درس ساختمان داده‌ها
دانشکده ریاضی. دانشگاه صنعتی خواجه نصیرالدین طوسی زمستان ۹۸
تدریس توسط: حسین جوهری

تحلیل زمان اجرای الگوریتمها

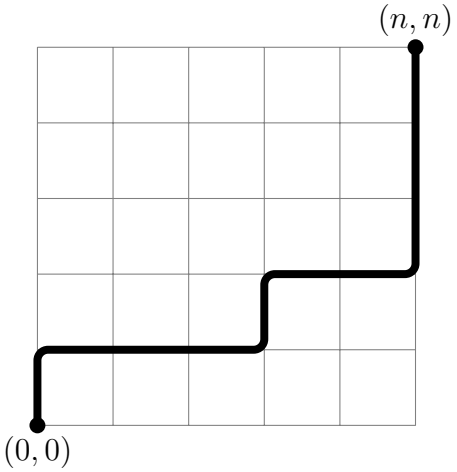
۱ مقدمه

بطور کلی تحلیل الگوریتمها به مسئله پیش بینی زمان اجرای الگوریتمها و منابعی که الگوریتم موقع اجرا مصرف می‌کند (مثل حافظه و ارتباطات شبکه‌ای) می‌پردازد. در این درس توجه خود را بر تحلیل زمان اجرای یک الگوریتم معطوف می‌کنیم.

زمان اجرای الگوریتم به عوامل مختلی وابسته است از جمله: سرعت پردازنده کامپیوتر، جزئیات پیاده‌سازی، کامپایلر مورد استفاده، حجم داده ورودی و پیچیدگی الگوریتم طراحی شده. در این میان دو عامل حجم داده ورودی و پیچیدگی الگوریتم بیشترین تاثیر را در زمان اجرا دارند. روشن است هر چه قدر حجم داده ورودی بیشتر باشد زمان اجرای الگوریتم هم بیشتر خواهد بود. برای مثال ما انتظار داریم که ضرب دو عدد ۱۰ رقمی به زمان بیشتری نسبت به ضرب دو عدد ٤ رقمی نیاز داشته باشد. به همین ترتیب، قاعدتا مرتب سازی یک آرایه حاوی ۱۰۰۰۰ عدد به زمان بیشتری نسبت به مرتب سازی یک آرایه حاوی ۵۰۰ عدد نیاز دارد. از طرفی دیگر پیچیدگی و ساختار الگوریتم مورد استفاده یک عامل مهم در تعیین زمان اجرای الگوریتم است. به مثال زیر توجه کنید:

فرض کنید در یک شبکه n در n که هر یال ارتباطی اش وزنی مثبت دارد می‌خواهیم طول کوتاهترین مسیر بین دو گره $(0,0)$ و (n,n) را پیدا کنیم. طول یک مسیر مجموع وزن یالهای آن مسیر است. فرض کنید صرفا علاقه‌مند به مسیرهایی هستیم که از پایین به بالا یا از چپ به راست ادامه پیدا کنند. یک نمونه مسیر در شکل زیر نشان داده شده است.

در یک الگوریتم برای این مسئله که اسمش را A میگذاریم، چنین راه حلی ارائه شده است. تمام مسیرهای ممکن بین $(0,0)$ و (n,n) در نظر گرفته می‌شود. فرض کنید مجموعه همه مسیرهای بین $(0,0)$ و (n,n) با علامت P نشان داده شود. الگورتیم هر بار مسیری پیمایش نشده از P را انتخاب می‌کند و طولش را محاسبه می‌کند. در صورتی که طول مسیر جدید از طول بهترین مسیر منتخب کمتر باشد، مسیر جدید به عنوان مسیر منتخب (تا به این لحظه) انتخاب می‌شود در غیر این صورت الگورتیم سراغ مسیر بعدی می‌رود. این کار ادامه می‌یابد تا اینکه همه مسیرهای P مورد بررسی قرار گیرند.

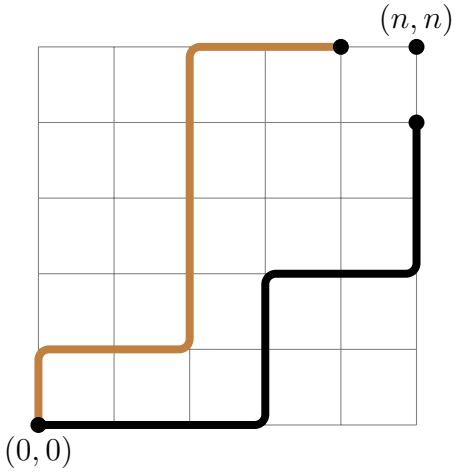


```
 Algorithm A:
-------------------------------------------------
let opt <-- infinity;
let P <-- all the paths between (0,0) and (n,n);
 while(there is more path in P)
   {
   p <-- next path in P
   if weight(p) < opt
      opt = weight(p)
   }
return opt
```

واضح است که زمان اجرای الگوریتم A بستگی به تعداد مسیرهای بین $(0,0)$ و (n,n) دارد که برابر با $\binom{2n}{n}$ است. از طرفی دیگر محاسبه هر بار تابع weight(p) به $2n-1$ عمل جمع نیاز دارد. علاوه بر جزئیات لازم برای محاسبه‌ی مسیرها، الگوریتم A در کل $\binom{2n}{n}$ عمل مقایسه و $n\binom{2n}{n}$ عمل جمع انجام می‌دهد.

خوشبختانه یک الگوریتم بسیار سریعتر برای این مسئله وجود دارد. این الگوریتم که اسمش را B می‌گذاریم بصورت استقرائی طول کوتاهترین مسیر را پیدا می‌کند. فرض کنید طول کوتاهترین مسیر از گره $(0,0)$ به گره (i,j) را با $opt[i,j]$ نشان دهیم. چون در هر صورت مسیر بهینه از $(0,0)$ به (n,n) باید یا از $(n-1,n)$ و یا $(n,n-1)$ بگذرد، با داشتن $opt[n-1,n]$ و $opt[n,n-1]$ می‌توانیم $opt[n,n]$ را بصورت زیر محاسبه کنیم.

$$opt[n,n] = \min\{opt[n-1,n] + w[(n-1,n),(n,n)],\\ opt[n,n-1] + w[(n,n-1),(n,n)]\}$$



در اینجا $w[(x,y),(x',y')]$ طول یال بین دو گره (x,y) و (x',y') می‌باشد. این الگوریتم در واقع ماتریس opt را بصورت سطری از پایین به بالا و از چپ به راست پر می‌کند. با محاسبه سطر iام ماتریس، می‌توان سطر $i+1$ ام را براحتی محاسبه کرد. محاسبه سطر و ستون صفرم ماتریس بدیهی است. این الگوریتم را می‌توان بصورت زیر پیاده‌سازی کرد.

```
 Algorithm B:
-----------------------------------------------
define matrix opt[n][n]
opt[0,0] = 0

for i=1 to n
   opt[0,i] = opt[0,i-1] + w[(0,i-1),(0,i)]    //fill the first row

for i=1 to n
   opt[i,0] = opt[i-1] + w[(i-1,0),(i,0)]      //fill the first column

for i =1 to n
    for j=1 to n
         opt[i,j] = min {opt[i,j-1]+w[(i,j-1),(i,j)],
                                          opt[i-1,j]+w[(i-1,j),(i,j)]}

return opt[n,n]
```

حال بیایید ببینیم الگوریتم B چند عمل محاسباتی در کل انجام می‌دهد. این الگورتیم از سه حلقه for تشکیل شده است. حلقه for اول n بار تکرار می‌شود پس n عمل جمع انجام می‌دهد. حلقه دوم نیز به همین شکل n عمل جمع انجام می‌دهد. حلقه سوم برای هر (i,j) مقدار $opt(i,j)$ را محاسبه می‌کند. پس در کل n^2 بار عمل محاسبه مینیمم انجام می‌شود. در نتیجه در اینجا $2n^2$ عمل جمع و n^2 مقایسه انجام می‌شود. با این توصیف الگوریتم B در کل $2n^2+2n$ عمل جمع و n^2 عمل مقایسه انجام می‌دهد.

ملاحظه می‌شود که الگوریتم B در مقایسه با الگوریتم A اعمال محاسباتی بمراتب کمتری را انجام می‌دهد. جدول زیر دو الگوریتم را از لحاظ زمانی مقایسه می‌کند. در اینجا هزینه اعمال جمع و مقایسه از لحاظ زمانی هر کدام یک میکروثانیه فرض شده است. $T(n)$ تعداد کل اعمال محاسباتی برای هر الگوریتم را بر حسب n نشان می‌دهد.

	$T(n)$	n=1	n=2	n=4	n=10	n=20
Algorithm A	$(n+1)\binom{2n}{n}$	4	18	350	2032316	2.8×10^{12}
Algorithm B	$3n^2+2n$	5	16	56	320	1240

توجه: برای سادگی کار، در این درس، و بطور معمول در تحلیل الگوریتمها، زمان اجرای اعمال محاسباتی اصلی مثل جمع و ضرب و مقایسه و انتصاب یکسان فرض می‌شود و یک واحد زمانی برای آن در نظر گرفته می‌شود.

چند تمرین

۱. دستور سوم در قطعه کد زیر چند بار اجرا می‌شود.

```
1.  for i in range(0,m):
2.    for j in range(0,n):
3.        count = count + 1
```

حل: mn

حلقه for دوم مستقل از شمارنده حلقه for اول اجرا می‌شود. هر بار دستور داخل حلقه for دوم n بار اجرا می‌شود. پس دستور سوم در کل nm بار اجرا می‌شود.

۲. دستور سوم در قطعه کد زیر چند بار اجرا می‌شود.

```
1.  for i in range(1,m+1):
2.    for j in range(1,i+1):
3.        count = count + 1
```

حل: $\sum_{i=1}^{m}\sum_{j=1}^{i} 1 = \sum_{i=1}^{m} i = m(m+1)/2$

در اینجا دستور سوم هر بار i دفعه اجرا می‌شود. چون i از 1 تا m تغییر می‌کند نتیجه بالا حاصل می‌شود.

۳. دستور چهارم در قطعه کد زیر چند بار اجرا می‌شود.

```
1. i = 1
2. while(i <= n):
3.     i = i * 2
4.     count = count + 1
```

حل: $\lfloor \log n \rfloor + 1$

در اینجا متغیر i هر بار دو برابر می‌شود. پس i تا کوچکترین توان ۲ بیشتر از n زیاد می‌شود. به عبارت دیگر دستور چهارم به تعداد توانهای ۲ کمتر یا مساوی n اجرا می‌شود. برای مثال وقتی $n = 5$ دستور چهارم سه بار اجرا می‌شود. $2^0, 2^1, 2^2$

۴. دستورات این برنامه در کل چند بار اجرا می‌شوند.

```
1. i = n
2. while(i => 1 ):
3.     i = i / 5
4.     count = count + 1
```

حل: $3(\lfloor \log_5 n \rfloor + 1) + 2$

در اینجا i هر بار تقسیم بر 5 می‌شود. مشابه تمرین بالا دستور چهارم در حلقه while به تعداد $\lfloor \log_5 n \rfloor + 1$ اجرا می‌شود. دستور دوم (دستور while) خود یک بار اضافه تکرار می‌شود (موقعی که شرط حلقه نقض می‌شود). پس دستورات ۲ تا ۴ در کل $3(\lfloor \log_5 n \rfloor + 1) + 1$ بار اجرا می‌شوند. دستور اول هم فقط یکبار اجرا می‌شود.

۵. خط چهارم در قطعه کد زیر چند بار اجرا می شود؟

```
1. for i in range(1,n):
2.   for j in range(i+1,n+1):
3.     for k in range(1,j+1):
4.       count = count + 1
```

حل:

$$\begin{aligned}
\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j}1 &= \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} j \\
&= \sum_{i=1}^{n-1}(\frac{n(n+1)}{2} - \frac{i(i+1)}{2}) \\
&= \frac{1}{2}(n(n-1)(n+1) - \sum_{i=1}^{n-1} i^2 - \sum_{i=1}^{n-1} i) \\
&= \frac{1}{2}(n(n-1)(n+1) - \frac{n(n-1)(2n-1)}{6} - \frac{n(n-1)}{2})
\end{aligned}$$

۱.۱ تحلیل بدترین حالت worst-case analysis

در الگوریتمها و قطعه کدهایی که بررسی کردیم تعداد اجرای دستورات تنها بستگی به اندازه داده‌ی ورودی داشت. به عبارت دیگر اگر n مشخصه اندازه داده ورودی باشد، در هر اجرای الگوریتم فارغ از اینکه محتوای داده ورودی چه باشد، الگوریتم به تعداد مشخصی (تابعی از n) دستور اجرا می‌کند و این میزان ثابت می‌ماند. اما در بسیاری از موارد زمان اجرای الگوریتم علاوه بر اندازه داده ورودی بستگی به محتوای داده ورودی نیز دارد. برای مثال در الگوریتم اقلیدس برای پیداکردن بزرگترین مقسوم علیه مشترک دو عدد a و b تعداد تقسیم‌هایی که الگوریتم انجام می‌دهد بستگی به مقدار a و b دارد. اگر a بر b بخشپذیر باشد در همان ابتدای کار با یک تقسیم کار خاتمه می‌یابد. اما مثالهایی وجود دارد که به تعداد زیادی تقسیم نیاز داریم تا به بزرگترین مقسوم علیه مشترک برسیم. در فن تحلیل الگوریتمها معمولا زمان اجرا برای بدترین ورودی (یا اصطلاحا بدترین حالت) به عنوان معیار تحلیل الگوریتمها در نظر گرفته می‌شود. یعنی با فرض اینکه اندازه داده ورودی n باشد الگوریتم در بدترین حالت چند دستورالعمل اصلی را اجرا می‌کند. این مقدار بستگی به محتوای داده ورودی و پیچیدگی الگوریتم دارد. گاهی پیداکردن بدترین حالت امری ساده است اما در بعضی موارد خود به یک مسئله مشکل تبدیل می‌شود. این در واقع یکی از چالشهای فن تحلیل الگوریتم‌هاست. در برخی موارد تنها می‌توانیم کرانی بالا و یا کرانی پایین برای زمان اجرای الگوریتم در بدترین حالت بدست آوریم.

چند نکته در مورد پیچیدگی زمانی time complexity

- پیچیدگی زمانی یک الگوریتم برابر با تعداد دستورالعملهای اصلی (از قبیل انتصاب، جمع، ضرب، مقایسه و ...) است که الگوریتم A در بدترین حالت برای پردازش داده‌های ورودی با اندازه n انجام می‌دهد. پیچیدگی

زمانی معمولا تابعی صعودی و وابسته به n است. در بسیاری از موارد محاسبه دقیق پیچیدگی زمانی یک الگوریتم میسر نیست اما می‌توان کران بالا و پایین برای آن پیدا کرد. در این جزوه از نماد $T(n)$ برای اشاره به پیچیدگی زمانی الگوریتمها استفاده می‌شود.

- در بعضی جاها پیچیدگی زمانی را تنها بر اساس تعداد اجرای یک دستورالعمل خاص بیان می‌کنند. این معمولا حالتی است که دستورالعمل مورد نظر سهم بیشتری نسبت به بقیه دارد و پیچیدگی زمانی را تحت الشعاع خود قرار می‌دهد. برای مثال در الگوریتمهای مرتب سازی که بر مبنای مقایسه عمل می‌کنند پیچیدگی زمانی عموما بر اساس تعداد مقایسه‌ی بین عناصر آرایه بیان می‌شود.

- اندازه داده ورودی n تعداد اعداد یا حروفی است که ورودی مسئله را بیان می‌کنند. برای مثال در مسئله مرتب سازی یک آرایه با n عنصر، اندازه ورودی n در نظر گرفته می‌شود. در مسئله ضرب ماتریس های مربعی با ابعاد $n \times n$ اندازه ورودی n^2 در نظر گرفته می‌شود. گاهی اندازه ورودی به شکل دقیقتر بر اساس تعداد بیتهای لازم برای نمایش داده ورودی در نظر گرفته می‌شود. برای مثال در مسئله تشخیص اول بودن یک عدد، ورودی مسئله یک عدد صحیح است. در صورتی که عدد صحیح ورودی کمتر از n باشد، اندازه ورودی را $\log n$ در نظر می‌گیرند که برابر با تعداد بیتهای لازم برای نمایش ورودی مسئله است.

- علاوه بر بدترین حالت، میانگین زمان اجرا هم به عنوان یک معیار در نظر گرفته می‌شود. در این معیار با فرض اینکه داده ورودی توزیعی تصادفی دارد (برای مثال توزیعی یکنواخت دارد) تعداد اجرای دستورالعملها بصورت امیدریاضی بیان می‌شود. یعنی بطور میانگین چند دستورالعمل اجرا می‌شود.

- در این درس برای تحلیل زمان اجرای الگوریتمها معیار بدترین حالت را بطور پیش فرض در نظر می‌گیریم مگر اینکه خلاف آن ذکر شود.

۲.۱ مرتب سازی حبابی Bubble sort

مرتب سازی حبابی یک روش ساده برای مرتب سازی یک آرایه n عضوی است که از راه مقایسه کردن و جابجایی عناصر به نتیجه مطلوب می‌رسد. در این روش هر بار آرایه از سمت چپ به راست پیمایش شده و هر دو عنصر مجاور مقایسه می‌شوند (هدف این است که در انتها آرایه بصورت صعودی از چپ به راست مرتب شود.) در صورتی که عنصر سمت چپی بزرگتر از عنصر سمت راست باشد، دو عنصر معاوضه می‌شوند، اصطلاحا عمل swap انجام می‌شود. آرایه حداکثر $n-1$ بار پیمایش می‌شود. اگر در پیمایشی هیچ عمل جابجایی (swap) انجام نشود الگوریتم خاتمه می‌یابد. لازم به ذکر است که در مرحله iام، آرایه تا مکان $n-i$ پیمایش می‌شود. دلیل این کار این است که در پایان مرحله i ام، i عنصر بزرگ آرایه در مکان $n-i+1$ تا n به ترتیب قرار می‌گیرند و دیگر نیازی به مقایسه شدن با بقیه ندارند.

در الگوریتم زیر یک متغیر به اسم flag را تعریف کرده‌ایم تا مرتب بودن آرایه را نشان دهد. در ابتدای کار فرض بر این است که آرایه ورودی مرتب است و flag با true مقداردهی می‌شود. اگر در پیمایشی عمل جابجایی انجام شود flag مقدارش false می‌شود. بعد از هر پیمایش، الگوریتم flag را چک می‌کند. اگر flag مقدار true داشت (یعنی در این پیمایش عمل جابجایی انجام نشده است)، حلقه for اول قطع شده و کار پایان می‌یابد در غیر این صورت flag دوباره true می‌شود و پیمایش بعدی شروع می‌شود.

```
def Bubble_Sort(A):
 1. n = len(A)
 2. flag = true
 3. for i in range(1,n):
 4.   for j in range(1,n-i):
 5.       if(A[j] > A[j+1]):
 6.            swap (A[j],A[j+1])
 7.            flag = false
 8.   if (flag == true): break   # no swap is done, exiting..
 9.   else:   flag = true
```

مثال نشان داده در شکل طرز کار این الگوریتم را نشان می‌دهد.

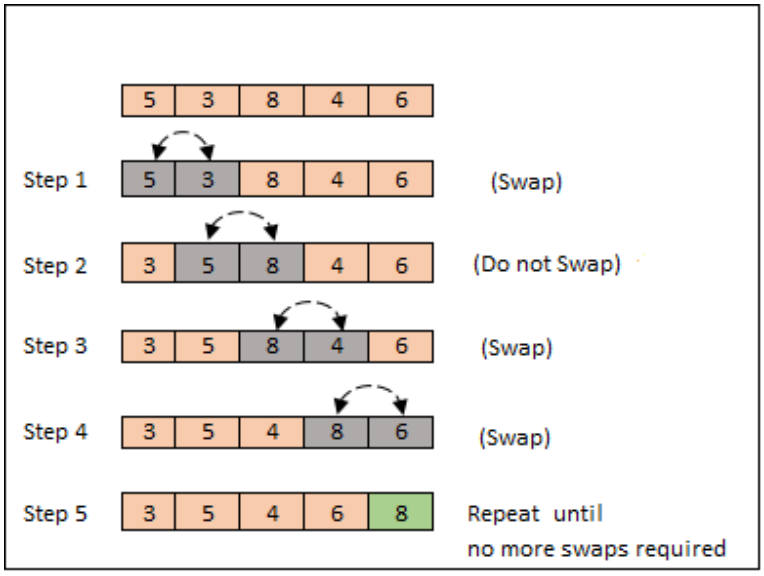


Figure ۱: نمونه‌ای از اجرای مرتب سازی حبابی در اولین پیمایش آرایه

حال می‌خواهیم بدانیم الگوریتم مرتب سازی حبابی چند عمل مقایسه انجام می‌دهد. اگر از چک کردن flag صرف نظر کنیم، در پیمایش اول $n-1$ مقایسه انجام می‌شود. در پیمایش دوم $n-2$ مقایسه و به همین ترتیب در پیمایش iام به تعداد $n-i$ مقایسه انجام می‌شود. در کل تعداد پیمایش ها حداکثر $n-1$ است. پس حداکثر

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

مقایسه انجام می‌شود. تعداد پیمایشهایی که الگوریتم انجام می‌دهد بستگی به آرایه ورودی دارد. اگر آرایه از قبل مرتب باشد، فقط یک پیمایش انجام می‌شود و تعداد مقایسه ها $n-1$ است. این بهترین حالت است. در بدترین حالت آرایه بصورت برعکس مرتب شده است (چرا؟) و $n-1$ پیمایش لازم است. پس در بدترین حالت مرتب سازی حبابی $n(n-1)/2$ مقایسه بین عناصر انجام می‌دهد.

۳.۱ مرتب سازی درجی Insertion sort

همانند مرتب سازی حبابی، مرتب سازی درجی روشی برای مرتب سازی عناصر یک آرایه است که بر اساس مقایسه و جابجایی عناصر طراحی شده است. در این روش آرایه به دو قسمت تقسیم میشود. قسمت مرتب و قسمت نامرتب. قسمت مرتب در سمت چپ آرایه قرار دارد و قسمت نامرتب در سمت راست آن. در ابتدای کار قسمت مرتب تنها شامل عنصر اول آرایه است؛ بازه ای به طول یک. الگوریتم آرایه را از چپ به راست پیمایش میکند و هر بار عنصری از قسمت نامرتب برمیدارد و در جای مناسب خود در قسمت مرتب شده قرار میدهد. اصطلاحا عنصر جدید در قسمت مرتب درج میشود. بدین ترتیب هر بار یکی از قسمت نامرتب کم میشود و عنصری به قسمت مرتب اضافه میشود. اینکار اینقدر ادامه میابد تا اینکه عنصری در قسمت نامرتب باقی نماند.

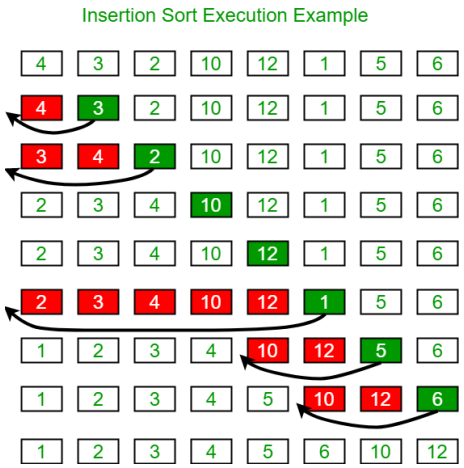


Figure ۲: مثالی از مرتب سازی درجی

برای درج عنصر x در قسمت مرتب، الگوریتم x را با بزرگترین عنصر قسمت مرتب (عنصر انتهای راست بازه مرتب) مقایسه کرده و در صورتی که از آن کوچکتر باشد، به سمت چپ حرکت کرده و آن را با عنصر بعدی بازه مرتب مقایسه میکند. این کار اینقدر ادامه می یابد تا اینکه بزرگترین عنصر بازه مرتب پیدا شود که از x کمتر است. همزمان با این عناصری که با x مقایسه شده اند به سمت راست شیفت داده می شوند تا جا برای درج عنصر جدید باز شود. هنگامی که بزرگترین عنصر پیدا شد که از x کوچکتر است دیگر جابجایی انجام نمیشود چون مکان مناسب برای درج x در بازه مرتب پیدا شده است.

شبه کد زیر جزئیات الگوریتم مرتب سازی درجی را نشان میدهد.

```
def Insertion_Sort(A):
1.      n = len(A)
2.      for k in range(1,n+1):
3.          key = A[k]
4.          i = k
5.          while( i > 1  and  A[i-1] > key):
6.                A[i]  = A[i-1]
7.                i = i-1
8.          A[i]  = key
```

تعداد مقایسه های مرتب سازی درجی. الگوریتم مرتب سازی درجی در کل $n-1$ عمل درج انجام میدهد. در هر عمل درج، عنصری که قرار است درج شود در بدترین حالت با همه عناصر بازه مرتب مقایسه میشود (موقعی که از همه آنها کوچکتر باشد). زمانی که iامین درج انجام میشود، بازه مرتب حاوی i عنصر میباشد. پس تعداد کل

مقایسه ها در بدترین حالت برابر است با

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

بدترین حالت، مشابه مرتب سازی حبابی، زمانی اتفاق می افتد که آرایه ورودی بصورت برعکس مرتب شده باشد. بهترین حالت زمانی است که آرایه ورودی مرتب شده باشد. در این حالت تنها $n-1$ مقایسه انجام میشود.

۲ تحلیل مجانبی و توابع رشد

همانطور که گفته شد، در بعضی موارد محاسبه دقیق تعداد اجرای دستورالعملها کار آسانی نیست اما غالبا می‌توان کران بالا و پایین خوبی برای آن پیدا کرد. برای مثال ممکن است گفته شود که الگوریتم مرتب سازی خاصی در بدترین حالت تعداد مقایسه‌ی که انجام می‌دهد بین n^2 و $n^2 - n$ است. یا اگر $T_A(n)$ پیچیدگی زمانی الگوریتم A باشد ممکن است ثابت شود که

$$\frac{1}{2} n \log n \leq T_A(n) \leq 2n \log n + n$$

گذشته از این جهت مقایسه الگوریتمها، زمان اجرای الگوریتمها را با در نظر گرفتن توابع رشد مختلف طبقه بندی می‌کنند. یک نماد برای طبقه بندی توابع از لحاظ رشد نماد O می‌باشد. نماد O وقتی به کار می‌رود که بخواهیم کران بالایی را برای پیچیدگی زمانی یک الگوریتم معرفی کنیم.

Definition 2.1. $O(g(n)) = \{f(n) : \exists\, c, n_0 > 0 \text{ such that } \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$

$O(g(n))$ در واقع مجموعه همه توابعی است که $g(n)$ برای آنها مثل یک کران بالا عمل می‌کند یا به عبارتی دیگر، آهنگ رشد آنها حداکثر متناسب با $g(n)$ است. نوشته می‌شود $f(n) \in O(g(n))$ و یا $f(n) = O(g(n))$ و گفته می‌شود که $g(n)$ یک کران بالای مجانبی برای تابع $f(n)$ است.

- ثابت کنید $100n^2 + 5n - 10 \in O(n^2)$

  حل: برای $c = 200$ و هر $n \geq 2$ داریم

$$100n^2 + 5n - 10 \leq 200n^2$$

پس طبق تعریف نماد O عبارت بالا درست است. □

برای اشاره به کران پایین از نماد Ω استفاده می‌شود.

Definition 2.2. $\Omega(g(n)) = \{f(n) : \exists\, c, n_0 > 0 \text{ such that } \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$

نوشته می‌شود $f(n) \in \Omega(g(n))$ (و یا $f(n) = \Omega(g(n))$) و گفته می‌شود که $g(n)$ یک کران پایین مجانبی برای $f(n)$ است. $\Omega(g(n))$ در واقع مجموعه همه توابعی است که $g(n)$ برای آنها مثل یک کران پایین عمل می‌کند یا به عبارتی دیگر، آهنگ رشد آنها حداقل متناسب با $g(n)$ است.

- نشان دهید $2^n \neq \Omega(4^n)$

حل: چون $\lim_{n\to\infty} \frac{2^n}{4^n} = 0$ پس برای هر ثابت غیر صفر مثبت c و عدد n_0 داریم

$$\forall n \geq n_0, \quad c4^n \not\leq 2^n$$

□

از ترکیب نمادهای O و Ω نماد Θ بدست می‌آید. برای بیان اینکه تابعی آهنگ رشدی متناسب با تابعی دیگر دارد از نماد Θ استفاده می‌شود. تعریف رسمی زیر را برای این نماد داریم.

Definition 2.3. $\Theta(g(n)) = \{f(n) : \exists\ c_1, c_2, n_0 > 0$ such that
$\forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

وقتی گفته می‌شود $f(n) = \Theta(g(n))$، این بدین معنی است که آهنگ رشد f و g برای مقادیر بزرگ n یکسان است. می‌نویسیم $f(n) \in \Theta(g(n))$ و می‌گوییم که $g(n)$ کران بسته مجانبی برای $f(n)$ است.
برای الگوریتم B که در بخش مقدمه توصیف شد داریم $T(n) \in \Theta(n^2)$. این بدین معناست که پیچیدگی زمانی الگوریتم B متناسب با تابع n^2 رشد می‌کند یا اصطلاحا آهنگ رشد n^2 را دارد. به زبان دقیق تر داریم

$$\exists\ c_1, c_2, n_0 > 0 \text{ such that for all } n \geq n_0, c_1 n^2 \leq T_B(n) \leq c_2 n^2$$

بعضی اوقات از عبارت $T(n) = \Theta(n^2)$ استفاده می‌شود که همان معنای بالا را دارد.

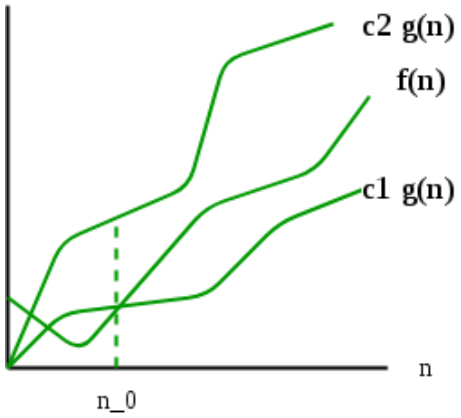


Figure ۳: نمایشی از تابع رشد $f(n) = \Theta(g(n))$

قضیه. $f(n) = \Theta(g(n))$ اگر و فقط اگر $f(n) = O(g(n))$ و $f(n) = \Omega(g(n))$.
اثبات: از تعریف نتیجه می‌شود.

قضیه.

1. $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} = k < \infty \quad \Rightarrow \quad f(n) = \Theta(g(n))$

2. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k < \infty \quad \Rightarrow \quad f(n) = O(g(n))$

3. $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \quad \Rightarrow \quad f(n) = \Omega(g(n))$

دقت کنید که در روابط بالا رابطه "اگر و فقط اگر" بین طرفین وجود ندارد. برای مثال ممکن است حد $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ وجود نداشته باشد اما $f(n) = O(g(n))$ برقرار باشد.

نتیجه. با فرض $a_k > 0$ و $k > 0$ داریم

$$a_k n^k + a_{k-1} + \ldots = \Theta(n^k)$$

- نشان دهید $100n^2 + 5n - 10 \neq \Theta(n^3)$

حل: نشان می‌دهیم که $100n^2 + 5n - 10 \neq \Omega(n^3)$. این بدین معنی است برای هر ثابت غیر صفر c و n_0 داریم $\forall n \geq n_0, \quad cn^3 \nleq 100n^2 + 5n - 10$

این درست است چون $\lim_{n\to\infty} \frac{100n^2+5n-10}{n^3} = 0$

- نشان دهید $\log(n!) = \Theta(n \log n)$

راه حل اول: با استفاده از تقریب استرلینگ–رابینز برای فاکتوریل. برای هر $n > 0$ صحیح داریم

$$\sqrt{2\pi} n^{n+\frac{1}{2}} e^{-n} e^{\frac{1}{12n+1}} \leq n! \leq \sqrt{2\pi} n^{n+\frac{1}{2}} e^{-n} e^{\frac{1}{12n}}$$

در نتیجه برای کران پایین داریم

$$\log(\sqrt{2\pi}) + (n + \frac{1}{2}) \log n + (\frac{1}{12n+1} - n) \log e \leq \log(n!)$$

پس

$$n \log n - n \log e \leq \log(n!)$$

برای $n > e^2$

$$\frac{1}{2} n \log n \leq \log(n!)$$

به همین ترتیب برای کران بالا داریم

$$\log(n!) \leq \log(\sqrt{2\pi}) + (n + \frac{1}{2}) \log n + (\frac{1}{12n}) \log e$$

برای $n \geq 1$

$$\log(n!) \leq 2n \log n + 8$$

در نتیجه برای $n \geq 4$ داریم

$$\log(n!) \leq 3n \log n$$

راه حل دوم:

$$\log(n!) = \sum_{i=1}^{n} \log i \leq n \log n$$

از طرفی دیگر

$$\frac{n}{2}\log n - \frac{n}{2} \leq \frac{n}{2}\log\frac{n}{2} \leq \sum_{i=1}^{n}\log i = \log(n!)$$

پس برای $n \geq 4$ داریم

$$\frac{1}{4}n\log n \leq \log(n!)$$

۳ الگوریتمهای بازگشتی

استفاده از استراتژی بازگشت یک روش معمول برای حل مسائل است. در این استراتژی مسئله با روش استقرائی اما بالعکس (از بالا به پایین) حل می‌شود. هر مسئله به زیرمسائلی از نوع مسئله اصلی اما به اندازه کوچکتر تقسیم می‌شود. زیرمسائل کوچکتر هم به نوبه خود بصورت بازگشتی، با استفاده از تقسیم به زیرمسائل کوچکتر از خود، حل می‌شوند. در نهایت مسئله آنقدر تقسیم می‌شود تا اندازه زیرمسئله به قدری کوچک باشد که بتوان آن را بطور مستقیم حل کرد.

فاکتوریل می‌تواند به عنوان یک نمونه از حل بازگشتی در نظر گرفته شود. برای محاسبه $n!$، مقدار $(n-1)!$ در n ضرب می‌شود. به همین ترتیب $(n-1)!$ خود بصورت بازگشتی محاسبه می‌شود.

۱.۳ مسئله برج هانوی

در یک معمای قدیمی، سه میله و تعدادی قرص سوراخدار داده شده بطوری که قرصها اندازه و قطر متفاوت دارند. قرصها به ترتیب اندازه قطر (بطوری که در شکل مشاهده میشود) در میله سمت چپ قرار داده شده‌اند. هدف انتقال قرصها به میله سمت راست (با کمک گرفتن از میله وسطی) است بطوریکه قوانین زیر رعایت شود:

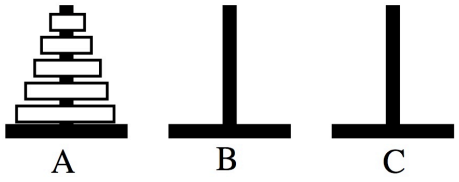


Figure ۴: مثالی از مسئله برج هانوی

- هر بار تنها یک قرص حرکت داده شود.
- در هر حرکت بالاترین قرص یک میله برداشته شود.
- هیچ قرصی نباید روی قرص کوچکتر از خود گذاشته شود.

در این معما هدف جابجایی قرصها با کمترین تعداد حرکت است. در حالت کلی n قرص داریم که باید از میله A به میله C انتقال داده شوند. راه حل بازگشتی زیر برای این معما پیشنهاد می‌شود. فرض کنید یک الگوریتم داریم که مسئله را برای $n-1$ قرص حل می‌کند. برای انتقال n قرص سه قدم اصلی باید انجام شود.

۱. ابتدا $n-1$ قرص بالای میله A به میله B منتقل می‌شوند (این کار با استفاده از الگوریتم $n-1$ قرص و با کمک گرفتن از میله C به عنوان میله کمکی انجام می‌شود)

۲. قرص باقیمانده در میله A به میله C منتقل می‌شود

۳. $n-1$ قرص میله B با کمک گرفتن از میله A به میله C منتقل می‌شوند.

این الگوریتم در شبه کد زیر بیان شده است.

```
1. TOWERS_OF_HANOI(n,A,B,C)
2.     if (n==1)  move the disk from A to C
3.     else
4.                TOWERS_OF_HANOI(n-1,A,C,B)
5.                move the only disk from A to C
6.                TOWERS_OF_HANOI(n-1,B,A,C)
```

لم. راه حل بازگشتی برای جابجایی n قرص 2^n-1 حرکت انجام می‌دهد.

اثبات: فرض کنید $T(n)$ تعداد حرکتهای راه حل بازگشتی برای انتقال n قرص باشد. داریم

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

با استقرا براحتی میتوان نشان داد که $T(n)=2^n-1$. □

لم. هر الگوریتم برای مسئله برجهای هانوی با n قرص حداقل 2^n-1 حرکت انجام میدهد.

اثبات: ابتدا نشان می‌دهیم در راه حل بهینه بزرگترین قرص فقط یکبار حرکت داده می‌شود. واضح است در هر راه حل آخرین باری که قرص بزرگ حرکت داده می‌شود باید به میله C منتقل شود. در زمان این حرکت بقیه قرصها همه باید در میله‌ای دیگری باشند و میله C خالی باشد. بعد از اینکه قرص بزرگ برای آخرین بار جابجا شد باید بقیه قرصها جابجا شوند که مانند حل یک مسئله برای $n-1$ قرص است. در این حالت وضعیت اولیه $n-1$ قرص، چه در میله A باشند و چه در میله B از لحاظ هزینه انتقال تفاوتی ایجاد نمی‌کند. (چون هر میله را می‌توان به عنوان میله کمکی برای دیگری در نظر گرفت). پس میتوان فرض کرد که $n-1$ قرص در میله B هستند زمانی که بزرگترین قرص جابجا می‌شود. با این فرض قرص بزرگ آخرین بار از میله A به میله C منتقل می‌شود. این کار را میتوان در اولین جابجایی قرص بزرگ انجام داد. پس نیازی به جابجایی های بیشتر نیست.

با در نظر گرفتن این واقعیت در هر راه حل بهینه، ابتدا $n-1$ قرص به میله B منتقل می‌شوند. سپس قرص بزرگ منتقل می‌شود و بعد از آن $n-1$ قرص از B به C منتقل می‌شوند. در نتیجه رابطه بازگشتی

$$T(n)=2T(n-1)+1$$

برای هر استراتژی بهینه برقرار است. این ادعای ما را اثبات می‌کند. □

۲.۳ جستجوی دودویی

جستجوی دودویی یا binary search الگوریتمی برای پیدا کردن مکان یک عنصر در آرایه‌ای مرتب است. به عبارت دیگر جستجوی دودویی الگوریتمی برای پیدا کردن محل رخداد x در یک آرایه مرتب است. اگر آرایه ورودی حاوی تکرار باشد، در صورت وجود، الگوریتم محل یکی از رخدادهای x را برمی‌گرداند. اگر x در آرایه نباشد الگوریتم -1 را برمی‌گرداند. الگوریتم جستجوی دودویی یک توصیف بازگشتی دارد که به صورت زیر است.

برای پیدا کردن x در زیر آرایه $A[i,j]$ ابتدا x با عنصر وسط زیر آرایه در محل $mid=i+(j-i)/2$ مقایسه می‌شود. اگر تساوی برقرار بود محل x پیدا شده است و جستجو خاتمه می‌یابد. اگر $A[mid]>x$ جستجوی

دودویی بصورت بازگشتی در بازه $[i, mid-1]$ ادامه می‌یابد در غیر این صورت جستجو در بازه $[mid+1, j]$ دنبال می‌شود. اگر $i = j$، و $A[i] == x$، اندیس i به عنوان جواب برگردانده میشود در غیر این صورت -1 برگردانده می‌شود.

```
def BinarySearch(A,i,j,x):
 1.
 2.   if(i<=j):
 3.       mid = i + (j-i)//2   #integer division
 4.       if(A[mid]==x): return mid
 5.       if(A[mid] > x): return  BinarySearch(A,i,mid-1,x)
 6.       return  BinarySearch(A,mid+1,j,x)
 7.
 8.    else: return -1
```

برای پیدا کردن محل رخدادی از x در آرایه مرتب A رویه $BinarySearch(A, 0, n-1, x)$ را فراخوانی میکنیم.

تحلیل زمان اجرای الگوریتم:

لم. اگر $T(n)$ تعداد مقایسه الگوریتم BinarySearch باشد، آنگاه $T(n) = \Theta(\log n)$.

اثبات: در هر فراخوانی BinarySearch حداکثر سه مقایسه در سطح اول انجام میشود و طول بازه هدف به $\lfloor n/2 \rfloor$ یا $\lfloor n/2 \rfloor - 1$ تقلیل می یابد. پس میتوان گفت رابطه بازگشتی زیر برای تعداد مقایسه ها $T(n)$ برقرار است.

$$T(n) \leq \begin{cases} 3 + T(\lfloor n/2 \rfloor) & n > 1 \\ 2 & n = 1 \end{cases}$$

بدترین حالت زمانی رخ میدهد که x در آرایه نیست و از همه عناصر بزرگتر است. در این حالت همه مقایسه ها انجام میشود و طول بازه جدید همیشه $\lfloor n/2 \rfloor$ است. پس برای حالتی که $x > \max A$ داریم

$$T(n) = \begin{cases} 3 + T(\lfloor n/2 \rfloor) & n > 1 \\ 2 & n = 1 \end{cases}$$

با استفاده از استقرا می‌توان نشان داد که $T(n) = \Theta(\log n)$. جزئیات راه حل را در بخش بعد نشان می‌دهیم. □

براحتی میتوان دید زمانی که $n = 2^k$ و $x > \max A$، تعداد مقایسه های الگورتیم برابر با $3k + 2$ میباشد که همان $3 \log n + 2$ است.

۳.۳ مرتب سازی ادغامی

مرتب سازی ادغامی روشی برای مرتب سازی یک آرایه است که از راه حل بازگشتی استفاده می‌کند. این الگوریتم ابتدا آرایه را به دو قسمت (تقریبا مساوی) نصف می‌کند و سپس هر قسمت را بصورت بازگشتی مرتب می‌کند و سپس دو نصفه مرتب را با هم ادغام می‌کند. شبه کد زیر روش کار این الگوریتم را بیان می‌کند.

```
 def merge_sort(S):
2 # Sort the elements of list S using the merge-sort algorithm.
3     n = len(S)
4     if n < 2:
5         return # list is already sorted
6
7     mid = n // 2
8     S1 = S[0:mid] # copy of first half
9     S2 = S[mid:n] # copy of second half
10
11    merge_sort(S1) # sort copy of first half
12    merge_sort(S2) # sort copy of second half
13
14    merge(S1, S2, S)
```

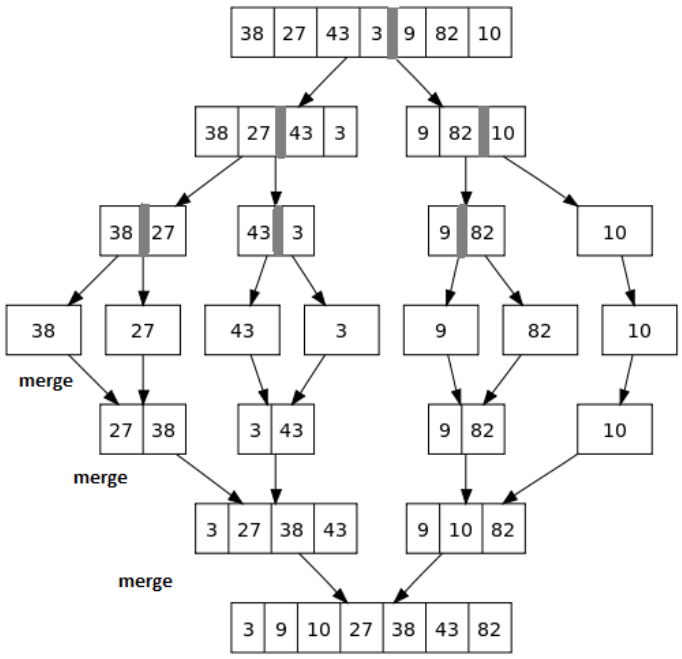


Figure ۵: مراحل اجرای مرتب سازی ادغامی برای یک آرایه با ۸ عنصر

برای ادغام دو آرایه مرتب فرض می‌کنیم که می‌توانیم از حافظه اضافی استفاده کنیم. (ادغام بدون استفاده از حافظه اضافی امکانپذیر است اما الگوریتمش قدری پیچیده‌تر است.) برای ادغام دو لیست مرتب $S1$ و $S2$ به ترتیب به طولهای n و m، از لیست S به طول $n+m$ استفاده می‌کنیم. ادغام را میتوان به روشهای مختلف انجام داد. در اینجا یک روش ساده برای ادغام بیان شده است که با انجام $n+m$ مقایسه دو لیست را در هم ادغام می‌کند و در لیست جدید قرار می‌دهد.

تعداد مقایسه های مرتب سازی ادغامی اگر دقت کنید مرتب سازی ادغامی تنها در مرحله ادغام دو آرایه عمل مقایسه را انجام میدهد. هر بازه به طول n به دو زیر آرایه به طولهای $\lceil \frac{n}{2} \rceil$ و $\lfloor \frac{n}{2} \rfloor$ تقسیم میشود. پس از مرتب شدن دو زیر آرایه عمل ادغام انجام میشود. برای ادغام دو زیر آرایه $n = \lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor$ مقایسه بین عناصر انجام میشود.

پس اگر $T(n)$ تعداد مقایسه ها برای مرتب سازی یک آرایه n عنصری باشد، رابطه بازگشتی زیر در مورد $T(n)$ صدق میکند.

```
1 def merge(S1, S2, S):
2 # Merge two sorted Python lists S1 and S2 into  list S
3    i = j = 0
4    while (i + j < len(S)):
5       if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
6               S[i+j] = S1[i]
7               i += 1
8       else:
9               S[i+j] = S2[j]
10              j += 1
```

از آنجا که در الگوریتم ارائه شده برای مرتب سازی ادغامی، تعداد مقایسه ها تنها بستگی به اندازه n دارد، بدترین حالت و بهترین حالت از لحاظ تعداد مقایسه یکسان هستند. این به این دلیل است که رویه Merge برای ادغام دو آرایه به طول n و m همواره $n+m$ عمل مقایسه انجام می‌دهد و محتوی آرایه‌ها روی آن تاثیری ندارد. می‌توان الگوریتم ادغامی طراحی کرد که تعداد مقایسه هایش وابسته به محتوی آرایه‌های ورودی است. در هر صورت در بدترین حالت هر الگوریتم ادغام نیاز به $O(n+m)$ مقایسه خواهد داشت.

۴ حل روابط بازگشتی

همانطور که در قسمت قبل مشاهده شد، پیچیدگی زمانی الگوریتمهای بازگشتی با استفاده از روابط بازگشتی بیان می‌شود. در این قسمت چند روش کلی را برای حل روابط بازگشتی ذکر می‌کنیم. لازم به ذکر است که هیچ کدام از روشهای پیشنهاد شده جامعیت لازم را نداشته و قدرت حل هر رابطه بازگشتی را ندارند. به همین دلیل، پیدا کردن مهارت در حل روابط بازگشتی مستلزم تجربه و گاها استفاده از روشهای ابتکاری است. در برخی موارد با استفاده از تغییر متغیر می‌توان یک رابطه بازگشتی پیچیده را به فرمی ساده تبدیل کرد که با روشهای معمول قابل حل است.

۱.۴ روش حدس و استقرا

در این روش ابتدا سعی می‌کنیم جواب را حدس بزنیم و سپس با استقرا آن را اثبات کنیم. برای اثبات $T(n) = O(f(n))$ باید نشان دهیم n_0 و c وجود دارند که برای هر $n \geq n_0$ رابط $T(n) \leq cf(n)$ برقرار است. معمولا در این موارد باید از استقرای قوی استفاده شود. یعنی فرض می‌گیریم که برای هر $n > k \geq n_0$ رابطه $T(k) \leq cf(k)$ برقرار است و سپس، با استفاده از این فرض، حکم استقرا را نشان می‌دهیم. دقت شود که ثابت c در این پروسه نباید تغییر کند. برای مثال نشان دادن اینکه $T(n) \leq 2cf(n)$ حکم استقرا را ثابت نمی‌کند.

- ثابت کنید $T(n) = \Theta(\log n)$ هنگامی که

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + 3 & n > 1 \\ 2 & n = 1 \end{cases}$$

حل: با استفاده از استقرای قوی نشان می‌دهیم برای هر $n \geq 1$، $T(n) \leq 3\log n + 2$. حکم برای $T(1)$ درست است. با فرض استقرا، برای هر $k < n$ داریم $T(k) \leq 3\log k + 2$. حال

$$\begin{aligned} T(n) &\leq (3\log\lfloor n/2\rfloor + 2) + 3 \\ &\leq 3\log(n) + 2 \end{aligned}$$

برای کران پایین، دوباره با استفاده از استقرای قوی برای هر $n \geq 1$ نشان می‌دهیم $T(n) \geq \frac{3}{2}\log n + 2$. حکم برای $T(1)$ درست است. با فرض استقرا، برای هر $k < n$ داریم $T(k) \geq \frac{3}{2}\log k + 2$. حال برای هر $n \geq 2$

$$\begin{aligned} T(n) &\geq (\frac{3}{2}\log\lfloor n/2\rfloor + 2) + 3 \\ &= \frac{3}{2}\log\lfloor n/2\rfloor + 5 \\ &\geq \frac{3}{2}(\log n - 2) + 5 \\ &\geq \frac{3}{2}\log n + 2 \end{aligned}$$

در نتیجه برای هر $n \geq 1$ داریم

$$\frac{3}{2}\log n + 2 \leq T(n) \leq 3\log n + 2$$

□

- ثابت کنید $T(n) = O(n\log n)$ زمانی که

$$T(n) = \begin{cases} T(\lfloor n/2\rfloor + 17) + n & n > 100 \\ 1 & 1 \leq n \leq 100 \end{cases}$$

حل: باید نشان دهیم ثابتهای n_0 و c وجود دارند بطوریکه $T(n) \leq cn\log n$ برای هر $n \geq n_0$. از استقرای قوی استفاده می‌کنیم.

فرض استقرا:

$$\exists \text{ constant } c \mid \forall\ 100 \leq k < n \qquad T(k) \leq ck\log k$$

با استفاده از فرض استقرا و فرض مسئله داریم،

$$\begin{aligned} \forall n \geq 100, \qquad T(n) &= T(\lfloor n/2\rfloor + 17) + n \\ &\leq 2\Big(c(\lfloor n/2\rfloor + 17)\log(\lfloor n/2\rfloor + 17)\Big) + n \\ &\leq 2c(n/2 + 17)\log(n/2 + 17) + n \\ &< (cn + 34c)(\log n - \frac{1}{2}) + n \\ &= cn\log n + \Big(34c\log n - 17c - \frac{cn}{2} + n\Big) \\ &< cn\log n \qquad \text{موقعی که } c \geq 8 \text{ و } n > 2^{10} \end{aligned}$$

عبارت داخل پرانتر موقعی که $c \geq 8$ و $n \geq 2^{10}$ کمتر از صفر است. پس در اینجا می‌توانیم فرض بگیریم $c = 8$. برای nهای کمتر از 2^{10}، می‌توان بدون استفاده از استقرا نشان داد $T(n) \leq 8n\log n$ (رابطه بازگشتی حداکثر چهار بار فراخوانی می‌شود.) اینها در واقع نقش پایه استقرا را بازی می‌کنند. از بحثهای بالا نتیجه می‌شود برای هر $n \geq 1$ داریم $T(n) \leq 8n\log n$. □

۲.۴ درخت بازگشت

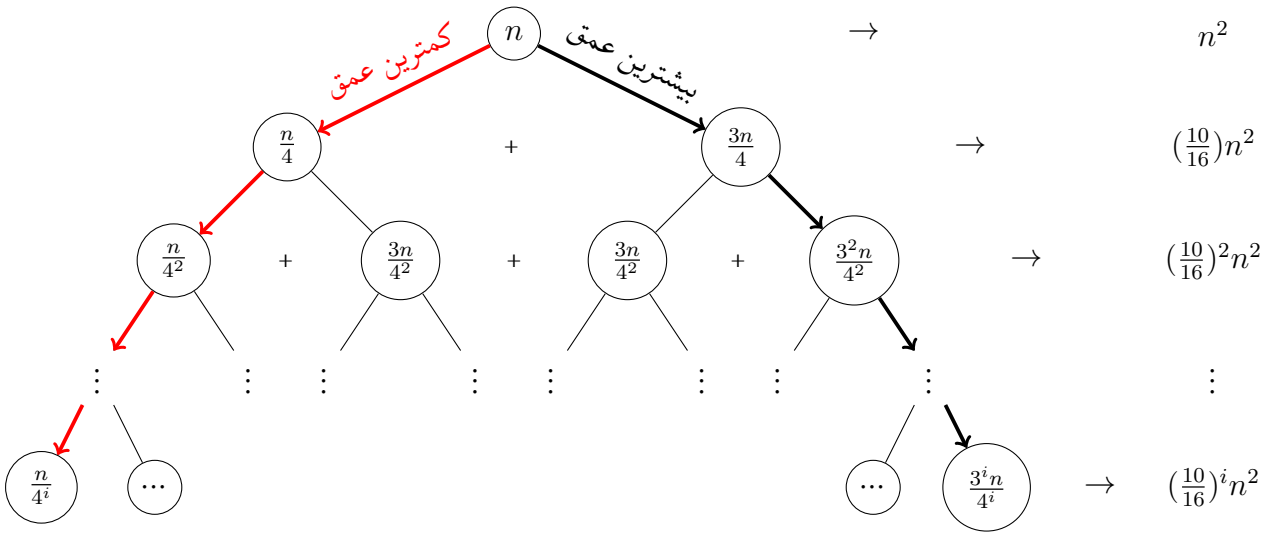
درخت بازگشت ابزاری مناسبی برای نمایش بسط یک رابطه بازگشتی در سطوح مختلف است. هر سطح حاوی یک سری فراخوانی بازگشتی از تابع مربوطه به همراه مقداری ثابت است که در نهایت با مقادیر ثابت بقیه سطوح جمع زده می‌شود. برای مثال برای رابطه بازگشتی $T(n) = T(n/3) + T(2n/3) + n$ در سطح اول مقدار ثابت n و فراخوانیهای بازگشتی $T(n/3)$ و $T(2n/3)$ قرار دارند. ثابت n تغییری نمی‌کند اما فراخوانی‌های بازگشتی در سطح بعدی با فراخوانی‌ها و مقادیر ثابت جدید جایگزین می‌شوند. بسط رابطه در سطوح مختلف ادامه می‌یابد تا زمانی که همه شاخه‌های درخت در عمق خود به مقادیری در حد $O(1)$ برسند.

برای پیدا کردن کران بالایی برای رابطه بازگشتی می‌توان طولانی‌ترین عمق را پیدا کرد و ارتفاع آن را در کران بالایی برای مقدار ثابت سطوح ضرب کرد. در بعضی موارد با دقت بیشتر می‌توان این کار را انجام داد و برای هر سطح از کران بالایی درخور آن استفاده کرد و حاصل را بصورت یک سری جمع زد.

- ثابت کنید $T(n) = \Theta(n^2)$ موقعی که

$$T(n) = \begin{cases} T(n/4) + T(3n/4) + n^2 & n > 1 \\ 1 & n \leq 1 \end{cases}$$

حل: شکل زیر درخت بازگشت رابطه را نشان می‌دهد.



برای پیدا کردن ارتفاع بیشترین عمق باید داشته باشیم

$$\frac{3^i n}{4^i} \leq 1$$

پس $i \geq \log_{4/3} n$. در نتیجه اگر d طول بیشترین عمق باشد داریم $d \leq \lceil \log_{4/3} n \rceil$. پس برای پیدا کردن کران بالایی برای $T(n)$ می‌توانیم بنویسیم

$$T(n) \leq n^2 \sum_{i=0}^{d} (\frac{10}{16})^i \leq n^2 \frac{8}{3}$$

برای کرای پایینی برای $T(n)$، واضح است $T(n) \geq n^2$ (طبق فرض مسئله). پس برای هر $n \geq 1$ داریم

$$n^2 \leq T(n) \leq \frac{8}{3}n^2$$

□

- ثابت کنید $T(n) = \Theta(n \log n)$ هنگامی که

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & n > 1 \\ 0 & n = 1 \end{cases}$$

حل: اول ثابت می‌کنیم برای هر $n \geq 1$ رابطه $T(n) \leq n \log n + 2n$ برقرار است. برای اثبات، رابطه $T(n)$ را آنقدر بسط می‌دهیم تا اندازه ورودی تابع T در همه جا کمتر از 2 شود. درخت بازگشت این رابطه در شکل نشان داده شده است. در هر سطح جمع مقادیر ثابت حداکثر n است. برای پیدا کردن کرانی بالا برای $T(n)$ می‌توانیم اندازه بیشترین عمق را ضرب در n کنیم. بیشترین عمق در شاخه انتهای سمت راست اتفاق می افتد، زمانی که هر بار پارامتر ورودی k با $\lceil \frac{k}{2} \rceil$ جایگزین می‌شود. حال سوال این است که چند بار عملگر $\lceil n/2 \rceil$ بطور متوالی روی n اعمال شود تا حاصل کمتر از 2 شود. از آنجا که

$$\overbrace{\lceil \frac{\lceil \frac{n}{2} \rceil}{2} \rceil \\ \vdots}^{i \text{ بار}} = \lceil \frac{n}{2^i} \rceil < 2$$

زمانی که i حداقل $\lceil \log n \rceil$ باشد مقدار حاصل کمتر از 2 خواهد شد. در نتیجه میتوان گفت که بیشترین عمق حداکثر $\log n + 1$ خواهد بود. پس $T(n)$ حداکثر $n(\log n + 2)$ است. این کران بالا برای $T(n)$ را اثبات می‌کند.

حال برای پیدا کردن کرانی پایین، باید کمترین عمق را در نظر بگیریم. در درخت بازگشت، شاخه انتهای سمت چپ سریعتر از بقیه به مقدار کمتر از 2 می‌رسد، پس کمترین عمق را دارد. برای پیدا کردن حدی پایین برای کمترین عمق، با در نظر گرفتن

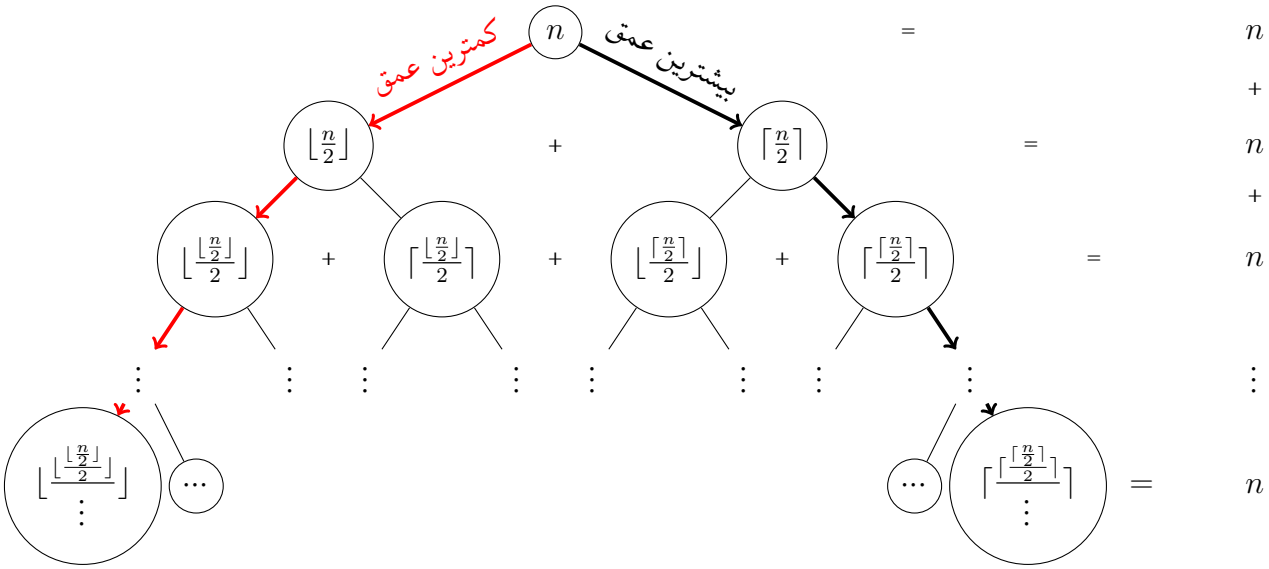
$$\overbrace{\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \rfloor \\ \vdots}^{i \text{ بار}} = \lfloor \frac{n}{2^i} \rfloor \geq 2$$

کافی است i را $\lfloor \log n - 1 \rfloor$ انتخاب کنیم. چون در این سطح همه شاخه‌ها حضور دارند، جمع سطح مربوطه برابر با n است. در نتیجه $T(n)$ حداقل $n(\log n - 1)$ خواهد بود.

از بحثهای بالا نتیجه می‌شود برای هر $n \geq 1$

$$n \log n - n \leq T(n) \leq n \log n + 2n$$

□



□

ترفندی برای حذف $\lfloor\ \rfloor$ و $\lceil\ \rceil$ موقع حل روابط بازگشتی

در الگوریتم‌های بازگشتی معمولا ورودی با اندازه n به چند قسمت صحیح تقسیم می‌شود. برای مثال با فرض اینکه $a > 1$ یک ثابت است، آرایه‌ای به طول n به دو قسمت $\lfloor n/a \rfloor$ و $\lceil (1-1/a)n \rceil$ تقسیم می‌شود. به همین خاطر عملگرهای گرد کردن $\lfloor\ \rfloor$ و $\lceil\ \rceil$ در روابط بازگشتی مربوطه ظاهر می‌شود. یک راه برای ساده کردن مسئله و حذف این عملگرها به این صورت است. اگر بتوانیم ثابت کنیم که $T(n)$ همواره یک تابع صعودی است، برای پیدا کردن کرانی بالا و پایین برای $T(n)$ می‌توانیم از روش زیر استفاده کنیم.

فرض کنید $a^k \leq n < a^{k+1}$. چون T صعودی است، پس

$$T(a^k) \leq T(n) \leq T(a^{k+1})$$

روشن است کران بالایی برای $T(a^{k+1})$ یک کران بالا برای $T(n)$ بدست می‌دهد. به همین شکل، یک کران پایین برای $T(a^k)$ کرانی پایین برای $T(n)$ خواهد بود. نتیجه اینکه در رابطه بازگشتی می‌توانیم فرض کنیم که n همواره توانی از a است. به همین خاطر می‌توان با خیال آسوده عملگرهای گرد کردن را از رابطه حذف کرد. موقعی هم که n توانی از a نیست، برای پیداکردن کران بالا و پایین از $T(a^{k+1})$ و $T(a^k)$ استفاده می‌کنیم.

البته باید توجه داشت این روش وقتی نتیجه خوبی به دست می‌دهد که $T(n)$ و $T(an)$ از لحاظ مجانبی تفاوتی نداشته باشند. با فرض اینکه a یک ثابت است، برای توابع چند جمله‌ای و بسیاری از توابع رشد، این خاصیت برقرار است.

- ثابت کنید $T(n) = \Theta(n \log n)$ هنگامی که

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & n > 1 \\ 0 & n = 1 \end{cases}$$

حل: ابتدا ثابت می‌کنیم $T(n)$ صعودی است. از استقرا استفاده می‌کنیم. واضح است $T(3) \geq T(2)$. حال فرض کنید برای $k \geq 3$، داریم $T(k) \geq T(k-1)$. نشان می‌دهیم $T(k+1) \geq T(k)$. با استفاده

از فرض مسئله و فرض استقرا داریم

$$\begin{aligned} T(k+1) &= T(\lceil \frac{k+1}{2} \rceil) + T(\lfloor \frac{k+1}{2} \rfloor) + k + 1 \\ &> T(\lceil k/2 \rceil) + T(\lfloor k/2 \rfloor) + k \qquad \leftarrow \text{با استفاده از فرض استقرا} \\ &= T(k) \end{aligned}$$

پس نتیجه می‌شود که $T(n)$ صعودی است. حال رابطه بازگشتی برای موقعی که n توانی از 2 است بازنویسی می‌شود.

$$T(n) = \begin{cases} 2T(n/2) + n & n = 2^i > 1 \\ 0 & n = 1 \end{cases}$$

با استفاده از روش درخت بازگشت یا هر روش دیگر، موقعی که n توانی از 2 است، می‌توان نتیجه گرفت

$$T(n) = n \log n$$

پس برای $2^k < n < 2^{k+1}$ داریم

$$\frac{n}{2}\lfloor \log n \rfloor \leq 2^k k \leq T(n) \leq 2^{k+1}(k+1) \leq 2n\lceil \log n \rceil$$

نتیجه می‌شود $T(n) = \Theta(n \log n)$. □

۳.۴ قضیه اصلی

قضیه اصلی یک جواب مستقیم برای حل مجانبی روابط بازگشتی با فرم

$$T(n) = aT(\frac{n}{b}) + f(n)$$

بدست می‌دهد. در اینجا فرض می‌شود که $a \geq 1$ و $b > 1$ و تابع $f(n)$ یک تابعی است که بصورت مجانبی مثبت است (یعنی برای مقادیر بالا مثبت است).

قضیه. فرض کنید $a \geq 1$ و $b > 1$ مقادیر ثابت هستند و $f(n)$ یک تابع است. رابطه بازگشتی

$$T(n) = aT(\frac{n}{b}) + f(n)$$

روی مقادیر مثبت n تعریف شده است که در آن n/b برابر با $\lceil n/b \rceil$ یا $\lfloor n/b \rfloor$ است. در آن صورت،

۱. اگر برای $\epsilon > 0$، داشته باشیم $f(n) = O(n^{\log_b a - \epsilon})$ در این صورت $T(n) = \Theta(n^{\log_b a})$

۲. اگر $f(n) = \Theta(n^{\log_b a})$ در این صورت $T(n) = \Theta(n^{\log_b a} \log n)$

۳. اگر $f(n) = \Omega(n^{\log_b a + \epsilon})$ در این صورت $T(n) = \Theta(f(n))$

- نشان دهید $T(n) = \Theta(n \log n)$ زمانی که $T(n) = 3T(n/4) + n \log n$.

حل: با استفاده از قضیه اصلی داریم $f(n) = n \log n$ و $g(n) = n^{\log_4 3}$. چون $f(n) = \Omega(g(n))$ پس داریم $T(n) = \Theta(n \log n)$.

- نشان دهید $T(n) = \Theta(n^2)$ زمانی که $T(n) = 9T(n/3) + n$

حل: با استفاده از قضیه اصلی داریم $f(n) = n$ و $g(n) = n^{\log_3 9} = n^2$. از آنجا که $f(n) = O(g(n)^{1-\epsilon})$ پس $T(n) = \Theta(g(n)) = \Theta(n^2)$.

قضیه اصلی برای همه حالات قابل استفاده نیست. برای مثال فرض کنید

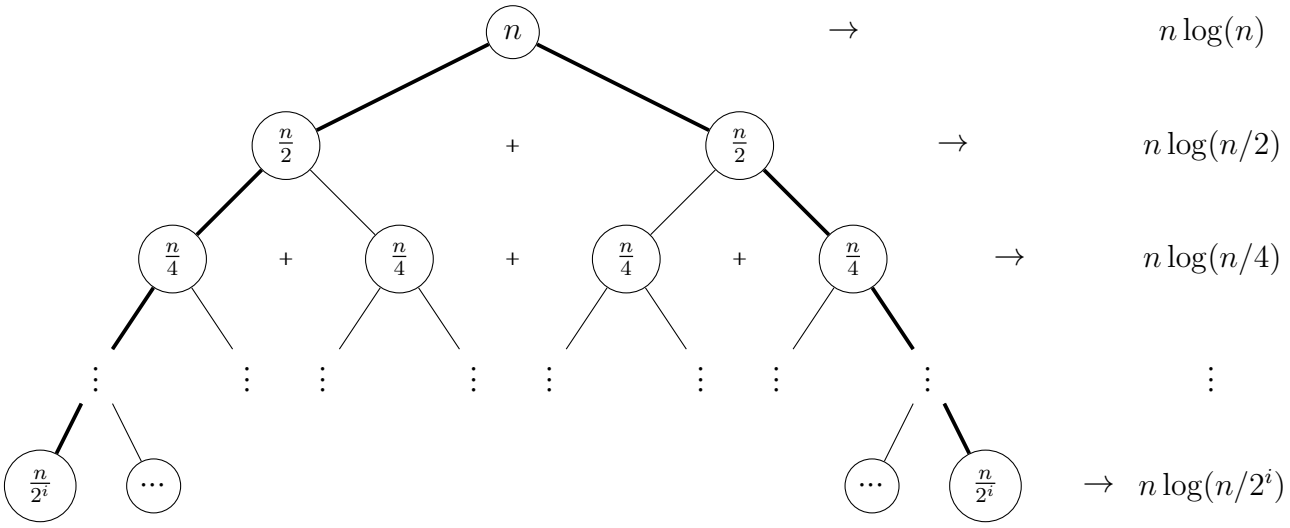
$$T(n) = 2T(n/2) + n \log n$$

در این حالت $f(n) = n \log n$ و $g(n) = n$. هیچکدام از حالات مورد نظر قضیه برقرار نیست.

$$\begin{cases} f(n) \neq O(g(n)^{1-\epsilon}) \\ f(n) \neq \Omega(g(n)^{1+\epsilon}) \\ f(n) \neq \Theta(g(n)) \end{cases}$$

دقت کنید که در اینجا ϵ باید یک ثابت مثبت غیر صفر باشد.

برای حل رابطه بازگشتی بالا می‌توانیم از درخت بازگشت استفاده کنیم.



با استقرا می‌توان نشان داد که $T(n)$ صعودی است. پس T را فقط برای توانهای 2 حل می‌کنیم.

$$T(n) = \sum_{i=0}^{\log n} n \log(n/2^i) = n \sum_{i=0}^{\log n} \log(n/2^i) = n \log n(\log n + 1) - n \sum_{i=0}^{\log n} i = \Theta(n \log^2 n)$$