طراحی الگوریتم ها

با استفاده از شبه کد C++

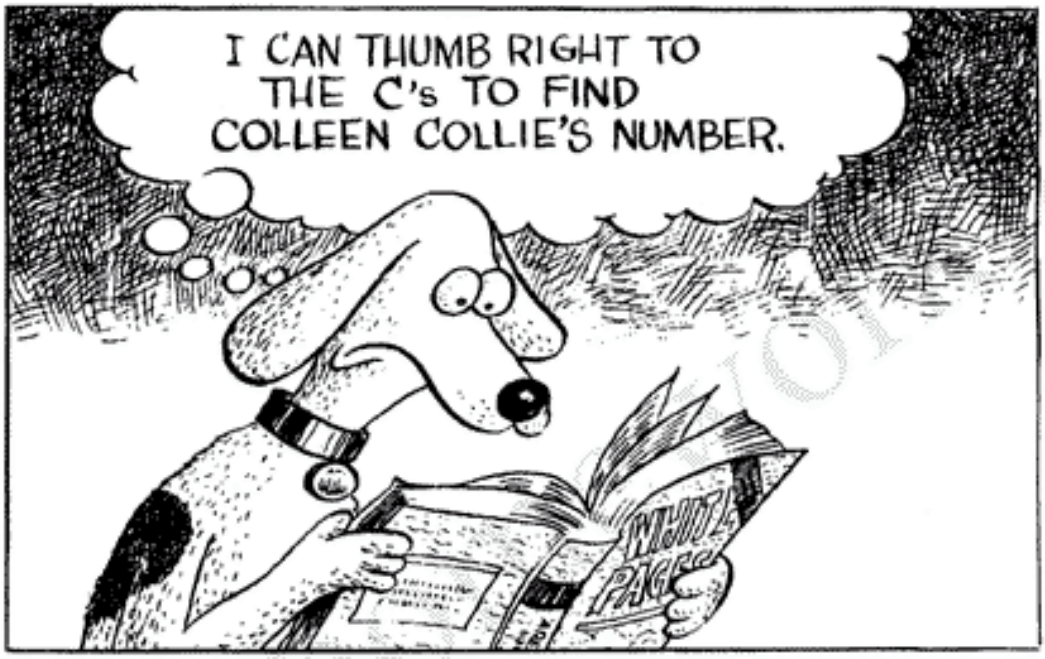
نویسنده

ریچارد نئوپولیتن کیومرث نعیمی پور

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؛ چراکه 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 ۲-۱ 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 ۳-۱ یک نمونه ازمسئله مطرح شده در مثال ۱-۱ چنین است :

$S = [۱۰، ۷، ۱۱، ۵، ۱۳، ۸]$ و $n = ۶$

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

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

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

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 ۳-۱ 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."

تعریف الگوریتم

مثال ۵-۱ یک الگوریتم برای مثال ۲-۱ 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 ۵-۱ 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;                    بهتر است از          exchange x and y;
y = temp;
```

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 بازگرداندن 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،

$$\begin{bmatrix} ۲ & ۳ \\ ۴ & ۱ \end{bmatrix} \times \begin{bmatrix} ۵ & ۷ \\ ۶ & ۸ \end{bmatrix} = \begin{bmatrix} ۲\times۵+۳\times۶ & ۲\times۷+۳\times۸ \\ ۴\times۵+۱\times۶ & ۴\times۷+۱\times۸ \end{bmatrix} = \begin{bmatrix} ۲۸ & ۳۸ \\ ۲۶ & ۳۶ \end{bmatrix}$$

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 ۲-۱ 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

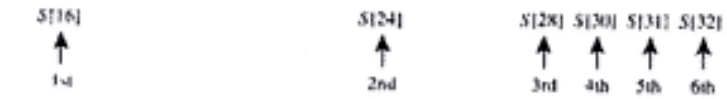
خروجی: مکان کلید x در آرایهٔ S (صفر، اگر x در آرایهٔ S نباشد).

```
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 $\lg ۳۲ + ۱ = ۶$ (منظور از lg همان $\log_۲$ 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.

اندازه 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.

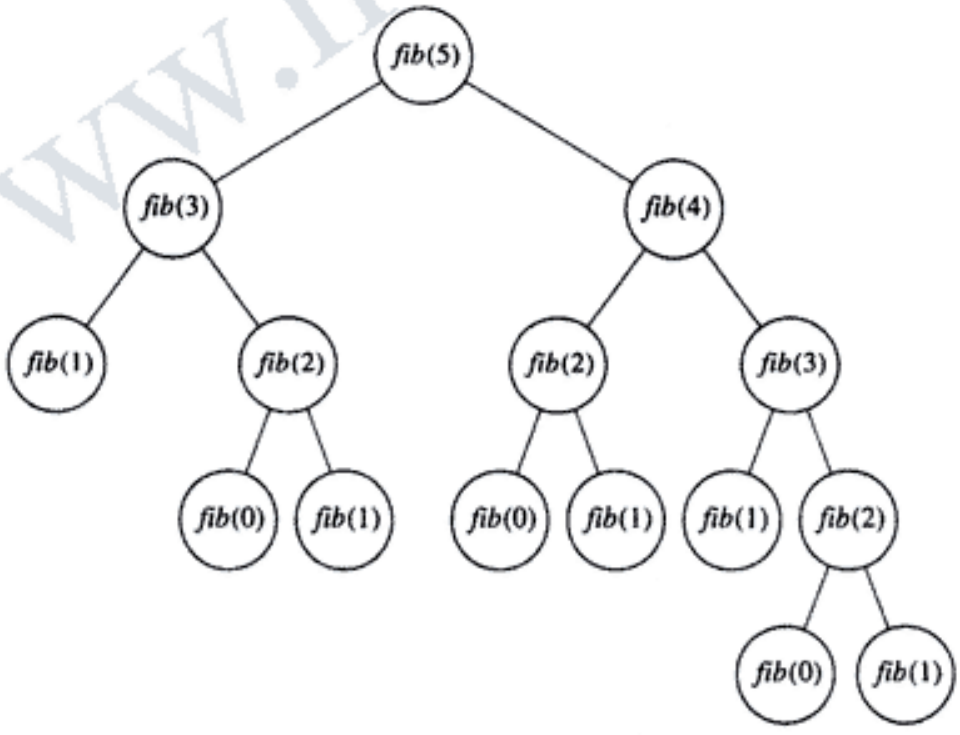
```
int fib (int n)
{
  if (n >= 1)
    retrun n;
  else
    return fib(n – 1) + fib(n – 2);
}
```

"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 $۰ \le n \le ۶$ 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 $1 \le n \le 5$ 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:



شکل ۲-۱ درخت متناظر با الگوریتم ۶-۱ به هنگام محاسبهٔ عنصر پنجم فیبوناچی.

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

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.

۱-۳ 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 $1 + \lfloor \lg n \rfloor$. 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.

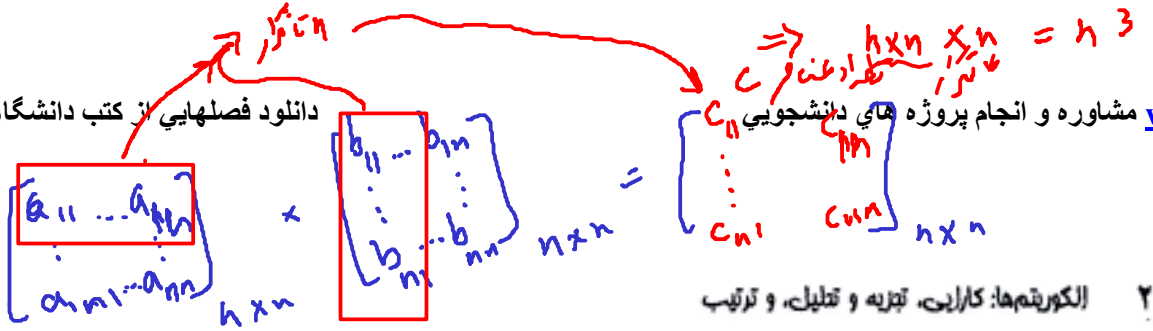
تحلیل پیچیدگی زمانی حالت معمول الگوریتم ۴-۱ (ضرب ماتریس‌ها)

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 اجرا 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 k گذر از حلقه انجام شود تا به عنصر x در اندیس kام دست یابیم یا n گذر 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، انگاه $A(n) = (n+1)/2$ و اگر $p = 1/2$ باشد، انگاه $A(n) = 3n/4 + 1/4$ 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 را به 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 مبانی 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، $1000t$ مدت زمانی است که الگوریتم اول برای یک بار پردازش عمل مبنایی‌اش صرف می‌کند. برای ساده‌تر شدن مسئله، از محاسبهٔ زمان 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 $1000t \times n$ 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:

$$n^2 \times t > n \times 1000t$$

با تقسیم طرفین نامساوی به nt داریم:

$$n > 1000$$

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 $0.01n^2$ 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

$$0.01n^2 > 100n$$

با تقسیم دو طرف نامساوی به $0.01n$ داریم:

$$n > 10,000$$

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 $0.01n^2$، 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؛ درحالیکه تابعی چون $100 + n + 0.01n^2$، به‌دلیل 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 $100 + n + 0.01n^2$ 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 $0.01n^3 + 10n^2 + 5n + 25$ 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، لذا با 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 ۳-۱ زمانهای اجرا برای الگوریتم‌هایی با پیچیدگی زمانی معین.

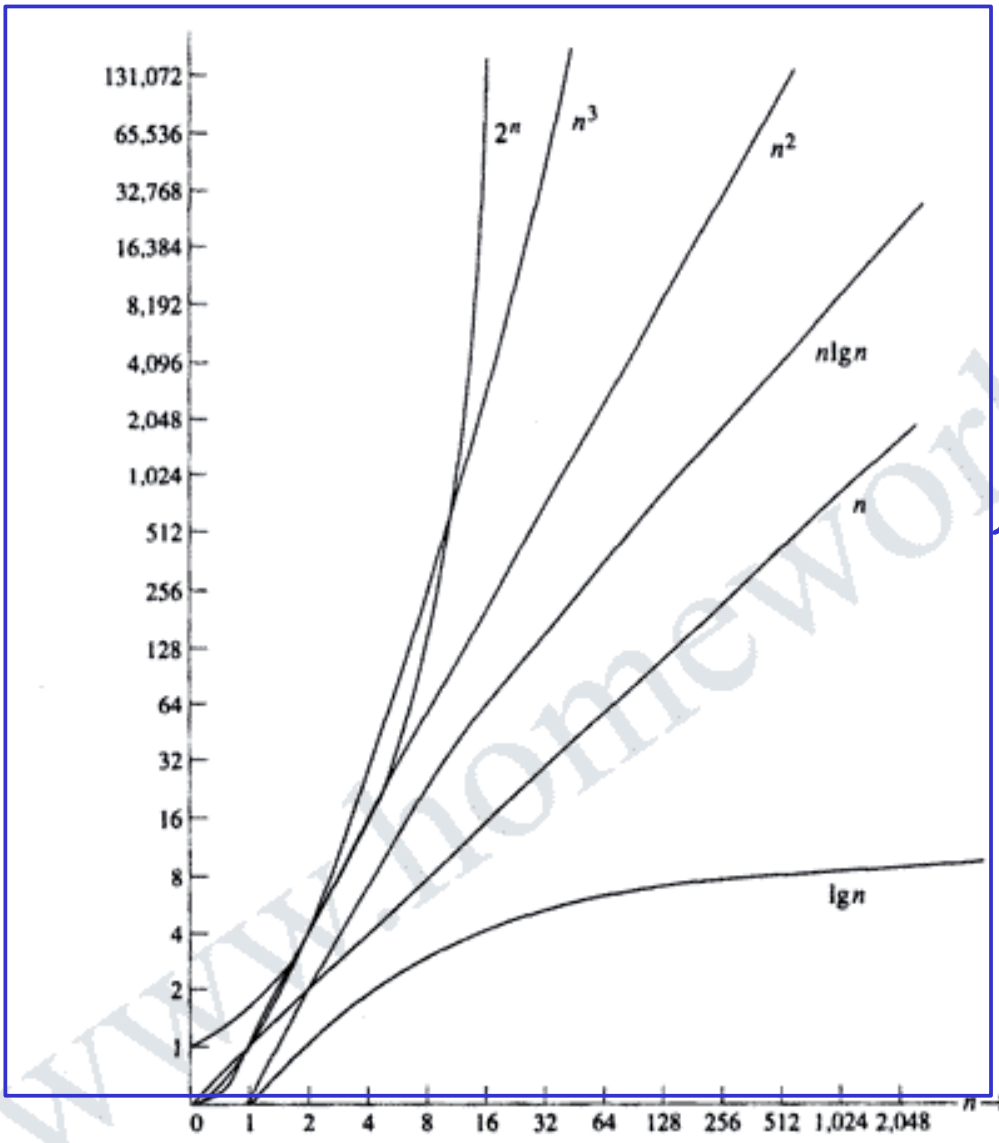
n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs*	0.01 μs	0.033 μs	0.1 μs	1 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 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.



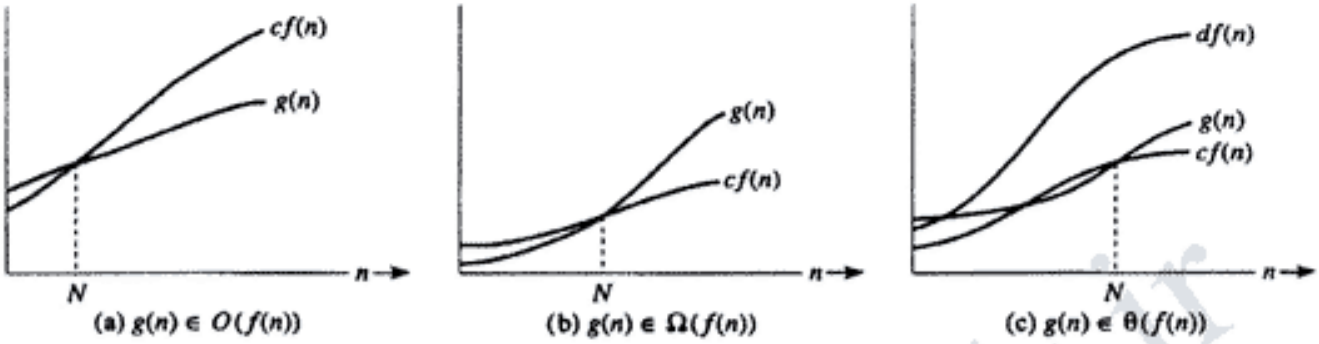
۲-۴-۱ معرفی کامل ترتیب

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:

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

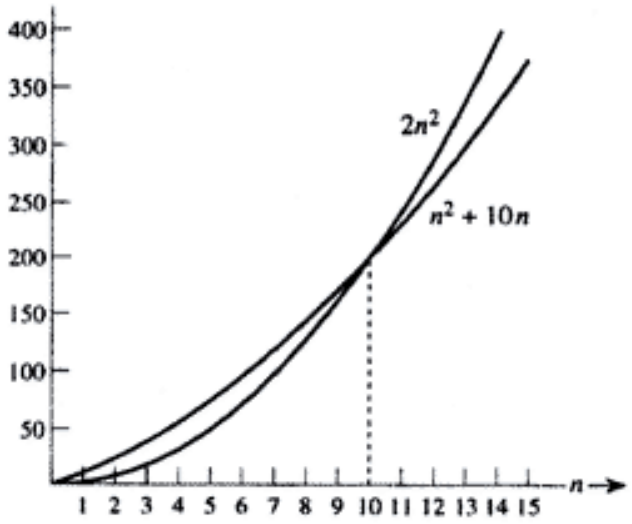
شکل ۱-۴ پیچیدگی‌های زمانی big O، Ω و Θ



اگر $g(n) \in O(f(n))$ باشد، می‌گوئیم $g(n)$، یک big O از $f(n)$ است. شکل ۱-۴(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 ۱-۵، 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 ۱-۵ 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 ۷-۱ 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 ۸-۱ 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 ۹-۱ 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 $c = 11$ 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 ۱۰-۱ 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

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

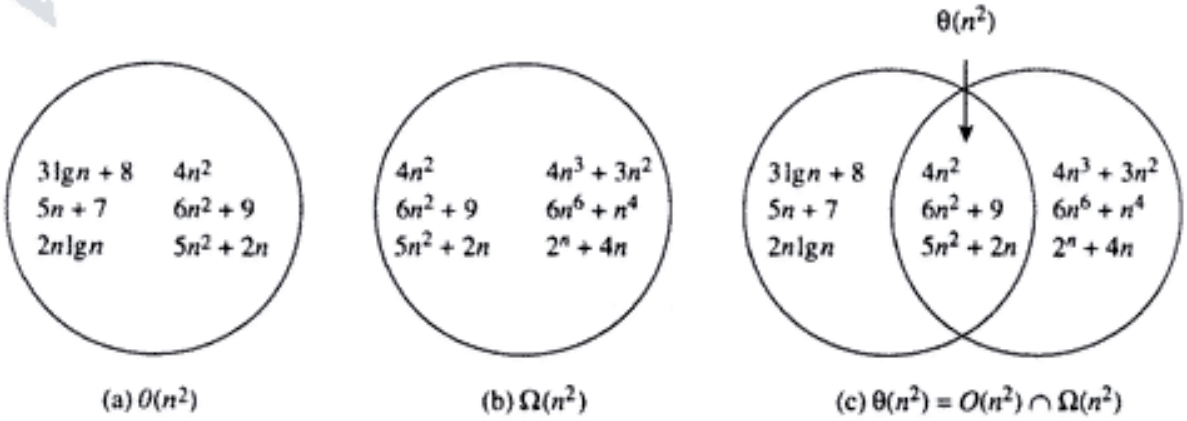
لذا با انتخاب $c = 1$ و $N = 1$ به نتیجه مطلوب خود می‌رسیم.

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 ۶-۱(b)، امگا را نشان می‌دهد. به مثالهای زیر توجه کنید:



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 ۶-۱ 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 ۱-۱۶ 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 (c)۶-۱ نشان می‌دهد که $\Theta(n^2)$ از اشتراک $O(n^2)$ و $\Omega(n^2)$ بدست می‌آید. شکل (c)۴-۱ نیز Θ را نشان داده است. در شکل (c)۶-۱ 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 ۱-۱۷ 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.

if $n \in \Omega(n^2) \Rightarrow \exists c, N : n \geq cn^2 \Rightarrow n \leq \frac{1}{c}$ ...

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

Small-O

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 ۱۸-۱ 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 ۱۹-۱ 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.

$5n \notin o(n)$

$5n \leq cn \Rightarrow 5 \leq 1$ ✗

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،

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

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. رده 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 $b > 1$ و $a > 1$ 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:

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

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،

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

مثال ۲۳-۱ ویـژگیهای ۶ و ۷ می‌توانند مکـرراً استفاده شـوند. بـه عـنوان مـثال، مـی‌توانیم بـا بکـارگیری مکـرر ویژگیهای ۶ و ۷، نشان دهیم که $\mathfrak{d}n + \mathfrak{r}lgn + \mathfrak{l}\circ n\,lgn + \mathfrak{r}n^{\mathfrak{r}} \in \Theta(n^{\mathfrak{r}})$ داریم:

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

که نتیجه می‌دهد

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

که نتیجه می‌دهد

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

که نتیجه می‌دهد

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

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؛ چرا که 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؛ چرا که 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 $a\le 1$ باشد، نتیجه جزئی و ناچیز است. فرض کنید که $a>1$ باشد. اگر n نیز به اندازه‌ای بزرگ باشد که $n>2\lceil a^2\rceil$، آنگاه داریم:

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

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 و جود 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- عمليات مبنايي را براي الگوريتم‌هاي تمرينات ۱ تا ۷ مشخص نموده، كارايي اين الگوريتم را مورد ارزيابي قرار دهيد. اگر الگوريتمي پيچيدگي زماني حالت معمول دارد، آن را مشخص كنيد؛ در غير اينصورت، پيچيدگي زماني بدترين حالت الگوريتم را تعيين نمائيد.

۱۱- پيچيدگي زماني بدترين حالت، حالت ميانی و بهترين حالت مرتب‌سازی درجی اصلی و مرتب‌سازی درجی تمرين ۴، كه در آن از جستجوی دودويی استفاده شده است، را مشخص كنيد.

۱۲- يک الگوريتم زمان-خطی بنويسيد كه n عدد صحيح غيرتكراری بين ۱ تا ۵۰۰ را مرتب كند.
راهنمائی: از يک آرايهٔ ۵۰۰ عنصری استفاده كنيد.

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

۱۴- برای مسئله‌ای با اندازه n، دو الگوريتم با نامهای Alg۱ و Alg۲ وجود دارند. Alg۱ در $n^۲$ ميكروثانيه و Alg۲ در $۱۰۰n\log n$ ميكروثانيه اجرا می‌شود. Alg۱ برای پياده‌سازی، به ۴ ساعت كار برنامه نويسی و ۲ دقيقه زمان CPU نياز دارد. از طرف ديگر، Alg۲ برای پياده‌سازی، به ۱۵ ساعت كار برنامه‌نويسی و ۶ دقيقه زمان CPU نياز دارد. اگر دستمزد برنامه‌نويسان برای هر ساعت كار، ۲۰ دلار باشد و هر دقيقه از كار CPU، ۵۰ دلار هزينه داشته باشد، يک نمونه مسئله با اندازه ۵۰۰، چند مرتبه بايستی به وسيله Alg۲ حل شود تا هزينهٔ اين كار را توجيه كند؟

بخش ۴-۱

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

۱۶- با استفاده از تعاريف O و Ω نشان دهيد كه $۶n^۲+۲۰n\in O(n^۳)$، اما $۶n^۲+۲۰n\notin\Omega(n^۳)$.

۱۷- با استفاده از ويژگيهای ترتيب در بخش ۲-۴-۱ نشان دهيد كه

$$۵n^۵+۴n^۴+۶n^۳+۲۰n^۲+n+۷\in\Theta(n^۵)$$

۱۸- فرض كنيد $P(n)=a_kn^k+a_{k-۱}n^{k-۱}+\ldots+a_۱n+a_۰$، كه $a_k>۰$ است. با استفاده از ويژگيهای ترتيب در بخش ۲-۴-۱ نشان دهيد كه

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

۱۹- توابع زير را در رده‌های پيچيدگی دسته‌بندی كنيد:

$$n\ln n \quad (\lg n)^۲ \quad ۵n^۲+۷n \quad n^{۵/۲} \quad n! \quad ۳^{n!} \quad ۴^n \quad n^n \quad n^n+\ln n$$
$$۵^{\lg n} \quad (\lg !) \quad (\lg n)! \quad \sqrt{n} \quad e^n \quad ۸n+۱۲ \quad ۱۰^n+n^{۲۰}$$

۲۰- ويژگيهای ۱، ۲، ۶ و ۷ از ويژگيهای ترتيب بخش ۲-۴-۱ را ثابت كنيد.

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) $T(n) \in \Theta(۱۰^n)$

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

۲۷- الگوريتمی برای مسئله زير ارائه دهيد و پيچيدگی زمانی آن را تعيين کنيد. يک ليست شامل n عنصر مثبت مجزا مفروض است، ليست را به دو زيرليست تقسيم کنيد بطوری که اندازهٔ هر زيرليست برابر n/۲ باشد و اختلاف بين مجموع اعداد صحيح دو زيرليست، حداکثر شود. می‌توانيد فرض کنيد که n مضربی از ۲ است.

۲۸- يک الگوريتم $\Theta(n \lg n)$ ارائه دهيد که باقيماندهٔ تقسيم x^n بر P را محاسبه کند. می‌توانيد n را توانی از ۲ درنظر بگيريد ($n = ۲^k$).

۲۹- توضيح دهيد چه توابعی در مجموعه‌های زير قرار می‌گيرند؟

$$(a)\ n^{o(۱)} \qquad (b)\ O(n^{o(۱)}) \qquad (c)\ O(O(n^{o(۱)}))$$

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

۳۱- الگوريتمی برای مسئله زير بنويسيد. يک ليست با n عدد صحيح مثبت مجزا مفروض است. آن را طوری به دو زير ليست به اندازه‌های n/۲ تقسيم کنيد که اختلاف بين مجموعه اعداد صحيح دو زيرليست، حداقل شود. پيچيدگی زمانی الگوريتم را تعيين کنيد. می‌توانيد n را مضربی از ۲ درنظر بگيريد.

۳۲- می‌دانيم که الگوريتم ۷-۱ (nامين عنصر فيبوناچی، تکرار) در n به صورت خطی است. آيا اين الگوريتم يک الگوريتم زمان-خطی می‌باشد؟ در اين الگوريتم، n يک ورودی است و تعداد بيتهايی که برای کد کردن n بکار می‌روند، اندازه ورودی می‌باشند. نشان دهيد که الگوريتم ۷-۱، برحسب اندازه ورودی، به صورت زمان-نمايی است. همچنين نشان دهيد که هر الگوريتمی که برای محاسبهٔ nامين عنصر فيبوناچی نوشته شود، بايد يک الگوريتم زمان-نمايی باشد زيرا اندازهٔ خروجی به صورت نمايی از اندازه ورودی است. در بخش ۲-۹، پيچيدگی زمانی الگوريتم ۶-۱ (عنصر nام فيبوناچی، بازگشتی) بر اساس اندازه ورودی آن تعيين 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
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)

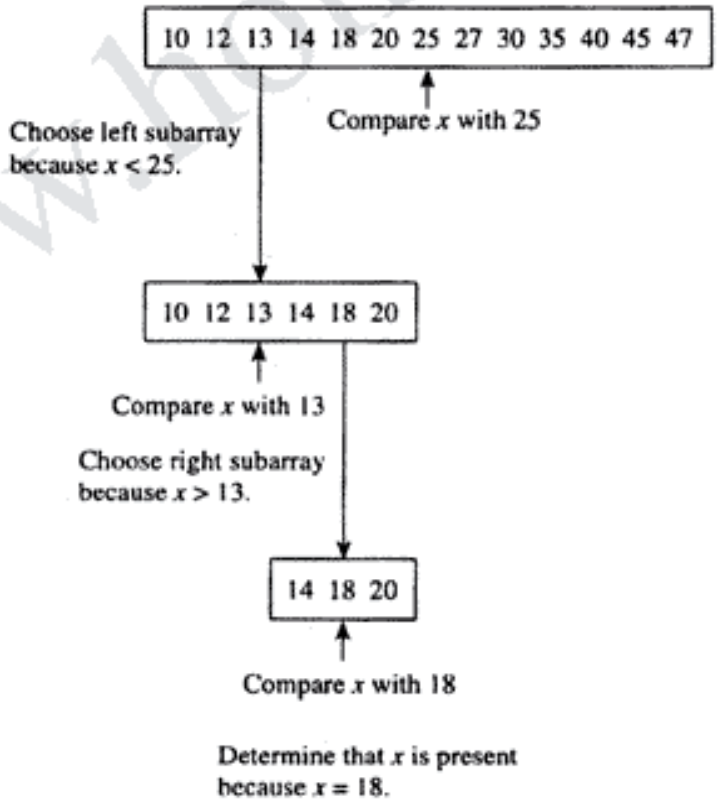
مسئله: تعیین کنید که آیا x در آرایه مرتب S با اندازه ورودی n وجود دارد یا خیر؟

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 باگشتی 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(۱, 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 B-۱ 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:

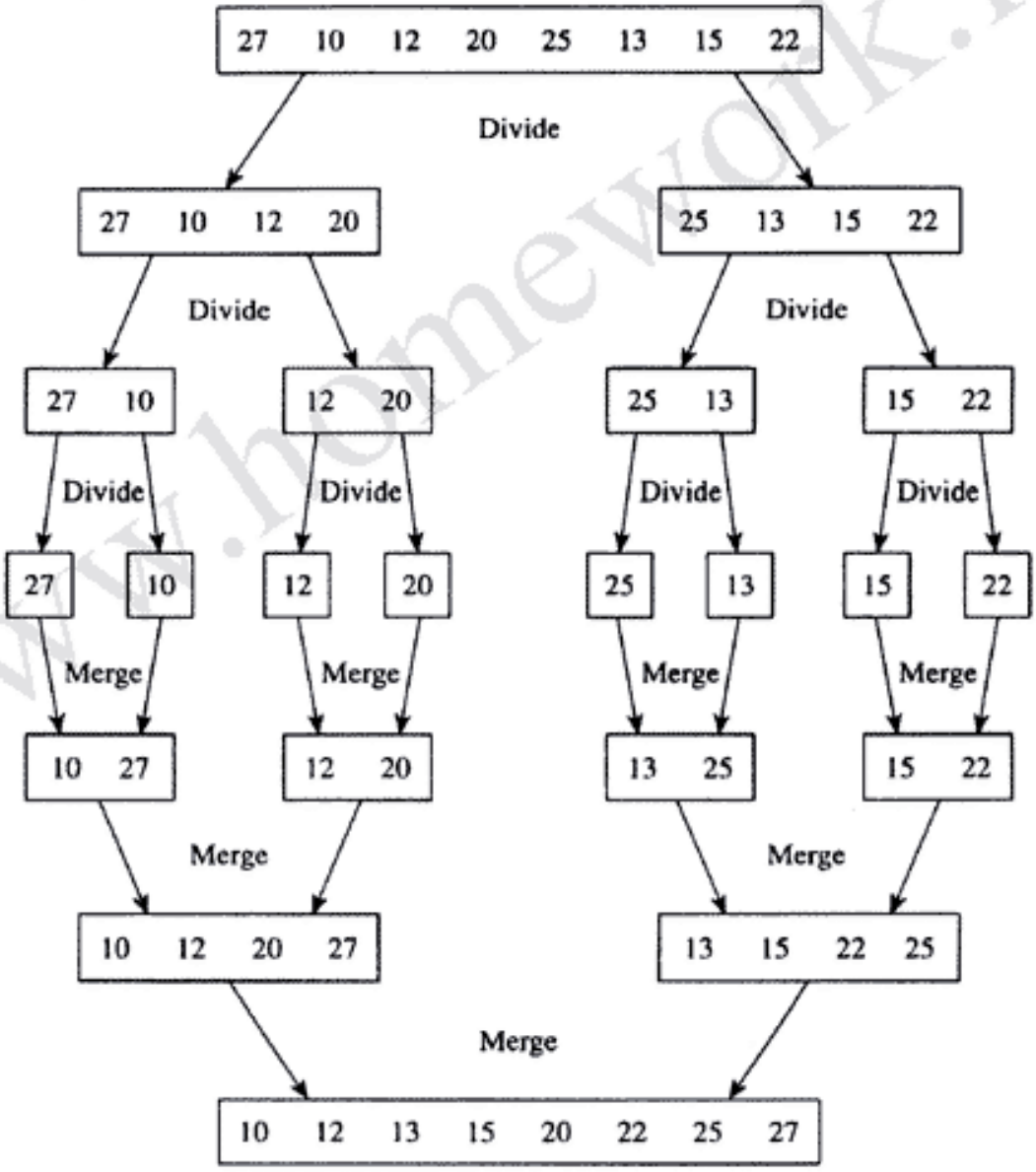
۲۵ ۲۲ ۱۵ ۱۳ و ۲۷ ۲۰ ۱۲ ۱۰

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، چراکه 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؛ چراکه 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 می‌کند، 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،

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

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

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

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

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 B-۱۹، 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

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

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 mergesort2(۱,n); 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 mid+۱ 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:

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:

۱۰ ۱۳ ۱۲ | ۱۵ | ۲۲ ۲۷ ۲۰ ۲۵

عناصرکوچکتر — عنصرمحوری — عناصربزرگتر

۲- زیرآرایه‌ها را مرتب کنید:

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.

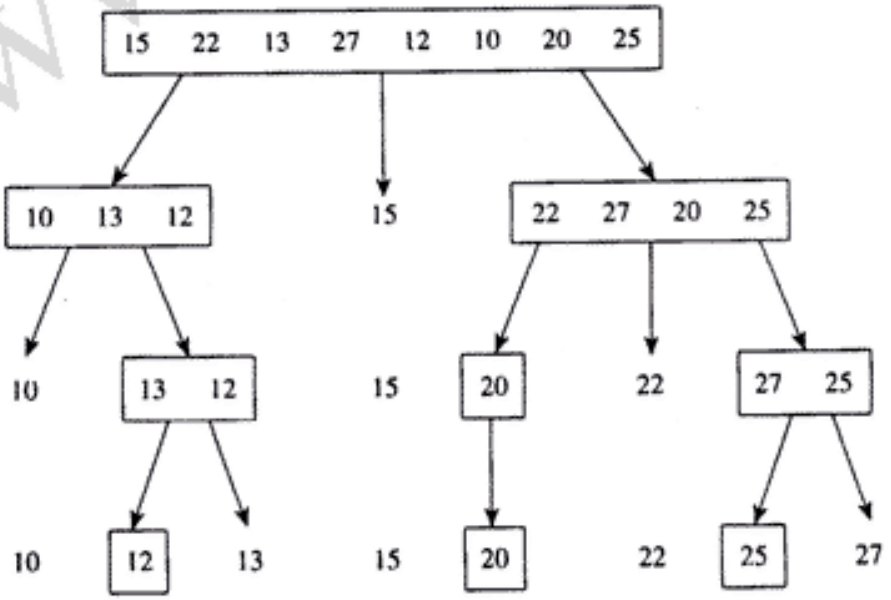
الگوریتم ۶-۲ مرتب‌سازی سریع (Quicksort)

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

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

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:

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

همانند بازگشت ارائه شده در مثال B-22، جواب تقریبی این بازگشت چنین است:

$$a_n \approx 2 \ln n$$

که اشاره دارد به اینکه

$$A(n) \approx (n+1)\, 2 \ln n = (n+1)\, 2 (\ln 2)(\lg n)$$
$$\approx 1/38 (n+1) \lg n \in \Theta(n \lg n)$$

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:

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