنند. تعداد مقایسات انجام شده به وسیله الگوریتم، ۳ مورد بیشتر از $lg\, n = lg\, ۴ = ۲$ است، یعنی اینکه $lg\, n$ حد پائین مناسبی برای الگوریتم نیست.

ما برای ارائه یک حد پائین‌تر از روش دیگری موسوم به **آرگومان مخالف (adversary argument)** استفاده می‌کنیم. در این روش، در هر نقطه‌ای که الگوریتم بایستی تصمیمی بگیرد (مثلاً بعد از مقایسهٔ یک کلید)، مخالف سعی می‌کند تصمیمی را انتخاب نماید که الگوریتم را تا حد امکان طولانی نگه دارد. اگر مخالف، الگوریتم را وادار کند تا عمل مبنایی را $f(n)$ مرتبه انجام دهد، آنگاه $f(n)$ یک حد پائین‌تر برای پیچیدگی زمانی بدترین حالت الگوریتم است. ما از آرگومان مخالف برای بدست آوردن یک حد پائین روی بدترین حالت تعداد مقایسات مورد نیاز برای یافتن کوچکترین و بزرگترین کلید استفاده می‌کنیم. برای مشخص نمودن یک حد پائین می‌توانیم فرض کنید که کلیدها مجزا از هم می‌باشند. قبل از ارائه قضیه، استراتژی مخالف را تشریح می‌کنیم. فرض کنید که چند الگوریتم، مسئله یافتن کوچکترین و بزرگترین کلید را تنها با استفاده از مقایسهٔ کلیدها حل می‌کنند. اگر همه کلیدها مجزا باشند، یک کلید معین در طول اجرای الگوریتم، یکی از حالات زیر را به خود می‌گیرد:

حالت	شرح حالت
X	کلید با هیچ مقایسه‌ای سروکار ندارد.
L	کلید حداقل در یک مقایسه بازنده شده و هیچ بردی ندارد.
W	کلید حداقل در یک مقایسه برنده شده و هیچ باختی ندارد.
LW	کلید حداقل در یک مقایسه بازنده شده و حداقل یک برد دارد.

ما می‌توانیم فرض کنیم که این حالات شامل واحدهای اطلاعاتی هستند. اگر کلیدی در حالت X قرار دارد، یعنی هیچ واحد اطلاعاتی وجود ندارد. اگر کلیدی در حالت L یا W قرار دارد، یعنی یک واحد اطلاعاتی وجود دارد زیرا می‌دانیم که کلید در یک مقایسه بازنده یا پیروز شده است و اگر کلید در حالت LW باشد، یعنی دو واحد اطلاعاتی وجود دارد زیرا می‌دانیم که کلید در طول مقایسات هم برد و هم باخت داشته است. الگوریتمی که یک کلید Small را به عنوان کوچکترین کلید و یک کلید Large را به عنوان بزرگترین کلید مشخص می‌کند، بایستی بداند که هر کلیدی غیر از Small، دارای یک برد و هر کلیدی غیر از Large، دارای یک باخت می‌باشد. این بدین معناست که الگوریتم بایستی به $(n-1)+(n-1)=2n-2$ واحد اطلاعاتی دست یابد. از آنجائیکه هدف مخالف این است که الگوریتم، تا حد امکان به سختی کار کند، لذا می‌خواهد که با هر مقایسه، کمترین اطلاعات ممکن را تهیه نماید. به عنوان مثال، اگر الگوریتم در ابتدا S_2 و S_1 را با هم مقایسه کند، پاسخ مخالف هیچ اهمیتی نخواهد داشت زیرا به هر ترتیب دو واحد اطلاعاتی تهیه می‌شوند. فرض می‌کنیم که جواب مخالف S_2 باشد، آنگاه حالت S_2 از X به W و حالت S_1

جدول ۲-۸ استراتژی مخالف برای الگوریتمی که کوچکترین و بزرگترین کلیدها را پیدا می‌کند.

Before Comparison		Key Declared Larger by Adversary	After Comparison		Units of Information Learned by Algorithm
s_i	s_j		s_i	s_j	
X	X	s_i	W	L	2
X	L	s_i	W	L	1
X	W	s_j	L	W	1
X	WL	s_i	W	WL	1
L	L	s_i	W	L	1
L	W	s_j	L	W	0
L	WL	s_j	L	WL	0
W	W	s_i	W	WL	1
W	WL	s_i	W	WL	0
WL	WL	Consistent with previous answers	WL	WL	0

*The keys s_i and s_j are being compared.

از X به L تغییر می‌کند. حال اگر مقایسهٔ بعدی بین $S_۳$ و $S_۱$ باشد و جواب مخالف تعیین کند که $S_۱$ بزرگتر است، آنگاه حالت $S_۱$ از W به WL و حالت $S_۳$ از X به L تغییر می‌کند. این بدین معناست که دو واحد اطلاعاتی مشخص شده‌اند. از آنجائیکه با تعیین $S_۳$ به عنوان بزرگتر، تنها یک واحد اطلاعاتی مشخص می‌شود، لذا مخالف چنین پاسخی را به ما ارائه نموده است. جدول ۲-۸ یک استراتژی مخالف را نشان می‌دهد که همیشه حداقل مقدار جدول ۲-۸ استراتژی مخالف برای الگوریتم "یافتن کوچکترین کلید" اطلاعات را مشخص می‌نماید.

قضیه ۸-۸ هر الگوریتم قطعی که بتواند کوچکترین و بزرگترین کلید را در میان n کلید با هر ورودی ممکن، فقط با استفاده از مقایسهٔ کلیدها پیدا کند بایستی در بدترین حالت حداقل به تعداد زیر مقایسه انجام دهد:

اگر n زوج باشد $\frac{۳n}{۲} - ۲$

اگر n فرد باشد $\frac{۳n}{۲} - \frac{۳}{۲}$

اثبات : نشان می‌دهیم که این تعداد یک حدّ پائین روی بدترین حالت الگوریتم است. این کار را با نشان دادن این نکته که "الگوریتم بایستی حداقل این تعداد مقایسه را در هنگام مجزا بودن کلیدها انجام دهد"، صورت می‌دهیم. همانطوریکه قبلاً اشاره شد، الگوریتم بایستی برای پیدا کردن کوچکترین و بزرگترین کلید، حداقل $۲n - ۲$ واحد اطلاعاتی را بدست آورد. فرض کنید که ما آرگومان مخالفی را به همراه الگوریتم ارائه نمودیم. جدول ۲-۸ نشان می‌دهد که مخالف، تنها زمانی در یک مقایسه به دو واحد اطلاعاتی دست می‌یابد که هیچ یک از دو کلید در مقایسهٔ قبل درگیر نباشند. اگر n زوج باشد، حداکثر $n/۲$ مقایسه می‌تواند انجام شود و این بدین معناست که الگوریتم می‌تواند به حداکثر $۲(n/۲) = n$ واحد اطلاعاتی دست یابد. از آنجائیکه مخالف حداکثر یک واحد اطلاعاتی را در مقایسات دیگر بدست می‌آورد، لذا الگوریتم بایستی حداقل $۲n - ۲ - n = n - ۲$ مقایسهٔ اضافی را برای تکمیل اطلاعات مورد نیاز انجام دهد. بنابراین مخالف، الگوریتم را به انجام حداقل $۳n/۲ - ۲ = n/۲ + n - ۲$ مقایسه وادار می‌کند. اثبات قضیه برای حالتی که n فرد است، به عنوان تمرین به شما واگذار می‌شود.

از آنجائیکه الگوریتم ۴-۸، تعداد مقایساتی در حد قضیه ۸-۸ انجام می‌دهد، لذا الگوریتم در بدترین حالت مطلوب به نظر می‌رسد. ما مخالف بدی انتخاب کردیم زیرا الگوریتمی پیدا شد که از حدود خوبی برخوردار است. بنابراین، هیچ مخالف دیگری نمی‌تواند حد بالاتری را ارائه نماید.

هنگامی که از آرگومانهای مخالف استفاده می‌کنیم، گاهی اوقات آن را به نام **اُراکل (oracle)** می‌شناسیم. در جنگ بین یونان و روم، اُراکل شخصیتی آگاه و عالم بود که از عهدهٔ پاسخگویی به سؤالات دیگران به خوبی برمی‌آمد.

جدول ۳-۸ جواب مخالف در الگوریتم ۲-۸ برای اندازه ورودی ۵

Comparison	Key Declared Larger by Adversary	States/Assigned Values					Units of Information Learned by Algorithm
		s_1	s_2	s_3	s_4	s_5	
$s_2 < s_1$	s_2	L/10	W/20	X	X	X	2
$s_2 > s_3$	s_2	L/10	W/20	X	X	X	0
$s_3 < s_1$	s_3	L/10	W/20	W/15	X	X	1
$s_3 > s_2$	s_3	L/10	WL/20	W/30	X	X	1
$s_4 < s_1$	s_4	L/10	WL/20	W/30	W/15	X	1
$s_4 > s_3$	s_4	L/10	WL/20	WL/30	W/40	X	1
$s_5 < s_1$	s_5	L/10	WL/20	WL/30	W/40	W/15	1
$s_5 > s_4$	s_5	L/10	WL/20	WL/30	WL/40	W/50	1

۳-۵-۸ یافتن دومین کلید بزرگتر

برای یافتن دومین کلید بزرگتر می‌توانیم با استفاده از الگوریتم ۲-۸، بزرگترین کلید را با $n-1$ مقایسه پیدا کنیم. سپس با حذف آن کلید از مجموعهٔ کلیدها و استفادهٔ مجدد از الگوریتم ۲-۸ روی کلیدهای باقیمانده با $n-2$ مقایسه به بزرگترین کلید دست یابیم. بنابراین، می‌توانیم دومین کلید بزرگتر را با $2n - 3$ مقایسه پیدا کنیم. به نظر می‌رسد که بتوانیم اصلاحی را روی الگوریتم انجام دهیم، زیرا بسیاری از مقایسات انجام شده برای یافتن بزرگترین کلید می‌تواند برای حذف کلیدها در مرحلهٔ یافتن دومین کلید بزرگتر استفاده شود؛ بدینصورت که هر کلیدی که به یک کلید غیر از بزرگترین کلید باخته است، نمی‌تواند در مقایسات دومین کلید بزرگتر شرکت کند. روش تورنمنت، از این حقیقت استفاده می‌کند.

روش تورنمنت (Tournament Method)، روشی است که از تورنمنت‌های حذفی الگوبرداری شده است. به عنوان مثال، برای انتخاب بهترین تیم بسکتبال دانشگاهی در ایالات متحده، ۶۴ تیم در تورنمنت NCAA با هم رقابت می‌کنند. تیمها به صورت جفت، ۳۲ بازی را در دور اول انجام می‌دهند. ۳۲ تیم برنده به صورت جفت، ۱۶ بازی را در دور دوم انجام می‌دهند. این روال تا آنجا ادامه می‌یابد که تنها دو تیم برای دور نهایی باقی بمانند. برندهٔ مسابقه در دور نهایی، تیم قهرمان است. بدین ترتیب، برای تعیین قهرمان، $\lg 64 = 6$ دور مسابقه صورت گرفته است.

برای سادگی، فرض می‌کنیم که اعداد مجزا هستند و n توانی از ۲ است. همانطوریکه در تورنمنت NCAA اشاره شد، ما کلیدها را با هم جفت نموده و جفتها را در دورهای مختلف با هم مقایسه می‌کنیم تا اینکه فقط یک دور باقی بماند. اگر هشت کلید وجود داشته باشد، چهار مقایسه در دور اول، دو مقایسه در دور دوم و یک مقایسه در دور آخر انجام می‌شود. برندهٔ دور آخر، بزرگترین کلید است. شکل ۱۰-۸، این روش را نشان می‌دهد. روش تورنمنت معمولاً زمانی بکار می‌رود که n توانی از ۲ است. اگر n توانی از ۲ نباشد، همچنین می‌توانیم با افزودن تعداد کافی عنصر به انتهای آرایه، اندازهٔ آن را به توانی از ۲ برسانیم. برای مثال، اگر آرایه شامل ۵۳ عدد صحیح باشد، می‌توانیم با افزودن ۱۱ عنصر، هر یک با مقدار ∞ - به انتهای آرایه، آرایه‌ای با اندازهٔ ۶۴ تشکیل دهیم.

اگرچه در آخرین دور، کلید پیروز بزرگترین کلید است، اما عنصر بازنده لزوماً دومین کلید بزرگتر نیست. در شکل ۱۰-۸، دومین کلید بزرگتر (۱۶) در دور دوم به بزرگترین کلید (۱۸) می‌بازد. این مشکلی است که بسیاری از تورنمنت‌های واقعی با آن روبرو هستند زیرا همیشه دو تیم برتر در دور پایانی با هم رقابت نمی‌کنند. یک روش برای حل این مشکل اینست که اثری از کلیدهای بازنده به بزرگترین کلید را نگهداشته، سپس با استفاده از الگوریتم ۲-۸ برای یافتن بزرگترین آنها اقدام می‌کنیم. اما چگونه می‌توانیم این کلیدها را شناسائی کنیم در حالیکه هنوز بزرگترین کلید را مشخص نکرده‌ایم؟ این کار را با استفاده از لیست‌های پیوندی برای کلیدها (یک لیست برای هر کلید) انجام می‌دهیم. بعد از اینکه یک کلید در یک رقابت بازنده شد، به لیست پیوندی کلید برنده اضافه می‌شود. به عنوان تمرین، الگوریتمی برای این روش بنویسید. اگر n توانی از ۲ باشد، n/۲ مقایسه در دور اول، $n/2^2$ مقایسه در دور دوم، ...، $n/2^{\lg n} = ۱$ مقایسه در دور آخر انجام می‌شود. لذا مجموع تعداد مقایسات در تمام دورها برابراست با

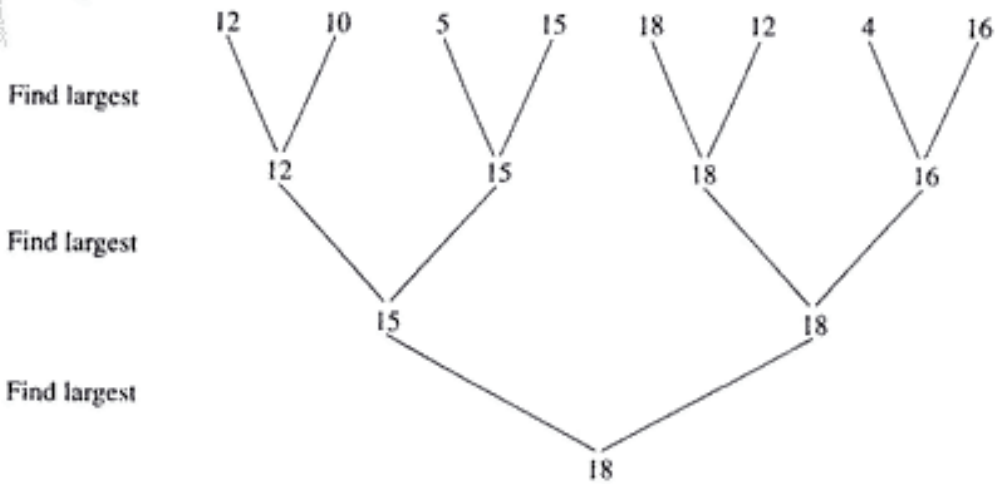
$$T(n) = n \sum_{i=۱}^{\lg n} (\frac{۱}{۲})^i = n[\frac{(۱/۲)^{\lg(n)+۱} - ۱}{۱/۲ - ۱} - ۱] = n - ۱$$

آخرین تساوی با استفاده از نتیجه مثال A-۲ در ضمیمه A بدست می‌آید. بنابراین، تعداد مقایسات مورد نیاز برای انجام یک تورنمنت برابر n-۱ است. بزرگترین کلید در $lg\ n$ مسابقه قرار می‌گیرد، بدین معنا که به تعداد lg n کلید در لیست پیوندی آن قرار می‌گیرد. با استفاده از الگوریتم ۲-۸، به $lg\ n$ - ۱ مقایسه برای یافتن بزرگترین کلید در این لیست پیوندی نیازمندیم. این کلید، دومین کلید بزرگتر در میان n کلید است. بنابراین، تعداد مقایسات مورد نیاز برای یافتن دومین کلید بزرگتر برابراست با

$$T(n) = n - ۱ + lg\ n - ۱ = n + lg\ n - ۲$$

به عنوان تمرین نشان خواهید داد که برای n در حالت کلی داریم:

$$T(n) = n + \lceil lg\ n \rceil - ۲$$



شکل ۱۰-۸ روش تورنمنت.

قضیه ۹-۸ هر الگوریتم قطعی که بتواند دومین کلید بزرگتر در میان n کلید با هر ورودی ممکن را تنها با مقایسهٔ کلیدها پیدا کند بایستی در بدترین حالت حداقل $n + \lceil \lg n \rceil - 2$ مقایسه انجام دهد.

اثبات: اثبات قضیه با استفاده از آرگومان مخالف انجام می‌شود.

۸-۵-۴ یافتن k امین کلید کوچکتر

در حالت کلی، مسئله انتخاب با یافتن k امین کلید کوچکتر یا بزرگتر در ارتباط است. تاکنون دربارهٔ یافتن بزرگترین کلید بحث کرده‌ایم، حال می‌خواهیم الگوریتمی برای یافتن k امین کلید کوچکتر ارائه دهیم. برای سادگی، فرض می‌کنیم که کلیدها از هم مجزا می‌باشند.

یک روش ساده برای یافتن k امین کلید کوچکتر با کارایی $\theta(n \lg n)$، مرتب‌سازی کلیدها و بازگرداندن k امین کلید است. ما روشی ارائه می‌دهیم که در آن، تعداد کمتری مقایسه نیاز داشته باشد. به خاطر دارید که روال Partition در الگوریتم ۷-۲، که در الگوریتم Quicksort مورد استفاده قرار می‌گرفت، یک آرایه را طوری تقسیم‌بندی می‌نمود که تمامی کلیدهای کوچکتر از عنصر محوری، قبل از آن و تمامی کلیدهای بزرگتر از عنصر محوری، بعد از آن قرار بگیرند. اندیسی که عنصر محوری در آن قرار می‌گرفت را pivotpoint نامیدیم. حال می‌توانیم مسئله انتخاب را با این تقسیم‌بندی به گونه‌ای حل کنیم که عنصر محوری در اندیس k ام قرار بگیرد. اگر k کوچکتر از pivotpoint باشد، تقسیم‌بندی بازگشتی را برای زیرآرایهٔ چپ و اگر k بزرگتر از pivotpoint باشد، تقسیم‌بندی بازگشتی را برای زیرآرایهٔ راست بکار می‌گیریم. هنگامی که k=pivotpoint باشد، کار انجام شده است. الگوریتم تقسیم و غلبه زیر، مسئله را با این روش حل می‌کند.

الگوریتم ۸-۵ انتخاب

مسئله: k امین کلید کوچکتر در آرایهٔ n عنصری S را پیدا کنید.

ورودی: اعداد صحیح مثبت k و n بطوری که $k \leq n$، آرایه‌ای از کلیدهای مجزا S با شاخصهای ۱ تا n.

خروجی: k امین کلید کوچکتر در S که بعنوان مقدار تابع Selection بازمی‌گردد.

```
keytype selection (index low, index high, index k)
{
  index pivotpoint;
  if (low == high)
    return S[low];
  else {
    partition(low, high, pivotpoint);
    if (k == pivotpoint)
      return S[pivotpoint];
    else if (k < pivotpoint)
      return selection(low, pivotpoint - 1, k);
    else
```

```
        return selection(pivotpoint + 1, high, k);
    }
}

void partition (index low, index high,      // This is the same routine that
                index& pivotpoint)          // appears in Algorithm 2.7.
{
    index i, j;
    keytype pivotitem;

    pivotitem = S[low];                     // Choose first item for pivotitem.
    j = low;
    for (i = low + 1; i <= high; i++)
        if (S[i] < pivotitem) {
            j++;
            exchange S[i] and S[j];
        }
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];      // Put pivotitem at pivotpoint.
}
```

فراخوانی سطح بالای تابع Selection به صورت زیر است:

```
kthsmallest = selection(1, n, k);
```

همانند Quicksort (الگوریتم ۶-۲)، بدترین حالت زمانی اتفاق می‌افتد که ورودی هر فراخوانی بازگشتی، شامل یک عنصر کمتر باشد. مثلاً زمانی که آرایه به صورت صعودی مرتب شده باشد و $k = n$ باشد. بنابراین، پیچیدگی زمانی بدترین حالت الگوریتم ۵-۸ مشابه الگوریتم ۶-۲ است، بدین معنا که پیچیدگی زمانی بدترین حالت تعداد مقایسهٔ کلیدها برای الگوریتم ۵-۸ برابراست با

$$W(n) = \frac{n(n-1)}{2}$$

اگرچه پیچیدگی زمانی الگوریتم ۵-۸ در بدترین حالت مشابه Quicksort است، اما نشان می‌دهیم که الگوریتم در حالت میانی خیلی بهتر از آن عمل می‌کند.

تحلیل پیچیدگی زمانی حالت میانی الگوریتم ۵-۸ (انتخاب)

عمل مبنایی: مقایسهٔ S[i] با pivotpoint در partition.

اندازه ورودی: n، تعداد عناصر آرایهٔ S.

فرض می‌کنیم که همه ورودی‌ها از احتمال مشابهی برخوردارند. برای سادگی، P را به جای pivotpoint

در نظر می‌گیریم. n خروجی وجود دارد که هیچ فراخوانی بازگشتی ندارند (در صورتیکه به ازاء k = ۱، ۲، ... ، n داشته باشیم p=k). دو خروجی وجود دارد که اندازهٔ ورودی آنها در اولین فراخوانی بازگشتی برابر ۱ است (در صورتی که به ازاء k=۱ داشته باشیم p=۲، یا به ازاء k=n داشته باشیم p=n-۱). ۲(۲)=۴ خروجی وجود دارد که اندازهٔ ورودی آنها در اولین فراخوانی بازگشتی برابر ۲ است (در صورتیکه به ازاء k=(۲ یا ۱) داشته باشیم p=۳، یا به ازاء k=(n یا n-۱) داشته باشیم p=n-۲).
در زیر لیستی از تعداد خروجی‌ها برای همهٔ اندازه ورودیها را آورده‌ایم:

اندازهٔ ورودی در اولین فراخوانی بازگشتی	*تعداد خروجی‌ها*
۰	n
۱	۲
۱	۲(۲)
۳	۲(۳)
⋮	⋮
i	۲(i)
⋮	⋮
n-۱	۲(n-۱)

از تحلیل حالت معمول الگوریتم ۷-۲ به خاطر دارید که تعداد مقایسات در روال partition برابراست با n-۱. بنابراین، میانگین با معادله بازگشتی زیر مشخص می‌شود:

$$A(n)=\frac{nA(0)+2[A(1)+2A(2)+\ldots+iA(i)+\ldots+(n-1)A(n-1)]}{n+2(1+2+\ldots+i+\ldots+n-1)}+n-1$$

با استفاده از نتیجه مثال A-۱ در ضمیمه A، با ساده‌کردن معادله و این حقیقت که A(۰)=۰ است داریم:

$$A(n)=\frac{2[A(1)+2A(2)+\ldots+iA(i)+\ldots+(n-1)A(n-1)]}{n^2}+n-1$$

که با اعمال چند محاسبه جبری و نتیجه‌گیری از مباحث قبلی خواهیم داشت:

$$A(n)=\frac{n^2-1}{n^2}A(n-1)+\frac{(n-1)(3n-2)}{n^2}<A(n-1)+3$$

و از آنجائیکه A(۰)=۰ است، لذا بازگشت زیر بدست می‌آید:

$$A(n)<A(n-1)+3 \qquad n>0$$
$$A(0)=0$$

این بازگشت می‌تواند با استفاده از استقراء حل شود. جواب بازگشت چنین است:

$$A(n)<3n$$

به روشی مشابه می‌توانیم با استفاده از بازگشت نشان می‌دهیم که $A(n)$ با یک حد پائین خطی محدود

می‌شود. بنابراین،

$$A(n) \in \theta(n)$$

پرواضح است که برای مقادیر بزرگ n داریم:

$$A(n) \approx ۳n$$

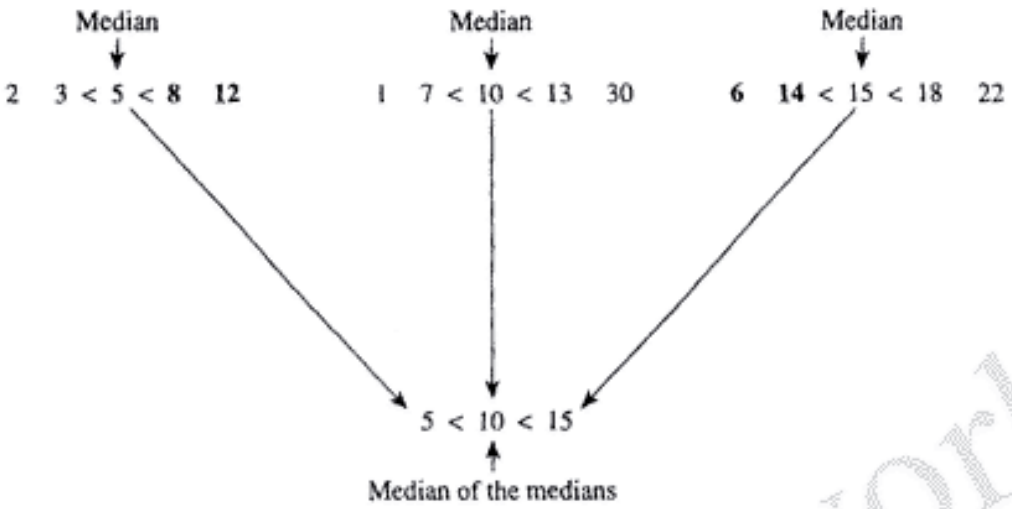
الگوریتم ۵-۸ در حالت میانی، تنها یک تعداد خطی مقایسه روی کلیدها انجام می‌دهد. البته دلیل بهتر بودن این الگوریتم نسبت به Quicksort در حالت میانی این است که Quicksort دو فراخوانی از partition دارد در حالیکه در این الگوریتم، یک فراخوانی از partition وجود دارد. به هر حال، وقتی که ورودی فراخوانی بازگشت برابر n-۱ باشد، کارایی هر دو الگوریتم تنزل پیدا می‌کند. (در این حالت، Quicksort یک زیرآرایهٔ خالی را می‌پذیرد). پیچیدگی زمانی آن مربعی است. نشان می‌دهیم که چگونه می‌توان با پیشگیری از وقوع چنین امری، کارایی زمان مربعی بدترین حالت را اصلاح نمود.

به خاطر دارید که **میانهٔ (median)** n کلید مجزا، کلیدی است که نیمی از کلیدها کوچکتر از آن و نیمی دیگر، بزرگتر از آن باشند. اگر می‌توانستیم همیشه میانه را برای متغیر pivotpoint انتخاب کنیم، از کارایی مطلوبی برخوردار می‌شدیم. اما چگونه می‌توان میانه را تعیین نمود؟ در روال pivotpoint می‌توانستیم تابع Selection را با یک ورودی شامل آرایهٔ اصلی و k که مقدار آن در حدود نصف اندازهٔ آن آرایه باشد فراخوانی کنیم. فرض کنید که n مضرب فردی از ۵ است. n کلید را به n/۵ گروه از کلیدها، هر یک شامل ۵ کلید تقسیم می‌کنیم. میانهٔ هر یک از این گروهها را مستقیماً پیدا می‌کنیم که این کار با انجام شش مقایسه صورت می‌گیرد. آنگاه با فراخوانی تابع Selection، میانهٔ تمامی میانه‌ها را تعیین می‌کنیم. میانهٔ میانه‌ها لزوماً میانهٔ n کلید نیست، بلکه مقداری نزدیک به آن می‌باشد. در شکل ۱۲-۸، کلیدهای چپ کوچکترین میانه (کلیدهای ۲ و ۳) بایستی کوچکتر از میانهٔ میانه‌ها و کلیدهای راست بزرگترین میانه (کلیدهای ۱۸ و ۲۲) بایستی بزرگتر از میانهٔ میانه‌های باشند و در حالت کلی، کلیدهای راست کوچکترین میانه (کلیدهای ۸ و ۱۲) و کلیدهای چپ بزرگترین میانه (کلیدهای ۶ و ۱۴) می‌توانستند در هر دوطرف میانهٔ میانه‌ها قرار بگیرند. توجه کنید که به تعداد $۲(\frac{۱۵}{۵} - ۱)$ کلید وجود دارد که می‌توانست در دوطرف میانهٔ میانه‌ها قرار بگیرد و می‌تواند نتیجه گرفت که اگر n مضرب فردی از ۵ باشد، به تعداد $۲(\frac{۱۵}{۵} - ۱)$ کلید وجود داد که می‌توانست در دو طرف میانهٔ میانه‌ها قرار بگیرید. بنابراین، حداکثر

$$\underbrace{\frac{۱}{۲}[n - ۱ - ۲(\frac{n}{۵} - ۱)]}_{\text{تعداد کلیدهایی که می‌دانیم در یک طرف قرار دارند.}} + ۲(\frac{n}{۵} - ۱) = \frac{۷n}{۱۰} - \frac{۳}{۲}$$

کلید در یک طرف میانهٔ میانه‌ها وجود دارد.

شکل ۱۲-۸ هر عدد نمایانگر یک کلید است. ما نمی‌دانیم که کلیدهای تیره کوچکتر از میانهٔ میانه‌ها هستند یا بزرگتر از آن.



الگوریتم ۶-۸ انتخاب با استفاده از میانه

مسئله: k امین کلید کوچکتر در آرایهٔ n عنصری در S را پیدا کنید.

ورودی: اعداد صحیح مثبت n و K بطوری که $k \leq n$، آرایه‌ای از کلیدها S با شاخصهای ۱ تا n.

خروجی: k امین کلید کوچکتر در S که توسط مقدار تابع Select بازگردانده می‌شود.

```
keytype select (int n,
                keytype S[ ],
                index k)
{
   return selection2(S, 1, n, k);
}

keytype selection2 (keytype S[ ],
                    index low, index high, index k)
{
   if (high == low)
      return S[low];
   else {
      partition2(S, low, high, pivotpoint);
      if (k == pivotpoint)
         return S[pivotpoint];
      else if (k < pivotpoint)
```

```
        return selection2(S, low, pivotpoint − 1, k);
      else
        return selection2(S, pivotpoint + 1, high, k);
    }
}
void partition2 (keytype S[ ],
                 index low, index high,
                 index& pivotpoint)
{
  const arraysize = high − low + 1;
  const r = ⌈arraysize / 5⌉;
  index i, j, mark, first, last;
  keytype pivotitem, T[1..r];

  for (i = 1; i <= r; i++) {
    first = low + 5*i − 5;
    last = minimum(low + 5*i − 1, arraysize);
    T[i] = median of S[first] through S[last];
  }
  pivotitem = select(r, T, ⌊(r + 1) / 2⌋);        // Approximate the medium.
  j = low;
  for (i = low; i <= high; i++)
    if (S[i] == pivotitem) {
      exchange S[i] and S[j];
      mark = j;                                    // Mark where pivotitem
      j++;                                         // placed.
    }
    else if (S[i] < pivotitem) {
      exchange S[i] and S[j];
      j++;
    }
  pivotpoint = j − 1;
  exchange S[mark] and S[pivotpoint];              // Put pivotitem at pivotpoint.
}
```

در الگوریتم ۶-۸، برخلاف الگوریتم‌های بازگشتی دیگر، تابع ساده‌ای که تابع بازگشتی ما را فراخوانی می‌کند را نشان می‌دهیم. به این دلیل که تابع بایستی در دو جا با دو ورودی متفاوت فراخوانی شود. بدینصورت که در روال partition۲ با ورودی T و در کل به صورت

kthsmallest = selection (۱, n, k)

فراخوانی می‌شود.

۸-۵-۵ یک الگوریتم احتمالی برای مسئله انتخاب

در حدود بدست آمده برای الگوریتم‌ها، فرض کرده‌ایم که الگوریتم‌ها قطعی هستند، اما به این نکته نیز اشاره نموده‌ایم که این حدود برای الگوریتم‌های احتمالی نیز صدق می‌کنند. ما با ارائه یک الگوریتم احتمالی برای مسئله انتخاب، این مطلب را مشخص کرده و نشان می‌دهیم که چه زمانی اینگونه الگوریتم‌ها مفید می‌باشند.

در بخش ۳-۵، یک الگوریتم احتمالی موسوم به الگوریتم مونت کارلو، برای تقریب کارایی یک الگوریتم بک‌تراکینگ ارائه نمودیم. به خاطر دارید که یک الگوریتم مونت کارلو لزوماً یک جواب درست تولید نمی‌کند، بلکه فقط تخمینی را از جواب ارائه می‌نماید که احتمال اینکه این تخمین به جواب درست نزدیک باشد، با توجه به شرایط الگوریتم افزایش می‌یابد. در اینجا، نوع متفاوتی از الگوریتم احتمالی، موسوم به الگوریتم Sherwood را ارائه می‌دهیم.

یک **الگوریتم Sherwood**، همواره جواب درست ارائه می‌دهد. این الگوریتم‌ها، زمانی مفید و کارا هستند که الگوریتم‌های قطعی متناظر، در حالت میانی بسیار سریعتر از بدترین حالت عمل می‌نمایند. به خاطر دارید که این مطلب برای الگوریتم ۵-۸ (انتخاب) درست است. بدترین حالت زمان-مربعی هنگامی حاصل می‌شود که pivotpoint برای یک ورودی خاص در فراخوانی‌های بازگشتی، مکرراً به مقدار low یا high نزدیک باشد. به عنوان مثال، زمانی این حالت پیش می‌آید که آرایه به صورت صعودی مرتب شده و k=n باشد. از آنجائیکه این حالت برای اکثر ورودی‌ها پیش نمی‌آید، لذا کارایی حالت میانی الگوریتم به صورت خطی است. فرض کنید که برای یک ورودی خاص، عنصر میانی را به طور تصادفی براساس یک توزیع غیریکنواخت انتخاب کردیم. آنگاه زمانی که الگوریتم برای آن ورودی اجرا می‌شود، pivotpoint بسیار تمایل دارد که دور از نقاط پایانی باشد. بنابراین، احتمال کارایی خطی آن بسیار بالا خواهد بود. از آنجائیکه تعداد مقایسات در هنگام گرفتن میانگین از همه ورودیها به صورت خطی است، لذا به نظر می‌رسد که **مقدار مورد انتظار** تعداد مقایسهٔ کلیدها برای یک ورودی خاص، وقتی که عنصر میانی به طور تصادفی براساس یک توزیع غیریکنواخت انتخاب می‌شود، به صورت خطی باشد. ما ثابت می‌کنیم که این حالت وجود دارد، اما ابتدا می‌خواهیم تفاوت بین این مقدار مورد انتظار و مقدار میانی بدست آمده از الگوریتم ۵-۸ را مشخص کنیم. با فرض اینکه تمامی ورودیهای ممکن از لحاظ تعداد معادل هستند. متوسط تعداد مقایسات انجام شده توسط الگوریتم ۵-۸ به صورت خطی خواهد بود. برای هر ورودی معین، الگوریتم ۵-۸ همواره تعداد یکسانی مقایسه را انجام می‌دهد که این تعداد برای برخی از ورودی‌ها به صورت مربعی است ولی الگوریتم Sherwood، گاهی اوقات به صورت خطی و گاهی اوقات به صورت مربعی عمل می‌کند. به هر حال، اگر الگوریتم چندین مرتبه با ورودی‌های مشابه اجرا شود، انتظار داریم که میانگین زمانهای اجرا به صورت خطی باشد.

شاید بخواهید بدانید که چرا از چنین الگوریتمی استفاده می‌کنیم وقتی که الگوریتم ۶-۸ (انتخاب با استفاده از میانه) کارایی خطی را تضمین می‌کند. به این علت که الگوریتم ۶-۸ به دلیل سربار مورد نیاز برای تقریب میانه از ثابت بالایی برخوردار است. برای یک ورودی معین، الگوریتم Sherwood در حالت میانی

سریعتر از الگوریتم ۶-۸ عمل می‌کند. تصمیم در مورد استفاده از الگوریتم Sherwood یا الگوریتم ۶-۸ به نیازهای یک کاربرد خاص بستگی دارد. اگر کارایی حالت میانی بسیار مهم باشد، از الگوریتم Sherwood استفاده می‌کنیم. اما اگر کارایی زمان مربعی به هیچ وجه قابل تحمل نباشد، از الگوریتم ۶-۸ استفاده می‌کنیم. لازم به ذکر است که الگوریتم ۵-۸ نیز در حالت میانی بهتر از الگوریتم ۶-۸ عمل می‌کند. در ادامه، الگوریتم احتمالی (Sherwood) را ارائه می‌دهیم.

الگوریتم ۷-۸ انتخاب احتمالی

مسئله: k امین کلید کوچکتر در آرایهٔ n عنصری S را پیدا کنید.

ورودی: اعداد صحیح مثبت n و k بطوری که $k \leq n$، آرایه‌ای از کلیدها S با شاخصهای ۱ تا n.

خروجی: k امین کلید کوچکتر در S که در مقدار تابع Selection۳ بازگردانده می‌شود.

```
keytype selection3 (index low, index high, index k)
{
  if (low == high)
    return S[low];
  else {
    partition3(low, high, pivotpoint);
    if (k == pivotpoint)
      return S[pivotpoint];
    else if (k < pivotpoint)
      return selection3(low, pivotpoint - 1, k);
    else
      return selection3(pivotpoint + 1, high, k);
  }
}

void partition3 (index low, index high,
                 index& pivotpoint)
{
  index i, j, randspot;
  keytype pivotitem;

  randspot = random index between low and high inclusive;
  pivotitem = S[randspot];                    // Randomly choose pivotitem.
  j = low;
  for (i = low + 1; i <= high; i++)
    if (S[i] < pivotitem) {
      j++;
      exchange S[i] and S[j];
    }
  pivotpoint = j;
  exchange S[low] and S[pivotpoint];          // Put pivotitem at pivotpoint.
}
```

تنها اختلاف الگوریتم ۷-۸ با الگوریتم ۵-۸ در انتخاب تصادفی عنصر میانی است. در ادامه، ثابت می‌کنیم که مقدار مورد انتظار تعداد مقایسات برای هر ورودی به صورت خطی است.

تحلیل پیچیدگی زمانی مقدار مورد انتظار الگوریتم ۷-۸ (انتخاب احتمالی)

عمل مبنایی: مقایسهٔ S[i] با pivotpoint در partition.

اندازه ورودی: n، تعداد عناصر آرایه S.

از آنجائیکه ما در حال جستجوی k امین کلید کوچکتر در یک ورودی به اندازهٔ n هستیم، لذا در جدول زیر اندازهٔ ورودی و مقدار جدید k در اولین فراخوانی بازگشتی برای هر مقدار pivotpoint را آورده‌ایم.

pivotpont	*اندازهٔ ورودی*	*مقدار جدید k*
۱	n-۱	k -۱
۲	n-۲	k -۲
⋮	⋮	⋮
k -۱	n-(k -۱)	۱
k	*	
k +۱	k	k
k +۲	k +۱	k
⋮	⋮	⋮
n	n -۱	k

از آنجائیکه تمامی مقادیر pivotpoint از احتمال یکسانی برخوردارند، بازگشت زیر را برای مقدار مورد انتظار ارائه می‌دهیم:

$$E(n,k)=\frac{۱}{n}\left[\sum_{p=۱}^{k-۱}E(n-p,k-p)+\sum_{p=k}^{n-۱}E(p,k)\right]+n-۱$$

ما می‌توانیم این بازگشت را با استفاده از استقراء ساختاری ثابت کنیم. بدینصورت که، از آنجائیکه ما از خطی بودن بازگشت مطمئن نیستیم، لذا جستجو را برای مقداری از C که آرگومان استقراء، نامساوی $E(n,k)\le cn$ را ثابت می‌کند، انجام می‌دهیم. در اینجا بحث استقراء را نشان می‌دهیم اما به عنوان تمرین، تعیین C= ۴ به عنوان کوچکترین ثابت را به شما واگذار می‌کنیم.

پایه استقراء: از آنجائیکه هیچ مقایسه‌ای برای n= ۱ انجام نمی‌شود، لذا

$$E(n,k)=۰\le ۴$$

فرض استقراء: فرض کنید که برای هر $E<n$ و هر $k\le m$،

$$E(mk)\le ۴m$$

گام استقراء: بایستی نشان دهیم که برای هر $k\le n$،

$$E(n,k)\le ۴n$$

با توجه به بازگشت و فرض استقراء داریم:

$$E(n,k) \le \frac{1}{n}\left[\sum_{p=1}^{k-1} 4(n-p) + \sum_{p=k}^{n-1} 4p\right] + n - 1 \qquad (۸-۳)$$

و ما داریم:

$$\sum_{p=1}^{k-1} 4(n-p) + \sum_{p=k}^{n-1} 4p = 4\left[\sum_{p=1}^{k-1} n - \sum_{p=1}^{k-1} p + \sum_{p=k}^{n-1} p\right]$$

$$= 4\left[(k-1)n - \sum_{p=1}^{k-1} p + \sum_{p=k}^{n-1} p\right]$$

$$= 4\left[(k-1)n - (k-1)k + \frac{(n-1)n}{2}\right]$$

$$= 4\left[(k-1)(n-k) + \frac{(n-1)n}{2}\right]$$

$$< 4\left[k(n-k) + \frac{n}{2}\right] \le 4\left[\frac{n^2}{4} + \frac{n^2}{2}\right] = 3n^2.$$

سومین تساوی با دو بار جایگزینی نتیجه مثال ۱-A در ضمیمه A بدست می‌آید و نامساوی اخیر از این حقیقت که در حالت کلی، $k(n-k) \le n^2/4$ است ناشی می‌شود. با ترکیب نتیجه بدست آمده با نامساوی ۳-۸ خواهیم داشت:

$$E(n,k) < \frac{1}{n}(3n^2) + n - 1 = 3n + n - 1 < 4n$$

ما نشان داده‌ایم که مستقل از مقدار k،

$$E(n,k) < 4n \in \theta(n)$$

تمرینات

بخش ۱-۸

۱- فرض کنید که S و T دو آرایهٔ m و n عنصری می‌باشند. الگوریتمی بنویسید که تمامی عناصر مشترک را پیدا کرده و آنها را در آرایهٔ U ذخیره نماید. نشان دهید که آین کار می‌تواند در زمان $\theta(n+m)$ انجام شود.

۲- نشان دهید که اگر x بتواند با احتمال مشابه در هر یک از اندیسهای آرایه وجود داشته باشد، آنگاه پیچیدگی زمانی حالت میانی جستجوی دودویی (الگوریتم ۵-۱) تقریباً به صورت زیر است:

$$\lfloor lg\, n \rfloor - 1 \le A(n) \le \lfloor lg\, n \rfloor$$

توجه: براساس پیش‌قضیه ۴-۸، به ازاء برخی k، تعداد گره‌ها در سطح پائینی برابر $(۱ - ۲^k) - n$ است. برای بدست TND درخت تصمیم این عبارت را به عبارت $۱ + ۲^k (۱ - k)$ (فرمول ارائه شده در تحلیل حالت میانی جستجوی دودویی) اضافه کنید.

۳- فرض کنید که تمامی $۱ + ۲n$ امکان زیر از احتمال یکسانی برخوردارند:

$$x = s_i \qquad ۱ \le i \le n$$
$$x < s_۱$$
$$s_i < x < s_{i+۱} \qquad ۱ \le i \le n-۱$$
$$x > s_n$$

نشان دهید پیچیدگی زمانی حالت میانی جستجوی دودویی (الگوریتم ۵-۱) تقریباً بصورت زیر است:

$$\lfloor lg\, n \rfloor - \frac{۱}{۲} \le A(n) \le \lfloor lg\, n \rfloor + \frac{۱}{۲}$$

۴- اثبات پیش‌قضیه ۶-۸ را کامل کنید.

بخش ۲-۸

۵- نشان دهید که پیچیدگی زمانی حالت میانی جستجوی درون‌یابی در $\theta(lg(lg\, n))$ است، با فرض اینکه کلیدها به صورت غیریکنواخت توزیع شده‌اند و کلید جستجوی x با احتمال مشابه می‌تواند در هر یک از اندیسهای آرایه باشد.

۶- نشان دهید که پیچیدگی زمانی بدترین حالت جستجوی درون‌یابی در $\theta((lg\, n)^۲)$ است. با فرض اینکه کلیدها به صورت غیریکنواخت توزیع شده‌اند و کلید جستجوی x با احتمال مشابه می‌تواند در هر یک از اندیسهای آرایه باشد.

بخش ۳-۸

۷- الگوریتمی بنویسید که بزرگترین کلید را در یک درخت جستجوی دودویی پیدا کند. الگوریتم را تحلیل نموده و نتایج را با استفاده از سمبلهای ترتیب نشان دهید.

۸- الگوریتمی بنویسید که با در نظر گرفتن تمام حالات ممکن، یک گره را از یک درخت جستجوی دودویی حذف کند. الگوریتم را تحلیل نموده و نتایج را با استفاده از سمبلهای ترتیب نشان دهید.

۹- الگوریتمی بنویسید که یک درخت ۲-۳ را از لیستی از کلیدها تولید نماید. الگوریتم را تحلیل نموده و نتایج را با استفاده از سمبلهای ترتیب نشان دهید.

۱۰- الگوریتمی بنویسید که تمامی کلیدهای موجود در یک درخت ۲-۳ را به همان ترتیب خودشان لیست کند. الگوریتم را تحلیل نموده و نتایج را با استفاده از سمبلهای ترتیب نشان دهید.

بخش ۴-۸

۱۱- یک استراتژی دیگر برای رفع مشکل تصادم، درهم‌سازی بسته است. در این استراتژی، تمامی عناصر در آرایه‌ای از باکتها (جدول درهم‌سازی) ذخیره می‌شوند. در هنگام وقوع یک تصادم، جدول برای باکت قابل دسترسی (آزاد) بعدی مورد جستجو قرار می‌گیرد. نشان دهید که درهم‌سازی بسته چگونه تصادم های موجود در نمونه مسئله شکل ۸-۸ را حل می‌کند. به درهم‌سازی بسته، رفع تصادم خطی نیز گرفته می‌شود.

۱۲- نقاط ضعف و قوت دواستراتژی رفع تصادم، درهم‌سازی باز و درهم‌سازی بسته را بررسی نمائید.

۱۳- الگوریتمی بنویسید که عنصری را از جدول درهم‌سازی حذف نماید. برای رفع تصادم از استراتژی درهم‌سازی بسته استفاده می‌شود.

بخش ۵-۸

۱۴- الگوریتم ۴-۸ را به گونه‌ای تغییر دهید که برای مقدار n فرد کار کند. نشان دهید که پیچیدگی زمانی آن برابراست با

اگر n زوج باشد $\frac{3n}{2} - 2$

اگر n فرد باشد $\frac{3n}{2} - \frac{3}{2}$

۱۵- اثبات قضیه ۸-۸ را کامل کنید. به عبارتی، نشان دهید که یک الگوریتم قطعی که کوچکترین و بزرگترین کلید را در یک آرایه n کلیدی، تنها با مقایسهٔ کلیدها پیدا می‌کند بایستی در بدترین حالت حداقل $3n - 3) / 2$ مقایسه برای مقادیر فرد n انجام دهد.

۱۶- الگوریتمی برای روش مطرح شده در بخش ۳-۵-۸ برای یافتن دومین کلید بزرگتر یک آرایه بنویسید.

۱۷- نشان دهید که در حالت کلی مجموع تعداد مقایسات مورد نیاز توسط روش مطرح شده در بخش ۳-۵-۸ برای یافتن دومین کلید بزرگتر در یک آرایهٔ معین برابراست با

$$T(n) = n + \lceil \lg n \rceil - 2$$

۱۸- با استفاده از استقراء نشان دهید که پیچیدگی زمانی بدترین حالت الگوریتم ۶-۸ (انتخاب با استفاده از میانه) تقریباً به صورت زیر است:

$$W(n) \leq 22n$$

تمرینات اضافی

۱۹- فرض کنید که S و T دو آرایهٔ n عنصری هستند که به صورت غیرنزولی مرتب شده‌اند. الگوریتمی بنویسید که میانهٔ تمامی 2n عنصر را با پیچیدگی زمانی $\theta(\lg n)$ پیدا کند.

فصل ۹

پیچیدگی محاسباتی و کنترل‌ناپذیری: مقدمه‌ای بر نظریهٔ NP

داستان زیر را درنظر بگیرید. فرض کنید که در یک شرکت مشغول به کار هستید و رئیس از شما درخواست الگوریتمی کارا برای حل یک مسئله بسیار مهم می‌کند. پس از روزها و یا حتی بیش از یک ماه تلاش، نمی‌توانید الگوریتمی کارا پیدا کنید و این موضوع را با شرمندگی به رئیس خود اطلاع می‌دهید. رئیس شما را تهدید به اخراج از شرکت می‌کند و می‌گوید که یک طراح باهوشتر را جایگزین شما خواهد کرد. شما در پاسخ می‌گوئید که شاید علت این امر، کند ذهنی من نباشد و الگوریتم کارایی اصلاً وجود نداشته باشد. رئیس یک فرصت یک ماههٔ دیگر به شما می‌دهد تا ثابت کنید که الگوریتمی کارا برای حل آن مسئله وجود ندارد. پس از تلاش شبانه‌روزی و بی‌خوابی‌کشیدنها باز هم موفق نمی‌شوید. شما نه توانسته‌اید الگوریتمی کارا پیدا کنید و نه اینکه عدم امکان وجود چنین الگوریتمی را ثابت کنید. در آستانهٔ اخراج از شرکت قرار می‌گیرید و به خاطر می‌آورید که برای ارائه یک الگوریتم کارا برای مسئله فروشندهٔ دوره‌گرد، حتی برخی از بزرگترین دانشمندان کامپیوتر هم نتوانسته‌اند الگوریتمی ارائه دهند که پیچیدگی زمانی بدترین حالت آن، بهتر از نمایی باشد. به علاوه، کسی هم نتوانسته است ثابت کند که ارائه چنین الگوریتمی غیرممکن است. به آخرین نقطه امید چشم می‌دوزید. اگر بتوانید ثابت کنید که یک الگوریتم کارا برای

مسئله شرکت منجر به تولید الگوریتمی کارا برای مسئله فروشندهٔ دوره‌گرد می‌شود، یعنی اینکه رئیس از شما چیزی را خواسته است که دانشمندان بزرگ علم کامپیوتر هم نتوانسته‌اند از عهده آن برآیند. از رئیس یک فرصت دیگر برای اثبات این موضوع درخواست می‌کنید. پس از تنها یک هفته تلاش ثابت می‌کنید که الگوریتم مسئله شرکت منجر به یافتن یک الگوریتم کارا برای مسئله فروشندهٔ دوره‌گرد می‌شود. حال به جای آنکه اخراج شوید، ترفیع می‌گیرید زیرا از هدررفتن سرمایهٔ شرکت جلوگیری می‌کنید.

آنچه گفته شد دقیقاً همان چیزی بود که دانشمندان کامپیوتر در ۲۵ سال گذشته انجام داده‌اند. ما نشان داده‌ایم که مسئله فروشندهٔ دوره‌گرد و هزاران مسئله دیگر دارای مشکل یکسانی هستند بطوری که اگر یک الگوریتم کارا برای هر یک از آنها پیدا شود، برای همه آنها الگوریتمی کارا خواهیم داشت. چنین الگوریتمی هرگز پیدا نشده است؛ البته هنوز هم کسی عدم امکان وجود آنرا ثابت نکرده است. چنین مسائلی، "NP-کامل" نامیده می‌شوند که مورد بحث این فصل می‌باشند. مسئله‌ای که نتوان برایش یک الگوریتم کارا پیدا کرد را "کنترل‌ناپذیر" گوئیم. در بخش ۱-۹ خواهیم گفت که معنای یک مسئله کنترل‌ناپذیر چیست. بخش ۲-۹ نشان می‌دهد که وقتی در حال بررسی کنترل‌ناپذیر بودن و یا نبودن یک مسئله هستیم، بایستی نسبت به انتخاب اندازه ورودی در الگوریتم، دقت لازم را به عمل آوریم. در بخش ۳-۹ به بررسی سه گروه کلی از مسائلی می‌پردازیم که می‌توان آنها را به عنوان کنترل‌ناپذیر دسته بندی کرد و بخش ۴-۹، نظریه مسائل NP و NP-کامل را بررسی می‌کند.

۱-۹ کنترل ناپذیری

زمانی یک مسئله در علم کامپیوتر کنترل‌ناپذیر نامیده می‌شود که کامپیوتر به سختی بتواند آن را حل نماید. همانطوری که مشاهده می‌کنید، این تعریف بسیار مبهم است. برای این که اصطلاح واقعی‌تری را بکار بگیریم، مفهوم "الگوریتم زمان- چندجمله‌ای" را معرفی می‌کنیم.

تعریف یک الگوریتم **زمان- چندجمله‌ای**، الگوریتمی است که حد بالای پیچیدگی زمانی بدترین حالت آن به یک تابع چندجمله‌ای از اندازه ورودیش محدود می‌شود. یعنی اگر n اندازه ورودی باشد، یک چندجمله‌ای p(n) وجود دارد بطوری که $W(n) \in O(p(n))$.

مثال ۱-۹ الگوریتمهای با پیچیدگی زمانی بدترین حالت زیر، زمان- چندجمله‌ای هستند.

$$۲n \qquad ۳n^{۳}+۴n \qquad ۵n+n^{۱۰} \qquad n\lg n$$

و الگوریتمهای با پیچیدگی زمانی بدترین حالت زیر، زمان- چندجمله‌ای نمی‌باشند.

$$۳^{n} \qquad ۲^{۰/۰۱n} \qquad ۲^{\sqrt{n}} \qquad n!$$

توجه داشته باشید که $n lg n$ یک چندجمله‌ای بر حسب n نیست. با وجود این، چون $n lg n < n^2$ توسط یک چندجمله‌ای n محدود شده است، یعنی اینکه یک الگوریتم با این پیچیدگی زمانی دارای معیار یک الگوریتم زمان - چندجمله‌ای می‌باشد.

در علم کامپیوتر، زمانی یک مسئله **کنترل‌ناپذیر** نامیده می‌شود که نتوان آن را با یک الگوریتم زمان- چندجمله‌ای حل نمود. ما تأکید می‌کنیم که کنترل ناپذیری از خصوصیات یک مسئله است، نه از خصوصیات هر یک از الگوریتمهای آن مسئله. برای آنکه یک مسئله را کنترل‌ناپذیر بنامیم نبایستی هیچ الگوریتم زمان- چندجمله‌ای برای حل آن پیدا شود. در واقع بدست آوردن یک الگوریتم زمان- چندجمله‌ای برای یک مسئله، آن را از کنترل‌ناپذیری خارج می‌کند. مثلاً الگوریتم brute_force برای مسئله ضرب ماتریس زنجیره‌ای (بخش ۴-۳)، یک الگوریتم زمان- غیرچندجمله‌ای است. همچنین الگوریتم تقسیم و غلبه ارائه شده در بخش ۴-۳ که از خاصیت بازگشتی استفاده می‌کند نیز چنین است. الگوریتم برنامه‌نویسی پویا (الگوریتم ۶-۳) که در بخش مذکور ارائه شد در $\theta(n^3)$ است. این مسئله کنترل‌ناپذیر نیست زیرا می‌توانیم آن را با یک الگوریتم زمان- چندجمله‌ای (الگوریتم ۵-۳) حل کنیم.

در فصل ۱ دیدیم که الگوریتمهای زمان- چندجمله‌ای معمولاً خیلی بهتر از الگوریتم‌های زمان- غیرچندجمله‌ای هستند. با نگاهی دوباره به جدول ۴-۱ می‌بینیم که اگر مدت زمان لازم برای پردازش دستورالعملهای مبنایی یک نانوثانیه باشد، آنگاه الگوریتمی با پیچیدگی زمانی n^3 می‌تواند یک نمونه به اندازهٔ ۱۰۰ را در یک میلی‌ثانیه انجام دهد؛ در حالیکه الگوریتمی با پیچیدگی زمانی 2^n برای انجام آن به میلیاردها سال زمان نیاز دارد.

نمونه‌های زیادی وجود دارد که در آنها الگوریتم زمان- غیرچندجمله‌ای بهتر از الگوریتم زمان- چندجمله‌ای عمل می‌کند. به عنوان مثال، اگر $n = 1,000,000$ باشد دراینصورت $2^{(0/000001)n} = 1024$؛ در حالیکه $n^{10} = 10^{60}$. علاوه بر این، بسیاری از الگوریتمهایی که پیچیدگی زمانی بدترین حالت آنها به صورت چندجمله‌ای نیست دارای زمان اجرایی کارایی برای بسیاری از نمونه‌های واقعی می‌باشند. و این حالت برای بسیاری از الگوریتمهای بک‌تراکینگ و شاخه‌وحد وجود دارد. در برخی موارد خاص، کارکردن با مسئله‌ای که برایش یک الگوریتم زمان- چندجمله‌ای پیدا شده است، کمی دشوارتر از مسئله‌ای است که برایش هیچ الگوریتمی پیدا نشده است. سه گروه کلی برای مسائل کنترل‌ناپذیر وجود دارد:

۱- مسائلی که برایشان الگوریتمهایی زمان- چندجمله‌ای پیدا شده است.

۲- مسائلی که کنترل‌پذیری آنها به اثبات رسیده است.

۳- مسائلی که کنترل‌ناپذیری آنها به اثبات نرسیده، اما هرگز الگوریتمهایی زمان- چندجمله‌ای برای آنها پیدا نشده است.

جالب اینکه بنظر می‌رسد بسیاری از مسائل علم کامپیوتر در گروه اول و یا سوم قرار دارند. به هنگام تعیین زمان- چندجمله‌ای بودن یک الگوریتم بایستی توجه کافی به اندازه ورودی داشته باشیم.

۹-۲ اندازه ورودی

تا بحال n را به عنوان اندازه ورودی الگوریتم می‌نامیدیم؛ چرا که n یک اندازهٔ منطقی برای میزان دادهٔ ورودی است. به عنوان مثال، در الگوریتمهای مرتب‌سازی، n (تعداد کلیدهایی که باید مرتب شوند) معیار خوبی برای میزان داده‌های ورودی است و به همین دلیل n را بعنوان اندازه ورودی در نظر گرفتیم. با وجود این نبایستی بدون تفکر، n را به عنوان اندازه ورودی یک الگوریتم معرفی کنیم. الگوریتم زیر تعیین می‌کند که آیا عدد صحیح مثبت n اول است یا خیر.

```
bool prime (int n)
{
  int i; bool switch;
  switch = true;
  i = 2;
  while (switch && i < n)
    if (n % i == 0)
      switch = false;
    else
      i++;
  return switch;
}
```

واضح است که پیچیدگی زمانی این الگوریتم در $\theta(n)$ است. با این حال، آیا این الگوریتم، یک الگوریتم زمان- چندجمله‌ای است یا خیر؟ پارامتر n، ورودی الگوریتم است ولی اندازه ورودی آن نمی‌باشد. یعنی هر مقدار n، یک نمونه از مسئله را تشکیل می‌دهد و این برخلاف الگوریتم‌های مرتب‌سازی است که مثلاً در آن n تعداد کلیدها و نمونه برابر n کلید است. اگر مقدار n به عنوان ورودی و نه اندازه ورودی تابع prime باشد، پس اندازهٔ ورودی آن چیست؟ پس از تعریف اندازهٔ ورودی به شکلی واقعی‌تر از آنچه که در بخش ۳-۱ ارائه نمودیم، به این سؤال برمی گردیم.

تعریف برای یک الگوریتم معین، **اندازهٔ ورودی** عبارت است از تعداد کاراکترهایی که آن الگوریتم برای نوشتن ورودی بکار می‌برد.

این تعریف مغایر با تعریف ارائه شده در بخش ۳-۱ نیست. برای شمارش کاراکترهایی که جهت نوشتن ورودی بکار می‌روند بایستی از چگونگی کدگذاری ورودی اطلاع داشته باشیم. فرض کنید که ورودی را در مبنای ۲ کدگذاری نمودیم. در اینصورت کاراکترهایی که برای کدگذاری استفاده می‌شوند، اعداد دودویی (بیت‌ها) هستند و تعداد کاراکترهایی که برای کدگذاری یک عدد صحیح مثبت x بکار می‌روند برابراست با $\lfloor \lg 31 \rfloor + 1$. به عنوان مثال $31 = 11111_2$ و $5 = 1 + \lfloor \lg 31 \rfloor$. برای سادگی، می‌گوئیم که تقریباً

lgx بیت برای کدگذاری یک عدد صحیح مثبت در مبنای ۲ بکار می‌رود. فرض کنید که از کدگذاری دودویی استفاده می‌کنیم و می‌خواهیم اندازه ورودی را برای الگوریتمی که n عدد صحیح مثبت را مرتب می‌کند، تعیین کنیم. اعداد صحیحی که باید مرتب شوند، ورودیهای الگوریتم هستند. بنابراین، اندازه ورودی برابر است با تعداد بیتهای مورد نیاز برای کدگذاری این اعداد، اگر بزرگترین عدد صحیح، L باشد و تعداد بیتهای لازم برای کدگذاری هر یک از این اعداد را همان تعداد بیتهای لازم برای کدگذاری L درنظر بگیریم، در اینصورت تقریباً $lg L$ بیت برای کدگذاری هر یک از آنها نیاز است. لذا اندازه ورودی برای n عدد صحیح در حدود $n lg L$ است. فرض کنید که از مبنای ۱۰ برای کدگذاری اعداد صحیح استفاده می‌کنیم. دراینصورت، کاراکترهای مورداستفاده برای کدگذاری، ارقام دهدهی هستند و در حدود $log L$ کاراکتر برای کدگذاری بزرگترین عدد بکار می‌رود و اندازه ورودی برای n عدد صحیح، تقریباً برابر با $n log L$ است. از آنجائیکه $n log L = (log ۲)(n lgL)$ است، لذا الگوریتم برحسب یکی از این اندازه‌های ورودی، زمان- چندجمله‌ای است اگر و فقط اگر برحسب دیگر اندازه‌های ورودی هم، زمان - چندجمله‌ای باشد.

در فصل‌های گذشته، برای سادگی، n (تعداد کلیدهایی که باید مرتب شوند) را برای الگوریتم‌های مرتب‌سازی به عنوان اندازه ورودی درنظر گرفتیم و با استفاده از آن نشان دادیم که آن الگوریتم‌ها، زمان- چندجمله‌ای هستند. اما اکنون که راجع به اندازه ورودی دقیق‌تر شدیم، آیا همچنان معتقدیم که آنها زمان- چندجمله‌ای هستند؟ در ادامه خواهیم دید که جواب مثبت است. وقتی که راجع به اندازه ورودی دقیق‌تر می‌شویم، بایستی راجع به تعریف پیچیدگی زمانی بدترین حالت نیز دقیق‌تر (از آنچه که در بخش ۱-۳-۱ آمده) شویم. تعریف دقیق آن بدین صورت است:

تعریف برای یک الگوریتم معین، $W(s)$ به عنوان حداکثر تعداد مراحل اجرای الگوریتم به ازای اندازه ورودی n تعریف می‌شود که به آن **پیچیدگی زمانی بدترین حالت** الگوریتم گوئیم.

یک **مرحله** می‌تواند یک مقایسه یا یک انتساب در ماشین و یا اگر بخواهیم مستقل از ماشین تحلیل کنیم، یک مقایسه یا انتساب یک بیت باشد.این تعریف با تعریف ارائه شده در بخش ۳-۱ مغایرت ندارد. در اینجا فقط دربارهٔ عمل مبنایی دقیق‌تر شدیم. براساس این تعریف، هر مرحله عبارت است از یک اجرای کامل عمل مبنایی. به جای n، از S برای اندازه ورودی استفاده کردیم زیرا (۱) پارامتر n همیشه معیاری برای اندازه ورودی الگوریتم‌ها نیست (برای مثال، در الگوریتم بررسی اعداد اول که در آغاز این بخش آورده شد) و (۲) وقتی که n معیاری برای اندازه ورودی باشد. نمی‌توان اندازه دقیقی برای آن بدست آورد. مطابق تعریفی که اخیراً ارائه نمودیم، بایستی تمام مراحل انجام شده توسط الگوریتم را بشماریم. اما این کار چگونه انجام می‌شود؟ الگوریتم ۳-۱ (مرتب سازی تبادلی) را درنظر بگیرید. برای سادگی، فرض می‌کنیم که همهٔ کلیدها اعداد صحیح مثبت هستند و دیگر فیلدی در رکوردها وجود ندارد. با نگاهی

مجدد به الگوریتم ۳-۱ درمی‌یابید که تعداد مراحل انجام شده برای افزایش حلقه‌ها و پرش‌ها به یک ثابت $c \times n^2$ بستگی دارد. اگر اعداد صحیح به اندازه کافی بزرگ باشند، نمی‌توان آنها را دریک مرحله توسط کامپیوتر مقایسه کرد و یا منتسب نمود. وضعیت مشابهی را در بحث محاسبات بر روی اعداد صحیح بزرگ (در بخش ۶-۲) مشاهده نمودیم. بنابراین، نبایستی تصور کنیم که عمل مقایسه یا انتساب در یک مرحله انجام می‌شود. برای آنکه تحلیل خود را مستقل از ماشین انجام دهیم، هر مقایسه یا انتساب یک بیت را به عنوان یک مرحله درنظر می‌گیریم. بنابراین، اگر L بزرگترین عدد صحیح باشد، حداکثر به تعداد $lg\ L$ مرحله برای مقایسه یک عدد صحیح با عدد صحیح دیگر و یا انتساب آنها نیاز داریم. در تحلیل الگوریتم ۳-۱ در بخش ۳-۱ و ۲-۷ دیدیم که مرتب سازی تبادلی در بدترین حالت، به تعداد $n(n-1)/2$ مرتبه عمل مقایسه و به تعداد $3n(n-1)/2$ مرتبه عمل انتساب را برای مرتب‌سازی n عدد صحیح مثبت انجام می‌دهد. بنابراین، حداکثر تعداد مراحل انجام شده توسط مرتب‌سازی تبادلی، از عبارت زیر بزرگتر نیست:

$$cn^2 + \frac{n(n-1)}{2} lg L + \frac{3n(n-1)}{2} lg L$$

می‌خواهیم از $s = n lg L$ به عنوان اندازه ورودی استفاده کنیم. در اینصورت

$$\begin{aligned} W(s) &= W(n lg L) \\ &\le cn^2 + \frac{n(n-1)}{2} lg L + \frac{3n(n-1)}{2} lg L \\ &< cn^2 (lg L)^2 + n^2 (lg L)^2 + 3n^2 (lg\ L)^2 \\ &< (c+4)(n lg L)^2 = (c+4)s^2 \end{aligned}$$

بدین ترتیب نشان داده‌ایم که مرتب سازی تبادلی برای وقتی که روی اندازه ورودی دقیق‌تر می‌شویم، همچنان زمان- چندجمله‌ای باقی می‌ماند. برای تمام الگوریتمها می‌توان نتایج مشابهی را بدست آورد. همچنین می‌توان نشان داد که الگوریتمهای زمان- غیرچندجمله‌ای (نظیر الگوریتم ۱۱-۳) همچنان زمان- غیرچندجمله‌ای باقی می‌مانند. بنابراین، مشاهده می‌کنید که وقتی n، به عنوان معیاری برای میزان داده‌های ورودی مطرح می‌شود، می‌توانیم نتایج درستی دربارهٔ زمان- چندجمله‌ای بودن یا نبودن یک الگوریتم بدست آوریم و برای این منظور، n را به عنوان اندازه ورودی درنظر می‌گیریم.

اکنون به الگوریتم بررسی عدد اول باز می‌گردیم. از آنجائیکه ورودی الگوریتم، مقدار n است، لذا اندازه ورودی عبارتست از تعداد کاراکترهایی که برای کدگذاری n بکار می‌روند. همانطوری که قبلاً گفتیم اگر از کدگذاری مبنای ۱۰ استفاده کنیم، به $\lfloor log n \rfloor + 1$ کاراکتر برای کدگذاری n نیاز داریم. مثلاً اگر n مساوی ۳۴۰ باشد، به ۳ رقم دهدهی برای کدگذاری آن نیاز است. بطور کلی اگر از کدگذاری مبنای ۱۰ استفاده کنیم و S را معادل $log n$ در نظر بگیریم، در این صورت S تقریباً برابر با اندازه ورودی است. در بدترین حالت، n-۲ گذر از حلقه در تابع prime وجود دارد. چون $n = 10^s$ است، لذا بدترین حالت تعداد گذرهای حلقه، حدود 10^s می‌باشد. از آنجائیکه تعداد کل مراحل، حداقل مساوی تعداد گذرها

از حلقه است، لذا پیچیدگی زمانی به صورت غیرچندجمله‌ای است. اگر از کدگذاری دودوی استفاده کنیم، حدود $lg\ n$ کاراکتر برای کدگذاری n لازم است که در اینصورت، $r = lg\ n$ تقریباً برابر با اندازه ورودی است و تعداد گذرهای حلقه نیز حدوداً برابر ۳ می‌باشد. می‌بینیم که باز هم پیچیدگی زمانی به صورت غیرچندجمله‌ای است. اما اگر از کدگذاری یکتایی استفاده کنیم، برای کدگذاری عدد n، به n کاراکتر نیاز داریم. مثلاً کدگذاری عدد ۷ به صورت ۱۱۱۱۱۱۱ می‌شود. با این کدگذاری، الگوریتم بررسی عدد اول یک پیچیدگی زمانی خطی پیدا می‌کند. بنابراین، اگر از سیستم کدگذاری معقول منحرف شویم (کدگذاری یکتایی یک روش عقلانی نیست)، نتایج ما تغییر خواهند کرد.

در الگوریتمهایی نظیر الگوریتم بررسی عدد اول، n را یک **بزرگی** در ورودی می‌نامیم. الگوریتم‌های دیگری را دیدیم که پیچیدگی زمانی آنها بر حسب بزرگی به صورت چندجمله‌ای است اما بر حسب اندازهٔ آنها چندجمله‌ای نیست. پیچیدگی زمانی الگوریتم ۷-۱ برای محاسبه عنصر nام دنباله فیبوناچی در $\theta(n)$ است. چون n یک بزرگی در ورودی است و $lg\ n$ اندازه ورودی را نشان می‌دهد، لذا پیچیدگی زمانی الگوریتم ۷-۱ برحسب بزرگی به صورت خطی است اما برحسب اندازه به صورت نمایی می‌باشد. پیچیدگی زمانی الگوریتم ۲-۳ برای محاسبه ضریب دوجمله‌ای در $\theta(n^2)$ است. چون n یک بزرگی برای ورودی است و lg n اندازه ورودی را نشان می‌دهد، لذا پیچیدگی زمانی الگوریتم ۲-۳ برحسب بزرگی به صورت مربعی است اما برحسب اندازه به صورت نمایی می‌باشد. پیچیدگی زمانی الگوریتم برنامه‌نویسی پویا برای مسئله کوله‌پشتی ۱-۰ که در بخش ۴-۴-۴ بحث شد، در $\theta(n\ W)$ است. در این الگوریتم، n معیاری برای اندازه ورودی است زیرا n معادل تعداد کالاها در ورودی مسئله است. با وجود این، W یک بزرگی است زیرا ماکزیمم ظرفیت کوله پشتی می‌باشد و lg W معیاری برای اندازه W می‌باشد. پیچیدگی زمانی این الگوریتم، برحسب بزرگی و اندازه ورودی به صورت چندجمله‌ای است اما بر حسب اندازه ورودی (به تنهایی) به صورت نمایی می‌باشد.

الگوریتمی که حد بالایی پیچیدگی زمانی بدترین حالت آن محدود به یک تابع چندجمله‌ای از اندازه بزرگیهایش باشد، **زمان- شبه‌چندجمله‌ای** نامیده می‌شود. چنین الگوریتمی اغلب مفید می‌باشد، زیرا تنها زمانی کارا نیست که با نمونه‌های شامل اعداد بسیار بزرگی مواجه شوند. چنین نمونه‌هایی ممکن است در کاربردهای ما مناسب نباشند. مثلاً در مسئله کوله‌پشتی ۱-۰، ممکن است اغلب با مواردی سروکار داشته باشید که W خیلی بزرگ نباشد.

۳-۹ سه گروه کلی از مسائل

در اینجا می‌خواهیم سه گروه کلی از مسائل را تا آنجا که به بحث کنترل‌ناپذیری مربوط می‌شود، بررسی کنیم.

۱-۳-۹ مسائلی برای الگوریتم‌های زمان- چندجمله‌ای

هر مسئله‌ای که برایش یک الگوریتم با پیچیدگی زمانی چندجمله‌ای پیدا کرده‌ایم، در این دسته قرار

می‌گیرد. الگوریتم‌هایی $\theta(n \lg n)$ برای مرتب‌سازی، $\theta(\lg n)$ برای جستجوی یک آرایهٔ مرتب شده، $\theta(n^{2/3})$ برای ضرب ماتریس، $\theta(n^3)$ برای ضرب ماتریس زنجیره‌ای و... پیدا نموده‌ایم. چون n معیاری برای مقدار داده‌ها در ورودیهای این الگوریتمها است، همگی آنها زمان- چندجمله‌ای می‌باشند. این لیست همچنان ادامه دارد.الگوریتمهایی وجود دارند که برای بسیاری از این مسائل، زمان- چندجمله‌ای نمی‌باشند. قبلاً ذکر کردیم که در مورد الگوریتم ضرب ماتریس زنجیره‌ای چنین است. دیگر مسائلی که برایشان الگوریتمهایی زمان- چندجمله‌ای ارائه نمودیم، اما الگوریتمهای brut_force برای آنها غیرچندجمله‌ای هستند، عبارتند از مسئله کوتاهترین مسیرها، مسئله درخت جستجوی دودویی بهینه و مسئله کوچکترین درخت پوشا.

۲-۳-۹ مسائلی که کنترل ناپذیری آنها ثابت شده است.

دو نوع از مسائل در این گروه قرار می‌گیرند. نوع اول مسائلی هستند که به یک مقدار غیرچندجمله‌ای از خروجی نیاز دارند. مسئله تعیین چرخه‌های هامیلتونی در بخش ۶-۵ را به خاطر آورید. اگر از هر رأس به رأس دیگر یک لبه وجود داشت، آنگاه $n-1$ چرخه به وجود می‌آمد. برای حل این مسئله، الگوریتم می‌بایست تمامی این چرخه‌ها را نمایش دهد که این درخواست، چندان معقول به نظر نمی‌رسید. در فصل ۵ اشاره داشتیم به این نکته که ما مسائل را طوری حل می‌کنیم که تمامی جوابهای ممکن را تولید کند، چراکه دراینصورت می‌توانستیم با اندکی تغییر در آن، الگوریتمی طراحی کنیم که تنها یک جواب را برگرداند. بدیهی است که مسئله چرخه‌های هامیلتونی که تنها یک جواب را درخواست می‌کند، از این نوع مسائل نیست. اگرچه شناخت این نوع کنترل‌ناپذیری مهم است، اما اینگونه مسائل مشکلی ایجاد نمی‌کنند.

نوع دوم کنترل ناپذیری زمانی اتفاق می‌افتد که درخواستهای ما معقول است (یعنی زمانی که ما تقاضای یک مقدار غیرچندجمله‌ای از خروجی نمی‌کنیم) و می‌توانیم ثابت کنیم که مسئله را نمی‌توان در یک زمان چندجمله‌ای حل نمود. تعدادی از این مسائل مشخص شده‌اند. اولین آنها، مسائل غیرقطعی می‌باشند. این مسائل «غیرقطعی» نامیده می‌شوند زیرا می‌توان ثابت کرد که الگوریتمی برای حل آن نمی‌تواند وجود داشته باشد. شناخته‌شده‌ترین این مسائل، مسئله Halting(توقف) است. در این مسئله، هر الگوریتمی را با هر ورودی درنظر می‌گیریم و تعیین می‌کنیم که با کدام ورودی، الگوریتم متوقف می‌شود. در سال ۱۹۳۶، آلن تورینگ نشان داد که این مسئله غیرقطعی است. در سال ۱۹۵۳، Grzegorczyk یک مسئله قطعی کنترل‌ناپذیر ارائه کرد. با وجود این، این مسائل «بطور مصنوعی» دارای خواص ویژه‌ای بودند. در اوایل دهه ۱۹۷۰ ثابت شد که برخی از مسائل تصمیم‌گیری قطعی، کنترل‌ناپذیرند. خروجی مسئله تصمیم‌گیری، جواب سادهٔ «بله» یا «خیر» است؛ لذا میزان خروجی درخواست شده معقول می‌باشد. یکی از شناخته شده‌ترین مسائل در این زمینه، ریاضیات Presburger است که کنترل‌ناپذیری آنها در سال ۱۹۷۴ توسط Fischer و Rabin ثابت شده است. تمام مسائلی که تا امروز کنترل‌ناپذیری آنها مشخص شده است، عدم حضورشان در مجموعهٔ NP نیز به اثبات رسیده است. با وجود این، بسیاری از مسائلی که بنظر می‌رسند کنترل‌ناپذیر باشند، در مجموعهٔ NP قرار دارند.

۹-۳-۳ مسائلی که کنترل ناپذیری آنها ثابت نشده و تابحال الگوریتم‌هایی زمان- چندجمله‌ای برای آنها پیدا نشده است.

این گروه شامل مسائلی است که هرگز برای آنها الگوریتمی زمان- چندجمله‌ای پیدا نشده، اما هنوز هم کسی ثابت نکرده که چنین الگوریتمی غیرممکن است. چنین مسائلی به وفور یافت می‌شوند.
بعنوان مثال، اگر مسئله مربوط به درخواست تنها یک جواب باشد، در اینصورت مسئله کوله پشتی ۰-۱، مسئله فروشندهٔ دوره‌گرد، مسئله مجموع زیرمجموعه‌ها، مسئله m-رنگ برای $m \geq 3$ مسئله چرخه‌های هامیلتونی و مسئله استنتاج ربایشی در شبکهٔ فرضی، همگی در این دسته قرار می‌گیرند. ما الگوریتم‌های شاخه و حد، یک‌تراکینگ و الگوریتم‌هایی دیگر برای این مسائل پیدا کرده‌ایم که برای نمونه‌های بزرگ کارا می‌باشند. بدین معنی که یک چندجمله‌ای برحسب n وجود دارد که تعداد اجرای عمل مبنایی را وقتی که نمونه از زیرمجموعه‌ای محدود انتخاب می‌شوند، محدود می‌کند. با وجود این، چنین چندجمله‌ای برای مجموعهٔ تمام نمونه‌ها وجود ندارد. برای نشان دادن این موضوع بایستی یک توالی نامتناهی از نمونه‌ها را بیابیم که برایش هیچ چندجمله‌ای برحسب n وجود نداشته باشد که تعداد اجرای عمل مبنایی را محدود کند. به خاطر آورید که این کار را برای الگوریتمهای یک‌تراکینگ در فصل ۵ انجام دادیم. یک ارتباط نزدیک و جالبی بین مسائل این گروه وجود دارد. بیان چنین ارتباطی، هدف ما در بخش بعد می‌باشد.

۹-۴ نظریهٔ NP

اگر خودمان را به مسائل تصمیم‌گیری محدود کنیم، ارائه این نظریه راحت‌تر خواهد بود. همانطوریکه قبلاً نیز اشاره کردیم، خروجی مسئله تصمیم‌گیری، جواب سادهٔ «بله» یا «خیر» می‌باشد. تاکنون، وقتی که این مسائل را مطرح می‌کردیم(در فصلهای ۳، ۴، ۵ و ۶)، آنها را به عنوان مسائل بهینه‌سازی معرفی می‌نمودیم؛ بدین معنا که خروجی یک جواب بهینه است. هر مسئله بهینه‌سازی دارای یک مسئله تصمیم‌گیری متناظر با خودش است که در مثالهای زیر نشان خواهیم داد.

مثال ۹-۲ مسئله فروشندهٔ دوره‌گرد

یک گراف وزن‌دار و جهت‌دار مفروض است. همانطوریکه می‌دانید، یک تور در چنین گرافی عبارت است از مسیری که از یک رأس شروع می‌شود و تمام رأسهای دیگر گراف را دقیقاً یکبار ملاقات می‌کند و دوباره به همان رأس آغازین ختم می‌شود. **مسئله بهینه‌سازی فروشندهٔ دوره‌گرد**، تعیین یک تور با حداقل مجموع وزن لبه‌ها می‌باشد.
مسئله تصمیم‌گیری فروشنده دوره‌گرد تعیین می‌کند که آیا برای یک عدد مفروض مثبت d، توری وجود دارد که مجموع وزن آن بیشتر از d نباشد یا خیر. این مسئله دارای پارامترهایی نظیر مسئله بهینه‌سازی فروشندهٔ دوره‌گرد است باضافهٔ یک پارامتر اضافی d.

مثال ۳-۹ مسئله کوله‌پشتی ۰-۱

هدف در **مسئله بهینه‌سازی کوله‌پشتی ۰-۱** تعیین حداکثر ارزش کالاهایی است که می‌توانند در یک کوله‌پشتی گذارده شوند. هر کالا دارای وزن و ارزشی معین است و یک وزن کل W وجود دارد که کوله‌پشتی قادر به تحمل بیش از آن نیست. **مسئله تصمیم‌گیری کوله پشتی ۰-۱** تعیین می‌کند که آیا این امکان وجود دارد که کوله‌پشتی را طوری پر کنیم که وزن کل آن بیشتر از W نشود و در عین حال، ارزش کل کالاها، حداقل برابر P باشد. پارامترهای این مسئله همانند مسئله بهینه سازی کوله‌پشتی ۰-۱ است باضافهٔ اضافی P.

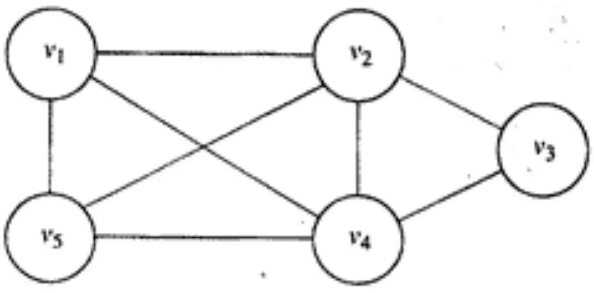
مثال ۴-۹ مسئله رنگ آمیزی گراف

هدف در مسئله **بهینه سازی رنگ‌آمیزی گراف** تعیین حداقل تعداد رنگهای لازم برای رنگ‌آمیزی یک گراف به شیوه‌ای است که هیچ دو رأس مجاور همرنگ نباشند. **مسئله تصمیم‌گیری رنگ‌آمیزی گراف** تعیین می‌کند که آیا یک روش رنگ‌آمیزی وجود دارد که از حداکثر m رنگ استفاده کند و هیچ دو گرهٔ مجاور همرنگ نباشند. پارامترهای این مسئله همانند بهینه‌سازی رنگ‌آمیزی گراف است باضافهٔ پارامتر اضافی m.

مثال ۵-۹ مسئله کلیک (Clique)

یک **کلیک** در گراف بدون جهت $G = (V, E)$ عبارت است از یک زیرمجموعهٔ W از V، که هر رأس W با تمامی رئوس دیگر W مجاور هم باشند. برای گراف شکل ۱-۹، $\{V_2, V_3, V_4\}$ یک کلیک است؛ در حالیکه $\{V_1, V_2, V_3\}$ یک clique نیست زیرا V_1 مجاور V_3 نیست. یک **کلیک بیشینه**، کلیلی با حداکثر اندازه می‌باشد. تنها کلیک بیشینه در گراف شکل ۱-۹ عبارت است از $\{V_1, V_2, V_4, V_5\}$.

هدف در **مسئله بهینه سازی کلیک** تعیین اندازهٔ کلیک بیشینه برای یک گراف مفروض است و **مسئله تصمیم‌گیری کلیک** تعیین می‌کند که آیا یک کلیک شامل حداقل k رأس وجود دارد یا خیر. پارامترهای این مسئله همانند مسئله بهینه‌سازی کلیک است باضافهٔ پارامتر اضافی k.



شکل ۱-۹ کلیک بیشنه عبارت است از $\{V_1, V_2, V_4, V_5\}$.

تاکنون هیچگونه الگوریتم زمان- چندجمله‌ای برای مسئله تصمیم‌گیری و مسئله بهینه‌سازی در هر یک از این مثالها پیدا نکرده‌ایم. با وجود این، اگر می‌توانستیم یک الگوریتم زمان- چندجمله‌ای برای مسئله بهینه‌سازی در هر یک از مثالهای فوق پیدا کنیم، الگوریتم زمان- چندجمله‌ای برای مسئله تصمیم‌گیری متناظر آن نیز یافت می‌شد؛ چون یک جواب برای یک مسئله بهینه‌سازی منجر به تولید یک جواب برای مسئله تصمیم‌گیری متناظر آن می‌شود. بعنوان مثال، اگر بدانیم که وزن کل یک تور بهینه برای یک نمونه از مسئله بهینه‌سازی فروشندهٔ دوره‌گرد ۱۲۰ است، جواب مسئله تصمیم‌گیری متناظر با آن برای $d \geq$ ۱۲۰ «بله»، و در غیراینصورت «خیر» می‌باشد. بطور مشابه اگر بدانیم که ارزش بهینه برای نمونه‌ای از مسئله بهینه سازی کوله پشتی ۱-۰، $۲۳۰ است، جواب مسئله تصمیم‌گیری متناظر با آن برای $p \leq$ $۲۳۰ «بله»، و در غیراینصورت «خیر» می‌باشد. با توجه به این مطالب، در ابتدا این نظریه را فقط برای مسائل تصمیم‌گیری ارائه می‌دهیم و ثابت می‌کنیم که برای بسیاری از مسائل تصمیم‌گیری (که شامل مسائل مثالهای قبل می‌باشد)، یک الگوریتم زمان- چندجمله‌ای منجر به تولید یک الگوریتم زمان- چندجمله‌ای برای مسئله بهینه سازی متناظر آن می‌شود.

۹-۴-۱ مجموعه‌های P و NP

در ابتدا مجموعه‌ای از مسائل تصمیم‌گیری را درنظر می‌گیریم که می‌توان آنها را با الگوریتم‌های زمان- چندجمله‌ای حل کرد.

تعریف p مجموعه‌ای است از تمامی مسائل تصمیم‌گیری که می‌توان آنها را توسط الگوریتمهای زمان- چندجمله‌ای حل کرد.

چه مسائلی در مجموعه p قرار دارند؟ یقیناً تمام مسائل تصمیم‌گیری که الگوریتمهای زمان- چندجمله‌ای برایشان پیدا شده در p قرار می‌گیرند. به عنوان مثال، مسئله تعیین وجود و یا عدم وجود یک کلید در یک آرایه، مسئله تعیین وجود یا عدم وجود یک کلید در یک آرایهٔ مرتب، و مسائل تصمیم‌گیری متناظر با مسائل بهینه‌سازی مطرح شده در فصلهای ۳ و ۴ که برایشان الگوریتمهای زمان- چندجمله‌ای پیدا شده است، همگی در p قرار دارند. اما آیا می‌توان یک مسئله تصمیم‌گیری که برایش الگوریتم زمان- چندجمله‌ای پیدا نشده است را در مجموعهٔ p قرار داد؟ برای مثال، آیا مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد در p قرار دارد؟ با وجود اینکه هنوز هیچ کس یک الگوریتم زمان- چندجمله‌ای برای حل این مسئله پیدا نکرده است، کسی هم ثابت نکرده که چنین الگوریتمی برای حل این مسئله وجود ندارد. بنابراین، این مسئله احتمالاً در p قرار دارد. برای آنکه بگوئیم یک مسئله تصمیم‌گیری در p قرار دارد یا خیر، باید ثابت کنیم که نمی‌توان یک الگوریتم زمان- چندجمله‌ای برای حل آن ارائه داد و این موضوع هنوز برای مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد ثابت نشده است. در مورد مثالهای ۲-۹ تا ۵-۹ نیز چنین است.

چه مسائل تصمیم‌گیری در p وجود ندارند؟ ما نمی‌دانیم که مسائل تصمیم‌گیری مطرح شده در مثالهای ۲-۹ تا ۵-۹ در p قرار دارند. بعلاوه، هزاران مسئله تصمیم‌گیری نیز در این گروه جای می‌گیرند، به عبارتی، ما نمی‌دانیم که آیا آنها در p قرار دارند یا خیر. در واقع، تعداد نسبتاً کمی از مسائل تصمیم‌گیری وجود دارند که مطمئن هستیم در p وجود ندارند. اینها مسائل تصمیم‌گیری هستند که ثابت کرده‌ایم برایشان الگوریتم‌های زمان- چندجمله‌ای وجود ندارد. چنین مسائلی را در بخش ۲-۳-۹ مطرح نموده‌ایم.

در اینجا به معرفی مجموعه وسیع‌تری از مسائل تصمیم‌گیری می‌پردازیم که شامل مسائل مثالهای ۲-۹ تا ۵-۹ می‌شود. قبل از این تعریف، اجازه دهید مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد را بیشتر مورد بررسی قرار دهیم. فرض کنید شخصی ادعا کرده که برای نمونه‌ای از این مسئله، جواب «بله» است. یعنی اظهار کرده که برای یک گراف و عدد d، توری وجود دارد که وزن کل آن تور بیشتر از d نمی‌باشد. منطقی است که از آن شخص بخواهیم ادعایش را با ایجاد آن تور ثابت کند. اگر شخص توری را تولید کرد، ما می‌توانیم الگوریتمی بنویسیم که مشخص کند آیا وزن تور او بزرگتر از d است یا خیر. ورودی این الگوریتم، گراف G، فاصله d، و رشته S است که ادعا شده یک تور با وزن کمتر از d می‌باشد.

```
bool verify (weighted_digraph G,
             number d,
             claimed_tour S)
{
  if (S is a tour && the total weight of edges in S is <= d)
    return true;
  else
    return false;
}
```

این الگوریتم در ابتدا بررسی می‌کند که آیا S واقعاً یک تور است یا خیر. اگر S یک توربود، آنگاه وزنهای روی تور را با هم جمع می‌کند. اگر مجموع اوزان بزرگتر از d نباشد، جواب 'true' را برمی‌گرداند؛ بدین معنا که جواب این مسئله تصمیم‌گیری، «بله» است. اما اگر S یک تور نباشد و یا اینکه مجموع اوزان بزرگتر از d باشد، الگوریتم، جواب 'false' را برمی‌گرداند که مفهومش این است که تور ادعا شده توری نیست که مجموع اوزانش کمتر از d باشد و البته بدین معنا نیست که چنین توری وجود ندارد زیرا ممکن است تور دیگری وجود داشته باشد که وزن کل آن کمتر از d باشد.

به عنوان تمرین، الگوریتم را بصورت واقعی‌تر پیاده‌سازی نموده و نشان دهید که آن زمان- چندجمله‌ای است. با توجه به مطالب فوق نمی‌توانیم ثابت کنیم که جواب مسئله تصمیم‌گیری ما در زمان چندجمله‌ای، «خیر» می‌باشد. این یکی از خصوصیات اثبات شدنی زمان- چندجمله‌ای است که مربوط به مسائل موجود در مجموعهٔ NP است و بعداً آن را تعریف می‌کنیم. این موضوع بدین معنا نیست که این مسائل حتماً می‌توانند در یک زمان چندجمله‌ای حل گردند. وقتی ثابت می‌کنیم که یک تور انتخابی، وزنی بیشتر از d ندارد، زمانی که برای یافتن آن تور صرف شده را درنظر می‌گیریم و فقط می‌گوئیم

که بخش اثبات، یک زمان چندجمله‌ای صرف می‌کند. برای آنکه موضوع قابلیت اثبات زمان- چندجمله‌ای را به صورت واقعی‌تری بیان کنیم، مفهوم **الگوریتم غیرقطعی** را تعریف می‌کنیم. می‌توان چنین الگوریتمی را ترکیبی از دو مرحله مجزای زیر دانست:

۱- **مرحله حدس (غیرقطعی):** این مرحله با داشتن یک نمونه از یک الگوریتم، رشتهٔ S را به سادگی تولید می‌کند. این رشته را می‌توان بعنوان یک جواب حدسی نمونه دانست. اگرچه آن می‌تواند تنها یک رشته بی‌معنی باشد.

۲- **مرحله اثبات (قطعی):** نمونه و رشتهٔ S بعنوان ورودی این مرحله می‌باشند. لذا این مرحله، طبق یک روش قطعی یا (۱) سرانجام با یک خروجی 'True' متوقف می‌شود؛ بدین معنا که جواب این نمونه «بله» است و یا (۲) با یک خروجی 'false' متوقف می‌شود و یا (۳) اصلاً متوقف نمی‌شود (یعنی درون یک حلقه نامتناهی گیر می‌کند.) در دو حالت اخیر، ثابت نشده که جواب این نمونه، «بله» است.

تابع verify مرحله اثبات را برای مسئله فروشندهٔ دوره‌گرد انجام می‌دهد. توجه داشته باشید آن، ذاتاً یک الگوریتم قطعی است و مرحله حدس است که غیرقطعی می‌باشد. این مرحله **غیرقطعی** نامیده می‌شود زیرا برای آن دستورالعملهای منحصربفرد گام به گامی مشخص نشده است. در عوض، در این مرحله، به ماشین این اجازه داده می‌شود که با یک روش دلخواه هر رشته‌ای را تولید کند. مرحلهٔ غیرقطعی، روشی واقعی برای حل یک مسئله تصمیم‌گیری نیست. حتی با وجود این که واقعاً از یک الگوریتم غیرقطعی برای حل یک مسئله استفاده نمی‌کنیم، می‌گوئیم که الگوریتم غیرقطعی، یک مسئله تصمیم‌گیری را "حل می‌کند" اگر

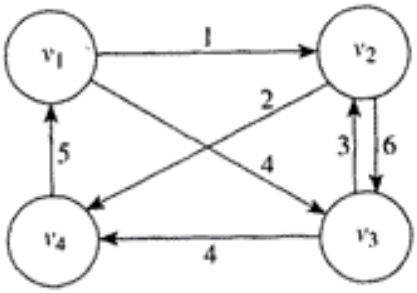
۱- برای هر نمونه‌ای که جوابش «بله» باشد، رشته‌ای به نام S وجود داشته باشد که مرحله اثبات، جواب 'true' را برایش برگرداند.

۲- برای هر نمونه‌ای که جوابش «خیر» باشد، رشته‌ای وجود نداشته باشد که مرحله اثبات، جواب 'true' را برایش برگرداند.

جدول زیر نتایج برخی از رشته‌های ورودی S به تابع verify برای وقتی که نمونه، گراف شکل ۲-۹ است و d برابر ۱۵ می‌باشد را نشان می‌دهد.

S	Output	Reason
$[v_1, v_2, v_3, v_4, v_1]$	False	Total weight is greater than 15
$[v_1, v_4, v_2, v_3, v_1]$	False	S is not a tour
#@12*&%a$_1$\	False	S is not a tour
$[v_1, v_3, v_2, v_4, v_1]$	True	S is a tour with total weight no greater than 15

شکل ۲-۹ وزن کل تور $[v_1, v_2, v_3, v_4, v_1]$بیشتر از ۱۵ نیست.



سومین ورودی نشان می‌دهد که S می‌تواند یک رشته بی‌معنی باشد (همانطوریکه قبلاً توضیح داده شد). بطورکلی، اگر جواب نمونه‌ای خاص، «بله» باشد، تابع verify مقدار 'true' را وقتی برمی‌گرداند که یکی از تورها با مجموع اوزان کمتراز d، بعنوان ورودی باشد. بنابراین، شرط اول برای یک الگوریتم غیرقطعی فراهم شده است. از طرفی دیگر، تابع verify تنها زمانی 'true' رابرمی‌گرداند که یک تور با وزن کل کمتر از d، به عنوان ورودی باشد. بنابراین، اگر جواب یک نمونه «خیر» باشد، تابع verify برای هر مقدار S مقدار 'true' را برنمی‌گرداند؛ یعنی اینکه در این صورت شرط ۲ فراهم شده است. یک الگوریتم غیرقطعی که در مرحله حدس، رشته‌ها را تولید و در مرحله اثبات، تابع verify را فراخوانی می‌کند، مسئله تصمیم‌گیری فروشنده دوره‌گرد را «حل می‌کند».

تعریف یک **الگوریتم غیرقطعی زمان- چندجمله‌ای**، الگوریتمی غیرقطعی است که مرحله اثبات آن یک الگوریتم زمان- چندجمله‌ای است.

تعریف **NP** مجموعه‌ای از تمامی مسائل تصمیم‌گیری است که می‌توانند توسط الگوریتم‌های غیرقطعی زمان- چندجمله‌ای حل شوند.

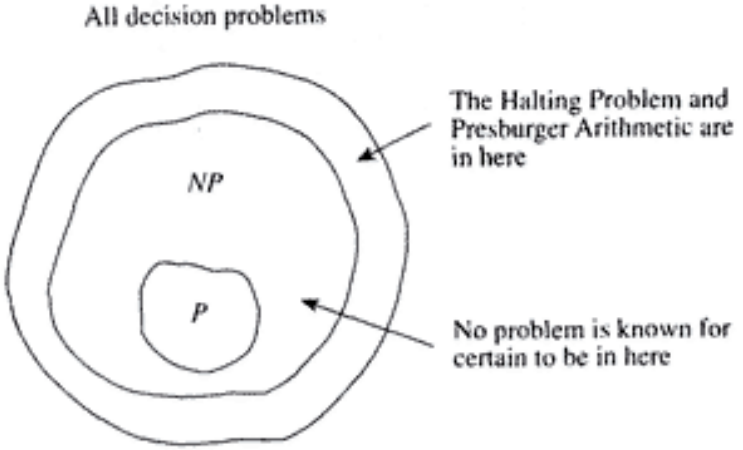
لغت NP مبین اصطلاح «چندجمله‌ای غیرقطعی» (Nondeterministic polynomial) است. برای آنکه مسئله‌ای در مجموعهٔ NP قرار گیرد، بایستی الگوریتمی وجود داشته باشد که مرحله اثبات را در یک زمان چندجمله‌ای انجام دهد. چون برای مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد چنین است، لذا این مسئله در NP قرار دارد. لازم به توضیح است که این بدین معنا نیست که ما حتماً یک الگوریتم زمان- چندجمله‌ای برای حل آن مسئله داریم. در واقع، در حال حاضر ما چنین الگوریتمی را برای مسئله تصمیم‌گیری فروشنده دوره‌گرد نداریم. اگر جواب نمونه‌ای خاص از آن مسئله «بله» باشد، ممکن است قبل از آنکه تابع verify جواب 'true' رابرگرداند، تمامی تورها را در مرحله غیرقطعی آزمایش کنیم. اگر از هر رأس به هر رأس دیگر یک لبه وجود داشته باشد،آنگاه $(n-1)!$ تور وجود دارد. بنابراین، اگر تمامی تورها آزمایش شوند، جواب را نمی‌توان در یک زمان چندجمله‌ای یافت. بعلاوه، اگر جواب یک نمونه «خیر» باشد، حل این مسئله

با این تکنیک مستلزم این است که تمامی تورها مورد بررسی قرار گیرند. هدف از طرح الگوریتمهای غیرقطعی و NP، دسته بندی الگوریتم‌ها است. معمولاً الگوریتم‌های بهتری برای حل واقعی یک مسئله وجود دارند تاالگوریتمهایی که رشته‌ها را تولید و اثبات می‌کنند. بعنوان مثال، الگوریتم شاخه و حد برای مسئله فروشندهٔ دوره‌گرد (الگوریتم ۳-۶) تورها را تولید می‌کند اما از تولید بسیاری از تورها با استفاده از تابع تحدید جلوگیری می‌کند. بنابراین، این الگوریتم بهتر از الگوریتمی است که کورکورانه تورها را تولید می‌نماید.

چه مسائل تصمیم‌گیری دیگری در NP وجود دارند؟ در تمرینات از شما خواسته شده که ثابت کنید مسائل تصمیم‌گیری در مثالهای ۲-۹ تا ۵-۹ همگی در NP هستند. بعلاوه، هزاران مسئله دیگر وجود دارند که هنوز کسی نتوانسته آنها را توسط الگوریتم‌های زمان- چندجمله‌ای حل کند اما ثابت شده که در مجموعهٔ NP قرار دارند زیرا برایشان الگوریتم‌های غیرقطعی زمان- چندجمله‌ای ارائه شده است. هر مسئله‌ای که در P باشد، در NP نیز هست. این موضوع تا حدودی صحیح است زیرا هر مسئله‌ای در P را می‌توان توسط یک الگوریتم زمان- چندجمله‌ای حل کرد. از آنجائیکه این الگوریتم با جواب «بله» یا «خیر» مسئله را حل می‌کند، لذا ثابت می‌کند که برای هر رشته ورودی S جواب «بله» است (برای نمونه‌ای که جوابش «بله» است).

چه مسائل تصمیم‌گیری در NP نیست؟ تنها مسائل تصمیم‌گیری که ثابت شده‌اند در NP نیستند، همان مسائلی هستند که کنترل‌ناپذیری آنها ثابت شده است. یعنی ثابت شده است که مسئله توقف، محاسبات Presburger و مسائل دیگری که در بخش ۲-۳-۹ بحث شدند. در NP قرار ندارند.

شکل ۳-۹ مجموعه‌ای از تمامی مسائل تصمیم‌گیری را نشان می‌دهد. توجه کنید که در این شکل، NP شامل زیرمجموعهٔ P می‌باشد. اما ممکن است این طور نباشد یعنی هیچ کسی تابحال ثابت نکرده است که *مسئله‌ای وجود دارد که در NP قرار دارد ولی در P نیست*. لذا مجموعهٔ NP - P ممکن است تهی باشد. در واقع، این سؤال که آیا P مساوی NP است یا خیر، یکی از سؤالات مهم و اساسی در علم کامپیوتر



شکل ۳-۹ مجموعهٔ تمامی مسائل تصمیم‌گیری.

می‌باشد. اهمیت این سؤال بدین دلیل است که بسیاری از مسائل تصمیم‌گیری مطرح شده، در NP قرار دارند. بنابراین اگر $P=NP$ باشد، برای بسیاری از مسائل تصمیم‌گیری ناشناخته، الگوریتم‌هایی زمان- چندجمله‌ای خواهیم داشت.

برای آنکه نشان دهیم $P\neq NP$ است، باید مسئله‌ای را پیدا کنیم که در NP باشد ولی در P نباشد و برای آنکه نشان دهیم $P=NP$ است، بایستی یک الگوریتم زمان- چندجمله‌ای برای هر مسئله در NP پیدا کنیم. در ادامه، خواهیم دید که این مورد اخیر را می‌توان تا حد زیادی ساده‌تر کرد یعنی نشان می‌دهیم که تنها لازم است یک الگوریتم زمان- چندجمله‌ای برای فقط یکی از مسائل یک گروه بزرگ پیدا کنیم. معذلک، بسیاری از محققان شک دارند که P برابر NP است.

۹-۴-۲ مسائل NP-کامل

ممکن است مسائل مطرح شده در مثالهای ۹-۲ تا ۹-۵ همگی به یک اندازه دشوار نباشند. بعنوان مثال، الگوریتم برنامه‌نویسی پویای ما برای مسئله فروشندهٔ دوره‌گرد(الگوریتم ۱۱-۳) در بدترین حالت، $\theta(n^2 3^n)$ است. از طرفی، الگوریتم برنامه‌نویسی پویا برای مسئله کوله‌پشتی ۱-۰(در بخش ۴-۴) در بدترین حالت، $\theta(3^n)$ است. علاوه‌براین، درخت فضای حالات در الگوریتم شاخه و حد (الگوریتم ۳-۶) برای مسئله فروشندهٔ دوره‌گرد دارای !(n-1) برگ است؛ در حالیکه همین درخت برای الگوریتم شاخه و حد برای مسئله کوله‌پشتی ۱-۰(الگوریتم ۲-۶) تنها در حدود $3^n + 1$ گره دارد و بالاخره اینکه الگوریتم برنامه‌نویسی ما برای مسئله کوله‌پشتی ۱-۰ در $\theta(nw)$ است و این بدین معناست که این الگوریتم برای زمانی که W خیلی بزرگ نباشد، کارا خواهد بود. با نگاهی به این مسائل به نظر می‌رسد که شاید مسئله کوله پشتی ۱-۰ آسانتر از مسئله فروشندهٔ دوره‌گرد باشد. نشان می‌دهیم که برخلاف این دو مسئله، سایر مسائل در مثالهای ۹-۲ تا ۹-۵ و هزاران مسئله دیگر همگی در این امر مشترکند که اگر یکی از آنها در P باشد، همگی آنها بایستی در P باشند. چنین مسائلی، **NP-کامل** نامیده می‌شوند. برای رسیدن به چنین نتیجه‌ای، ابتدامسئله‌ای را تحت عنوان مسئلهٔ حل‌پذیری-*CNF* تعریف می‌کنیم که پایهٔ NP-کامل است.

مثال ۹-۶ مسئلهٔ حل‌پذیری-CNF

یک **متغیر منطقی (بولی)** متغیری است که می‌تواند یکی از دو مقدار درست (true) یا نادرست (false) را داشته باشد. اگر x یک متغیر منطقی باشد، $\overline{x}$ نقیض آن است. یعنی اینکه x درست است اگر و فقط اگر $\overline{x}$ نادرست باشد. یک **کلمه (literal)** عبارت است از یک متغیر منطقی یا نقیض یک متغیر منطقی، و یک **جمله (Clause)** عبارت است از دنباله‌ای از کلمات که با عملگر منطقی **or** (∨) از هم جدا شده‌اند. یک عبارت منطقی در **فرم نرمال عطفی (CNF)** عبارتست از دنباله‌ای از جملات که بوسیلهٔ عملگر منطقی **and** (∧) از هم جدا شده‌اند. بعنوان مثال، عبارت زیر یک نمونه از عبارت منطقی در CNF است.

$$(\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_4) \wedge (\overline{x}_2 \vee x_3 \vee x_4)$$

مسئله تصمیم‌گیری حل‌پذیری-CNF این است که در یک عبارت منطقی CNF، تعیین شود که آیا یک انتساب درستی (مجموعه‌ای از انتسابهای درست و نادرست به متغیرها) وجود دارد که ارزش عبارت را درست (true) کند یا خیر.

مثال ۷-۹ برای نمونهٔ

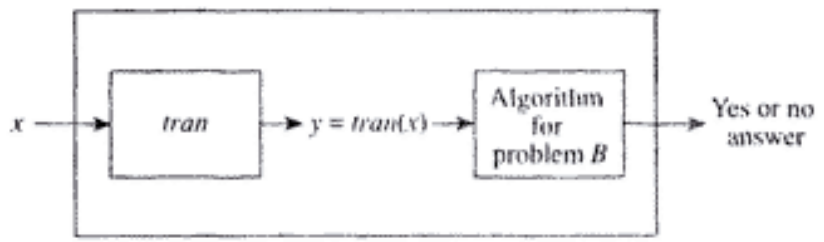
$$(x_1 \vee x_2) \wedge (x_2 \vee \bar{x}_3) \wedge \bar{x}_2$$

جواب حل‌پذیری-CFN، «بله» است زیرا انتسابهای $x_1 = true$,$x_2 = false$,$x_3 = false$ ارزش کل عبارت را درست می‌کند. برای نمونهٔ

$$(x_1 \vee x_2) \wedge \bar{x}_1 \wedge \bar{x}_2$$

جواب حل‌پذیری-CNF، «خیر» است زیرا هیچ انتساب درستی وجود ندارد که ارزش عبارت را درست کند.

می‌توان به راحتی الگوریتمی زمان- چندجمله‌ای نوشت که یک عبارت منطقی CNF و همچنین مجموعه‌ای از انتسابهای درستی به متغیرها را به عنوان ورودی بگیرد و ثابت کند که آیا آن عبارت برای آن انتساب درست است یا نه. بنابراین این مسئله در NP است. تاکنون هیچ کسی یک الگوریتم زمان- چندجمله‌ای برای این مسئله پیدا نکرده است و هیچ کسی هم ثابت نکرده است که این مسئله را نمی‌توان در یک زمان چندجمله‌ای حل کرد. لذا نمی‌دانیم که آیا این مسئله در P است یا خیر. یک موضوع قابل ذکر را جمع به این مسئله این است که در سال ۱۹۷۱، استفن کوک مقاله‌ای را ارائه داد که ثابت می‌کرد که اگر حل پذیری -CNF در P باشد، در این صورت P=NP. قبل از آنکه این قضیه را توضیح دهیم، نیاز به یک مفهوم جدید با نام «کاهش‌پذیری زمان- چندجمله‌ای» داریم.



شکل ۴-۹ الگوریتم tran هر نمونهٔ x از مسئلهٔ تصمیم‌گیری A را به یک نمونهٔ y از مسئلهٔ تصمیم‌گیری B تبدیل می‌کند.

فرض کنید می‌خواهیم مسئله تصمیم‌گیری A را حل کنیم و الگوریتمی داریم که مسئله تصمیم‌گیری B را حل می‌کند. همچنین فرض کنید می‌خواهیم الگوریتمی بنویسیم که نمونه y از مسئله B را از هر نمونه x از مسئله A طوری بسازد که یک الگوریتم برای مسئله B جوابهای «بله» برای y بدهد اگر و فقط اگر جواب مسئله A برای نمونه x «بله» باشد. چنین الگوریتمی، **الگوریتم تبدیل** نامیده می‌شود. در واقع، تابعی است که هر نمونه از مسئله A را به نمونه‌ای از مسئله B تصویر می‌کند. می‌توان این موضوع را بدینصورت بیان کرد:

$$y = tran(x)$$

الگوریتم تبدیل به همراه الگوریتمی برای مسئله B، منجر به الگوریتمی برای مسئله A می‌شود. شکل ۹-۲ این موضوع را نشان می‌دهد. مثال زیر نمونه‌ای از یک الگوریتم تبدیل است.

مثال ۸-۹ یک الگوریتم تبدیل

فرض کنید اولین مسئله تصمیم‌گیری ما عبارت باشد از n متغیر منطقی؛ آیا حداقل یکی از آنها دارای مقدار «درست» است؟ فرض کنید مسئله تصمیم‌گیری دوم ما عبارت باشد از n عدد صحیح؛ آیا بزرگترین آنها، عددی مثبت است؟ و فرض کنید تبدیل ما بدین صورت باشد:

$$k_1, k_2, \ldots, k_n = tran(x_1, x_2, \ldots, x_n)$$

که در آن k_i در صورتی ۱ است که x_i درست باشد و اگر x_i نادرست باشد، k_i صفر است. یک الگوریتم برای مسئله دوم، «بله» را برمی‌گرداند اگر و تنها اگر حداقل یک k_i برابر ۱ باشد.(یعنی اگر و فقط اگر حداقل یک x_i درست باشد که در این صورت تبدیل ما موفقیت آمیز است و می‌توانیم مسئله اول را با استفاده از الگوریتم مسئله دوم حل کنیم.

تعریف اگر یک الگوریتم تبدیل زمان- چندجمله‌ای از مسئله تصمیم‌گیری A به مسئله تصمیم‌گیری B وجود داشته باشد، مسئله A **کاهش‌پذیر چندبه‌یک زمان- چندجمله‌ای** به مسئله B می‌باشد. (معمولاً فقط می‌گوئیم که مسئله A به مسئله B کاهش می‌یابد.) که با این نماد نشان می‌دهیم:

$$A \propto B$$

می‌گوئیم «چندبه‌یک» زیرا یک الگوریتم تبدیل تابعی است که چندین نمونه از مسئله A را به یک نمونه از مسئله B تصویر می‌کند. یعنی اینکه آن یک تابع چند به یک است.

اگر الگوریتم تبدیل، زمان- چندجمله‌ای باشد و الگوریتمی زمان- چندجمله‌ای برای مسئله B داشته باشیم، در ابتدا بنظر می‌رسد که الگوریتم مسئله A که حاصل از ترکیب الگوریتم تبدیل و الگوریتم مسئله B است، باید یک الگوریتم زمان- چندجمله‌ای باشد. بعنوان مثال، واضح است که الگوریتم تبدیل در مثال ۸-۹، زمان- چندجمله‌ای است. بنابراین، بنظر می‌رسد که اگر آن الگوریتم را اجرا کنیم و به دنبال آن

الگوریتمی زمان- چندجمله‌ای را برای مسئله دوم اجرا نمائیم، در اینصورت مسئله اول را در یک زمان چندجمله‌ای حل می‌کنیم. قضیه زیر صحت چنین چیزی را ثابت می‌کند.

قضیه ۱-۹ اگر مسئله تصمیم‌گیری B در P باشد و $A \propto B$، دراینصورت مسئله تصمیم‌گیری A نیز در P خواهد بود.

اثبات: فرض کنید که P یک چندجمله‌ای باشد که پیچیدگی زمانی الگوریتم تبدیل زمان- چندجمله‌ای را از مسئله A به مسئله B محدود می‌سازد. همچنین فرض کنید q یک چندجمله‌ای باشد که پیچیدگی زمانی الگوریتم زمان- چندجمله‌ای B را محدود می‌سازد و فرض کنید نمونه‌ای از مسئله A را با اندازه n داریم. از آنجائیکه حداکثر p(n) مرحله در الگوریتم تبدیل ما وجود دارد و الگوریتم در بدترین حالت، در هر مرحله یک نماد را به عنوان خروجی می‌فرستد، لذا اندازهٔ نمونه مسئله B که توسط الگوریتم تبدیل تولید شده، حداکثر p(n) است. وقتی آن نمونه به عنوان ورودی الگوریتم مسئله B باشد، یعنی اینکه حداکثر q(p(n)) مرحله وجود دارد. بنابراین، میزان کل کار مورد نیاز برای تبدیل نمونه مسئله A به نمونه‌ای از مسئله B و سپس حل مسئله B برای رسیدن به جوابی برای مسئله A، حداکثر $p(n)+q(p(n))$ می‌باشد که بر حسب n بصورت چندجمله‌ای است.

تعریف مسئله B، **NP-کامل** نامیده می‌شود اگر

۱- NP باشد. ۲- برای هر مسئله دیگری در NP مانند A، $A \propto B$ باشد.

اگر بتوانیم با استفاده از قضیه ۱-۹ نشان دهیم که هر مسئله NP-کامل در P است، می‌توانیم نتیجه بگیریم که P=NP است. در سال ۱۹۷۱، استفن کوک تلاش کرد تا مسئله‌ای را بیابد که NP-کامل باشد.

قضیه ۲-۹ (قضیهٔ کوک) یک مسئله از نوع حل‌پذیری-CNF، NP-کامل است.

اثبات: اثبات این قضیه را در کتاب cook(۱۹۷۱) و کتاب Johnson و Garey(۱۹۷۹) مطالعه کنید.

قضیه ۳-۹ یک مسئله C در صورتی NP-کامل است که

۱- در NP باشد. ۲- برای مسئله NP-کامل دیگری مانند B، $B \propto C$ باشد.

اثبات: چون B، NP-کامل است، برای هر مسئله NP مانند A، $A \propto B$

می‌توان به راحتی فهمید که کاهش‌پذیری خاصیت تراگذاری دارد. بنابراین $A \propto C$. چون C در NP است می‌توان نتیجه گرفت که C، NP-کامل است.

با استفاده از قضیه کوک و قضیه ۳-۹ می‌توانیم NP-کامل بودن یک مسئله را بدینصورت نشان دهیم که اولاً، آن مسئله در NP است و ثانیاً، حل‌پذیری-CNF به آن مسئله کاهش می‌یابد. این کاهش‌ها معمولاً پیچیده‌تر از آن چیزی هستند که در مثال ۷-۹ آمده است.

مسئله چرخه‌های هامیلتونی را در بخش ۶-۵ بررسی کردیم. مسئله تصمیم‌گیری چرخه‌های هامیلتونی تعیین این موضوع است که آیا یک گراف متصل و بدون‌جهت دارای حداقل یک تور می‌باشد یا خیر. می‌توان نشان داد که

مسئله تصمیم‌گیری چرخه‌های هامیلتونی $\propto$ حل‌پذیری-CNF

بعنوان تمرین، یک الگوریتم اثبات زمان- چندجمله‌ای برای این مسئله بنویسید. بنابراین، می‌توانیم نتیجه بگیریم که مسئله تصمیم‌گیری چرخه‌های هامیلتونی در NP-کامل است. اکنون که می‌دانیم مسئله تصمیم‌گیری چرخه‌های هامیلتونی و مسائلی نظیر مسئلهٔ تصمیم‌گیری کلیک، NP-کامل است، می‌توانیم نشان دهیم که برخی از مسائل دیگر که در NP هستند، در NP-کامل نیز می‌باشند و اینکار را می‌توانیم بدینصورت انجام دهیم که نشان دهیم مسئله تصمیم‌گیری کلیک و یا مسئله تصمیم‌گیری چرخه‌های هامیلتونی به آن مسئله کاهش می‌یابد (با توجه به قضیه ۳-۹). یعنی لازم نیست که به حل‌پذیری-CNF برگردیم تا هر NP-کامل را ثابت کنیم.

مثال ۹-۹ یک حالت تغییر یافته از مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد را درنظر بگیرید. یک گراف وزن‌دار و بدون‌جهت، و عدد صحیح مثبت d مفروض است. تعیین کنید آیا یک تور بدون‌جهت با وزن کمتر از d وجود دارد یا خیر. واضح است که الگوریتم اثبات زمان- چندجمله‌ای که قبلاً برای مسئله فروشندهٔ دوره‌گرد داشتیم، برای این مسئله نیز کار می‌کند. بنابراین، این مسئله در NP است و تنها بایستی نشان دهیم که یک مسئله NP-کامل ی به آن کاهش می‌یابد تا بدین ترتیب نتیجه بگیریم که این مسئله در NP-کامل است. می‌خواهیم نشان دهیم که

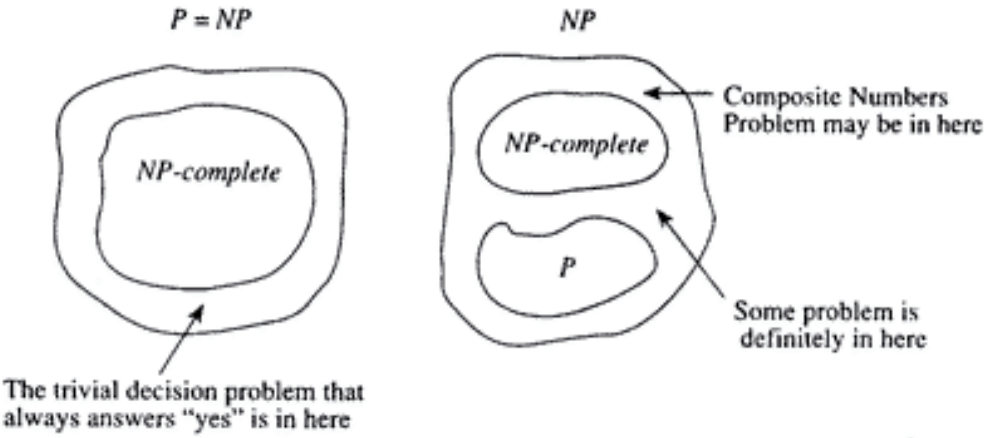
مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد (بدون جهت) $\propto$ مسئله تصمیم‌گیری چرخه‌های هامیلتونی

یک نمونهٔ (V, E) از مسئله تصمیم‌گیری چرخه‌های هامیلتونی را به نمونه (V, E') از مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد (بدون جهت) تبدیل می‌کنید بطوری که دارای همان مجموعه رئوس V، لبه‌ای بین هر زوج از رئوس با اوزان زیر می‌باشد:

$$\text{weight of } (u,v) \text{ equal to } \begin{cases} ۱ & \text{if } (u,v) \in E \\ ۲ & \text{if } (u,v) \notin E \end{cases}$$

بدیهی است که (V, E) دارای چرخه هامیلتونی است اگر و فقط اگر (V, E') دارای یک تور با مجموع وزن کمتر از n باشد که n، تعداد رئوس در V است. به عنوان تمرین نشان دهید که این تبدیل، یک تبدیل زمان- چندجمله‌ای است و بدین ترتیب مثال را تکمیل کنید.

شکل ۵-۹ نمایش مجموعهٔ NP.



مثال ۱۰-۹ قبلاً یک الگوریتم اثبات زمان- چندجمله‌ای برای مسئله متداول تصمیم‌گیری فروشنده دوره‌گرد نوشتیم. این مسئله در NP است و می‌توانیم نشان دهیم که

مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد $\propto$ مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد (بدون جهت)

و بدین ترتیب ثابت کنیم که این مسئله NP-کامل است. یک نمونهٔ (V, E) از مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد (بدون جهت)، را به نمونه (V,E') از مسئله فروشندهٔ دوره‌گرد تبدیل می‌کنیم بطوری که دارای همان مجموعه از رئوس V باشد و هرگاه (u, v) در E باشد، لبه‌های $<v,u>, <u,v>$ در E' باشند. اوزان جهتدار $<v,u>, <u,v>$ همانند وزن بدون جهت (u,v) است. بدیهی است که (V, E) دارای یک تور بدون جهت با مجموع وزن کمتر از d است اگر و تنها اگر (V,E') دارای یک تور با مجموع وزن کمتر از d باشد. بعنوان تمرین نشان دهید که این تبدیل، زمان- چندجمله‌ای است.

وضعیت NP

شکل ۳-۹، P را به عنوان زیرمجموعهٔ مناسبی از NP نشان می‌دهد اما همانطوری که قبلاً ذکر شد، ممکن است آن دو مجموعه، یکسان باشند. اما مسائل NP-کامل را چگونه می‌توان در این تصویر قرار داد؟ در ابتدا، طبق تعریف، این مسائل زیرمجموعه‌ای از NP می‌باشند. بنابراین، محاسبات Presburger، مسئلهٔ توقف و بسیاری از مسائل تصمیم‌گیری دیگر که در NP نیستند، در NP-کامل نیز قرار ندارند. یک مسئله تصمیم‌گیری که در NP است ولی در NP-کامل نیست، برای تمامی نمونه‌ها جواب «بله» می‌دهد (و یا اینکه برای تمامی نمونه‌ها جواب «خیر» می‌دهد). اگر P=NP باشد. وضعیت مشابه شکل سمت چپ تصویر ۷-۹ می‌شود و اگر $P \neq NP$ باشد، وضعیت مطابق تصویر سمت راست آن شکل می‌شود. یعنی اگر $P \neq NP$ باشد، در اینصورت $\emptyset$ = NP-کامل $\cap$ P، که در آن، NP-کامل مبین مجموعهٔ

تمام مسائل NP-کامل است. علت این امر اینست که اگر مسئله‌ای در P، NP-کامل باشد، قضیه ۱-۹ به طور ضمنی می‌گوید که می‌توانیم هر مسئله‌ای در NP را در یک زمان چندجمله‌ای حل کنیم.

شکل ۵-۹ (تصویر سمت راست) می‌گوید که مسئله اعداد مرکب ممکن است در

NP - (P ∪ NP-کامل)

وجود داشته باشد. این مسئله بصورت زیر است:

مثال ۱۱-۹ مسئله اعداد مرکب

با فرض یک عدد صحیح مثبت n، آیا اعداد صحیح $m > 1$, $k > 1$ وجود دارند که $n = mk$ باشد؟

الگوریتم زیر تعیین می‌کند که آیا این مسئله در یک زمان چندجمله‌ای، «بله» است. (یعنی آیا مسئله در NP است) یا خیر؟

```
bool verify_composite (int n, int m, int k)
{
  if (m amd k are both integers < 1 && n == m*k)
    return true;
  else
    return false;
}
```

هیچ کسی تاکنون الگوریتمی زمان- چندجمله‌ای برای این مسئله پیدا نکرده است (توجه داشته باشید که n ورودی مسئله است و این بدان معناست که اندازه ورودی در حدود $lg\ n$ می‌باشد) و هیچ کسی هم ثابت نکرده که این مسئله NP-کامل است. بنابراین، نمی‌دانیم که آیا این مسئله در P است یا خیر و نمی‌دانیم که آیا آن، NP-کامل است یا خیر. هیچ کسی هم تاکنون ثابت نکرده که مسئله‌ای که در NP است، نه در P قرار دارد و نه در NP-کامل (چنین اثباتی، خودبه خود ثابت می‌کند که $P \neq NP$). اما ثابت شده که اگر $P \neq NP$ باشد، چنی مسئله‌ای بایستی وجود داشته باشد. این نتیجه که در شکل سمت راست تصویر ۷-۹ نشان داده شده، در قضیه زیر به صورت فرمول درآمده است.

قضیه ۴-۹ اگر $P \neq NP$ باشد آنگاه مجموعهٔ NP - (P ∪ NP-کامل)، مجموعه زیر تهی نیست.

اثبات: اثبات قضیه در کتاب Ladner (۱۹۷۵) آمده است.

مکمل مسائل

به شباهت بین مسئله اعداد مرکب و مسئله زیر توجه کنید.

مثال ۱۲-۹ مسئله اعداد اول

با یک عدد صحیح مثبت مفروض n، آیا n یک عدد اول است؟

مسئله اعداد اول، توسط الگوریتمی در ابتدای بخش ۲-۹ حل شده است. این یک مسئله مکمل برای مسئله اعداد مرکب می‌باشد. بطور کلی مسئله مکمل برای یک مسئله تصمیم‌گیری، مسئله‌ای است که هرگاه مسئله اصلی جواب «خیر» می‌دهد، جواب آن «بله» است و هرگاه مسئله اصلی جواب «بله» می‌دهد، جواب آن، «خیر» است.

مثال ۱۳-۹ مکمل مسئله تصمیم‌گیری فروشنده دوره‌گرد

با یک گراف وزن دار و عدد صحیح مثبت d، آیا توری وجود دارد که وزن کل آن بزرگتر از d نباشد؟

بدیهی است که اگر یک الگوریتم زمان- چندجمله‌ای قطعی برای مسئله پیدا کرده باشیم، یک الگوریتم زمان- چندجمله‌ای قطعی نیز برای مکمل آن مسئله خواهیم داشت. بعنوان مثال، اگر بتوانیم در یک زمان چندجمله‌ای تعیین کنیم که آیا یک عدد مرکب است یا خیر، در اینصورت می‌توانیم تعیین کنیم که آن عدد اول است یا خیر. با وجود این، یافتن یک الگوریتم زمان- چندجمله‌ای قطعی برای یک مسئله بطور خودکار یک الگوریتم زمان- چندجمله‌ای غیرقطعی برای مکمل آن تولید نمی‌کند. یعنی اثبات این که یکی در NP است، خودبه‌خود ثابت نمی‌کند که دیگری نیز در NP قرار دارد. الگوریتم اثبات مسئله اعداد اول بایستی بتواند در یک زمان چندجمله‌ای ثابت کند که یک عدد اول است؛ در حالیکه الگوریتم اثبات اعداد مرکب بایستی ثابت کند که آن عدد مرکب است که این کار با تابع verify-composite تا اندازه‌ای ساده‌تر است. پرات در سال ۱۷۵ نشان داد که مسئله اعداد اول در NP است. در مورد مکمل مسئله فروشندهٔ دوره‌گرد، الگوریتم بایستی بتواند در یک زمان چندجمله‌ای ثابت کند که هیچ توری با وزن کوچکتر یا مساوی d وجود ندارد. این کار بسیار پیچیده‌تر از اثبات وجود یک تور با وزن کوچکتر یا مساوی d است. هیچ کسی تاکنون یک الگوریتم اثبات زمان- چندجمله‌ای برای مکمل مسئله فروشنده دوره‌گرد پیدا نکرده است. در واقع، هنیچ کسی تاکنون ثابت نکرده که مسئله مکمل هر مسئله NP-کامل شناخته شده در NP است. از طرفی دیگر، هیچ کسی هم ثابت نکرده که یک مسئله می‌تواند در NP باشد؛ در حالیکه مکمل آن در NP نباشد. نتیجه زیر بدست آمده است.

قضیه ۵-۹ اگر مسئله مکمل هر مسئله NP-کامل در NP باشد، آنگاه مسئله مکمل هر مسئله NP نیز در NP خواهد بود.

اثبات: اثبات این قضیه در کتاب Garey و Johnson (۱۹۷۹) آمده است.

۳-۴-۹ مسائل NP-مشکل، NP-آسان، و NP-معادل

تاکنون فقط درباره مسائل تصمیم‌گیری بحث کرده‌ایم. اکنون می‌خواهیم نتایج بدست آمده را در حالت کلی بسط دهیم. یادآور می‌شویم که قضیهٔ ۱-۹ اشاره دارد به اینکه اگر مسئله تصمیم‌گیری A، کاهش‌پذیر چندبه‌یک زمان- چندجمله‌ای به مسئله B باشد، در اینصورت می‌توانیم مسئله A را با استفاده از یک الگوریتم زمان- چندجمله‌ای برای مسئله B حل کنیم. این مطلب را به مسائل غیرتصمیم‌گیری نیز تعمیم می‌دهیم.

تعریف اگر مسئله A بتواند در یک زمان چندجمله‌ای و با استفاده از الگوریتمی زمان- چندجمله‌ای فرضی برای مسئله B حل شود، در اینصورت مسئله A، **کاهش‌پذیر تورینگ زمان- چندجمله‌ای** به مسئله B است. (معمولاً فقط می‌گوئیم تورینگ A به B کاهش می‌یابد) و آن را به صورت زیر نشان می‌دهیم:

$$A \propto_T B$$

در این تعریف نیازی به وجود یک الگوریتم زمان- چندجمله‌ای برای مسئله B نمی‌باشد. این تعریف تنها می‌گوید که اگر چنین الگوریتمی وجود داشته باشد، مسئله A نیز در یک زمان چندجمله‌ای قابل حل خواهد بود. بدیهی است که اگر A و B، هر دو مسائل تصمیم‌گیری باشند، دراینصورت

$A \propto_T B$ اشاره دارد به اینکه $A \propto B$

در ادامه، موضوع NP-کامل را به مسائل غیرتصمیم‌گیری بسط می‌دهیم.

تعریف مسئله B، **NP-مشکل** نامیده می‌شود اگر به ازاء مسئله NP-کاملی مانند A داشته باشیم:

$$A \propto_T B$$

کاهشهای تورینگ، خاصیت تراگذاری دارند. بنابراین، تمامی مسائل NP به هر مسئله NP-مشکلی کاهش می‌یابند؛ یعنی *اگر یک الگوریتم زمان- چندجمله‌ای برای هر مسئله NP وجود داشته باشد، آنگاه* $P = NP$ *خواهد شد.*

چه مسائل در NP-مشکل وجود دارند؟ بدیهی است که هر مسئله NP-کامل ، NP-مشکل است. بنابراین، به دنبال مسائل غیرتصمیم‌گیری می‌گردیم که NP-مشکل باشند. قبلاً گفتیم که اگر بتوانیم یک الگوریتم زمان- چندجمله‌ای برای یک مسئله بهینه‌سازی پیدا کنیم، خودبه‌خود الگوریتمی زمان- چندجمله‌ای برای مسئله تصمیم‌گیری متناظر با آن داریم. بنابراین، مسئله بهینه‌سازی، متناظر با هر مسئله NP-کامل و NP-مشکل است. مثال زیر از تعریف کاهش‌پذیری تورینگ استفاده می‌کند تا این نتیجه را برای مسئله فروشندهٔ دوره‌گرد نشان دهد.

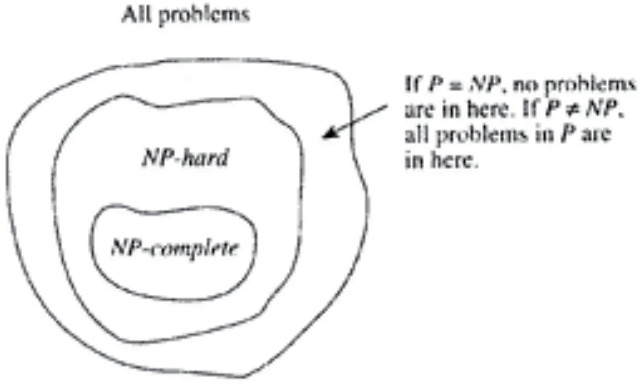
مثال ۱۴-۹ مسئله بهینه‌سازی فروشندهٔ دوره‌گرد، NP-مشکل است.

فرض کنید یک الگوریتم زمان- چندجمله‌ای فرضی برای مسئله بهینه‌سازی فروشنده دوره‌گرد و نمونه‌ای از مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد شامل گراف G و یک عدد صحیح مثبت d داریم. با بکارگیری الگوریتمی فرضی بر روی گراف G، جواب بهینه mindist را بدست آورید. آنگاه جواب ما برای این نمونه از مسئله تصمیم‌گیری در صورتی «بله» است که $d \leq mindist$ باشد و در غیراینصورت «خیر» است. بدیهی است که الگوریتم زمان- چندجمله‌ای فرضی برای مسئله بهینه‌سازی، به همراه این مرحلهٔ اضافی، جواب مسئله تصمیم‌گیری را در یک زمان چندجمله‌ای می‌دهد. بنابراین،

مسئله بهینه سازی‌فروشندهٔ دوره‌گرد $\propto_T$ مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد

چه مسائی در NP-مشکل نیستند؟ از وجود چنین مسئله‌ای خبر نداریم. در واقع اگر می‌توانستیم ثابت کنیم که مسئله‌ای در NP-کامل نیست، در اینصورت ثابت می‌شد که $P \neq NP$ است. به این دلیل که اگر باشد، آنگاه هر مسئله در NP توسط یک الگوریتم زمان- چندجمله‌ای قابل حل خواهد بود. بنابراین، می‌توانستیم هر مسئله در NP را (با استفاده از یک الگوریتم زمان- چندجمله‌ای فرضی برای هر مسئله‌ای مانند B) به سادگی و با فراخوانی الگوریتم زمان- چندجمله‌ای برای هر مسئله، حل کنیم. حتی به الگوریتمی فرضی برای مسئله B نیازی نداریم. بنابراین، تمام مسائل می‌بایست در NP-مشکل باشند.

از طرفی دیگر، هر مسئله‌ای که برایش یک الگوریتم زمان- چندجمله‌ای پیدا کردیم، ممکن است NP-مشکل نباشد. در واقع اگر ثابت می‌کردیم که مسئله‌ای که برایش یک الگوریتم زمان- چندجمله‌ای پیدا کردیم NP-کامل است، در اینصورت ثابت می‌شد که P = NP است. به این دلیل که، دراینصورت یک الگوریتم زمان- چندجمله‌ای واقعی (نه فرضی) برای برخی مسائل NP-مشکل وجود داشت. بنابراین، می‌توانستم هر مسئله در NP را با استفاده از کاهش تورینگ از آن مسئله به مسئله NP-مشکل، در یک زمان چندجمله‌ای حل کنیم. شکل ۶-۹، ارتباط این مجموعه از مسائل NP-مشکل را با مجموعه کل مسائل، نشان می‌دهد.



شکل ۶-۹ مجموعهٔ تمامی مسائل.

اگر مسئله‌ای NP-مشکل باشد، حداقل به سختی مسائل NP-کامل است. به عنوان مثال، مسئله بهینه‌سازی فروشندهٔ دوره‌گرد، حداقل به دشواری مسائل NP-کامل است. اما آیا عکس آن هم صحیح است؟ یعنی آیا مسائل NP-کامل نیز حداقل به سختی مسئله بهینه سازی فروشندهٔ دوره‌گرد هستند؟ برای پاسخ به این سوال نیاز به تعریف دیگری داریم.

تعریف مسئله A، **NP-آسان** نامیده می‌شود اگر برای مسئله‌ای مانند B که در NP است، داشته باشیم

$$A \alpha_T B$$

بدیهی است که اگر P= NP باشد، در اینصورت یک الگوریتم زمان- چند جمله‌ای برای تمام مسائل NP-آسان وجود دارد. توجه داشته باشید که تعریف ما از NP-آسان، دقیقاً متقارن با تعریف ما از NP-مشکل نیست. بعنوان تمرین نشان دهید که یک مسئله، NP-آسان است اگر و تنها اگر به یک مسئله NP2-کامل دیگر کاهش یابد.

چه مسائلی، NP-آسان هستند؟ مسلماً مسائل موجود در P، مسائل موجود در NP و مسائل غیر تصمیم‌گیری که برایشان الگوریتمهای زمان- چندجمله‌ای پیدا شده، همگی NP-آسان هستند. معمولاً می‌توان نشان داد که مسئله بهینه سازی (متناظر با یک مسئله تصمیم‌گیری NP-کامل)، یک مسئله NP-آسان است.

تعریف یک مسئله، **NP-معادل** نامیده می‌شود اگر هم NP-مشکل باشد و هم NP-آسان.

واضح است که *P = NP است اگر و تنها اگر برای تمام مسائل NPمعادل، الگوریتمهای زمان- چندجمله‌ای وجود داشته باشد.*

همانطوریکه مشاهده می‌کنیم محدود کردن نظریه به مسائل تصمیم‌گیری، به عمومیت آن لطمه‌ای وارد نمی‌سازد زیرا معمولاً می‌توانیم نشان دهیم که مسئله بهینه سازی متناظر با مسئله تصمیم‌گیری NP-کامل، NP-معادل است. این بدان معنی است که یافتن یک الگوریتم زمان- چندجمله‌ای برای مسئله بهینه سازی، معادل است با یافتن چنین الگوریتمی برای مسئله تصمیم‌گیری.

تمرینات

بخشهای ۱-۹ تا ۳-۹

۱- سه مسئله را نام ببرید که دارای الگوریتمهای زمان- چندجمله‌ای باشند توضیح دهید؟

۲- یک مسئله و دو طرح کدگذاری برای ورودی آن ارائه دهید. کارایی طرحهای خود را بررسی کنید.

۳- مسئله محاسبه جمله nام دنبالهٔ فیبوناچی به کدامیک از سه دسته کلی بحث شده در بخش ۳-۹ تعلق دارد؟ توضیح دهید.

۴- یک گراف دارای یک چرخهٔ اویلر است اگر و تنها اگر (a) گراف متصل باشد و (b) درجهٔ هر رأس زوج باشد. حد پائینی را برای پیچیدگی زمانی تمام الگوریتمهایی بیابید که تعیین می‌کنند آیا یک گراف دارای چرخهٔ اویلر است یا خیر. این مسئله به کدامیک از سه دسته کلی بحث شده در بخش ۳-۹ تعلق دارد؟

۵- حداقل دو مسئله را نام ببرید که به هر یک از سه دستهٔ بحث شده در بخش ۳-۹ تعلق داشته باشند.

بخش ۴-۹

۶- الگوریتم اثبات برای مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد (که در بخش ۱-۴-۹ بحث شده) را بر روی سیستم خود پیاده‌سازی نموده و کارایی زمان- چندجمله‌ای آن را بررسی کنید.

۷- ثابت کنید که مسائل مثالهای ۲-۹ تا ۵-۹ در NP هستند.

۸- یک الگوریتم اثبات زمان- چندجمله‌ای برای مسئله تصمیم‌گیری کلیک بنویسید.

۹- نشان دهید که کاهش از مسئله حل‌پذیری-CNF به مسئله تصمیم‌گیری کلیک را می‌توان در یک زمان چندجمله‌ای انجام داد.

۱۰- یک الگوریتم اثبات زمان- چندجمله‌ای برای مسئله تصمیم‌گیری چرخه‌های هامیلتونی بنویسید.

۱۱- نشان دهید کاهش از مسئله تصمیم‌گیری چرخه‌های هامیلتونی به مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد (بدون جهت) را می‌توان در یک زمان چندجمله‌ای انجام داد.

۱۲- نشان دهید کاهش از مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد (بدون جهت) به مسئله تصمیم‌گیری فروشندهٔ دوره‌گرد را می‌توان در یک زمان چندجمله‌ای انجام داد.

۱۳- نشان دهید که یک مسئله NP-آسان است اگر و تنها اگر به یک مسئله NP-کامل کاهش یابد.

۱۴- فرض کنید مسئله A و مسئله B، دو مسئله مختلف از مسائل تصمیم‌گیری باشند. همچنین فرض کنید که مسئله A، کاهش‌پذیر چندبه‌یک زمان- چند جمله‌ای به مسئله B باشد. اگر مسئله A، NP-کامل باشد، آیا مسئله B نیز NP-کامل است؟ توضیح دهید.

۱۵- هنگامی که تمام نمونه‌های مسئله حل‌پذیری-CNF دارای دقیقاً سه کلمه در هر جمله باشند، مسئله حل‌پذیری-۳ نامیده می‌شود. با دانستن این موضوع که مسئله حل‌پذیری-۳، NP-کامل است، نشان دهید که مسئله ۳-رنگ‌آمیزی گراف نیز NP-کامل است.

۱۶- نشان دهید که اگر یک مسئله NP نباشد، NP-آسان نیست. لذا محاسبات presburger و مسئله توقف نیز NP-آسان نمی‌باشند.

فصل ۱۰

الگوریتم‌های موازی

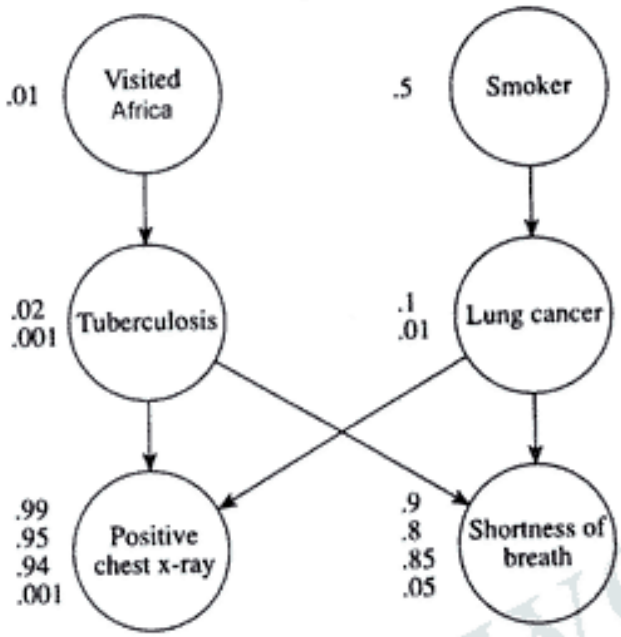
فرض کنید که می‌خواهید یک حصار در حیاط منزلتان بسازید و برای این کار لازم است که ۱۰ چالهٔ عمیق حفر کنید. شاید این کار ناخوشایند و دشوار باشد که بخواهید ۱۰ چاله را به ترتیب و بصورت جداگانه حفر نمائید، لذا به دنبال چاره‌ای خواهید بود. به خاطر دارید که چگونه تام سایر، شخصیت مشهور مارک توانیز، دوستش را برای کمک در سفید کردن حصار با حیله‌هایی جالب فریب داد. شما هم می‌توانید از حیلهٔ زیرکانه مشابهی استفاده کنید. برای این کار پرنده‌ها را برای خبر کردن همسایه‌های جوان و نیرومندتان جهت همکاری در حفر چاله در حیاط خلوتتان پرواز دهید. فرد مناسب‌تر کسی است که قادر باشد یک چاله را سریعتر حفر کند و در این میدان همکاری پیروز شود. می‌توانید هدیه‌ای نه چندان مهم را نیز برای شخص برنده پیشنهاد کنید. بیشتر آنها، فقط می‌خواهند شایستگی و لیاقت خود را ثابت کنند. بدین ترتیب، در روز همکاری، ۱۰ همسایهٔ قوی و بامعرفت، در یک زمان ۱۰ چاله را حفر می‌کنند. این روش حفر چاله‌ها، روش موازی نامیده می‌شود. در عین حال که خودتان را از کار حفر چاله‌ها نجات داده‌اید، توانسته‌اید کندن چاله‌ها را سریعتر از وقتی که خودتان با به ترتیب کندن آنها انجام می‌دادید، تمام کنید.

عیناً همانطوری که می‌توانید کار حفر چاله‌های را با همکاری همزمان دوستانتان سریعتر تمام کنید، یک کامپیوتر نیز می‌تواند یک کار را سریعتر انجام دهد اگر در این سیستم چندین پردازنده به صورت موازی دستورالعملها را به اجرا درآورند (یک پردازنده در یک کامپیوتر، یک جزء سخت‌افزاری است که دستورالعملها و داده‌ها را پردازش می‌کند.) تاکنون فقط در مورد پردازش ترتیبی بحث کرده‌ایم و همهٔ الگوریتم‌هایی که تا به حال ارائه نموده‌ایم، برای اینکه در کامپیوترهای ترتیبی انجام شوند، طراحی شده‌اند. چنانچه کامپیوتری تنها یک پردازنده برای اجرای ترتیبی دستورالعملها داشته باشد، شبیه به این است که خودتان حفر ۱۰ چاله را به تنهایی و به ترتیب انجام دهید. این کامپیوترها، براساس مدل ارائه شدهٔ جان وان نیومن پایه‌ریزی شدند. همانطوریکه در شکل ۱-۱۰ شرح داده شد، این مدل از یک تک‌پردازنده - موسوم به پردازندهٔ مرکزی یا CPU - و حافظه تشکیل شده است. این مدل، دنباله‌ای از دستورالعملها را گرفته و بر روی یک سری داده عمل می‌کند. چنین کامپیوتری، کامپیوتر **جریان تک دستوری، تک داده‌ای (SISD)** نامیده می‌شود که اغلب لفظ **کامپیوتر ترتیبی** نیز به آن اطلاق می‌گردد.

اگر در کامپیوتری، چندین پردازنده به طور همزمان دستورالعملها را اجرا کنند، مسائل زیادی می‌توانند به سرعت حل شوند که این کار در واقع، شبیه به داشتن ۱۰ همسایهٔ قوی جهت حفر ۱۰ چاله در یک زمان خواهد بود. بعنوان مثال، شبکهٔ فرضی که در بخش ۳-۶ معرفی شده است را درنظر بگیرید (شکل ۲-۱۰). هر رأس در آن شبکه، یک حالت ممکنه از یک بیماری را نشان می‌دهد. در اینجا، اگر داشتن یک حالت در یک رأس بتواند سبب ایجاد حالتی در رأس دیگر شود، یک ارتباط (لبه) از اولین گره به گرهٔ دوم وجود خواهد داشت. برای مثال، گره بالایی سمت راست، حالت سیگاری بودن را نشان می‌دهد و خروج لبه از آن گره به این معنی است که سیگار کشیدن می‌تواند سبب سرطان ریه شود. یک علت مشخص همیشه بر آنچه که به عنوان معلول ذکر کرده‌ایم، اثر گذار نیست. از اینرو لازم است که احتمال تشخیص اثر هر یک از علتها نیز در شکبه ذخیره شود. به عنوان مثال، احتمال اینکه یک شخص سیگاری باشد، برابر ۰/۵ است که در گرهٔ "سیگاری" ذخیره می‌شود. به احتمال ۰/۱، شخصی که سیگار می‌کشد به سرطان ریه مبتلا می‌شود و به احتمال ۰/۰۱، شخصی که سیگار نمی‌کشد به سرطان ریه دچار می‌شود. به احتمال ۰/۹۹، شخصی که هم سل و هم سرطان ریه دارد به پرتودرمانی به وسیله اشعهٔ ایکس نیازمند است. احتمالات دیگری که در گرهٔ "پرتو درمانی با X" مشاهده می‌کنیم برای سه ترکیب دیگر سل و سرطان ریه است. نتیجه‌گیری اصلی از یک شبکهٔ فرضی ،تعیین احتمال وقوع حالات برای گره‌هایی است که در آنها حالات و وضعیتهای مختلف ارائه شده است. به عنوان مثال، اگر تشخیص داده شد که بیماری معتاد است و به پرتو درمانی به کمک اشعه X نیاز دارد، شاید بدانیم که به چه احتمالی بیمار سرطان ریه داشت،

شکل ۱-۱۰ یک کامپیوتر ترتیبی سنتی.

شکل ۲-۱۰ یک شبکه فرضی.



سل داشت، تنگی نفس داشت و یا اخیراً از آفریقا دیدن کرده است. در سال ۱۹۸۶، یک الگوریتم استنباطی، برای حل این مسئله ارائه شد. در این الگوریتم، هر گره پیامهایی را برای والدین (ریشه) و فرزندانش (زیرشاخه‌ها) می‌فرستد. بعنوان مثال، هنگامی که تشخیص داده شد بیماری به پرتودرمانی با X نیاز دارد، گره "پرتو درمانی با X" پیامهایی را برای والدینش "سل" و "سرطان ریه" ارسال می‌کند. هنگامی که هر یک از این گره‌ها، پیامشان را دریافت کردند، احتمال این حالت در گره محاسبه شده، سپس این گره نیز پیامها را برای والدین دیگر و فرزندانش ارسال می‌کند. بعد از اینکه تمامی گره‌ها، پیامهای خود را دریافت کردند، احتمالهای جدید این حالتها در کلیهٔ گره‌ها محاسبه شده و گره‌ها پیامها را ارسال می‌کنند. روند گذر پیام در ریشه‌ها و برگها خاتمه می‌یابد. هنگامی که تشخیص داده شد بیماری معتاد نیز هست، جریان پیام دیگری در آن گره شروع می‌شود. یک کامپیوتر ترتیبی سنتی، تنها می‌تواند اندازهٔ یک پیام یا یک احتمال را در یک زمان محاسبه نماید. اندازهٔ پیام مربوط به "سل" می‌تواند در ابتدا محاسبه شود، سپس احتمال جدید سل، بعد اندازهٔ پیام "سرطان ریه" و احتمال آن و به همین ترتیب ادامه می‌یابد.

اگر هر گره خودش پردازنده‌ای داشت که قادر به فرستادن پیام‌ها به پردازنده‌های موجود در دیگر رأسها بود می‌توانستیم در ابتدا محاسبه را انجام داده و پیام‌ها را به "سل" و "سرطان ریه" ارسال کنیم. هنگامی که هر یک از گره‌ها، پیام مربوطه را دریافت کردند، می‌توانند به طور مستقل محاسبات را انجام داده و پیام‌ها را به والدین و دیگر فرزندانش بفرستند. علاوه بر این، اگر بدانیم که بیماری معتاد است، محتوای گرهٔ "معتاد" می‌توانست به طور همزمان محاسبه شده و پیامی را به فرزندانش ارسال کند. روشن است که اگر موارد فوق اتفاق می‌افتاد، استنتاج بسیار سریعتر انجام می‌شد. یک شبکه فرضی که در اجراهای واقعی استفاده می‌شود، اغلب شامل صدها گره بوده و احتمالات حاصل از آن بلافاصله مورد نیاز می‌باشند. این موضوع به

این معنی است که زمان ذخیره‌سازی‌ها می‌تواند بسیار قابل توجه باشد.

چیزی که اکنون شرح دادیم، یک معماری برای نوع خاصی از کامپیوترهای موازی است. چنین کامپیوترهایی به این علت **موازی(Parallel)** نامیده می‌شوند که هر پردازنده می‌تواند دستورالعملها را همزمان به صورت موازی و به همراه دیگر پردازنده‌ها اجراکند. قیمت پردازنده‌ها در بیشتر از سه دههٔ اخیر، به صورت قابل توجهی کاهش یافته است. در حال حاضر، سرعت یک پردازنده، یکی از شاخصه‌های مهم سرعت کامپیوترهای ترتیبی است. از اینرو با اتصال موازی ریزپردازنده‌ها -همانگونه که در پاراگراف قبل شرح دادیم- این امکان که با کمترین هزینه، توان محاسباتی بالاتری نسبت به سریعترین کامپیوتر ترتیبی بدست آید، فراهم می‌شود. کاربردهای زیادی وجود دارند که می‌توانند به طور قابل توجهی از محاسبات موازی سود ببرند. کاربردهایی در هوش مصنوعی نظیر شبکهٔ فرضی، استنتاج در شبکه‌های عصبی، درک زبان طبیعی، تشخیص گفتار و بینایی ماشین و کاربردهایی نظیر پردازش پرس‌وجو در پایگاه داده، پیش‌بینی وضع هوا، اخطار آلودگی و تحلیل ساختارهای پروتئینی نیز از این قبیل کاربردها می‌باشند.

روشهای متعددی برای طراحی کامپیوترهای موازی وجود دارند. بخش ۱-۱۰ در مورد برخی از نکات ضروری در طراحی موازی و بعضی از معماریهای متداول‌تر موازی بحث می‌کند. بخش ۲-۱۰ نشان می‌دهد که چگونه الگوریتم‌هایی را برای یک نوع خاص از کامپیوترهای موازی به نام PRAM (ماشین با دستیابی موازی مستقیم) بنویسیم. آنچنانکه بعدها خواهیم دید، این نوع خاص از کامپیوترها، خیلی متداول و مرسوم نیستند. در واقع مدل PRAM یک تعمیم آشکار از مدل ترتیبی محاسبات است. اما از این گذشته، الگوریتم PRAM می‌تواند به الگوریتم‌هایی برای بسیاری از ماشینهای متداول و پرکاربرد امروزی ترجمه شود. لذا الگوریتم‌های PRAM را به عنوان مقدمه‌ای مناسب برای معرفی الگوریتم‌های موازی در نظر گرفته‌ایم.

۱-۱۰ معماریهای موازی

ساختار کامپیوترهای موازی به هر یک از سه حالت زیر می‌تواند تغییر کند:

۱- مکانیسم کنترلی

۲- سازمان فضای آدرس‌دهی

۳- شبکهٔ سلسله‌مراتبی (تسلسلی)

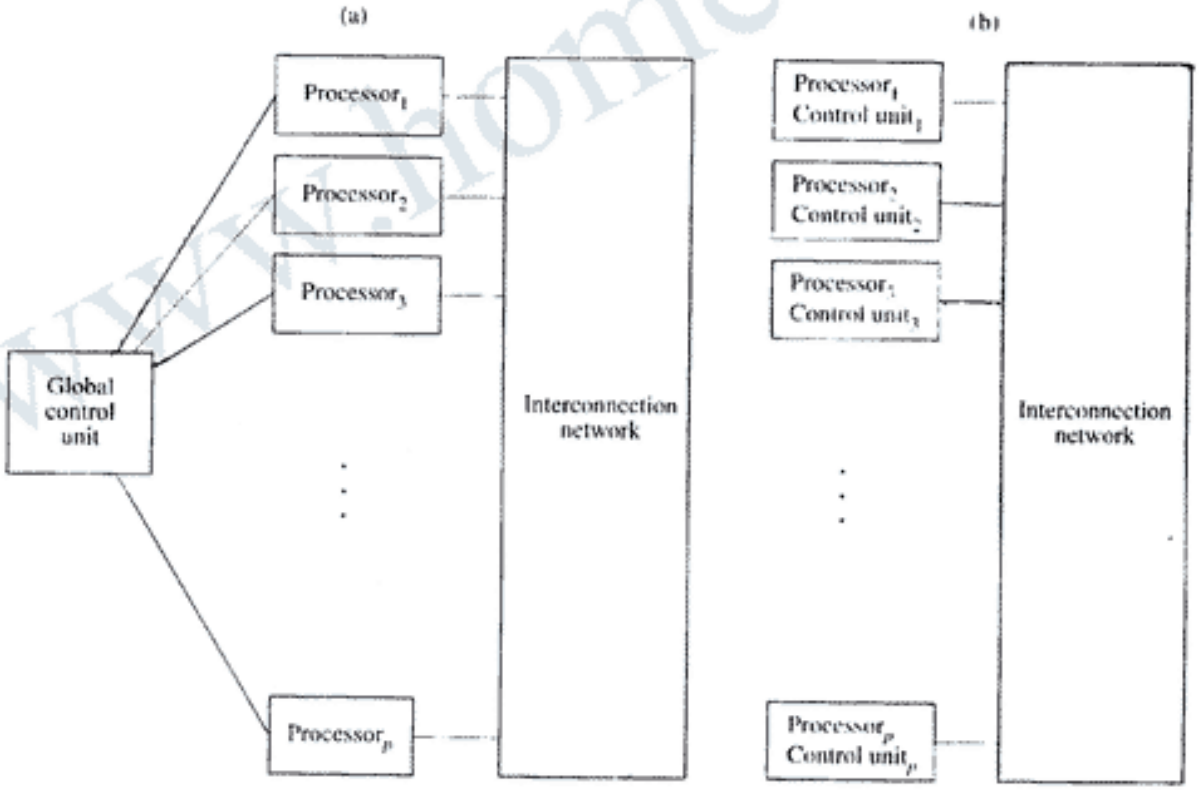
۱-۱-۱۰ مکانیسم کنترلی

هر پردازنده در کامپیوتر موازی می‌تواند براساس یک واحد کنترل مرکزی و یا به طور مستقل، تحت واحد کنترلی خودش کار کند. اولین نوع معماری **جریان تک دستوری، چند داده‌ای (SIMD)** نامیده می‌شود. شکل ۳(a)-۱۰، یک معماری SIMD را شرح می‌دهد. شبکهٔ سلسله‌مراتبی که در این شکل نمایش داده شده، سخت‌افزاری را نشان می‌دهد که قادر به اتصال پردازنده‌ها به یکدیگر می‌باشد. شبکه‌های

سلسله‌مراتبی در بخش ۳-۱-۱۰ مطرح شده‌اند. در معماری SIMD، دستورالعمل مشابهی به طور همزمان توسط واحدهای پردازش، تحت کنترل واحد کنترل مرکزی اجرا می‌شود. این بدین معنا نیست که همهٔ پردازنده‌ها بایستی حتماً دستورالعملی را در هر چرخه اجرا کنند بلکه هر پردازنده بنا به ضرورت می‌تواند در هر چرخهٔ دلخواه غیرفعال شود.

کامپیوترهای موازی که در آن هر پردازنده براساس واحد کنترل خودش کار می‌کند به کامپیوترهای **جریان چند دستوری، چند داده‌ای (MIMD)**، موسومند. شکل ۳-۱۰(b)، معماری MIMD را نشان می‌دهد. در این کامپیوترها، یک نسخه از سیستم عامل و برنامه در هر پردازنده نگهداری می‌شود.

کامپیوترهای SIMD، برای برنامه‌هایی مناسب هستند که دستورالعملهای مشابهی را روی داده‌های متفاوت اجرا می‌کنند. چنین برنامه‌هایی، برنامه‌های **داده-موازی** نامیده می‌شوند. یک مشکل کامپیوترهای SIMD این است که آنها نمی‌توانند دستورالعملهای متفاوتی را در یک چرخهٔ زمانی اجرا کنند. بعنوان مثال، فرض کنید که جملهٔ شرطی زیر بخواهد اجرا شود:



شکل ۳-۱۰ (a) یک معماری جریان تک دستوری، چند داده‌ای (SIMD).
(b) یک معماری جریان چند دستوری، چند داده‌ای (MIMD).

```
if (x == y)
   execute intructions A;
eles
   execute intructions B;
```

هر پردازنده‌ای که به $x \neq y$ برسد، (توجه داشته باشید که پردازنده‌ها در حال پردازش عناصر داده‌ای متفاوتی هستند) هیچ کاری نمی‌تواند انجام دهد تا زمانی که پردازنده‌هایی که به x=y رسیدند، دستورالعملهای A را به طور کامل اجرا کنند و همینطور آنهایی x=y را پیدا کردند بایستی بیکار بمانند تا زمانی که بقیه در حال اجرای دستورالعملهای B هستند. بطور کلی، کامپیوترهای SIMD بیشتر برای الگوریتم‌های موازیی مناسب هستند که در آنها نیاز به همزمانی نیست. بیشتر کامپیوترهای MIMD، سخت‌افزاری اضافه‌ای دارند که همزمانی را پشتیبانی می‌کنند. به این معنی که آنها می‌توانند کامپیوترهای SIMD را شبیه‌سازی کنند.

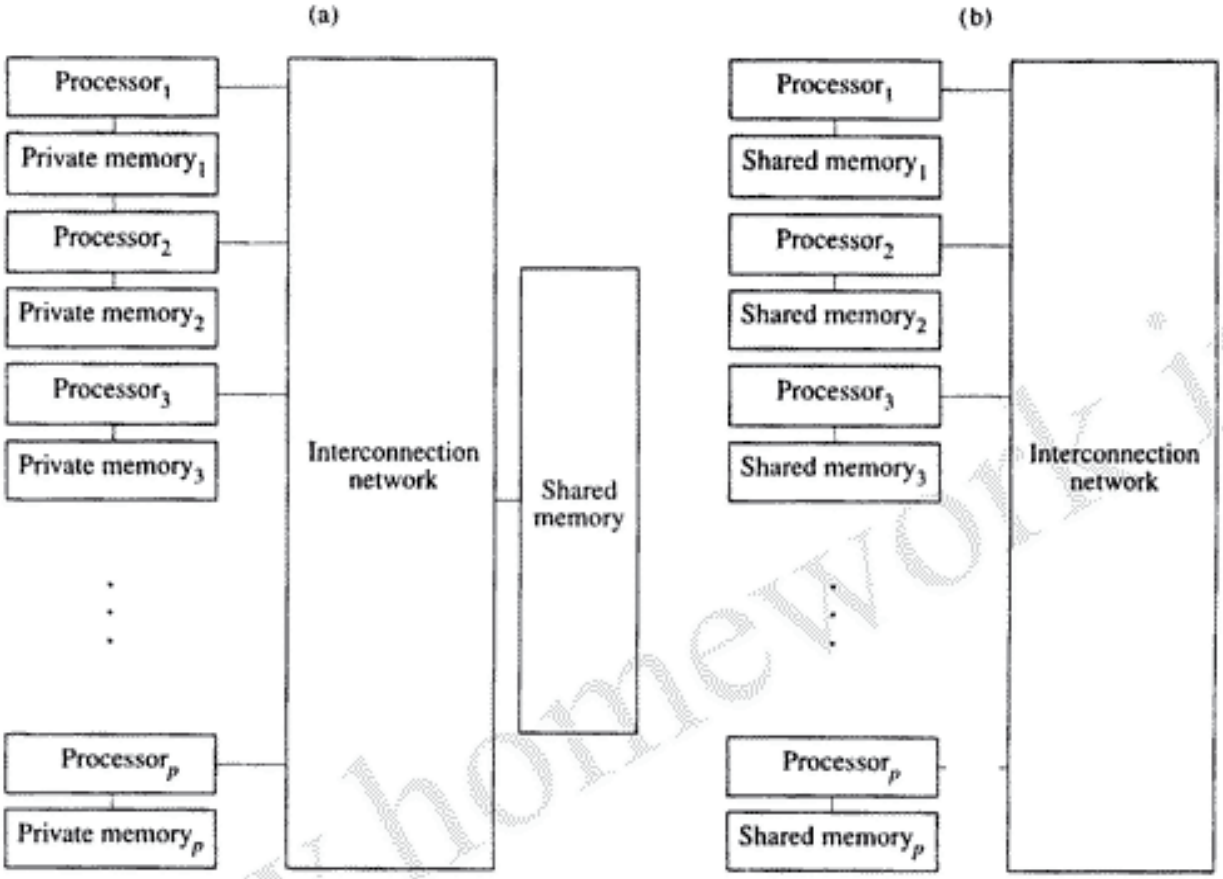
۲-۱-۱۰ سازماندهی فضای آدرسی

پردازنده‌ها می‌توانند به وسیلهٔ تغییر داده‌ها در یک فضای آدرسی مشترک و یا به وسیلهٔ ارسال پیام با یکدیگر ارتباط برقرار کنند. فضای آدرسی به صورتهای مختلفی برحسب روش ارتباطی مورد استفاده سازماندهی می‌شود.

معماری فضای آدرسی مشترک

در معماری فضای آدرسی مشترک، سخت‌افزاری لازم است تا امکان خواندن و نوشتن به یک آدرس اشتراکی را برای تمامی پردازنده‌ها فراهم نماید. در این ساختار، پردازنده‌ها با تغییر داده در یک فضای آدرسی مشترک، با یکدیگر ارتباط برقرار می‌کنند. شکل ۴(a)-۱۰، یک معماری فضای آدرس اشتراکی که در آن، زمان دستیابی به هر کلمه در حافظه یک مقدار ثابت است را نشان می‌دهد. چنین کامپیوتری، **کامپیوتر با دستیابی به حافظهٔ همگن (UMA)** نامیده می‌شود. در یک کامپیوتر UMA، ممکن است هر پردازنده خودش حافظهٔ اختصاصی داشته باشد؛ آنچنانکه در شکل ۴(a)-۱۰ نشان داده شد. این حافظهٔ اختصاصی فقط برای نگهداری متغیرهای محلی، جهت انجام عملیات محاسباتی هر پردازنده بکار می‌رود. هیچکدام از ورودیهای حقیقی الگوریتم در این ناحیهٔ اختصاصی وجود ندارد. مشکل کامپیوترهای UMA این است که شبکهٔ سلسله‌مراتبی باید به طور همزمان، دستیابی به حافظه اشتراکی را برای هر پردازنده فراهم کند که این امر موجب کاهش کارایی می‌شود. یک راه چاره، مجهز کردن هر پردازنده به یک بخش از حافظهٔ اشتراکی است (شکل ۴(b)-۱۰)، این حافظه، اختصاصی نیست بلکه هر پردازنده می‌تواند به محتوای حافظه در پردازنده‌های دیگر نیز دسترسی داشته باشد. پرواضح است که دسترسی پردازنده به حافظه خودش، سریعتر از دسترسی به حافظه‌های دیگر پردازنده‌ها است. لذا اگر بیشتر دستیابی‌های پردازنده به حافظهٔ خودش باشد، کارایی خوبی خواهیم داشت. چنین کامپیوتری، یک **کامپیوتر با دستیابی به حافظهٔ غیرهمگن (NUMA)** نامیده می‌شود.

شکل ۴-۱۰ (a) یک کامپیوتر با دستیابی به حافظه همگن (UMA).
(b) یک کامپیوتر با دستیابی به حافظهٔ غیر همگن (NUMA).



معماری پیام گذر

در یک معماری پیام گذر، هر پردازنده، خودش حافظهٔ اختصاصی دارد که فقط توسط همان پردازنده قابل دسترسی است. پردازنده‌ها به وسیله فرستادن پیامهایی به پردازنده‌های دیگر (اغلب به وسیلهٔ عناصر داده‌ای) با یکدیگر ارتباط برقرار می‌کنند. شکل ۵-۱۰، یک معماری پیام گذر را نشان می‌دهد. ملاحظه می‌کنید که شکل ۵-۱۰ بسیار شبیه به شکل ۴-۱۰(b) بنظر می‌رسد. تنها تفاوت موجود در چگونگی شکل‌گیری شبکهٔ سلسله‌مراتبی است. در کامپیوترهای NUMA، شبکهٔ سلسله‌مراتبی به هر پردازنده اجازه می‌دهد که به حافظهٔ دیگر پردازنده‌ها دسترسی داشته باشد؛ در حالیکه در کامپیوترهای پیام گذر، این شبکه به گونه‌ای است که در آن هر پردازنده فقط می‌تواند پیامی را مستقیماً به دیگر پردازنده‌ها ارسال نماید.

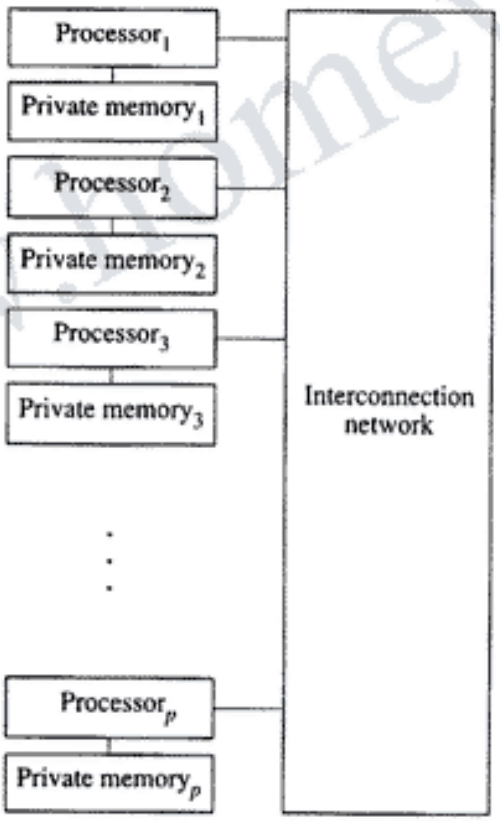
۳-۱-۱۰ شبکه‌های سلسله‌مراتبی

به طور کلی، شبکه‌های سلسله‌مراتبی به دو گروه تقسیم می‌شوند: ایستا و پویا. شبکه‌های ایستا، عموماً

برای تشکیل معماریهای پیام گذر بکار می‌روند، در حالی که از شبکه‌های پویا در معماریهای فضای آدرسی اشتراکی استفاده می‌شود. هر یک از این انواع شبکه‌ها را به ترتیب مطرح می‌کنیم.

شبکه‌های سلسله‌مراتبی ایستا

شبکهٔ سلسله‌مراتبی ایستا دارای اتصالهای مستقیم بین پردازنده‌ها است که به همین دلیل به **شبکه مستقیم** نیز موسوم است. چندین نوع مختلف از شبکه‌های سلسله‌مراتبی وجود دارند که در مورد بعضی از متداولترین آنها بحث می‌کنیم. کارآمدترین و در عین حال پرهزینه‌ترین نوع شبکه سلسله مراتبی، **شبکه سلسله مراتبی کامل** است که در شکل (a)۶-۱۰ شرح داده شد. در چنین شبکه‌ای هر پردازنده مستقیماً به پردازنده‌های دیگر متصل می‌شود. بنابراین، یک پردازنده می‌تواند مستقیماً از طریق خط ارتباطی، پیغامی را به پردازنده دیگر ارسال کند. از آنجائیکه تعداد اتصالها در این شبکه، مربعی از تعداد پردازنده‌های موجود است، لذا این نوع شبکه کاملاً پرهزینه است.



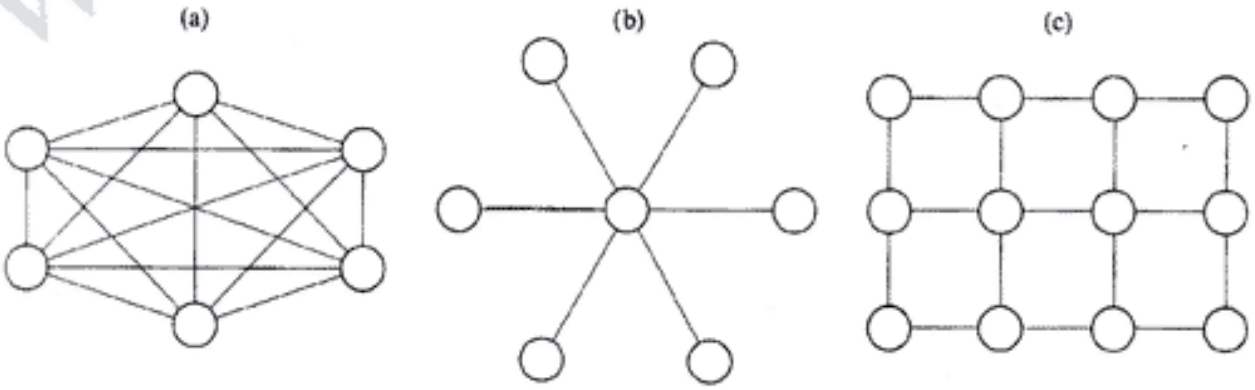
شکل ۵-۱۰ یک معماری پیام گذر. حافظهٔ هر پردازنده فقط به وسیله همان پردازنده قابل دسترسی است. یک پردازنده با ارسال پیام به پردازنده دیگر از طریق شبکهٔ سلسله‌مراتبی با یکدیگر ارتباط برقرار می‌کنند.

در یک **شبکهٔ سلسله مراتبی ستاره‌ای**، یک پردازنده به عنوان پردازندهٔ مرکزی عمل کرده و دیگر پردازنده‌ها تنها یک اتصال به این پردازندهٔ مرکزی دارند. شکل ۶-۱۰(b)، یک شبکهٔ سلسله‌مراتبی ستاره‌ای را نشان می‌دهد. در یک شبکهٔ سلسله‌مراتبی ستاره‌ای، یک پردازنده به وسیلهٔ ارسال پیام به پردازندهٔ مرکزی، پیامی را به پردازندهٔ دریافت کننده می‌فرستد و از این طریق با دیگر پردازنده‌ها مرتبط می‌شود.

در یک **شبکهٔ درجه محدود** از درجهٔ d، هر پردازنده به حداکثر d پردازندهٔ دیگر متصل می‌شود. در این شبکه، یک پیام ابتدا در یک مسیر و سپس در مسیرهای دیگر فرستاده می‌شود. شکل ۶-۱۰(c)، یک شبکهٔ درجه محدود از درجهٔ ۴ را نشان می‌دهد.

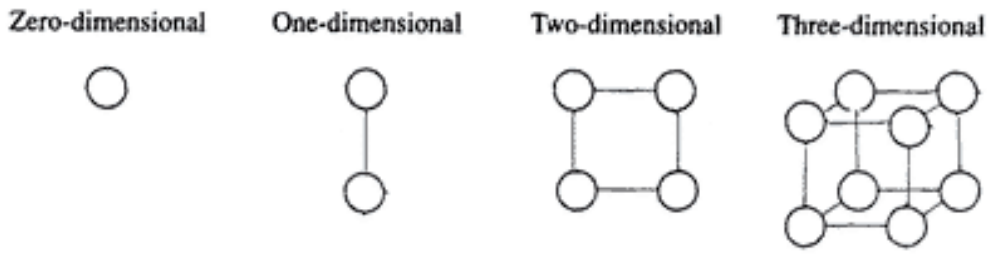
یک شبکهٔ ایستا با پیچیدگی بیشتر ولی متداول‌تر، **شبکهٔ فوق مکعبی** است. یک فوق مکعب صفربعدی شامل یک پردازنده است. فوق مکعب یک‌بعدی، از اتصال پردازنده‌ها در دو فوق مکعب صفربعدی تشکیل می‌شود. فوق مکعب دوبعدی از اتصال هر پردازنده در یک فوق مکعب یک‌بعدی به پردازنده‌ای در فوق مکعب یک‌بعدی دیگر تشکیل می‌شود و به همین ترتیب، یک فوق مکعب $d+1$ بعدی از طریق اتصال پردازنده‌ای در یک فوق مکعب d بعدی به یک پردازنده در فوق مکعب d بعدی دیگر تشکیل می‌شود. توجه داریم که یک پردازنده مشخص در فوق مکعب اول تنها به پردازنده‌ای در موقعیت مشابه خود در فوق مکعب دوم متصل می‌شود. شکل ۷-۱۰ شبکه‌های فوق مکعبی از درجات مختلف را نشان می‌دهد.

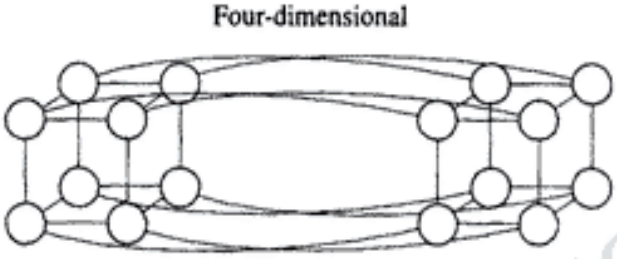
اغلب، شبکه‌های ایستا به عنوان ابزار معماریهای پیام گذر بکار می‌روند زیرا در چنین شبکه‌هایی که پردازنده‌ها می‌توانند مستقیماً به یکدیگر متصل شوند، عبور جریان پیام‌ها به راحتی و در اسرع وقت صورت می‌پذیرد.



شکل ۶-۱۰ (a) یک شبکهٔ سلسله‌مراتبی کامل. (b) یک شبکهٔ سلسله‌مراتبی ستاره‌ای.
(c) یک شبکهٔ درجه محدود از درجهٔ ۴.

شکل ۷-۱۰ شبکه‌های فوق مکعبی.



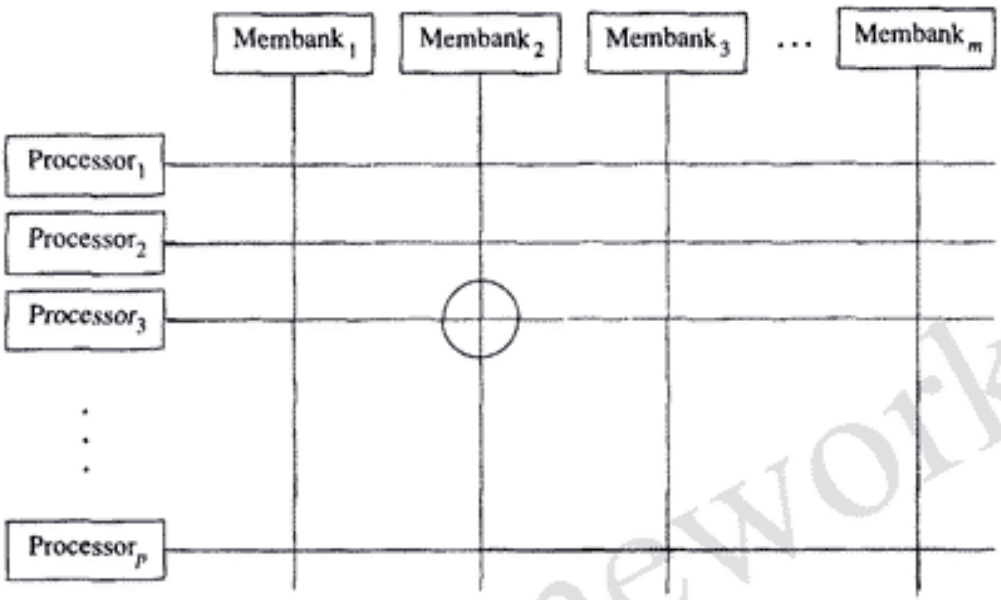


شبکهٔ سلسله‌مراتبی پویا

در یک **شبکهٔ سلسله‌مراتبی پویا**، پردازنده‌ها به وسیلهٔ مجموعه‌ای از عناصر سوئیچی به حافظه متصل می‌شوند. بهترین روش برای انجام این کار استفاده از **شبکه سوئیچی مشبک** است. در چنین شبکه‌ای، P پردازنده از طریق یک چارچوب مشبک از کلیدهای سوئیچی به m بانک حافظه متصل می‌شوند، آنچنانکه در شکل ۸-۱۰ نشان می‌دهیم. به عنوان مثال، اگر پردازندهٔ ۳ بتواند به بانک حافظهٔ ۲ دسترسی پیدا کند، نتیجه می‌گیریم که کلید موجود در محل تلاقی این دو -که در شکل ۸-۱۰ با دایره مشخص کرده‌ایم- بسته شده است. بسته شدن کلید به این معناست که جریان الکتریسته می‌تواند از این قسمت مدار عبور کند. این شبکه، به این دلیل پویا نامیده می‌شود که با بسته شدن یک کلید، ارتباط میان یک پردازنده و یک بانک حافظه به طور پویا برقرار می‌گردد. لازم به ذکر است که در یک شبکه سوئیچی، مانع وجود ندارد. بدین معنا که اتصال یک پردازنده به یک بانک حافظه، مانع اتصال پردازنده دیگر به بانک حافظهٔ دیگر نمی‌شود. ایده‌ال‌ترین حالت این است که در یک شبکه سوئیچی مشبک، یک بانک برای هر کلمهٔ حافظه موجود باشد به هر حال، روشن است که این کار عملی نیست. معمولاً تعداد بانک‌ها در یک شبکه، حداقل به تعداد پردازنده‌ها است. بنابراین در یک زمان معین، هر پردازنده قادر به دستیابی اقلاً یک بانک حافظه می‌باشد. تعداد سوئیچ‌ها در شبکه سوئیچی مشبک برابر pm است. لذا اگر در این شبکه، m بزرگتر یا مساوی p باشد، آنگاه تعداد سوئیچ‌ها بزرگتر یا مساوی P^2 خواهد بود. در نتیجه هنگامی که تعداد پردازنده‌ها زیاد باشد، شبکه‌ای سوئیچی مشبک بسیار پرهزینه خواهند شد.

دیگر شبکه‌های سلسله‌مراتبی پویا، که در اینجا تشریح نشده‌اند، شامل شبکه‌های گذرگاه مشترک و شبکه‌های سلسله‌مراتبی چند مرحله‌ای می‌باشند.

شکل ۸-۱۰ یک شبکهٔ سوئیچی مشبکه. در هر موقعیت از تقاطع شبکه یک سوئیچ وجود دارد. سوئیچی که با دایره مشخص شده، بسته شده است و قادر به عبور جریان اطلاعات بین پردازندهٔ ۳ و بانک حافظهٔ ۲ می‌باشد.



شبکه‌های سلسله مراتبی پویا، معمولاً به عنوان ابزاری برای معماریهای فضای آدرسی مشترک استفاده می‌شوند، زیرا در چنین شبکه‌هایی، هر پردازنده اجازهٔ دسترسی به هر کلمه از حافظه را دارد. البته نمی‌تواند پیامی را مستقیماً به دیگر پردازنده‌ها بفرستد.

بخش عمدهٔ مطالب ذکر شده از کتاب kumar (۱۹۹۴) گرفته شده است. شما می‌توانید برای آشنایی بیشتر با مقولهٔ سخت‌افزار موازی به ویژه شبکه‌های سلسله‌مراتبی چندمرحله‌ای و گذرگاه مشترک، به این کتاب مراجعه کنید.

۱۰-۲ مدل PRAM

همانطوریکه در بخش قبل مطرح شد، چندین معماری موازی مختلف وجود دارد و کامپیوترهای امروزی، براساس تعدادی از این معماریها ساخته می‌شوند. تمامی کامپیوترهای ترتیبی براساس معماریی که در شکل ۱-۱۰ نشان داده شده است، شکل گرفته‌اند. به این معنا که مدل وان نیومن، یک مدل جامع و عمومی برای کلیهٔ کامپیوترهای ترتیبی است. تنها فرضی که در طراحی الگوریتمها در فصلهای گذشته منظور شد این بود که آنها روی یک کامپیوتر براساس مدل وان نیومن اجرا می‌شوند، لذا هر یک از این الگوریتم‌ها پیچیدگی زمانی مشابهی بدون توجه به زبان برنامه‌نویسی یا کامپیوتر بکارگیرندهٔ آن دارند. این، یک عامل کلیدی در افزایش مؤثر بکارگیری کامپیوترهای ترتیبی است.

شاید بهتر باشد که یک مدل جامع برای محاسبات موازی پیدا کنیم. هر مدل این چنینی، برای تصرف خصوصیات کلیدی گروه بزرگی از معماریهای موازی مختلف، بایستی به اندازهٔ کافی جامعیت داشته باشد. و نکتهٔ دیگر اینکه، الگوریتم‌های طراحی شده براساس این مدل بایستی به صورتی کارا و مؤثر بر روی کامپیوترهای موازی واقعی اجرا شوند. چنین مدلی تا به حال شناخته نشده است و در واقع دستیابی به آن، غیرممکن به نظر می‌رسد.

اگرچه تا به حال مدل جامعی بدست نیامده است، لیکن کامپیوتر **PRAM (ماشین با دسترسی تصادفی موازی)** به عنوان یک مدل تئوریک برای ماشینهای موازی مطرح شده است. یک کامپیوتر PRAM از p پردازنده، که همهٔ آنها امکان دسترسی یکسانی را به یک حافظه بزرگ اشتراکی دارند، تشکیل شده است. پردازنده‌ها، یک ساعت(clock) عمومی را به طور مشترک در اختیار دارند؛ در حالیکه ممکن است هر یک، دستورالعمل متفاوتی را در هر چرخه اجرا کنند. بنابراین، یک کامپیوتر PRAM، یک کامپیوتر UMA، MIMD و همزمان (سنکرون) است. شکل‌های ۱۰-۳(b) و ۱۰-۴(b) معماری یک کامپیوتر PRAM را نمایش می‌دهند و شکل ۱۰-۸، یک شبکهٔ سلسله‌مراتبی ممکن را برای چنین ماشینی ارائه می‌نماید. همانطوریکه قبلاً نیز گفته شد، پیاده‌سازی واقعی چنین کامپیوتری، بسیار پرهزینه خواهد بود. به هر حال مدل PRAM یک گسترش ساده از مدل ترتیبی محاسبات است. این باعث می‌شود که مدل PRAM به هنگام ارائه الگوریتم‌های متفاوت، از لحاظ مفهومی آسان به نظر برسد. بعلاوه، الگوریتم‌های ارائه شده برای این مدل می‌توانند به الگوریتم‌هایی برای بسیاری از کامپیوترهای واقعی‌تر ترجمه شوند. به عنوان مثال، یک دستورالعمل PRAM می‌تواند در دستورالعملهای $\theta(lg\ p)$ که در آن p تعداد پردازنده‌ها است، بر روی یک شبکهٔ درجه بالا ظاهر شود. از این گذشته، الگوریتم‌های PRAM برای بسیاری از مسائل، دقیقاً به همان سرعت الگوریتم‌های فوق مکعبی عمل می‌کنند و به همین دلایل است که مدل PRAM،به عنوان یک مقدمهٔ مناسب برای الگوریتم‌های موازی بکار می‌رود.

در یک کامپیوتر با حافظهٔ اشتراکی، نظیر یک PARM، بیش از یک پردازنده می‌تواند به طور همزمان از یک مکان حافظه بخواند یا بر آن بنویسد. چهار نسخهٔ مختلف از PARM وجود دارد که براساس چگونگی دستیابی‌های حافظه به طور همزمان شکل گرفته‌اند.

۱- ***خواندن انحصاری، نوشتن انحصاری (EREW)***. در این نسخه، هیچ خواندن یا نوشتنی به صورت همزمان پذیرفته نیست. در اینجا فقط یک پردازنده می‌تواند در یک زمان معین به یک مکان معین از حافظه دسترسی داشته باشد. این نوع، پائین‌ترین نگارش از کامپیوتر PARM است، زیرا حداقل همزمانی را می‌پذیرد.

۲- ***خواندن انحصاری، نوشتن همزمان (ERCW)***. در این نسخه، عملیات نوشتن به صورت همزمان پذیرفته می‌شود، اما عملیات خواندن به صورت همزمان امکان‌پذیر نیست.

۳- ***خواندن همزمان، نوشتن انحصاری (CREW).*** در این نسخه، عملیات خواندن به صورت همزمان پذیرفته می‌شود، اما عملیات نوشتن نمی‌تواند به صورت همزمان انجام گیرد.

۴- ***خواندن همزمان، نوشتن همزمان (CRCW).*** در این نسخه، هر دو عملیات نوشتن و خواندن، اجازهٔ اجرای همزمان را دارند.

حال می‌خواهیم درباره یک طرح الگوریتمی برای مدلهای CREW PARM و CRCW PARM بحث کنیم. ابتدا مدل CREW را عنوان می‌کنیم و آنگاه نشان می‌دهیم که چگونه الگوریتم‌های کارآمدتری را می‌توان با استفاده از مدل CRCW ارائه نمود. قبل از شروع بحث، بیائید چگونگی ارائه الگوریتم‌های موازی را بررسی کنیم. اگرچه زبانهای برنامه‌نویسی خاصی برای الگوریتم‌های موازی وجود دارند، ولی ما شبه‌کدهای استاندارد خود را با بعضی ترکیبات جدید استفاده خواهیم کرد که بعداً تشریح خواهند شد.

فقط یک نسخه از الگوریتم نوشته شده است که پس از کامپایل، توسط تمامی پردازنده‌ها به صورت همزمان اجرا می‌شود. بنابراین، هر پردازنده نیاز دارد که شاخص خودش را هنگام اجرای الگوریتم بداند. ما فرض خواهیم کرد که پردازنده‌ها به صورت P_1 , P_2 , P_3 ,... شاخص‌دهی شده‌اند، در اینصورت دستورالعمل

P = index of this processor;

شاخص یک پردازنده را برمی‌گرداند. یک متغیر در الگوریتم می‌تواند یک متغیر در حافظهٔ اشتراکی باشد. به این معنی که قابل دسترسی برای همه پردازنده‌ها است یا می‌تواند در حافظهٔ اختصاصی باشد (به شکل ۲(a)-۱۰ مراجعه کنید.) در این مورد اخیر، هر پردازنده کپی خودش را از متغیر دارد. ما کلمه کلیدی local را هنگام تعریف یک متغیر از این نوع استفاده می‌کنیم.

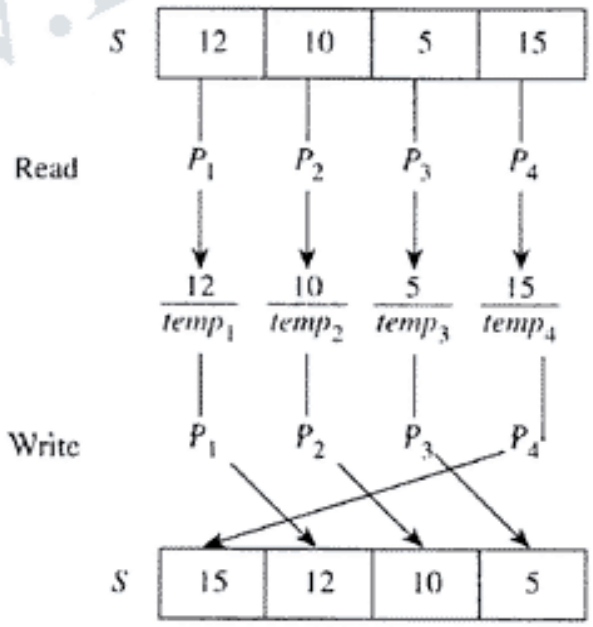
همه الگوریتم‌های ما، الگوریتم‌های داده-موازی خواهند بود، همانطوریکه در بخش ۱-۱-۱۰ شرح داده شد. زیرا پردازنده‌ها، مجموعهٔ مشابهی از دستورالعملها را روی عناصر مختلف یک مجموعهٔ داده‌ای اجرا می‌کنند. مجموعهٔ داده‌ای در حافظه اشتراکی ذخیره خواهد شد. اگر دستورالعملی مقدار یک عنصر این مجموعهٔ داده‌ای را به یک متغیر محلی (local) نسبت دهد، می‌گوئیم عمل خواندن از حافظهٔ اشتراکی صورت گرفته است. در صورتی که اگر دستورالعملی مقدار یک متغیر محلی را به یک عنصر از مجموعهٔ داده‌ای نسبت دهد، به آن نوشتن در حافظه اشتراکی می‌گوئیم. دستورالعملهایی که ما برای دستکاری عناصر این مجموعهٔ داده‌ای استفاده می‌کنیم، خواندن از/نوشتن به حافظهٔ اشتراکی است. به عنوان مثال، ما هرگز دو عنصر از مجموعه داده‌ای را مستقیماً مقایسه نمی‌کنیم، بلکه ترجیحاً مقادیرشان را به متغیرهایی در حافظهٔ محلی خوانده، سپس مقادیر آن متغیرها را با هم مقایسه می‌کنیم. یک الگوریتم داده-موازی، دنباله‌ای از مراحل را شامل می‌شود که هر پردازنده می‌تواند هر مرحله را در زمان مشابهی شروع کرده و آن را در زمان مشابهی به پایان برساند. علاوه بر این، همهٔ پردازنده‌ها در طول یک مرحله معین خواندن، در زمان مشابهی می‌خوانند و همهٔ پردازنده‌ها در طول مرحلهٔ معین نوشتن، در زمان مشابهی می‌نویسند.

در نهایت، فرض می‌کنیم که همواره پردازنده‌هایی که ما نیاز داریم، همیشه در دسترس ما هستند که این امر در عمل همانطوری که قبلاً ذکر شد، یک فرض غیرواقعی است.

الگوریتم زیر، این مطلب را نشان می‌دهد. فرض می‌کنیم اعداد صحیح S با شاخصهای ۱ تا n در حافظهٔ اشتراکی و n پردازنده با شاخصهای ۱ تا n در حال اجرای الگوریتم به صورت موازی هستند.

```
void example (int n,                    // S یک مجموعه داده‌ای در حافظه اشتراکی است
              int S[ ])
{
  local index p;
  local int temp;
  P = index of this processor;
  read S[p] int temp;                   //خواندن از حافظه اشتراکی
  if (p < n)
    write temp into S[p+1];
  else
    write temp into S[1];
}
```

همه مقادیر آرایهٔ S به طور همزمان به n متغیر منطقی مختلف tamp خوانده می‌شوند. آنگاه مقادیر موجود در n متغیر temp، مجدداً به طور همزمان در S نوشته می‌شوند. به طور مؤثر، به هر عنصر آرایه S، مقدار پردازنده متناظرش داده می‌شود. شکل ۹-۱۰، عملکرد الگوریتم را نشان می‌دهد. توجه کنید که هر پردازنده همیشه به تمام آرایه S دست یابد زیرا S در حافظه اشتراکی قرار دارد. بنابراین، پردازنده Pام



شکل ۹-۱۰ کاربرد روال exmaple.

می‌تواند در p+۱امین اندیس آرایه نوشته شود. در اینجا فقط یک مرحله در این الگوریتم ساده وجود دارد. هنگامی که بیش از یک مرحله وجود داشته باشد، حلقه‌هایی را نظیر نمونه زیر می‌نویسیم:

```
for (step=1 ; step <= numsteps; step++){
  کدی که باید در هر مرحله اجرا شود در اینجا قرار می گیرد.//
}
```

این حلقه می‌تواند به طرق مختلفی بکار گرفته شود. یک روش، داشتن یک واحد کنترلی مجزا است که روالهای افزایش و تست را انجام می‌دهد. این واحد باید دستورالعملهایی را ارسال کند تا به دیگر پردازنده‌ها بگوید چه زمانی بخوانند، چه زمانی بنویسند و چه زمانی دستورالعملها را روی متغیرهای محلی اجرا نمایند. درون حلقه، گاهی اوقات محاسباتی را روی متغیری که همواره مقدار یکسانی برای همه پردازنده‌ها دارد، انجام می‌دهیم. به عنوان مثال، هر دو الگوریتم ۱-۱۰ و ۳-۱۰ دستورالعمل زیر را اجرا می‌کنند.

```
size = ۱ * size;
```

که در آن متغیر size، مقدار یکسانی برای همه پردازنده‌ها دارد. برای روشن شدن این نکته که هر پردازنده به کپی خودش از چنین متغیری نیاز ندارد، فرض می‌کنیم که هر متغیر مانند یک متغیر در حافظهٔ اشتراکی است. دستورالعمل، با داشتن یک واحد کنترلی مجزا که آن را اجرا می‌کند، انجام می‌شود. ما پیش از این به نحوهٔ بکارگیری و اجرای این مدل نمی‌پردازیم. ترجیحاً سعی می‌کنیم که الگوریتم هایی را از این مدل ارائه نمائیم.

۱-۲-۱۰ طراحی الگوریتم‌ها برای مدل CREW PRAM

ما الگوریتم‌های CREW PRAM را با نمونه مسائل زیر توضیح خواهیم داد.

یافتن بزرگترین کلید در یک آرایه

قضیه ۷-۸ ثابت می‌کند که برای پیدا کردن بزرگترین کلید در یک آرایه، تنها به وسیله مقایسه کلیدها، حداقل n-۱ مقایسه انجام می‌شود. بدین معنی که هر الگوریتمی که برای این مسئله جهت اجرا روی یک کامپیوتر ترتیبی طراحی شده باشد می‌بایست از نوع $\theta(n)$ باشد. با استفاده از محاسبهٔ موازی می‌توانیم این زمان اجرا را بهبود ببخشیم. الگوریتم موازی هنوز هم باید لااقل n-۱ مقایسه را انجام دهد. اما با انجام دادن بسیاری از این مقایسه‌ها به موازات هم، عمل مقایسه زودتر به اتمام می‌رسد. ما این الگوریتم را در آینده ارائه خواهیم داد. به یاد آورید که الگوریتم ۲-۸ (پیدا کردن بزرگترین کلید)، بزرگترین کلید را در مدت زمان مناسبی به صورت زیر پیدا می‌کرد:

```
void find_largest (int n,
                   Const keytypess[ ],
                   Keytype& large)
```

```
  index i;
  large = s[1];
  for (i = 2; i <= n; i++)
    if (s[i] > large)
      large = s[i];
}
```

استفاده از پردازنده‌های بیشتر در این الگوریتم، نتیجه‌ای را عاید ما نمی‌کند زیرا حاصل هر تکرار از حلقه، برای تکرار بعدی لازم خواهد بود. از بخش ۳-۵-۸، روش تورنمنت برای پیدا کردن بزرگترین کلید را به خاطر آورید. این روش، ابتدا اعداد را به گروههای ۲تایی تبدیل نموده و بزرگترین مقدار از هر جفت را پیدا می‌کند. سپس این مقادیر بزرگتر را به گروههای دوتایی تقسیم کرده و بزرگترین مقدار در هر یک از این جفتها را پیدا می‌کند. این کار، تا زمانی که تنها یک کلید باقی بماند ادامه می‌یابد. شکل ۱۰-۸، این روش را تشریح می‌کند. یک الگوریتم ترتیبی برای روش تورنمنت، پیچیدگی زمانی مشابه الگوریتم ۲-۸ دارد. روشن است که این روش می‌تواند با استفاده از پردازنده‌های بیشتر، نتیجه بهتری بدست آورد. بعنوان مثال، فرض می‌کنیم که بخواهید با استفاده از این روش، بزرگترین کلید را از هشت کلید موجود پیدا کنید. شما بایستی قبل از اینکه به دور دوم راه یابید، چهار برنده را به ترتیب در دور اول مشخص کرده باشید. اگر به کمک سه نفر از دوستانتان دور اول را آغاز کرده باشید. هر یک از شما می‌تواند به طور همزمان، یکی از برنده‌های دور اول را مشخص نماید؛ به این معنی که دور اول می‌تواند چهار مرتبه سریعتر از حالت ترتیبی انجام شود. بعد از این دور، دو نفر از شما می‌توانند استراحت کرده و دو نفر دیگر مقایسات لازم در دور دوم را انجام دهند و در دور نهایی، تنها یکی از شما برای انجام آخرین مقایسه کفایت می‌کند.

شکل ۱۰-۱۰، چگونگی عملکرد یک الگوریتم موازی برای این روش را نشان می‌دهد. ما تنها به اندازهٔ نیمی از تعداد عناصر آرایه، به پردازنده نیازمندیم. هر پردازنده، دو عنصر آرایه را به متغیرهای منطقی first و second خوانده، آنگاه عنصر بزرگتر میان first و second را در اولین اندیس آرایه از همان جائیکه آن را خوانده است، می‌نویسد. بعد از سومین دور این چنینی، بزرگترین کلید در S[1] قرار می‌گیرد. هر دور، یک مرحله در الگوریتم است. در مثال نشان داده شده در شکل ۱۰-۱۰، n=۸ است، لذا در آنجا $lg\ 8 = 3$ مرحله وجود خواهد داشت. الگوریتم ۱-۱۰، یک الگوریتم برای عملیات نشان داده شده در شکل ۱۰-۱۰ می‌باشد. توجه کنید که این الگوریتم به صورت یک تابع نوشته شده است. هنگامی که یک الگوریتم موازی به صورت تابع نوشته شود، لازم است که حداقل یک پردازنده مقداری را برگردانده و همهٔ پردازنده‌هایی که عمل بازگرداندن مقادیر را انجام می‌دهند نیز مقدار مشابهی را برگردانند.

الگوریتم ۱-۱۰ یافتن بزرگترین کلید به روش موازی

مسئله: بزرگترین کلید در آرایهٔ n عنصر S را پیدا کنید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S با شاخصهای ۱ تا n.

خروجی: مقدار بزرگترین کلید در آرایهٔ S.

توضیح: فرض کنید که n توانی از ۲ است و تعداد n/۲ پردازنده در حال اجرای الگوریتم به صورت موازی می‌باشند. پردازنده‌ها از ۱ تا n/۲ شاخص‌دهی شده و دستور "index of this processor"، شاخص یک پردازنده را برمی‌گرداند.

```
keytype parlargest (int n, keytype S[ ])
{
  index step, size;
  local index p;
  local keytype first, second;
  p=index of this processor;
  size = 1;
  for (step = 1; step <= lg n; step++)
    if (this processor needs to execute in this step){
      read S[2 × p - 1] into first;
      read S[2 × p - 1 + size] into second;
      write maximum (first, second) into S[2 * p - 1];
      S = 2 × size;
    }
  retern S[1];
}
```

شبه‌کد سطح بالای "if(this processor needs to execute in this step)" را به ترتیبی بکار بردیم که الگوریتم را به حداکثر وضوح ممکن برساند. در شکل ۱۰-۱۰ می‌بینیم که برای هر یک از پردازنده‌هایی که در یک مرحلهٔ معین استفاده می‌شوند، دستورالعمل

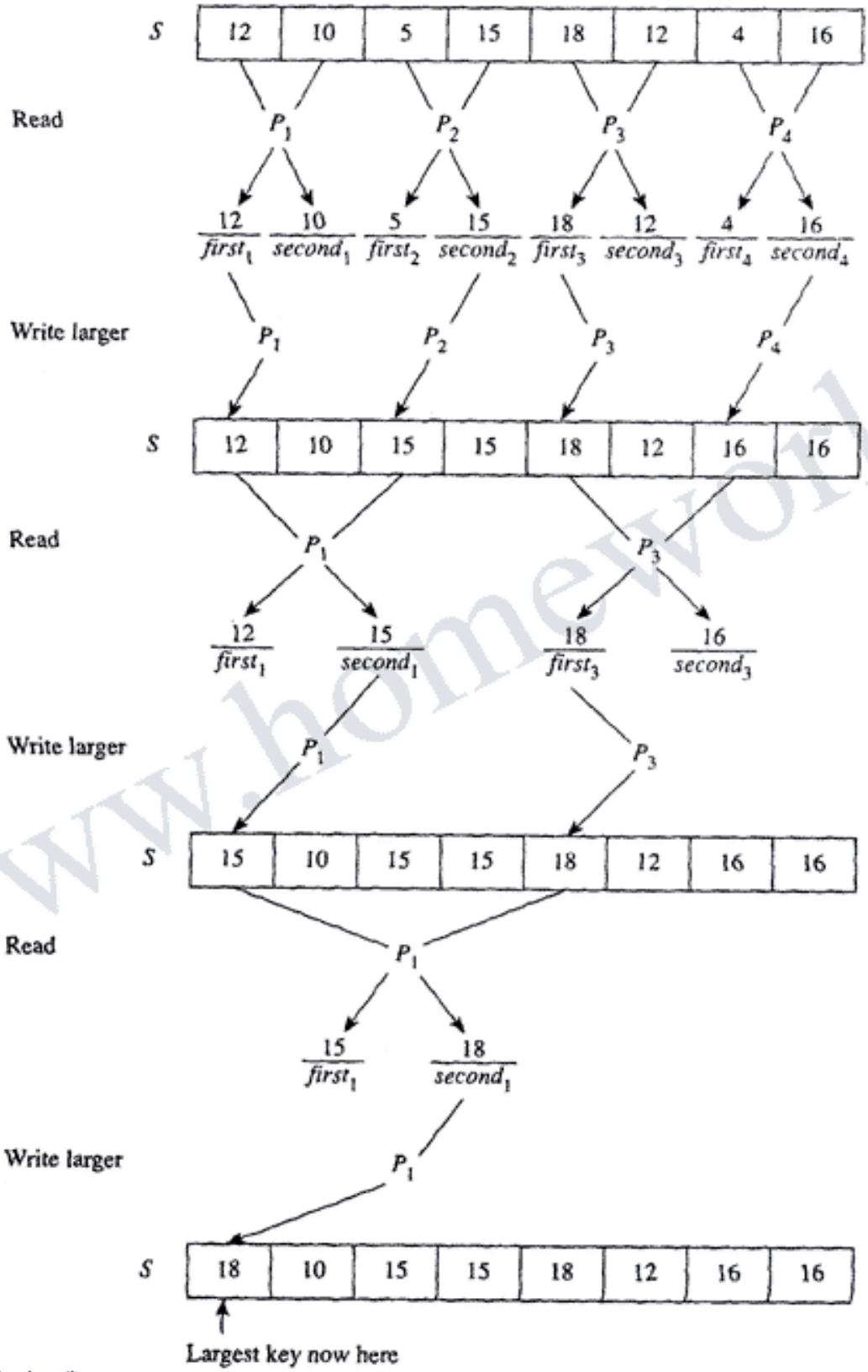
$$p = 1 + size * k$$

اجرا می‌شود که در آن k یک عدد صحیح است (توجه داشته باشید که مقدار size در هر مرحله دو برابر می‌شود). شرط واقعی در مورد اینکه «آیا پردازنده‌ای باید اجرا شود یا خیر؟»، چنین است:

$$if\ ((p-1)\%size == 0) \quad //\ \%\text{ باقیماندهٔ تقسیم } p-1 \text{ بر size را برمی‌گرداند}$$

به طور متناوب می‌توانیم همه پردازنده‌ها را برای اجرا در هر مرحله در نظر بگیریم. آن پردازنده‌ای که لازم نیست اجرا شود، مقایسات بیهوده‌ای را انجام می‌دهد. همواره آن چیزهای مهم هستند که پردازنده‌های در حال اجرا آن را اجرا می‌کنند و پردازنده‌های دیگر که در حال تغییر مکانهای حافظه نیستند، اهمیتی ندارند. هنگامی که یک الگوریتم موازی را تجزیه و تحلیل می‌کنیم، مقدار کل کار انجام شده به وسیلهٔ الگوریتم، تحلیل نمی‌شود؛ بلکه مجموع کار انجام شده توسط هر یک از پردازندها را مورد بررسی قرار می‌دهیم؛ زیرا این کار، ما را از سرعت کامپیوتری که ورودی را پردازش می‌کند، مطلع خواهد کرد. از آنجائیکه هر پردازنده، تقریباً $log\ n$ گذر از حلقهٔ for_step انجام می‌دهد، لذا داریم $T(n) \in \theta(lg\ n)$.

شکل ۱۰-۱۰ استفاده از پردازنده‌های موازی به روش تورنمنت، جهت پیدا کردن بزرگترین کلید.



کاربردهای برنامه‌نویسی پویا

بسیاری از کاربردهای برنامه‌نویسی پویا، تابع طراحی موازی هستند. چرا که اغلب، تمامی ورودیها می‌توانند به طور همزمان در یک ردیف معین به صورت سطری یا قطری محاسبه شوند. این روش را با بازنویسی الگوریتم ضریب دوجمله‌ای (الگوریتم ۲-۳) به صورت موازی، تشریح می‌کنیم. در این الگوریتم، ورودی‌ها در یک ردیف معین از مثلث پاسکال (به شکل ۱-۳ توجه کنید) به موازات هم محاسبه می‌شوند.

الگوریتم ۲-۱۰ محاسبه ضریب دوجمله‌ای به روش موازی

مسئله: ضریب دوجمله‌ای را محاسبه کنید.

ورودی: اعداد صحیح غیرمنفی n و k که در آن $k \leq n$ است.

خروجی: ضریب دوجمله‌ای $\binom{n}{k}$.

توضیح: فرض می‌شود که k+۱ پردازنده در حال اجرای موازی الگوریتم هستند. پردازنده‌ها از ۰ تا k شاخص‌دهی شده‌اند و دستورالعمل "index of this processor" شاخص یک پردازنده را برمی‌گرداند.

```
int parbin (int n, int k)
{                                        // Use i instead of step to
   index i;                              // control the steps to be
   int B[0...n][0..k];                   // control with Algorithm 3.2.
   local index j;                           // Use j instead of p to obtain
   local int first, second;              // consistent with Algorithm 3.2
   j = index of this processor;
   for (i = 0; i <= n; i++)
     if (j == 0 || j == i)
       write 1 into B[i][j];
     else{
       read B[i-1][j-1] into first;
       read B[i-1][j] into second;
        write first + second into B[i][j];
      }
   return B[n][k];
}
```

عبارت کنترلی بکار رفته در الگوریتم ۲-۳ یعنی for(j = ۰; j <= minimum(i, k); j++) می‌تواند جایگزین عبارت for (j <= minimum(i, k)) در این الگوریتم شود؛ زیرا تمامی k پردازنده موجود در هر گذر از حلقهٔ for_i اجرا می‌شوند. به جای محاسبهٔ ترتیبی مقادیر B[i][j] با j در محدوده صفر تا minimium(i, k)، الگوریتم موازی پردازنده‌هایی دارد که از صفر تا minimum(i, k) شاخص‌دهی شده و به طور همزمان مقادیر را محاسبه می‌کنند. به عبارت بهتر، در یک حلقه از الگوریتم موازی، n+۱ گذر انجام

می‌شود؛ در حالیکه در الگوریتم ترتیبی(الگوریتم ۲-۳)، $\theta(nk)$ گذر از یک حلقه وجود دارد.

از تمرین ۴-۳ به خاطر دارید که بکارگیری ۲-۳ تنها با استفاده از یک آرایهٔ یک‌بعدی B که از صفر تا K شاخص‌دهی شده است، امکان‌پذیر می‌باشد. این تغییر در الگوریتم‌های موازی بسیار ساده است. زیرا در ورودی نام حلقهٔ for_i، تمام سطر (i-۱)ام از مثلث پاسکال می‌تواند B به K زوج محلی از متغیرهای first و second خوانده شود، سپس در خروجی، تمام سطر نام می‌تواند در B نوشته شود. شبه‌کد مورد نظر به صورت زیر است:

```
for (i = 0; i <= n; i++)
  if (j <= mimimum(i, k)
    if (j == 0 || j == i)
      write 1 into B[j];
    else
      read B[j-1] into first;
      read B[j] into second;
      write first+second into B[j];
    }
return B[k];
```

مرتب‌سازی موازی

بررسی اینکه آیا یک پردازنده باید در یک مرحلهٔ معین اجرا شود، شبیه به بررسی در الگوریتم ۱-۱۰ است؛ بدینصورت که لازم است بررسی زیر انجام شود:

```
if((p-۱) % size == ۰)
```

نسخهٔ برنامه‌نویسی پویا از مرتب سازی ادغامی (الگوریتم ۳-۷) را به خاطر آورید. این الگوریتم به سادگی با کلیدهایی به صورت منفرد شروع شده، زوج کلیدها را به لیستهای مرتب شده شامل ۲ کلید ادغام نموده، آنگاه این لیستها را به لیستهای شامل ۴ کلید ادغام می‌کند و الی آخر. این روند را در شکل ۲-۲ نشان دادیم. روش فوق بسیار شبیه به بکارگیری روش تورنمنت برای پیدا کردن بزرگترین مقدار است. بدینصورت که ما می‌توانیم عملیات ادغام در هر مرحله را به موازات هم انجام دهیم. الگوریتم زیر از این روش استفاده می‌کند. مجدداً برای سادگی فرض می‌کنیم که n توانی از ۲ است. در غیراینصورت، با اندازهٔ آرایه می‌توان همانند توانی از ۲ رفتار نمود اما توجه داریم که ادغام خارج از محدودهٔ n انجام نمی‌شود. نسخهٔ برنامه‌نویسی پویا از مرتب‌سازی ادغامی۳ (الگوریتم ۳-۷) چگونگی انجام آن را نشان می‌دهد.

الگوریتم ۳-۱۰ مرتب سازی ادغامی موازی

مسئله: n کلید را به صورت غیرنزولی مرتب نمائید.

ورودی: عدد صحیح مثبت n، آرایه‌ای از کلیدها S که از ۱ تا n شاخص‌دهی شده است.

خروجی: آرایهٔ S شامل کلیدهایی که به صورت غیرنزولی مرتب شده‌اند.

توضیح: فرض شده است که n توانی از ۲ بوده و n/۲ پردازنده در حال اجرای موازی الگوریتم هستند. پردازنده‌ها از ۱ تا n شاخص‌دهی شده‌اند.

```
void parmergesort (int n, keytype S[ ])
{
   index step, size;
   local index p, low, mid, high;

   p = index of this processor;
   size = 1;                                  // size is the size of the subarrays
   for (step = 1; step <= lgn; step++)        // being merged.
      if (this processor needs to execute in this step) {
         low = 2*p - 1;
         mid = low + size - 1;
         high = low + 2*size - 1;
         parmerge(low, mid, high, S);
         size = 2 * size;
      }
}

void parmerge (local index low, local index mid, local index high,
               keytype S[ ])
{
   local index i, j, k;
   local keytype first, second, U[low..high];

   i = low; j = mid + 1; k = low;
   while (i <= mid && j <= high) {
      read S[i] into first;
      read S[j] into second;
      if (first < second) {
         U[k] = first;
         i++;
      }
      else {
         U[k] = second;
         j++;
      }
      k++;
   }
    if (i > mid)
      read S[j] through S[high] into U[k] through U[high];
    else
      read S[i] through S[mid] into U[k] through U[high];
    write U[low] through U[high] into S[low] through S[high];
}
```

همانطوریکه می‌دانید ما می‌توانیم تعداد انتساب رکوردها در نسخهٔ ترتیبی (تک پردازنده‌ای) مرتب‌سازی ادغامی (الگوریتم ۳-۷) را با جابجایی نقش U و S در هر گذر از حلقهٔ **for**، کاهش دهیم. در اینجا نیز می‌توانستیم چنین عمل اصلاحی انجام دهیم که در اینصورت می‌بایست U را به عنوان آرایه‌ای که از ۱ تا n شاخص‌دهی شده در حافظه اشتراکی داشته باشیم. پیچیدگی زمانی این الگوریتم، چندان واضح و روشن نیست. بنابراین، یک تحلیل صوری از این الگوریتم ارائه می‌دهیم.

تحلیل پیچیدگی زمانی بدترین حالت الگوریتم ۳-۱۰

عمل مبنایی: مقایسه‌ای که در parmerge انجام می‌شود.

اندازه ورودی: n، تعداد کلیدهای آرایه.

این الگوریتم دقیقاً تعداد مقایساتی را انجام می‌دهد که در مرتب‌سازی ادغامی ترتیبی به طور عادی انجام می‌شود؛ با این تفاوت که بسیاری از آنها به موازات هم اجرا می‌شوند. در اولین گذر از حلقه for_step، n/۲ از زوجهای آرایه، هر یک شامل تنها یک کلید، به طور همزمان با هم ادغام می‌شوند. بنابراین، بدترین حالت تعداد مقایساتی که توسط هر پردازنده انجام می‌شود برابر ۱-۱=۲-۲ است. (به تحلیل الگوریتم ۳-۲ در بخش ۲-۲ توجه کنید.) در دومین گذر، n/۴ از زوجهای آرایه هر یک شامل دو کلید به طور همزمان با هم ادغام می‌شوند. بنابراین، بدترین حالت تعداد مقایسات برابر ۳=۱-۴ است. در گذر سوم حلقه، n/۸ از زوجهای آرایه، هر یک شامل ۴ کلید، به طور همزمان با هم ادغام می‌شوند. لذا بدترین حالت تعداد مقایسات برابر ۷=۱-۸ است. به طور کلی درگذر iام حلقه، $n/2^i$ از زوجهای آرایه، هر یک شامل 2^{i-1} است و سرانجام در آخرین گذر از حلقه، دو آرایه هر یک شامل $2^{i}-1$ کلید با هم ادغام می‌شوند. به این معنی که بدترین حالت تعداد مقایسات در این گذر برابر n-۱ است. مجموع بدترین حالت تعداد مقایسات انجام شده توسط هر پردازنده برابراست با

$$W(n) = 1 + 3 + 7 + \ldots + 2^i - 1 + \ldots + n - 1$$

$$= \sum_{i=1}^{\lg n} (2^i - 1) = 2n - 2 - \lg n \in \theta(n)$$

آخرین تساوی از نتیجه مثال A-۳ در ضمیمه A و برخی محاسبات جبری حاصل می‌شود.

ما توانسته‌ایم مرتب سازی موازی را با مقایسه کلیدها در یک زمان خطی، که یک ساختار مطلوب و قابل توجه نسبت به $\theta(n \lg n)$ است، به انجام برسانیم. ما می‌توانیم الگوریتم ادغام موازی نوشته شده را نیز اصلاح کنیم، بطوری که مرتب‌سازی ادغامی موازی در $\theta((n \lg n)^2)$ انجام شود. چگونگی این اصلاح را در تمرینات خواهیم دید. پرواضح است که این زمان، یک زمان بهینه نیست، زیرا مرتب‌سازی موازی می‌تواند در زمان $\theta(\lg n)$ انجام شود.

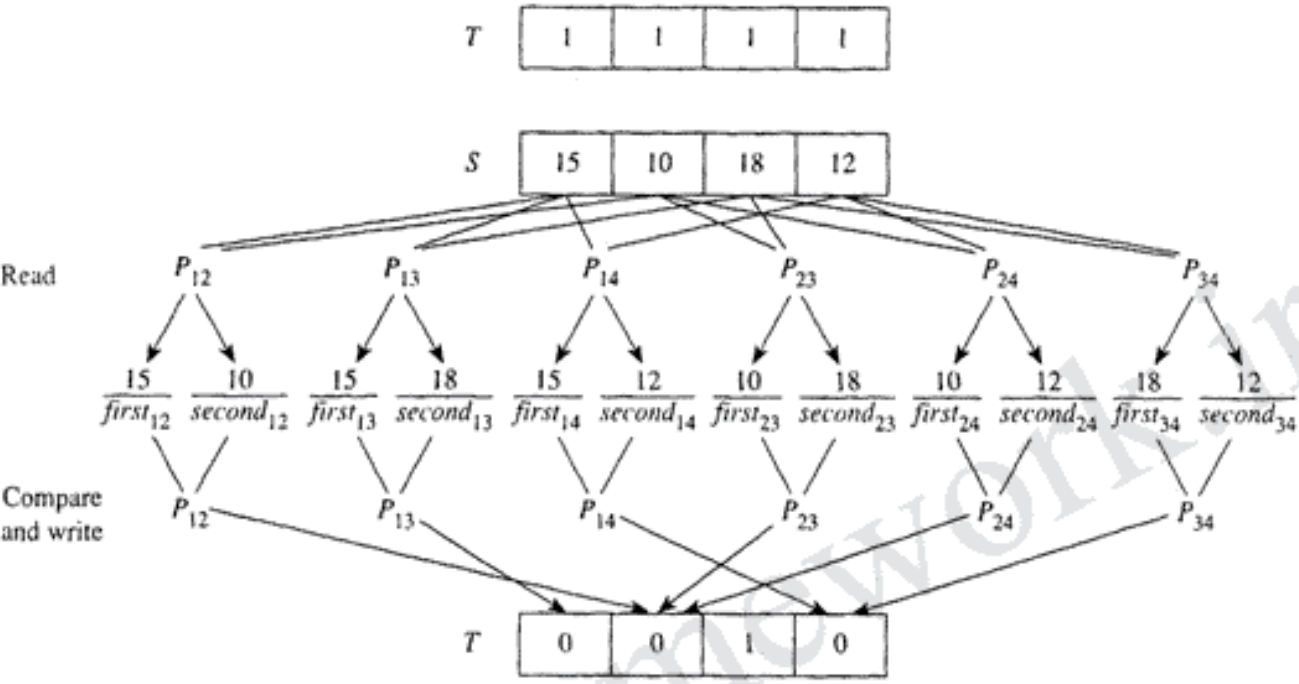
۲-۲-۱۰ طراحی الگوریتم‌ها برای مدل CREW PRAM

به خاطر دارید که CRCW مخفف خواندن همزمان و نوشتن همزمان است. بجز حالت CRCW، زمانی که دو پردازنده سعی می‌کنند در مرحلهٔ یکسانی در یک مکان حافظه مشابه بنویسند، بایستی به دقت مورد بررسی قرار گیرد. اغلب، پروتکلهای استفاده شده جهت بررسی برخوردهای (confict) این چنینی،به صورت زیر می‌باشند:

- **مشترک.** این پروتکل، نوشتنهای همزمان را تنها اگر همه پردازنده‌ها بخواهند مقادیر یکسانی را بنویسند، می‌پذیرد.
- ***اختیاری.*** این پروتکل یک پردازندهٔ دلخواه که برای نوشتن در یک مکان حافظه منظور می‌شود را انتخاب می‌کند.
- ***اولویت.*** در این پروتکل، همه پردازنده‌ها در یک لیست اولویت از پیش تعریف شده سازماندهی می‌شوند و تنها پردازنده‌ای که بالاترین اولویت را دارد، جهت نوشتن پذیرفته می‌شود.
- **مجموع.** این پروتکل، مجموع مقادیری که باید توسط پردازنده‌ها نوشته شوند را می‌نویسد. این پروتکل می‌تواند به هر عملگر شرکت‌پذیر از پیش تعریف شده بر روی مقادیری که باید نوشته شوند، توسعه یابد.

ما یک الگوریتم برای پیدا کردن بزرگترین کلید در یک آرایه که با پروتکلهای نوشتن-مشترک، نوشتن-اختیاری و نوشتن-اولویت کار می‌کند می‌نویسیم که این الگوریتم سریعتر از الگوریتم ۱-۱۰ عمل خواهد کرد. بدینصورت که فرض می‌کنیم n کلید در یک آرایهٔ S در حافظهٔ اشتراکی قرار دارند. آرایهٔ دوم T، متشکل از n عدد صحیح را در حافظه اشتراکی تعریف نموده و همه عناصر T را با عدد یک مقداردهی می‌کنیم. سپس فرض می‌کنیم که از $n(n-1)/2$ پردازنده که به صورت P_{ij} $(1 \le i \le j \le n)$ شاخص‌دهی شده‌اند، می‌توانیم استفاده نمائیم. در روش موازی، پردازنده‌ها، S[i] را با S[j] مقایسه می‌کنند. به عبارت دیگر، در این روش هر عنصر در S با هر عنصر دیگر در S مقایسه می‌شود، هر پردازنده، یک صفر در T[i] می‌نویسد اگر S[i] در مقایسه شکست بخورد و یک صفر در T[j] می‌نویسد اگر S[j] مقایسه را ببازد. تنها بزرگترین کلید، هرگز در مقایسه‌ای بازنده نمی‌شود. از اینرو، تنها عنصر T که مساوی با ۱ باقی می‌ماند، عنصری است که با k شاخص‌دهی شده است بطوری که S[k] دارای بزرگترین کلید آرایه می‌باشد. بنابراین، کافی است که الگوریتم، مقدار S[k]ای را بازگرداند که T[k] متناظر آن برابر یک باشد. شکل ۱۱-۱۰ و الگوریتم ۴-۱۰ این مراحل را به خوبی شرح می‌دهند. توجه کنید که در الگوریتم زیر، هنگامی که بیش از یک پردازنده در مکان حافظه یکسان می‌نویسند، مقادیر نوشته شده یکسان می‌باشند.

شکل ۱۱-۱۰ یک اجرا از الگوریتم ۳-۱۰. فقط T[۲] به ۱ ختم می‌شود زیرا S[۲] بزرگترین کلید است و بنابراین هرگز مقایسه‌ای را از دست نمی‌دهد.



ریتم ۴-۱۰ پیدا کردن بزرگترین کلید به روش CRCW موازی

مسئله: بزرگترین کلید را در آرایهٔ n کلیدی S پیدا کنید.

ورودی: عدد صحیح مثبت n آرایه‌ای از کلیدها S با شاخصهای ۱ تا n.

خروجی: مقدار بزرگترین کلید در آرایهٔ S.

توضیح: فرض می‌شود که n توانی از ۲ است و $n(n-1)/2$ پردازندهٔ در حال اجرای الگوریتم به صورت موازی هستند. پردازنده‌ها به صورت $(1 \le i < j \le n)$ شاخص‌دهی شده‌اند و دستور "first index of this processor" مقدار i و دستور "second index of this processor" مقدار j را باز می‌گردانند.

```
keytype parlargest2 (int n, const keytype S[ ])
{
  int T[1..n];
  local index i, j;
  local keytype first, second;
  local int chkfrst, chkscnd;

  i = first index of this processor;
  j = second index of this processor;
  write 1 into T[i];                    // Because 1 ≤ i ≤ n − 1 and
```

```
    write 1 into T[j];                  // 2 ≤ j ≤ n, these write
    read S[i] into first;               // instructions initialize each
    read S[j] into second;              // array element of T to 1.
    if (first < second)
      write 0 into T[i];                // T[k] ends up 0 if and only
    else                                // if S[k] loses at least one
      write 0 into T[j];                // comparison.
    read T[i] into chkfrst;
    read T[j] into chkscnd;
    if (chkfrst == 1)                   // T[k] still equals 1 if and only
      return S[i];                      // if S[k] contains the largest key.
    else if (chkscnd = 1)               // Need to check T[j] in case
      return S[j];                      // the largest key is S[n]. Recall
}                                       // i ranges in value only from
                                        // 1 to n − 1.
```

در این الگوریتم، هیچ حلقه‌ای وجود ندارد. این بدین معنی است که الگوریتم، بزرگترین کلید را در مدت زمان ثابتی پیدا می‌کند و این امر بسیار مؤثر است. چراکه در اینصورت می‌توانستیم بزرگترین کلید را در بین ۱،۰۰۰،۰۰۰ کلید در مقدار زمانی مشابهی که برای پیدا کردن بزرگترین کلید از ۱۰ کلید صرف می‌شود، پیدا کنیم. به هر حال، این پیچیدگی زمانی مطلوب از هزینهٔ پیچیدگی پردازنده‌ای زمان‌مربعی بدست می‌آید. یعنی برای پیدا کردن بزرگترین کلید در میان ۱،۰۰۰،۰۰۰ کلید به $۱،۰۰۰،۰۰۰^۲/۲$ پردازنده نیاز خواهیم داشت. این فصل تنها به عنوان مقدمه‌ای بر معرفی الگوریتمهای موازی مطرح شده است. بحثهای تکمیلی در این مورد را می‌توانید در کتاب Kumar (۱۹۹۴) پیدا کنید.

تمرینات

بخش ۱-۱۰

۱- با فرض اینکه یک شخص می‌تواند دو عدد را در مدت زمان t_a با هم جمع کند، چه مدتی طول می‌کشد تا آن شخص دو ماتریس n×n را، با درنظرگرفتن عمل جمع به عنوان عمل مبنایی با هم جمع نماید؟ صحت جواب خود را بررسی کنید.

۲- با فرض اینکه یک شخص می‌تواند دو عدد را در مدت زمان t_a با هم جمع کند، چه مدت زمانی لازم است تا دو نفر بتوانند دو ماتریس n×n را، با درنظر گرفتن عمل جمع به عنوان عمل مبنایی با هم جمع نمایند؟ صحت جواب خود را بررسی کنید.

۳- با فرض اینکه یک شخص می‌تواند دو عدد را در مدت زمان t_a با هم جمع کند، چند نفر لازم است

تابتوانند مجموع زمان صرف شده برای جمع دوماتریس n×n را به حداقل برسانند؟ صحت جواب خود را بررسی کنید.

۴- با فرض اینکه یک شخص می‌تواند دو عدد را در مدت زمان t_a با هم جمع کند، چه مدت زمانی لازم است تا آن شخص بتواند n عدد در یک لیست را، با درنظر گرفتن عمل جمع به عنوان عمل مبنایی، با هم جمع نماید؟ صحت جواب خود را بررسی کنید.

۵- با فرض اینکه یک شخص می‌تواند دو عدد را در مدت زمان t_a با هم جمع کند. چه مدت زمانی لازم است تا دو نفر بتوانند n عدد در یک لیست را، با درنظر گرفتن عمل جمع بعنوان عمل مبنایی و مدت زمان t_p برای ارسال نتیجه از یکی به دیگری، با هم جمع نماید؟ صحت جواب خود را بررسی کنید.

۵- با فرض اینکه یک شخص می‌تواند دو عدد را در مدت زمان t_a با هم جمع کند و ارسال نتیجه از یکی به دیگری، به مدت زمان t_p نیاز دارد، تعیین کنید چند نفر لازم است تا مجموع زمان صرف شده برای بدست آوردن نتیجه نهایی جمع n عدد در یک لیست را به حداقل برسانند؟ صحت جواب خود را بررسی کنید.

بخش ۲-۱۰

۷- یک الگوریتم CREW PRAM، برای جمع n عدد در یک لیست بنویسید که از کارایی $\theta(lg\, n)$ برخوردار باشد.

۸- یک الگوریتم CREW PRAM بنویسد که از n^2 پردازنده برای ضریب دو ماتریس n×n استفاده کند. الگوریتم شما بایستی بهتر از الگوریتم ترتیبی استاندارد $\theta(n^3)$ باشد.

۹- یک الگوریتم PRAM، برای مرتب‌سازی سریع یک لیست n عنصری با استفاده از n پردازنده بنویسید.

۱۰- یک الگوریتم ترتیبی بنویسید که با استفاده از روش تورنمنت، بزرگترین کلید در یک آرایه n کلیدی را پیدا کند. نشان دهید که این الگوریتم کاراتر از الگوریتم ترتیبی استاندارد نیست.

۱۱- یک الگوریتم PRAM بنویسید که با استفاده از n^2 پردازنده، دو ماتریس n×n را ضرب کند. الگوریتم شما بایستی در زمان $\theta(lgn)$ اجرا شود.

۱۲- یک الگوریتم PRAM، برای مسئله درخت جستجوی دودویی مطلوب بخش ۵-۳ بنویسید. کارایی این الگوریتم را با الگوریتم درخت جستجوی دودویی مطلوب (الگوریتم ۹-۳) مقایسه کنید.

تمرینات اضافی

۱۳- یک الگوریتم PRAM برای مسئله مرتب‌سازی ادغامی (Mergesort) بنویسید که در زمان $\theta((lgn)^2)$ اجرا شود. (**نکته:** از n پردازنده استفاده نمائید و هر پردازنده را به یک کلید نسبت دهید)

۱۴- یک الگوریتم PRAM، برای مسئله فروشندهٔ دوره‌گرد بخش ۶-۳ بنویسید. کارایی این الگوریتم را با الگوریتم فروشندهٔ دوره‌گرد (الگوریتم ۱۱-۳) مقایسه کنید.

ضمیمهٔ A

مروری بر ریاضیات ضروری

در مطالعه این کتاب به استثناء بخش‌هایی که با علامت ❁ مشخص شده‌اند، نیازی به یک پیش‌زمینهٔ کامل از ریاضیات نداریم. در واقع، ما فرض نموده‌ایم که شما با ریاضیات آشنایی زیادی ندارید؛ اگرچه در تحلیل الگوریتم‌ها لازم است که تا حدود زیادی با ریاضیات آشنا باشیم.

A-۱ نمادگذاری

گاهی اوقات لازم است که به کوچکترین عدد صحیح بزرگتر یا مساوی عدد حقیقی x، اشاره کنیم. در این صورت آن را با نماد $\lceil x \rceil$ نشان می‌دهیم. برای مثال،

$$\lceil ۶ \rceil = ۶ \qquad \lceil \tfrac{۹}{۲} \rceil = ۵ \qquad \lceil ۳/۳۱ \rceil = ۴$$

$$\lceil -۶ \rceil = -۶ \qquad \lceil -۳/۷۱ \rceil = -۳ \qquad \lceil -۳/۳۱ \rceil = -۳$$

نماد $\lceil x \rceil$ را **جزء صحیح بالای** x می‌گوئیم و برای هر عدد صحیح n داریم $\lceil n \rceil = n$. گاهی اوقات لازم است که به بزرگترین عدد صحیح کوچکتر یا مساوی عدد حقیقی x اشاره کنیم که در این صورت آن را با نماد $\lfloor x \rfloor$ نشان می‌دهیم. برای مثال،

$$\lfloor ۶ \rfloor = ۶ \qquad \lfloor \tfrac{۹}{۲} \rfloor = ۴ \qquad \lfloor ۳/۳۱ \rfloor = ۳$$

$$\lfloor -۶ \rfloor = -۶ \qquad \lfloor -۳/۷۱ \rfloor = -۴ \qquad \lfloor -۳/۳۱ \rfloor = -۴$$

نماد $\lfloor x \rfloor$ را **جزء صحیح پائین** x می‌گوئیم و برای هر عدد صحیح n داریم $\lfloor n \rfloor = n$.

هنگامی که فقط بتوانیم مقدار تقریبی یک نتیجهٔ مورد انتظار را تعیین کنیم از نماد $\approx$ که به معنای "تقریباً مساوی" است، استفاده می‌کنیم. به عنوان مثال، شما با عدد π که در محاسبهٔ محیط و مساحت یک دایره بکار می‌رود آشنا هستید. مقدار π با یک تعداد متناهی از ارقام دهدهی مشخص نمی‌شود. (در واقع، حتی یک الگوی مشخص برای پایان بخشیدن به این ارقام وجود ندارد، نظیر $\frac{۱}{۳} = ۰/۳۳۳۳۳۳۳ \ldots$). بنابراین، از آنجائیکه شش رقم اول π برابر ۳/۱۴۱۵۹ است، می‌نویسیم:

$$\pi \approx ۳/۱۴۱۵۹$$

از نماد $\neq$ به معنای "نامساوی" استفاده می‌کنیم. برای مثال، اگر بخواهید نشان دهید که مقدار متغیر x

با متغیر y مساوی نیست، می‌نویسیم:

$$x \neq y$$

اغلب لازم است که به مجموع عناصری مشابه و یکسان اشاره کنیم. این کار آسانی است اگر تعداد عناصر زیاد نباشد. مثلاً بخواهیم به مجموع اولین هفت عدد صحیح مثبت اشاره کنیم که در این صورت می‌نویسیم:

$$۱ + ۲ + ۳ + ۴ + ۵ + ۶ + ۷$$

یا بخواهیم به مجموع مربعات اولین هفت عدد صحیح مثبت اشاره کنیم که در این صورت می‌نویسیم:

$$۱^۲ + ۲^۲ + ۳^۲ + ۴^۲ + ۵^۲ + ۶^۲ + ۷^۲$$

همانطوری که قبلاً ذکر شد، استفاده از این روش زمانی میسر است که تعداد عناصر زیاد نباشد. واضح است که اگر بخواهیم به مجموع اولین صد عدد صحیح مثبت اشاره کنیم، استفاده از این روش غیرمنطقی خواهد بود. یک روش برای انجام این کار، نوشتن چند عنصر ابتدایی، یک عنصر عمومی و آخرین عنصر است. بدینصورت که بنویسیم:

$$۱ + ۲ + ... + i + ... ۱۰۰$$

و اگر به مجموع مربعات اولین صد عدد مثبت اشاره داشته باشیم، می‌نویسیم:

$$۱^۲ + ۲^۲ + ... + i^۲ + ... + ۱۰۰^۲$$

یک روش خلاصه‌تر برای نمایش مجموع عناصر، استفاده از حرف یونانی Σ (سیگما) است. به عنوان مثال، برای نمایش مجموع اولین صد عدد صحیح مثبت با استفاده از Σ می‌نویسیم:

$$\sum_{i=۱}^{۱۰۰} i$$

این نماد به این معناست که تا زمانی که متغیر i، مقادیر ۱ تا ۱۰۰ را به خود می‌گیرد، مقادیر آن با هم جمع می‌شوند. به طور مشابه، برای نمایش مجموع مربعات اولین صد عدد صحیح مثبت با استفاده از Σ می‌نویسیم:

$$\sum_{i=۱}^{۱۰۰} i^۲$$

اغلب، آخرین عدد صحیح در مجموع عناصر را به صورت عدد صحیح اختیاری n بیان می‌کنیم و در این صورت برای نمایش مجموع اولین n عدد صحیح مثبت می‌نویسیم:

$$۱ + ۲ + ... + i + ... + n \qquad \text{یا} \qquad \sum_{i=۱}^{۱۰۰} i$$

و برای نمایش مجموع مربعات اولین n عدد صحیح مثبت می‌نویسیم:

$$۱^۲ + ۲^۲ + ... + i^۲ + ... + n^۲ \qquad \text{یا} \qquad \sum_{i=۱}^{n} i^۲$$

گاهی اوقات به یک مجموع نیازمندیم، به عنوان مثال،

$$\sum_{i=1}^{4}\sum_{j=1}^{i} j = \sum_{j=1}^{1} j + \sum_{j=1}^{2} j + \sum_{j=1}^{3} + \sum_{j=1}^{4} j$$
$$= (1) + (1+2) + (1+2+3) + (1+2+3+4) = 20$$

به طور مشابه، می‌توانیم مجموع یک مجموع یک مجموع را بگیریم و الی آخر.

در نهایت، گاهی اوقات می‌خواهیم به موجودیتی اشاره کنیم که بزرگتر از هر عدد حقیقی است. ما آن را **بی‌نهایت** نامیده و به صورت ∞ نشان می‌دهیم. برای هر عدد حقیقی x داریم:

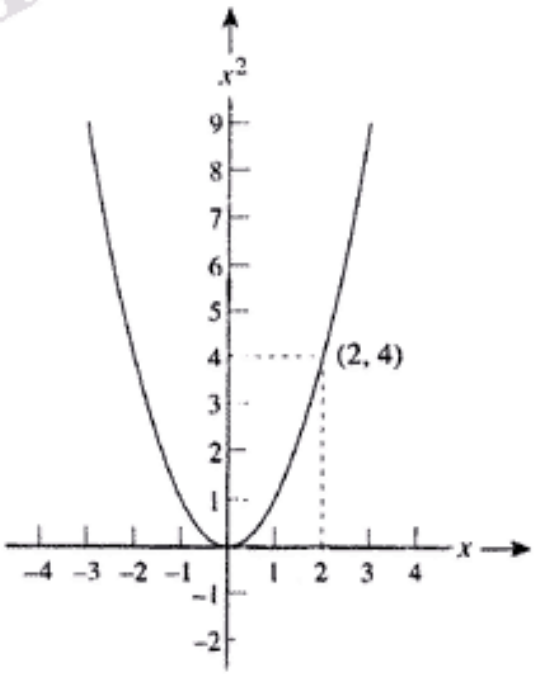
$$x < \infty$$

A-۲ توابع

تابع f از یک متغیر، قاعده و قانونی است که به ازاء یک مقدار x مقدار منحصربفردی را به $f(x)$ نسبت می‌دهد. به عنوان مثال، تابع f که مربع یک عدد حقیقی را به یک عدد حقیقی معین x نسبت می‌دهد، عبارت است از:

$$f(x) = x^2$$

یک تابع، مجموعه‌ای از زوجهای مرتب است. برای مثال، تابع $f(x) = x^2$ همهٔ زوجهای مرتب (x, x^2) را تعیین می‌کند. **نمودار** یک تابع، مجموعه‌ای است از کلیهٔ زوجهای مرتب که به وسیلهٔ تابع تعیین می‌شود. نمودار تابع $f(x) = x^2$ در شکل A-۱ نشان داده شده است. تابع $f(x) = \frac{1}{x}$، تنها زمانی تعریف می‌شود که $x \neq 0$ باشد. **دامنهٔ** یک تابع، مجموعه مقادیری است که تابع به ازاء آنها تعریف می‌شود.



شکل A-۱ نمودار تابع $f(x) = x^2$. زوج مرتب (۲،۴) مشخص شده است.

به عنوان مثال، دامنهٔ تابع $f(x)=\frac{۱}{x}$، کلیهٔ اعداد حقیقی غیر از صفر است؛ برخلاف تابع $f(x)=x^۲$ که دامنهٔ آن کلیهٔ اعداد حقیقی است. توجه کنید که تابع $f(x)=x^۲$ می‌تواند تنها مقادیر غیرمنفی را به خود بگیرد. منظور از "مقادیر غیرمنفی"، مقادیری بزرگتر یا مساوی صفر می‌باشد؛ برخلاف "مقادیر مثبت" که تنها مقادیر بزرگتر از صفر را شامل می‌شود. **برد** یک تابع، مجموعه مقادیری است که تابع می‌تواند به خود بگیرید. برد $f(x)=x^۲$، اعداد حقیقی غیرمنفی و برد $f(x)=\frac{۱}{x}$، همهٔ اعداد حقیقی بجز صفر و برد $f(x)=(\frac{۱}{x})^۲$، همهٔ اعداد حقیقی مثبت است. می‌گوئیم یک تابع، مجموعه‌ای از اجزاء دامنه به برد می‌باشد. برای مثال، تابع $f(x)=x^۲$ از اعداد حقیقی به اعداد حقیقی غیرمنفی است.

A-۳ استقراء ریاضی

برخی حاصل جمعها معادل عبارات شبه جمله‌ای می‌باشند. به عنوان مثال

$$۱+۲+...+n=\frac{n(n+۱)}{۲}$$

این مساوی را به ازاء برخی از مقادیر n نشان می‌دهیم:

$$۱+۲+۳+۴=۱۰=\frac{۴(۴+۱)}{۲}$$

$$۱+۲+۳+۴+۵=۱۵=\frac{۵(۵+۱)}{۲}$$

به هر حال، از آنجائیکه بی‌نهایت عدد صحیح مثبت وجود دارد، هرگز نمی‌توانیم مطمئن شویم که این تساوی به ازاء تمامی اعداد صحیح مثبت n صادق است. بررسی حالتهای منحصربفرد، فقط می‌تواند به ما اطلاع دهد که "به نظر می‌رسد تساوی درست باشد". یک ابزار بسیار قوی برای بدست آوردن یک نتیجه از تمامی اعداد صحیح مثبت n، استقراء ریاضی است.

استقراء ریاضی، همانند اصول بازی دومینو عمل می‌کند. شکل A-۲ نشان می‌دهد که اگر فاصلهٔ بین هر دو مهرهٔ دومینو کمتر از ارتفاع آنها باشد، می‌توانیم با زدن ضربه به اولین مهره، به تمامی مهره‌ها ضربه بزنیم و آنها را بیندازیم. این کار انجام می‌شود زیرا

۱- ما به اولین مهره ضربه می‌زنیم.

شکل A-۲ با زدن اولین ضربه به مهرهٔ دومینو، همهٔ مهره‌ها خواهند افتاد.

۲- با توجه به فضای بین مهره‌ها که این فاصله کمتر از ارتفاع آنهاست، قطعاً می‌توانیم بگوئیم که با افتادن مهرهٔ nام، مهرهٔ n+۱ام نیز خواهد افتاد.

اگر ما به اولین مهرهٔ دومینو ضربه بزنیم، این مهره به دومین مهره ضربه می‌زند، دومین مهره به سومین مهره ضربه می‌زند و الی آخر. در تئوری می‌توانیم عدد بزرگی را به دلخواه برای تعداد مهره‌ها در نظر گرفته و آنها را به روش فوق بیاندازیم. استقراء نیز دقیقاً به همین روش عمل می‌کند. ابتدا نشان می‌دهیم که چیزی که در حال اثبات آن هستیم، به ازاء n=۱ درست می‌باشد؛ سپس نشان می‌دهیم که اگر عبارت به ازاء یک عدد صحیح مثبت دلخواه n درست باشد، به ازاء n+۱ نیز درست می‌باشد. این مطلب را یک مرتبه نشان دادیم. می‌دانیم که چون عبارت برای n=۱ درست است، پس برای n=۲ نیز درست می‌باشد و چون عبارت برای n=۲ درست است پس برای n=۳ نیز درست می‌گردد و الی آخر (تا بی‌نهایت). بنابراین، می‌توانیم نتیجه بگیریم که برای کلیهٔ اعداد صحیح مثبت n نیز درست است. در استقراء ریاضی از مفاهیم زیر نیز استفاده می‌کنیم:

پایه استقراء: عبارت، به ازاء n=۱ (یا هر مقدار اولیه دیگر) درست است.

فرض استقراء: عبارت برای هر عدد دلخواه $n \geq ۱$ (یا هر مقدار اولیه دیگر) درست است.

گام استقراء: اگر عبارت به ازاء n درست است، آنگاه به ازاء n+۱ نیز درست می‌باشد.

پایه استقراء، در واقع همان ضربه اولیه به مهرهٔ دومینو است؛ برخلاف گام استقراء که نشان می‌دهد اگر مهرهٔ nام دومینو بیافتد، آنگاه مهرهٔ n+۱ام نیز خواهد افتاد.

مثال A-۱ برای تمامی اعداد صحیح مثبت n نشان می‌دهیم که

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

پایه استقراء: برای n=۱ داریم:

$$۱ = \frac{۱(۱+۱)}{۲}$$

فرض استقراء: فرض کنید که برای یک عدد صحیح مثبت دلخواه n داریم:

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

گام استقراء: لازم است که نشان دهیم

$$۱ + ۲ + \dots + (n+۱) = \frac{(n+۱)[(n+۱)+۱]}{۲}$$

بدینصورت که

$$۱ + ۲ + \dots + (n+۱) = ۱ + ۲ + \dots + n + n + ۱$$

$$= \frac{n(n+۱)}{۲} + n + ۱$$

$$\frac{(n+۱)[(n+۱)+۱]}{۲}$$

در گام استقراء عناصری که با فرض استقراء معادل هستند را مشخص نموده‌ایم تا نشان دهیم که فرض استقراء، کجا به کار گرفته شده است. با فرض استقراء،

$$۱+۲+\ldots+n=\frac{n(n+۱)}{۲}$$

و با انجام چند محاسبه جبری ساده به این نتیجه رسیدیم که

$$۱+۲+\ldots+(n+۱)=\frac{(n+۱)[(n+۱)+۱]}{۲}$$

بنابراین، اگر فرض برای n درست باشد، آنگاه برای n+۱ نیز درست است. از آنجائیکه در پایه استقراء نشان دادیم که عبارت به ازاء n=۱ درست است، لذا می‌توانیم نتیجه بگیریم که با استفاده از اصل دومینو، عبارت به ازاء تمامی اعداد صحیح و مثبت n نیز درست می‌باشد.

لازم به ذکر است که لزومی ندارد مقدار اولیه حتماً n=۱ باشد؛ چراکه ممکن است عبارت، فقط به ازاء $n \geq ۱۰$ درست باشد که در این صورت، پایه استقراء بایستی n=۱۰ باشد. زمانی پایه استقراء، n=۰ است که ما در حال اثبات صحت یک عبارت، به ازاء تمامی اعداد صحیح غیرمنفی باشیم.

مثال A-۲ نشان می‌دهیم که برای هر عدد صحیح مثبت n داریم:

$$۱^۲+۲^۲+\ldots+n^۲=\frac{n(n+۱)(۲n+۱)}{۶}$$

پایه استقراء: برای n=۱ داریم:

$$۱^۲=۱=\frac{۱(۱+۱)[(۲\times۱)+۱]}{۶}$$

فرض استقراء: فرض می‌کنیم که برای هر عدد صحیح مثبت دلخواه n داریم

$$۱^۲+۲^۲+\ldots.+n^۲=\frac{n(n+۱)(۲n+۱)}{۶}$$

گام استقراء: بایستی نشان دهیم

$$۱^۲+۲^۲+\ldots+(n+۱)^۲=\frac{(n+۱)[(n+۱)+۱][۲(n+۱)+۱]}{۶}$$

بدینصورت که

$$\begin{aligned}۱^۲+۲^۲+\ldots+(n+۱)^۲&=۱^۲+۲^۲+\ldots+n^۲+(n+۱)^۲\\&=\frac{n(n+۱)(۲n+۱)}{۶}+(n+۱)^۲\\&=\frac{n(n+۱)(۲n+۱)+۶(n+۱)^۲}{۶}\\&=\frac{(n+۱)[(n+۱)+۱][۲(n+۱)+۱]}{۶}\end{aligned}$$

مثال A-۳ نشان می‌دهیم که برای هر عدد صحیح غیرمنفی n داریم:

$$۲^۰ + ۲^۱ + ۲^۲ + \ldots ۲^n = ۲^{n+۱} - ۱$$

که با علامت Σ به صورت زیر در می‌آید:

$$\sum_{i=۰}^{n} ۲^i = ۲^{n+۱} - ۱$$

پایه استقراء: برای n=۰ داریم:

$$۲^۰ = ۱ = ۲^{۰+۱} - ۱$$

فرض استقراء: فرض کنید که برای هر عدد صحیح غیرمنفی n داریم:

$$۲^۰ + ۲^۱ + ۲^۲ + \ldots + ۲^n = ۲^{n+۱} - ۱$$

گام استقراء: بایستی نشان دهیم

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

بدینصورت که

$$۲^۰ + ۲^۱ + ۲^۲ + \ldots + ۲^{n+۱} + ۲^{n+۱} = ۲^۰ + ۲^۱ + ۲^۲ + \ldots + ۲^n + ۲^{n+۱}$$

$$= ۲^{n+۱} - ۱ + ۲^{n+۱}$$

$$= ۲(۲^{n+۱}) - ۱$$

مثال ۳-۸، یک حالت خاص از نتیجه مثال بعد است.

مثال A-۴ برای تمام اعداد صحیح غیرمنفی n و اعداد حقیقی $r \neq ۱$ نشان می‌دهیم که

$$\sum_{i=۰}^{n} r^i = \frac{r^{n+۱} - ۱}{r - ۱}$$

عناصر این مجموع به **تصاعد هندسی** موسومند.

پایه استقراء: برای n=۰ داریم:

$$r^۰ = ۱ = \frac{r^{۰+۱} - ۱}{r - ۱}$$

فرض استقراء: فرض کنید که برای هر عدد صحیح غیرمنفی دلخواه n داریم:

$$\sum_{i=۰}^{n} r^i = \frac{r^{n+۱} - ۱}{r - ۱}$$

گام استقراء: بایستی نشان دهیم که

$$\sum_{i=۰}^{n+۱} r^i = r^{n+۱} + \sum_{i=۰}^{n} r^i$$

$$= r^{n+۱} + \frac{r^{n+۱} - ۱}{r - ۱}$$

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

گاهی اوقات، نتایجی که با استفاده از استقراء بدست می‌آید به روشهای دیگر نیز می‌توانند بدست آیند. برای نمونه، در مثال قبل نشان دادیم که

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1}$$

به جای استفاده از استقراء می‌توانیم عبارت سمت چپ را در مقسوم علیه عبارت سمت راست جذب کرده و آن را ساده کنیم. بدینصورت که

$$\begin{aligned}(r-1)\sum_{i=0}^{n} r^i &= \sum_{i=0}^{n} r^i - \sum_{i=0}^{n} r^i \\ &= (r+r^2+\dots+r^{n+1}) - (1+r+r^2+\dots+r^n) \\ &= r^{n+1}-1\end{aligned}$$

با تقسیم طرفین تساوی به $r-1$، نتیجهٔ مورد نظر بدست می‌آید.

مثال A-۵ برای هر عدد صحیح مثبت n نشان می‌دهیم که

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

پایه استقراء: برای $n=1$ داریم:

$$1\times 2^1 = 2 = (1-1)\,2^{1+1} + 2$$

فرض استقراء: فرض کنید که برای هر عدد صحیح مثبت دلخواه n داریم:

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

گام استقراء: بایستی نشان می‌دهیم

$$\sum_{i=1}^{n+1} i\,2^i = [(n+1)-1]\,2^{(n+1)+1} + 2$$

بدینصورت که

$$\begin{aligned}\sum_{i=1}^{n+1} i\,2^i &= \sum_{i=1}^{n} i\,2^i + (n+1)\,2^{n+1} \\ &= (n-1)\,2^{n+1} + 2 + (n+1)\,2^{n+1} \\ &= [(n+1)-1]\,2^{(n+1)+1} + 2\end{aligned}$$

روش دیگری که در فرض استقراء مرسوم می‌باشد، این است که ابتدا فرض کنیم عبارت به ازاء هر k بزرگتر یا مساوی مقدار ابتدایی و کوچکتر از n درست است. آنگاه در گام استقراء ثابت کنیم که عبارت به ازاء n نیز درست می‌باشد. این کار را در اثبات قضیه ۱-۱ در بخش ۲-۲-۱ انجام دادیم.

اگرچه مثالهای ما همگی با تعیین شبه‌جمله‌ای حاصل جمعها سروکار داشته‌اند، ولی کاربردهای دیگری نیز برای استقراء وجود دارد که بعداً به برخی از آنها اشاره می‌کنیم.

A-۴ قضایا و پیش‌قضایا

قضیه در فرهنگ لغت به معنای گفتاری است که صحت آن باید اثبات شود. در ریاضیات نیز دارای همین معنا است. هر یک از مثالهای بخش قبل می‌توانست به صورت یک قضیه مطرح شود و اثبات آن از طریق استقراء، اثبات آن قضیه باشد. برای مثال می‌توانستیم مثال A-۱ را به صورت زیر مطرح کنیم:

قضیه A-۱ برای هر عدد صحیح $n > ۰$ داریم:

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

اثبات: در این قسمت، اثبات قضیه خواهد آمد که از اثبات استقراء انجام شده در مثال A-۱ می‌توانیم استفاده کنیم.

معمولاً هدف از ارائه و اثبات یک قضیه، بدست آوردن یک نتیجهٔ کلی است که می‌تواند در حالتهای متعددی بکار گرفته شود. برای مثال، می‌توانیم با استفاده از قضیه A-۱، مجموع اولین n عدد صحیح مثبت را به ازاء هر مقدار n به سرعت محاسبه کنیم.

گاهی اوقات دانشجویان برای فهم اختلاف بین حرف شرطی "اگر" و "اگر و فقط اگر" دچار مشکل می‌شوند. دو قضیه زیر این اختلاف را نشان می‌دهد.

قضیه A-۲ برای هر عدد حقیقی x اگر $x > ۰$ باشد، آنگاه $x^۲ > ۰$ خواهد بود.

اثبات: قضیه از این حقیقت استفاده می‌کند که حاصلضرب دو عدد صحیح مثبت، عددی مثبت است.

عکس قضیه A-۲ درست نیست زیرا اگر $x^۲ > ۰$ باشد، آنگاه نمی‌توان نتیجه گرفت که حتماً $x > ۰$ است. برای مثال، $(-۳)^۲ = ۹ > ۰$ و روشن است که ۳- بزرگتر از صفر نیست. در واقع مربع هر عدد منفی، بزرگتر از صفر است. بنابراین، قضیه A-۲ یک مثال از عبارت "اگر" می‌باشد. هر گاه عکس یک قضیه نیز درست باشد، آنگاه قضیه به صورت عبارت "اگر و فقط اگر" خواهد بود و لازم است که هم قضیه و هم عکس آن اثبات شود. قضیه زیر، مثالی از عبارت "اگر و فقط اگر" می‌باشد.

قضیه A-۳ برای هر عدد حقیقی x، $x > ۰$ است اگر و فقط اگر $\frac{۱}{x} > ۰$ باشد.

اثبات: برای اثبات، فرض کنید که $x > ۰$ باشد، آنگاه $\frac{۱}{x} > ۰$ خواهد بود زیرا خارج قسمت دو عدد مثبت، بزرگتر از صفر است و برای اثبات عکس قضیه، فرض کنید که $\frac{۱}{x} > ۰$ باشد، آنگاه $x = \frac{۱}{۱/x} > ۰$ خواهد بود زیرا خارج قسمت دو عدد مثبت، بزرگتر از صفر است.

در فرهنگ لغت، **پیش‌قضیه** به عنوان یک قضیهٔ کمکی که برای اثبات قضایای دیگر بکار می‌رود، تعریف شده است. در واقع، معمولاً به هنگام اثبات یک قضیه از یک یا چند پیش‌قضیه استفاده می‌کنیم و اغلب پیش‌قضایای مربوط به اثبات یک قضیه را قبل از آن بیان نموده و آن را، در صورت لزوم اثبات می‌کنیم.

A-۵ لگاریتم‌ها

لگاریتم‌ها، یکی از ابزارهای ریاضی هستند که بیشتر در تجزیه و تحلیل الگوریتم‌ها بکار می‌روند، در ادامه، مروری مختصر بر ویژگی لگاریتم‌ها خواهیم داشت.

A-۵-۱ تعریف و ویژگیهای لگاریتم‌ها

لگاریتم عمومی یک عدد، عددی است که ۱۰ بایستی به توان آن برسد تا عدد موردنظر بدست آید. اگر x یک عدد معین باشد لگاریتم عمومی آن را به صورت $\log x$ نشان می‌دهیم.

مثال A-۶ به لگاریتم‌های عمومی زیر توجه کنید:

$\log ۱۰ = ۱$ زیرا $۱۰^{۱} = ۱۰$

$\log ۱۰,۰۰۰ = ۴$ زیرا $۱۰^{۴} = ۱۰,۰۰۰$

$\log ۰/۰۰۱ = -۳$ زیرا $۱۰^{-۳} = (\frac{۱}{۱۰})^{۳} = ۰.۰۰۱$

$\log ۱ = ۰$ زیرا $۱۰^{۳} = ۱$

(به خاطر دارید که مقدار هر عدد غیرصفر به توان صفر برابر یک است.)

در حالت کلی، لگاریتم عدد x عددی است که عدد a -موسوم به پایه- باید به توان آن برسد تا x حاصل شود. عدد a می‌تواند هر عدد مثبت غیر از یک باشد؛ برخلاف xکه شامل کلیهٔ اعداد مثبت می‌شود. این بدین معناست که لگاریتم عدد منفی یا صفر وجود ندارد. در نمادگذاری لگاریتم می‌نویسیم $\log_a x$.

مثال A-۷ چند مثال از $\log_a x$ به صورت زیر است:

$\log_۲ ۸ = ۳$ زیرا $۲^{۳} = ۸$

$\log_۳ ۸۱ = ۴$ زیرا $۳^{۴} = ۸۱$

$\log_۲ (۱/۱۶) = -۴$ زیرا $۲^{-۴} = (۱/۲)^{۴} = ۱/۱۶$

$\log_۲ ۷ \approx ۲/۸۰۷$ زیرا $۲^{۲/۸۰۷} \approx ۷$

توجه دارید که آخرین نتیجه در مثال فوق، برای عددی است که توان صحیحی از پایه لگاریتم نمی‌باشد. لگاریتم‌ها برای تمامی اعداد مثبت وجود دارند؛ نه فقط برای توان صحیحی از پایه. البته بحث کاملی از لگاریتم‌هایی که مقدار آنها، توان صحیحی از پایه نمی‌باشد، در این مقوله نمی‌گنجد. در اینجا فقط به لگاریتم‌هایی اشاره می‌کنیم که به صورت یک تابع افزایشی باشند. بدینصورت که

$$\text{اگر } x>y \text{ آنگاه } \log_a^x < \log_a^y$$

بنابراین

$$۲ = \log_۲^۴ < \log_۲^۷ < \log_۲^۸ = ۳$$

در مثال A-۷ دیدیم که $\log_۲^۷$ در حدود ۲/۸۰۷، یعنی بین ۲ و ۳ می‌باشد. برخی از ویژگیهای مهم لگاریتم که در تحلیل الگوریتم‌ها بکار می‌روند را در زیر آورده‌ایم.

برخی ویژگیهای لگاریتم‌ها (در تمامی موارد $a>۱, b>۰, x>۰, y>۰$ می‌باشد)

۱- $\log_a ۱ = O$

۲- $a^{\log_a^x} = x$

۳- $\log_a (xy) = \log_a x + \log_a y$

۴- $\log_a \frac{x}{y} = \log_a x + \log_a y$

۵- $\log_a x^y = y \log_a x$

۶- $x^{\log_a y} = y^{\log_a x}$

۷- $\log_a x = \log_b x / \log_b a$

مثال A-۸ چند مثال از ویژگیهای فوق به صورت زیر است:

$۲^{\log_۲^۸} = ۸$ {ویژگی ۲}

$\log_۲ (۴ \times ۸) = \log_۲ ۸ = ۲ + ۳ = ۵$ {ویژگی ۳}

$\log_۳ \frac{۲۷}{۹} = \log_۳ ۲۷ - \log_۳ ۹ = ۳ - ۲ = ۱$ {ویژگی ۴}

$\log_۲ ۴^۳ = ۳ \log_۲ ۴ = ۳ \times ۲ = ۶$ {ویژگی ۵}

$۸^{\log_۲^۴} = ۴ \log_۲^۸ = ۴^۳ = ۶۴$ {ویژگی ۶}

$\log_۴ ۱۶ = \log_۲ ۱۶ / \log_۲ ۴ = ۴/۲ = ۲$ {ویژگی ۷}

$\log_۲ ۱۲۸ = \log ۱۲۸ / \log ۲ \approx ۲/۱۰۷۲۱/۰/۳۰۱۰۳ = ۷$ {ویژگی ۷}

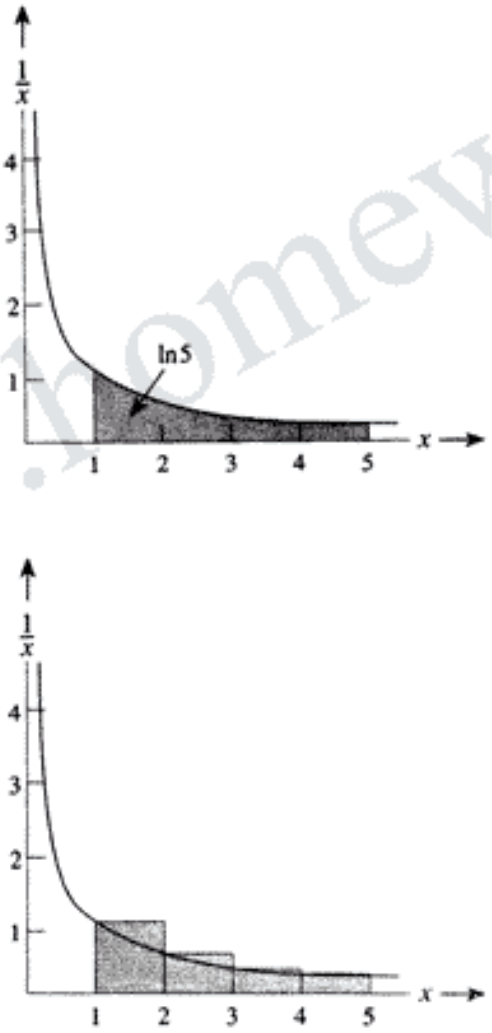
$\log_۳ ۶۷ = \log ۶۷ / \log ۳ \approx ۱/۸۲۶۰۷/۰/۴۷۷۱۲ \approx ۳/۸۲۷۲۸$ {ویژگی ۷}

از آنجائیکه معمولاً ماشینهای حساب یک تابع log دارند (به خاطر دارید که log همان $log_{۱۰}$ است)، دو نمونه آخر در مثال فوق نشان می‌دهد که چگونه می‌توان از ماشین حساب برای هر پایهٔ دلخواهی استفاده نمود. اغلب در تحلیل الگوریتم‌ها از پایهٔ ۲ لگاریتم استفاده می‌کنیم که برای سهولت کار به جای $log_۲ x$، می‌نویسیم $log x$.

A-۵-۲ لگاریتم طبیعی

به خاطر دارید که مقدار تقریبی عدد e برابراست با ۲/۷۱۸۲۸۱۸۲ . عدد e همانند π نمی‌تواند با یک تعداد متناهی از ارقام دهدهی بیان شود و در واقع حتی یک الگوی تکرار ارقام نیز در قسمت اعشاری آن وجود ندارد، $log_e x$ را به صورت $ln\ x$ نشان داده و آن را **لگاریتم طبیعی** x می‌خوانیم. برای مثال،

$$ln\ ۱۰ \approx ۲/۳۰۲۵۸۵۱$$



شکل ۳-۸ قسمت هاشورشده در نمودار بالا، ln ۵ است و قسمت هاشورشده در نمودار پائین که از مستطیلهای متصل به هم تشکیل شده، تقریباً برابر ln ۵ است.

ممکن است تعجب کنید که این جواب از کجا بدست آمده است. ما با استفاده از ماشین حسابی که تابع ln دارد می‌توانیم این محاسبه را انجام دهیم. بدون استفاده از ماشین حساب، فهمیدن چگونگی محاسبهٔ لگاریتم طبیعی و اینکه چرا به آن "طبیعی" گفته می‌شود غیرممکن است. زیرا هنگامی که به عدد e نگاه می‌کنیم، لگاریتم طبیعی بسیار غیرطبیعی به نظر می‌رسد. در اینجا سعی می‌کنیم ویژگیهایی از لگاریتم طبیعی که در تحلیل الگوریتم‌ها بکار می‌رود را مطرح کنیم.

با محاسبات جبری می‌توانیم نشان دهیم که $ln\ x$ برابر سطح زیرنمودار تابع $\frac{۱}{x}$ در محدودهٔ ۱ و x می‌باشد. شکل A-۳، سطح زیرنمودار را برای ln ۵ نشان می‌دهد. در نمودار پائینی شکل A-۳ نشان دادیم که این سطح می‌تواند با مجموع مستطیل‌هایی که پهنای هر کدام به اندازه یک واحد است، به طور تقریبی محاسبه گردد. این نمودار نشان می‌دهد که ln ۵ تقریباً برابراست با

$$(۱\times ۱)+(۱\times\frac{۱}{۲})+(۱\times\frac{۱}{۳})+(۱\times\frac{۱}{۴})\approx ۲/۰۸۳۳$$

توجه کنید که این سطح همیشه بزرگتر از سطح واقعی است. با استفاده از ماشین حساب می‌توانیم مقدار دقیق آن را حساب کنیم که برابر است با

$$ln\approx ۱/۶۰۹۴۴$$

مساحت مستطیل‌ها، تقریب خوبی برای ln ۵ نمی‌باشد. اگرچه مساحت آخرین مستطیل (مستطیل بین مقادیر ۴ و ۵) با سطح زیرمنحنی از x=۴ تا x=۵، تفاوت چندانی ندارد ولی برای اولین مستطیل این چنین نیست. هر مستطیل بعدی دارای تقریب بهتری نسبت به مستطیل قبلی است. بنابراین، هنگامی که عدد کوچک نباشد، مجموع مساحت مستطیل‌ها، نزدیک به مقدار لگاریتم طبیعی آن است. مثال زیر کاربرد این نتیجه را نشان می‌دهد.

مثال A-۹ فرض کنید بخواهیم عبارت زیر را محاسبه کنیم.

$$۱+\frac{۱}{۲}+\ldots+\frac{۱}{n}$$

هیچ شبه جمله‌ای برای این مجموع وجود ندارد. بهر حال، براساس بحث قبلی، اگر n کوچک نباشد، آنگاه

$$(۱\times ۱)+(۱\times\frac{۱}{۲})+\ldots+(۱\times\frac{۱}{n-۱})\approx ln\,n$$

هنگامی که مقدار n کوچک نباشد، مقدار n در مقایسه با مجموع، مقدار ناچیزی خواهد بود. بنابراین، می‌توانیم آن را نیز به مجموع فوق اضافه نمائیم:

$$۱+\frac{۱}{۲}+\ldots+\frac{۱}{n}\approx ln\ n$$

ما از این نتیجه در تحلیل الگوریتم‌ها استفاده خواهیم نمود. در حالت کلی می‌توانیم نشان می‌دهیم که

$$\frac{۱}{۲}+\ldots+\frac{۱}{n}<ln\ n<۱+\frac{۱}{۲}+\ldots+\frac{۱}{n-۱}$$

A-۶ مجموعه‌ها

مجموعه، به اشیائی که در یک جا گرد آمده باشند یا به عبارتی کلکسیونی از اشیاء اطلاق می‌شود. مجموعه‌ها را با حروف بزرگ نظیر S نشان داده و تمامی اشیاء مجموعه را در یک جفت اکولاد ({}) قرار می‌دهیم. برای مثال {۱، ۲، ۳، ۴}=S، مجموعه‌ای شامل اولین چهار عدد صحیح مثبت می‌باشد. ترتیب اشیائی که در مجموعه قرار می‌گیرند اهمیتی ندارد. به این معنا که

{۱، ۲، ۳، ۴} و {۳، ۱، ۴، ۲}

مجموعهٔ یکسانی هستند. یک مثال دیگر از یک مجموعه به صورت زیر است:

S = {پنج شنبه ،سه شنبه ،جمعه ،دوشنبه ،چهارشنبه}

که این مجموعه ، مجموعه‌ای از روزهای هفته است. هنگامی که یک مجموعه نامتناهی باشد می‌توانیم با استفاده از تشریح اشیاء مجموعه، آن را نمایش دهیم. برای مثال، اگر بخواهیم مجموعهٔ اعداد صحیح مثبت که مضرب صحیحی از ۳ هستند را نشان دهیم، می‌توانیم به صورت زیر بنویسیم:

$$S = \{n \mid n = 3i, \quad i \text{ هر عدد صحیح مثبت است}\}$$

که می‌توانیم S را بصورت زیر نشان دهیم:

$$S = \{3, 6, 9, \ldots, 3i, \ldots\}$$

به اشیاء درون یک مجموعه، **عناصر** یا **اعضاء** آن مجموعه گوئیم. اگر x، یک عضو مجموعهٔ S باشد، می‌نویسیم $x \in S$ و اگر x عضوی از مجموعه s نباشد، می‌نویسیم $x \notin S$ به عنوان مثال،

اگر {۱، ۲، ۳، ۴}=S، آنگاه $2 \in S$ ، $5 \in S$

اگر S و T دو مجموعه باشند بطوری که هر عضو مجموعهٔ S در مجموعهٔ T نیز موجود باشد، گوئیم S یک **زیرمجموعه** از مجموعهٔ A است و می‌نویسیم $S \subseteq T$. برای مثال،

اگر {۱، ۳، ۴}=S و {۲، ۱، ۴، ۳}=T، آنگاه $S \subseteq T$

هر مجموعه، یک زیرمجموعه از خودش است. به عبارت دیگر، برای هر مجموعهٔ S داریم $S \subseteq S$ اگر S زیرمجموعه‌ای از T باشد ولی مساوی آن نباشد، گوئیم S یک **زیرمجموعهٔ مطلق** T است و می‌نویسیم $S \subset T$ برای مثال،

اگر {۱، ۳، ۴}=S و {۲، ۱، ۴، ۳}=T، آنگاه $S \subset T$

مجموعه‌های S و T را مساوی گوئیم اگر عناصر مشابهی داشته باشند و می‌نویسیم S=T. اگر دو مجموعه با هم مساوی نباشند می‌نویسیم $S \neq T$ برای مثال،

اگر {۱، ۲، ۳، ۴}=S و {۲، ۴، ۱، ۳}=T، آنگاه $S = T$

اشتراک دو مجموعهٔ S و T، مجموعهٔ همه عناصری است که هم در S و هم در T باشند و می‌نویسیم $S \cap T$. برای مثال،

اگر {۱، ۴، ۵، ۶}=S و {۱، ۳، ۵}=T، آنگاه $S \cap T = \{1, 5\}$

اجتماع دو مجموعهٔ S و T مجموعهٔ همه عناصری است که در S و یا در T باشند و می‌نویسیم $S \cup T$. برای مثال،

اگر $S=\{۱، ۲، ۵، ۶\}$ و $T=\{۱، ۳، ۵\}$، آنگاه $S \cap T=\{۱، ۳، ۲، ۵، ۶\}$

تفاضل دو مجموعهٔ S و T، مجموعهٔ همه عناصری است که در S بوده ولی در T نباشد و می‌نویسم S – T. برای مثال،

اگر $S=\{۱، ۲، ۵، ۶\}$ و $T=\{۱، ۳، ۵\}$، آنگاه $S - T=\{۲، ۶\}$

$T - S=\{۳\}$

مجموعهٔ جهانی (مرجع) U، مجموعه‌ای است که شامل تمامی عناصر مورد نظر باشد. بدین معنا که اگر S هر مجموعهٔ دلخواهی باشد. آنگاه $S \subseteq U$ است. برای مثال، اگر مجموعه‌هایی از اعداد صحیح مثبت موردنظر باشند، آنگاه

$U=\{۱، ۲، ۳، ...، i، ...\}$

A-۷ ترتیب و ترکیب

فرض کنید چهار توپ که با حروف A، B، C و D مشخص شده‌اند را درون ظرفی قرار داده‌ایم. قرار است دو توپ از درون ظرف برداشته شود. اگر توپها به ترتیب حروف برداشته شوند، در مسابقه پیروز می‌شویم. برای روشن شدن موضوع، کلیه حالات ممکنهٔ خروج توپها را می‌نویسیم.

AB	AC	AD
BA	BC	BD
CA	CB	CD
DA	DB	DC

خروجی AB و BA با هم متفاوتند زیرا توپها بایستی به ترتیب برداشته شوند. ۱۲ حالت مختلف را ذکر کردیم. آیا مطمئن هستیم که این ۱۲ حالت، کلیهٔ حالات ممکنه را شامل می‌شود؟ توجه کنید که خروجی‌ها در چهار سطر و سه ستون مرتب شده‌اند بطوری که هر سطر نمایانگر یک انتخاب برای اولین توپ و هر ستون نمایانگر یک انتخاب برای دومین توپ می‌باشد. برای اولین انتخاب، چهار توپ و برای دومین انتخاب، سه توپ در اختیار داریم. بنابراین، مجموع تعداد انتخابهای ممکن برابر است با

$(۴)(۳) = ۱۲$

این نتیجه می‌تواند به صورت کلی‌تری نیز مطرح شود. به عنوان مثال، اگر ما چهار توپ داشته باشیم و بخواهیم سه تا از آنها را برداریم، آنگاه چهار انتخاب برای اولین توپ، سه انتخاب برای دومین توپ و دو انتخاب برای سومین توپ خواهیم داشت که در این صورت تعداد حالات ممکنه برداشت توپ برابراست با $(۴)(۳)(۲) = ۲۴$، و در حالت کلی، اگر n توپ داشته باشیم و بخواهیم k توپ را از بین آنها

برداریم، تعداد حالات ممکنه برابراست با

$$(n)(n-۱)...(n-k+۱)$$

که به آن تعداد **ترتیبهای** k تایی از n شی گوئیم. اگر n=۴ و k=۳، آنگاه

$$(۴)(۳)...(۴-۳+۱) = (۴)(۳)(۲) = ۲۴$$

که این نتیجه را قبلاً بدست آوردیم. اگر n=۱۰ و k=۵، آنگاه

$$(۱۰)(۹)....(۱۰-۵+۱) = (۱۰)(۹)(۸)(۷)(۶) = ۳۰،۲۴۰$$

اگر k=n باشد، یعنی اینکه می‌خواهیم کلیه توپها را برداریم. که در این صورت به آن تعداد ترتیبهای n شی می‌گوئیم که با استفاده از فرمول قبلی برابراست با

$$(n)(n-۱)...(n-n+۱)=n!$$

به خاطر دارید که برای یک عدد صحیح مثبت n، !n به حاصلضرب این عدد و تمامی اعداد کوچکتر از خودش اطلاق می‌گردد. مقدار !۰ برابر یک است و !n برای اعداد منفی، تعریف نشده است. حال مسئله انتخاب توپها را در نظر بگیرید و فرض کنید زمانی در مسابقه پیروز می‌شوید که دو توپ درست را از بین چهار توپ انتخاب نمائید. در اینصورت، خروج AB و BA هیچ تفاوتی با هم نخواهند داشت و ما این خروجی‌ها را خروج A و B می‌گوئیم. از آنجائیکه دو خروجی مسابقهٔ قبلی یک خروجی این مسابقه محسوب می شوند، لذا تعداد حالات ممکنه بر ۲ تقسیم می‌شود؛ یعنی

$$\frac{(۴)(۳)}{۲}=۶$$

که این شش حالت می‌توانند به صورت زیر باشند:

A و B A و C A و D B و C B و D C و D

حال فرض کنید که بخواهیم سه توپ را بدون الزام به رعایت ترتیب، از درون ظرف برداریم. دراینصورت، خروجیهایی که در مسابقهٔ قبلی حضور داشتند ولی در این مسابقه یک خروجی محسوب می‌شوند، عبارتند از:

CBA CAB BCA BAC ACB ABC

این خروجی‌ها، ترتیبهای سه شیء هستند. به خاطر دارید که تعداد چنین ترتیبهایی برابراست با ۶=!۳. حال برای تعیین خروجیهای مجزا در این مسابقه نیاز داریم که تعداد خروجیهای قبلی را بر !۳ تقسیم کنیم؛ بدینصورت که

$$\frac{(۴)(۳)(۲)}{۳!}=۴$$

که عبارتند از:

A و B و C A و B و D A و C و D B و C و D

در حالت کلی، اگر n توپ داشته باشیم و بخواهیم k توپ را از بین آنها انتخاب کنیم بطوری که ترتیب در آن اهمیتی نداشته باشد، تعداد حالت ممکنه برابراست با

$$\frac{(n)(n-1)\dots(n-k+1)}{k!}$$

که به آن تعداد **ترکیبهای** k‌تایی از n شی می‌گوئیم و از آنجائی که

$$(n)(n-1)\dots(n-k+1)=(n)(n-1)\dots(n-k+1)\times\frac{(n-k)!}{(n-k)!}=\frac{n!}{(n-k)!}$$

لذا فرمول تعداد ترکیبهای k‌تایی از n شی به صورت زیر در می‌آید:

$$\frac{n!}{k!(n-k)!}$$

با استفاده از این فرمول، تعداد ترکیبهای سه شی برابراست با

$$\frac{8!}{3!(8-3)!}=56$$

تئوری دوجمله‌ای، که در متون جبری ثابت شده است، بیان می‌کند که برای هر عدد صحیح غیرمنفی n و اعداد حقیقی a و b داریم

$$(a+b)^n=\sum_{k=0}^{n}\frac{n!}{k!(n-k)!}a^k b^{n-k}$$

از آنجائیکه تعداد ترکیبهای k‌تایی n شیء، $a^k b^{n-k}$ در این عبارت محسوب می‌شود، لذا به آن **ضریب دوجمله‌ای** نیز می‌گوئیم و آن را به شکل $\begin{bmatrix} n \\ k \end{bmatrix}$ نشان می‌دهیم.

مثال A-۱۰ نشان می‌دهیم که تعداد زیرمجموعه‌های، یک مجموعهٔ n عنصری با احتساب مجموعهٔ تهی برابر 2^n است. برای $0\le k\le n$، تعداد زیرمجموعه‌های k عنصری برابر ترکیبهای k‌تایی از n یعنی $\begin{bmatrix} n \\ k \end{bmatrix}$ می‌باشد و این بدین معناست که مجموع تعداد زیرمجموعه‌ها برابراست با

$$\sum_{k=0}^{n}\begin{bmatrix} n \\ k \end{bmatrix}=\sum_{k=0}^{n}\begin{bmatrix} n \\ k \end{bmatrix}1^k 1^{n-k}=(1+1)^n=2^n$$

تساوی دوم تا آخر از تئوری دو جمله‌ای نتیجه شده است.

A-۸ احتمال

شاید تا بحال در موقعیتهای مختلفی نظیر درآوردن توپ از ظرف، کشیدن کارت از دسته کارتها و پرتاب سکه، تئوری احتمالات را آزموده باشید. به کشیدن کارت، درآوردن توپ و پرتاب سکه، **آزمون** می‌گوئیم. در حالت کلی، تئوری احتمالات زمانی بکار می‌آید که یک آزمون دارای چندین خروجی مجزا از هم باشد. مجموعهٔ همه خروجیهای ممکن به **فضای نمونه** یا **جمعیت** موسوم است. ریاضیدانان، معمولاً از لفظ "فضای نمونه" استفاده می‌کنند؛ برخلاف جامعه‌شناسان، که برای مطالعهٔ انسانها لفظ "جمعیت" را بکار

می‌گیرند. هر زیرمجموعه از یک فضای نمونه را یک **پیشامد** می‌گوئیم و به زیرمجموعه‌هایی که شامل تنها یک عنصر باشد، **پیشامد ابتدایی** طلاق می‌کنیم.

مثال A-۱۱ در آزمون انتخاب کارت از دسته کارتها، فضای نمونه شامل ۵۲ کارت مختلف است. مجموعهٔ

$$S=\{\text{شاه پیک ،شاه گشنیز ،شاه خشت ،شاه دل}\}$$

یک پیشامد است و مجموعهٔ

$$E=\{\text{شاه دل}\}$$

یک پیشامد ابتدایی است. ۵۲ پیشامد ابتدایی در این فضای نمونه وجود دارد.

منظور از یک پیشامد (زیرمجموعه) این است که یکی از عناصر زیرمجموعه، خروجی آزمون می‌باشد. در مثال A-۱۱، منظور از پیشامد S این است که کارت کشیده شده یکی از چهار نوع شاه باشد و منظور از پیشامد ابتدایی E این است که کارت کشیده شده «شاه دل» می‌باشد.

هر پیشامد، با یک عدد حقیقی موسوم به احتمال پیشامد مشخص می‌شود. در زیر یک تعریف کلی از احتمال را با یک فضای نمونهٔ محدود آورده‌ایم.

تعریف فرض کنید یک فضای نمونه شامل n خروجی مجزا به صورت زیر داریم:

$$\text{فضای نمونه} = \{e_1, e_2, ..., e_n\}$$

تابعی که یک عدد حقیقی P(s) را به پیشامد S نسبت دهد، **تابع احتمال** نامیده می‌شود اگر دارای شرایط زیر باشد:

۱- $0 \le P(e_i) \le 1$ برای $1 \le i \le n$

۲- $P(e_1)+P(e_2)+...+P(e_n)=1$

۳- برای هر پیشامد S که یک پیشامد ابتدایی نیست، P(S) برابر مجموع احتمالات پیشامدهای ابتدایی است که خروجیهای آنها در S وجود دارد. برای مثال، اگر

$$S=\{e_1, e_2, e_7\}$$

$$P(S)=P(e_1)+P(e_2)+P(e_7)$$

فضای نمونه همراه با تابع P را **فضای احتمال** گوئیم.

از آنجائیکه احتمال را به عنوان تابعی از یک مجموعه تعریف نمودیم، لذا به جای $P=(e_i)$، وقتی که e_i یک پیشامد ابتدایی است باید بنویسیم $P(\{e_i\})$. ما برای حفظ هماهنگی، این کار را انجام نمی‌دهیم و به همین ترتیب، برای احتمال یک پیشامد غیرابتدایی نیز از آکولاد استفاده نمی‌کنیم. به عنوان مثال، برای احتمال پیشامد $\{e_1, e_2, e_7\}$ می‌نویسیم $P\{e_1, e_2, e_7\}$.

ساده‌ترین روش برای انتساب احتمالات، استفاده از **اصل یکنواختی** است. این اصل بیان می‌کند که اگر هیچ دلیلی برای ترجیح دادن یک خروجی به دیگری نباشد، خروجی‌ها بایستی با یک احتمال در نظر گرفته شوند. براساس این قانون، هنگامی که n خروجی مجزا وجود داشته باشد، احتمال هر یک از آنها نسبت $\frac{۱}{n}$ خواهد بود.

مثال A-۱۲ فرض کنید چهار توپ علامت‌دار A، B، C و D در یک ظرف وجود دارد که می‌خواهیم یکی از آنها را برداریم. فضای نمونه برابراست با {A, B, C, D} و براساس اصل یکنواختی

$$P(A)=P(B)=P(C)=P(D)=\frac{۱}{۴}$$

پیشامد {A, B}، به این معناست که توپ A یا توپ B برداشته می‌شود که در این صورت احتمال آن برابراست با

$$P(A,B)=P(A)+P(B)=\frac{۱}{۴}+\frac{۱}{۴}=\frac{۱}{۲}$$

مثال A-۱۳ فرض کنید آزمون کشیدن کارت از دسته کارتها در حال انجام است. از آنجائیکه ۵۲ کارت وجود دارد، براساس اصل یکنواختی، احتمال هر کارت برابر $\frac{۱}{۵۲}$ است. برای مثال،

$$P = (\text{شاه دل}) = \frac{۱}{۵۲}$$

پیشامد {شاه گشنیز ،شاه خشت ،شاه پیک ،شاه دل}=S، به این معناست که کارت کشیده شده، شاه است که در این صورت احتمال آن برابراست با

$$P(s) = P(\text{شاه دل}) + P(\text{شاه پیک}) + P(\text{شاه خشت}) + P(\text{شاه گشنیز})$$

$$= \frac{۱}{۵۲}+\frac{۱}{۵۲}+\frac{۱}{۵۲}+\frac{۱}{۵۲}=\frac{۱}{۱۳}$$

گاهی اوقات می‌توانیم با استفاده از فرمولهای ترتیب و ترکیب، احتمالات را محاسبه کنیم. مثال زیر چگونگی انجام این کار را نشان می‌دهد.

مثال A-۱۴ فرض کنید پنج توپ علامت‌دار A، B، C، D و E درون ظرفی قرار دارند و آزمون ما، درآوردن سه توپ از درون ظرف است بدون آنکه ترتیب آنها اهمیتی داشته باشد. می‌خواهیم P(A, B, C) را محاسبه کنیم. می‌دانیم که "C و B و A"، درآوردن توپهای A و B و C با هر ترتیب است. برای تعیین احتمال با استفاده از اصل یکنواختی، لازم است که تعداد خروجیهای مجزا محاسبه شود. در این صورت بایستی تعداد ترکیبهای سه تایی از پنج عنصر را محاسبه نمائیم. با استفاده از فرمول بخش قبل داریم:

$$\frac{۵!}{۳!(۵-۳)!}=۱۰$$

بنابراین براساس اصل یکنواختی،

$$P(A,B,C)=\frac{۱}{۱۰}$$

که دقیقاً مشابه احتمالات نُه خروجی دیگر می‌باشد.

اغلب، دانشجویانی که علاقهٔ چندانی به مطالعهٔ احتمالات ندارند فکر می‌کنند که بحث احتمالات به سادگی مقولهٔ نسبت و تناسب است. حتی در این بخشِ مروری بر احتمالات نیز چنین نیست. در واقع، مهم‌ترین کاربردهای احتمالات هیچ سنخیتی با مقولهٔ تناسب ندارند. برای روشن شدن موضوع به دو مثال زیر توجه نمائید.

یک مثال کلاسیک از احتمالات، مسئله پرتاب سکه است. از آنجائیکه ماهیت سکه به گونه‌ای است که اصل یکنواختی در آن رعایت می‌شود، بنابراین،

$$P(\text{رو}) = P(\text{پشت}) = \frac{1}{2}$$

همچنین می‌توانیم مسئله پرتاب پونز را مطرح کنیم. پونز، همانند سکه، می‌تواند به دو حالت بر زمین قرار بگیرد. یکی روی سرپونز و دیگری لبهٔ آن. این دو حالت در شکل A-۴ مشخص شده است. با استفاده از سکه، به دو حالت "پشت" و "رو" دست می‌یابیم؛ در حالیکه براساس ماهیت فیزیکی پونز هیچ دلیلی برای استفاده از اصل یکنواختی وجود ندارد. پس چطور می‌توانیم احتمالاتی را به آن نسبت دهیم؟ در حالت پرتاب سکه، وقتی می‌گوئیم $P(\text{رو}) = \frac{1}{2}$، یعنی فرض می‌کنیم که اگر سکه را ۱۰۰۰ مرتبه پرتاب کنیم، حدود ۵۰۰ مرتبه بر "رو"ی سکه قرار می‌گیرد. در واقع، اگر تنها ۱۰۰ مرتبه بر "رو" قرار بگیرد. نتیجه می‌گیریم که سکه از نظر وزنی استاندارد نبوده و احتمال آن $\frac{1}{2}$ نیست. این شیوهٔ انجام مکرّر یک آزمایش مشابه، یک روش تقریباً واقعی محاسبه یک احتمال را به ما نشان می‌دهد؛ بدینصورت که اگر ما آزمونی را چندین مرتبه تکرار کنیم، می‌توانیم مطمئن شویم که احتمال یک خروجی در حدود تعداد دفعاتی است که به طور واقعی رخ داده است. به عنوان مثال، دانشجویی یک پونز را ۱۰۰۰۰ مرتبه پرتاب کرده است که از این تعداد ۳۷۶۱ مرتبه بر "رو" قرار گرفته است. در اینصورت برای پونز داریم

$$P(\text{رو}) \approx \frac{۳۷۶۱}{۱۰۰۰۰} = ۰/۳۷۶۱ \qquad P(\text{پشت}) \approx \frac{۶۲۳۹}{۱۰۰۰۰} = ۰/۶۲۳۹$$

مشاهده می‌کنیم که لزومی ندارد احتمالات دو پیشامد یکسان شود. البته مجموع این احتمالات، همچنان برابر یک می‌باشد. این روش تعیین احتمالات به **روش تکرار نسبی** موسوم است. هنگامی که با این روش احتمالاتی را محاسبه می‌کنیم، بایستی از نماد $\approx$ استفاده نمائیم زیرا نمی‌توانیم مطمئن باشیم که تکرار نسبی، دقیقاً معادل با احتمال موردنظر باشد. برای مثال، فرض کنید دو توپ علامت‌دار A و B در ظرفی قرار دارد و ما آزمون برداشتن توپ را ۱۰۰۰۰ مرتبه تکرار می‌کنیم. نمی‌توانیم مطمئن باشیم که توپ A را دقیقاً ۵۰۰۰ مرتبه برداشته‌ایم. ممکن است این امر، تنها ۴۹۶۷ مرتبه اتفاق افتاده باشد. لذا با استفاده از اصل یکنواختی می‌نویسیم $P(A) = ۰.۵$ و با استفاده از روش تکرار نسبی داریم $P(A) \approx ۰.۴۹۶۷$. روش تکرار نسبی، تنها به آزمایش دو حالت ممکنه محدود نمی‌شود. برای مثال، اگر یک مکعب ناقص شش وجهی داشته باشیم، احتمالات شش پیشامد ابتدایی می‌توانند

شکل A-۴ نحوهٔ قرار گرفتن پونز در آزمون.

با هم متفاوت باشد. به هر حال، هنوز هم مجموع احتمالات برابر یک است. مثال زیر، این وضعیت را نشان می‌دهد.

مثال A-۱۵ فرض کنید یک معکب ناقص شش وجهی داریم و در ۱۰۰۰ پرتاب تعیین نموده‌ایم که هر یک از شش وجه چند مرتبه ظاهر شده‌اند که به صورت زیر می‌باشد:

وجه	تعداد دفعات
۱	۲۰۰
۲	۱۵۰
۳	۱۰۰
۴	۲۵۰
۵	۱۲۰
۶	۱۸۰

دراینصورت

$$P(۱) \approx ۰/۲$$
$$P(۲) \approx ۰/۱۵$$
$$P(۳) \approx ۰/۱$$
$$P(۴) \approx ۰/۲۵$$
$$P(۵) \approx ۰/۱۲$$
$$P(۶) \approx ۰/۱۸$$

با توجه به شرط ۳ در تعریف فضای احتمال داریم:

$$P(۲, ۳) = P(۲) + P(۳) \approx ۰/۱۵ + ۰/۱ = ۰/۲۵$$

این مقدار احتمال ظهور دو وجه ۲ یا ۳ را در پرتاب مکعب مشخص می‌کند.

شما می‌توانید برای آشنایی بیشتر با فلسفهٔ احتمالات به کتاب Fine (۱۹۷۳)، برای مطالعهٔ دقیق روش تکرار نسبی احتمال به کتاب Neapolitan (۱۹۹۲)، به منظور شناخت اصل یکنواختی به کتاب keyne (۱۹۴۸) و برای کشف تضادهای حاصله از بکارگیری اصل یکنواختی به کتاب Neapolitan (۱۹۹۰) مراجعه نمائید.

A-۸-۱ ارقام تصادفی

اگرچه به راحتی از عبارت "تصادفی" در مکالمات روزمره استفاده می‌کنیم ولی تعریف آن کمی مشکل به نظر می‌رسد. فرآیند تولید تصادفی، دارای دو ویژگی عمدهٔ زیر است: اول اینکه ، این فرآیند بایستی

قابلیت تولید یک رشتهٔ طولانی و دلخواه را از خروجی موردنظر داشته باشد. به عنوان مثال، فرآیند تکرار پرتاپ سکه می‌تواند به هر تعداد دلخواهی سکه را پرتاب نماید که در اینصورت خروجی آن می‌تواند "پشت" یا "رو" باشد. دوم آنکه، نتایج و خروجی‌ها باید غیرقابل پیش‌بینی باشند. معنای "غیرقابل پیش‌بینی"، هرچه که باشد، تا حدودی مبهم است و بنظر می‌رسد که درست در جای شروع ایستاده‌ایم زیرا به راحتی عبارت "تصادفی" را با عبارت "غیرقابل پیش‌بینی" جایگزین کرده‌ایم.

در اوایل قرن بیستم، ریچارد وُن مایسز، تعریفی از تصادفی‌ها ارائه نمود. وی گفت: «فرآیندی غیرقابل پیش‌بینی که اجازه ندهد هیچ استراتژی شرط‌بندی پیروز شود.» به عنوان مثال، فرض کنید که ما تصمیم گرفته‌ایم در پرتاپ مکرر سکه، بر طرف «رو»ی سکه شرط‌بندی کنیم. همهٔ ما می‌دانیم که هرگز نمی‌توانیم با یک پرتاپ، شانس بردمان را نسبت به پرتاپ قبلی افزایش دهیم. اگر به واقع نتوانیم با شرط‌بندی روی یک وجه معین شانس برد خود را افزایش دهیم، آنگاه پرتاپ مکرر سکه، یک فرآیند تصادفی خواهد بود. این تعریف مایسز می‌تواند ما را به تعریف بهتری از تصادفی‌ها رهنمون شود. یک فرآیند قابل پیش‌بینی یا غیرتصادفی، این امکان را می‌دهد که یک استراتژی شرط‌بندی موفقی داشته باشیم. با این وجود، تعریف مایسز قادر نیست که یک تعریف ریاضی ارائه کند. آندره کلموگرف توانست با مفهوم رشته‌های قابل فشرده‌سازی، این کار را انجام دهد. به طور خلاصه، یک رشتهٔ متناهی را **قابل فشرده‌سازی** گوئیم اگر بتوان با تعداد بیتهائی کمتر از هر عنصر موجود در رشته، آن را کدگذاری نمود. برای مثال،

۱۰۱۰۱۰۱۰۱۰۱۰۱۰۱۰۱۰۱۰۱۰۱۰۱۰۱۰

که به طور ساده ۱۶ تکرار از "۱۰" است می‌تواند به صورت زیر نشان داده شود:

۱۶۱۰

از آنجائیکه این رشته، از تعداد بیتهای کمتری نسبت به رشتهٔ اصلی تشکلی شده است، لذا رشتهٔ اصلی را «قابل فشرده‌سازی» می‌گوئیم. یک رشتهٔ متناهی که قابل فشرده‌سازی نباشد به یک **رشتهٔ تصادفی** موسوم است. برای مثال، رشتهٔ

۱۰۰۱۱۰۱۰۰۰۱۰۱۱۰۱

یک رشتهٔ تصادفی است زیرا هیچ الگوی خاصی در این رشته یافت نمی‌شود. به عبارتی، نمی‌توانیم شکل نمایشی کاراتری برای این رشته پیدا نمود.

براساس فرضیه کلموگرف، یک **فرآیند تصادفی**، فرآیندی است که یک رشتهٔ تصادفی را به اندازهٔ دلخواه تولید کند. بعنوان مثال، فرض کنید که سکه‌ای را چند بار پرتاپ می‌کنیم، روی سکه را با ۱ و پشت آن را با ۰ مشخص کرده‌ایم. پس از شش پرتاپ سکه، رشتهٔ زیر بدست می‌آید:

۱۰۱۰۱۰

مثال ۱۶-A فرض کنید ظرفی شامل یک توپ سیاه و یک توپ سفید داریم. مکرراً توپی را برداشته و دوباره به ظرف برمی‌گردانیم. این فرآیند تصادفی، یک فضای احتمال به صورت زیر تعیین می‌کند:

$$P(\text{سیاه}) = P(\text{سفید}) = ۰/۵$$

مثال A-۱۷ پرتاب مکرر یک مکعب ناقص شش‌وجهی در مثال A-۱۵، یک فرآیند تصادفی است که فضای احتمال زیر را مشخص می‌کند:

$$P(۱) \approx ۰/۲$$
$$P(۲) \approx ۰/۱۵$$
$$P(۳) \approx ۰/۱$$
$$P(۴) \approx ۰/۲۵$$
$$P(۵) \approx ۰/۱۲$$
$$P(۶) \approx ۰/۱۸$$

اگرچه نظریه وُن مایسز، امروزه کاربرد چندانی ندارد، اما دیدگاهش در زمان ارائه آن، یعنی اوایل قرن بیستم، بسیار تازه و نو بود. بزرگترین مخالف او، پروفسور کی مارب، عقیده داشت که طبیعت، رخدادها را در یک حافظه ثبت می‌کند. براساس این نظریه، اگر در ۱۵ پرتاپ مکرر یک سکه سالم (که احتمال آمدن طرف "رو"ی سکه برابر ۰/۵ است)، سکه فقط به طرف پشت بر زمین بنشیند، آنگاه احتمال اینکه در پرتاپ بعدی، سکه به طرف "رو" قرار بگیرد افزایش می‌یابد، زیرا طبیعت تمامی دفعاتی که سکه به پشت بر زمین نشسته است را جبران خواهد کرد. اگر این تئوری درست باشد، ما می‌توانستیم بعد از هر بار پرتاپ سکه و نشتن آن به طرف "پشت" با شرط‌بندی بر طرف "رو"ی سکه، شانس پیروزی خود را افزایش دهیم. ایورسون ات آل، آزمایشاتی را دربارهٔ نظریات مایسز و کلموگروف انجام داد. آزمایشات او به طور مشخص نشان می‌داد که سکه‌های پرتاپ شده و توپهای بیرون آمده از ظرف، رشته‌هایی تصادفی هستند.

تئوری اصلی ون مایسز، در کتاب وی (۱۹۱۹) مطرح شده است و در کتابی که در سال ۱۹۷۵ به چاپ رسانیده، مفصلاً در مورد آن بحث نموده است. برای آشنایی با بحث رشته‌های قابل فشرده‌سازی و رشته‌های تصادفی به کتاب Lambalgen (۱۹۸۷) و به منظور شناخت فرآیند تصادفی و نحوهٔ تولید یک رشتهٔ تصادفی به کتاب Neapolitan (۱۹۹۲) و Lambalgen (۱۹۸۷) مراجعه نمائید.

تمرینات

بخش A-۱

۱- هر یک از مقادیر زیر را تعیین کنید.

(a) ⌊۲/۸⌋ (b) ⌊-۱۰/۴۲⌋ (c) ⌈۴/۲۶⌉

(d) ⌈-۳۴/۹۲۶⌉ (e) ⌊۴/۷ - ۵/۲⌋ (f) ⌈۲π⌉

۲- نشان دهید که $\lfloor -n \rfloor = -\lceil n \rceil$ است.

۳- نشان دهید که برای هر عدد حقیقی x

$$\lfloor 2x \rfloor = \lfloor x \rfloor + \lfloor x + \frac{1}{2} \rfloor$$

۴- نشان دهید که برای هر عدد صحیح $a>0$ و $b>0$ و n

(a) $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n$ (b) $\lfloor \frac{\lfloor n/a \rfloor}{b} \rfloor = \lfloor \frac{n}{ab} \rfloor$

۵- هر یک از حاصل جمعهای زیر را با استفاده از نماد Σ بنویسید.

(a) $2 + 4 + 6 + \ldots + 2(100)$

(b) $2 + 4 + 6 + \ldots + 2(n-1) + 2(n)$

(c) $3 + 12 + 27 + \ldots + 1200$

۶- هر یک از حاصل جمعهای زیر را ارزیابی کنید.

(a) $\sum_{i=1}^{5} (2i+4)$

(b) $\sum_{i=1}^{10} (i^2 + 4i)$

(c) $\sum_{i=1}^{200} (\frac{i}{i+1} - \frac{i-1}{i})$

(d) $\sum_{i=0}^{5} (3^i n^{5-i})$ هرگاه $n=4$ باشد.

(e) $\sum_{i=1}^{4} \sum_{j=1}^{i} (j+5)$

بخش A-۲

۷- نمودار توابع زیر را رسم نموده، دامنه و برد هر یک از آنهارا مشخص نمائید.

(a) $f(x) = \sqrt{x-4}$

(b) $f(x) = (x-2)/(x+5)$

(c) $f(x) = \lfloor x \rfloor$

(d) $f(x) = \lceil x \rceil$

بخش A-۳

۸- با استفاده از استقراء ریاضی نشان دهید که برای هر عدد صحیح $n>0$

$$\sum_{k=1}^{n} k(k!) = (n+1)! - 1$$

۹- با استفاده از استقراء ریاضی نشان دهید که برای هر عدد صحیح n، مقدار $n^2 - n$ زوج خواهد بود.

۱۰- با استفاده از استقراء ریاضی نشان دهید که برای هر عدد صحیح n>۴،

$$۳^n > n^۲$$

۱۱- با استفاده از استقراء ریاضی نشان دهید که برای هر عدد صحیح n>۰

$$\left(\sum_{i=۱}^{n} i\right)^۲ = \sum_{i=۱}^{n} i^۳$$

بخش A-۴

۱۲- ثابت کنید که اگر a و b هر دو عددی صحیح و فرد باشند، a+b یک عدد صحیح زوج خواهد بود. آیا عکس این مطلب نیز درست است؟

۱۳- ثابت کنید که a+b یک عدد صحیح فرد است اگر و تنها اگر هر دو از اعداد صحیح زوج یا هر دو از اعداد صحیح فرد نباشند.

بخش A-۵

۱۴- هر یک از مقادیر زیر را تعیین کنید.

(a) $\log ۱۰۰۰$ (b) $\log ۱۰۰۰۰۰$ (c) $\log_۴ ۶۴$

(d) $\lg \frac{۱}{۱۶}$ (e) $\log_۵ ۱۲۵$ (f) $\log ۲۳$

(g) $\log(۱۶ \times ۸)$ (h) $\log(۱۰۰۰/۱۰۰۰۰۰۰)$ (i) $۲^{\lg ۱۲۵}$

۱۵- نمودار توابع $g(x) = \lg x$, $f(x) = ۳^x$ را در یک سیستم مختصات نمایش دهید.

۱۶- به ازاء چه مقادیری از x نامساویهای زیر برقرار است؟

(a) $x^۲ + ۶x + ۱۲ > ۸x + ۲۰$

(b) $x > ۵۰۰ \lg x$

۱۷- نشان دهید که تابع $f(x) = ۲^{۳ \lg x}$، یک تابع نمایی نیست.

۱۸- نشان دهید که برای هر عدد صحیح مثبت n،

$$\lfloor \lg n \rfloor + ۱ = \lceil \lg(n+۱) \rceil$$

۱۹- با استفاده از مقدار تقریبی Stirling برای !n که در زیر آمده است، یک فرمول برای lg(n!) پیدا کنید.

$$n! = \sqrt{۲\pi n}\ \left(\frac{n}{e}\right)^n \qquad \text{(برای مقادیر نسبتاً بزرگ n)}$$

بخش A-۶

۲۰- با فرض U={۲، ۴، ۶، ۸، ۱۰، ۱۲} (مجموعهٔ جهانی)، S={۲، ۴، ۵} و T={۲، ۶، ۸، ۱۰}، عناصر هر یک از مجموعه‌های زیر را تعیین کنید:

(a) $S \cup T$ (b) $S \cap T$ (c) $S - T$

(d) $T - S$ (e) $((S \cap T) \cup S)$ (f) $U - S$ (بخوانید مکمل S)

۲۱- نشان دهید که هر مجموعهٔ n عنصری S دارای 2^n است.

۲۲- با فرض این که $|S|$ نمایانگر تعداد عناصر مجموعهٔ S است، صحت عبارت زیر را بررسی کنید.

$$|S \cup T| = |S| + |T| - |S \cap T|$$

۲۳- دو مجموعهٔ s و T، به شرط $S \subset T$ مفروض است. نشان دهید که

$$S \cup T = T \quad \text{(a)} \qquad S \cap T = S \quad \text{(b)}$$

۲۴- تعداد ترتیبهای شش تایی از ۱۰ شی را تعیین کنید.

۲۵- تعداد ترکیبهای شش تایی از ۱۰ شی را تعیین کنید.

۲۶- با استفاده از استقراء ریاضی، تئوری دو جمله‌ای که در بخش A-۷ ارائه شده است را ثابت کنید.

۲۷- نشان دهید که

$$\binom{2n}{2} = 2\binom{n}{2} + n^2$$

۲۸- فرض کنید k_1 شی از اولین نوع، k_2 شی از دومین نوع،... و k_m شی از mامین نوع وجود دارد که در آن $k_1 + k_2 + \ldots + k_m = n$ است. نشان دهید که تعداد ترتیبهای مجزا از n شی برابراست با

$$\frac{n!}{(k_1!)(k_2!)\ldots(k_m!)}$$

۲۹- نشان دهید که تعداد حالتهای توزیع n شی مشخص به m گروه مجزا برابراست با

$$\binom{n+m-1}{n} = \binom{n+m-1}{m-h}$$

۳۰- صحت تساوی زیر را بررسی کنید.

$$\binom{n}{k+1} = \frac{n-k}{k+1}\binom{n}{k}$$

بخش A-۸

۳۱- فرض کنید که با یک مکعب شش وجهی کامل شماره‌دار، مسابقه‌ای ترتیب داده‌ایم بدینصورت که هر شخص به مقدار شماره‌ای که آورده است پول دریافت می‌کند؛ به استثنای هنگامی که پنج یا شش بیاورد که در اینصورت بایستی ۵ دلار یا ۶ دلار بپردازد. با فرض این که بازی ۱۰۰ بار تکرار می‌شود، مقدار پولی که بازیکن از دست خواهد داد و مقدار پولی که بدست خواهد آورد را با تئوری احتمالات محاسبه کنید.

ضمیمهٔ B

حل معادلات بازگشتی

تحلیل الگوریتم‌های بازگشتی، به اندازهٔ الگوریتم‌های تکرار واضح و روشن نمی‌باشد. برای نمایش پیچیدگی یک الگوریتم بازگشتی با استفاده از معادلهٔ بازگشتی، مشکلی وجود ندارد؛ مگر اینکه معادلات بازگشتی بایستی حل شوند تا پیچیدگی زمانی تعیین گردد. در اینجا روشهایی را برای حل معادلات بازگشتی مورد بحث قرار خواهیم داد و نیز از روشهایی صحبت خواهیم کرد که نحوهٔ بکارگیری راه حلها در تحلیل الگوریتم بازگشتی را تشریح می‌کند.

B-۱ حل معادلات بازگشتی به روش استقراء

در ضمیمهٔ A، روش استقراء مورد بررسی قرار گرفت. در اینجا توضیح خواهیم داد که چگونه می‌توان از این روش برای تحلیل برخی الگوریتم‌های بازگشتی استفاده نمود. برای شروع، الگوریتمی را در نظر می‌گیریم که مقدار n! را محاسبه می‌نماید.

الگوریتم B-۱ فاکتوریل

مسئله: مقدار $n! = n(n-۱)(n-۲)\ldots(۳)(۲)(۱)$ را تعیین کنید هرگاه $n \geq ۱$ باشد.

$$۰! = ۱$$

ورودی: یک عدد صحیح غیرمنفی n.

خروجی: n!.

```
int fact (int n)
{
  if (n == 0)
    return 1;
  else
    return n * fact(n - 1);
}
```

برای مشخص شدن کارایی این الگوریتم کافی است تعیین کنیم که این تابع به چند تعداد دستورالعمل ضرب را به ازاء مقادیر مختلف n انجام می‌دهد. اگر T(n) را به عنوان تعداد ضربهای انجام شده برای مقدار معین n در نظر بگیریم، رابطهٔ زیر برقرار می‌شود:

$$t_n = t_{n-1} + 1$$

ضرب در سطح بالا — تعداد ضربها در فراخوانی‌های بازگشتی

به چنین معادلاتی، **معادلات بازگشتی** گوئیم زیرا مقدار تابع به ازاء n به مقدار تابع به ازاء مقادیر کوچکتری از n بستگی دارد. در این توابع، نقطهٔ شروعی به نام وضعیت ابتدایی وجود دارد. در این الگوریتم، زمانی که n=۰ است، هیچ ضربی انجام نمی‌شود. بدین ترتیب، وضعیت ابتدایی برابر است با

$$t_0 = 0$$

می‌توان t_n را برای مقادیر بزرگتری از n نیز محاسبه نمود. به موارد زیر توجه کنید:

$$t_1 = t_{1-1} + 1 = t_0 + 1 = 0 + 1 = 1$$
$$t_2 = t_{2-1} + 1 = t_1 + 1 = 1 + 1 = 2$$
$$t_3 = t_{3-1} + 1 = t_2 + 1 = 2 + 1 = 3$$

با ادامهٔ این روند، مقادیر بیشتری از t_n بدست می‌آید. ولی با این روش نمی‌توان بدون نقطهٔ شروع صفر، مقدار t_n را محاسبه نمود. لازم است که برای t_n، اصطلاح مناسبی را در نظر بگیریم. چنین عباراتی، **جواب معادله بازگشتی** نامیده می‌شوند. لازم به ذکر است که با روش استقراء نمی‌توان جوابها را مشخص کرد بلکه این روش فقط می‌تواند صحت و سقم جواب را مورد بررسی قرار دهد. (روش استقراء ساختاری، که در بخش ۴-۵-۸ توضیح داده شد، می‌تواند در یافتن جواب نیز مورد استفاده قرار گیرد). با ارزیابی تعدادی از مقادیر اولیه می‌توان راه‌حلی را برای معادلهٔ بازگشتی پیشنهاد نمود. یک ارزیابی از مقادیر فوق، ما را به این نتیجه می‌رساند که $t_n = n$، یک جواب است. حال می‌توانیم با استفاده از روش استقراء، صحت یا سقم آنرا بررسی کنیم.

پایه استقراء: برای n=۰ داریم $t_0 = 0$.

فرض استقراء: فرض می‌کنیم که برای هر عدد صحیح مثبت n داریم $t_n = n$.

گام استقراء: بایستی نشان دهیم که $t_{n+1} = n + 1$.

اگر عبارت n+۱ را در معادله قرار دهیم، خواهیم داشت.

$$t_{n+1} = t_{(n+1)-1} + 1 = t_n + 1 = n + 1$$

تحلیل یک الگوریتم بازگشتی شامل دو مرحله است: تعیین بازگشت و حل معادلهٔ بازگشتی. در این ضمیمه سعی می‌کنیم به تشریح چگونگی حل معادلات بازگشتی بپردازیم. تعیین بازگشت برای الگوریتم‌های بازگشتی را بعداً بررسی خواهیم کرد. لذا در اینجا از الگوریتم‌ها بحثی به میان نمی‌آید. حال، به چند مثال در مورد حل معادلات بازگشتی با استفاده از روش استقراء توجه کنید:

مثال B-۱ بازگشت زیر را در نظر بگیرید:

$t_n = t_{n/2} + 1$ n توانی از ۲ است
$t_1 = 1$

چند مقدار اولیه عبارتند از:

$$t_۲ = t_{۲/۲} + ۱ = t_۱ + ۱ = ۱ + ۱ = ۲$$
$$t_۴ = t_{۴/۲} + ۱ = t_۲ + ۱ = ۲ + ۱ = ۳$$
$$t_۸ = t_{۸/۲} + ۱ = t_۴ + ۱ = ۳ + ۱ = ۴$$
$$t_{۱۶} = t_{۱۶/۲} + ۱ = t_۸ + ۱ = ۴ + ۱ = ۵$$

به نظر می‌رسد که

$$t_n = lg\, n + ۱$$

با استفاده از روش استقراء، صحت مسئله فوق ثابت می‌شود

پایه استقراء: برای n=۱ داریم $t_۱ = ۱ = lg\ \ ۱ + ۱$

فرض استقراء: فرض کنید به ازاء هر عدد دلخواه n>۰، که n توانی از ۲ است، داریم $t_n = lg\ n + ۱$

گام استقراء: از آنجائیکه بازگشت تنها برای توانی از ۲ است. مقدار بعدی که برای n در نظر گرفته می‌شود برابر با ۲n است. بنابراین بایستی نشان دهیم که $t_{۲n} = lg(۲n) + ۱$.

اگر عبارت ۲n را در معادله قرار دهیم، خواهیم داشت:

$$t_{۲n} = t\,(۲n/۲) + ۱ = t_n + ۱ = lg\, n + ۱ + ۱$$
$$= lg\, n + lg\ ۲ + ۱$$
$$= lg\ (۲n) + ۱$$

مثال B-۲ بازگشت زیر را در نظر بگیرید.

$$t_n = ۷t_{n/۲} \qquad n>۱ \text{ و } n \text{ توانی از } ۲ \text{ است}$$
$$t_۱ = ۱$$

تعدادی از مقادیر اولیه عبارتند از:

$$t_۲ = ۷t_{۲/۲} = ۷t_۱ = ۷$$
$$t_۴ = ۷t_{۴/۲} = ۷t_۲ = ۷^۲$$
$$t_۸ = ۷t_{۸/۲} = ۷t_۴ = ۷^۳$$
$$t_{۱۶} = ۷t_{۱۶/۲} = ۷t_۸ = ۷^۴$$

به نظر می‌رسد که

$$t_n = ۷^{lg\, n}$$

برای اثبات درستی این عبارت، از روش استقراء استفاده می‌کنیم.

پایه استقراء: برای n=۱ داریم $t_۱ = ۱ = ۷^۰ = ۷^{lg\ ۱}$.

فرض استقراء: فرض کنید که به ازاء هر عدد دلخواه n>۰، که n توانی از ۲ است داریم $t_n = ۷^{lg\ n}$.

گام استقراء: بایستی نشان دهیم که $t_{۲n} = ۷^{lg\ (۲n)}$.

اگر عبارت ۲n را در معادلهٔ بازگشتی قرار دهیم، خواهیم داشت:

$$t_{۲n} = ۷t\,(۲n)/۲ = ۷t_n = ۷ \times ۷^{lg\,n} = ۷^{۱+lg\,n} = ۷^{lg\,۲ + lgn} = ۷^{lg\,(۲n)}$$

که با این نتیجه اثبات استقراء کامل می‌شود و در نهایت از آنجائی که $۷^{lg\,n} = n^{lg\,۷}$، لذا جواب معادلهٔ بازگشتی به صورت زیر است:

$$t_n = n^{lg\,r} \approx n^{۲/۸۱}$$

مثال B-۳ معادلهٔ بازگشتی زیر را در نظر بگیرید:

$$t_n = ۲t_{n/۲} + n - ۱$$
$$t_۱ = ۰$$

تعدادی از مقادیر اولیه عبارتند از:

$$t_۲ = ۲t_{۲/۲} + ۲ - ۱ = ۲t_۱ + ۱ = ۱$$
$$t_۴ = ۲t_{۴/۲} + ۴ - ۱ = ۲t_۲ + ۳ = ۵$$
$$t_۸ = ۲t_{۸/۲} + ۸ - ۱ = ۲t_۴ + ۷ = ۱۷$$
$$t_{۱۶} = ۲t_{۱۶/۲} + ۱۶ - ۱ = ۲t_۸ + ۱۵ = ۴۹$$

در این مورد جواب پیشنهادی روشنی وجود ندارد که به وسیله این مقادیر مورد بررسی قرار گیرد. همانطور که اشاره شد، روش استقراء فقط برای بررسی صحت یا سقم جواب بکار می‌رود و به علت عدم وجود جواب نمی‌توان از روش مطرح شده استفاده نمود.

B-۲ حل معادلات بازگشتی با استفاده از معادلهٔ شاخص

روش ارائه شده در این بحث، برای تعیین جواب بسیاری از معادلات بازگشتی مورد استفاده قرار می‌گیرد.

B-۲-۱ معادلهٔ بازگشتی خطی همگن

تعریف یک معادلهٔ بازگشتی به شکل $a_۰t_n + a_۱t_{n-۱} + \ldots + a_kt_{n-k} = ۰$ که در آن k و عناصر a_i مقادیری ثابت هستند، **معادلهٔ بازگشتی خطی همگن با ضرایب ثابت** نامیده می‌شود.

علت "خطی" بودن معادله این است که هر عنصر t_i فقط با توان مشخص می‌شود. به همین دلیل، عناصری نظیر t_{n-i}^2، $t_{n-i}t_{n-j}$ و غیره در آن وجود ندارند. همچنین عنصری نظیر $t_{c(n-i)}$ که در آن c یک مقدار ثابت غیر از یک است، یافت نمی‌شود. بعنوان مثال، عناصری نظیر $t_{n/2}$، $t_{3(n-4)}$ و غیره در آن وجود ندارند واز آنجائیکه ترکیب خطی عبارت برابر صفر است، به آن "همگن" یا یکسان گویند.

مثال B-۴ در نمونهٔ زیر چند معادلهٔ بازگشتی خطی همگن با ضریب ثابت را نشان می‌دهیم،

$$7t_n - 3t_{n-1} = 0$$
$$6t_n - 5t_{n-1} + 8t_{n-2} = 0$$
$$8t_n - 4t_{n-3} = 0$$

مثال B-۵ دنبالهٔ فیبوناچی، که در بخش ۲-۲-۱ معرفی شده است، به صورت زیر تعریف می‌شود:

$$t_n = t_{n-1} + t_{n-2}$$
$$t_0 = 0$$
$$t_1 = 1$$

اگر از دو طرف معادله عبارت t_{n-1} و t_{n-2} را کم کنیم، عبارت $t_n - t_{n-1} - t_{n-2} = 0$ بدست می‌آید و مبین این است که دنباله فیبوناچی به وسیله یک معادلهٔ بازگشتی خطی همگن تعریف می‌شود.

در ادامه، نشان می‌دهیم که چگونه می‌توان یک معادلهٔ بازگشتی خطی همگن را حل نمود.

مثال B-۶ معادلهٔ بازگشتی زیر را در نظر بگیرید:

$$t_n - 5t_{n-1} - 6t_{n-2} = 0$$
$$t_0 = 0$$
$$t_1 = 1$$

توجه داشته باشید که اگر بجای t_n عبارت 7^n را قرار دهیم، آنگاه خواهیم داشت:

$$t_n - 5t_{n-1} + 6t_{n-2} = 7^n - 5\cdot 7^{n-1} + 6\cdot 7^{n-2}$$

بنابراین، جواب معادله برابر است با $t_n = 7^n$، اگر r ریشه معادله زیر باشد:

$$r^n - 5r^{n-1} + 6r^{n-2} = 0$$

از آنجائیکه $r^n - 5r^{n-1} + 6r^{n-2} = 0$ است، لذا یکی از ریشه‌ها برابر صفر و بقیه، ریشه‌های معادلهٔ $r^2 - 5r - 6 = 0$ هستند. با استفاده از روش تجزیه، دو ریشه $r=3$ و $r=2$ بدست می‌آیند. بنابراین،

$$t_n = 0 \quad , \quad t_n = 3^n \quad , \quad t_n = 2^n$$

جوابهای بالا، جوابهای معادلهٔ بازگشتی هستند. اگر در سمت چپ معادله، عبارت 3^n را جایگزین کنیم، یعنی

$$\begin{array}{ccc} 3^n & 3^{n-1} & 3^{n-2} \\ \downarrow & \downarrow & \downarrow \\ t_n & -5t_{n-1} & +6t_{n-2} \end{array}$$

آنگاه عبارت زیر حاصل می‌شود:

$$3^n - 5(3^{n-1}) + 6(3^{n-2}) = 3^n - 5(3^{n-1}) + 2(3^{n-1}) = 3^n - 3(3^{n-1}) = 3^n - 3^n = 0$$

و این بدین معناست که عبارت 3^n، یکی از جوابهای معادلهٔ بازگشتی است.

تاکنون سه جواب برای معادلهٔ بازگشتی پیدا نمودیم؛ درحالیکه جوابهای دیگری نیز وجود دارد. از آنجائیکه 3^n و 2^n جوابهای معادله هستند، لذا می‌توان نتیجه گرفت که

$$t_n = c_1 3^n + c_2 2^n \qquad (c_1 \text{ و } c_2 \text{ ثوابتی دلخواه هستند})$$

نیز یک جواب معادله است. نتایج حاصل از تمرینات، بیانگر این مطلب هستند که این نتایج، تنها جوابهای معادله هستند یا به عبارتی، جواب عمومی معادله را نشان می‌دهند (با قرار دادن $c_1 = c_2 = 0$ در جواب عمومی، $t_n = 0$ بدست می‌آید) پرواضح است که بی‌نهایت جواب برای معادله وجود دارد. اما کدامیک از این نتایج، جواب مسئله ما است؟ با توجه به شرایط اولیه، این نکته روشن می‌شود. به خاطر دارید که در شرایط اولیه داشتیم $t_0 = 1$ ، $t_1 = 1$ این دو حالت مقادیری از c_1 و c_2 را تعیین می‌کنند. بدینصورت که

$$\begin{aligned} t_0 &= c_1 3^0 + c_2 2^0 = 1 \\ t_1 &= c_1 3^1 + c_2 2^1 = 1 \end{aligned} \longrightarrow \begin{aligned} c_1 + c_2 &= 1 \\ 3c_1 + 2c_2 &= 1 \end{aligned}$$

به راحتی می‌توان نتیجه گرفت که $c_1 = 1$ ، $c_2 = -1$ بنابراین، جواب معادلهٔ بازگشتی ما چنین است:

$$t_n = 1(3^n) - 1(2^n) = 3^n - 2^n$$

اگر در مثال قبل، وضعیت ابتدایی متفاوتی داشتیم، جوابهای متفاوتی نیز بدست می‌آوریم. بیائید برای معادلهٔ بازگشتی بدست آمده در مثال B-۶، وضعیت ابتدایی را بدینصورت در نظر بگیریم:

$$t_0 = 1 \quad , \quad t_1 = 2$$

باید ببینیم که در این وضعیت چه نتایجی بدست می‌آید؟ با بکارگیری جواب عمومی در مثال B-۶ و با در نظر گرفتن هر یک از این شرایط خواهیم داشت:

$$t_0 = 1 \quad , \quad t_1 = 2$$

که جوابهای $c_1 = 0$ و $c_2 = 1$ از آن بدست می‌آید. بنابراین، جواب معادلهٔ بازگشتی چنین است:

$$t_n = 0(3^n) + 1(2^n) = 2^n$$

معادلهٔ B-۱ در مثال B-۶ به عنوان معادلهٔ شاخص برای معادلهٔ بازگشتی معرفی می‌شود. در حالت کلی، معادلهٔ شاخص به صورت زیر تعریف می‌شود:

تعریف **معادلهٔ شاخص** برای معادلهٔ بازگشتی خطی همگن با ضرایب ثابت

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$$

به صورت زیر تعریف می‌شود

$$a_0 r^k + a_1 r^{k-1} + \ldots + a_k r^0 = 0$$

مقدار r^0 برابر یک است. عناصری نظیر r^0 را برای این می‌نویسیم که رابطهٔ بین معادلهٔ شاخص و معادلهٔ بازگشتی به خوبی روشن شود.

مثال B-۷ معادلهٔ شاخص برای معادلهٔ بازگشتی به صورت زیر نشان داده می‌شود:

$$5t_n - 7t_{n-1} + 6t_{n-2} = 0$$

$$\Downarrow$$

$$5r^2 - 7r + 6 = 0$$

با استفاده از یک فلش نشان می‌دهیم که ترتیب معادلهٔ شاخص برابر k (در این حالت برابر ۲) است.

مراحل بدست آوردن جواب در مثال B-۶ را می‌توان به صورت یک قضیه تعمیم داد. برای حل معادلهٔ بازگشتی خطی همگن با ضریب ثابت کافی است به قضیهٔ زیر اشاره کنیم:

قضیه B-۱ معادلهٔ بازگشتی خطی همگن با ضرائب ثابت به صورت زیر داده شده است:

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$$

اگر معادلهٔ شاخص آن، یعنی $a_0 r^k + a_1 r^{k-1} + \ldots + a_k r^0 = 0$، دارای k جواب مجزای $r_1, r_2, \ldots, r_k$ باشد می‌توان نتیجه گرفت که تنها جواب معادله به صورت زیر است:

$$t_n = c_1 r_1^n + \ldots + c_k r_k^n \qquad (c_i \text{ها ثوابتی اختیاری هستند})$$

مقادیر k و ثابت c_iها به وسیله وضعیت ابتدایی تعیین می‌شوند. برای تعیین k ثابت منحصر به فرد به k وضعیت ابتدایی نیاز داریم. روش تعیین مقدار ثوابت را در مثال زیر آورده‌ایم.

مثال B-۸ معادلهٔ زیر را حل کنید:

$$\begin{cases} t_n - ۳t_{n-۱} - ۴t_{n-۲} = ۰ & ,n>۱ \\ t_۰ = ۰ \\ t_۱ = ۱ \end{cases}$$

۱- تعیین معادلهٔ شاخص:

$$t_n - ۳t_{n-۱} - ۴t_{n-۲} = ۰$$
$$\Downarrow$$
$$r^۲ - ۳r - ۴ = ۰$$

۲- حل معادلهٔ شاخص:

$$r^۲ - ۳r - ۴ = (r-۴)(r+۱) = ۰$$

که در آن $r=۴$ و $r=-۱$ ریشه‌های معادله هستند.

۳- بکارگیری قضیه B-۱ برای تعیین جواب عمومی معادلهٔ بازگشتی:

$$t_n = c_۱ ۴^n + c_۲ (-۱)^n$$

۴- تعیین مقادیر ثوابت با استفاده از جواب عمومی و با در نظر گرفتن وضعیت ابتدایی:

$$t_۰ = ۰ = c_۱ ۴^۰ + c_۲ (-۱)^۰ \qquad c_۱ + c_۲ = ۰$$
$$t_۱ = ۱ = c_۱ ۴^۱ + c_۲ (-۱)^۱ \qquad ۴c_۱ - c_۲ = ۱$$

۵- جایگزینی ثوابت در جواب عمومی برای تعیین دقیق جواب نهایی:

$$t_n = \frac{۱}{۵} ۴^n - \frac{۱}{۵}(-۱)^n$$

مثال B-۹ معادلهٔ بازگشتی زیر دنباله فیبوناچی را تولید می‌کند. معادله را حل می‌کنیم

$$\begin{cases} t_n - t_{n-۱} - t_{n-۲} = ۰ & ,n>۱ \\ t_۰ = ۰ \\ t_۱ = ۱ \end{cases}$$

۱- تعیین معادلهٔ شاخص

$$t_n - t_{n-۱} - t_{n-۲} = ۰$$
$$\Downarrow$$
$$r^۲ - r - ۱ = ۰$$

۲- حل معادلهٔ شاخص: برای حل این معادله از روش معادلهٔ دو مجهولی عمل می‌کنیم که ریشه‌های زیر بدست می‌آید:

$$r = \frac{۱+\sqrt{۵}}{۲} \quad , \quad r = \frac{۱-\sqrt{۵}}{۲}$$

۳- بکارگیری قضیه B-۱ برای تعیین جواب عمومی معادلهٔ بازگشتی:

$$t_n = c_۱ \left(\frac{۱+\sqrt{۵}}{۲}\right)^n + c_۲ \left(\frac{۱-\sqrt{۵}}{۲}\right)^n$$

۴- تعیین مقدار ثوابت با استفاده از جواب کلی و با توجه به وضعیت ابتدایی:

$$t_0 = c_1(\frac{1+\sqrt{5}}{2})^0 + c_2(\frac{1-\sqrt{5}}{2})^0 = 0 \quad \Rightarrow \quad c_1 + c_2 = 0$$

$$t_1 = c_1(\frac{1+\sqrt{5}}{2})^1 + c_2(\frac{1-\sqrt{5}}{2})^1 = 1 \quad \Rightarrow \quad (\frac{1+\sqrt{5}}{2})c_1 + (\frac{1-\sqrt{5}}{2})c_2 = 1$$

که با حل این معادله خواهیم داشت:

$$c_1 = 1/\sqrt{5} \quad , \quad c_2 = -1/\sqrt{5}$$

۵- حل معادله با استفاده از جایگزینی ثوابت در جواب کلی:

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

اگرچه در مثال فوق، فرمول روشن و واضحی برای تعیین جمله nام فیبوناچی، بدست آمده است؛ ولی ارزش کاربردی چندانی ندارد. زیرا درجه دقت لازم برای نمایش $\sqrt{5}$ با افزایش n، افزایش می‌یابد.

در قضیه B-۱ بایستی تمامی k ریشهٔ معادلهٔ شاخص از هم مجزا باشند. در این قضیه، معادلهٔ شاخصی به شکل $(r-2)^3(r-1) = 0$ نمی‌توان یافت، زیرا عبارت $r-2$ به توان ۳ رسیده است. به عبارت دیگر، عبارت $r-2$، به تعداد سه بار در هم ضرب شده است. عدد ۲ به یک ریشه توان ۳ موسوم است. قضیه زیر، مکان ریشه‌های توان‌دار را به ما نشان می‌دهد.

قضیه B-۲ اگر r، یک ریشه به توان m معادلهٔ شاخص برای یک معادلهٔ بازگشتی خطی همگن با ضرایب ثابت باشد، آنگاه

$$t_n = r^n, \; t_n = nr^n, \; t_n = n^2r^n, \; t_n = n^3r^n, \; t_n = n^{m-1}r^n$$

جوابهای معادلهٔ بازگشتی هستند. بنابراین هر یک از این عبارات، همانند قضیه B-۱، یک عبارت در جواب عمومی معادله می‌باشند.

مثال B-۱۰ معادلهٔ بازگشتی زیر را حل می‌کنیم:

$$t_n - 7t_{n-1} - 15t_{n-2} - 9t_{n-3} = 0 \quad , n > 2$$
$$t_0 = 0$$
$$t_1 = 1 \qquad t_2 = 2$$

۱- تعیین معادلهٔ شاخص:

$$t_n - ۷t_{n-۱} - ۱۵t_{n-۲} - ۹t_{n-۳} = ۰$$

$$r^۳ - ۷r^۲ + ۱۵r - ۹ = ۰$$

۲- حل معادلهٔ شاخص:

$$r^۳ - ۷r^۲ - ۱۵r - ۹ = (r-۱)(r-۳)^۲ = ۰$$

که $r=۱$ و $r=۳$ ریشه‌های معادله بوده و $r=۳$، ریشه توان ۲ معادله می‌باشد.

۳- بکارگیری قضیه B-۲ برای تعیین جواب عمومی معادله:

$$t_n = c_۱ ۱^n + c_۲ ۳^n + c_۳ n ۳^n$$

در اینجا عباراتی برای ۳^n و $n۳^n$ آورده‌ایم، زیرا ۳، یک ریشه توان ۲ معادله است.

۴- تعیین مقادیر ثوابت با استفاده از جواب عمومی معادله و با توجه به وضعیت ابتدائی:

$$\begin{aligned} t_۰ = ۰ = c_۱ ۱^۰ + c_۲ ۳^۰ + c_۳(۰)(۳^۰) \\ t_۱ = ۱ = c_۱ ۱^۱ + c_۲ ۳^۱ + c_۳(۱)(۳^۱) \\ t_۲ = ۲ = c_۱ ۱^۲ + c_۲ ۳^۲ + c_۳(۲)(۳^۲) \end{aligned} \quad \Rightarrow \quad \begin{aligned} c_۱ + c_۲ = ۰ \\ c_۱ + ۳c_۲ + ۳c_۳ = ۱ \\ c_۱ + ۹c_۲ + ۱۸c_۳ = ۲ \end{aligned}$$

که با حل این معادلات به جواب $c_۱ = -۱$، $c_۲ = ۱$ و $c_۳ = -\frac{۱}{۳}$ دست می‌یابیم.

۵- تعیین جواب عمومی با جایگزینی ثوابت:

$$t_n = (-۱)(۱^n) + (۱)(۳^n) + (\frac{-۱}{۳})(n۳^n)$$

$$= -۱ + ۳^n - n۳^{n-۱}$$

B-۲-۲ معادلات بازگشتی خطی غیرهمگن

تعریف معادلهٔ بازگشتی به شکل $a_۰ t_n + a_۱ t_{n-۱} + \ldots + a_k t_{n-k} = f(n)$ که در آن k و عناصر a_i مقادیری ثابت بوده و f(n) یک تابع غیر از تابع صفر است، **معادلهٔ بازگشتی خطی غیرهمگن با ضرایب ثابت** نامیده می‌شود.

"تابع صفر" یعنی اینکه مقدار تابع برابر صفر باشد که اگر تابع برابر صفر شود، یک معادلهٔ بازگشتی خطی همگن خواهیم داشت. برای حل یک معادلهٔ بازگشتی خطی غیرهمگن، یک روش جامع و کلی شناخته شده‌ای وجود ندارد. ما یک روش برای حل حالت ویژه و عمومی زیر بیان می‌کنیم:

$$a_۰ t_n + a_۱ t_{n-۱} + \ldots + a_k t_{n-k} = b^n P(n) \qquad (B-۲)$$

که در آن b، یک مقدار ثابت و P(n)، یک چندجمله‌ای از n است.

مثال B-۱۱ معادلهٔ بازگشتی

$$t_n - ۳t_{n-۱} = ۴^n$$

یک مثال از معادلهٔ بازگشتی B-۲ است که در آن k=۱، b=۴، p(n)=۱ می‌باشد.

مثال B-۱۲ معادلهٔ بازگشتی

$$t_n - ۳t_{n-۱} = ۴^n (۸n + ۷)$$

یک مثال از معادلهٔ بازگشتی B-۲ است که در آن k=۱، b=۴، p(n)=۸n+۷ می‌باشد.

حالت ویژه‌ای که در معادلهٔ بازگشتی B-۲ معرفی شده است با تغییر آن به صورت یک معادلهٔ بازگشتی خطی همگن، حل می‌گردد. مثال بعد، چگونگی انجام این کار را تشریح می‌کند.

مثال B-۱۳ معادلهٔ بازگشتی زیر را حل می‌کنیم:

$$\begin{cases} t_n - ۳t_{n-۱} = ۴^n & , n>۱ \\ t_۰ = ۰ \\ t_۱ = ۴ \end{cases}$$

از آنجائیکه در سمت راست معادله، عبارت ۴^n وجود دارد، لذا معادلهٔ بازگشتی فوق همگن نیست. برای حل آن می‌توان به صورت زیر عمل نمود:

۱- n-۱ را جایگزین n در معادلهٔ بازگشتی می‌کنیم، لذا خواهیم داشت:

$$t_{n-۱} - ۳t_{n-۲} = ۴^{n-۱}$$

۲- معادله را بر ۴ تقسیم می‌کنیم و تغییرات جزئی لازم را انجام می‌دهیم:

$$\frac{t_n}{۴} - \frac{۳t_{n-۱}}{۴} = ۴^{n-۱}$$

۳- دو معادلهٔ بدست آمده از مراحل ۱ و ۲ را از هم تفریق می‌کنیم، در این صورت

$$\frac{t_n}{۴} - \frac{۷t_{n-۱}}{۴} + ۳t_{n-۲} = ۰$$

که با ضرب در ۴ به صورت ساده‌تر زیر در می‌آید:

$$t_n - ۷t_{n-۱} + ۱۲t_{n-۲} = ۰$$

این یک معادلهٔ بازگشتی خطی همگن است که می‌تواند با استفاده از تئوری B-۱، به راحتی حل شود. معادلهٔ شاخص آن، $r^۲ - ۷r + ۱۲ = (r-۳)(r-۴) = ۰$ خواهد بود که جواب عمومی $t_n = c_۱۳^n + c_۲۴^n$ را نتیجه می‌دهد. با اعمال شرایط اولیه $t_۰ = ۰$ و $t_۱ = ۴$ به جواب زیر می‌رسیم:

$$t_n = ۴^{n+۱} - ۴(۳^n)$$

در مثال B-۱۳، جواب عمومی شامل $c_1 3^n$ و $c_2 4^n$ می‌باشد. اگر معادلهٔ بازگشتی همگن باشد، اولین عبارت از معادلهٔ شاخص بدست می‌آید، در حالیکه دومین عبارت از بخش غیرهمگن معادله، موسوم به b حاصل می‌شود. چندجمله‌ای P(n) در این مثال برابر یک است. وقتی چنین حالتی وجود نداشته باشد، دستکاریهای لازم برای تبدیل معادلهٔ بازگشتی به یک معادلهٔ همگن بسیار پیچیده می‌گردد. (نتیجهٔ حاصل صرفاً برای ارائه توان b در معادلهٔ شاخص بکار می‌رود که از این معادله در حل معادلات بازگشتی همگن استفاده می‌شود.) نتایج فوق را در قضیه بعد اعمال می‌کنیم.

قضیه B-۳ معادلهٔ بازگشتی خطی غیرهمگن

$$a_0 t_n + a_1 t_{n-1} + \dots a_k t_{n-k} = b^n P(n)$$

می‌تواند به یک معادلهٔ بازگشتی خطی همگن تبدیل شود بطوری که معادلهٔ

$$(a_0 r^k + a_1 r^{k-1} + \dots a_k)\,(r-b)^{d+1} = 0$$

معادلهٔ شاخص آن باشد، که در آن d درجه P(n) است. توجه کنید که معادلهٔ شاخص از دو بخش تشکیل شده است:

۱- معادلهٔ شاخص برای معادلهٔ بازگشتی همگن

۲- عبارت حاصله از بخش غیرهمگن معادلهٔ بازگشتی

اگر در سمت راست، بیش از یک عبارت نظیر $b^n P(n)$ وجود داشته باشد، هر یک از آنها را به عنوان یک جمله در معادلهٔ شاخص قرار می‌دهیم.

قبل از بکارگیری این قضیه یادآور می‌شویم که درجه چند جمله‌ای P(n) عبارتست از بالاترین توان n به عنوان مثال،

چندجمله‌ای	درجه
$P(n) = 3n^2 + 4n - 2$	۲
$P(n) = 5n + 7$	۱
$P(n) = 8$	۰

مثال B-۱۴ معادلهٔ بازگشتی زیر را حل می‌کنیم:

$$t_n - t_{n-1} = n - 1 \quad , \quad n > 0$$
$$t_0 = 0$$

۱- تعیین معادلهٔ شاخص برای معادلهٔ بازگشتی همگن متناظر:

$$t_n - t_{n-1} = 0$$
$$r^1 - 1 = 0$$

۲- تعیین یک عبارت، از بخش غیرهمگن معادلهٔ بازگشتی:

$$n - 1 = 1^n (n^1 - 1)$$

(d ↓ روی n^1، d ↑ روی 1^n)

که عبارت $1^{n+1} (1 - r)$ بدست می‌آید.

۳- بکارگیری قضیه B-۳ برای تعیین معادلهٔ شاخص از عبارات بدست آمده در مراحل ۱ و ۲:

$$(r-1)(r-1)^2$$

۴- حل معادلهٔ شاخص:

$$(r-1)^3 = 0$$

که $r = 1$ یک ریشه توان ۳ معادله است.

۵- بکارگیری قضیه B-۲ برای تولید جواب عمومی معادله:

$$t_n = c_1 1^n + c_2 n 1^n + c_3 n^2 1^n$$
$$= c_1 + c_2 n + c_3 n^2$$

ما به دو شرط دیگر نیز نیازمندیم:

$$t_1 = t_0 + 1 - 1 = 0 + 0 = 0$$
$$t_2 = t_1 + 2 - 1 = 0 + 1 = 1$$

در تمرینات از شما خواسته می‌شود که (۶) مقدار ثوابت را تعیین نموده و (۷) با جایگزینی ثوابت در معادله جواب نهایی را بدست آورید:

$$t_n = \frac{n(n-1)}{2}$$

B-۲-۳ تغییر متغیرها (تغییر دامنه‌ها)

اغلب معادلات بازگشتی به صورتی نیستند که با استفاده از قضیه B-۳ قابل حل باشند. برای حل اینگونه معادلات می‌توان با استفاده از تغییر متغیرها، آنها را به صورت بازگشتی جدیدی تبدیل نمود که در این حالت قابل حل می‌گردند. روش حل این نوع معادلات در مثالهای زیر توضیح داده شده است. در این مثالها، از تابع T(n) به عنوان معادلهٔ بازگشتی اصلی استفاده می‌شود؛ زیرا عبارت T_k برای معادلهٔ بازگشتی جدید استفاده شده است. نماد T(n) به معنای تمامی موارد همگن نظیر t_n است که یک عدد منحصر به فرد مرتبط با هر مقدار n می‌باشد.

مثال B-۱۵ می‌خواهیم معادلهٔ زیر را حل کنیم:

$$T(n)=T(\frac{n}{۲})+۱ \qquad n>۱ \text{ و } n \text{ توانی از } ۲$$
$$T(۱)=۱$$

به خاطر آورید که این معادله را با استفاده از استقراء در بخش B-۱ حل نمودیم. حال می‌خواهیم با استفاده از تکنیک تغییر متغیرها، این کار را انجام دهیم. این معادلهٔ بازگشتی به شکلی نیست که بتوان با استفاده از قضیهٔ B-۳، آن را حل نمود و علت آن وجود جملهٔ $\frac{n}{۲}$ است. معادله را به شکل موردنظر در می‌آوریم. بدینصورت که ابتدا می‌نویسیم $n=۳^k$، که ب این معناست که $k=lg\ n$ سپس ۳^k را به ازاء n در معادلهٔ بازگشتی قرار می‌دهیم که عبارت زیر حاصل می‌شود:

$$T(۳^k)=T(\frac{۳^k}{۲})+۱ \qquad \text{(B-۳)}$$
$$=T(۳^{k-۱})+۱$$

در ادامه، عبارت $t_k=T(۳^k)$ را در معادلهٔ بازگشتی B-۳ قرار می‌دهیم تا بازگشت جدید زیر بدست آید:

$$t_k=T_{k-۱}+۱$$

این معادلهٔ بازگشتی جدید به شکلی است که می‌توان با استفاده از قضیه B-۳ آن را حل نمود. با استفاده از قضیه B-۳ جواب عمومی معادله به صورت زیر تعیین می‌شود:

$$t_k=c_۱+c_۲k$$

با انجام دو مرحله زیر می‌توان به جواب عمومی نهایی دست یافت:

۱- جایگزینی $T(۳^k)$ به جای t_k در جواب عمومی معادلهٔ بازگشتی جدید:

$$T(۳^k)=c_۱+c_۲k$$

۲- جایگزینی n به جای ۳^k و $lg\ n$ به جای k در معادلهٔ حاصل از مرحله اول:

$$T(n)=c_۱+c_۲lg\ n$$

همانطوری که در مثالهای قبل عمل می‌کردیم، در اینجا نیز جواب عمومی را پیدا نموده، سپس با استفاده از وضعیت ابتدایی $T(۱)=۱$ و تعیین وضعیت ابتدایی ثانویه مقدار ثوابت را محاسبه می‌کنیم تا اینکه جواب زیر بدست آید:

$$T(n)=۱+lg\ n$$

مثال B-۱۶ معادلهٔ زیر را حل می‌کنیم:

$$T(n)=۷t(\frac{n}{۲})+۱۸(\frac{n}{۲})^۲ \qquad n>۱ \text{ و } n \text{ توانی از } ۲$$
$$T(۱)=۰$$

با جایگزینی ۳^k به جای n در معادلهٔ بازگشتی به معادلهٔ زیر می‌رسیم:

$$T(۳^k) = ۷t\ (۳^{k-۱}) + ۱۸(۳^{k-۱})^۲ \qquad (B-۴)$$

سپس $t_k = T(۳^k)$ را در معادلهٔ B-۴ قرار می‌دهیم که در این صورت با جایگزینی ۳^k به جای n در معادلهٔ بازگشتی به معادلهٔ زیر می‌رسیم:

$$T(۳^k) = ۷T(۳^{k-۱}) + ۱۸(۳^{k-۱})^۲$$

سپس $t_k = T(۳^k)$ را در معادلهٔ B-۴ قرار می‌دهیم که در اینصورت

$$t_k = ۷t_{k-۱}(۳^{k-۱}) + ۱۸(۳^{k-۱})^۲$$

این معادله به صورتی که در قضیه B-۳ معرفی شده است، نمی‌باشد. بنابراین، بایستی تغییرات لازم روی آن صورت گیرد، بدینصورت که

$$\begin{aligned} t_k &= ۷t_{k-۱} + ۱۸(۳^{k-۱})^۲ \\ &= ۷t_{k-۱} + ۱۸(۳^{k-۱}) \\ &= ۷t_{k-۱} + ۴^k(\frac{۱۸}{۴}) \end{aligned}$$

حال با بکارگیری قضیه B-۳ روی این معادله جدید خواهیم داشت:

$$t_k = c_۱ ۳^k + c_۲ ۴^k$$

مراحل زیر را برای رسیدن به جواب عمومی معادله انجام می‌دهیم:

۱- جایگزینی $T(۳^k)$ به جای t_k در جواب عمومی:

$$T(۳^k) = c_۱ ۳^k + c_۲ ۴^k$$

۲- جایگزینی n به جای ۳^k و $\lg n$ به جای k در معادلهٔ بدست آمده از مرحله اول:

$$\begin{aligned} T(n) &= c_۱ ۳^{\lg n} + c_۲ ۴^{\lg n} \\ &= c_۱ n^{\lg ۳} + c_۲ n^۲ \end{aligned}$$

با استفاده از شرط ابتدایی $T(۱) = ۰$ و تعیین یک شرط ابتدایی دیگر و محاسبه مقادیر ثابت به جواب عمومی زیر می‌رسیم:

$$T(n) = ۶n^{\lg ۳} - ۶n^۲ \approx ۶n^{۱/۸۱} - ۶n^۲$$

B- حل معادلهٔ بازگشتی با استفاده از جایگزینی

یکی از روشهای مورد استفاده برای حل معادلات بازگشتی، روش جایگزینی است. اگر نمی‌توانید از روشهای قبلی برای حل معادلات بازگشتی استفاده کنید، این روش را به شما پیشنهاد می‌کنیم. با استفاده از مثال زیر، چگونگی حل معادلات بازگشتی با روش جایگزینی را توضیح می‌دهیم.

مثال B-۱۷ می‌خواهیم معادلهٔ بازگشتی زیر را حل کنیم:

$$t_n = t_{n-1} + n \qquad ,n > 1$$
$$t_1 = 1$$

در واقع، روش جایگزینی، نقطه مقابل روش استقراء است. در این روش برای حل معادله از n شروع کرده و رو به عقب معادله را حل می‌کنیم:

$$t_n = t_{n-1} + n$$
$$t_{n-1} = t_{n-2} + n - 1$$
$$t_{n-2} = t_{n-3} + n - 2$$
$$\vdots$$
$$t_2 = t_1 + 2$$
$$t_1 = 1.$$

سپس هر یک از معادلات را در معادلهٔ قبلی جایگزین می‌کنیم، بدینصورت که

$$\begin{aligned} t_n &= t_{n-1} + n \\ &= t_{n-2} + n - 1 + n \\ &= t_{n-3} + n - 2 + n - 1 + n \\ &\vdots \\ &= t_1 + 2 + \ldots + n - 2 + n - 1 + n \\ &= 1 + 2 + \ldots + n - 2 + n - 1 + n \\ &= \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \end{aligned}$$

آخرین تساوی این معادله در مثال A-۱ از ضمیمه A بیان شده است.

مثال B-۱۸ برای حل معادلهٔ بازگشتی

$$t_n = t_{n-1} + \frac{2}{n} \qquad ,n > 1$$
$$t_1 = 0$$

ابتدا، از n رو به عقب معادله را حل می‌کنیم:

$$t_n = t_{n-1} + \frac{2}{n}$$
$$t_{n-1} = t_{n-2} + \frac{2}{n-1}$$
$$t_{n-2} = t_{n-3} + \frac{3}{n-2}$$

$$\vdots$$
$$t_۲ = t_۱ + \frac{۲}{۲}$$
$$t_۱ = ۰$$

آنگاه با جایگزینی هر معادله در معادلهٔ قبلی خواهیم داشت:

$$\begin{aligned} t_n &= t_{n-۱} + \frac{۲}{n} \\ &= t_{n-۲} + \frac{۲}{n-۱} + \frac{۲}{n} \\ &= t_{n-۳} + \frac{۲}{n-۲} + \frac{۲}{n-۱} + \frac{۲}{n} \\ &\vdots \\ &= t_۱ + \frac{۲}{۲} + \ldots + \frac{۲}{n-۲} + \frac{۲}{n-۱} + \frac{۲}{n} \\ &= ۰ + \frac{۲}{۲} + \ldots + \frac{۲}{n-۲} + \frac{۲}{n-۱} + \frac{۲}{n} \\ &= ۲ \sum_{i=۲}^{n} \frac{۱}{i} \approx ۲ \ln n \end{aligned}$$

که به ازاء n نه چندان کوچک درست است. تساوی تقریبی اخیر در مثال A-۹ از ضمیمه A آمده است.

B-۴ تعمیم برخی نتایج برای n

برای بررسی مطالب زیر، باید مروری بر مطالب قبلی داشته باشید و مطالب فصل ۱ را مطالعه کنید. در برخی حالات، الگوریتم‌های بازگشتی دارای پیچیدگی‌های زمانی هستند که اگر n توانی بر مبنای b بوده و b یک ثابت مثبت باشد، آنگاه می‌توانید پیچیدگی زمانی دقیق این الگوریتم‌ها را تعیین کنید. اغلب مبنای b برابر ۲ است و این مطلب به ویژه در مورد بسیاری از الگوریتم‌های تقسیم و غلبه صدق می‌کند (به فصل ۲ رجوع کنید.) اینطور بنظر می‌رسد که نتایجی که برای توان n برپایه b قرار می‌گیرند به طور تقریبی، برای n در حالت کلی نیز قرار داده می‌شوند، برای مثال، اگر برای الگوریتمی رابطه زیر برقرار باشد:

$$T(n) = ۲n \lg n \quad \text{n توانی از ۲ است}$$

در اینصورت، برای n در حالت کلی نیز جواب زیر بدست می‌آید:

$$T(n) \in \theta(n \lg n)$$

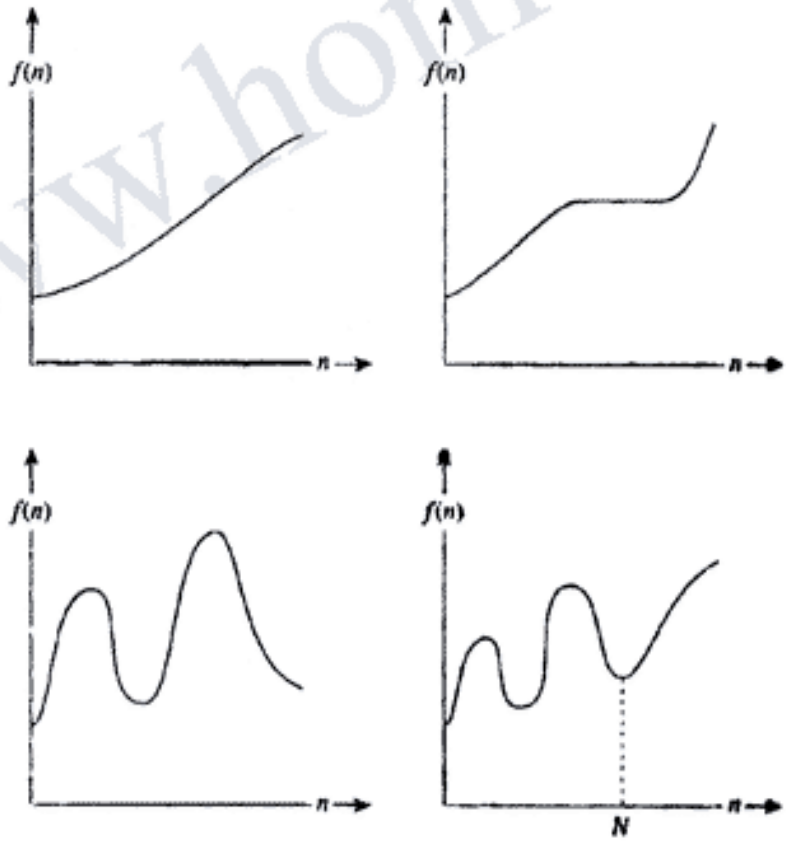
بعداً در مورد چگونگی استفاده از روش جایگزینی برای چنین حالاتی بحث خواهیم کرد. در اینجا به چند تعریف نیاز داریم. این تعاریف برای توابعی بکار می‌روند که دامنهٔ آنها کلیه اعداد حقیقی می‌باشند. اما ما از این تعاریف در مورد پیچیدگی توابعی از اعداد صحیح نامنفی به اعداد حقیقی نامنفی استفاده خواهیم کرد.

تعریف تابع پیچیدگی $f(n)$، یک تابع **صعودی محض** نامیده می‌شود اگر برای مقادیر بزرگتر n، تابع $f(n)$ نیز مقادیر بزرگتری به خود بگیرد. به عبارتی اگر $n_1 > n_2$، آنگاه $f(n_1) > f(n_2)$.

توابع نشان داده شده در شکل B-۱(a) از توابع صعودی محض می‌باشند (دامنهٔ شکل B-۱، اعداد حقیقی است) اگر توابعی که در تحلیل الگوریتم‌ها با آنها روبرو نمی‌شویم به ازاء مقادیر مثبت n، از نوع صعودی محض هستند. به عنوان مثال، توابع $n, lgn, n lg n, n^2, 3^n$، تا زمانی که n عددی غیرمنفی است، صعودی محض می‌باشند.

تعریف تابع پیچیدگی $f(n)$ یک **تابع غیرنزولی** است اگر با بزرگتر شدن مقدار n، هیچگاه مقدار $f(n)$ کاهش نیابد. به عبارت دیگر، اگر $n_1 > n_2$، آنگاه $f(n_1) > f(n_2)$.

هر تابع صعودی محض یک تابع غیرنزولی است، اما هر تابع غیرنزولی نمی‌تواند یک تابع صعودی محض باشد. یک مثال از چنین توابعی را می‌توان در نمودار شکل شکل B-۱(c) مشاهده نمود. نمودار شکل B-۱(c)، یک تابع غیرنزولی نیست. پیچیدگی‌های زمانی یا حافظه‌ای بسیاری از الگوریتم‌ها معمولاً غیرنزولی هستند زیرا با افزایش اندازه ورودی‌ها، زمان لازم برای پردازش آنها کاهش نمی‌یابد. با توجه به



شکل B-۱ چهار تابع.

شکل B-۱ درمی‌یابیم که می‌توانیم تحلیل توان n بر مبنای b را به n در حالت کلی تعمیم دهیم. برای مثال، فرض کنید که ما مقدار تابع $f(n)$ را به ازاء n توانی از ۲ تعیین کرده‌ایم. در چنین توابعی از شکل B-۱(c)، هر وضعیتی ممکنه است بین دو حالت $۸ = ۲^۳$ و $۱۶ = ۲^۴$ روی دهد. بنابراین از وضعیت توابع بین ۸ و ۱۶ هیچ نتیجه‌ای نمی‌توان گرفت) به هر حال، در وضعیت تابع غیرنزولی $f(n)$، اگر $۸ \le n \le ۱۶$، آنگاه $f(۸) \le f(n) \le f(۱۶)$ می‌توان ترتیب تابع $f(n)$ را از روی مقادیر $f(n)$ برای nهایی که توانی از ۲ هستند تعیین نمود. به چند تعریف زیر توجه کنید:

تعریف تابع پیچیدگی $f(n)$ یک تابع **غیرنزولی نهایی** است اگر به ازاء تمامی مقادیر n قبل از یک نقطه مشخص با افزایش مقدار n، هیچگاه تابع، مقدار کوچکتر به خود نگیرد. به عبارتی اگر $n_۱ > n_۲ > N$ آنگاه

$$f(n_۱) \ge f(n_۲)$$

هر تابع غیرنزولی می‌تواند یک تابع غیرنزولی نهایی باشد. تابع نشان داده شده در شکل B-۱(d)، یک تابع غیرنزولی نهایی است امّا یک تابع غیرنزولی نیست. قبل از اینکه بخواهیم قضیه‌ای را مطرح کنیم و از نتایج آن برای توان n در مبنای b استفاده کنیم به تعاریفی نیازمندیم:

تعریف تابع پیچیدگی $f(n)$ یک تابع **هموار** است اگر $f(n)$، یک تابع غیرنزولی نهایی بوده و

$$f(۲n) \in \theta(f(n))$$

مثال B-۱۹ توابع $n^k (k \ge ۰), n \lg n, n, \lg n$ توابعی هموار می‌باشند. این مطلب را برای $\lg n$ نشان می‌دهیم و در تمرینات از شما می‌خواهیم که آن را برای توابع دیگر بررسی کنید. همانطور که می‌دانیم، تابع $\lg n$ یک تابع غیرنزولی نهایی است و برای دو عین شرط داریم:

$$\lg(۲n) = \lg ۲ + \lg n \in \theta(\lg n)$$

مثال B-۲۰ تابع ۳^n، یک تابع هموار نیست زیرا بر اساس ویژگیهای ترتیب در بخش ۲-۴-۱ از فصل ۱ داریم:

$$۳^n \in \theta(۲^n)$$

بنابراین

$$۳^n = ۳^n \notin \theta(۲^n)$$

حال قضیه‌ای را مطرح می‌کنیم که می‌توان از آن برای تعمیم نتایج حاصله برای n، توانی از b استفاده نمود.

قضیه B-۴ فرض می‌کنیم $b \geq ۲$ یک عدد صحیح، $f(n)$ یک تابع پیچیدگی هموار و $T(n)$ یک تابع پیچیدگی غیرنزولی نهایی است. اگر $T(n) \in \theta(f(n))$ (n توانی از b است) باشد، آنگاه داریم

$$T(n) \in \theta(f(n))$$

اگر θ به وسیله "big O"، "small O" یا π جایگزین شود نیز همین نتیجه بدست می‌آید. در نهایت، یک روش کلی برای تعیین ترتیب برخی معادلات بازگشتی متداول ارائه می‌دهیم.

قضیه B-۵ فرض کنید که T(n)، یک تابع پیچیدگی غیرنزولی نهایی به صورت زیر است:

$$T(n) = aT(\frac{n}{b}) + Cn^k \qquad ,n > ۱ \text{ و n توانی از b است}$$
$$T(۱) = d$$

که در آن $b \geq ۲$, $k \geq ۰$ جزء اعداد صحیح، $a > ۰$, $c > ۰$, $d \geq ۰$ می‌باشند. بنابراین

$$T(n) \in \begin{cases} \theta(n^k) & a < b^k \\ \theta(n^k lg\, n) & a = b^k \\ \theta(n^{log_b a}) & a > b^k \end{cases} \qquad \text{(B-۵)}$$

به عبارت دیگر، اگر در تعریف معادلهٔ بازگشتی، عبارت $T(n) = aT(\frac{n}{b}) + cn^k$ با یکی از عبارات زیر جایگزین شود:

$$T(n) \geq aT(\frac{n}{b}) + cn^k \qquad \text{یا} \qquad T(n) \leq aT(\frac{n}{b}) + cn^k$$

آنگاه در نتیجه B-۵، "big O" یا Ω جایگزین θ می‌شوند.

قضیه فوق را می‌توان با حل معادلهٔ بازگشتی عمومی با استفاده از معادلهٔ شاخص و بکارگیری قضیه B-۴، به اثبات رساند. یک مثال کاربردی از قضیه B-۵ در زیر آمده است.

مثال B-۲۱ فرض کنید که T(n) یک تابع غیرنزولی نهایی به صورت زیر می‌باشد:

$$T(n) = ۸T(n/۴) + ۵n^۲ \qquad ,n > ۱ \text{ و n توانی از ۲ است}$$
$$T(۱) = ۳$$

با توجه به قضیه B-۵، چون $۴^۲ < ۸$ است بنابراین،

$$T(n) \in \theta(n^{log_۴ ۹}) = \theta(n^۲)$$

قضیه B-۵، به منظور معرفی یک قضیه مهم که در زیر آمده، بیان شده است. قضیه B-۵، در واقع یک حالت خاص از قضیه B-۶ است که در آن ثابت S برابر یک می‌باشد.

قضیه B-۶ فرض کنید که تابع پیچیدگی T(n)، یک تابع غیرنزولی نهایی به صورت زیر است:

$$T(n) = aT(\frac{n}{b}) + cn^k \qquad , n > s$$
$$T(s) = d$$

n توانی از b است و

که در آن s توانی از b، $b \geq ۲$ ، $k \geq ۰$ جزء اعداد صحیح، $a > ۰$ ، $c > ۰$ ، $d \geq ۰$ می‌باشد. نتایج تئوری B-۵، در اینجا نیز صدق می‌کند.

مثال B-۲۲ فرض کنید تابع T(n) یک تابع غیرنزولی نهایی به صورت زیر است:

$$T(n) = ۸T(n/۲) + ۵n^۳ \qquad , n > ۶۴$$
$$T(۶۴) = ۲۰۰$$

n توانی از ۲ است و

با توجه به فرضیهٔ B-۶ این حقیقت که $۸ = ۲^۳$ می‌توان نتیجه گرفت که

$$T(n) \in \theta(n^۳ lg\ n)$$

روش‌های دیگری نیز برای حل معادلات بازگشتی وجود دارند که یکی از آنها استفاده از "توابع مولد" است. این روش در کتاب Shani (۱۹۸۸) مورد بحث قرار گرفته است. Bently و Haken نیز در سال ۱۹۸۰، یک روش کلی برا حل معادلات بازگشتی معرفی نمودند که در این روش از تحلیل الگوریتم‌های تقسیم و غلبه استفاده شده است.

تمرینات

بخش B-۱

۱- با استفاده از استقراء جواب کاندید هر یک از معادلات بازگشتی زیر را بررسی کنید:

$$t_n = ۴t_{n-۱} \qquad , n > ۱ \qquad \text{(a)}$$
$$t_۱ = ۳$$

جواب کاندید $t_n = ۳(۴^{n-۱})$ است.

(b) $t_n = t_{n-1} + n \quad , n > 1$

$t_1 = 1$

جواب کاندید $t_n = \frac{n(n+1)}{2}$ است.

(c) $t_n = t_{n-1} n^2 \quad , n > 1$

$t_1 = 1$

جواب کاندید $t_n = \frac{n(n+1)(2n+1)}{6}$ است.

(d) $t_n = t_{n-1} + \frac{1}{n(n+1)} \quad , n > 1$

$t_1 = \frac{1}{2}$

جواب کاندید $t_n = \frac{n}{n+1}$ است.

۲- یک معادلهٔ بازگشتی برای جمله nام رشته اعداد ۲، ۶، ۱۸، ۵۴،... بنویسید و با استفاده از استقراء، جواب کاندید را بررسی کنید.

۳- تعداد حرکتهای لازم (m_n برای n حلقه) برای مسئله برجهای هانوی (تمرین ۱۷ در فصل ۲) به صورت معادلهٔ بازگشتی زیر داده شده است.

$$m_n = 2m_{n-1} + 1 \quad , n > 1$$
$$m_n = 1$$

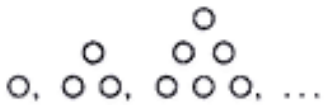
با استفاده از استقراء نشان دهید که جواب معادلهٔ بازگشتی $m_n = 2^n - 1$ می‌باشد.

۴- الگوریتم زیر، موقعیت بزرگترین عنصر آرایهٔ S را بازمی‌گرداند. یک معادلهٔ بازگشتی بنویسید که تعداد مقایسه‌های لازم t_n برای پیدا کردن بزرگترین عنصر را مشخص کند. با استفاده از روش استقراء نشان دهید که جواب معادله برابراست با $t_n = n - 1$

```
index max_position (index low,index high)
{
  index position;
  if (low==high)
     returnlow;
  else{
     position=max_position(low + 1,high);
     if (S[low]>S[position])
       position=low;
     return position;
  }
}
```

فراخوانی سطح بالا در این الگوریتم، max_position(1, n) است.

۵- یونانیان باستان علاقه زیادی به نتایج حاصله از توالی اشکال هندسی داشتند. نمونه‌ای از این توالی در زیر نشان داده شده است:



یک معادلهٔ بازگشتی برای جمله nام این توالی نوشته، یک جواب کاندید برای آن ارائه داده و با استفاده از استقراء، صحت آنرا بررسی کنید.

۶- با استفاده از روش استقرا نشان دهید که $B(n,k) = \binom{n+k}{n}$ جواب معادلهٔ بازگشتی زیر می‌باشد:

$$B(n,k) = B(n-1,k) + B(n,k-1) \qquad ,0 < k < n$$
$$B(n,0) = 1$$
$$B(n,n) = 1$$

۷- الگوریتمی بنویسید که مقدار بازگشت زیر را محاسبه کند. آن را با نمونه مسئله‌های مختلف اجرا کرده، از نتایج حاصل، یک جواب کاندید برای معادله ارائه نموده و با استفاده از استقراء، صحت جواب کاندید خود را بررسی کنید.

$$t_n = t_{n-1} + 2n - 1 \qquad ,n > 1$$
$$t_1 = 1$$

بخش B-۲

۸- مشخص کنید که معادلات بازگشتی تمرینات بخش B-۱ در کدامیک از دسته‌های زیرقرار می‌گیرند:

(a) معادلات خطی

(b) معادلات همگن

(c) معادلات با ضرایب همگن

۹- برای آن دسته از معادلات بازگشتی در بخش B-۱ که از نوع خطی با ضریب ثابت می‌باشد، معادلهٔ شاخص پیدا کنید.

۱۰- نشان دهید که اگر $f(n)$, $g(n)$ جوابهای معادلهٔ بازگشتی خطی همگن با ضرایب ثابت باشند، آنگاه $c \times f(n) + d \times g(w)$ نیز یک جواب این معادله خواهد بود. (c و d مقادیر ثابت هستند.)

۱۱- با استفاده از معادلهٔ شاخص، معادلات بازگشتی زیر را حل کنید:

$$t_n = 4t_{n-1} - 3t_{n-2} \qquad ,n > 1 \qquad \text{(a)}$$
$$t_0 = 0$$
$$t_1 = 1$$

$$t_n = 3t_{n-1} - 2t_{n-2} + n^2 \qquad , n > 1 \qquad \text{(b)}$$
$$t_0 = 0$$
$$t_1 = 1$$

$$t_n = 5t_{n-1} - 6t_{n-2} + n^5 \qquad , n > 1 \qquad \text{(c)}$$
$$t_0 = 0$$
$$t_1 = 1$$

۱۲- جواب معادلهٔ بازگشتی مثال B-۱۴ را کامل کنید.

۱۳- با استفاده از معادلهٔ شاخص، معادلهٔ بازگشتی زیر را حل کنید.

$$t_n = 6t_{n-1} - 9t_{n-2} \qquad , n > 1$$
$$t_0 = 0$$
$$t_1 = 1$$

بخشهای B-۳ و B-۴

۱۴- معادلات بازگشتی تمرین ۱ را با استفاده از روش جایگزینی حل کنید.

۱۵- جواب معادلهٔ بازگشتی مثال B-۱۷ را کامل کنید.

۱۶- نشان دهید که

(a) تابع $f(n) = n^3$، یک تابع صعودی محض است.

(b) تابع $g(n) = 2n^3 - 6n^2$، یک تابع غیر نزولی نهایی است.

(a) تابع $h(n) = n^k$ برای مقادیر $k \geq 0$ یک تابع هموار است.

۱۷- در مورد تابع $f(n)$ که به ازاء تمامی مقادیر n هم غیرنزولی و هم غیرصعودی است چه توصیفی می‌توانید ارائه دهید؟

ضمیمهٔ C

ساختارهای داده‌ای برای مجموعه‌های غیرالحاقی

همانطور که می‌دانیم، الگوریتم Kruskal (الگوریتم ۲-۴ در بخش ۲-۱-۴) به زیرمجموعه‌های غیرالحاقی نیازمند است. این زیرمجموعه‌ها که هر کدام یک گرهٔ مجزا در یک گراف هستند، تا زمانی که در یک مجموعهٔ یکسانی قرار بگیرند با هم ادغام می‌شوند. برای بکارگیری این الگوریتم، ما به یک ساختار داده‌ای برای مجموعه‌های غیرالحاقی نیازمندیم. لازم به ذکر است که این مجموعه‌ها، کاربردهای دیگری نیز دارند. برای مثال، در بخش ۳-۴ می‌توانند برای اصلاح پیچیدگی زمانی الگوریتم ۴-۴ (زمانبندی مهلت‌دار) بکار گرفته شوند. لذا، قبل از اینکه نوع داده‌ای مجردی را برای مجموعه‌های غیرالحاقی تعریف کنیم، لازم است ابتدا اشیاء و عملیات روی آن را مشخص نمائیم. کار را با مجموعه جهانی U آغازمی کنیم و به عنوان مثال، U را برابر مجموعهٔ زیر در نظر می‌گیریم:

$$U = \{A,\ B\ ,\ C,\ D,\ E\}$$

وجود یک روال (موسوم به Makeset)، برای تولید یک مجموعه از هر عنصر U الزامی است. مجموعه‌های غیرالحاقی در شکل C-۱(a) می‌توانند با فراخوانی دستورالعمل زیر تولید شوند:

```
for (each X ε U)
    Makeset (X);
```

همچنین به یک نوع اشاره‌گر مجموعه و یک تابع Find نیازمندیم که اگر P و q از نوع اشاره‌گرهای مجموعه باشند و داشته باشیم:

```
P = find('B');
q = find ('C');
```

آنگاه p به مجموعهٔ B و q به مجموعهٔ C اشاره خواهد کرد. به شکل C-۱(a) توجه کنید. علاوه‌براین، وجود یک روال merge جهت ادغام دو مجموعهٔ مورد نظر ضروری است. به عنوان مثال، اگر دستورات زیر را اجرا کنیم:

```
p = find('B');
q = find('C');
merge (p , q) ;
```

آنگاه مجموعه‌های شکل C-۱(a) به مجموعه‌های شکل C-۱(b) تبدیل خواهند شد و اگر بر مجموعه‌های شکل C-۱(b) دستورالعمل p = find('B'); را اعمال کنیم، نتیجهٔ مشخص شده در شکل C-۱(c) بدست می‌آید.

شکل C-۱ یک مثال از ساختار داده‌ای مجموعه‌های غیرالحاقی.

(a) پنج مجموعهٔ غیرالحاقی وجود دارد. ما دستورات P=find(B) و q=find(c) را اجرا نموده‌ایم.

(b) پس از ادغام {B},{C}، چهار مجموعهٔ غیرالحاقی وجود دارد.

(c) دستور P=find(B) را اجرا نموده‌ایم

در نهایت، یک روال equal لازم است تا بررسی کند آیا دو مجموعه با هم یکسانند یا خیر؟ برای مثال، اگر بر مجموعه‌های شکل C-۱(b) دستورات زیر را اعمال کنیم، آنگاه با اجرای دستور equal(p, q) مقدار 'true' و با اجرای دستور equal (p, r) مقدار 'false' برگردانده می‌شود. دستورات چنین است:

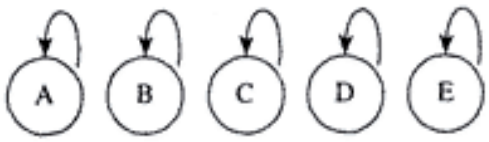
```
p = find('B');
q = find('C');
r = find('A');
```

تاکنون توانسته‌ایم نوع داده‌ای مجرد را با اشیایی شامل عناصر یک مجموعهٔ جهانی، مجموعه‌های غیرالحاقی این عناصر و عملیات makeset، find، merge، و equal مشخص نمائیم. یک روش برای ارائه مجموعه‌های غیرالحاقی، استفاده از درختها با اشاره‌گرهای معکوس است. در این درختها عنصر ریشه به خودش و هر عنصر غیرریشه‌ای به والدینش اشاره می‌کند. شکل C-۲(a)، درخت مرتبط با مجموعهٔ غیرالحاقی در شکل (a) است و شکل C-۲(b)، درخت مرتبط با مجموعه‌های غیرالحاقی در شکل C-۱(b) را نمایش می‌دهد. برای بکارگیری این درختها به ساده‌ترین شکل ممکن می‌توانیم فرض کنیم که مجموعهٔ جهانی ما فقط شامل (اعداد صحیح) می‌باشد و برای توسعه این عمل به دیگر مجموعه‌های جهانی لازم است که عناصر مجموعهٔ جهانی خود را شاخص دهی کنیم. برای بکارگیری درختها از یک آرایه U، که هر شاخص به U معرف یک شاخص در مجموعهٔ جهانی است استفاده می‌کنیم. اگر شاخص i، نمایانگر یک عنصر غیرریشه‌ای باشد، مقدار U[i] برابر شاخص مرتبط با والدینش است و اگر شاخص i معرف یک عنصر ریشه‌ای باشد، مقدار U[i] برابر i است. برای مثال، اگر ۱۰ عنصر در مجموعهٔ جهانی وجود داشته باشد، آنها را در یک آرایه (U) با شاخصهای ۱ تا ۱۰ ذخیره کرده و برای مشخص نمودن مکان عناصر در مجموعه‌های غیرالحاقی از دستور زیر برای تمامی آها استفاده می‌کنیم:

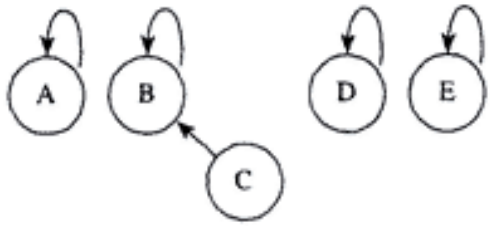
```
U[i] = i
```

شکل C-۲ درخت معکوس متناظر با ساختار داده‌ای یک مجموعهٔ غیرالحاقی.

(a) پنج مجموعهٔ غیرالحاقی با درختهای معکوس مشخص شده‌اند.



(b) درختهای معکوس، پس از ادغام {B} و {C}.



درخت مرتبط با ۱۰ مجموعهٔ غیرالحاقی و آرایه بکار گرفته شده برای آنها در شکل C-۳(a) نشان داده شده است. هنگامی که مجموعه‌های {۴} و {۱۰} با هم ادغام می‌شوند، گرهٔ ۱۰ به عنوان یک فرزند از گرهٔ ۴ تعریف می‌شود که آن را با U[i] = ۴ نمایش می‌دهیم. در حالت کلی، زمانی که دو مجموعه با هم ادغام می‌شوند، ابتدا درختی که شاخص ذخیره شده در ریشه‌اش بزرگتر است را مشخص کرده، سپس این ریشه را به عنوان یک فرزند از ریشه درخت معرفی می‌کنیم. شکل C-۳(c)، درخت مرتبط با مجموعه‌ها و آرایهٔ بکار گرفته شده برای آنها را پس از اعمال چندین عمل ادغام نشان می‌دهد. در اینجا تنها سه مجموعهٔ غیرالحاقی وجود دارد.

درادامه، کاربردی از روالهای معرفی شده را نشان می‌دهیم. برای سهولت، هم در اینجا و هم در بحث الگوریتم kruskal (بخش ۲-۱-۴) از انتقال مجموعهٔ جهانی U به عنوان پارامتر به روال‌ها اجتناب می‌کنیم. مقداری که توسط دستور find(i) بازگردانده می‌شود شاخصی است که درریشهٔ درخت i ذخیره شده است. ما از یک روال به نام initial برای آماده سازی n مجموعهٔ غیرالحاقی استفاده خواهیم کرد. وجود این روال در اغلب الگوریتم‌هایی که از مجموعه‌های غیرالحاقی استفاده می‌کنند ضروری است.

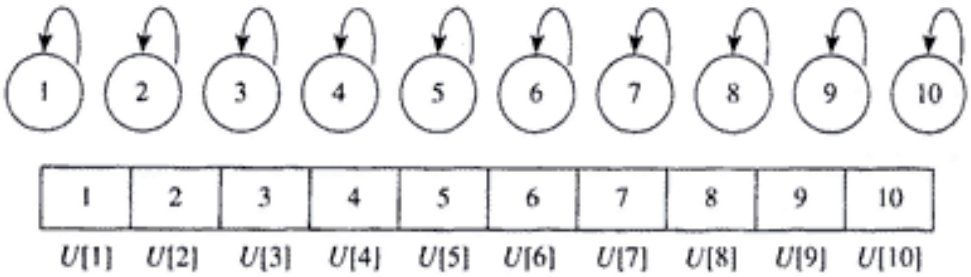
در بسیاری از الگوریتم‌هایی که مجموعه‌های غیرالحاقی را به کار می‌گیرند، ابتدا n مجموعهٔ غیرالحاقی را تولید کرده، سپس m بار از حلقه‌ای گذر می‌کنیم (m و n لزوماً یکسان نیستند) که در هر گذر از این حلقه، تعداد یکسانی از روال‌های equal، find، و merge فراخوانی می‌شوند. به منظور تحلیل الگوریتم، به پیچیدگی‌ها زمانی مقداردهی و عملکرد حلقه در عناصر n و m نیازمندیم. پرواضح است که پیچیدگی زمانی روال initial برابراست با $\theta(n)$

ساختار داده‌ای مجموعهٔ غیرالحاقی I

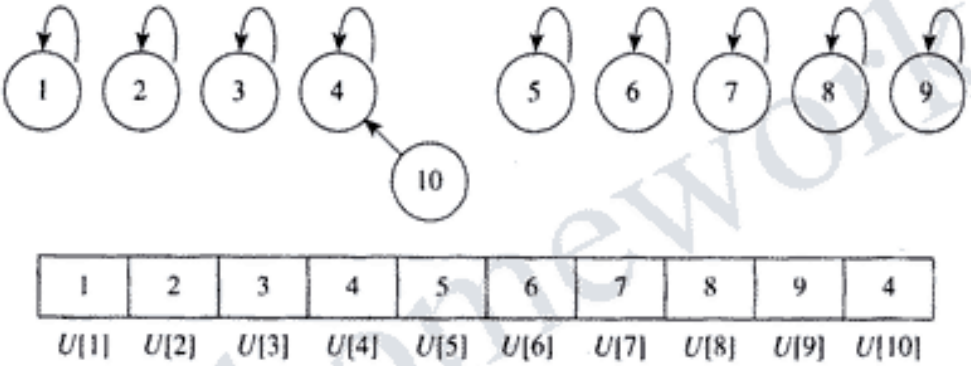
```
const n = the number of elements in the universe;

typedef int index;
typedef index set_pointer;
typedef index universe[1..n];          // universe is indexed from 1 to n.

universe U;

void makeset (index i)
{
  U[i] = i;
}

set_pointer find (index i)
{
  index j;

  j = i;
  while (U[j] != j)
    j = U[j];
  return j;
}

void merge (set_pointer p, set_pointer q)
{
  if (p < q)                           // p points to merged set;
    U[q] = p;                          // q no longer points to a set.
  else
    U[p] = q;                          // q points to merged set;
}                                      // p no longer points to a set.

bool equal (set_pointer p, set_pointer q)
{
  if (p == q)
    return true;
  else
    return false;
}

void initial (int n)
{
  index i;

  for (i = 1; i <= n; i++)
    makeset(i);
}
```

شکل C-۳ بکارگیری آرایهٔ درخت معکوس متناظر با ساختار داده‌ای یک مجموعه غیرالحاقی.

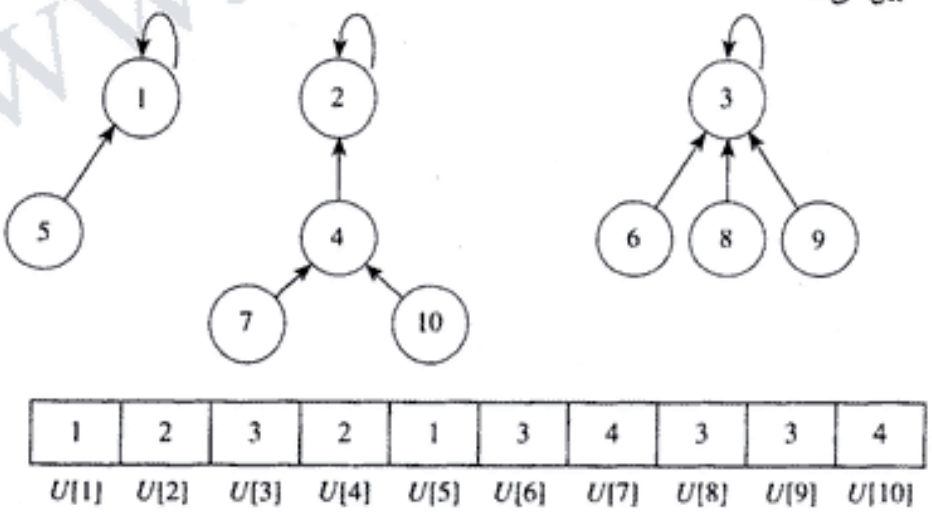
(a) درختهای معکوس و آرایه بکار گرفته شده برای ۱۰ مجموعه غیرالحاقی.



(b) مجموعه‌های {۴} و {۱۰} در بخش (a) با هم ادغام شده‌اند. اندیس آرایهٔ دهم، مقدار ۴ می‌گیرد.



(c) درختهای معکوس و آرایهٔ بکار گرفته شده، پس از چندین مرتبه ادغام. ترتیب ادغام‌ها، ساختار درختها را تعیین می‌کند.



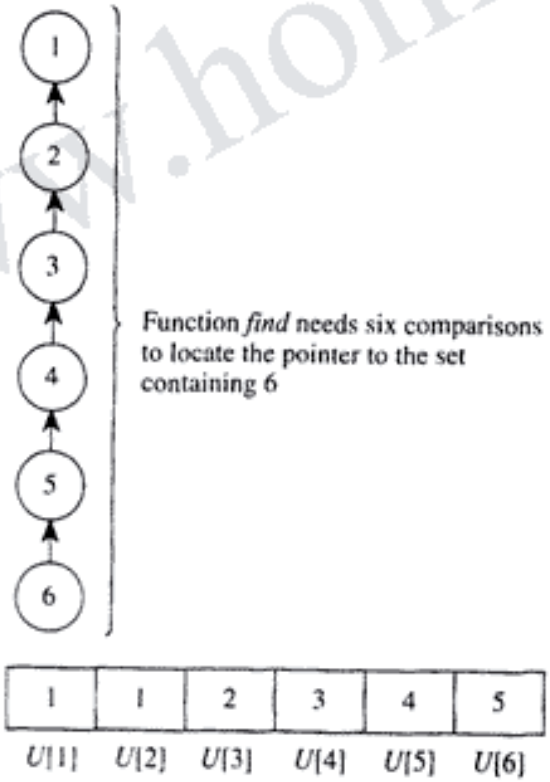
ازآنجائیکه ثابت ضربی بر ترتیب بی‌تأثیر است، لذا می‌توانیم فرض کنیم که روالهای equal، find، و merge در مدت زمان ثابتی اجرا می‌شوند و فقط روال find شامل یک حلقه است. بنابراین، ترتیب پیچیدگی زمانی تمامی فراخوانی‌ها به تابع find وابسته است. بیائید بدترین حالت تعداد مقایسات را در این روال بررسی کنیم. برای مثال، فرض کنیم m=۵ که در این صورت بدترین حالت زمانی رخ می‌دهد که چنین دنباله‌ای از دستورات merge را داشته باشیم:

```
merge({۵}, {۶});
merge({۴}, {۵ ،۶});
merge({۳}, {۴ ،۵ ،۶});
merge({۲}, {۳ ،۴ ،۵ ،۶});
merge({۱}, {۲ ،۳ ،۴ ،۵ ،۶});
```

و بعد از هر فراخوانی روال merge تابع find را برای جستجو شاخص ۶ فراخوانی کنیم. درخت نهایی و آرایهٔ بکار گرفته شده در شکل C-۴ نشان داده شده است. مجموع تعداد مقایسات در تابع find برابر است با

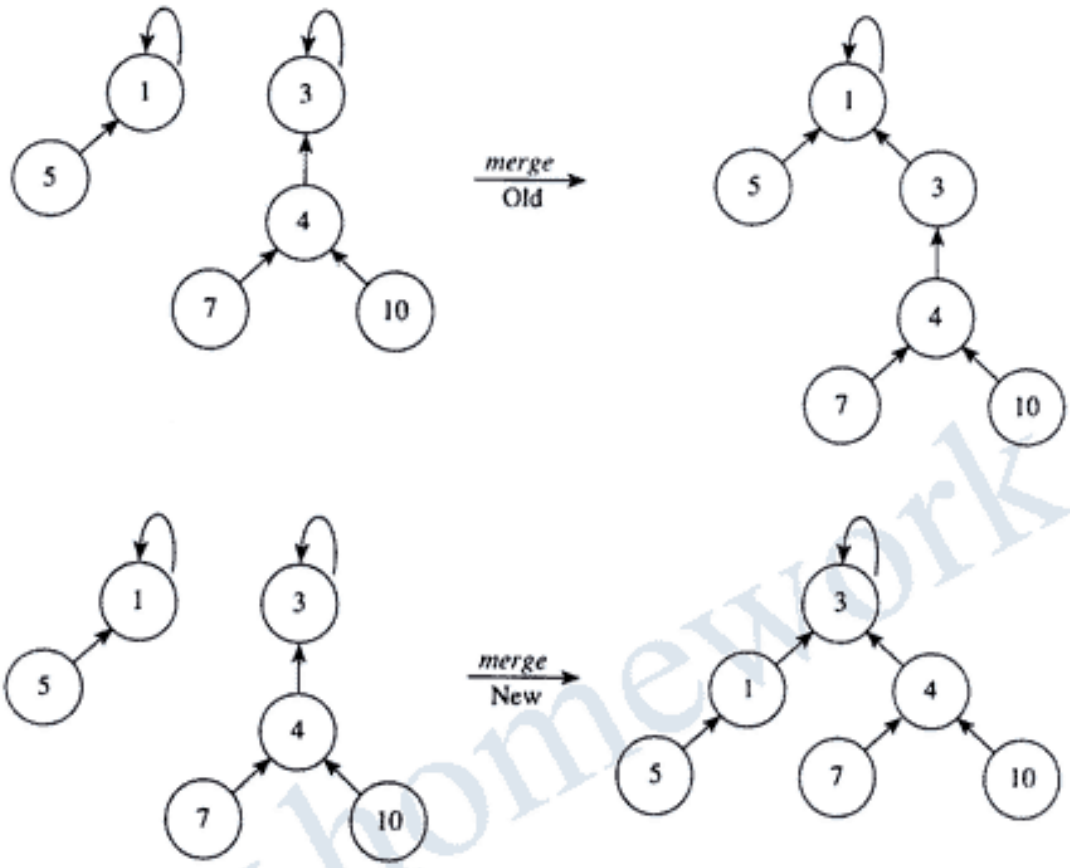
$$۲ + ۳ + ۴ + ۵ + ۶$$

و برای یک m اختیاری بدترین تعداد مقایسات برابر با $۲ + ۳ + \ldots + (m + ۱) \in \theta(m^۲)$. ما تابع equal را درنظر نگرفتیم زیرا هیچ تأثیری بر تعداد مقایسات انجام شده در find ندارد.



شکل C-۴ یک مثال از بدترین حالت ساختار داده‌ای مجموعه‌های غیرالحاقی، وقتی n=۵ است.

شکل C-۵ در روش جدید ادغام، ریشه درخت با عمق کمتر را به عنوان فرزند درخت دیگر قرار می‌دهیم.



گره‌های آن است انجام شود. اگر بتوانیم روال merge را به گونه‌ای تغییر دهیم که این حالت پیش نیاید. توانسته‌ایم کارایی الگوریتم را بهبود بخشیم. برای این منظور می‌توانیم مسیری از عمق درخت را نگهداری کرده و هنگام انجام عمل merge، درختی بسازیم که عمقی کوچکتر از دیگر درختهای ممکن داشته باشد. شکل C-۵، روش قدیمی ادغام را با روش جدید آن مقایسه کرده است. توجه دارید که روش جدید تنها در یک درخت با عمق کوچکتر معنا می‌یابد. برای بکارگیری این روش لازم است که عمق درخت در هر ریشه ذخیره شود. الگوریتم زیر این موارد را شامل می‌شود:

ساختار داده‌ای مجموعهٔ غیرالحاقی II

```
const n = the number of elements in the universe;

typedef int index;
typedef index set_pointer;

struct nodetype
{
   index parent;
   int depth;
```

```
typedef nodetype univere[1..n];        // universe is indexed from 1 to n.
universe U;

void makeset (index i);
{
   U[i].parent = i;
   U[i].depth = 0;
}

set_pointer find (index i)
{
   index j;

   j = i;
   while (U[j].parent != j)
      j = U[j].parent;
   return j;
}

void merge (set_pointer p, set_pointer q)
{
   if (U[p].depth == U[q].depth) {
      U[p].depth = U[p].depth + 1;      // Tree's depth must increase.
      U[q].parent = p;
   }
   else if (U[p].depth < U[q].depth)    // Make tree with lesser depth
      U[p].parent = q;                  // the child.
   else
      U[q].parent = p;
}

bool equal (set_pointer p, set_pointer q)
{
   if (p == q)
      return true;
   else
      return false;
}

void initial (int n)
{
   index i;

   for (i = 1; i <= n; i++)
      makeset(i);
}
```

ما می‌توانیم نشان دهیم که بدترین حالت تعداد مقایسات انجام شده در m گذر از حلقه، شامل تعداد ثابتی از فراخوانی روال‌های equal، find، و merge، و برابر با $\theta(m \lg m)$ می‌باشد.

در برخی از کاربردها لازم است که محل کوچکترین عضو یک مجموعه مشخص شود. پرواضح است که با بکارگیری روش اول، این مسئله حل شده است زیرا در این روش کوچکترین عضو مجموعه همیشه در ریشهٔ درخت جای دارد. اما در روش دوم، لزوماً چنین نیست. لذا می‌توانیم با یک تغییر جزئی در روش فوق این مسئله را به راحتی حل کنیم و آن اینکه از یک متغیر به نام smallest در ریشه M هر درخت برای ذخیره کوچکترین شاخص درخت استفاده کنیم. به الگوریتم زیر توجه نمائید:

ساختار داده‌ای مجموعهٔ غیرالحاقی III

```
const n = the number of elements in the universe;

typedef int index;
typedef index set_pointer;

struct nodetype
{
   index parent;
   int depth;
   int smallest;
};
typedef nodetype universe[1..n];      // universe is indexed from 1 to n.

universe U;

void makeset (index i)
{
   U[i].parent = i;
   U[i].depth = 0;
   U[i].smallest = i;                 // The only index i is smallest.
}

void merge (set_pointer p, set_pointer q)
{
   if (U[p].depth == U[q].depth) {
      U[p].depth = U[p].depth + 1;    // Tree's depth must increase.
      U[q].parent = p;
      if (U[q].smallest < U[p].smallest)   // q's tree contains smallest
         U[p].smallest = U[q].smallest;    // index.
   }
```

```
    else if (U[p].depth < U[q].depth) {    // Make tree with lesser depth
      U[p].parent = q;                     // the child.
      if (U[p].smallest < U[q].smallest)   // p's tree contains smallest
        U[q].smallest = U[p].smallest;     // index.
    }
    else {
      U[q].parent = p;
      if (U[q].smallest < U[p].smallest)   // q's tree contains smallest
        U[p].smallest = U[q].smallest;     // index.
    }
}

int small (set_pointer p)
{
    return U[p].smallest;
}
```

در الگوریتم فوق، تنها روالهایی را ذکر کردیم که با روالهای مذکور در ساختار داده‌ای مجموعهٔ غیرالحاقی II متفاوت بودند. تابع small، کوچکترین عضو یک مجموعه را برمی‌گرداند. بدلیل اینکه این تابع، زمان اجرای ثابتی دارد، لذا بدترین حالت تعداد مقایسات انجام شده در m گذر از حلقه، شامل تعداد ثابتی از فراخوانی روال‌های equal، find، merge و small، و دقیقاً برابر با روش دوم، یعنی $\theta(m \lg m)$ می‌باشد. این تکنیک که به **فشرده‌سازی مسیر** موسوم است، امکان توسعهٔ این ساختار را، که بدترین حالت تعداد مقایسات در m گذر از حلقه روی m تقریباً خطی است، فراهم ساخته است. این روش در کتاب Bartley و Brassard (۱۹۸۸) بحث شده است.