مرور فصل ۲

اهداف

- معرفی چارچوب‌هایی برای توصیف و تحلیل الگوریتم‌ها
- بررسی دو الگوریتم برای مرتب‌سازی: مرتب‌سازی درجی و مرتب‌سازی ادغامی
- مشاهده نحوه توصیف الگوریتم‌ها به صورت شبه‌کد
- شروع استفاده از نماد مجانبی برای بیان تحلیل زمان اجرا
- یادگیری تکنیک «تقسیم و حل» در چارچوب مرتب‌سازی ادغامی

مرتب‌سازی درجی

در یک مسئله مرتب‌سازی

- **ورودی:** دنباله‌ای از n عدد $(a_1, a_2, \dots, a_n)$.
- **خروجی:** یک جایگشت (بازچینی) $(a'_1, a'_2, \dots, a'_n)$ از دنباله ورودی به طوری که $a'_1 \le a'_2 \le \cdots \le a'_n$.

دنباله‌ها معمولاً در آرایه‌ها ذخیره می‌شوند. به اعداد با نام کلید نیز اشاره می‌شود. همراه با هر کلید ممکن است اطلاعات اضافی‌ای وجود داشته باشد که به عنوان داده‌های همراه شناخته می‌شوند. [لازم به ذکر است که «داده‌های همراه» لزوماً از ماهواره نمی‌آیند.]

بیان الگوریتم‌ها

ما الگوریتم‌ها را به هر روشی که واضح‌ترین باشد بیان می‌کنیم. همیشه زبان انگلیسی بهترین روش نیست. وقتی مسائل نیاز به توضیح کامل دارند، اغلب از **شبه‌کد (pseudocode)** استفاده می‌کنیم.

شبه کد (Pseudocode) چیست؟

شبه کد یک روش غیررسمی و انسان‌فهم برای توصیف الگوریتم‌ها (مراحل حل یک مسئله) بدون در نظر گرفتن قواعد سختگیرانه یک زبان برنامه‌نویسی خاص نوشته می‌شود.

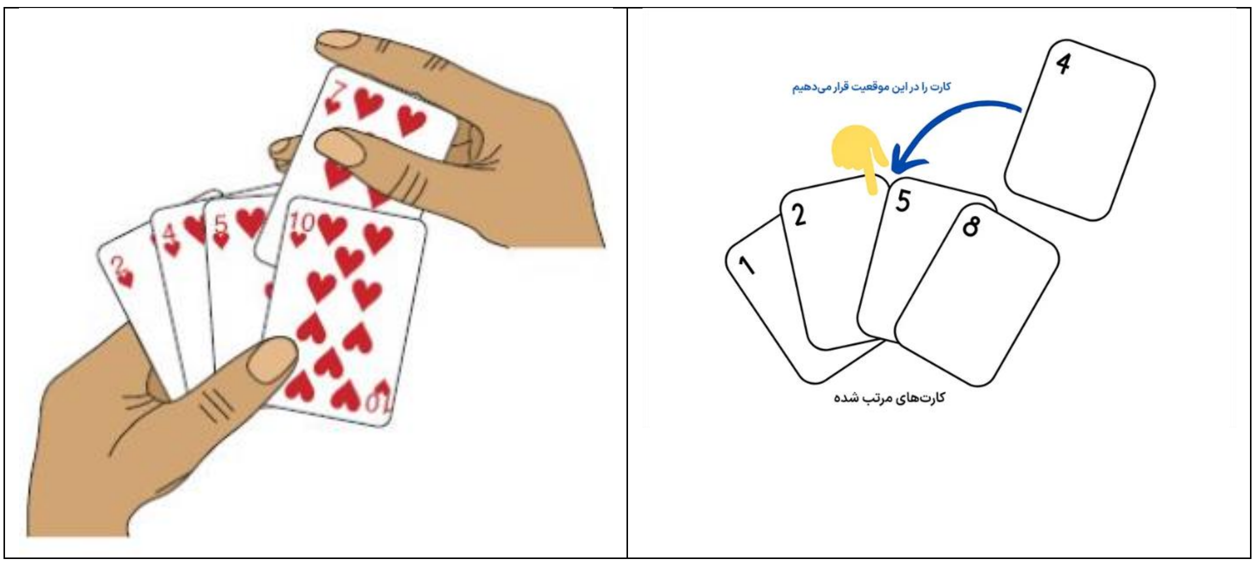
می‌توان آن را به عنوان "طرح اولیه" یا "اسکلت‌بندی" یک برنامه قبل از نوشتن کد واقعی در نظر گرفت

ویژگی‌های شبه‌کد

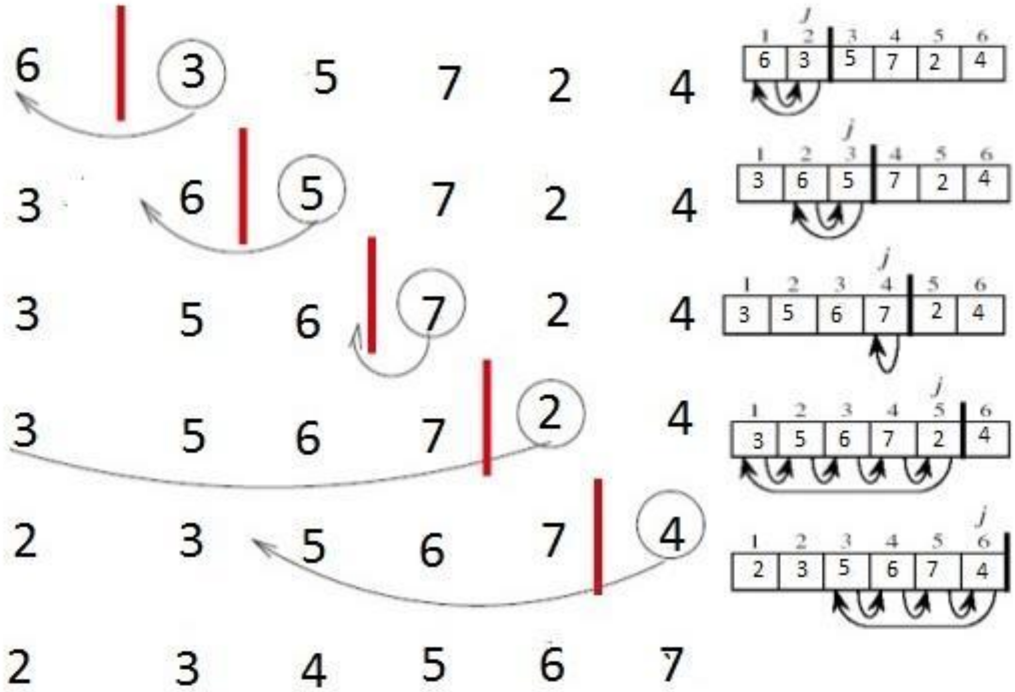
- شبه‌کد شبیه به C ، C++، Java، Python، JavaScript و بسیاری از زبان‌های برنامه‌نویسی پرکاربرد دیگر است. اگر شما هیچ کدام از این زبانها را نمی دانید مشکلی نیست چون شبه کد را متوجه خواهید شد.
- شـــبه‌کد برای بیان الگوریتم‌ها به انســـان‌ها طراحی شـــده اســـت. بنابراین مهندســـان نرم افزار از خطاهای نحوی زبانهای برنامه‌سازی موجود در شبه کد صرف نظر می‌کنند
- ما گاهی اوقات جملات انگلیسی را در شبه‌کد قرار می‌دهیم. بنابراین، برخلاف زبان‌های برنامه‌نویسی «واقعی»، نمی‌توانیم کامپایلری ایجاد کنیم که کد شبه را به کد ماشین ترجمه کند.

الگوریتم مرتب‌سازی درجی

الگوریتم خوبی برای مرتب‌سـازی تعداد کمی از عناصـر. این الگوریتم به همان روشـی کار می‌کند که ممکن اسـت یک دسـته کارت بازی را مرتب کنید:



- ابتدا دست چپ خالی است و کارت ها روی میز بازی به پشت قرار دارند و شما عدد آنها را نمی‌دانید.
- سپس یک کارت را از روی میز بردارید و آن را در موقعیت صحیح در دست چپ قرار دهید.
- برای پیدا کردن موقعیت صحیح یک کارت، آن را با هر یک از کارت‌های موجود در دست، از راست به چپ مقایسه کنید.



شبه‌کد

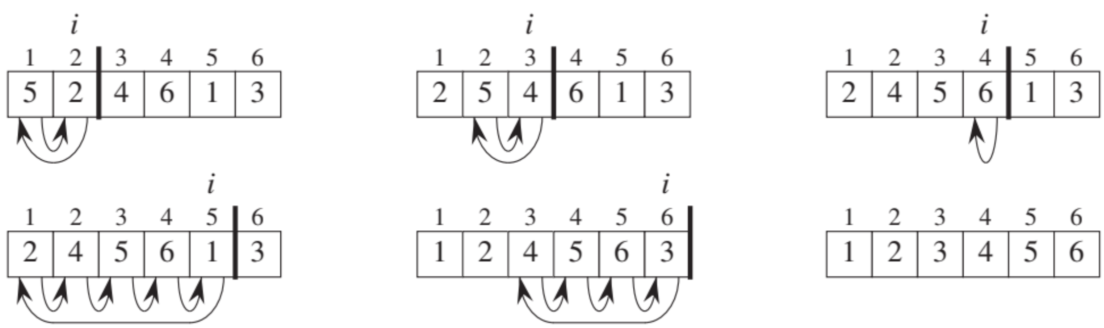
ما از رویه INSERTION-SORT استفاده می‌کنیم.

- این رویه یک آرایه A[1:n] و طول n آرایه را به عنوان پارامتر می‌پذیرد.
- از علامت ":" برای نشـــان دادن محدوده یا زیرآرایه درون یک آرایه اســـتفاده می‌کنیم. نماد A[i:j] نشـــان‌دهنده j-i+1 عنصر آرایه از A[i] تا و شامل A[j] است. [توجه داشته باشید که معنای نماد زیرآرایه ما با معنای آن در پایتون متفاوت است. در پایتون، A[i:j] نشان‌دهنده j-i عنصر آرایه از A[i] تا A[j-1] است و A[j] را شامل نمی‌شود. علاوه بر این، در پایتون، اندیس‌های منفی از انتها شمارش می‌شوند. ما از اندیس‌های منفی استفاده نمی‌کنیم.]
- در این کتاب خانه شـروع آرایه را معمولا یک در نظر گرفته اسـت. البته ممکن اسـت خانه شـروع آرایه در برخی از موارد صفر در نظر گرفته شود که در توضیحات بیان می شود.
- آرایه A به صورت درجا (in-place) مرتب می‌شود: اعداد درون آرایه بازآرایی می‌شوند و در هر زمان حداکثر تعداد ثابتی از عناصر خارج از آرایه قرار می‌گیرند. اگر برای مرتب سازی از آرایه اصلی استفاده شود (آرایه ورودی)، به این روش مرتب سازی درجا گفته میشود.

```
INSERTION-SORT(A, n)
  for i = 2 to n
      key = A[i]
      // Insert A[i] into the sorted subarray A[1 : i − 1].
      j = i − 1
      while j > 0 and A[j] > key
          A[j + 1] = A[j]
          j = j − 1
      A[j + 1] = key
```

مثال

Example



[Read this figure row by row. Each part shows what happens for a particular iteration with the value of i indicated. i indexes the "current card" being inserted into the hand. Elements to the left of $A[i]$ that are greater than $A[i]$ move one position to the right, and $A[i]$ moves into the evacuated position. The heavy vertical lines separate the part of the array in which an iteration works—$A[1:i]$—from the part of the array that is unaffected by this iteration—$A[i+1:n]$. The last part of the figure shows the final sorted array.]

اثبات درستی الگوریتم

ما معمولا از مفهوم loop invariant برای بررسی و نشان دادن درستی الگوریتم استفاده می‌کنیم.

مفهوم **loop invariant** به معنای مستقل از حلقه است و به عبارت یا هر شرطی است که در ابتدای هر چرخش (iteration) مقدارش درست یا ثابت باشد، گفته می‌شود. به عبارت دیگر قسمتی از برنامه که در هر تکرار حلقه دارای مقدار درست باشد.

برای استفاده از یک ثابت حلقه برای اثبات درستی، باید سه نکته را در مورد آن نشان دهیم:

- **مقداردهی اولیه**: قبل از اولین تکرار حلقه درست است.
- نگهداری (**Maintenance**) : اگر قبل از یک تکرار حلقه درست باشد، قبل از تکرار بعدی نیز درست باقی می‌ماند.
- پایان (**Termination**) : حلقه خاتمه می‌یابد، loop invariant یک ویژگی مفید به ما می‌دهد که به نشـــان دادن صحت الگوریتم کمک می‌کند.

برای مثال ***loop invariant*** که در مرتب سازی درجی استفاده می شود می تواند بصورت زیر باشد:

در شبکه کد مرتب سازی درجی ، در ابتدای هر تکرار حلقه for خارجی که دارای شمارنده i است ، آرایه A[1:i-1] شامل i-1 عنصر اول آرایه A (آرایه ورودی) است بصورت مرتب شده می‌باشد.

شباهت با استقرای ریاضی

استفاده از نابرابری‌های حلقه شباهت زیادی به استقرای ریاضی دارد:

- **برای اثبات یک ویژگی، باید حالت پایه و گام استقرا را اثبات کنید.**
- **نشان دادن برقراری نابرابری قبل از اولین تکرار، مشابه اثبات حالت پایه است.**
- **نشان دادن حفظ نابرابری در تکرارها، مشابه گام استقرا است.**
- **تفاوت اصلی در بخش پایان است، زیرا در استقرای ریاضی معمولاً گام استقرا به صورت نامتناهی استفاده می‌شود، اما در اینجا "استقرا" با پایان حلقه متوقف می‌شود.**
- **می‌توانیم این سه بخش را به هر ترتیبی که مناسب باشد نشان دهیم.**

بررسی درستی مرتب سازی درجی به کمکloop invariant

در شبکه کد مرتب سازی درجی ، در ابتدای هر تکرار حلقه for خارجی که دارای شمارنده i است ، آرایه A[1:i-1] شامل i-1 عنصر اول آرایه A (آرایه ورودی) است بصورت مرتب شده می‌باشد.

مقداردهی اولیه

درست قبل از اولین تکرار، i=2. زیرآرایه A[1:1] فقط شامل عنصر منفرد A[1] است که همان عنصر اولیه در A[1] بوده و به صورت بدیهی مرتب شده است.

حفظ

برای دقت بیشتر، باید یک نابرابری حلقه برای حلقه while داخلی نیز تعریف و اثبات کنیم. اما به جای پیچیده کردن موضوع، توجه می‌کنیم که بدنه حلقه داخلی با جابجایی A[i-1], A[i-2], A[i-3] و ... به اندازه یک موقعیت به راست کار می‌کند تا موقعیت مناسب برای key (که مقدار اولیه آن در A[i] بوده) پیدا شود. در این نقطه، مقدار key در موقعیت صحیح قرار می‌گیرد.

پایان

حلقه for بیرونی با i=2 شروع می‌شود. هر تکرار، i را یک واحد افزایش می‌دهد. حلقه وقتی پایان می‌یابد که i > n، که در i=n+1 اتفاق می‌افتد. بنابراین حلقه پایان یافته و در آن زمان i-1=n است. با جایگزینی n به جای i-1 در نابرابری حلقه، زیرآرایه A[1:n] شامل عناصر اولیه A[1:n] بوده اما به ترتیب مرتب شده است. به عبارت دیگر، کل آرایه مرتب شده است.

تحلیل الگوریتم ها

ما می خواهیم منابع مورد نیاز الگوریتم که بطور معمول ***مدت زمان لازم برای اجرا شدن*** است، را پیش بینی کنیم

عوامل موثر بر مدت زمان اجرای یک برنامه:

- سرعت پردازنده
- نوع کامپایلر
- اندازه ورودی
- ترکیب یا ساختار ورودی
- پیچیدگی الگوریتم

چرا نیاز داریم الگوریتم ها را تحلیل کنیم؟چرا فقط الگو ریتم را پیاده سازی نکنیم، سپس کد را اجرا کنیم و زمان آن را اندازه بگیریم؟

پاسخ: زیرا در این صورت:

- فقط مدت زمان اجرا را روی یک کامپیوتر خاص خواهیم داشت
- با ورودی خاص

- مرتب سازی ۱۰۰۰ عدد بیشتر از مرتب سازی ۳ عدد طول می کشد.
- یک الگوریتم مرتب‌سازی معین ممکن است حتی در دو ورودی با اندازه یکسان زمان‌های متفاوتی را ببرد. برای مثال، خواهیم دید که مرتب‌سازی درجی زمان کمتری برای مرتب‌سازی n عنصر زمانی که از قبل مرتب شده‌اند نسبت به زمانی که به ترتیب معکوس مرتب شده‌اند، طول می‌کشد.

- با پیاده سازی خاص
- با کامپایلر یا مفسر خاص
- با کتابخانه های خاص

شما نمی توانید پیش بینی کنید که همان کد روی کامپیوتر متفاوت،با ورودی متفاوت، در زبان برنامه نویسی متفاوت و غیره چقدر زمان می برد

چگونه زمان اجرای یک الگوریتم را تجزیه و تحلیل کنیم

بطور معمول زمان اجرای یک الگوریتم را با توجه به اندازه ورودی مسئله n بدست خواهیم آورد و به صورت یک تابع مانند $T(n)$ بیان خواهیم کرد

اندازه ورودی در مسائل مختلف متفاوت است:

- در یک مسئله جستجو و یا مرتب سازی می تواند طول آرایه ورودی باشد
- در یک مسئله گراف می تواند تعداد راس ها و یال ها باشد
- در مسئله ضرب دو عدد می تواند تعداد بیت های دو عدد باشد

در محاسبه زمان اجرا موارد زیر را در نظر خواهیم گرفت:

- بطور معمول دستورات را مستقل از ماشین در نظر می گیریم
- فرض می کنیم هر خط از شبه کد زمان ثابتی دارد
- ممکن است زمان اجرای یک خط با خط دیگر متفاوت باشد، اما هر بار اجرای خط k زمان یکسانی معادل c_k نیاز دارد
- اگر خط شامل فراخوانی یک زیرروال باشد، خود فراخوانی زمان ثابتی می برد، اما اجرای زیرروال فراخوان شده ممکن است زمان ثابتی نداشته باشد.
- زمان اجرای یک برنامه را می توان به صورت تابعی از مشخصه های مهم ورودی بیان کرد $T(n)$

- کوچکترین واحد زمانی برای اجرای الگوریتم ها را یک **Step** یا **گام محاسباتی** می گوییم.

تعیین گام های یک برنامه:

- توضیحات (comments): توضیحات دستورات غیر قابل اجرا هستند و تعداد گام آنها صفر است.
- دستورهای تعریفی (Declarative Statement): شامل همه عباراتی که متغیرها و ثابت ها را مشخص می کند؛ مانند: int، char، ... . تعداد گام این عبارات نیز صفر است چون اجرایی نیستند.
- عبارات و دستورات انتساب (Expression and Assignment statement): بیشتر این عبارات تک گام هستند به استثنای عباراتی که شامل فراخوانی توابع هستند که نیاز به تعیین هزینه فراخوانی تابع داریم.

- دستورات تکرار (Iteration Statement): شامل دستورات for، while، do است. شمارش گام این دستورات را فقط برای بخش کنترل آنها در نظر می گیریم.

- دستور if-else: این دستور شامل سه قسمت می باشد:

```
If (<expr>)
        <statement 1> ;
Else
        <statement 2> ;
```

- عبارت نام تابع: تعداد گام این دستور برابر صفر است
- عبارت return: تعداد گام این دستور برابر ۱ است.

- با توجه به مطالب فوق می توانیم تعداد گام مورد نیاز یک برنامه برای حل یک مساله خاص را تعیین کنیم. برای تعیین تعداد گام یک برنامه جدولی ایجاد می کنیم و تعداد گام هر دستور را یادداشت می کنیم و سپس تعداد دفعات اجرای هر دستور را تعیین می کنیم. با ادغام این دو کمیت تعداد کل گام بدست می آید.
- مثال

```
1  int sum(int a[], int n)
2  {
3  int s = 0;
4  For (int i=0 ; i<n; i++)
5          s+= a[i];
6  return s;
7  }
```

	گام اجرا	تعداد دفعات تکرار	کل مراحل
1			
2			
3			
4			
5			
6			
7			

T(n) = 2n+3

- مثال

```
1  int add(int a[][], int b[][], int c[][])
2  {
3  for(int i=0 ; i<m; i++)
4          for(int i=0 ; i<n; i++)
5                  c[i][j] =a[i][j]+b[i][j] ;
6  }
```

	گام اجرا	تعداد دفعات تکرار	کل مراحل
1			
2			
3			
4			
5			
6			

T(n) = 2mn+2m+1

```
1   int factorial(int n)
2   {
3   int fact = 1;
4   for(int i=2 ; i<=n; i++)
5                   fact*=i;
6   return fact;
7   }
```

	گام اجرا	تعداد دفعات تکرار	کل مراحل
1			
2			
3			
4			
5			
6			
7			

T(n) = 2n+1

```
1   void fibo(int n)
2   {
3   If (n<=1)
4           Cout << n
5   else{
6           int fn;
7           Int fnm2 = 0;
8           Int fnm1  = 1 ;
9           For (int i=2; i<=n ; i++)
10          {
11                  Fn= fnm1 + fnm2;
12                  Fnm2 = fnm1;
13                  Fnm1 = fn;
14          }
15          Cout << fn;
16          }
17  }
```

	گام اجرا	تعداد دفعات تکرار		کل مراحل	
		n<=1	n>1	n<=1	n>1
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16,17					

$$T(n) = \begin{cases} 2 & n \le 1 \\ 4n+1 & n > 1 \end{cases}$$

تحلیل مرتب سازی درجی

- فرض کنید خط kام زمان c_k می‌برد، که یک ثابت است.
- برای $i = 2, 3, ..., n$، فرض کنید t_i تعداد دفعاتی است که تست حلقه while برای آن مقدار i اجرا می‌شود.
- توجه داشته باشید که وقتی یک حلقه for یا while به روش معمول خارج می شود - به دلیل آزمایش در سرآیند حلقه - تست یک بار بیشتر از بدنه حلقه اجرا می شود.

زمان اجرای الگوریتم:

$$\sum_{\text{همه دستورات}} (\text{تعداد دفعات اجرای دستور}) \cdot (\text{هزینه دستور})$$

فرض کنید T(n) زمان اجرای Insertion-Sort باشد.

INSERTION-SORT(A, n)	cost	times
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$j = i-1$	c_4	$n-1$
while $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i-1)$
$j = j-1$	c_7	$\sum_{i=2}^{n}(t_i-1)$
$A[j+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n}(t_i-1) + c_7 \sum_{i=2}^{n}(t_i-1) + c_8(n-1)$$

بهترین حالت مرتب سازی درجی

آرایه از قبل مرتب شده است. در این صورت خطهای ۶ و ۷ اجرا نخواهد شد. همچنین شرط A[j]>key همیشه نادرست است، بنابراین $t_i=1$ خواهد بود. به عبارت دیگر هزینه خط ۵ برابر $C_5 n$ است.

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

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

می‌توان $T(n)$ را به صورت $an + b$ برای ثابت‌های a و b بیان کرد $T(n)$ یک تابع خطی از n است.

بدترین حالت مرتب سازی درجی

زمانی اتفاق می‌افتد که آرایه بصورت معکوس مرتب باشد

- همیشه در تست حلقه while شرط A[i] > key برقرار است
- نیاز به مقایسه key با تمام عناصر سمت چپ موقعیت iام داریم <= مقایسه با i-1 عنصر
- از آنجا که حلقه while وقتی خاتمه می‌یابد که i به 0 برسد، یک تست اضافی بعد از i-1 تست انجام می‌شود <= $t_i = i$

$$\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} i \text{ و } \sum_{i=2}^{n} (t_i - 1) = \sum_{i=2}^{n} (i - 1)$$

$$\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i \right) - 1 = \frac{n(n+1)}{2} - 1$$

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

- $\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} i$ and $\sum_{i=2}^{n} (t_i - 1) = \sum_{i=2}^{n} (i-1)$.
- $\sum_{i=1}^{n} i$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\frac{n(n+1)}{2}$.
- Since $\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.
  [The parentheses around the summation are not strictly necessary. They are there for clarity, but it might be a good idea to remind the students that the meaning of the expression would be the same even without the parentheses.]
- Letting $l = i - 1$, we see that $\sum_{i=2}^{n} (i-1) = \sum_{l=1}^{n-1} l = \frac{n(n-1)}{2}$.
- Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of n.

تحلیل بدترین حالت و حالت متوسط

بطور معمول تلاش می شود زمان اجرا در بدترین حالت محاسبه شود. زیرا:

- زمان اجرای بدترین حالت یک حد بالایی تضمین شده برای زمان اجرای هر ورودی ارائه می‌دهد.
- برای برخی الگوریتم‌ها، بدترین حالت اغلب اتفاق می‌افتد. مثلاً در جستجو، بدترین حالت معمولاً وقتی رخ می‌دهد که آیتم مورد جستجو وجود نداشته باشد.

مثال: فرض کنید n عدد را به صـــورت تصـــادفی به عنوان ورودی برای مرتب‌ســـازی درجی انتخاب کنیم. به طور میانگین، کلید در$A[i]$ از نصف عناصر در $A[1{:}\, i-1]$کوچکتر و از نصف دیگر بزرگتر است.

به طور میانگین، حلقه while باید حدود نیمی از زیرآرایه مرتب شـــده$A[1{:}\, i-1]$ را بررســـی کند تا محل قرارگیری کلید را مشخص کند. $t_i \approx (i/2)$.

اگرچه زمان اجرای حالت میانگین تقریباً نصف بدترین حالت است، اما همچنان یک تابع درجه دوم از n محسوب می‌شود.

مرتبه رشد یک الگوریتم

برای آسانتر کردن آنالیز و تمرکز بر نکات مهمتر، تنها جملات و قسمت‌های مهمتر فرمول (تابع پیچیدگی) توجه خواهیم کرد:

- از جملات مرتبه پایین صرف نظر می‌کنیم
- از مقادیر و ضرایب ثابت صرف نظر خواهیم کرد

مثال: برای مرتب‌سازی درج، ما قبلاً هزینه‌های را انتزاع کرده‌ایم تا نتیجه بگیریم که بدترین زمان اجرا $an^2 + bn + c$ است.

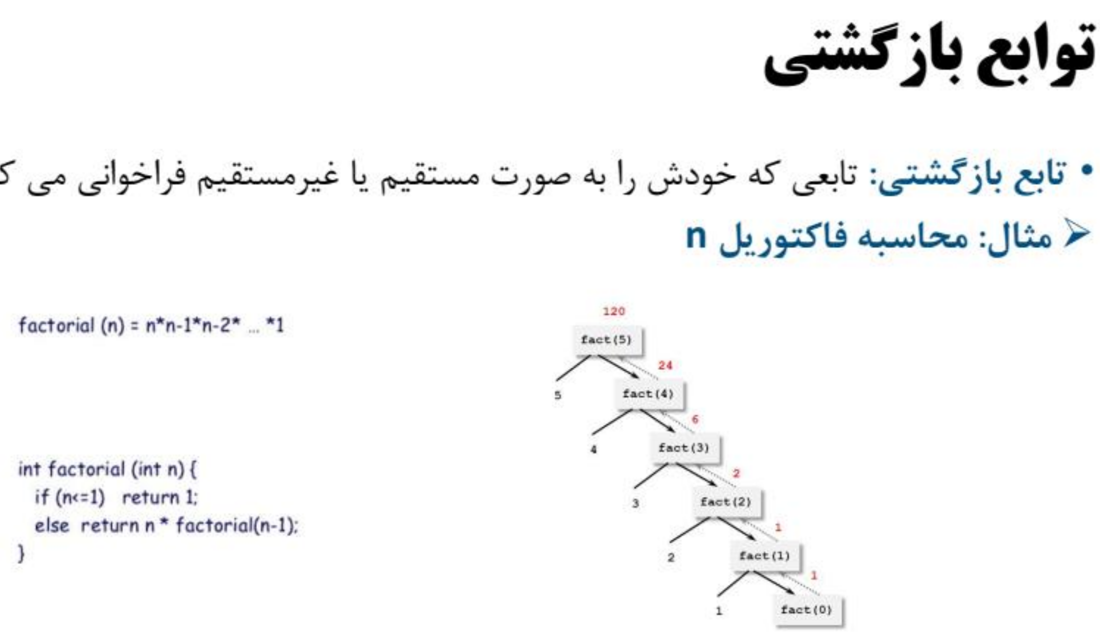
- جمله مرتبه پایین تر را حذف کنید در نتیجه خواهیم داشت an^2.
- نادیده گرفتن ضریب ثابت پس n^2. اما نمی توانیم بگوییم که در بدترین حالت زمان اجرای $T(n)$ برابر n^2 است.

و می‌گوییم مانند n^2 رشد می کند. اما برابر n^2 نیست. یعنی نمی‌توانیم مقدار دقیقی برای زمان بدست آوریم.

ما می گوییم که زمان اجرا $\Theta(n^2)$ است تا این مفهوم را به دست آوریم و نشان دهیم که ترتیب یا مرتبه رشد n^2 است.

ما معمولاً یک الگوریتم را کارآمدتر از الگوریتم دیگری در نظر می گیریم که در بدترین حالت زمان اجرای آن مرتبه یا ترتیب رشد کمتری داشته باشد.

یادآوری توابع بازگشتی

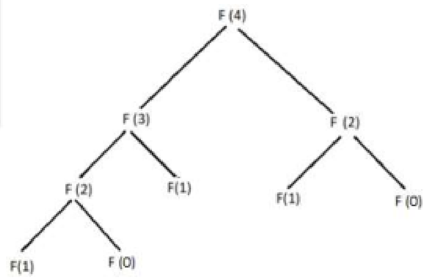


توابع بازگشتی

➢ مثال: محاسبه فیبوناچی

0,1,1,2,3,5,8,13 ,21 ,34 ,...

```
public static long F(int n)
{
   if (n == 0) return 0;
   if (n == 1) return 1;
   return F(n - 1) + F(n - 2);
}
```



تحلیل توابع بازگشتی

مثال : بدست آوردن حاصل جمع عناصر یک آرایه بصورت بازگشتی

```
float Rsum(float* a, const int n)
{
    if (n <= 0) return 0;
    else return (Rsum(a, n - 1) + a[n - 1]);
}
```

$$T_{Rsum}(n) = 2 + T_{Rsum}(n-1)$$

$$\begin{aligned} T_{Rsum}(n) &= 2 + T_{Rsum}(n-1) \\ &= 2+2+T_{Rsum}(n-2) \\ &= 2*2+T_{Rsum}(n-2) \\ &= \\ &= 2*i+T_{Rsum}(n-i) \\ &= 2*n+T_{Rsum}(0) \end{aligned}$$

فراخوانی بازگشـــتی تا زمانی ادامه پیدا می کند که مقدار ورودی یعنی n برابر صـــفر شـــود. به عبارت دیگر n-i=0 شود، یعنی n=i باشد بنابراین در جمله آخر n=i است

=2*n+2

.

مثال:

$$
\begin{aligned}
T(n) &= T(n-1) + n \quad T(1)=1\\
&= T(n-2) + (n-1) + n\\
&= T(n-3) + (n-2) + (n-1) + n\\
&.\\
&.\\
&.\\
&= T(n-i) + (n-(i-1)) + \dots + n \rightarrow n-i = 1 \rightarrow i = n-1\\
&= T(n-n+1) + (n-(n-1-1)) + \dots + n\\
&= T(1) + 2 + \dots + n\\
&= 1+2+\dots+n\\
&= \frac{n(n+1)}{2} \rightarrow \theta(n^2)
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= T(\tfrac{n}{2}) + 1 \quad T(1) = 1\\
&= T(\tfrac{n}{4}) + 1 + 1\\
&= T(\tfrac{n}{8}) + 1 + 1 + 1\\
&.\\
&.\\
&= T(\tfrac{n}{2^i}) + i \rightarrow \frac{n}{2^i} = 1 \rightarrow n = 2^i \rightarrow log_2 n = i\\
&= T(1) + \log_2 n\\
&\rightarrow \theta(\log_2 n)
\end{aligned}
$$

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

$$T(\tfrac{n}{2}) = T(\tfrac{n}{4}) + 1$$

$$T(\tfrac{n}{4}) = T(\tfrac{n}{8}) + 1$$

طراحی یک الگوریتم

روش‌های بسیاری برای طراحی الگوریتم وجود دارد. برای مثال مرتب سازی درجی را که مشاهده کردید یک روش افزایشی است. زیر آرایه A[1:i-1] مرتب است و عنصر A[i] را در جای مرتب خود قرار خواهیم داد.

تقسیم و حل

یکی دیگر از رهیافت‌های رایج برای حل مسئله، روش تقسیم و حل (**Divide and conquer**) است.

- **تقسیم**: مسئله را به چند زیرمسئله تقسیم کنید که نمونه‌های کوچکتری از همان مسئله هستند
- **حل**: زیرمسئله‌ها را با حل بازگشتی آن‌ها حل کنید.
  - حالت پایه: اگر زیرمسئله‌ها به اندازه کافی کوچک باشند، که بتوان بطور مستقیم و به آسانی آن را حل کرد.
- **ترکیب**: راه‌حل‌های زیرمسئله‌ها را برای بدست آوردن راه‌حل مسئله اصلی، ترکیب کنید

مرتب سازی ادغامی

یک الگوریتم مرتب سازی مبتنی بر تقسیم و حل است که زمان اجرای آن در بدترین حالت مرتبه پایین‌تری نسبت به مرتب سازی درجی دارد.

از آنجا که با زیرمسئله‌ها سروکار داریم، هر زیرمسئله را به صورت مرتب کردن یک زیرآرایه A[p:r] بیان می‌کنیم. در ابتدا p=1 و r=n است، اما این مقادیر با پیشرفت بازگشتی در زیرمسئله‌ها تغییر می‌کنند.

می‌خواهیم آرایه A[p:r] را با استفاده از این سه گام مرتب کنیم:

- **تقسیم**: با تقسیم به دو زیرآرایه A[p:q] و A[q+1:r]، که در آن q نقطه میانی A[p:r] است.
- **حل**: با مرتب کردن بازگشتی دو زیرآرایه A[p:q] و A[q+1:r].
- **ترکیب**: با ادغام دو زیرآرایه مرتب شده A[p:q] و A[q+1:r] برای تولید یک زیرآرایه مرتب شده A[p:r]. برای این مرحله، یک روال MERGE(A, p, q, r) تعریف می‌کنیم.

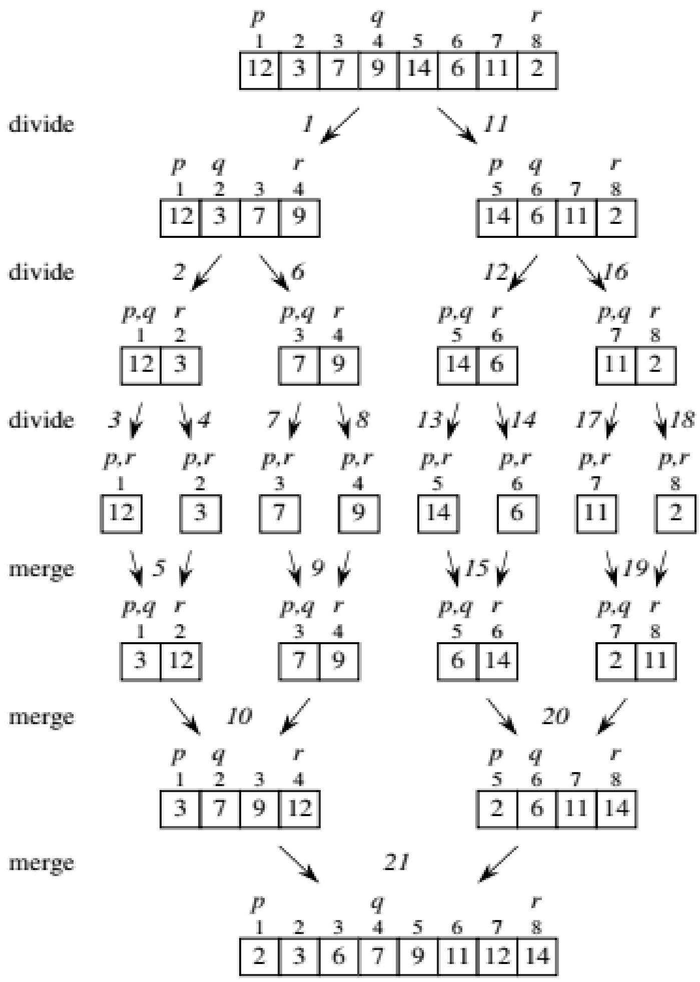
حالت پایه: بازگشت زمانی پایان می‌یابد که زیرآرایه فقط یک عنصر داشته باشد که در این صورت به صورت بدیهی مرتب شده است.

Algorithm ۲ الگوریتم مرتب‌سازی ادغامی

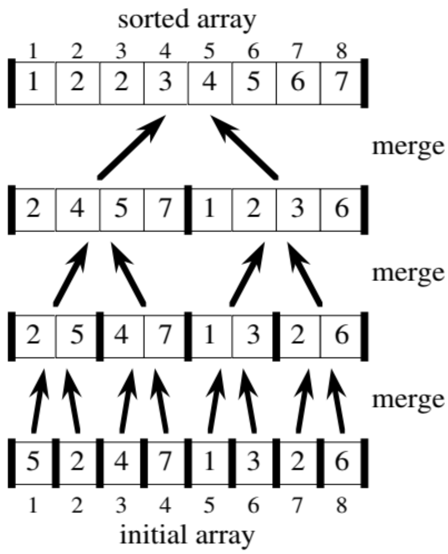
```
procedure Merge-Sort(A, p, r)
if p ≥ r then
    return
end if
q = ⌊(p + r)/2⌋
Merge-Sort(A, p, q)
Merge-Sort(A, q + 1, r)
Merge(A, p, q, r)
end procedure
```

Example

MERGE-SORT on an array with $n = 8$: *[Indices p, q, r appear above their values. Numbers in italics indicate the order of calls of* MERGE *and* MERGE-SORT *after the initial call* MERGE-SORT$(A, 1, 8)$.*]*

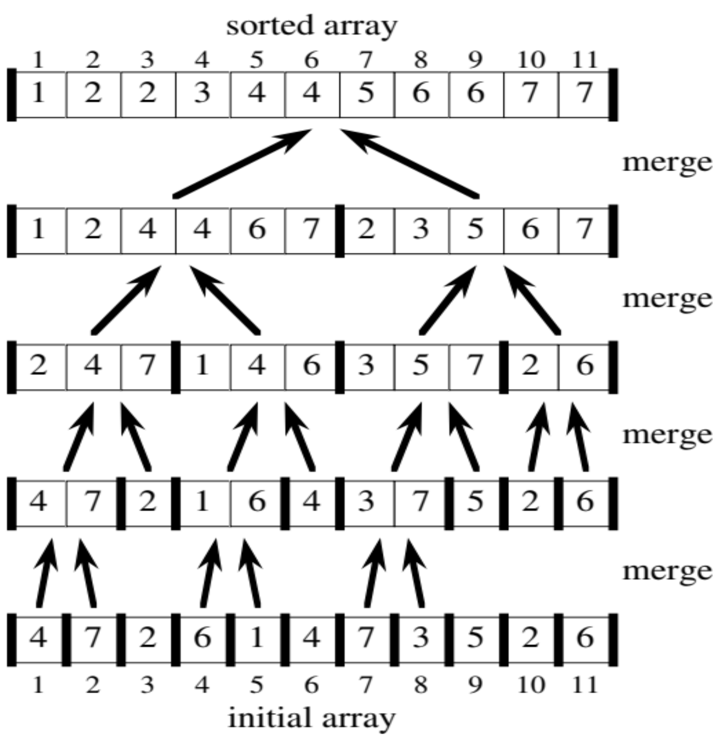


مثال : نمای پایین به بالا برای $n = 8$: [خطوط پررنگ، زیرآرایه‌های مورد استفاده در زیرمسئله‌ها را مشخص می‌کنند. دو شکل زیر را از پایین به بالا مرور کنید.]



مثال‌هایی که n توانی از ۲ است، بسیار سرراست هستند، اما دانش‌آموزان ممکن است مثالی هم بخواهند که n توانی از ۲ نباشد.

Bottom-up view for $n = 11$:



در اینجا، در سطح ماقبل آخر بازگشت، برخی از زیرمسئله‌ها فقط یک عنصر دارند. بازگشت در این زیرمسئله‌های تک عنصری به پایین‌ترین حد خود می‌رسد.

ما تابع Merge-Sort را پیاده سازی کردیم و حالا نوبت پیاده سازی تابع Merge است.

ورودی تابع Merge شامل آرایه A و اندیس‌های p,q,r است بطوریکه:

- $p \leq q < r$
- زیرآرایه A[p:q] مرتب شده و زیرآرایه A[q+1:r] مرتب شده است

خروجی : دو زیرآرایه در یک زیرآرایه مرتب شده در A[p:r] ادغام می‌شوند

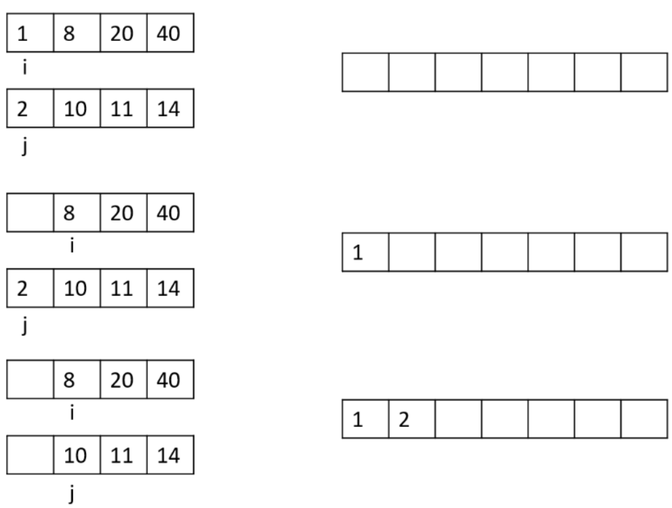
این روش در زمان $\Theta(n)$ اجرا می‌شود، که در آن $n = r - p + 1$ تعداد عناصر در حال ادغام است

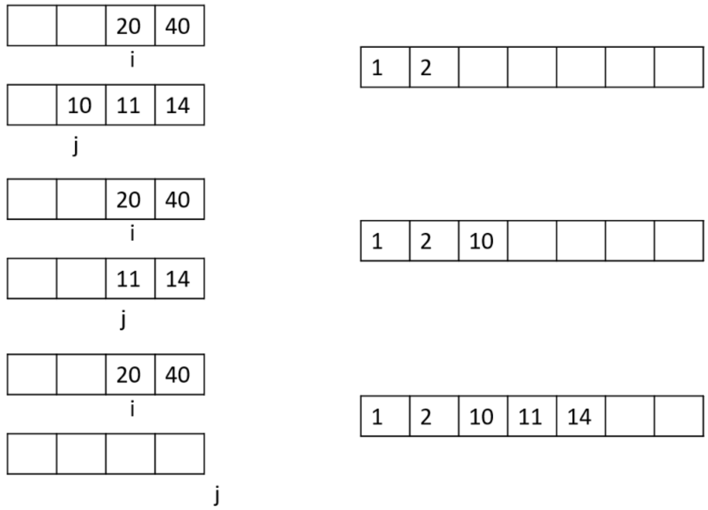
نکته: تا به حال، n به معنای اندازه مسئله اصلی بوده است. اما اکنون ما از آن به عنوان اندازه یک مشکل فرعی استفاده می‌کنیم. زمانی که الگوریتم های بازگشتی را تحلیل می کنیم از این تکنیک استفاده خواهیم کرد. اگرچه ممکن است اندازه مسئله اصلی را با n نشان دهیم، به طور کلی n اندازه یک زیرمسئله معین خواهد بود

ادغام دو زیر آرایه مرتب

فرض کنید دو زیرآرایه مرتب شده بصورت صعودی را می‌خواهید در هم ادغام کنید. با انجام مراحل پایه زیر می‌توانید این کار را انجام دهید:

- انتخاب کوچکترین عدد از بین دو عدد ابتدای زیرآرایه‌ها
- انتخاب آن عدد برای انتقال به زیرآرایه جدید و افزایش شاخص زیرآرایه
- قرار دادن عدد انتخاب شده در اولین جای خالی زیرآرایه جدید





- مراحل پایه را تا خالی شدن یکی از زیرآرایه‌ها تکرار کنید.
- پس از خالی شدن یک زیرآرایه، باقیمانده زیرآرایه دیگر را به زیرآرایه خروجی اضافه کنید.
- هر مرحله پایه زمان ثابتی می‌برد، زیرا فقط مقایسه دو عدد است.
- در کل n مرحله پایه وجود دارد، زیرا هر مرحله یک عدد از دسته‌های ورودی برمی‌دارد.
- بنابراین این روش در زمان $\Theta(n)$ اجرا می‌شود.

جزئیات بیشتر درباره روال ادغام

روال MERGE دو زیرآرایه A[p:q] و A[q+1:r] را در آرایه‌های موقت L (چپ) و R (راست) کپی کرده، سپس مقادیر را در A[p:r] ادغام می‌کند. در این تابع :

- ابتدا طول‌های n_L و n_R را برای زیرآرایه‌ها محاسبه می‌کند.
- آرایه‌های $L[0:n_L - 1]$ و $R[0:n_R - 1]$ را ایجاد می‌کند.
- با دو حلقه for، زیرآرایه‌ها را در L و R کپی می‌کند.
- حلقه while اول کوچکترین مقدار را در L و R یافته و در A کپی می‌کند.
- اندیس k موقعیت در حال پر شدن در A را نشان می‌دهد
- اندیس‌های i و j موقعیت‌های کوچکترین مقادیر باقیمانده در L و R را نشان می‌دهند

Pseudocode

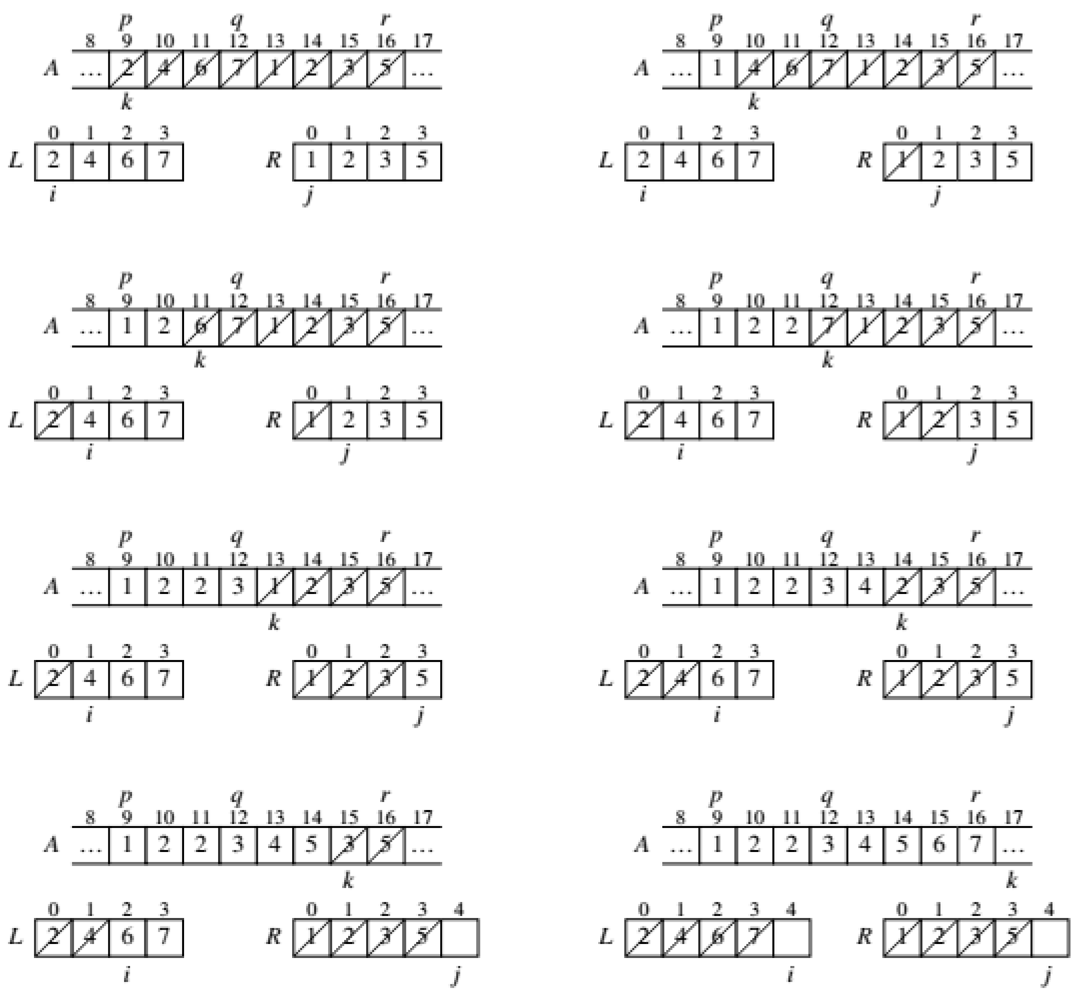
```
MERGE(A, p, q, r)
  n_L = q - p + 1          // length of A[p : q]
  n_R = r - q              // length of A[q + 1 : r]
  let L[0 : n_L - 1] and R[0 : n_R - 1] be new arrays
  for i = 0 to n_L - 1     // copy A[p : q] into L[0 : n_L - 1]
      L[i] = A[p + i]
  for j = 0 to n_R - 1     // copy A[q + 1 : r] into R[0 : n_R - 1]
      R[j] = A[q + j + 1]
  i = 0                    // i indexes the smallest remaining element in L
  j = 0                    // j indexes the smallest remaining element in R
  k = p                    // k indexes the location in A to fill
  // As long as each of the arrays L and R contains an unmerged element,
  //       copy the smallest unmerged element back into A[p : r].
 while i < n_L and j < n_R
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
      k = k + 1
 // Having gone through one of L and R entirely, copy the
 //       remainder of the other to the end of A[p : r].
 while i < n_L
      A[k] = L[i]
      i = i + 1
      k = k + 1
 while j < n_R
      A[k] = R[j]
      j = j + 1
      k = k + 1
```

زمان اجرا

دو حلقه اول for زمان $\Theta(n_L + n_R) = \Theta(n)$ می‌گیرند. هر یک از ســـه خط قبل و بعد از حلقه‌های for زمان ثابتی می‌گیرند

هر تکرار از ســه حلقه while دقیقاً یک مقدار از L یا R را در A کپی می‌کند، و هر مقدار دقیقاً یک بار در A کپی می‌شـــود. بنابراین، این سه حلقه در مجموع n تکرار دارند، که هر کدام زمان ثابتی می‌گیرند، برای زمان $\Theta(n)$

زمان اجرای کل: $\Theta(n)$



تحلیل الگوریتم‌های تقسیم و حل

برای توصیف زمان اجرای یک الگوریتم تقسیم و حل از recurrence equation (معادله بازگشتی) استفاده می‌کنیم

فرض کنید T(n) زمان اجرا برای مسئله‌ای با اندازه n باشد:

- اگر اندازه مسئله به اندازه کافی کوچک باشد (مثلاً $n \leq n_0$ برای مقداری ثابت n_0، حالت پایه داریم. راه‌حل مستقیم زمان ثابت $\Theta(۱)$ می‌برد.
- در غیر این صورت، مسئله را به a زیرمسئله تقسیم می‌کنیم، هر کدام به اندازه $۱/b$ اندازه اصلی. (در مرتب‌سازی ادغامی، a = b = 2)
- زمان تقسیم یک مسئله با اندازه n را D(n) می‌نامیم
- a زیرمسئله داریم که هر کدام اندازه n/b دارند ⇐ هر زیرمسئله $T(n/b)$ زمان می‌برد در کل aT(n/b) زمان برای حل زیرمسئله‌ها صرف می‌شود.
- زمان ترکیب راه‌حل‌ها را S(n) می‌نامیم

$$T(n) = \begin{cases} \Theta(1) & \text{اگر } n \leq n_0 \\ aT(n/b) + D(n) + S(n) & \text{در غیر این صورت} \end{cases}$$

تحلیل مرتب‌سازی ادغامی

برای سادگی فرض می‌کنیم n توانی از 2 باشد $\Leftarrow$ هر مرحله تقسیم دو زیرمسئله با اندازه دقیقاً $n/2$ تولید می‌کند.

حالت پایه وقتی رخ می‌دهد که n = 1. وقتی $n \geq 2$، زمان مراحل مرتب‌سازی ادغامی:

- **تقسیم**: محاسبه q به عنوان میانگین p و r $\Leftarrow$ D(n) = Θ(1)
- **حل**: حل بازگشتی 2 زیرمسئله، هر کدام به اندازه $n/2$ $2T(n/2)$
- **ترکیب**: MERGE روی یک زیرآرایه n عنصری $\Theta(n)$ زمان می‌برد $S(n) = \Theta(n)$

از آنجا که $D(n) = \Theta(1)$ و $S(n) = \Theta(n)$، در مجموع تابعی خطی بر حسب n به دست می‌آید: $\Theta(n)$ $\Leftarrow$ معادله بازگشتی زمان اجرای مرتب‌سازی ادغامی:

$$T(n) = \begin{cases} \Theta(1) & \text{اگر } n = 1 \\ 2T(n/2) + \Theta(n) & \text{اگر } n > 1 \end{cases}$$

حل معادله بازگشتی مرتب‌سازی ادغامی

با استفاده از قضیه اصلی در فصل ۴، می‌توان نشان داد که این رابطه بازگشتی راه‌حل $\mathrm{T(n)} = \Theta(n \lg n)$ دارد. [تذکر: $\lg n$ نمایانگر $\log_2 n$ است.]

در مقایسه با مرتب‌سازی درجی ($\Theta(n^2)$ در بدترین حالت)، مرتب‌سازی ادغامی سریع‌تر است. معامله یک فاکتور n با یک فاکتور $\lg n$ معامله خوبی است.

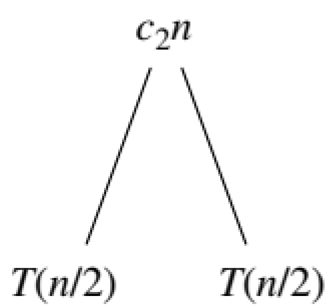
برای ورودی‌های کوچک، مرتب‌سازی درجی ممکن است سریع‌تر باشد. اما برای ورودی‌های به اندازه کافی بزرگ، مرتب‌سازی ادغامی همیشه سریع‌تر خواهد بود، زیرا زمان اجرای آن با نرخ کمتری نسبت به مرتب‌سازی درجی رشد می‌کند.

می‌توانیم بدون استفاده از قضیه اصلی نیز روش حل این رابطه بازگشتی را درک کنیم. اجازه دهید $c1$ ثابتی باشد که زمان اجرای حالت پایه را توصیف می کند و C_2 ثابتی برای زمان هر عنصر آرایه برای مراحل تقسیم و غلبه باشد.

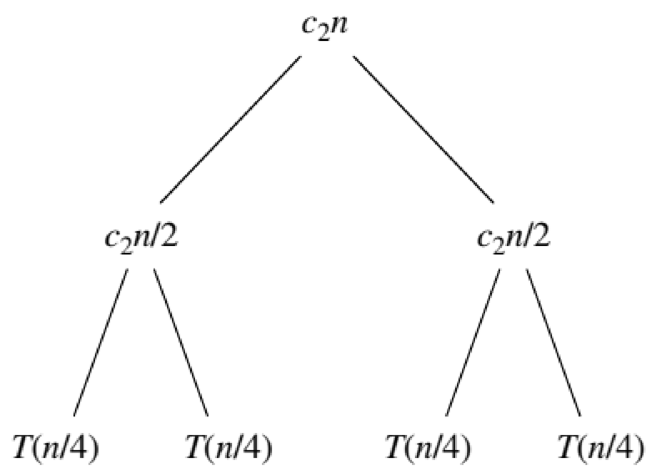
فرض کنید که حالت پایه برای $n = 1$ رخ می دهد، به طوری که $n_0 = 1$.

$$T(n) = \begin{cases} C1 & \text{اگر } n = 1 \\ 2T(n/2) + c_2 n & \text{اگر } n > 1 \end{cases}$$

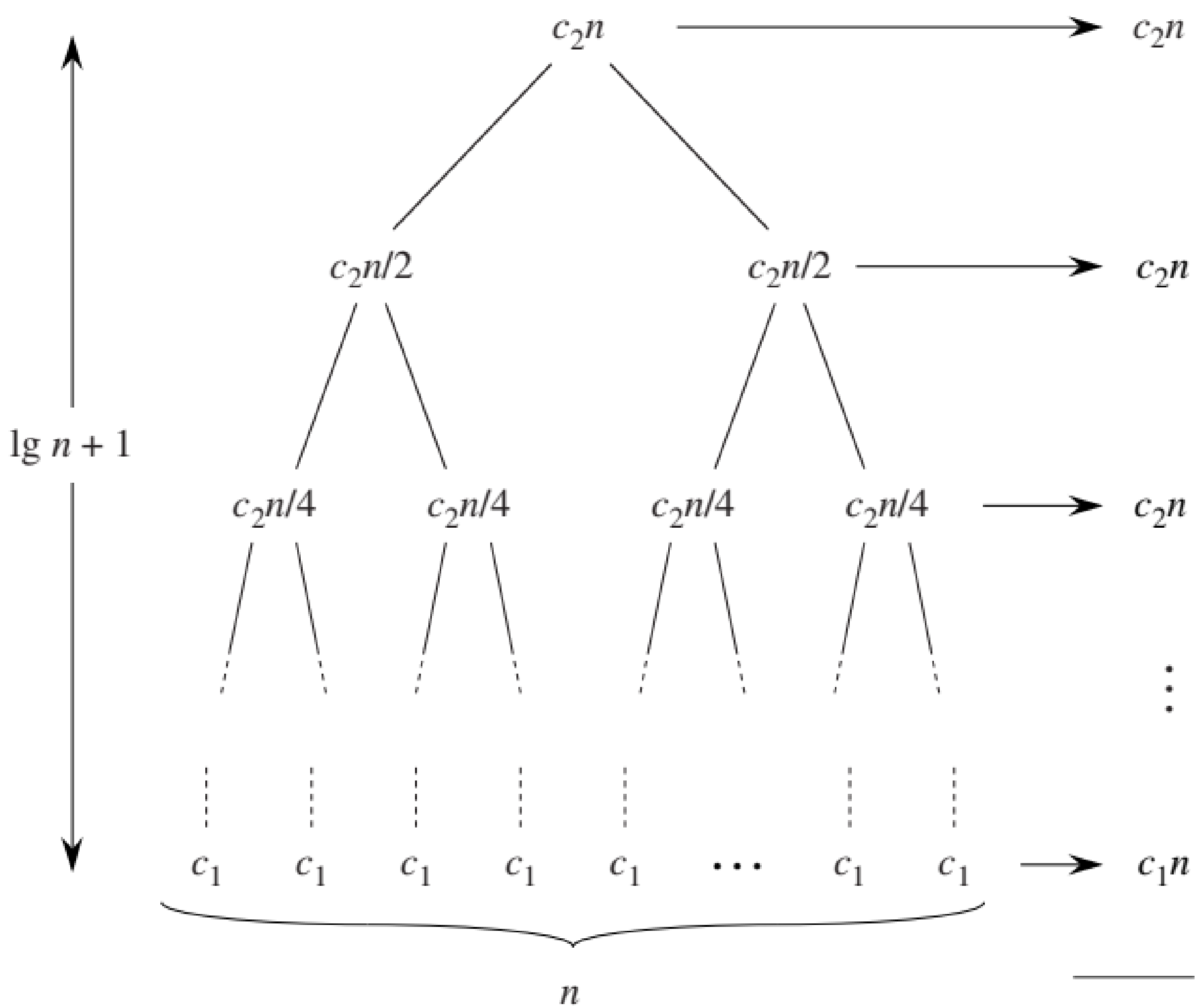
- یک درخت بازگشتی رسم کنید که بسط‌های متوالی بازگشت را نشان دهد.
- برای مسئله اصلی، هزینه c_2n را به علاوه دو زیر مسئله که هر کدام هزینه T(n/2) دارند، در نظر بگیرید.



- برای هر یک از زیرمسئله‌های با اندازه $n/2$، هزینه‌ای برابر با $c_2n/2$ به علاوه دو زیرمسئله که هر کدام هزینه‌ای برابر با $T(n/4)$ دارند، در نظر بگیرید.



- گسترش را ادامه دهید تا اندازه مسائل به ۱ برسد



Total: $c_2n \lg n + c_1n$

درخت بازگشت برای مرتب‌سازی ادغامی دارای ویژگی‌های زیر است:

- تعداد سطوح: $\lg n + 1$
- هزینه هر سطح (به جز پایین‌ترین): $c_2 n$
- هزینه پایین‌ترین سطح: $c_1 n$
- هزینه کل: $c_2 n \lg n + c_1 n$

۶.۴.۱ اثبات با استقرا

برای اثبات تعداد سطوح از استقرا استفاده می‌کنیم:

- حالت پایه: برای $n = 1$، تعداد سطوح $\lg 1 + 1 = 1$ است.
- فرض استقرا: برای $n = 2^i$، تعداد سطوح $i + 1$ است.
- گام استقرا: برای $n = 2^{i+1}$، تعداد سطوح $i + 2$ است.

نتیجه‌گیری

با صرفنظر از جمله مرتبه پایین $c_1 n$ و ضریب ثابت C_2، زمان اجرای مرتب‌سازی ادغامی $\Theta(n \lg n)$ است.

در مقایسه با مرتب‌سازی درجی $\left(\Theta(n^2)\right)$، مرتب‌سازی ادغامی برای ورودی‌های بزرگ‌تر کارایی بهتری دارد.