مسائل و نکاتی در مورد تحلیل زمان اجرای الگوریتمها

۱ مقدمه

بطور کلی تحلیل الگوریتمها با هدف پیش بینی زمان اجرای الگوریتم و حافظه مصرفی آن انجام می‌شود. 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، اندازه داده ورودی و پیچیدگی الگوریتم طراحی شده. در این میان دو عامل اندازه داده ورودی و پیچیدگی الگوریتم بیشترین تاثیر را در زمان اجرا دارند. روشن است هر چه قدر اندازه داده ورودی بیشتر باشد زمان اجرای الگوریتم هم بیشتر خواهد شد. 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 شکل مرتب سازی یک آرایه حاوی ۱۰۰۰۰ عدد به زمان بیشتری نسبت به مرتب سازی یک آرایه حاوی ۵۰۰ عدد نیاز داشته باشد. از طرفی دیگر، پیچیدگی و ساختار الگوریتم مورد استفاده نیز یک عامل اساسی در تعیین زمان اجراست. بطور کلی تعداد دستورالعملهایی که یک الگوریتم از ابتدا تا انتها اجرا می‌کند یک معیار خوب برای تحلیل زمان اجرای الگوریتمهاست. (در این درس، و بطور معمول در تحلیل الگوریتمها، اعمال محاسباتی اصلی مثل جمع و ضرب و مقایسه و انتصاب یکسان فرض می‌شود و یک واحد برای آن در نظر گرفته می‌شود.) در زیر به بررسی چند مثال می‌پردازیم.

چند تمرین

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

```
1.  for i=1 to m
2.    for j=1 to n
3.        count ++;
```

حل: $\sum_{i=1}^{m}\sum_{j=1}^{n} 1 = \sum_{i=1}^{m} n = mn$

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=1 to m
2.    for j=1 to i
3.        count ++;
```

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 ++; }
```

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 ++; }
```

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(int i=1; i < n ; i++)
2.   for(int j=i+1; j<=n ; j++)
3.     for(int k=1 ; k<=j ; k++)
4.       count++;
```

حل:

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

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

```
1. for(int i=n; i >0 ; i--)
2.   for(int j=1; j<n ; j*=2)
3.     for(int k=0 ; k<j ; k++)
4.       count++;
```

حل: حلقه for اول کاملا مستقل از دو حلقه for دیگر است. به همین خاطر کافیست تعداد اجرای خط چهارم را با در نظر گرفتن تنها دو حلقه for دوم و سوم بدست آوریم و حاصل را در n ضرب کنیم. برای $n = 1$ دستور چهارم در قطعه کد زیر اصلا اجرا نمیشود، در غیر اینصورت این دستور ابتدا ۱ بار، بعد ۲ بار، بعد ۴ بار، بعد ۸ بار، و به همین ترتیب در مرحله آخر به تعداد $2^{\lceil \log n \rceil -1}$ بار اجرا میشود.

```
2. for(int j=1; j<n ; j*=2)
3.   for(int k=0 ; k<j ; k++)
4.     count++;
```

پس در کل در قطعه کد بالا دستور خط چهارم برای $n > 1$ به تعداد $\sum_{k=0}^{\lceil \log n \rceil -1} 2^k = 2^{\lceil \log n \rceil} - 1$ اجرا میشود. در نتیجه در قطعه کد اصلی، خط چهارم برای $n > 1$ به تعداد $n(2^{\lceil \log n \rceil} - 1)$ بار اجرا میشود.

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 با دانستن n، در هر اجرای الگوریتم فارغ از اینکه محتوای داده ورودی چه باشد، الگوریتم به تعداد مشخصی (که تابعی از n است) 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. 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 الگوریتم A 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 میسر نیست اما می‌توان کران بالا و پایین برای آن پیدا کرد. در این نوشته از نماد $T(n)$ برای اشاره به پیچیدگی زمانی الگوریتمها استفاده می‌شود.

- در بعضی جاها پیچیدگی زمانی الگوریتم را تنها بر اساس تعداد اجرای یک دستورالعمل خاص بیان می‌کنند. 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 ورودی p باشد، اندازه ورودی را $\log p$ در نظر می‌گیرند که برابر با تعداد بیتهای لازم برای نمایش ورودی مسئله است.

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

- **مرتب سازی حبابی** [1]

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.

```
Bubble-Sort(A)
 1. n = length(A)
 2. flag = TRUE
 3. for i=1 to n-1
 4.  {for  j=1 to n-i-1
 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.    otherwise flag = TRUE}
```

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

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.

[1] bubble sort

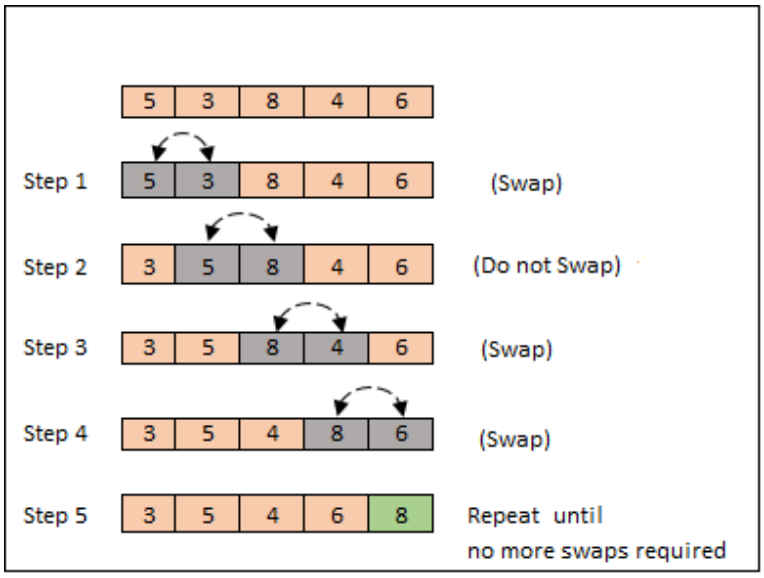

Figure ۱: نمونه‌ای از اجرای مرتب سازی حبابی در اولین پیمایش آرایه

- **مرتب سازی درجی** [۲]

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.

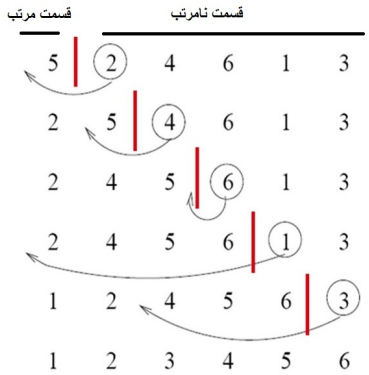

Figure ۲: مثالی از مرتب سازی درجی

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

[۲] insertion sort

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.

```
1. Insertion-Sort(A,n)
2.     for k = 2 to n
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 جستجوی دودویی

جستجوی دودویی یا binary search الگوریتمی برای تشخیص وجود یک عنصر در یک آرایه مرتب است. در صورت وجود، الگوریتم مقدار $true$ را برمی‌گرداند. اگر q در آرایه نباشد الگوریتم مقدار $false$ را برمی‌گرداند.

برای پیاده‌سازی جستجوی دودویی از سه متغیر با نامهای L و R و mid استفاده می‌کنیم. در هر مرحله L و R دو سر بازه مورد جستجو را نشان می‌دهند و mid وسط دو بازه را نشان می‌دهد. هر بار عدد مورد جستجو یعنی q با $A[mid]$ مقایسه می‌شود و در صورت نیاز مقادیر L و R بروز می‌شوند و جستجو در بازه جدید ادامه پیدا می‌کند.

به بیان دقیقتر، برای پیدا کردن q در زیر آرایه $A[L,R]$ ابتدا q با عنصر وسط زیر آرایه در محل

$$mid = L + \lfloor (R-L)/2 \rfloor$$

مقایسه می‌شود. اگر تساوی برقرار بود q پیدا شده است و جستجو خاتمه می‌یابد. اگر $A[mid] > q$ جستجو در بازه $[L, mid-1]$ ادامه می‌یابد در غیر این صورت جستجو در بازه $[mid+1, R]$ دنبال می‌شود. رخداد $L > R$ به این معنی است که q پیدا نشده است و الگوریتم مقدار $false$ را برمی‌گرداند.

```
// Binary Search
// Searcching for q
// Input array A is sorted. A[0] <= A[1] <= A[2] <= ... <= A[n-1]

1.     L = 0
2.     R = n-1
3.     While (L <= R){
3.1.       mid = L + (R-L)/2;
3.2.       if (A[mid] == q) print ''found''. exit
3.3.       if (A[mid] > q)
                  R = mid-1
            else
                  L = mid+1
         }
4.      print ''not found''. exit
```

تحلیل زمان اجرای الگوریتم:

لم. اگر $T(n)$ تعداد دستورالعملهای الگوریتم binary_search باشد که در بدترین حالت اجرا می‌شوند، آنگاه $T(n) = 5\log n + 10$.

اثبات: فعلا دو انتصاب خطوط .1 و .2 را در نظر نمی‌گیریم. در هر بار اجرای حلقه while چه q پیدا شود یا نشود به تعداد 5 دستوالعمل اجرا می‌شود. همچنین طول بازه مورد جستجو اگر n باشد در اجرای بعدی حلقه $\lfloor n/2 \rfloor$ و یا $\lfloor n/2 \rfloor - 1$ خواهد بود. (کدام حالت اتفاق می‌افتد، بستگی به این دارد که n فرد یا زوج باشد، و اینکه q از عنصر محل mid بزرگتر و یا کوچکتر باشد). پس اگر $T(n)$ ماکزیمم تعداد دستورالعملهایی باشد که اجرا می‌شود رابطه بازگشتی زیر را برای $T(n)$ می‌توانیم بنویسیم.

$$T(n) = \begin{cases} 5 + \max\{T(\lfloor n/2 \rfloor), T(\lfloor n/2 \rfloor - 1)\} & n \geq 1 \\ 3 & n = 0 \end{cases}$$

حالت $n = 0$ وقتی رخ می‌دهد که $L > R$. در این حالت طول بازه مورد جستجو صفر است. در اینجا سه دستورالعمل اجرا می‌شود (شرط حلقه while و دستورات خط .4).

یک نکته مهم: $T(i) \leq T(j)$ برای $i \leq j$. (اثبات این را به عنوان تمرین انجام دهید.)
پس می‌توانیم رابطه بازگشتی بالا را بصورت زیر ساده کنیم.

$$T(n) = \begin{cases} 5 + T(\lfloor n/2 \rfloor) & n \geq 1 \\ 3 & n = 0 \end{cases}$$

بدترین حالت زمانی رخ می‌دهد که n توانی از 2 باشد و عدد مورد جستجو از همه عناصر آرایه بزرگتر باشد. در این حالت هر بار طول بازه جدید دقیقا $n/2$ خواهد بود. به بیان دقیقتر، زمانی که $n = 2^k$ و $q > \max A$، طول بازه جدید 2^{k-1} خواهد بود. پس در کل به تعداد $5(k+1) + 3$ دستورالعمل اجرا می‌شود که همان $5\log n + 8$ است. در پایان، با در نظر گرفتن دو انتصاب شروع الگوریتم، در بدترین حالت تعداد دستورالعملها در حین اجرا $5\log n + 10$ خواهد بود. □

۳ 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 $T(n)$ پیچیدگی زمانی الگوریتم A باشد ممکن است ثابت شود که

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

گذشته از این، با توجه به اینکه ما الگوریتمها را در سطحی بالا مورد بررسی قرار می‌دهیم (توصیف الگوریتم با زبانی سطح بالا نوشته شده است) تعریف دقیقی برای واحد دستورالعمل وجود ندارد. برای مثال ممکن است کسی جابجایی دو عنصر آرایه (swap) را یک دستورالعمل حساب کند در حالیکه دیگری برای آن سه دستورالعمل در نظر بگیرد. با این اوصاف، برای مقایسه الگوریتمها از لحاظ زمان اجرا به معیاری نیاز داریم که نسبت به ضرایب ثابت حساس نباشد. یک معیار خوب در این راستا، کران مجانبی است که در زیر به تعریف آن می‌پردازیم.

برای طبقه بندی توابع از لحاظ رشد از نماد O استفاده می‌شود. نماد O وقتی به کار می‌رود که بخواهیم کران بالایی را برای یک تابع معرفی کنیم.

Definition 3.1. $O(g(n)) = \{f(n) : \exists\, c, n_0 > 0 \text{ such that } \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$

$O(g(n))$ 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 3.2. $\Omega(g(n)) = \{f(n) : \exists\, c, n_0 > 0 \text{ such that } \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$

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 تابع f آهنگ رشدی متناسب با تابع g دارد از نماد Θ استفاده می‌شود. تعریف رسمی زیر را برای این نماد داریم.

Definition 3.3. $\Theta(g(n)) = \{f(n) : \exists\, c_1, c_2, n_0 > 0 \text{ such that } \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

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 الگوریتمهای مرتب سازی حبابی و درجی که در بخش قبل توصیف شد داریم $T(n) \in \Theta(n^2)$. این بدین معناست که پیچیدگی زمانی الگوریتم مرتب سازی حبابی (یا درجی) متناسب با تابع n^2 رشد می‌کند یا اصطلاحا آهنگ رشد n^2 را دارد. بعضی اوقات از عبارت $T(n) = \Theta(n^2)$ استفاده می‌شود که همان معنای بالا را دارد.

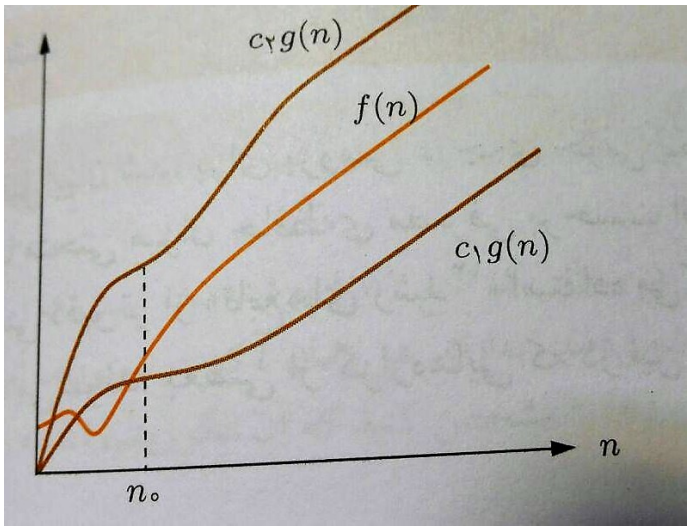

Figure ۳: نمایشی از تابع رشد

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

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 $\forall n \geq n_0, \quad cn^3 \not\leq 100n^2 + 5n - 10$

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 ـ 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

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