# ساختمان داده ها
# Data Structure

## فصل اول
## توابع  بازگشتی و پیچیدگی زمانی

تهیه و تنظیم : محمد نعیمی

عضو هیات علمی دانشگاه آزاد اسلامی

# تابع بازگشتی:

**تابع بازگشتی با ساده کردن مسئله به یک یا چند مرحله قبل تر و فراخوان خودش برای حل قسمت ساده مسئله انقدر مسئله را ساده میکند تا به ساده ترین حالت خود برسد و سپس با حل ساده ترین حالت فراخوانها را به ترتیب معکوس حل کرده و پایان میدهد تا اولین فراخوان به صورت کامل حل شود.**

**مثال:** منطق بازگشتی تابع فاکتوریل به فرم رو برو است.

$$n! = \underbrace{1 * 2 * 3 * \cdots * (n-2) * (n-1)}_{(n-1)!} * n \quad \Rightarrow \quad \boxed{n! = (n-1)! * n}$$

لذا تابع فاکتوریل را میتوان بر اساس مراحل قبل ساده تر نمود

$$fact(n) = \begin{cases} n * fact(n-1) & n > 1 \\ 1 & n = 1 \end{cases}$$

برای فراخوانی تابع برای محاسبه !5 مراحل زیر رخ میدهد.

$$\boxed{120 = 5 * 24}$$

$$\text{fact}(5) = 5 * \text{fact}(4) \quad \boxed{24 = 4 * 6}$$

$$\text{fact}(4) = 4 * \text{fact}(3) \quad \boxed{6 = 3 * 2}$$

$$\text{fact}(3) = 3 * \text{fact}(2) \quad \boxed{2 = 2 * 1}$$

$$\text{fact}(2) = 2 * \text{fact}(1) \quad \boxed{1}$$

$$\text{fact}(1) = 1$$

# روش نوشتن تابع بازگشتی:

**جهت نوشتن تابع بازگشتی لازم است مراحل زیر را انجام دهیم**

**۱–یک تعریف از تابع شامل چه میگیرد و چه میکند و چه بر میگرداند برای خود بیان کنید(مثلا تابع عدد گرفته فاکتوریل بر میگرداند)**

**۲– با فرض داشتن مرحله یا مراحل قبل تر چگونه میتوان مرحله فعلی را حل نمود (مثلا با داشتن n-1!  و n-2! و ... چگونه میتوان n! را ساخت که در این حالت فقط با داشتن مرحله قبل میتوان عمل کرد n!=n*(n-1)!)**

**۳– ساده ترین حالت مسئله چیست و برای آن حالت خروجی چیست؟ (1! می شود 1)**

**۴– برنامه را با if برای حالت ۲و ۳ نوشته و فرض کنید در حالت ۲ برای قسمت های فراخوان مجدد تابع دقیقا همان کاری که در مرحله ۱ مشخص کردید برای مراحل قبل انجام می دهد.**

# تابع فاکتوریل:

۱– تابع fact(n) یک عدد گرفته فاکتوریل آن را بر می گرداند.

منطق ریاضی تابع فاکتوریل به فرم زیر است.

۲– نحوه ساده کردن یک حالت پیچیده به حالی ساده تر

۳– حالت ساده مسئله

$$fact(n) = \begin{cases} n * \overbrace{fact(n-1)}^{\text{یک مرحله قبل تر}} & n > 1 \\ 1 & n = 1 \end{cases}$$

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

در ساده کردن حالت پیچیده، فرض کنید حالات یک مرحله یا هر چند مرحله قبل تر را دارید و با توجه با آنها چگونه میتوانید حالت فعلی را بنویسید. یعنی اگر تابع برای حالت n قرار است بنویسیم فرض کنید جواب مراحل n-1 و n-2 و ... را دارید و بر اساس هر کدام لازم است جواب را بنویسید. (معمولا یک مرحله قبل تر )

## تابع محاسبه ضرب $a * b$:

فرمول بازگشتی ضرب به فرم مقابل است.
ضرب چون دو عملوند دارد باید چک کنیم با ساده کردن کدامیک میتوان مسئله را ساده کرد. در ضرب هر کدام را کاهش دهیم مسئله ساده می شود. فرم رو برو b را یک واحد کم میکند و بر اساس آن ضرب را محاسبه میکند.

یک مرحله قبل تر طبق b

$$a * b = \begin{cases} \overbrace{a * (b-1)} + a & b > 0 \\ \boxed{0} & b = 0 \end{cases}$$

باید در برنامه نویسی ضرب را به صورت تابع بنویسیم لذا a*b را به صورت mul(a,b) مینویسیم.
شکل تابع نویسی فرمول ضرب به فرم زیر است.

$$mul(a,b) = \begin{cases} mul(a, b-1) + a & b > 0 \\ \boxed{0} & b = 0 \end{cases}$$

```
int mul (int a,int b)
  {
     if(b==0)
        return (0);
     return(mul(a,b-1)+a);
  }
```

7*4=mul(7,4)
28 mul(7,4)=mul(7,3)+7
28 mul(7,3)=mul(7,2)+7
21 mul(7,2)=mul(7,1)+7
14 mul(7,1)=mul(7,0)+7
7 mul(7,0)
0

بازگشت جواب ساده به سمت حل مسائل قبلتر

تهیه و تنظیم : محمد نعیمی

# تابع محاسبه توان $a^b$ :

فرمول بازگشتی **توان** به فرم مقابل است.
**توان** چون دو عملوند دارد باید چک کنیم با ساده کردن کدامیک میتوان مسئله را ساده کرد. در توان با کم کردن پایه نمیتوان رابطه ایجاد کرد و فقط میتوان با کاهش توان به رابطه مقابل رسید.

یک مرحله قبل تر طبق b

$$a^b = \begin{cases} \underbrace{a^{b-1}} * a & b > 0 \\ 1 & b = 0 \end{cases}$$

باید در برنامه نویسی توان را به صورت تابع بنویسیم لذا $a^b$ را به صورت pow(a,b) مینویسیم.
شکل تابع نویسی فرمول ضرب به فرم زیر است.

$$\text{pow}(a,b) = \begin{cases} pow(a, \text{b}-1) * a & b > 0 \\ 1 & b = 0 \end{cases}$$

```
int pow (int a,int b)
 {
    if(b==0)
       return (1);
    return(pow(a,b-1)*a);
 }
```

$3^4$ =pow(3,4)

81 pow(3,4)=pow(3,3)*3

81 pow(3,3)=pow(3,2)*3

27 pow(3,2)=pow(3,1)*3

9 pow(3,1)=pow(3,0)*3

3 pow(3,0)

1

بازگشت جواب ساده به سمت حل مسائل قبلتر

تهیه و تنظیم : محمد نعیمی

# تابع محاسبه جمله n ام فیبوناچی:

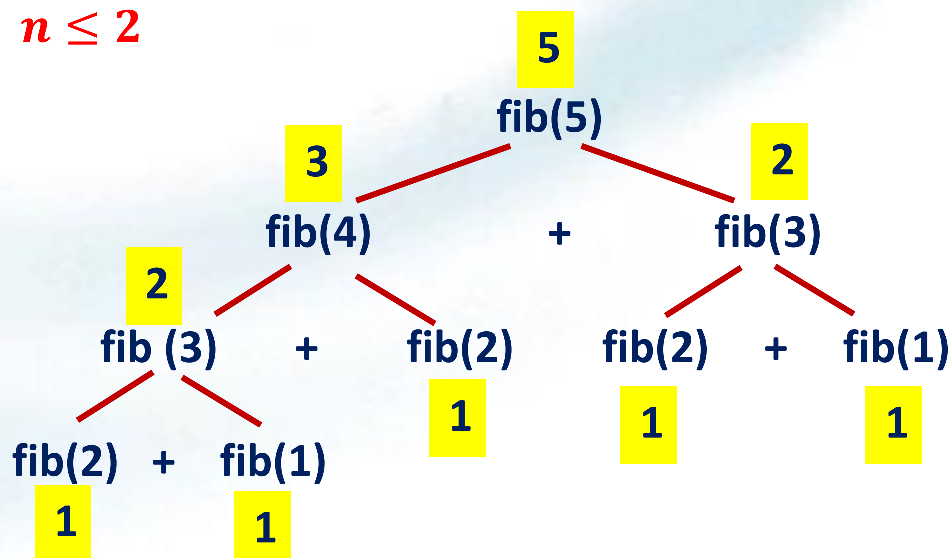
اعداد فیبوناچی 1 1 2 3 5 8 13 21 34 ...

تعریف عدد فیبوناچی به این صورت است که هر عدد از جمع دو عدد قبل محاسبه می شوند و دو عدد ابتدایی 1 هستند. لذا ساده ترین حالت مسئله عدد اول و دوم فیبوناچی است که مقدار 1 دارند و سایر اعداد میشوند حاصل دو عدد قبلتر

منطق ریاضی تابع فیبو ناچی به فرم زیراست.

$$fib(n) = \begin{cases} \underbrace{fib(n-1)}_{\text{یک مرحله قبل تر}} + \underbrace{fib(n-2)}_{\text{دو مرحله قبل تر}} & n > 2 \\ 1 & n \leq 2 \end{cases}$$

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



# تابع محاسبه خارج قسمت a div b :

منطق ریاضی تابع خارج قسمت به فرم زیراست.

خارج قسمت a div b یعنی تعداد b هایی که در a وجود دارد پس با کم کردن هر b یک واحد به خارج قسمت اضافه می شود و زمانی که عدد کمتر از b شود خارج قسمت 0 می شود.

پس خارج قسمت تقسیم a بر b وقتی عدد a بزرگتر مساوی b است می شود ۱ واحد بیشتر از خارج قسمت تقسیم a-b بر عدد b

حالت ساده مسئله هم زمانی است که a<b باشد که جواب 0 است

$$div(a,b)=\begin{cases} div(a-b,b)+1 & a\geq b \\ 0 & a<b \end{cases}$$

مثال

15 div 6 = (15-6) div 6 +1 = 9 div 6 +1

```
int div (int a,int b)
 {
    if(a<b)
        return (0);
    return(div(a − b, b) + 1);
 }
```

# تابع محاسبه باقی مانده a mod b :

منطق ریاضی تابع خارج قسمت به فرم زیراست.

باقی مانده a mod b یعنی پس از برداشتن تمام b های ممکن از داخل a چه مقدار باقی می ماند. پس با کم کردن هر b باقی مانده تغییری نمیکند تا زمانی که عدد باقی مانده از b کمتر شود و آن عدد می شود باقی مانده.

پس باقی مانده تقسیم a بر b وقتی عدد a بزرگتر مساوی b است می شود همان باقی مانده تقسیم a-b بر عدد b

حالت ساده مسئله هم زمانی است که a<b باشد که جواب a است

مثال

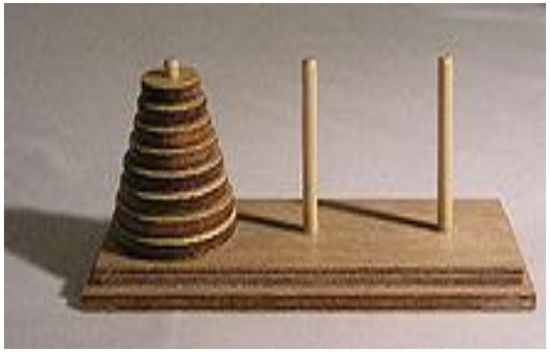
15 mod 6 = 9 mod 6

منطق ریاضی تابع باقی مانده به فرم زیراست.

$$mod(a,b)=\begin{cases} mod(a-b,b) & a\geq b \\ a & a<b \end{cases}$$

```
int mod (int a,int b)
 {
     if(a<b)
         return (a);
     return(mod(a − b, b));
  }
```

# برج هانوی:



همانند شکل سه میله داریم. یکی از میله‌ها میله مبدأ (A) حاوی n صفحه با سایز های مختلف که به صورت نزولی روی آن چیده شده است، دیگری میله کمکی (B) و دیگری میله مقصد (C) است. هدف انتقال تمام دیسک‌ها از میله مبدأ به میله مقصد با رعایت شرایط زیر است:

- در هر زمان فقط یک دیسک را می‌توان جابجا نمود.
- نباید در هیچ زمانی دیسکی بر روی دیسک با اندازه کوچکتر قرار بگیرد.

```
void hanoy (char from,char to,char help, int n)
 {
     if(n==1)
        cout<< from<<“ – “ <<to;
     else
       {
          hanoy (from,help,to, n-1);
          cout<< from<<“ – “ <<to;
          hanoy (help,to,from, n-1);
        }
}
```

# چاپ اعداد:

### تابع بازگشتی چاپ اعداد 1 تا n

```
void prt (int n)
 {
     if(n==1)
        cout<<1;
     else
      {
          prt(n-1);
          cout<<n;
      }
 }
```

### تابع بازگشتی چاپ اعداد 1 تا n از اخر به اول

```
void rec_prt (int n)
 {
     if(n==1)
        cout<<1;
     else
      {
          cout<<n;
          rec_prt(n-1);
      }
 }
```

# تمرین:

- عمل ترکیب در آمار دارای فرمول مقابل است. تابع محاسبه بازگشتی آن را بنویسید.

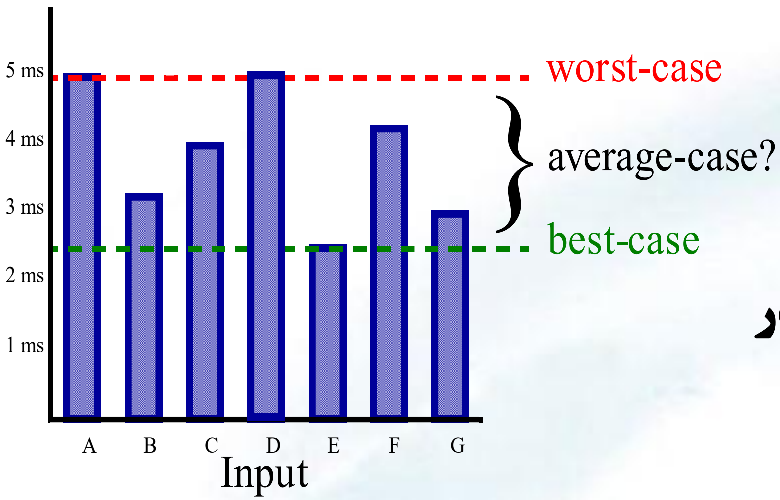
$$\binom{m}{n} = \binom{m-1}{n-1} + \binom{m-1}{n}$$

- برای محاسبه ب.م.م در روش نردبانی اگر m بر n بخشپذیر باشد ب.م.م می شود n وگرنه پاسخ می شود ب.م.م n و باقیمانده تقسیم m بر n. تابع محاسبه بازگشتی آن را بنویسید.

- تابع بازگشتی بنویسید که بر اساس فرمول مقابل محاسبات را انجام دهد.

$$\text{test}(a,b) = \begin{cases} test(a, \mathrm{b}-3) * 4 * a & b > 0 \\ 12 & b = 0 \end{cases}$$

# پیچیدگی زمانی




**پیچیدگی در بدترین حالت(Worst-case):**

**حداکثر زمان مورد نیاز برای اجرای برنامه (آهسته‌ترین اجرا در بین تمام ورودی‌ها). در بدترین حالت، ما کران بالایی زمان اجرای یک الگوریتم را محاسبه می‌کنیم.**

**پیچیدگی در حالت متوسط(Average-case):**

**میانگین زمان لازم برای اجرای برنامه. اطلاعات لازم را در مورد رفتار الگوریتم در حالت ورودی تصادفی می دهد.**

**پیچیدگی در بهترین حالت(Best-case):**

**حداقل زمان لازم برای اجرای برنامه (سریعترین اجرا در بین تمام ورودی ها). در بهترین حالت، ما کران پایین زمان اجرای یک الگوریتم را محاسبه می‌کنیم.**

**چرا پیچیدگی در بدترین حالت؟**

**پیچیدگی بدترین حالت تضمین میکند که الگوریتم هرگز بیش از این طول نخواهد کشید.**

**برای برخی از الگوریتم‌ها، بدترین حالت اغلب اتفاق می‌افتد.**

**مثال: در جستجوی یک پایگاه داده برای یک قطعه خاص از اطلاعات، بدترین حالت الگوریتم جستجو اغلب زمانی رخ می دهد که اطلاعات در پایگاه داده وجود نداشته باشد.**

**«پیچیدگی در حالت متوسط» اغلب به بدی پیچیدگی بدترین حالت است.**

# نماد های مجانبی برای نمایش پیچیدگی زمانی

برای نمایش پیچیدگی معمولا از سه نماد زیر استفاده می شود.
نمادهای مجانبی پیچیدگی زمانی را به عنوان «سریعترین زمان ممکن»، «کندترین زمان ممکن» یا «متوسط زمان» نشان می‌دهند.

❑ نماد Big O :

$f(x)$ عضو Big O تابع $\mathrm{g}(\mathrm{x})$ است اگر وجود داشته باشند دو مقدار صحیح مثبت $C$ و $n_0$ به نحوی که برای هر $n > n_0$ رابطه $f(n) \leq Cg(n)$ برقرار باشد. (رشد زمان f کمتر مساوی g)

$$\mathbf{f(x) \in O\big(g(x)\big) \Leftrightarrow \exists n_0 > 0, \mathit{C} > 0 \text{ such } \forall n > n_0 \;\; f(n) \leq \mathit{C}g(n)}$$

❑ نماد امگا Ω :

$f(x)$ عضو اومگای $\mathrm{g}(\mathrm{x})$ است اگر وجود داشته باشند دو مقدار صحیح مثبت $C$ و $n_0$ به نحوی که برای هر $n > n_0$ رابطه $f(n) \geq Cg(n)$ برقرار باشد. (رشد زمان f بیشتر مساوی g)

$$\mathbf{f(x) \in \Omega\big(g(x)\big) \Leftrightarrow \exists n_0 > 0, \mathit{C} > 0 \text{ such } \forall n > n_0 \;\; f(n) \geq \mathit{C}g(n)}$$

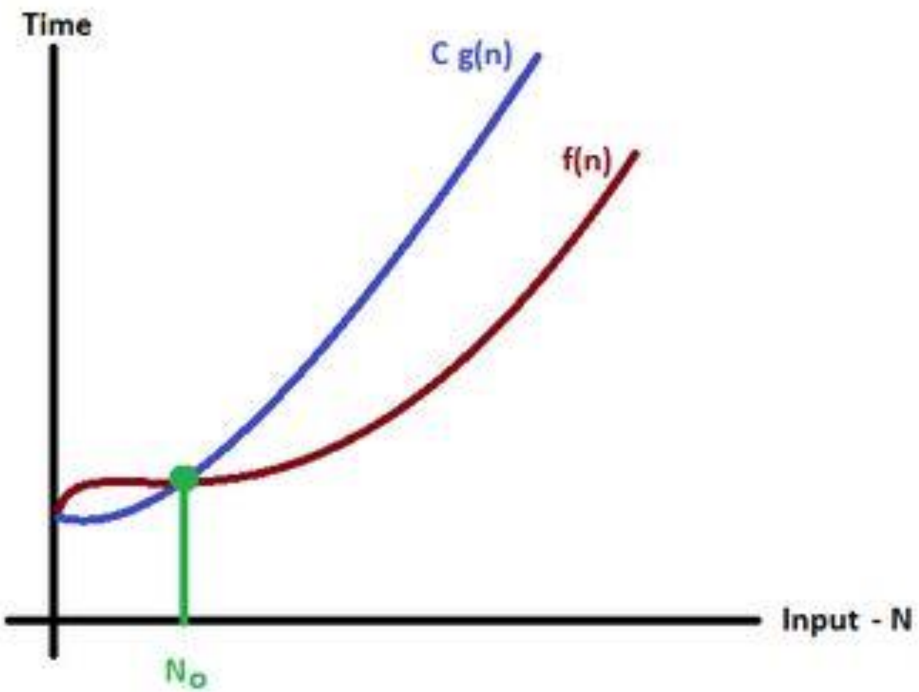
❑ نماد تتا ϴ :

$f(x)$ عضو تتای $\mathrm{g}(\mathrm{x})$ است اگر وجود داشته باشند سه مقدار صحیح مثبت $C_1$، $C_2$ و $n_0$ به نحوی که برای هر $n > n_0$ رابطه $f(n) \geq C_0 g(n)$ برقرار باشد. (رشد زمان f مساوی g)

$$\mathbf{f(x) \in \Theta\big(g(x)\big) \Leftrightarrow \exists n_0, C_1, C_2 > 0 \text{ such } \forall n > n_0 \; C_1 g(n) \leq f(n) \leq C_2 g(n)}$$

# Big O (کران بالا—بدترین حالت)

Big O شیوه رسمی برای بیان کران بالای زمان اجرای الگوریتم است.
همیشه حداکثر زمان مورد نیاز یک الگوریتم را برای همه مقادیر ورودی نشان می دهد.
این بدان معناست که نماد Big O بدترین حالت پیچیدگی زمانی الگوریتم را توصیف می کند




f(n) و g(n) زیر را در نظر بگیرید

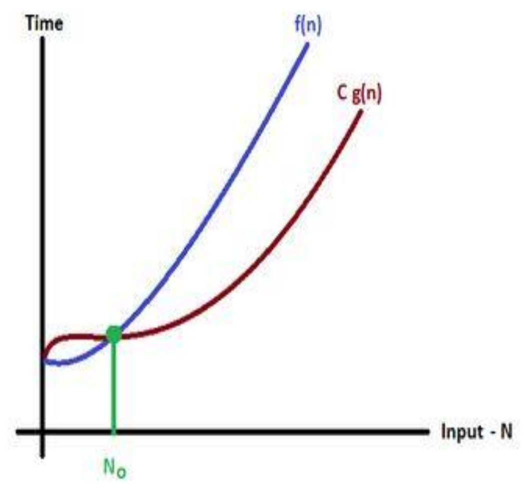
$f(n) = 3n + 2$

$g(n) = n$

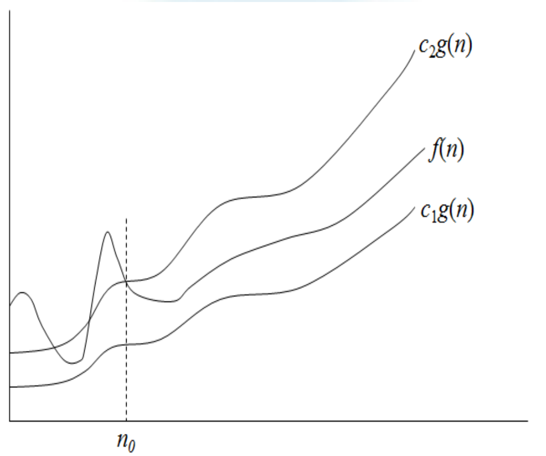
تعریف Big O برای $C = 4$ و $n_0=2$ همیشه برقرار است.

Big O به بیان ساده میگوید:
از جایی به بعد ($n_0$) تا بینهایت مقدار f(n) هرگز **بیشتر** از ضریب ثابتی (C) در g(n) نخواهد شد.
یعنی رشد f(n) از Cg(n) بیشتر نخواهد شد.

# Ω (کران پایین– بهترین حالت)– Θ(حالت متوسط)

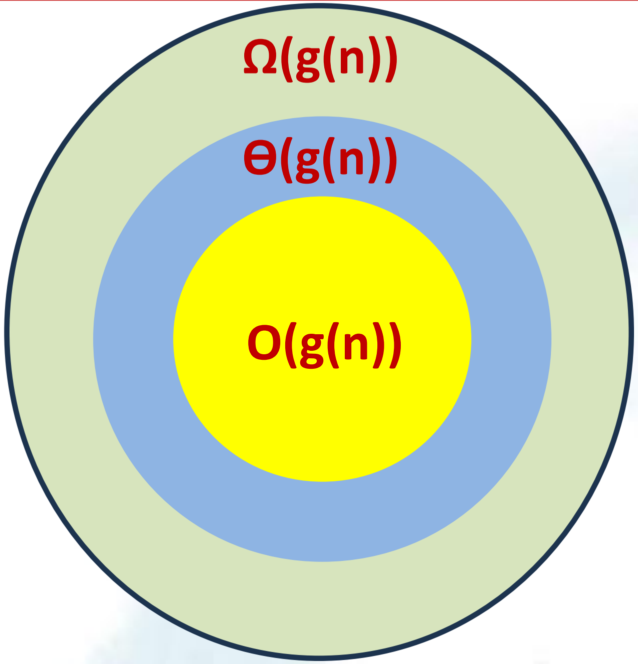



**Ω به بیان ساده میگوید:**
**از جایی به بعد ($n_0$) تا بینهایت مقدار f(n) هرگز کمتر از ضریب ثابتی (C) در g(n) نخواهد شد.**
**یعنی رشد f(n) از Cg(n) بیشتر نخواهد شد.**




**Θ به بیان ساده میگوید:**
**از جایی به بعد ($n_0$) تا بینهایت مقدار f(n) بین دو ضریب ثابت ($C_1$ و $C_2$) از g(n) خواهد بود.**
**یعنی رشد f(n) بین کران هایی ازضرایب g(n) خواهد بود.**

# مقایسه نماد های مجانبی




- O(g(n)) بیانگر مجموعه توابعی است که از نظر پیچیدگی زمانی معادل تابع g(n) است.
- Θ(g(n)) بیانگر مجموعه توابعی است که از نظر پیچیدگی زمانی معادل تابع g(n) است.
- Ω(g(n)) بیانگر مجموعه توابعی است که از نظر پیچیدگی زمانی معادل تابع g(n) است.

**اگر دایره آبی رنگ را g(n) بدانیم خواهیم داشت.**

**روابط زیر همیشه برقرار است.**

$$f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n)) \qquad f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$$

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \text{ and } g(n) \in O(f(n))$$

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))$$

**نکته : کافیست بجای $\in O$ عبارت $\leq$ ، بجای $\in \Omega$ عبارت $\geq$ و بجای $\in \Theta$ عبارت $=$ را قرا دهیم و رابطه ها را چک کنیم.**

# ویژگی های نماد های مجانبی (سایر موارد نیز مانند Big O)

## پیچیدگی های متداول

$O(1) < O(\log n) < O(n) < O(n*\log n) < O(n^2) \; \dots < O\left(n^{\text{عدد}}\right) < O(2^n) < O(3^n) < \cdots < O\left(\text{عدد}^{n}\right) < O(n!) < O(n^n)$

ثابت لگاریتمی خطی تـــــوانـــــــی نـــــمـــــــایـــــــــی فاکتوریل

- ❑ پیچیدگی عدد ثابت می شود O(1)  $15000000 \in O(1)$

- ❑ ضریب ثابت عددی پشت تابع در پیچیدگی اثر ندارد و میتوان آن را حذف کرد.

$5n^3 \in O(n^3)$

بر اساس ویژگی فوق میتوان ثابت کرد پایه log اهمیتی ندارد و قابل تغییر به هر عددی می باشد.

$\log_a n = \log_a b * \log_b n$ , $\log_a b$ عدد ثابت $\Rightarrow \log_a n \in \log_b n$

- ❑ پیچیدگی حاصلجمع دو تابع میشود ماکزیمم پیچیدگی آنها

$7n^3 + \log n \in \max\left(O(n^3), O(\log n)\right) \in O(n^3)$

بر اساس ویژگی فوق میتوان گفت که پیچیدگی چند جمله ای می شود n به توان بزرگترین توان

$a_m n^m + \cdots a_2 n^2 + a_1 n + a_0 \in O(n^m)$

- ❑ پیچیدگی حاصلضرب دو تابع می شود ضرب پیچیدگی آنها

**در محاسبه پیچیدگی توابع سعی میکنیم به یکی از حالات متداول یا حاصلضرب آنها در یکدیگر برسیم**

# پیچیدگی دستورات در زبان های برنامه نویسی

**برای تشخیص عملکرد یک برنامه در گام اول دستورات اجرایی برنامه را می شماریم سپس پیچیدگی معادل آن را محاسبه میکنیم.**

- ❑ دستورات مقدار دهی و محاسبه، داری یک واحد دستور اجرایی می باشند.
- ❑ دستور حلقه For اگر n دور بچرخد خودش بابت دور اخر که وارد حلقه نمی شود n+1 دستور اجرایی محاسبه می شود.

(از آنجایی که در نهایت تعداد دستورات برنامه در قالب پیچیدگی نمایش داده می شوند در بسیاری از موارد تقریبی عمل میکنیم مثلا حلقه for)

| | | |
|---|---|---|
| برای حلقه for(i=a;i<=b;i++) | واحد اجرایی خود حلقه | (b-a+1)+1 |
| | تکرار دستورات بدنه حلقه | (b-a+1) |

| | واحد اجرایی | تکرار | کل |
|---|---|---|---|
| x=5; | 1 | 1 | 1 |
| for (i=1;i<=n;i++) | n+1 | 1 | n+1 |
| x++; | 1 | n | n |
| | T(n)=2n+2 | | |

$$T(n)=2n+2 \in O(n)$$

# پیچیدگی دستورات مثال

| | واحد اجرایی | تکرار | کل |
|---|---|---|---|
| x=5; | 1 | 1 | 1 |
| for (i=0;i<=n;i++) | n+2 | 1 | n+2 |
| x++; | 1 | n+1 | n+1 |
| | T(n)=2n+4 | | |

$$T(n)=2n+4 \in O(n)$$

| | واحد اجرایی | تکرار | کل |
|---|---|---|---|
| x=0; | 1 | 1 | 1 |
| for (i=1;i<=n;i++) | n+1 | 1 | n+1 |
| for (j=1;j<=m;j++) | m+1 | n | mn+n |
| x++; | 1 | mn | mn |
| | T(n)=2mn+2n+2 | | |

T(n)=2mn+2n+2 if m=n $\Rightarrow T(n) = 2n^2 + 2n + 2 \in O(n^2)$

# پیچیدگی دستورات مثال–۲

واحد اجرایی خط تعریف متغیر(بدون مقدار دهی)، } و { مقدار 0 می باشد.

| | واحد اجرایی | تکرار | کل |
|---|---|---|---|
| int fact (int n) | 1 | 1 | 1 |
| { | 0 | 1 | 0 |
| int i; | 0 | 1 | 0 |
| long f=1; | 1 | 1 | 1 |
| for (i=2;i<=n;i++) | n | 1 | n |
| f*=I; | 1 | n-1 | n-1 |
| return(f) | 1 | 1 | 1 |
| } | 0 | 1 | 0 |
| | | | |
| | T(n)=2n+2 | | |

$$T(n)=2n+2 \in O(n)$$

# پیچیدگی دستورات با سیگما

برای محاسبه تعداد دستورات حلقه هایی که شمارنده آنها به صورت ++ یا -- تغییر میکنند، میتوان از $\sum$ استفاده کرد.

هر حلقه معادل یک $\sum$ می باشد.

حلقه های تو در تو $\sum$ های تو در تو می شوند.

بدنه ساده حلقه فارغ از اینکه چه عملیاتی دارد معادل 1 می باشد.

| | |
|---|---|
| for (i=1;i<=n;i++) | $\sum_{i=1}^{n}$ |
| for (j=1;j<=n;j++) | $\sum_{j=1}^{n}$ |
| x=x*35; | 1 |

| | |
|---|---|
| for (i=1;i<=n;i++) | $\sum_{i=1}^{n}$ |
| for (j=1;j<=i;j++) | $\sum_{j=1}^{i}$ |
| x=x*35; | 1 |

| | |
|---|---|
| for (i=1;i<=n;i++) | $\sum_{i=1}^{n}$ |
| for (j=i;j<=n;j++) | $\sum_{j=i}^{n}$ |
| x=x*35; | 1 |

$$\sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2 \in O(n^2)$$

$$\sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \in O(n^2)$$

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = \sum_{i=1}^{n}(n-i+1) = n^2 - \frac{n(n+1)}{2} + n = \frac{n^2}{2} + \frac{n}{2} \in O(n^2)$$

# محاسبه پیچیدگی بدون محاسبه تعداد دستورات

۱– برای محاسبه پیچیدگی هر حلقه باید نکات زیر در نظر گرفت

الف) آیا شمارنده وابسته به n است؟ شمارنده ای وابسته به n است که مقدار ابتدا یا انتهای آن ضرایب متفاوت از n باشد مثال:

ابتدا i=5 انتها i<n+30 وابسته است

ابتدا i=n انتها i<n+8000 وابسته نیست

ابتدا i=3 انتها i<150000000 وابسته نیست

ب) گام حرکتی شمارنده جمع و تفریق است یا ضرب و تقسیم

- اگر حلقه وابسته به n نبود پیچیدگی قطعا O(1) است
- حلقه وابسته به n بود اگر شمارنده با جمع و تفریق تغییر می کرد O(n)
- حلقه وابسته به n بود اگر شمارنده با ضرب و تقسیم تغییر می کرد O(logn)

- پیچیدگی حلقه های تو در تو میشود ضرب پیچیدگی آنها
- پیچیدگی دستورات ساده O(1)
- پیچیدگی دستور if میشود ماکزیمم پیچیدگی قسمت if و قسمت else
- پیچیدگی حلقه های جدا از هم می شود ماکزییم آنها
- اگر حلقه داخلی به شمارنده حلقه بیرونی وابسته بود، کافیست یکبار جای شمارنده خارجی مقدار ابتدایی و بار دیگر مقدار انتهایی را قرار دهیم، اگر در یک از این دو حالات وابسته به n شد حلقه وابسته است وگرنه نیست (در ۹۹ درصد حالات وابسته است)

# مثال

## پیچیدگی حلقه های زیر چیست

```
x=5;
i=1;
While( i<=n)
 {
   x=x+2;
   i=i*2;
 }
```

O(logn)

```
x=5;
i=n;
While( i>1)
 {
   x=x+2;
   i=i/2;
 }
```

O(logn)

```
x=5;
i=1;
While( i<=n)
 {
   x=x+2;
   i++;
 }
```

O(n)

```
x=5;
i=n;
While( i>1)
 {
   x=x+2;
   i--;
 }
```

O(n)

# مثال –۲

## پیچیدگی دستورات زیر چیست

```
For (p=1;p<=2500;p++)          O(1)
  For (i=1 ; i<=n ;i++)        O(n)
    {
      for(j=1;j<=i;j++)        O(n)
         x=x*3;                O(1)
      for(k=n;k>=1;k/=2)       O(logn)
         x=x*10;               O(1)
    }
```

O(1*n)=O(n)

O(1*logn)=O(logn)

max

O(n)

$O(1*n*n)=O(n^2)$

# محاسبه پیچیدگی توابع بازگشتی

در این حالت لازم است رابطه بازگشتی تعداد فراخوان را نوشته و تعداد دستورات را محاسبه کنیم.
اگر تابع حلقه نداشت کل دستورات تابع را 1 در نظر میگیریم و رابطه بازگشتی میشود 1 بعلاوه تعداد فراخوان مراحل قبل.(فقط تعداد فراخوان مهم است و فرمولی که تابع فراخوان دارد اهمیتی ندارد.

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

**رابطه بازگشتی تابع فاکتوریل**

$$fact(n) = \begin{cases} n * fact(n-1) & n > 1 \\ 1 & n = 1 \end{cases}$$

**رابطه بازگشتی تعداد دستورات طبق فراخوان بازگشتی**

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

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 1 + 1 + T(n-2) \\ &= 1 + 1 + 1 + T(n-3) \\ &\ldots\ldots \\ &= k + T(n-k) \end{aligned}$$

T(1)ساده ترین حالت مسئله است پس
$$n - k = 1 \quad \Rightarrow \quad k = n - 1$$

$$= n - 1 + T(1) = n - 1 + 1 = n \in O(n)$$

# محاسبه پیچیدگی توابع بازگشتی – مثال ۱

```
int test (int n)
 {
     if(n==1)
         return (1500);
     return(2*test(n-1)+4*n+2);
  }
```

علی رغم تفاوت فرمول تابع رو برو اما از نظر تعداد فراخوان دقیقا مشابه مسئله قبل است لذا فرمول هم طبق صفحه قبل انجام می شود

**رابطه بازگشتی تعداد دستورات طبق فراخوان بازگشتی**

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

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 1 + 1 + T(n-2) \\ &= 1 + 1 + 1 + T(n-3) \\ &\ldots\ldots \\ &= k + T(n-k) \end{aligned}$$

T(1)ساده ترین حالت مسئله است پس

$$n - k = 1 \quad \Rightarrow \quad k = n - 1$$

$$= n - 1 + T(1) = n - 1 + 1 = n \quad \in O(n)$$

# محاسبه پیچیدگی توابع بازگشتی – مثال ۲

```
int test (int n)
 {
     if(n<=2)
         return (99);
     return(2*test(n-2));
 }
```

یک فراخوان اما به مرحله n-2

**رابطه بازگشتی تعداد دستورات طبق فراخوان بازگشتی**

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

$$\begin{aligned} T(n) &= 1 + T(n-2) \\ &= 1 + 1 + T(n-4) \\ &= 1 + 1 + 1 + T(n-6) \\ &= 1 + 1 + 1 + 1 + T(n-8) \\ &\ldots\ldots \\ &= k + T(n-2k) \end{aligned}$$

T(2)ساده ترین حالت مسئله است پس

$$n - 2k = 2 \ \Rightarrow \ k = \frac{n-2}{2}$$

$$= \frac{n-2}{2} + T(2) = \frac{n-2}{2} + 1 = \frac{n}{2} \in O(n)$$

# محاسبه پیچیدگی توابع بازگشتی– مثال ۳

```
int test (int n)
 {
     if(n<=1)
         return (4);
     return(2*test(n/2)+n);
 }
```

در این مثال نیز مانند مثال قبل یک فراخوان داریم اما بجای مرحله قبل سراغ مرحله n/2 می رویم لذا ساختار رابطه بازگشتی تعداد دستورات به فرم زیر است

**رابطه بازگشتی تعداد دستورات طبق فراخوان بازگشتی**

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

$$\begin{aligned} T(n) &= 1 + T(n/2) \\ &= 1 + 1 + T(n/4) \\ &= 1 + 1 + 1 + T(n/8) \\ &\ldots\ldots \\ &= k + T\left(\frac{n}{2^k}\right) \end{aligned}$$

T(1)ساده ترین حالت مسئله است پس

$$2^k = n \Rightarrow \quad k = \log_2 n$$

$$= \log_2 n + T(1) = \log_2 n + 1 \in O(\log_2 n)$$

# محاسبه پیچیدگی توابع بازگشتی – مثال ۴

```
int test (int n)
 {
     if(n==1)
         return (4);
     return(test(n-1)+test(n-1));
 }
```

در این مثال دو بار فراخوان بازگشتی رخ داده است لذا

رابطه بازگشتی تعداد دستورات طبق فراخوان بازگشتی

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

$$\begin{aligned} T(n) &= 1 + 2T(n-1) = 1+2[1+2T(n-2)] \\ &= 1 + 2 + 4T(n-2) \\ &= 1 + 2 + 4 + 8T(n-3) \\ &\ldots\ldots \\ &= 1 + 2 + 4 + \cdots + 2^{k-1} + 2^k T(n-k) \end{aligned}$$

T(1)ساده ترین حالت مسئله است پس

$$n - k = 1 \Rightarrow k = n - 1$$

$$= 1 + 2 + 4 + \cdots + 2^{n-2} + 2^{n-1} = 2^n - 1 \in O(2^n)$$

# محاسبه پیچیدگی توابع بازگشتی بدون محاسبه تعداد دستورات

**۱– در حالتی که در قسمت بازگشتی فقط یک فراخوان بازگشتی داشته باشیم:**

**الف) متغیر اگر با جمع و تفریق تغییر کرده باشد مثلا (n-a) تعداد دستورات متناسب با $\frac{n}{a}$ و پیچیدگی O(n) خواهد بود.**

**ب) متغیر اگر با ضرب و تقسیم تغییر کرده باشد مثلا (n/a) تعداد دستورات متناسب با $\log_a n$ و پیچیدگی $O(\log n)$ خواهد بود.(قبلا گفتیم پایه لگاریتم اهمیت ندارد اما در تست جواب صحیح تر با پایه a می باشد)**

**۲–اگر تعداد فراخوان بازگشتی b بار بود:**

**الف) متغیر اگر با جمع و تفریق تغییر کرده باشد مثلا (n-a) پیچیدگی $O(b^{\frac{n}{a}})$ خواهد بود.**

**ب) متغیر اگر با ضرب و تقسیم تغییر کرده باشد مثلا (n/a) پیچیدگی $O(b^{\log_a n})$ خواهد بود.(در این حالت پایه لگاریتم باید a باشد چون لگاریتم قسمتی از توان است و پیچیدگی لگاریتم ساده نیست)**