۱.۴ توابع هزینه

گاهی اوقات لازم است که زمان نسبی انجام یک الگوریتم را بدانیم تا میزان سریع (یا کند) بودن ان را بدست بیاوریم . از انجا که ارتقا سخت افزار یک کامپیوتر به مراتب پر هزینه تر از ارتقا نرم افزار ان است ما همواره تلاش میکنیم تا الگوریتم هایی با هزینه کمتر بسازیم تا با امکانات سخت افزاری سابق سریع تر به نتیجه برسیم.

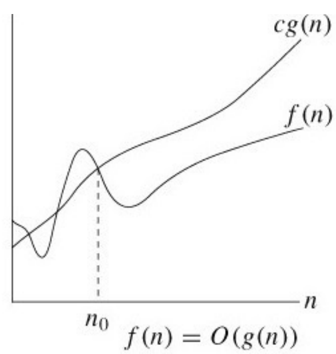
اما سوالی که مطرح میشود این است که زمان نسبی یک الگوریتم را چطور بدست بیاوریم ؟ برای این منظور تعاریف زیر را بیان میکنیم :

O(order): تخمین بالای واقعیت (سقف)

تعریف O :مجموع تمامی توابع از یک نقطه مشخص به بالا

$f(n) = O(g(n)) \iff \ni c ,> 0 \forall n > \quad , \quad f(n) \leq cg(n)$

g را سقف f میگوییم اگر بتوان c ای پیدا کرد تا تابع f به ازای همه n های بزرگ تر از n0 کمتر مساوی cg(n) باشد.

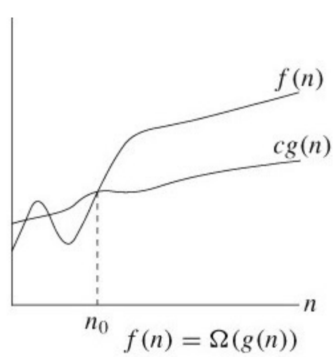


Ω (omega) : تخمین پایین واقعیت (کف)

تعریف Ω:مجموع تمامی توابع از یک نقطه مشخص به پایین

$f(n) = \Omega(g(n)) \iff \quad \ni c ,> 0 \quad \forall n > \quad , \quad f(n) \geq cg(n$

g را کف f میگوییم اگر بتوان c ای پیدا کرد تا تابع f به ازای همه n های بزرگ تر از n0 بزرگتر مساوی cg(n) باشد.

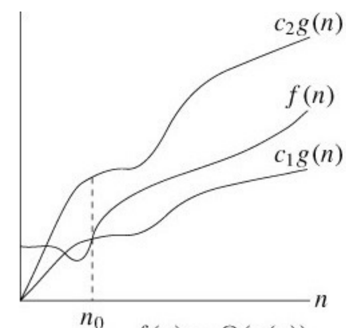


Θ (theta) : تخمین دقیق (ضرایب را تعیین نکرده ایم ولی درجه معلوم است.)

تعریف Θ:دقیق ترین تخمین است.

$$f(n) = \Theta(g(n)) \iff \quad \ni ,> 0 \quad \forall n >, \; g(n) \le f(n) \le g(n)$$

$$f(n) = O(g(n)) \;\&\; f(n) = \Omega(g(n))$$



$f(n) = \Theta(g(n))$

o: اکیدا بالای واقعیت

$$f(n) = o(g(n)) \iff \quad \ni c ,> 0 \quad \forall n > \quad , \quad f(n) < cg(n)$$

همان O است که حالت تساوی را ندارد

ω:اکیدا پایین واقعیت

$$f(n) = \omega(g(n)) \iff \quad \ni c ,> 0 \quad \forall n > \quad , \quad f(n) > cg(n$$

همان Ω است که حالت تساوی را ندارد

خواص تخمین ها:

Reflexitity :(بازتابی) O, Ω, Θ

Summetiy :(تقارنی)Θ

Transitivity همه ی تخمین ها :(تعدی)

$O(f(n)) \,.\, O(g(n)) = O(f(n) \,.\, g(n))$

$O(f(n)) + O(g(n)) = O(Max(g(n),f(n))$

If $f(n) = O(kg(n))$ for Constant $k > 0 \Rightarrow f(n) = O(g(n))$

۱.۵ آنالیز زمان اجرا

برای آنالیز هر خط و زمان اجرای آن را در نظر می گیریم.

Simple Sentences (read,write,...) : O(1)

Simple Rules(+,=,*,==,...) : O(1)

Loops(for,while,..) : O(1)

Conditional Sentences : O(Max(T(body1),T(body2))

معمولا انالیز زمان اجرای Conditional Sentences پیچیده است و ما قادر به محاسبه ان نمی باشیم.

گلوگاه کد قسمتی از کد است که بیشترین زمان اجرا را در بر می گیرد.

چند مثال:

1)

```
for i = 1 to n
for j = i to n
sum++;
```

T(n) = = = - + = - +

T(n) = Ɵ()

2)

```
for  (i = 1 ; i  < n  ;  i  *= 2)
S++;
```

T(n) = Ɵ(logn)

3)

```
for (i  = 0 ;i < n ; i++)
for(j = 1 ; j < n ; j+=j)
Sum++;
```

T(n) = = nlogn

T(n) = Θ(nlogn)

4)

```
for (i = 1 ; i < n ; i+=i)
for(j = 0 ;  j <i ; j ++)
s++;
```

T(n) =   = -   =   -1

T(n) = Θ (n)

5)

```
Int  i=n;
While(i>1){
i/=2;
j=n;
while(j > 1)
j/=3;
}
```

T(n) =   = =

T(n) = Θ()

یک مثال 1.5.1

فرض کنید ارایه ای از اعداد صحیح داشته باشیم می خواهیم بزرگ ترین زیر دنباله از این ارایه را به دست بیاوریم که مجموع اعضای ان بیشینه باشد .

زیردنباله : برای ارایه a به ازای تمام i هایی که k1<= i <=k2 اعضای a[i] عضو زیردنباله s از k1 تا k2 می باشد.

4 الگوریتم متفاوت را بررسی می کنیم.

الگوریتم اول:

بزرگ ترین زیردنباله ای که از ان شروع می شود را بدست (a) اولین راهی که به ذهن می رسد این است که به ازای هر عضو ارایه اوریم و بیشینه انها جواب مساله است

```
int Maxsum=0;
for (int i=0 ; i< a.size();i++)                    O()
      for(int j=i; j < a.size() ; j++)             O()
             int ThisSum=0;                          O(1)
            for(int k=i; k<= j ; k++)              O(j-i)
                        ThisSum+=a[k];               O(1)
          if(ThisSum > MaxSum)                     O(1)
             MaxSum=ThisSum;                       O(1)
Return MaxSum;                                     O(1)
T(n) = O()
```

الگوریتم دوم:در این الگوریتم 1 حلقه کم شده است.

در واقع همان الگوریتم اول است با این تفاوت که به ازای هر a[i] دیگر از اول تا اخر بازه را محاسبه نمی کنیم بلکه با مقدار قبلی مقایسه و جمع می کنیم.

```
int MaxSum = 0 ;
for (int i=0 ; i< a.size() ; i++){
           int ThisSum=0;
           for(int j=i; j <= a.size() ; j++){
                   ThisSum += a[j];
                   If(ThisSum > MAsSum)
                          MaxSum = ThisSum;
          }
}
```

Return MaxSum;

T(n) = O()

الگوریتم سوم: Divide & Conquer(تقسیم و غلبه)

در این قسمت مساله را به چند زیر مساله تقسیم می کنیم و به صورت بازگشتی زیرمساله ها را حل می کنیم تا به شرط خاتمه یا حالتی بدیهی برسیم . این روش یکی از کاربردی ترین راه حل های حل مساله است (یاد گیری ایده ان به شدت توصیه میشود)

برای این منظور یک تابع تعریف می کنیم که یک ارایه میگیرد و بزرگ ترین مجموع را برمی گرداند حال وسط ارایه را به دست می اوریم . جواب نیمه اول را s1 و جواب نیمه دوم را s2 در نظر میگیریم . حال تنها حالاتی باقی می ماند که ابتدا ان در نیمه اول و انتهای ان در نیمه دوم است که این کار با O(n) امکان پذیر است.

```
int MaxSubSeq(int A[] , int f , int t){
if(f==t){
if(A[f] < 0 )
return 0;
return A[f];
}
}
M = (f+t)/2;
S1=MaxSubSeq(A,f,m)                  T(n/2)
S2=MaxSubSeq(A,m+1,t)                  T(n/2)
S3=temp=0;
for (i=m;i>=f;i--)
temp += A[i];
if(temp > S3)
S3=temp;
}
S4 = temp=0;
for (i=m+1;i<=t;i++){
```

```
temp += A[i];

if(temp>S4)

S4 = temp;

}
```

S3 و S4 قسمت غلبه این الگوریتم را نشان می دهد

$T(n) = T(n/2) + T(n/2) + O(n)$

$T(n) = n$

الگوریتم چهارم:

در این الگوریتم چند نکته حائز اهمیت است.

1-جواب با عدد منفی شروع نمی شود.

2-جواب با اعداد منفی پایان نمی یابد.

3-جواب با پیشوند منفی شروع نمی شود

پس به این صورت عمل می کنیم که از سر ارایه شروع می کنیم و مجموع اعضا را بدست می اوریم هر جا مجموع منفی شد اعداد دیده شده را دور ریخته از نو شروع می کنیم

```
int MaxSum=0;

int ThisSum=0;

for(int j = 0 ; j < a.size() ; j++){

        ThisSum += a[j];

        If(ThisSum>MaxSum)

                MaxSum=ThisSum;

       else if(ThisSum< 0 )

  ThisSum=0;

}

Return MAxSum;
```

چون در کل هر عضو ارایه را یک بار دیدیم پس o(n) است

در الگوریتم اول زمان پاسخگویی به اندازه طول عمر ما ادامه پیدا خواهد کرد اما الگوریتم اول در کمتر از 1 ثانیه جواب = n برای خواهد داد.