فصل اول

الگوریتم (مشخصات، تجزیه و تحلیل)

مقدمه

در این فصل به مطالعه الگوریتم‌ها می‌پردازیم و مطالبی را در مورد مشخصات یک الگوریتم و نحوهٔ تجزیه و تحلیل الگوریتم‌ها، بیان می‌کنیم. برای بسیاری از مسائل، الگوریتم‌هائی وجود دارد و طراحی یک الگوریتم کارآمد، برای توسعهٔ سیستمهای کامپیوتری، نقش مهمی را بازی می‌کند. به هر حال قبل از شروع، باید مفاهیمی را به طور کامل شرح دهیم.

❖ تعریف : **الگوریتم، مجموعهٔ محدودی از دستورالعملها است که اگر دنبال و اجرا شود، هدف خاصی را برآورده می‌کند. به علاوه موارد زیر در هر الگوریتم قابل بررسی است:**

۱ـ ورودی : یک الگوریتم می‌تواند هیچ یا چندین کمیت بعنوان ورودی داشته باشد ،که از محیط خارج تامین می‌شود.

۲ـ خروجی : الگوریتم بایستی حداقل یک کمیت به عنوان خروجی داشته باشد.

۳ـ قطعیت : هر دستورالعمل باید واضح و خالی از هر نوع ابهامی باشد.

۴ـ محدودیت : الگوریتم بایستی پس از طی مراحل محدودی خاتمه یابد.

۵ـ کارآیی : هر دستورالعمل باید به گونه‌ای باشد که شخصی با استفاده از قلم وکاغذ بتواند، آن را به طور دستی اجرا کند. در واقع فقط قطعیت کافی نیست، بلکه هر دستورالعمل باید انجام‌پذیر باشد.

در علم کامپیوتر بایستی بتوان بین یک الگوریتم و یک برنامه تفاوتهائی قائل شد، مثلاً یک برنامه نبایستی حتماً پایان‌پذیر باشد. بعنوان نمونه، سیستم‌عامل برنامه‌ای است که هیچگاه پایان نمی‌پذیرد و دائماً در یک سیکل انتظار است، تا برنامهٔ بعدی وارد شود.

تحلیل الگوریتم‌ها

تعیین این نکته که یک الگوریتم با چه میزان کارآیی مسئله را حل می‌کند، نیاز به تجزیه و تحلیل دارد، برای تحلیل یک الگوریتم روشهای مختلفی وجود دارد که در ذیل به بررسی برخی از مهمترین این روشها می‌پردازیم.

۱ـ تعیین گامهای یک برنامه

تعداد گامهای عبارت هر برنامه، به طبیعت آن عبارت بازمی‌گردد. در این بحث سعی بر آن داریم که برخی عبارت مهم که در هر برنامه می‌تواند وجود داشته باشد را بررسی کنیم.

۱ـ۱ـ عبارات توضیحی (Comments) : توضیحات، عبارات غیراجرائی هستند و تعداد گامهای آن صفر است.

۱ـ۲ـ عبارت تعیین نوع (Declarative statements) : این طبقه شامل عباراتی نظیر معرفی متغیرها و ثابتها و نوعهای جدید و... می‌باشد. تعداد گامهای این عبارات صفر است، چون اینها اجرائی نیستند.

۱ـ۳ـ عبارات و دستورات انتساب (Expressions and assignment statements) : تعداد تکرار بیشتر دستورات، یک است. بجز عباراتی که شامل فراخوانی توابع می‌باشد. در این حالت باید هزینهٔ لازم برای اجرای توابع را محاسبه کنیم.

۱ـ۴ـ عبارات تکرار (Iteration statements) : این دسته از عبارات، شامل عبارت for، while است. شمارش گام را فقط برای قسمت کنترل این عبارت در نظر می‌گیریم.

به طور کلی حلقه‌های تکرار به دو دسته تقسیم می‌شوند:

۱ـ حلقه‌های تکراری که ابتدا شرط را بررسی کرده و سپس دستورات را اجرا می‌کنند، در این مورد اگر تعداد تکرار دستورات k بار باشد، تعداد تکرار قسمت کنترل حلقه $k+1$ بار خواهد بود.

۲ـ حلقه‌های تکراری که ابتدا دستورات را اجرا کرده و در آخر شرط را بررسی می‌کنند، در این مورد اگر تعداد تکرار دستورات k بار باشد، تعداد تکرار قسمت کنترل حلقه نیز k بار خواهد بود.

۵ـ۱ـ **عبارات { و}** : تعداد گام هر عبارت {و} صفر است.

۶ـ۱ـ **عبارت نام تابع**: تعداد گام این عبارت صفر است.

۷ـ۱ـ **عبارت return** : تعداد گام این عبارت یک است.

با انتساب مقادیر فوق به شمارش گام عبارات، می‌توان تعداد گامهای یک برنامهٔ خاص را تعیین کرد، برای تعیین شمارش گام برنامه باید، کل تعداد گامهای هر عبارات را لیست کنیم. در این حالت، ابتداد تعداد گامها را در هر اجرای عبارت، و تعداد کل دفعات اجرای هر عبارت را تعیین می‌کنیم. با ترکیب این دو مقدار کل زمان اجرای هر عبارت به دست می‌آید. با جمع کردن این مقادیر برای همهٔ عبارات، شمارش گام برنامه به دست خواهد آمد.

مثال۱ : تعداد گام برنامه زیر کدام است؟

شمارهٔ خطی

```
1    float sum (int  n)
2    {
3    float    s= o ;
4    int    i;
5    for (i = 1  ; i < =n  ; ++ i )
6    S + = i  ;
7    Return   S ;
8    }
```

۱) $2n+1$

۲) $2n+2$

۳) $2n+3$

۴) $2n$

☑ **پاسخ:** گزینه «۳»

در جدول ۱ـ۱، تعداد گامهای هر اجرا (s/e) و تکرار هر دستور تابع sum آورده شده است، تعداد کل گامها $2n+3$ می‌باشد. نکته‌ای که باید بدان توجه کنید، آن است که، تکرار دستور خط 5، $n+1$ مرتبه است نه n مرتبه. زیرا در خاتمه حلقهٔ for، مقدار نهائی i ، $n+1$ است.
و گام دستور 3، یک است زیرا در این خط یک دستور انتساب وجود دارد.

« جدول ۱ـ۱ جدول گام‌ها برای مثال ۱»

خط	s/e	تکرار	کل مراحل
1	0	1	0
2	0	1	0
3	1	1	1
4	0	1	0
5	1	n+1	n+1
6	1	n	n
7	1	1	1
8	0	1	0
		مجموع همهٔ مراحل	$2n+3$

مثال ۲: تعداد گامهای برنامهٔ زیر کدام است؟

شماره خط

```
1     int   test(int   n)
2     {
3     int  i , j , a=5 , b=10 , c=a+ b ;
4     for (i= 0 ; i< =n ; ++i)
5     for(j=1,j<=n;++j)
6     {
7     a+=1;
8     b+=2 ;
9     c+=a+b;
10    }
11    Return    c ;
12    }
```

۱) $4n^2+2n+2$

۲) $4n^2+2n+5$

۳) $4n^2+n+5$

۴) $4n^2+n+2$

☑ **پاسخ:** گزینه «۲» در جدول زیر تعداد گامهای هر اجرا (s/e) و تکرار هر دستور تابع Test آورده شده است. تعداد کل گامها $4n^2+2n+5$ است.

خط	s/e	تکرار	کل مراحل
1	0	1	0
2	0	1	0
3	3	1	3
4	1	n+1	n +1
5	1	n(n+1)	n (n+1)
6	0	$n \times n$	0
7	1	$n \times n$	n^2
8	1	$n \times n$	n^2
9	1	$n \times n$	n^2
10	0	$n \times n$	0
11	1	1	1
12	0	1	0
		مجموع همهٔ مراحل	$4n^2+2n+5$

پیچیدگی زمانی (Time Complexity): به مقدار زمانی که صرف اجرای یک الگوریتم می‌شود پیچیدگی زمانی گفته می‌شود.

* اندازه مسأله : به تعداد ورودیها یا خروجیها یا ترکیب آنها که با تغییر آن دفعات انجام عمل مبنایی تغییر نماید اندازه مسأله گفته می‌شود.
* عمل مبنایی: قسمتی از الگوریتم که عمدهٔ زمان اجرای الگوریتم را به خود اختصاص می‌دهد عمل مبنایی گفته می‌شود.
* تابع پیچیدگی: برای یک الگوریتم، تابع پیچیدگی آن عبارت است از $T(n)$ که رابطهٔ بین اندازه مسأله و تعداد دفعات انجام عمل مبنایی را مشخص می‌کند (n اندازه مسأله است).
* حد بالای تابع پیچیدگی (بدترین حالت): اگر تابع پیچیدگی یک الگوریتم مانند $T(n)=f(n)$ باشد، گوییم $f(n)$ متعلق به $O(g(n))$ است $(f(n)\in O(g(n)))$ اگر: $\exists c, n_\circ > \circ \quad \forall n > n_\circ \quad f(n) \le cg(n) \Rightarrow f(n)\in O(g(n))$
* حد پایین تابع پیچیدگی (بهترین حالت): اگر تابع پیچیدگی یک الگوریتم مانند $T(n)=f(n)$ باشد، گوئیم $f(n)$ متعلق به $\Omega(g(n))$ است $(f(n)\in \Omega(g(n)))$ اگر: $\exists c, n_\circ > \circ \quad \forall n > n_\circ \quad f(n) \ge cg(n) \Rightarrow f(n)\in \Omega(g(n))$
* مقدار دقیق (رده) تابع پیچیدگی: اگر تابع پیچیدگی یک الگوریتم مانند $T(n)=f(n)$ باشد، گوییم $f(n)$ متعلق به $\theta(g(n))$ است $(f(n)\in \theta(g(n)))$ اگر: $\exists c_1, c_2, n_\circ > \circ \quad \forall n > n_\circ \quad c_1 g(n) \le f(n) \le c_2 g(n)$

نکته ۱ : توابع چند جمله‌ای که متعلق به θ یکدیگر هستند، بزرگترین توان آنها با هم برابر است.

مثال ۳: کدام گزینه نادرست است؟

۱) $3n^2+3n+1\in O(n^2)$ ۲) $5n+1\in \Omega(n)$ ۳) $3n^3+5n+1\in \Omega(n^2)$ ۴) $3n^2+5n\in O(n)$

☑ **پاسخ:** گزینه «۴» هیچگاه منحنی n از منحنی $3n^2+5n$ بالاتر قرار نمی‌گیرد و n حد بالائی تابع پیچدگی $3n^2+5n$ نمی‌باشد.

۲ـ تحلیل پیچیدگی زمانی

در این روش کارآئی الگوریتم را به عنوان تابعی از اندازه ورودی، با تعیین تعداد دفعات انجام برخی عملیات اصلی، تجزیه و تحلیل می‌کنیم. این روش یکی از تکنیکهای استاندارد تحلیل سیستم است.

به عنوان مثال الگوریتم روبرو را که به مرتب‌سازی حبابی (تبادلی) معروف است در نظر بگیرید

```
Void  bubblesort (int n, Keytype  s[])
{
  index  i, j;
  for (i = 1; i <= n − 1; i++)
     for (j = i + 1; j <= n; j++)
        if (s[j]<s[i])
           exchange  s[j] and s[i];
}
```

(الگوریتم ۱ـ۱)

در الگوریتم فوق می‌خواهیم آرایهٔ s را که دارای n عنصر است به صورت صعودی مرتب نمائیم. برای بسیاری از الگوریتم‌ها، یافتن یک واحد ورودی ـ موسوم به اندازهٔ ورودی ـ مشکل نیست. به عنوان مثال در الگوریتم ۱ـ۱ (مرتب‌سازی حبابی)، تعداد عناصر آرایه، یک مقدار ساده برای ورودی است که به همین دلیل به آن اندازهٔ ورودی می‌گوئیم. بعد از تعیین اندازهٔ ورودی بایستی دستور یا دستوراتی را انتخاب کنیم، که کل عملیات انجام شده توسط الگوریتم با تعداد دفعاتی که این دستور یا دستورات در الگوریتم اجرا می‌شوند، متناسب باشد.

این دستور یا دستورات در الگوریتم، عمل مبنایی نامیده می‌شوند. به عنوان مثال، در الگوریتم ۱ـ۱ دستور مقایسه می‌تواند انتخاب خوبی برای مشخص نمودن عمل مبنایی باشد.

با تعیین تعداد تکرارهای عمل مبنایی در الگوریتم با مقادیر مختلف n، کارآیی الگوریتم به روشنی مشخص می‌شود. در حالت کلی، تحلیل پیچیدگی زمانی یک الگوریتم، تعیین تعداد دفعاتی است که عمل مبنایی به ازای هر یک از مقادیر اندازهٔ ورودی انجام می‌شود.

گاهی اوقات ممکن است بخواهیم دو عمل مبنایی مختلف را در یک الگوریتم بررسی کنیم. به عنوان مثال، در الگوریتم ۱ـ۱ می‌خواهیم دستورالعمل مقایسه و دستورالعمل انتساب را بعنوان عمل مبنایی در نظر بگیریم. این بدین معنا نیست که دو دستورالعمل با هم عمل مبنایی را تشکیل می‌دهند، بلکه ما دو عمل مبنائی مجزا داریم یکی دستورالعمل مقایسه (قیاس) و دیگری دستورالعمل انتساب. ما به این دلیل این کار را انجام می‌دهیم که در یک الگوریتم مرتب‌سازی، تعداد مقایسه های انجام شده با تعداد اجرای دستورات یکسان نیست. لذا با انتخاب دو عمل مبنائی در یک الگوریتم و تعیین تعداد اجرای هر یک از آنها، می‌توانیم آگاهی بیشتری نسبت به کارآئی الگوریتم به دست آوریم.

گاهی برای تحلیل پیچیدگی زمانی، تعداد دفعات اجرای عمل مبنایی، نه تنها به اندازهٔ ورودی، بلکه به مقادیر ورودی نیز بستگی دارد.

بعنوان مثال الگوریتم زیر را که به جستجوی دودوئی موسوم است در نظر بگیرید.

```
Void  binary Search(int  n,
                    Const  Keytype s[],
                    KeyType   x,
                    index  & location)
{
index   Low, high, mid;
low = 1; high = n;
location = 0;
while(low <= high && locaton == 0){
  mid = ⌊(low + high) / 2⌋;
  if(x == s[mid])
    location = mid;
  else  if(x < s[mid])
    high = mid − 1;
  else
    low = mid + 1;}
}
```

(الگوریتم ۲ـ۱)

در الگوریتم فوق می‌خواهیم تعیین کنیم که آیا عدد x در آرایهٔ n کلیدی s وجود دارد یا خیر که البته آرایهٔ s به صورت صعودی مرتب شده است.

در الگوریتم ۲ـ۱ (جستجوی دودوئی) اگر x اولین عنصر آرایه باشد، عمل مبنایی تنها یک مرتبه انجام می‌شود، در حالیکه اگر x در آرایه وجود نداشته باشد، عمل مبنایی، $\left\lfloor \log_2^n \right\rfloor + 1$ مرتبه اجرا می‌گردد.

در حالت دیگر، نظیر الگوریتم ۱ـ۱ (مرتب‌سازی حبابی)، عمل مبنایی مقایسه برای هرنمونه از اندازهٔ ورودی n، به تعداد مشخص و یکسانی انجام می‌شود. در چنین حالتی، $T(n)$ مبین تعداد عمل مبنایی است که الگوریتم به ازای یک نمونه از اندازهٔ ورودی n انجام می‌دهد، $T(n)$ را پیچیدگی زمانی حالت معمول الگوریتم و تعیین $T(n)$ را تحلیل پیچیدگی زمانی حالت معمول الگوریتم می‌نامیم، مثالی از این تحلیل را در زیر آورده‌ایم:

تحلیل پیچیدگی زمانی حالت معمول الگوریتم ۱ـ۱ (مرتب‌سازی حبابی)

همانطور که قبلاً نیز اشاره شد، در حالتی که الگوریتم با مقایسه کلیدها عمل مرتب‌سازی را انجام می دهد، می‌توانیم دستورالعمل مقایسه یا دستورالعمل انتساب را به عنوان عمل مبنایی در نظر بگیریم. در اینجا تعداد مقایسات را مورد تجزیه و تحلیل قرار می‌دهیم :

عمل مبنایی : مقایسه S[j] با S[i].

اندازهٔ ورودی: n، تعداد عناصری که باید مرتب شوند.

حال بایستی تعداد گذرهای موجود در حلقه for j را تعیین کنیم. برای هر n معین، همواره تعداد n-1 گذر از حلقه for i وجود دارد. در اولین گذر از حلقهٔ for i ,n-1 گذر از حلقه for j، در دومین گذر از حلقه for i ، $n-2$ گذر از حلقه for j,... و در آخرین گذر، تنها یک گذر از حلقه for j وجود خواهد داشت. بنابراین، تعداد کل گذرهای انجام شده از حلقه for j برابر است با :

$$T(n)=(n-1)+(n-2)+(n-3)+...+1=\frac{n(n-1)}{2}$$

همانطوری که قبلاً اشاره کردیم، عمل مبنایی در الگوریتم ۲ـ۱ (جستجوی دودوئی) برای همه نمونه‌های اندازه ورودی n به تعداد یکسانی انجام نشده است. بنابراین، نمی‌توانیم آن را دارای یک پیچیدگی زمانی حالت معمول بدانیم. این مطلب برای بسیاری از الگوریتم‌ها نیز صادق است. البته منظور ما این نیست که چنین الگوریتم‌هایی نمی‌توانند تجزیه و تحلیل شوند، چون هنوز سه روش دیگر نیز برای تحلیل الگوریتم‌ها وجود دارد. در این روش $w(n)$ بعنوان حداکثر دفعات اجرای عمل مبنایی، و برای یک الگوریتم معین تعیین $w(n)$ بعنوان پیچیدگی زمانی بدترین حالت الگوریتم در نظر گرفته می‌شود. واضح است که اگر $T(n)$ وجود داشته باشد. $w(n)=T(n)$ خواهد بود. در زیر تحلیلی از $w(n)$، در حالتی که $T(n)$ وجود ندارد را آورده‌ایم:

الگوریتم جستجوی ترتیبی یا خطی را به صورت زیر در نظر بگیرید.

```
void linear search (int  n ,
                    const key type   s[] ,
                    key type   x ,
                    index  δ location )
{
  index   i ;
location = 0;
for(i=1 ; i<=n ; ++i)
if (x== S [i]){
                location = i ;
                break ;
                }
}
```

الگوریتم (۳ـ۱)

تحلیل پیچیدگی زمانی بدترین حالت الگوریتم جستجوی ترتیبی

عمل مبنایی : مقایسه یک عنصر آرایه با x

اندازه ورودی : n، تعداد عناصر آرایه

عمل مبنایی، حداکثر n مرتبه انجام شده است و این حالتی است که x در آرایه وجود ندارد و یا x آخرین عنصر آرایه است.

بنابراین: $w(n)=n$

اگر چه تحلیل بدترین حالت، ما را از حداکثر زمانی که صرف الگوریتم می‌شود، آگاه می‌سازد، اما در بعضی حالات، ممکن است به میانگین زمانی الگوریتم نیز علاقه‌مند باشیم. برای یک الگوریتم معین $A(n)$ بعنوان میانگین (مقدار مورد انتظار) تعداد دفعاتی که الگوریتم، عمل مبنایی را به ازاء هر اندازه ورودی n اجرا می‌کند، معرفی شده وتعیین آن را پیچیدگی زمانی حالت میانی الگوریتم می‌نامیم. همانند حالت ذکر شده برای $w(n)$، اگر $T(n)$ وجود داشته باشد، آنگاه $A(n)=T(n)$ خواهد بود. معمولاً تحلیل حالت میانی مشکل‌تر از تحلیل بدترین حالت است.

در زیر تحلیلی از $A(n)$، در حالتی که $T(n)$ وجود ندارد آورده‌ایم:

تحلیل پیچیدگی زمانی حالت میانی الگوریتم جستجوی ترتیبی

عمل مبنایی : مقایسه یک عنصر آرایه با x

اندازه ورودی : n، تعداد عناصر موجود در آرایه

مسئله را در حالتی تحلیل می‌کنیم که می‌دانیم عنصر x حتماً در آرایه وجود دارد. همه عناصر آرایهٔ S به صورت مجزا هستندو هیچ دلیلی وجود ندارد که احتمال x در یک اندیس آرایه را بیشتر از اندیس دیگر بدانیم. براساس این اطلاعات، برای $1<K<n$، احتمال اینکه x در اندیس Kام باشد برابر $\frac{1}{n}$ است. اگر x در ا ندیس Kام باشد، تعداد دفعاتی که عمل مبنایی باید اجر شود تا محل x در آرایه مشخص شود (و از حلقه خارج شود) برابر K است و این موضوع بدین معناست که پیچیدگی زمانی حالت میانی برابر است با :

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

یک مورد دیگر در تحلیل پیچیدگی زمانی، تعیین حداقل تعداد دفعاتی است که یک عمل مبنایی انجام می‌شود. در یک الگوریتم در حال بررسی، $B(n)$ نشان‌دهندهٔ حداقل تعداد دفعات اجرای عمل مبنایی به ازاء ورودی n است. به همین دلیل، تعیین $B(n)$، پیچیدگی زمانی بهترین حالت الگوریتم نامیده می‌شود. همانند حالات $w(n)$, $A(n)$، اگر $T(n)$ وجود داشته باشد، آنگاه $B(n) = T(n)$.

تحلیل پیچیدگی زمانی بهترین حالت الگوریتم جستجوی ترتیبی

عمل مبنایی : مقایسه یک عنصر آرایه با x

اندازه ورودی : n، تعداد عناصر آرایه

از آنجائیکه $n > 1$ است، لذا بایستی حداقل یک گذر از حلقه وجود داشته باشد و اگر $x = S[1]$ باشد بدون توجه به مقدار اندازه ورودی n، تنها یک گذر از حلقه وجود خواهد داشت.

بنابراین: $B(n) = 1$

بطور کلی، یک تابع پیچیدگی می‌تواند هر تابعی از اعداد صحیح غیرمنفی به اعداد حقیقی غیرمنفی باشد. هرگاه در تحلیل برخی از الگوریتمهای خاص به پیچیدگی زمانی یا پیچیدگی حافظه اشاره نشود، معمولاً از توابع استانداردی نظیر $f(n)$ و $g(n)$ به عنوان توابع پیچیدگی استفاده می‌شود.

بعنوان مثال توابع:

$f(n) = n$

$f(n) = n^2$

$f(n) = \log n$

$f(n) = 3n^2 + 4n$

را در نظر بگیرید، همگی مثالهایی از توابع پیچیدگی هستند زیرا تمامی آنها تابعی از اعداد صحیح غیرمنفی به اعداد حقیقی غیرمنفی می‌باشند.

مجموعهٔ تمامی توابع پیچیدگی که می توانند با توابع مربعی محض طبقه‌بندی شوند ، $\theta(n^2)$ نامیده می‌شوند [علامت θ (تتا) یک حرف بزرگ یونانی است]. اگر یک تابع ، عنصری از مجموعهٔ $\theta(n^2)$ باشد ، می‌گوئیم که آن تابع یک ترتیب از n^2 است. برای مثال ، چون می‌توانیم از عناصر با ترتیب پایین صرف نظر کنیم ، لذا $g(n) = 5n^2 + 100n + 20 \in \theta(n^2)$

به این معنی که$g(n)$ ترتیبی از n^2 است . برای ارائه یک مثال واقعی‌تر ، الگوریتم ۲ـ۱ (مرتب‌سازی حبابی) را یادآور می‌شویم که پیچیدگی زمانی آن را به صورت $T(n) = n(n-1)/2$ بیان کرده‌ایم . از آنجائیکه $n(n-1)/2 = \frac{n^2}{2} - \frac{n}{2}$ است ، لذا با کنار گذاشتن عنصر کم ترتیب $\frac{n}{2}$ می‌توان گفت که: $T(n) \in \theta(n^2)$

هنگامی که پیچیدگی زمانی یک الگوریتم در $\theta(n^2)$ است ، الگوریتم را الگوریتم زمان ـ مربعی یا الگوریتم $\theta(n^2)$ می‌نامیم مرتب‌سازی حبابی ،یک الگوریتم زمان ـ مربعی است . به طور مشابه ، سری توابع پیچیدگی که می‌توانند به همراه توابع مکعبی کامل دسته‌بندی شوند، $\theta(n^3)$ و یا ترتیب n^3 نامیده می‌شوند والی آخر . ما به این سری‌ها ، رده‌های پیچیدگی می‌گوئیم.

رده‌های زیر برخی از رایج‌ترین رده‌های پیچیدگی هستند: $\theta(1)\quad \theta(\log n)\quad \theta(n)\quad \theta(n\log n)\quad \theta(n^2)\quad \theta(n^3)\quad \theta(2^n)$

در ادامه چند مثال در مورد، پیچیدگی زمانی ارائه می‌گردد.

مثال۴: تکه برنامهٔ زیر را در نظر بگیرید:

```
int  S = O , i =1 ;
while (i<=n)
{
S+=i ;
i += 2;
}
```

پیچیدگی زمانی این برنامه در حالت معمول به صورت روبرو است: $T(n) = [\log_2^n] = +1 \in \theta(\log n)$

مثال۵: تکه برنامهٔ زیر را در نظر بگیرید:

```
int S =O , i =n
while (i >1)
{
S + = i ;
i /= 2 ;
}
```

پیچیدگی زمانی این برنامه در حالت معمول به صورت روبرو است: $T(n) = [\log_2^n] \in \theta(\log n)$

نکته۲ : در مورد مثالهای فوق با کمی دقت می‌توان گفت، اگر شمارندهٔ یک حلقه تقسیمات متوالی بر K و یا ضرب متوالی در K باشد پیچیدگی زمانی حلقه، تابعی از Log_k خواهد بود.

مثال۶ : تکه برنامهٔ زیر را در نظر بگیرید:

```
S= O ;
 for(x=1;x<=n,++x)
   for(y=1;y< = x ; ++y)
     for(z=1;z<=y;++z)
       S +=1 ;
```

در این برنامه سه حلقهٔ for متداخل داریم که هر سه حلقه به هم وابسته می‌باشند، یعنی به طور مثال اگر $x = K$ شود، حلقهٔ for y، از 1 تا K یعنی K بار و به ازای y از 1 تا K حلقهٔ for z، یک بار، دو بار و ... k بار اجرا می‌شود، بنابراین پیچیدگی زمانی این برنامه در حالت معمول به صورت زیر است:

$$T(n) = 1 + (1+2) + (1+2+3) + \cdots + (1+2+\cdots+k) + \cdots + (1+2+\cdots+n)$$

$$= \sum_{K=1}^{n} \frac{K(K+1)}{2} = \frac{1}{2}\sum_{K=1}^{n}(K^2+K) = \frac{1}{2}(\sum_{K=1}^{n} K^2 + \sum_{K=1}^{n} K)$$

می‌توان با کمک استقراء اثبات کرد که :

$$\sum_{K=1}^{n} K = \frac{n(n+1)}{2} \qquad (۱ - ۵)$$

$$\sum_{K=1}^{n} K^2 = \frac{n(n+1)(2n+1)}{6} \qquad (۱ - ۶)$$

و با استفاده (۵ ـ ۱) و (۶ ـ ۱) داریم:

$$\frac{1}{2}(\sum_{K=1}^{n} K^2 + \sum_{K=1}^{n} K) = \frac{1}{2}(\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}) = \frac{(n^2+n)(2n+1)+3n^2+3}{12} \in \theta(n^3)$$

نکته ۳: مفهوم واقعی ترتیب، θ نامیده می‌شود، البته در سئوالات کنکور و بعضی از کتابها، بجای θ از O(big O) استفاده شده است.

مثال ۷ : پیچیدگی زمانی تکه برنامه زیر چیست؟

```
int  i, j, x
for(i = o; i <= n – 1; + + i)
for(j = o; j <= i; + + j)
x+ = 1;
}
```

۱) $O(n^3)$
۲) $O(n)$
۳) $O(n^2)$
۴) $O(n\text{Log}n)$

☑ **پاسخ :** گزینه «۳» برای پیدا کردن مرتبه زمانی کافیست تعداد دفعاتی که دستور درون حلقه for j اجرا می‌شود را به دست آوریم. بنابراین بدین ترتیب عمل می‌کنیم:

1) I = 0 1 2 3 4 5 6 ⋯ n − 1
2) forj = 1 2 3 4 5 6 7 ⋯ n

شماره (۱) مقادیری که I به خود می‌گیرد را مشخص می‌کند و ردیف شماره (۲) دفعاتی که حلقه for(j) به ازای هر مقدار I اجرا می‌شود را مشخص می‌کند. بنابراین مرتبه اجرایی الگوریتم برابر است با مجموع دفعاتی که حلقه for j اجرا می‌شود که برابر است با:

$$1+2+3+\ldots+n = n(n+1)/2 \rightarrow T(n) = f(n) = n(n+1)/2$$
$$T(n) \in O(n \wedge 2)$$

مثال ۸ : پیچیدگی زمانی الگوریتم زیر چیست.

```
int k,i,j,x;
for (i=o; i<=n; ++i)
{
 k=i/2;
   for(j=o;j<=k;++j)
     x+=1;
   }
```

۱) $O(n^3)$
۲) $O(n^2)$
۳) $O(n)$
۴) $O(n\,\text{Log}\,n)$

☑ **پاسخ:** گزینه «۲» تکنیک دقیقاً مانند سؤال قبل است. از جمع تعداد دفعات اجرای دستور درون حلقهٔ for j مرتبه زمانی الگوریتم به دست می‌آید:

1) $I=0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad \ldots \quad n-1 \qquad n$

2) for $j=1 \quad 1 \quad 2 \quad 2 \quad 3 \quad 3 \quad \ldots \quad (n-1)/2 \quad (n-1)/2$

بنابراین داریم: $T(n)=(n-1)(n+1)/4 \;,\; T(n)\in O(n\wedge 2)$

مثال ۹: پیچیدگی زمانی روال مقابل چیست؟

```
int i,j,x;
for(i=0,i<=n-1;++i)
 for(j=n;j>=i+1;--j)
  x+=1;
```

۱) $o(n^2)$ ۲) $o(n)$

۳) $o(n^3)$ ۴) $o(n\log n)$

☑ **پاسخ:** گزینه «۱» برای پیدا کردن مرتبه زمانی کافیست تعداد دفعاتی که دستور درون حلقه for $j=\ldots$ اجرا می‌شود را به دست آوریم. بنابراین بدین ترتیب عمل می‌کنیم:

1) $I=0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \ldots n-1$

2) for $j=n \quad n-1 \quad n-2 \quad n-3 \quad \ldots \quad 1$

شماره (۱) مقادیری که I به خود می‌گیرد را مشخص می‌کند و ردیف شماره (۲) دفعاتی که حلقه for(j) به ازای هر مقدار I اجرا می‌شود را مشخص می‌کند. بنابراین مرتبه اجرایی الگوریتم برابر است با مجموع دفعاتی که حلقه for (j) اجرا می‌شود که برابر است با:

$$1+2+3+\ldots+n=n(n+1)/2 \;\rightarrow T(n)=f(n)=n(n+1)/2$$
$$T(n)\in O(n\wedge 2)$$

الگوریتم‌های بازگشتی

بیش از هر چیز دیگر ، تأکید بر نوشتن برنامه‌هایی است که تا حد ممکن آسان بیان شده باشند و به طور خوانا و صحیح هدف مورد نظر را بیان کنند. در واقع این موارد از خصوصیات ضروری روالها محسوب می‌شود . مجموعه‌ای از دستورالعمل‌ها که عمل منطقی را انجام می‌دهند ، می‌توانند در کنار هم قرارگرفته و به عنوان یک روال خوانده شوند . نام روال و پارامترهایش به عنوان دستورالعمل جدید در دیگر روالها آورده می‌شود . با تعیین ورودی ـ خروجی‌های یک روال به آن دسترسی داریم و حتی‌لازم نیست که، نحوه عملکردش را بدانیم. با چنین دیدی به یک روال ، ابتدا فراخوانی شده ، اجرا می‌شود و سپس کنترل به نقطهٔ فراخوانی شده، برمی‌گردد. البته بایستی بگوییم که روالها می‌توانند، خودشان را قبل از به پایان رسیدنشان فراخوانی کنند (بازگشتی مستقیم) یا روالهای دیگر را صدا زده و به روال صدا زننده برگردند (بازگشتی غیرمستقیم). این روشهای بازگشتی، کاربردهای زیادی داشته و حتی می‌توان گفت ، خیلی مهم هستند و خیلی وقتها می‌توان اعمال پیچیده را با بازگشتی به آسانی حل کرد. بعنوان مثال به الگوریتم محاسبهٔ عنصر n ام فیبوناچی که به صورت بازگشتی زیر تعریف می‌شود ،اشاره می‌کنیم :

$$f_{\circ}=\circ$$
$$f_1=1$$
$$f_n=f_{n-1}+f_{n-2} \qquad n\geq \text{برای } ۲ \quad n\in z^+$$

با محاسبهٔ چند عنصر اولیهٔ این دنباله داریم:

$$f_2=f_1+f_0=1+0=1$$
$$f_3=f_2+f_1=1+1=2$$
$$f_4=f_3+f_2=2+1=3$$
$$f_5=f_4+f_3=3+2=5,\cdots$$

دنبالهٔ فیبوناچی دارای کاربردهای مختلفی در علوم ریاضیات و کامپیوتر می‌باشد و بدلیل اینکه این دنباله بصورت بازگشتی تعریف شده است، الگوریتم بازگشتی زیر نیز بر همین اساس تعریف می‌شود.

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

مثال ۱۰: تابع زیر روی عدد طبیعی X چه عملی انجام می‌دهد؟

```
function   g(x)
IF X > 1 Then
return    X * g(X − 1)
else
return   1
end g
```

۱) 1

۲) X^X

۳) $X!$

۴) $\sum_{i=1}^{X} i$

☑ **پاسخ:** گزینه «۳» برای عدد مفروض X با توجه به تعریف تابع $g(x)$ داریم:

$$g(X) = X * g(X-1)$$
$$g(X-1) = (X-1) * g(X-2)$$
$$\vdots$$
$$g(1) = 1$$

و لذا:

$$g(X) = X * (X-1) * (X-2) * \cdots * 1 = X!$$

مثال ۱۱: تابع زیر چه فرمولی را محاسبه می‌کند؟

```
function   f(n : integer; x : real) : real;
begin
     if  n <= 1  then
                f := x
        else
                f := x * f(n − 1, x)
end
```

۱) x^n

۲) x^{n-1}

۳) $x!$

۴) $(x-1)!$

☑ **پاسخ:** گزینه «۱» باتوجه به تعریف تابع f، اگر $n > 1$ باشد، داریم:

$$f(n,x) = x * f(n-1,x)$$
$$f(n-1,x) = x * f(n-2,x)$$
$$\vdots$$
$$f(1,x) = x$$

و لذا:

$$f(n,x) = \underbrace{x * x * x * x}_{n} = x^n$$

مثال ۱۲ : با توجه به تابع روبرو مقدار func(۱۰۰) چه خواهد بود؟

```
int   func(int  n)
  {
     if (n = = 0)return(0);
    return(n + func(n − 1));
   }
```

۱) 199

۲) 200

۳) 5050

۴) 10000

☑ **پاسخ:** گزینه «۳» با توجه به تعریف تابع، مقدار $func(100) = 100 + func(99)$ است و $func(99) = 99 + func(98)$ و با ادامهٔ این روند تا صفر داریم:

$$func(100) = 100 + 99 + 98 + \cdots + 1 = \frac{100(100+1)}{2} = 5050$$

مثال ۱۳ : پیچیدگی زمانی تابع مقابل چیست؟

```
Function Test (n:integer) : integer;
Begin
      If n< = 1 then
          Test := 0
      Else
          Test := 1+Test (n div 2);
End;
```

☑ **پاسخ :** همانطور که پیداست در این تابع بازگشتی هر بار اندازه مسأله نصف می‌شود. یعنی تابع خودش را مجدداً فراخوانی می‌کند اما با مقدار نصف n بنابراین داریم: $T(n) = T(n/2) + a$

a مقدار ثابتی مستقل از اندازه مسأله است که در هر فراخوانی صرف اعمالی چون مقایسه مقدار n یا عمل جمع می‌شود. در نتیجه داریم:
$T(n) \in O(\log n)$, $T(n) = f(n) = \log n$

مثال ۱۴ : پیچیدگی زمانی تابع مقابل چیست؟

```
Function Test (n:integer) : integer;
Begin
    If n< = 1 then
        Test := 1
    Else
        If n div 2 = 0 then
            Test :=Test (n-1)
        Else
            Test : = Test (n-1);
End;
```

☑ **پاسخ :** همانطور که پیداست در این تابع بازگشتی هر بار اندازه مسأله یک واحد کم می‌شود. یعنی تابع خودش را مجدداً فراخوانی می‌کند اما با مقدار $n-1$. دقت کنید که اگر چه در تابع دو بار فراخوانی مجدد مشاهده می‌شود اما چون n یا فرد است یا زوج بنابراین هر بار فقط یک فراخوانی انجام می‌گیرد و داریم: $T(n) = T(n-1) + a$

a مقدار ثابتی مستقل از اندازه مسأله است که در هر فراخوانی صرف اعمالی چون مقایسه مقدار n می‌شود در نتیجه داریم: $T(n) \in O(n)$

مثال۱۵ : پیچیدگی زمانی تابع مقابل چیست؟

```
Function Test (n:integer) : integer;
Begin
    If n< = 1 then
        Test := 1
    Else
        Test :=Test (n-1) + Test (n-2);
End;
```

☑ **پاسخ :** در این تابع هر بار دو فراخوانی انجام می‌گیرد. یکی با اندازه $n-1$ و دیگری با اندازه $n-2$. بنابراین $T(n)$ به صورت روبرو به دست می‌آید: $T(n) = T(n-1) + T(n-2) + a$.

a مقدار ثابتی مستقل از اندازه مسأله است که در هر فراخوانی صرف اعمالی چون مقایسه مقدار n یا عمل جمع می‌شود. برای به دست آوردن $T(n)$ به صورت اصولی بایستی توسط روشهای حل معادلات بازگشتی عمل نمود که این روشها برای حل تست مناسب نمی‌باشد. بنابراین به گونه دیگر عمل می‌نماییم. چون همیشه مرتبه زمانی برای n های بزرگ در نظر گرفته می‌شود بنابراین مقدار $n-1$ و $n-2$ بسیار به هم نزدیک است. لذا چون ما به دنبال بدترین حالت هستیم هر دو مقدرا را همان $n-1$ در نظر می‌گیریم پس داریم: $T(n) = 2 * T(n-1)$
بنابراین خواهیم داشت: $T(n) \in O(2 \wedge n)$

دقت کنید که اگر هر دو مقدار را $n-2$ در نظر بگیریم داریم: $T(n) = 2 * T(n-2)$
که در این صورت $T(n) \in O(2 \wedge n/2)$.

مثال ۱۶ : پیچیدگی زمانی تابع test چیست؟

```
Function Test (n:integer) : integer;
Begin
    Test:= F1 (n);
End;
Function F1 (n:integer): integer;
Begin
     if n < = 1 then
        F1 := 1
     Else
        F1 := F2 (n-1);
End;
Function F2 (n: integer) : integer;
Begin
        if n < = 1 then
        F2 := 1
     Else
        F2 := F1(n-1);
End;
```

☑ **پاسخ :** $T(n) \in O(n)$. در مورد چگونگی به دست آوردن $T(n)$ توضیح دهید.

چگونگی به دست آوردن مرتبه زمانی توابع بازگشتی به صورت اصولی

هر تابع بازگشتی به طور مشخص دارای سه ویژگی می‌باشد:

۱) شرایط آغاز ۲) شرط توقف ۳) الگوریتم کاهش اندازه مسأله

شرط آغاز عبارت است از مقداری از اندازه مسأله که تابع برای اولین بار با آن فرا خوانده می‌شود. شرط توقف عبارت است از مقداری از اندازه مسأله مانند x که وقتی اندازه مسأله به آن مقدار برسد فراخوانی مجدد تابع را متوقف خواهیم ساخت. الگوریتم کاهش اندازه مسأله عبارت است از فراخوانی مجدد تابع به نحوی که ما را از شرط آغاز به شرایط توقف سوق دهد.

برای به دست آوردن پیچیدگی زمانی با توجه به نکات زیر برای $T(n)$ یک رابطه بازگشتی به دست می‌آوریم:

۱) همیشه زمانیکه اندازه مسأله به شرط توقف برسد الگوریتم زمان ثابت (مستقل از اندازه مسأله) a را صرف می‌نماید چون دیگر فراخوانی مجدد قطع می‌شود. این زمان ثابت جهت مقایسه اندازه مسأله با شرط توقف و یا عمل return می‌شود.

۲) زمانیکه هنوز اندازه مسأله به شرط توقف نرسیده باشد فراخوانی مجدد صورت می‌گیرد که به این معنی است که زمان $T(n)$ بر اساس زمان فراخوانی مجدد اما با اندازه مسأله کاهش یافته به صورت یک رابطه بازگشتی به دست می‌آید. همچنین باز هم زمان ثابت a به این مقدار اضافه می‌شود.

مثال ۱۷ :

```
Function Test (n: integer): integer;
Begin
      If n<=c then
         Test :=1
      Else
         Test := Test (n-1) + Test (n-1);
End;
```

T(n) = a n <= c (شرط توقف c است)

**** T(n) = 2*T(n-1)+a n>c**

مثال۱۸ :

```
Function Test (n: integer): integer;
Begin
      If n<=c then
         Test :=0
      Else
         Test := 1+Test (n div 2);
End;
```

T(n) = a n <= c (شرط توقف c است)

**** T(n) = T(n/2)+a n>c**

مثال ۱۹ :

```
Function Test (n: integer): integer;
Begin
      If n<=c then
         Test :=2
      Else
         Test := Test (n div 2) * Test (n div 2);
End;
```

T(n) = a n <= c (شرط توقف c است)

**** T(n) = 2*T(n/2)+a n>c**

مثال ۲۰ :

```
Function Test (n: integer): integer;
Begin
      If n<=c then
         Test :=10
      Else
         Test := 1+Test (n-1);
End;
```

T(n) = a n <= c (شرط توقف c است)

**** T(n) = T(n-1)+a n>c**

۳) در این مرحله باید روابط بازگشتی که برای $T(n)$ به دست آمده را حل نموده و $T(n)$ را به صورت تابعی از n مانند $f(n)$ بـه دسـت آوریـم. برای حل معادلات بازگشتی دو راه موجود است. یکی حل به روش تبدیل به معادلات دیفرانسیل و حل معادله که ایـن روش مـوردنظر مـا نیـست. روش دیگر جایگزاری‌های متوالی $T(n)$ بر اساس فرمول بازگشتی می‌باشد. برای حل به این روش از رابطه بازگشتی بـه دسـت آمـده بـرای $T(n)$ استفاده می‌نماییم و آنقدر مقادیر $T(k)$ را براساس رابطه بازگشتی جایگزین می‌نمـاییم تـا بـه $T(c)$ برسـیم. (c شـرط توقـف اسـت). حـال بـه جای $T(c)$ مقدار ثابت a را می‌گزاریم. با پیدا کردن حاصل‌جمع به دست آمده $T(n)$ محاسبه خواهد شد. در ادامه مثال‌های بالا را به ایـن روش حل می‌نماییم:

☑ **پاسخ ۱۷:**

$$T(n)=2*T(n-1)+a=2*[2*T(n-2)+a]+a=4T(n-2)+2a+a=$$
$$4*[2*T(n-3)+a]+2a+a=8*T(n-3)+4a+2a+a=.....=$$
$$(2\wedge k)*T(n-k)+(2\wedge(k-1))a+(2\wedge(k-2))a+...+4a+2a+a$$

چون باید به شرط توقف رسیده باشیم پس باید $n-k=c$ شده باشد. حال به جای $T(n-k)$ مقدار ثابت a را می‌گذاریم. پـس از فـاکتورگیری از a داریم:

$$T(n)=[(2\wedge k)+(2\wedge(k-1))+(2\wedge(k-2))+...+4+2+1]a$$

همانطور که می‌دانیم حاصل مقدار درون براکت طبق رابطه $\sum_{i=0}^{k}2^i=2^{(k+1)}-1$ برابر $(2\wedge(k+1))-1$ خواهد شد. اگر از مقدار ۱ صـرف‌نظـر کنیم خواهیم داشت: $T(n)=(2\wedge(k+1))*a$ حال کافیست که مقدار k را بر اساس n بنویسیم. داشتیم $n-k=c$ بنـابراین $k=n-c$.
پس داریم: $T(n)=(2\wedge(n-c+1))*a$
با صرف‌نظر از مقادیر ناچیز c و 1 در مقابل n داریم: $T(n)=(2\wedge n)*a\ \rightarrow T(n)\in O(2\wedge n).$

☑ **پاسخ ۱۸ :**

$$T(n)=T(n/2)+a=[T(n/4)+a]+a=T(n/4)+2a=[T(n/8)+a]+2a=$$
$$T(n/8)+3a=.....=T(n/(2\wedge k))+(k)*a$$

چون باید به شرط توقف رسیده باشیم پس باید $n/(2\wedge k)=c$ شده باشد. حال به جای $T(n/(2\wedge k))$ مقدار ثابت a را مـی‌گـذاریم. پـس از فاکتورگیری از a داریم: $T(n)=a+k*a=(k+1)a$
حال کافیست که مقدار k را بر اساس n بنوسیم. داشتیم $n/(2\wedge k)=c$ بنابراین $k=(1/c)\log n$، که پایه لگاریتم 2 است پس داریم:
$$T(n)=(a/c)\log n+a$$
با صرف‌نظر از ضریب (a/c) داریم: $T(n)\in O(\log n)$

☑ **پاسخ ۱۹ :**

$$T(n)=2*T(n/2)+a=2*[2*T(n/4)+a]+a=4*T(n/4)+2a+a=$$
$$4*[2*T(n/8)+a]+2a+a=8*T(n/8)+4a+2a+a=.....=$$
$$(2\wedge k)*T(n/(2\wedge k))+(2\wedge(k-1))a+(2\wedge(k-2))a+...+4a+2a+a$$

چون باید به شرط توقف رسیده باشیم پس باید $n/(2\wedge k)=c$ شده باشد. حال به جای $T(n/(2\wedge k))$ مقدار ثابت a را مـی‌گـذاریم. پـس از فاکتورگیری از a داریم:

$$T(n)=[(2\wedge k)+(2\wedge k-1)+(2\wedge k-2)+...+4+2+1]a$$

همانطور که می‌دانیم حاصل مقدار درون براکت طبق رابطه $\sum_{i=0}^{k}2^i=2^{(k+1)}-1$ برابـر $(2\wedge(k+1))-1$ خواهـد شـد. اگـر از مقـدار یـک‌هـا صرف‌نظر کنـیم خـواهیم داشـت: $T(n)=(2\wedge(k))*a$ حـال کافیـست کـه مقـدار k را بـر اسـاس n بنویـسیم. داشـتیم $n/(2\wedge k)=c$
بنابراین $k=(1/c)\log n$. پس داریم: $T(n)=(2\wedge((1/c)(\log n)))*a$
با صرف‌نظر از ضریب $(1/c)$ داریم (پایه لگاریتم ۲ است). $T(n)=(2\wedge\log n)*a\ \rightarrow T(n)=an\ \rightarrow T(n)\in O(n)$

☑ **پاسخ ۲۰ :**

$$T(n)=T(n-1)+a=[T(n-2)+a]+a=T(n-2)+2a=[T(n-3)+a]+2a=$$
$$T(n-3)+3a=.....=T(n-k)+ka$$

چون باید به شرط توقف رسیده باشیم پس باید $n-k=c$ شده باشد. حال به جای $T(n-k)$ مقدار ثابت a را می‌گذاریم. پس از فاکتورگیری از a داریم: $T(n)=[k+1]a$

حال کافیست که مقدار k را بر اساس n بنویسیم. داشتیم $n-k=c$ بنابراین $k=n-c$. پس داریم: $T(n)=(n-c+1)*a$

با صرف‌نظر از مقادیر ناچیز c و 1 در مقابل n داریم: $T(n)=n*a \rightarrow T(n)\in O(n).$

هنگامی که برای یک تابع بازگشتی رابطهٔ $T(n)$ به دست آمد، بدون استفاده از جایگذاری و با استفاده از روابط زیر می‌توان مرتبهٔ زمانی $T(n)$ را به دست آورد. جدول زیر نشان‌دهندهٔ این روابط می‌باشد:

مرتبه زمانی	تابع پیچیدگی
$T(n)\in O(3^n)$	$T(n)=T(n-1)+\frac{T(n-1)}{T(n-1)+a}$
$T(n)\in O(n^{\log_b^a})$	$T(n)=aT(\frac{n}{b})$
$T(n)\in O(n)$	$T(n)=T(\frac{n}{2})+T(\frac{n}{2})$
$T(n)\in O(2^{\frac{n}{2}})$	$T(n)=T(n-2)+T(n-2)$
$T(n)\in O(\log_a^n)$	$T(n)=T(\frac{n}{a})+b$
$T(n)\in O(2^n)$	$T(n)=T(n-1)+T(n-1)$
$T(n)\in O(n)$	$T(n)=\sqrt{T(\frac{n}{2})+T(\frac{n}{2})}$
$\begin{cases}T(n)\in O(a^n) & a>1\\ T(n)\in O(n) & a=1\end{cases}$	$T(n)=aT(n-1)+b$
$T(n)\in O(2^n)$	$T(n)=\frac{T(n-1)}{T(n-1)+a}$

مثال ۲۱: شمارش گامهای برنامهٔ زیر کدام است؟

```
Var      s , i : integer;
begin
      i :=1;
      while    i <= n    do
      begin
         i := 2 * i;
         s := s+i;
      end;
end.
```

۱) $3([\log_2^n]+1)+2$

۲) $3([\log_2^n]+2)$

۳) $3[\log_2^n]+4$

۴) $3([\log_2^n]+1)$

☑ **پاسخ:** گزینه «۱» مانند جدول ۱ـ ۱ با استفاده از شیوهٔ جدول گامها، داریم:

کل مراحل	تکرار	s/e
0	1	0
1	1	1
$[\log_2^n]+2$	$[\log_2^n]+2$	1
0	1	0
$[\log_2^n]+1$	$[\log_2^n]+1$	1
$[\log_2^n]+1$	$[\log_2^n]+1$	1
0	1	0
0	1	0

$3[\log_2^n]+5$: مجموع همهٔ مراحل

```
Var      s , i : integer;
begin
      i :=1
      while    i <= n    do
      begin
         i := 2 * i;
         s := s+i;
      end;
end.
```

مثال۲۲: مقدار تابع $a(4,3)$ کدام است؟

$$a(n,m)=\begin{cases}1 & \text{اگر } n=0 \quad n=m\\ a(n-1,m)+a(n-1,m-1) & \text{در غیر این صورت}\end{cases}$$

۱) 3 ۲) 4 ۳) 5 ۴) 6

☑ **پاسخ:** گزینه «۳»

$$a(4,3)=a(3,3)+a(3,2)=1+4=5$$
$$a(3,3)=1$$
$$a(3,2)=a(2,2)+a(2,1)=1+3=4$$
$$a(2,2)=1$$
$$a(2,1)=a(1,1)+a(1,0)=1+2=3$$
$$a(1,1)=1$$
$$a(1,0)=a(0,0)+a(0,-1)=1+1=2$$
$$a(0,0)=1$$
$$a(0,-1)=1$$