دانلود جزوات دانشگاهی رشته کامپیوتر

با مدیریت سمیه عرب باقری

فصل اول:

پيچيدگي زماني ومرتبه اجرايي

فصل اول
پیچیدگی زمانی و مرتبه اجرایی

پیچیدگی زمانی (Time Complexity)

در ارزیابی یک الگوریتم دو فاکتور مهمی که باید مورد توجه قرار گیرد یکی حافظه مصرفی و دیگری زمان مصرفی الگوریتم است. یعنی الگوریتمی بهتر است که فضا و زمان کمتری را بخواهد. البته غالباً در الگوریتم‌های این کتاب فاکتور زمان مهمتر از فضا می‌باشد. از آنجا که کامپایل برنامه فقط یکبار صورت می‌گیرد، لذا در مورد زمان، فقط زمان اجرای برنامه (T_{Run}) را درنظر گرفته و از زمان کامپایل صرف‌نظر می‌کنیم.

معمولاً بازدهی الگوریتم‌ها را برحسب زمان تحلیل می‌کنیم. در این تحلیل نمی‌خواهیم تک تک دستورات اجراءشده را شمارش کنیم زیرا تعداد دستورها به نوع زبان برنامه‌نویسی و نحوه نوشتن برنامه بستگی دارد. در عوض به میزانی نیاز داریم که مستقل از کامپیوتر و زبان برنامه‌نویسی باشد. به طور کلی زمان اجرای یک الگوریتم با افزایش اندازهٔ ورودی (n) زیاد می‌شود و زمان اجراء با تعداد دفعاتی که عملیات اصلی انجام می‌شود تناسب دارد. بنابراین بازدهی الگوریتم را با تعیین تعداد دفعاتی که یک عمل اصلی انجام می‌شود، به عنوان تابعی از ورودی تحلیل می‌کنیم.

مثال ۱ : تابع زیر جمع عناصر یک آرایه را در زبان C محاسبه می‌کند.

```
float  sum (float list[ ], int  n)
{
  float  s=0 ;       int i;
    for (i = 0;  i<n;  i++)
            s = s + list[i];
      return    s;
  }
```

در این برنامه اندازه ورودی همان n یا تعداد عناصر آرایه است و عمل اصلی s=s+list[i]; می‌باشد که n بار انجام می‌گیرد.

پس از تعیین اندازه ورودی یک دستور یا گروهی از دستورها را انتخاب می کنیم به طوری که کل کار انجام شده توسط الگوریتم تقریباً متناسب با تعداد دفعاتی باشد که توسط این دستور یا گروه دستورها انجام می‌شوند. این دستور یا گروه دستورها را «عمل اصلی» در الگوریتم می نامند.

به طور کلی تحلیل پیچیدگی زمانی یک الگوریتم عبارت از تعیین تعداد دفعاتی است که عمل اصلی به ازاء هر مقدار از اندازه ورودی انجام می شود. در واقع هیچ قاعده صریحی برای انتخاب عمل اصلی وجود ندارد و این کار معمولاً با تجربه و داوری درست صورت می پذیرد.

معمولاً پیچیدگی زمانی الگوریتم را با T(n) نمایش می‌دهند. در مثال فوق اگر عمل اصلی را دستور s=s+list[i]; فرض کنیم T(n) = n خواهد بود.

مثال ۲ : تعداد کل مراحل برنامه مثال ۱ را محاسبه کنید.

```
float  sum (float  list[ ], int  n)          0
{                                            0
  float  s = 0;                              1
  int      i;                                0
  for (i = 0;  i<n; i++)                     n+1
       s = s + list[i];                      n
  return  s;                                 1
}                                            0
                                            2n+3
```

در مثال فوق زمان اجرای هر عبارت ساده را مساوی ۱ واحد زمانی فرض می‌کنیم. عبارت ساده شامل زیر برنامه نمی‌باشد. یک عبارت ساده می‌تواند به اندازه یک دستور x = 2; کوچک باشد و یا مشابه دستور زیر طولانی، در هر حال هر دو را 1 واحد زمانی می‌گیریم :

```
x = 5 * y + 6 * a - 5 / w ;
```

توجه کنید } و { و نیز خط اول تعریف تابع و تعریف متغیر دستوراتی نیستند که توسط CPU اجراء شوند پس مرحله اجرائی آنها صفر است. در خط float s = 0; چون عدد صفر در s ریخته می‌شود پس یک مرحله می‌باشد. همچنین توجه کنید دستور داخل حلقه n بار انجام می‌شود ولی آزمایش کردن شرط حلقه در خط for به تعداد n + 1 بار صورت می‌گیرد. دستور return نیز توسط CPU باید اجراء شود. همانطور که قبلاً گفتیم اگر عمل اصلی را فقط خط ;s =s+list[i] فرض کنیم آنگاه T(n)=n خواهد بود. پس توجه داشته باشید که گاهی اوقات فقط می‌خواهیم بدانیم یک دستور ویژه چند بار تکرار می‌شود و گاهی اوقات نیز تعداد کل مراحل یا گام‌های برنامه را می‌خواهیم. البته عموماً تنها تعداد اجرای عمل اصلی مدنظر قرار می‌گیرد.

نکته : هنگام محاسبه تعداد دفعاتی که یک دستور درون حلقه‌ها اجراء می‌گردد می‌توان از فرمولهای زیر استفاده کرد :

$\sum_{i=1}^{n} 1 = n$ $\qquad$ $\sum_{i=1}^{n} kf(i) = k\sum_{i=1}^{n} f(i)$ $\qquad$ K عدد ثابتی است.

$\sum_{i=1}^{n} i = 1+2+3+\ldots+n = \frac{n(n+1)}{2}$ $\qquad$ $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

جمع تصاعد عددی $= \frac{n(a_1 + a_n)}{2}$: a_1 جمله اول، a_n جمله آخر و n تعداد جملات است

راه دیگر Trace کردن برنامه است.

مثال ۳ : دستور اصلی ;x: = x+1 در تکه برنامه زیر چند بار اجرا می‌شود؟ تعداد کل گامهای برنامه چقدر است؟

```
for i: = 1 to m do
   for j: = 1 to n do
         x: = x+1;
```

راه‌حل اول : حلقه‌های داده شده مستقل از یکدیگرند بنابراین طبق آنچه در زبانهای برنامه‌نویسی پاسکال و C خوانده‌اید تعداد اجرای دستور درون حلقه‌ها برابر mn می‌باشد.

راه حل دوم : تعداد اجرای دستور اصلی $= \sum_{i=1}^{m}\sum_{j=1}^{n}1=\sum_{i=1}^{m}n=n\left(\sum_{i=1}^{m}1\right)=nm$

حال برای محاسبه تعداد کل گام‌های برنامه ابتدا حلقه بیرونی i را کنار گذاشته و در نظر نمی‌گیریم. در این حال دستور ;x: = x+1 به تعداد n بار و عبارت for j به تعداد (n+1) بار اجرا می‌گردد. یعنی تعداد کل (n+1+n). حال خود این ۲ خط درون حلقه i بوده و به تعداد m بار اجرا می‌شوند یعنی m(n+1)+mn. از آنجا که عبارت for i نیز m+1 بار اجرا می‌شود، پس :

تعداد کل مراحل برنامه = (m + 1) + m(n + 1) + mn

مثال ٤ : دستور اصلی ;x: = x+1 در تکه برنامه زیر چند بار اجرا می‌شود؟

```
for j:= 1 to n do
   for i: = 1 to j do
         x: = x+1;
```

راه حل اول : حلقه‌های داده شده به یکدیگر وابسته‌اند :

j	تغییرات i	تعداد اجرا شدن دستور اصلی
1	1	1 بار
2	1,2	2 بار
3	1,2,3	3 بار
.....		
n	1,2,3,...,n	n بار

تعداد اجرا شدن دستور ;x: = x +1 $= 1+2+3+\ldots+n=\frac{n(n+1)}{2}$

راه خل دوم : $\sum_{j=1}^{n}\sum_{i=1}^{j}1=\sum_{j=1}^{n}j=\frac{n(n+1)}{2}$

مثال ٥ : تعداد اجرا شدن دستور اصلی x: = x+1 در تکه برنامه زیر چیست؟

```
i:=n;
while (i>1) do begin
        x:=x+1;
        i: = i div 2;
end;
```

اگر n = 16 باشد :

i	شرط i >1	تعداد اجرا شدن دستور اصلی
16	درست	۱ بار
8	درست	۱ بار
4	درست	۱ بار
2	درست	۱ بار
1	غلط	-
		جمعاً ٤ بار

حال اگر n را برابر 14 فرض کنیم :

i	شرط i >1	تعداد اجرا شدن دستور اصلی
14	درست	۱ بار
7	درست	۱ بار
3	درست	۱ بار
1	غلط	-
		جمعاً ۳ بار

پس در حالت کلی دستور اصلی به تعداد $\lfloor \log_2^n \rfloor$ بار اجرا می‌شود.

یادآوری : کف یک عدد اعشاری :

$$\lfloor 3.7 \rfloor = \lfloor 3.2 \rfloor = 3$$

سقف یک عدد اعشاری :

$$\lceil 3.7 \rceil = \lceil 3.2 \rceil = 4$$

کف و سقف یک عدد صحیح با خود عدد برابر است :

$$\lfloor 3 \rfloor = \lceil 3 \rceil = 3$$

تذکر : اغلب در کتاب‌های ساختمان داده‌ها منظور از Log n عبارت Log_2^n می‌باشد.

تحلیل پیچیدگی زمانی برای حالات بهترین، بدترین و متوسط

برخی مسائل برای همه موارد یک تابع پیچیدگی دارند مثل الگوریتم جمع عناصر یک آرایه :

```
A : Array [1 .. n]  of Integer;
S := 0;
For I := 1  To  n  do
   S := S + A[i] ;
```

در برنامه فوق عمل اصلی S := S + A[i] به تعداد n بار اجرا شده و همواره T(n) = n می‌باشد. ولی در الگوریتمی مثل جستجوی خطی (ترتیبی) تابع پیچیدگی برای حالات مختلف ممکن است متفاوت باشد. فرض کنید در آرایه n خانه‌ای A می‌خواهیم خانه به خانه از اول تا انتها به دنبال عدد معین x بگردیم. اگر x را در آرایه A پیدا کردیم بگوئیم Yes و در غیر این صورت بگوئیم No :

```
A : Array [1 .. n] of Integer;
For i := 1  to n do
    if (x = A[i]) {
          write ('Yes');
            exit ( ) ; → خروج از برنامه
    }
write ('No');
```

در برنامه فوق عمل اصلی شرط if (x = A[i]) می‌باشد.

در بدترین حالت عدد x ، در خانه آخر قرار دارد و یا اصلاً در آرایه نیست که در این حالت باید n بار عمل اصلی آزمایش کردن، انجام گیرد. برای بدترین حالت به جای نماد T(n) از نماد W(n) استفاده می‌کنیم که W مخفف Worst یعنی بدترین است. پس در برنامه فوق W(n) = n می‌باشد.

در بهترین حالت عدد x در اولین خانه قرار دارد و تنها به یک عمل if مورد نیاز است. بهترین حالت را با B نمایش می‌دهیم که مخفف Best است لذا در برنامه فوق داریم : B(n) = 1

برای تحلیل برنامه فوق در حالت متوسط که آن را با A (مخفف Average) نشان می‌دهیم باید به صورت زیر عمل کنیم. ابتدا فرض می‌کنیم x در آرایه A وجود داشته باشد. بدیهی است احتمال آنکه x در خانه i ام باشد برابر $\frac{1}{n}$ است و تعداد دفعات آزمایش کردن در این حالت برابر i است. لذا :

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

پس به طور متوسط نیمی از عناصر آرایه باید جستجو شود.

حال موردی را در نظر می‌گیریم که x ممکن است اصلاً در آرایه نباشد. فرض کنید احتمال وجود x در آرایه برابر p است. پس احتمال آنکه x در یکی از خانه‌های آرایه (مثل خانه i ام) باشد برابر $\frac{p}{n}$ است. احتمال آنکه x در آرایه نباشد برابر p-1 است. اگر x در خانه i ام باشد دستور اصلی i بار اجرا می‌شود و اگر x در آرایه نباشد دستور اصلی n بار اجرا می‌شود، لذا :

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

توجه کنید که اگر p = 1 باشد همان حالت قبلی $A(n)=\frac{n+1}{2}$ بدست می‌آید و اگر $p=\frac{1}{2}$ باشد $A(n)=\frac{3n}{4}+\frac{1}{4}$ می‌شود و این بدان معناست که حدود $\frac{3}{4}$ آرایه به طور میانگین باید جستجو شود.

برای الگوریتم‌هایی که فاقد پیچیدی زمانی در هر حالت می‌باشند، اغلب تحلیل‌های بدترین و حالت متوسط را محاسبه می‌کنیم. در برخی الگوریتم‌ها مثل مرتب‌سازی که به طور مکرر برای داده‌های متفاوت ورودی اعمال می‌شوند تحلیل میانگین مناسب‌تر است. ولی مثلاً تحلیل حالت میانگین در سیستم کنترل نیروگاه هسته‌ای مناسب نیست و در این حالت تحلیل بدترین حالت مفیدتر است.

تمرین اول : تست‌های شماره ۱ تا ۲۴ با عنوان «**تست‌های پیچیدگی زمانی**» را حل کنید.

مرتبه اجرایی الگوریتم (O اُی بزرگ)

در قسمت قبلی به طور دقیق محاسبه کردیم که یک دستور اصلی دقیقاً چند بار اجرا می‌شود و یا اینکه تعداد کل مراحل برنامه چند گام است. در عمل، محاسبات دقیق فوق اغلب مشکل بوده و از طرف دیگر این محاسبه دقیق مورد نیاز نیست. در کاربردهای واقعی که سرعت کامپیوترها و نوع کامپایلر می‌تواند سرعت اجرای برنامه‌ها را چندین مرتبه کاهش یا افزایش دهد، بحث بر سر اینکه فلان دستورالعمل 1000 بار اجرا می‌شود یا 999 بار، لازم نیست.

به جای محاسبات دقیق ما به ابزاری نیاز درایم که زمان اجرای الگوریتم‌ها را به صورت حدودی و طبقه‌بندی شده نشان می‌دهد. این بحث تا حدی شبیه بحث هم‌ارزی‌ها در مسائل حد ریاضیات است. مثلاً در ریاضیات خوانده‌اید که $\lim_{n\to\infty} 5n^2 + 4n - 3$ هم‌ارز با عبارت $5n^2$ است یعنی هنگامی که n زیاد می‌شود عبارت 4n-3 در مقابل $5n^2$ قابل صرف‌نظر بوده و می‌توانیم فقط $5n^2$ را در نظر بگیریم.

مرتبه اجرایی یک الگوریتم نیز شبیه هم‌ارزی فوق است. مثلاً الگوریتمی که پیچیدگی زمانی آن $T(n) = 5n - 4$ می‌باشد از مرتبه n است که آن را با $O(n)$ نمایش می‌دهیم. یا مثلاً الگوریتمی که پیچیدگی زمانی آن $6n^2 - 3n + 2$ می‌باشد از مرتبه $O(n^2)$ است. در ادامه این مفهوم مرتبه را به صورت دقیق ریاضی تعریف می‌کنیم.

تعریف : $f(n) = O(g(n))$ می‌باشد (خوانده می‌شود "f(n) بیگ اُی (big O) g(n) است") اگر و فقط اگر به ازای مقادیر ثابت و مثبتی از C و n_0 ، f(n) برای تمامی مقادیر بزرگتر یا مساوی n_0 ، کمتر یا مساوی Cg(n) باشد یعنی :

$$f(n) = O(g(n)) \Leftrightarrow \exists\ C, n_0 > 0\ :\ \forall n \geq n_0 \quad f(n) \leq Cg(n)$$

$\exists C$ به معنی آن است که حداقل یک C وجود دارد. $\forall n$ به معنای همه مقادیر n می‌باشد.

f و g توابعی غیرمنفی می‌باشند. در این حال می‌گوئیم مرتبه اجرائی تابع f(n) ، تابع g(n) می‌باشد.

مثال ٦ : $3n + 3 = O(n)$ می‌باشد چرا که اگر $C = 4$ و $n_0 = 3$ فرض کنیم، داریم :

برای $n \geq 3$: $3n + 3 \leq 4n$

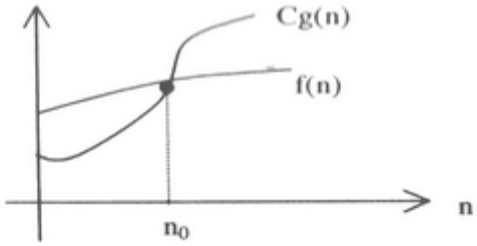
توجه کنید رابطهٔ فوق برای $n = 1$ صادق نیست ولی طبق تعریف کافی است n_0 ی وجود داشته باشد (حداقل یک n_0) که از آن به بعد $f(n) \leq Cg(n)$ باشد. در این مثال می‌توانیم n_0 را برابر 4 یا بیشتر نیز در نظر بگیریم. همچنین C را می‌توانیم 5 یا 6 یا بیشتر نیز در نظر بگیریم.

مثال ۷ : $100n + 6 = O(n)$ می‌باشد چرا که اگر $C = 101$ و $n_0 = 10$ فرض کنیم، داریم:

برای $n \geq 10$: $100n + 6 \leq 101n$

تذکر ۱ : گاهی اوقات عبارت $f(n) = O(g(n))$ را به صورت $f(n) \in O(g(n))$ نیز نمایش می‌دهند.

از نظر نموداری مفهوم اُی بزرگ به صورت زیر است :

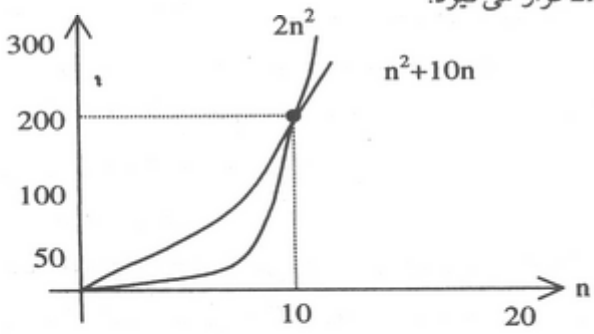


یعنی از n_0 به بعد عبارت $Cg(n)$ همواره بزرگتر از $f(n)$ می‌باشد.

مثال ۸ : $n^2 + 10n = O(n^2)$ چرا که برای $n_0 = 10$ و $C = 2$ داریم :

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

یعنی شکل نمودار $n^2 + 10n$ سرانجام در زیر تابع $2n^2$ قرار می‌گیرد.



اثبات کلی : برای $n \geq 1$ می‌دانیم که $10n \leq 10n^2$ پس :

برای $n \geq 1$: $n^2 + 10n \leq n^2 + 10n^2 = 11n^2$

پس اگر $C = 11$ و $n_0 = 1$ باشد رابطه $f(n) \leq Cg(n)$ درست خواهد بود.

پس در واقع O رفتار مجانبی تابع را توصیف می‌کند زیرا فقط با رفتار نهائی سر و کار دارد. در واقع O یک مرز بالایی مجانبی را روی تابع قرار می‌دهد.

قضیه اصلی : اگر $f(n) = a_m n^m + \ldots + a_1 n + a_0$ باشد آنگاه $f(n) = O(n^m)$ خواهد بود.

مثال ۹ :

$$f(n) = \frac{n}{2}(n-1) = \frac{1}{2}n^2 - \frac{n}{2} = O(n^2)$$

مثال ۱۰ : مرتبه اجرائی برنامه های زیر را بدست آورید :

الف) $x := x + 1$; $\Rightarrow O(1)$

ب)
```
for i:= 1 to n do
    x:= x+1;
```
$\Rightarrow O(n)$

ج)
```
for i: = 1 to n do
    for j: = 1 to n do        ⇒ O(n²)
        x: = x + 1 ;
```

O(1) زمان محاسبه ثابتی را نشان می‌دهد که تابعی از n نیست.

مثال ۱۱ : تعداد دقیق اجرا شدن دستور write('OK') و مرتبه اجرایی آن را در تکه برنامه زیر بدست آورید:

```
for i:=1 to n do
    for j:=i to n do
        write('OK');
```

حل : حلقه‌های فوق وابسته به یکدیگر بوده و همانطور که قبلاً دیدید تعداد دقیق اجرا شدن دستور write برابر جمع تصاعد عددی از 1 تا n می‌باشد یعنی $\frac{n(n+1)}{2}$ و بنابر قضیه اصلی مرتبه اجرایی آن $O(n^2)$ می‌باشد.

مثال ۱۲ : مرتبه اجرایی حلقه زیر O(1) می‌باشد چرا که حلقه به تعداد معین و محدودی اجرا می‌شود:

```
for i:=5 to 13 do
    write('OK');
```

نکته ۱ : با توجه به مثال‌های فوق می‌توان گفت ۲ حلقه for تودرتو (چه وابسته به هم باشند و چه مستقل) همواره از مرتبه $O(n^2)$ و به همین ترتیب ۳ حلقه for تودرتو (مستقل یا وابسته) از مرتبه $O(n^3)$ می‌باشد البته به شرطی که تمامی حلقه‌ها به نحوی تابعی از n باشند.

مثال ۱۳ : مرتبه اجرائی برنامه زیر چیست؟

فرض کنید n = 32 باشد آنگاه

```
x : = 0;
i : = n ;
while (i > 1) do begin
      x : = x + 1;
      i : = i div 2;
end;
```

i	x
32	1
16	2
8	3
4	4
2	5
1	-

پس برای n = 32 عمل اصلی x: = x +1; ۵ بار انجام می‌شود ($5 = \log_2 32$). پس در حالت کلی مرتبه اجرائی الگوریتم فوق برابر O(logn) می‌باشد. توجه کنید در درس ساختمان داده عموماً منظور از logn یعنی $\log_2 n$.

نکته ۲ : در حلقه while که به طور طبیعی شمارنده آن از n تا 1 تغییر می‌کند اگر مرتباً شمارنده آن با دستور i:=i div k; بر عدد k تقسیم شود مرتبه اجرایی آن $O(\log_k^n)$ خواهد بود. به همین ترتیب اگر شمارنده با دستور i:=i*k; از 1 تا n تغییر کند باز هم مرتبه اجرایی آن $O(\log_k^n)$ می‌باشد :

```
i: = n;                          i:=1;
while (i >1) {                   while(i < n) {
   عمل اصلی ;          یا          عمل اصلی ;        ⇒      O(log_k^n)
     i: = i div k;                    i:=i*k;
  }                                }
```

نکته۳ : در جدول زیر مرتبه اجرائی چند تابع به ترتیب صعودی از چپ به راست نوشته شده است:

نام تابع	ثابت	لگاریتمی	خطی	———	مرتبه ۲	توانی	فاکتوریل
مرتبه اجراء	$O(1)$	$O(\log n)$	$O(n)$	$O(n\log n)$	$O(n^2)$	$O(2^n)$	$O(n!)$

نکته ٤ : نماد O زمان اجرای الگوریتم را برای حد بالای n نشان می‌دهد و برای nهای کمتر از حد خاصی ممکن است اطلاعات مناسبی را به ما ندهد مثلاً زمان اجرای یک الگوریتم $1000n^2$ و زمان اجرای الگوریتمی دیگر $10n^3$ می‌باشد. با نماد O الگوریتم اول از مرتبه $O(n^2)$ و الگوریتم دوم از مرتبه $O(n^3)$ است. ولی برای n های کمتر از ۱۰۰ الگوریتم اول سریعتر از الگوریتم دوم خواهد بود:

$$1000n^2 \le 10n^3 \Rightarrow 100 \le n$$

لذا هنگام مقایسه دقیق سرعت اجرای الگوریتم‌ها به محدودهٔ n نیز باید توجه داشته باشیم.

نکته ٥ : برنامه‌هایی با پیچیدگی نمایی تنها برای مقادیر کوچک n (اغلب $n \le 40$) سودمند هستند.

نکته ٦ : از نظر عملی برای nهای بزرگ $(n \ge 100)$ تنها برنامه‌هایی با پیچیدگی کم (مانند n ، $n\log n$، n^2 و n^3) سودمند می‌باشند.

نکته ۷ : برای اعداد صحیح r, b, a بزرگتر از صفر، از نظر مرتبه داریم :

$$\log n < (\log n)^r < n^b < a^n < n! < n^n$$

نکته ۸ : به عنوان مثال همان‌طور که $n^2 + 10n \in O(n^2)$ می باشد، همچنین $n^2 \in O(n^2 + 10n)$

است چرا که به ازای $n > 0$ داریم : $n^2 \le 1 \times (n^2 + 10n)$ پس به ازای $C = 1$ و $n \ge n_0 = 1$ رابطه مذکور همواره صحیح است. این مثال نشان می‌دهد که تابع درون O نباید الزاماً یک تابع ساده باشد ولی عموماً آن را تابع ساده‌ای مثل $O(n^2)$ در نظر می‌گیریم.

نکته ۹ : اگر $O(n_1)$ ، t_1 و V_1 به ترتیب مرتبه اجرایی و زمان اجرایی الگوریتمی در کامپیوتری با سرعت V_1 باشند و به همین ترتیب $O(n_2)$ ، t_2 و V_2 به ترتیب مرتبه اجرایی و زمان اجرای الگوریتمی دیگر در کامپیوتری با سرعت V_2 باشند، خواهیم داشت :

$$\frac{O(n_2)}{O(n_1)} = \frac{t_2}{t_1} \times \frac{V_2}{V_1}$$

مثال ١٤ : الگوریتمی با مرتبه زمانی $O(n \log n)$ در کامپیوتری در مدت زمان ۱ ثانیه اجرا می‌شود. همان الگوریتم روی کامپیوتر دیگری با سرعت 100 برابر در چه مدت زمانی اجرا خواهد شد؟

حل :

$$\frac{O(n_2)}{O(n_1)} = 1 = \frac{t_2}{t_1} \times \frac{V_2}{V_1} = \frac{t_2}{1} \times \frac{100}{1} \Rightarrow t_2 = \frac{1}{100} = 10^{-2}\ \text{sec}$$

مثال ۱۵ : برنامه‌ای با اندازه ۱۰ و مرتبه اجرایی $O(n^2)$ روی سیستمی در مدت زمان 1 msec اجرا می‌شود. همان مسأله با اندازه 100 روی همان کامپیوتر در چه مدت زمان اجرا می‌شود؟

حل :

$$\frac{O(n_2)}{O(n_1)} = \frac{(100)^2}{(10)^2} = \frac{t_2 \times V_2}{t_1 \times V_1} = \frac{t_2}{1} \Rightarrow t_2 = 10^2 = 100\text{m sec}$$

نمادهای Ω و θ (اُی کوچک و امگای کوچک)

همانطور که قبلاً گفتیم نماد O حد بالایی را برای یک تابع مشخص می‌سازد ولی در مورد مطلوب بودن این حد چیزی را نشان نمی‌دهد. مثلاً دیدید که $3n+3 \in O(n)$ ولی در عین حال عبارت $3n+3 \in O(n^2)$ نیز درست می‌باشد چرا که مثلاً به ازای $C=4$ و $n_0=2$ همواره رابطه $3n+3 \le 4n^2$ برقرار است. به همین صورت عبارات $3n+3 \in O(n^3)$ یا $3n+3 \in O(2^n)$ نیز درست هستند. ولی در عمل منظور ما از مرتبه اجرایی $3n+3$ عبارت $O(n)$ می‌باشد. لذا تعریف O چندان دقیق نیست. به همین دلیل تعریف دقیقتری به نام θ (تتا) ارائه شده است که قبل از آن بهتر است تعریف Ω را نیز بیان کنیم.

تعریف (اُمگا) : $f(n) = \Omega(g(n))$ می‌باشد (خوانده می‌شود f(n) امگای g(n) است) اگر و فقط اگر به ازای مقادیر ثابت مثبت C و n_0 ، f(n) برای تمامی مقادیر بزرگتر یا مساوی n_0، بزرگتر یا مساوی Cg(n) باشد، یعنی:

$$f(n) = \Omega(g(n)) \Leftrightarrow \exists C, n_0 > 0 : \forall n \ge n_0 \qquad f(n) \ge Cg(n)$$

با مقایسه تعریف Ω با تعریف O درمی‌یابیم که Ω برعکس O یک مرز پائینی را برای f(n) مشخص می‌سازد.

مثال ۱۶ : $3n+2 \in \Omega(n)$ می‌باشد چرا که اگر $n_0=1$ و $C=3$ باشد آنگاه به ازای همه مقادیر $n \ge 1$ رابطه $3n+2 \ge 3n$ برقرار است.

مثال ۱۷ : نشان دهید که $5n^2 \in \Omega(n^2)$

حل : می‌دانیم که به ازای $n \ge 0$ داریم : $5n^2 \ge 1 \times n^2$

پس با در نظر گرفتن $C=1$ و $n_0=1$ رابطه فوق درست است.

مثال ۱۸ : نشان دهید که $\frac{n(n-1)}{2} \in \Omega(n^2)$

حل : به ازای $n \ge 2$ داریم $n-1 \ge \frac{n}{2}$ چرا که :

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

مثال داریم :

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

پس به ازای $C = \frac{1}{4}$ و $n_0 = 2$ رابطه فوق درست است.

قضیه : اگر $a_m > 0, f(n) = a_m n^m + \ldots + a_1 n + a_0$ باشد آنگاه $f(n) = \Omega(n^m)$ خواهد بود.

تعریف Ω نیز همان مشکل تعریف O را دارد چرا که با آنکه $10n^2 + 4n \in \Omega(n^2)$ است ولی روابط $10n^2 + 4n \in \Omega(n)$ و $10n^2 + 4n \in \Omega(1)$ نیز درست می‌باشند.

تعریف (تتا) : $f(n) = \theta(g(n))$ می‌باشد (خوانده می‌شود f(n) تتای g(n) است) اگر و فقط اگر به ازای مقادیر ثابت مثبت $C_1 g(n), C_2 g(n)$ برای تمامی مقادیر بزرگتر یا مساوی n_0 ، مابین $f(n), n_0, C_2, C_1$ است یعنی :

$$f(n) = \theta(g(n)) \Leftrightarrow \exists C_1, C_2, n_0 > 0 : \forall n \geq n_0 \qquad C_1 g(n) \leq f(n) \leq C_2 g(n)$$

مثال ۱۹ : $3n + 2 \in \theta(n)$ چرا که اگر $C_2 = 4, C_1 = 3, n_0 = 2$ باشد آنگاه به ازای همه مقادیر $n \geq 2$ رابطه $3n \leq 3n + 2 \leq 4n$ برقرار است.

قضیه : اگر $a_m > 0, f(n) = a_m n^m + \ldots + a_1 n + a_0$ باشد آنگاه $f(n) = \theta(n^m)$ خواهد بود.

نشانه‌گذاری θ از نشانه‌گذاری‌های O و Ω دقیق‌تر است چرا که :

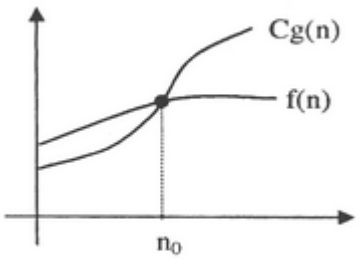
$3n + 2 \in O(n)$, $3n + 2 \in O(n^2)$

$3n + 2 \in \Omega(n)$, $3n + 2 \in \Omega(1)$

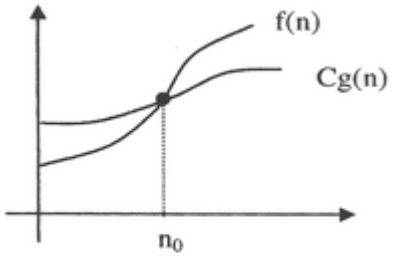
$3n + 2 \in \theta(n)$, $3n + 2 \notin \theta(n^2)$, $3n + 2 \notin \theta(1)$

تذکر : در اغلب موارد منظور از O همان θ می‌باشد.

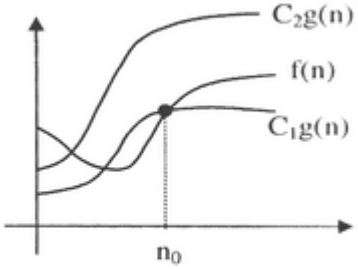
نمودارهای زیر مقایسه تعاریف O، Ω و θ را نشان می‌دهند :



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



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



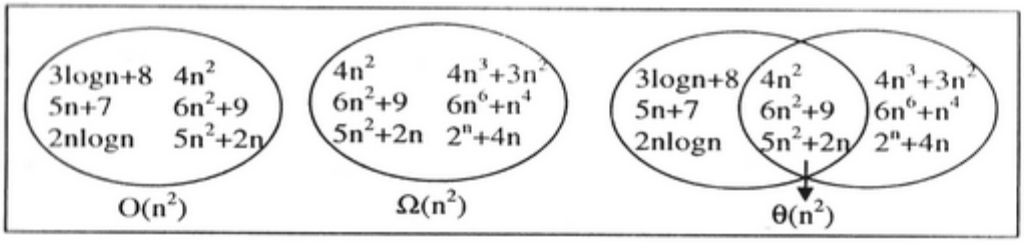
$f(n) \in \theta(g(n))$

در واقع با توجه به تعاریف فوق داریم :

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

اگر $f(n) \in \theta(g(n))$ باشد می‌گوئیم $f(n)$ ازمرتبهٔ $g(n)$ است.

مثال ۲۰ :



نکته ۱ : $g(n) \in O(f(n))$ اگر و فقط اگر $f(n) \in \Omega(g(n))$

نکته ۲ : $g(n) \in \theta(f(n))$ اگر و فقط اگر $f(n) \in \theta(g(n))$

نکته ۳ : اگر $a > 1$ و $b > 1$ باشند آنگاه $\log_a^n \in \theta(\log_b^n)$

یعنی همه توابع پیچیدگی لگاریتمی در یک دسته پیچیدگی قرار دارند. بـه عبـارتی دیگـر $\theta(\log_3^n)$ فرقـی بـا $\theta(\log_2^n)$ ندارد ولی عموماً در کتابهای ساختمان داده‌ها منظور از $\theta(\log n)$ همان $\theta(\log_2^n)$ می‌باشد.

نکته ٤ : $f(n) + g(n) \in O(\max(f(n), g(n))$

نکته ٥ : $f^2(n) \in \Omega(f(n))$

نکته ٦ : اگر $g(n) \in O(f(n))$ و $h(n) \in \theta(f(n))$ آنگاه برای $c, d > 0$

$$cg(n) + dh(n) \in \theta(f(n))$$

مثلاً $5n \in O(n^2)$ و $3n^2 \in \theta(n^2)$ پس داریم : $5n + 3n^2 \in \theta(n^2)$

مثال ۲۱ : عبارات زیر همگی درست می‌باشند :

۱) $5n^2 - 6n = \theta(n^2)$

۲) $n! = O(n^n)$

۳) $2n^2 2^n + n\log n = \theta(n^2 2^n)$

٤) $\sum_{i=0}^{n} i^2 = \theta(n^3)$

٥) $\sum_{i=0}^{n} i^3 = \theta(n^4)$

٦) $n^{2^n} + 6 \times 2^n = \theta(n^{2^n})$

۷) $n^3 + 10^6 n^2 = \theta(n^3)$

۸) $6n^3 / (\log n + 1) = O(n^3)$

۹) $n^{1.001} + n\log n = \theta(n^{1.001})$

۱۰) $10n^3 + 15n^4 + 100n^2 2^n = O(n^2 2^n)$

۱۱) $33n^3 + 4n^2 = \Omega(n^2)$

۱۲) $33n^3 + 4n^2 = \Omega(n^3)$

۱۳) $6n^2 + 20n \in O(n^3)$

مثال ۲۲ : عبارات زیر همگی نادرست هستند :

۱) $10n^2 + 9 = O(n)$ ۲) $n^2 \log n = \theta(n^2)$

۳) $n^2 / \log n = \theta(n^2)$ ۴) $3^n = O(2^n)$

۵) $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$ ۶) $6n^2 + 20n \in \Omega(n^3)$

نمادهای o و ω (اُی کوچک و امگای کوچک)

موضوع را با یک مثال شروع می‌کنیم.

مثال ۲۳ : نشان دهید که n در $\Omega(n^2)$ نیست.

حل : از روش برهان خلف می‌رویم. فرض کنید $n \in \Omega(n^2)$ باشد، در این حالت یک ثابت مثل C و یک عدد غیرمنفی n_0 وجود دارند به گونه‌ای که : $(n \geq n_0)$

$$n \geq Cn^2 \rightarrow \text{طرفین تقسیم بر } Cn \Rightarrow \frac{1}{C} \geq n$$

ولی بدیهی است که به ازای هر $n \geq n_0$ نامساوی فوق نمی‌تواند برقرار باشد. پس شرط اولیه نادرست بوده و n در $\Omega(n^2)$ نیست.

برای اینکه روابطی مانند رابطه n و n^2 را بتوانیم نشان دهیم (که مثلاً $n \notin \Omega(n^2)$) نماد o یعنی اُی کوچک را تعریف می‌کنیم.

تعریف : $f(n) = o(g(n))$ می‌باشد اگر این شرط را برآورده سازد که به ازای هر ثابت حقیقی مثبت C، یک عدد غیرمنفی n_0 وجود داشته باشد به قسمی که به ازای $n \geq n_0$ داشته باشیم : $f(n) \leq Cg(n)$.

توجه کنید که O بزرگ بدان معناست که باید حداقل یک ثابت مثبت C وجود داشته باشد ولی o کوچک می‌گوید که شرط باید به ازای هر ثابت مثبت C برقرار باشد.

مثال ۲۴ : نشان دهید که $n \in o(n^2)$

حل : برای هر $C > 0$ باید یک n_0 پیدا کنیم به قسمی که برای $n \geq n_0$ داشته باشیم $n \leq C(n^2)$ اگر طرفین نامعادله مذکور را به Cn تقسیم کنیم یعنی باید داشته باشیم $\frac{1}{C} \leq n$. پس کافی است که برای هر C مقدار n_0 را بزرگتر از $\frac{1}{C}$ بگیریم. مثلاً اگر $C = 0.01$ باشد باید n_0 را برابر 100 بگیریم و برای اعداد $n \geq 100$ داریم : $n \leq 0.01n^2$.

مثال ۲۵ : نشان دهید $n \notin o(5n)$ یعنی n در $o(5n)$ نیست.

حل : از برهان خلف استفاده می‌کنیم. فرض کنید $C = \frac{1}{6}$ باشد اگر $n \in o(5n)$ باشد، می‌بایست n_0 ی وجود داشته باشد به گونه‌ای که برای مقادیر $n \geq n_0$ داشته باشیم :

$$n \le \frac{1}{6} \times 5n \Rightarrow n \le \frac{5}{6} n$$

در صورتی که رابطه فوق همواره غلط است پس فرض اولیه نادرست بوده و n در $o(5n)$ نیست.

قضیه :

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

$$g(n) \in o(f(n)) \Rightarrow g(n) \in [o(f(n)) - \Omega(f(n))]$$

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

در برخی کتاب‌ها تعریف o کوچک به صورت زیر آمده است.

$f(n) = o(g(n))$ است، اگر و تنها اگر عبارت زیر برقرار باشد :

$$f(n) = o(g(n)) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

مثال ۲۶ : $3n + 2 = o(n^2)$ است چرا که :

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

مثال ۲۷ : مشابه مثال فوق می‌توان نشان داد که :

$$6 * 2^n + n^2 = o(3^n) \quad , \quad 3n + 2 = o(n \log n)$$

$$6 * 2^n + n^2 \neq o(2^n) \quad , \quad 3n + 2 \neq o(n)$$

$$6 * 2^n + n^2 = o(2^n \log n)$$

مثال ۲۸ : $5n^2 - 3n + 4 = O(n^2)$ ولی $5n^2 - 3n + 4 \neq o(n^2)$

نکته : همان‌طور که قبلاً گفتیم دسته‌های پیچیدگی معروف به ترتیب از چپ به راست عبارتند از :

$$(2 < i < k \quad , \quad 1 < a < b)$$

$$\theta(\log n), \theta(n), \theta(n \log n), \theta(n^2), \theta(n^i), \theta(n^k), \theta(a^n), \theta(b^n), \theta(n!)$$

اگر تابع $g(n)$ در طرف چپ تابع $f(n)$ باشد آنگاه $g(n) \in o(f(n))$

مثلاً برای $a < b$ داریم $a^n \in o(b^n)$ چرا که :

$$\lim_{n \to \infty} \frac{a^n}{b^n} = \lim_{n \to \infty} (\frac{a}{b})^n = 0$$

زیرا $\frac{a}{b} < 1$ می‌باشد :

برعکس o کوچک، می‌توان ω کوچک را به صورت زیر تعریف کرد :

$$f(n) = \omega(g(n)) \Leftrightarrow \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

یا :

$$f(n) = \omega(g(n)) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

مثال ۲۹ : $5n^2-3n+4=\omega(n)$ چرا که : $\lim_{n\to\infty}\frac{n}{5n^2-3n+4}=0$

توجه کنید که $5n^2-3n+4=\Omega(n^2)$ ولی $5n^2-3n+4\neq\omega(n^2)$

مقایسه خواص O، Ω، θ ، o و ω

۱ـ خاصیت تقارنی : این خاصیت را فقط θ دارد :

$$f(n)=\theta(g(n))\Leftrightarrow g(n)=\theta(f(n))$$

۲ـ خاصیت بازتابی :

$$f(n)=\theta(f(n)) \quad , \quad f(n)=\Omega(f(n)) \quad , \quad f(n)=O(f(n))$$

خاصیت بازتابی را o و ω ندارند.

۳ـ خاصیت ترانهاده تقارنی :

$$f(n)=O(g(n))\Leftrightarrow g(n)=\Omega(f(n))$$
$$f(n)=o(g(n))\Leftrightarrow g(n)=\omega(f(n))$$

٤ـ خاصیت تعدی :

$$f(n)=O(g(n)) \ , \ g(n)=O(h(n))\Rightarrow f(n)=O(h(n))$$
$$f(n)=\Omega(g(n)) \ , \ g(n)=\Omega(h(n))\Rightarrow f(n)=\Omega(h(n))$$
$$f(n)=\theta(g(n)) \ , \ g(n)=\theta(h(n))\Rightarrow f(n)=\theta(h(n))$$
$$f(n)=o(g(n)) \ , \ g(n)=o(h(n))\Rightarrow f(n)=o(h(n))$$
$$f(n)=\omega(g(n)) \ , \ g(n)=\omega(h(n))\Rightarrow f(n)=\omega(h(n))$$

تذکر : با توجه به خواص گفته شده می‌توان تشابهات مفهومی زیر را در نظر گرفت :

$$f\le g\approx f(n)=O(g(n)) \ , \ f\ge g\approx f(n)=\Omega(g(n))$$
$$f<g\approx f(n)=o(g(n)) \ , \ f>g\approx f(n)=\omega(g(n))$$
$$f=g\approx f(n)=\theta(g(n))$$

تمرین : تست‌های ۲۵ تا ۵۹ با عنوان «**مرتبه اجرایی**» را حل کنید.

مرتبه اجرایی توابع بازگشتی

از آنجا که بسیاری از الگوریتم‌های مطرح شده در این کتاب از نوع بازگشتی می‌باشند، لازم است نحوه بدست آوردن مرتبه اجرایی توابع بازگشتی را فرابگیرید. عموماً منظور از مرتبه اجرایی توابع بازگشتی آن است که این توابع در حالت کلی n، چند بار خودشان را فراخوانی می‌کنند. یعنی عمل اصلی را صدا زدن تابع می‌گیریم. روش کلی برای بدست آوردن مرتبه اجرایی اینگونه توابع آن است که ابتدا آنها را به فرم توبع ریاضی بازگشتی درآورده و سپس به کمک اصول معادلات بازگشتی (که در ریاضی گسسته مطرح می‌شود) آنها را حل کنیم. البته در موارد ساده‌ای می‌توان با ترسیم درخت بازگشتی مسأله را حل کرد.

مثال ۳۰ : مرتبه اجرایی الگوریتم بازگشتی محاسبه فاکتوریل را بدست آورید :

```
int fact (int n)
{
   if (n == 1) return 1;
     return (n*fact(n-1));
}
```

روش اول : ترسیم درخت بازگشتی : اگر مثلاً n = 4 باشد :

```
  fact(4)
    |
4*fact(3)
    |
3*fact(2)
    |
2*fact(1)
```

همانطور که مشاهده می‌شود به ازای n = 4 تابع به تعداد ۳ بار خودش را فراخوانی می‌کند که با احتساب بار اول یعنی fact(4) تابع مذکور ٤ بار فراخوانی می‌شود. پس در حالت کلی به ازای n ، تابع n بار فراخوانی می‌شود. لذا مرتبه اجرایی این الگوریتم T(n)=O(n) می‌باشد. در واقع مرتبه تعداد گره‌های درخت بازگشتی، همان مرتبه اجرایی الگوریتم است.

روش دوم : تابع پیچیدگی الگوریتم را به صورت ریاضی می‌نویسیم. اگر n = 1 باشد تنها یکبار خط if(n==1)return 1; اجرا شده و در نتیجه اگر n = 1 باشد آنگاه T(n)=1 است.

اگر n > 1 باشد آنگاه پس از آزمایش if تابع fact (n -1) صدا زده می‌شود. اگر پیچیدگی زمانی fact(n) برابر T(n) باشد پس پیچیدگی زمانی fact(n -1) برابر T(n-1) است لذا در این حالت T(n)=T(n-1)+C است. توجه کنید مقدار زمان ثابت C که محدود است مربوط به عملیات if و ضرب می‌باشد. پس داریم :

$T(n) = 1$ اگر $n = 1$

$T(n) = T(n-1) + C$ در غیر اینصورت

معادله فوق یک معادله بازگشتی است که روش‌های استانداردی برای حل آن وجود دارد از جمله روش جایگذاری:

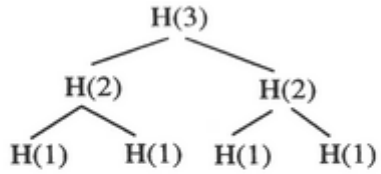
$$T(n) = T(n-1) + C = T(n-2) + 2C = T(n-3) + 3C = \ldots$$
$$= T(1) + (n-1)C = 1 + (n-1)C = O(n)$$

پس $T(n) = O(n)$ است.

مثال ۳۱ : مرتبه اجرایی الگوریتم بازگشتی برج‌های هانوی را بدست آورید:

```
void Hanoi (int n, A, B, C)
{
    if (n==1) Move a disk from A to C;
    else {
          Hanoi (n - 1, A, C, B);
          Move a disk from A to C;
          Hanoi (n-1, B, A, C);
    }
}
```

روش اول : ترسیم درخت بازگشتی. مثلاً اگر $n = 3$ باشد، تعداد کل گره‌های درخت ۷ عدد است :



به همین ترتیب اگر $n = 4$ باشد، تعداد کل گره‌های درخت بازگشتی ۱۵ می‌شود و در حالت کلی برای n تعداد کل گره‌ها 2^n -1 یعنی از مرتبه $O(2^n)$ است.

تذکر : تعداد کل گره‌های درخت دودویی پر با n سطح، از مرتبه $O(2^n)$ می‌باشد.

روش دوم : اگر $n = 1$ باشد فقط یک انتقال صورت می‌گیرد. در غیر اینصورت علاوه بر یک انتقال تابع دو بار با مقدار $n - 1$ فراخوانی می‌گردد. لذا :

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

حال با روش جایگذاری معادله بازگشتی فوق را حل می‌کنیم :

$$T(n)=2T(n-1)+1=2(2T(n-2)+1)+1=2^2T(n-2)+2+1$$
$$=2^3T(n-3)+2^2+2+1=\dots$$
$$=2^{n-1}T(1)+2^{n-2}+2^{n-3}+\dots+2+1 \qquad \left(\frac{t_1(q^n-1)}{q-1}\right) \quad \text{جمع تصاعد هندسی}$$

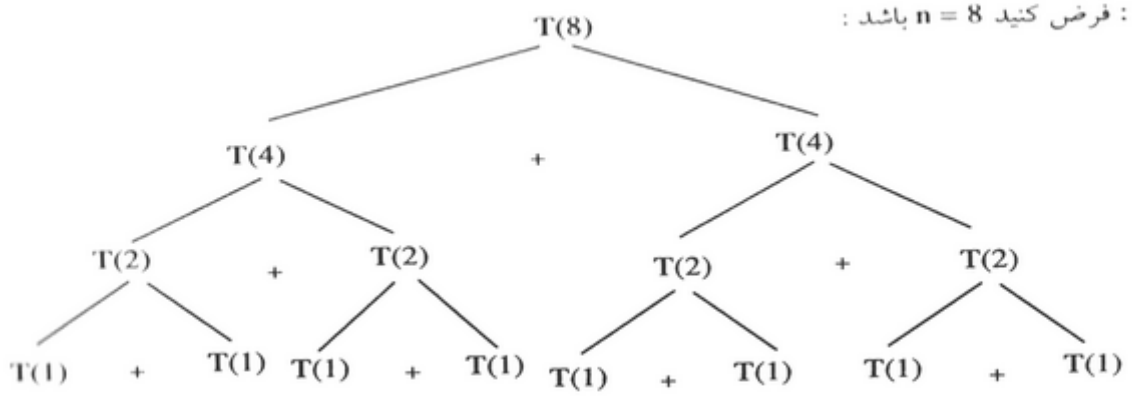
$$\frac{1\times(2^n-1)}{2-1}=2^n-1 \qquad \Rightarrow \qquad T(n)=O(2^n)$$

مثال ۳۲ : مرتبه اجرایی تابع زیر کدام است؟

```
int T(int n)
{
  if (n <= 1) return 1;
  else return T(n/2) + T(n/2) ;
}
```

۱) $O(\log n)$ ۲) $O(n)$ ۳) $O(2^{\frac{n}{2}})$ ۴) $O(2^n)$

حل : فرض کنید n = 8 باشد :



عمل اصلی همان فراخوانی است که در مثال بالا به تعداد ۱۵ بار (با احتساب فراخوانی اولیه T(8)) صورت می‌پذیرد. تعداد فراخوانی‌ها با توجه به مقادیر مختلف n برابر است با :

$T(1) = 1 \quad , \quad T(2) = 3 \, , \quad T(4) = 7 \, , \quad T(8) = 15 \quad , \quad T(16) = 31$

پس در حالت کلی $T(n) = 2n - 1$ است یعنی $T(n) = O(n)$ می‌باشد.

نکته : الگوریتم بازگشتی محاسبه ب. م. م به صورت زیر از مرتبه $O(\log_2^a)$ است :

```
int BMM (int a, int b)    (a > b > 0)
{
  if (b==0) return a;
   else return BMM (b, a mod b);
}
```

مثلاً داریم : $BMM(30,26) = BMM(26, 4) = BMM(4, 2) = BMM (2, 0) = 2$

اثبات : دنباله اعداد تولید شده توسط فراخوانی تابع فوق از چپ به راست به صورت زیر است :

$m_1, m_2, ..., m_{i-1}, m_i, m_{i+1}, ..., 0$

که در دنباله فوق $m_1 = a$ و $m_2 = b$ و $m_{i+1} = m_{i-1} \mod m_i$ است. بدیهی است برای حالت $a > b > 1$ داریم $a \bmod b < \frac{a}{2}$.

به همین ترتیب در دنباله فوق داریم $m_{i-1} \mod m_i < \frac{m_{i-1}}{2}$ می‌باشد. پس :

$$m_{i+1} = m_{i-1} \mod m_i < \frac{m_{i-1}}{2} \Rightarrow m_{i+1} < \frac{m_{i-1}}{2}$$

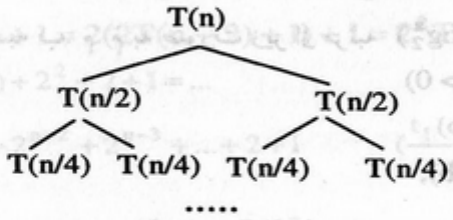
یعنی در بدترین شرایط در هر بار فراخوانی بازگشتی، پیچیدگی نصف می‌شود لذا مرتبه الگوریتم مذکور $O(\log_2 a)$ است.

توجه کنید که حل معادلات بازگشتی در حالت کلی جزو مباحث ریاضی گسسته بوده که در اینجا از ذکر آن خودداری کرده و فقط جدولی از الگوریتم‌های معروف را که عموماً با آن برخورد خواهیم کرد را آورده‌ایم. البته

با ترسیم درخت بازگشتی و پیدا کردن مرتبه تعداد گره‌های آن می‌توانید به سادگی فرمول‌های زیر را نتیجه بگیرید :

رابطه بازگشتی تابع	مرتبه اجرایی
$T(n)=T(n/2)$	$O(\log_2^n)$
$T(n)=T(n/2)+T(n/2)$	$O(2^{\log_2^n})=O(n)$
$T(n)=T(n-1)\times T(n-1)$	$O(2^n)$
$T(n)=2T(n-1)$	$O(n)$
$T(n)=T(n-2)\times T(n-2)$	$O(2^{\frac{n}{2}})$
$T(n)=T(n-1)+T(n-1)$	$O(2^n)$

به عنوان مثال برای دومین رابطه یعنی $T(n)=T(n/2)+T(n/2)$ درخت بازگشتی به صورت زیر است:



تعداد سطوح این درخت $\log_2^n$ می‌باشد و چون درخت دودویی است، مرتبه تعداد گره‌های آن برابر $O(2^{\log_2^n})$ می‌باشد که آن هم برابر $O(n)$ است.

یادآوری : $a^{\log_a^b}=b$

روش‌های برهان خلف و استقراء

در حل بسیاری از مسائل این فصل جهت اثبات فرمول‌ها می‌توان از روش‌های خلف و استقراء استفاده کرد که جهت یادآوری آنها را ذکر می‌کنیم.

برهان خلف : جهت اثبات حکم p، ابتدا فرض می‌کنیم نقیض آن یعنی $\sim p$ درست باشد. سپس به کمک قوانین و احکام اثبات شده قبلی به نتیجه‌ای برخلاف فرض اولیه یا قوانین اثبات شده قبلی می‌رسیم و نتیجه می‌گیریم که $\sim p$ نادرست است لذا p اثبات می‌شود.

استقرای ضعیف : می‌خواهیم ثابت کنیم حکم $A(n)$ به ازای $n \geq n_1$ درست است (n عدد صحیح است).
مراحل کار به صورت زیر است :

۱ـ **(پایه استقرا)** ثابت می‌کنیم $A(n)$ به ازای $n=n_1$ درست می باشد.

۲ـ **(فرض استقرا)** فرض می‌کنیم $A(n)$ به ازای عدد صحیح $k \geq n_1$ درست است.

۳ـ **(حکم استقرا)** ثابت می‌کنیم به ازای عدد $k+1$ نیز درست خواهد بود.

استقرای قوی : می‌خواهیم ثابت کنیم حکم $A(n)$ به ازای $n \geq n_1$ درست است (n عدد صحیح ا...) و روال کار به صورت زیر است :

۱ـ **(پایه استقرا)** ثابت می‌کنیم $A(n)$ به ازای $n = n_1$ درست می‌باشد.

۲ـ **(فرض استقرا)** فرض می‌کنیم حکم $A(n)$ به ازای همه مقادیر صحیح i $(n_1 \leq i < k)$ درست است.

۳ـ **(حکم استقرا)** ثابت می‌کنیم $A(n)$ به ازای عدد صحیح k نیز درست خواهد بود.

روش تقسیم و غلبه و روش پویا

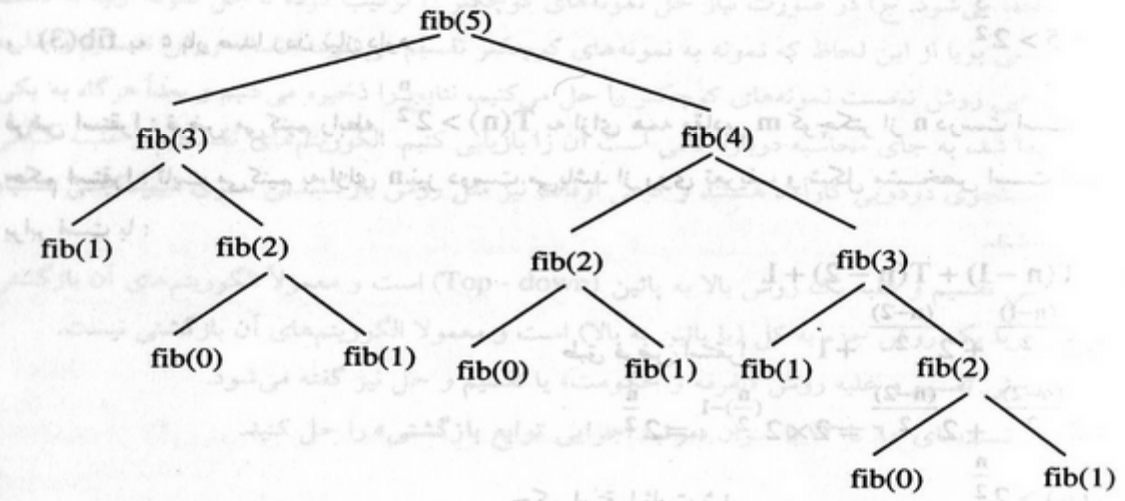
اصطلاحاً می‌گویند بعضی از الگوریتم‌ها از نوع «تقسیم و غلبه» (Divide and Conquer) یا از نوع «پویا» می‌باشند. در ابتدا با مثال فرق بین این دو دسته الگوریتم‌ها را نشان می‌دهیم.

مثال ۳۳ : تابع زیر به صورت بازگشتی جمله n ام سری فیبو ناچی را محاسبه می‌کند. در سری فیبوناچی هر جمله، جمع دو جمله قبلی خود است و دو جمله اولیه برابر "1" می‌باشند یعنی :

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

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

درخت بازگشتی این الگوریتم برای محاسبه fib(5) به صورت زیر است:



با توجه به درخت بازگشتی مشاهده می‌کنید که fib(2) سه بار و fib(3) دوبار محاسبه شده است و همین محاسبات تکراری زمان اجرای این الگوریتم را بسیار زیاد می‌کند.

برای تعیین fib(n) به ازای $6 \le n \le 0$ تعداد جملات زیر باید محاسبه شود :

n	0	1	2	3	4	5	6
تعداد جملاتی که باید محاسبه شود	1	1	3	5	9	15	25

اگر $T(n)$ تعداد جملات موجود در درخت بازگشتی فوق باشد در آن صورت اثبات می‌شود برای $n \ge 2$ ، $T(n) > 2^{n/2}$ می‌باشد یعنی مرتبه اجرای الگوریتم فوق $O(2^{n/2})$ است.

برای اثبات این موضوع به شکل فوق نگاه کنید. مثلاً fib(4) دو بار fib(2) را صدا می‌زند و fib(5) دوبار fib(3) را صدا می‌زند، پس داریم :

$$\begin{aligned} T(n) &> 2 \times T(n-2) \\ &> 2 \times 2 \times T(n-4) \\ &> 2 \times 2 \times 2 \times T(n-6) \\ &\vdots \\ &> \underbrace{2 \times 2 \times 2 \ldots \times 2}_{\text{مرتبه } \frac{n}{2}} \times T(0) \end{aligned}$$

با توجه به اینکه $T(0) = 1$ است پس $T(n) > 2^{n/2}$ می‌شود. البته برای اثبات دقیق باید از استقرا (مثلاً استقرای قوی) استفاده کنیم.

پایه استقرا : رابطه فوق به ازای $n = 2$ و $n = 3$ درست است چرا که fib(2) سه بار به صدا زدن تابع نیاز دارد:

$$T(2) = 3 > 2 = 2^{\frac{2}{2}}$$

و fib(3) به ۵ بار صدا زدن نیاز دارد :

$$T(3) = 5 > 2^{\frac{3}{2}}$$

فرض استقرا : فرض می‌کنیم رابطه $T(n) > 2^{\frac{n}{2}}$ به ازای همه مقادیر m کوچکتر از n درست است.

حکم استقرا : ثابت می‌کنیم به ازای n نیز درست می‌باشد از روی تعریف و شکل مشخص است که $T(n)$ برابر است با :

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &> 2^{\frac{(n-1)}{2}} + 2^{\frac{(n-2)}{2}} + 1 \qquad \text{طبق فرض استقرا} \\ &> 2^{\frac{(n-2)}{2}} + 2^{\frac{(n-2)}{2}} = 2 \times 2^{(\frac{n}{2})-1} = 2^{\frac{n}{2}} \\ &\Rightarrow T(n) > 2^{\frac{n}{2}} \end{aligned}$$

حکم استقرا ثابت شد.

مثال ۳٤ : حال جمله n ام سری فیبوناچی را به کمک آرایه و بدون استفاده از روش بازگشتی به دست می‌آوریم.

```
int fib2 (int n)
{
   int  i, 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];
}
```

در این الگوریتم برای تعیین fib2(n) ، n+1 جمله محاسبه می‌شود پس مرتبه اجرائی این الگوریتم O(n) می‌باشد که بسیار سریعتر از روش قبلی است.

هدف ما از این دو مثال دو چیز بوده است:

۱) با آنکه الگوریتم‌های بازگشتی در مواردی دارای مفهوم ساده‌ای هستند و به راحتی پیاده‌سازی می‌شوند و گاهی اوقات نیز دارای سرعت و کارایی بالایی می‌باشند ولی گاهی اوقات نیز مانند مثال ۳۳ دارای کارایی بسیار کمی هستند.

۲) مثال ۳۳ نمونه‌ای از الگوریتم‌های «تقسیم و غلبه» و مثال ۳٤ نمونه‌ای از الگوریتم‌های «پویا» می‌باشند. الگوریتم‌های تقسیم و غلبه شامل مراحل زیر می‌شود : الف) تقسیم نمونه‌ای از یک مسأله به یک یا چند نمونه کوچکتر ب) حل هر نمونه کوچکتر: اگر نمونه‌های کوچکتر به قدر کافی کوچک نبودند، برای این منظور از بازگشت استفاده می‌شود. ج) در صورت نیاز حل نمونه‌های کوچکتر را ترکیب کرده تا حل نمونه اولیه به دست آید. برنامه‌نویسی پویا از این لحاظ که نمونه به نمونه‌های کوچکتر تقسیم می‌شود، مشابه روش تقسیم و غلبه است ولی در این روش نخست نمونه‌های کوچکتر را حل می‌کنیم، نتایج را ذخیره می‌کنیم و بعداً هرگاه به یکی از آنها نیاز پیدا شد، به جای محاسبه دوباره کافی است آن را بازیابی کنیم. الگوریتم‌های تقسیم و غلبه گاهی اوقات مثل جستجوی دودویی کارآمد هستند و گاهی اوقات نیز مثل روش بازگشتی سری فیبوناچی بسیار ناکارآمد می‌باشند.

تذکر ۱ : روش تقسیم و غلبه یک روش بالا به پائین (Top - down) است و معمولاً الگوریتم‌های آن بازگشتی است. روش پویا یک روش جزء به کل (یا پائین به بالا) است و معمولا الگوریتم‌های آن بازگشتی نیست.

تذکر ۲ :به روش تقسیم و غلبه روش «تفرقه و حکومت» یا تقسیم و حل نیز گفته می‌شود.

تمرین سوم : تست‌های ٦۰ تا ۷۳ با عنوان **«مرتبه اجرایی توابع بازگشتی»** را حل کنید.