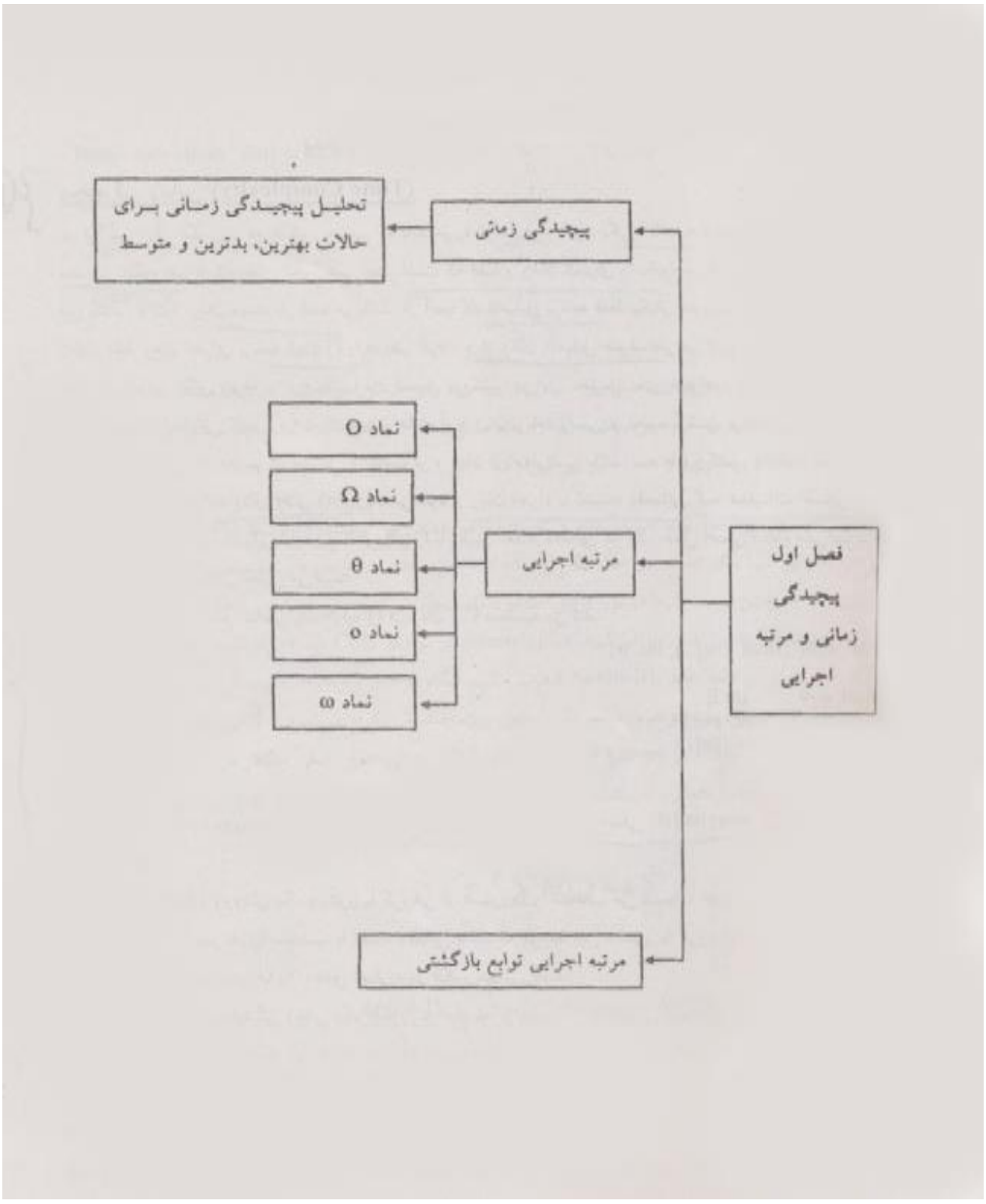
فصل اول

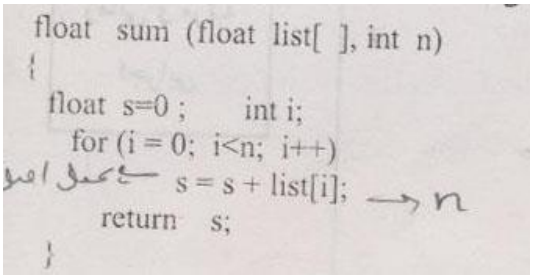
پیچیدگی زمانی و

مرتبه اجرایی


فصل اول
پیچیدگی
زمانی و مرتبه
اجرایی
پیچیدگی زمانی
تحلیل پیچیدگی زمانی برای
حالات بهترین، بدترین و متوسط
مرتبه اجرایی
نماد O
نماد Ω
نماد θ
نماد o
نماد ω
مرتبه اجرایی توابع بازگشتی

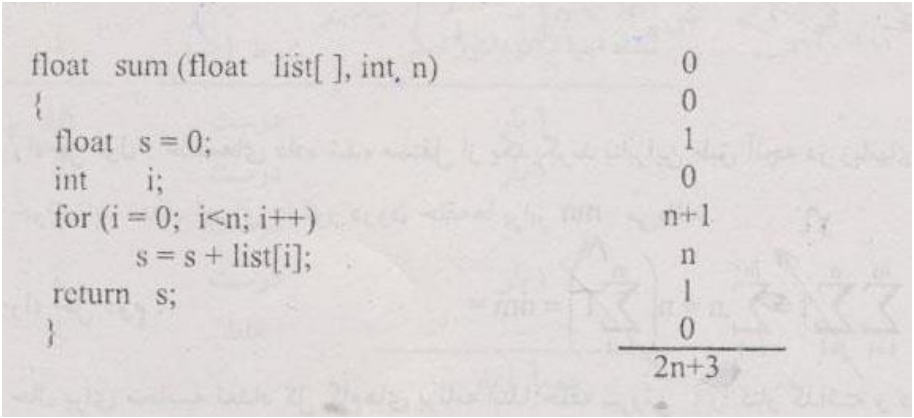
پیچیدگی زمانی (Time Complexity)

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



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

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



نکته : هنگام محاسبه تعداد دفعاتی که یک دستور درون حلقه ها اجرا می گردد می توان از فرمول های زیر استفاده کرد :

عدد ثابتی است

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=1}^{n} kf(i) = k\sum_{i=1}^{n} f(i)$$

K

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

a_1 جمله اول ، a_n جمله آخر و n تعداد جملات است : $= \frac{n(a_1+a_n)}{2}$ جمع تصاعدی عددی راه دیگر *Trace* کردن برنامه است.

مثال ۳ :

دستور اصلی x : x+1 *در تکه* برنامه زیر چند بار اجرا می شود ؟ تعداد کل گام های برنامه چه قدر است ؟

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

راه حل اول :

حلقه های داده شده مستقل از یکدیگرند بنابراین طبق آن چه در زبان های برنامه نویسی پاسکال و *c* خوانده اید تعداد اجرای دستور درون حلقه ها برابر *mn* می باشد.

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

راه حل دوم :

حال برای محاسبه تعداد کل گام های برنامه ابتدا حلقه بیرونی *i* را کنار گذاشته و در نظر نمی گیریم . در این حال دستور *x : x+1* به تعداد *n* بار و عبارت *for j* به تعداد *(n+1)* بار اجرا می گردد.یعنی تعداد کل *(n+1+n)* . حال خود این دو خط درون حلقه *i* بوده و به تعداد *m* بار اجرا می شوند یعنی m(n+1)+mn . *از آنجا که عبارت* for i *نیز* m+1 *بار اجرا می شود ، پس :*

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

مثال ۴ :

دستور اصلی *x : x+1* در تکه برنامه زیر چند بار اجرا می شود ؟

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

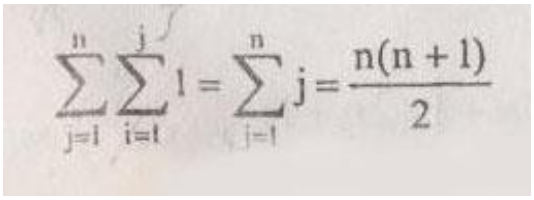
راه حل اول :

حلقه های داده شده به یکدیگر وابسته اند:

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

راه حل دوم:



مثال ۵ :

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;
```

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

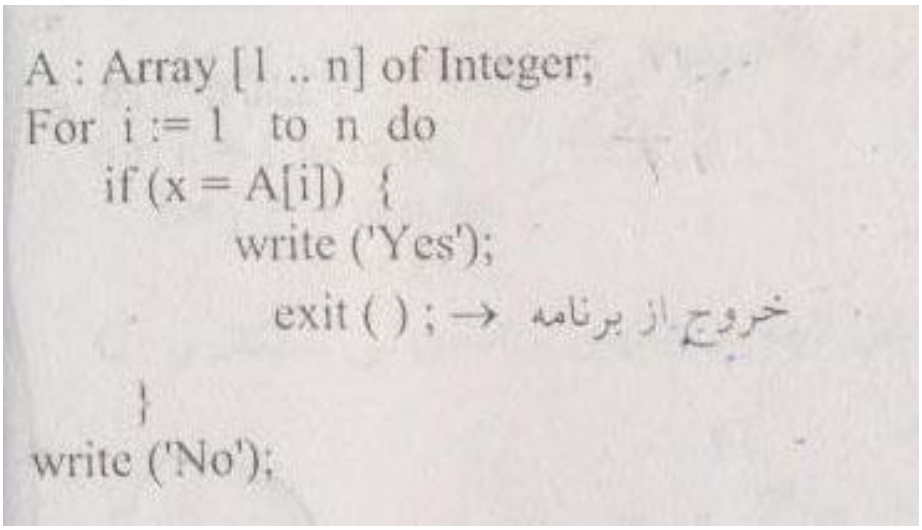
i	شرط i >1	تعداد اجرا شدن دستور اصلی
14	درست	۱ بار
7	درست	۱ بار
3	درست	۱ بار
1	غلط	-
		جمعاً ۳ بار

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 می باشد. ولی در الگوریتمی مثل جستجوی خطی (ترتیبی) تابع پیچیدگی برای حالات مختلف ممکن است متفاوت باشد . فرض گنید در آرایه n خانه ای A می خواهیم خانه به خانه از اول تا انتها به دنبال عدد معین x بگردیم.اگر x را در آرایه A پیدا کردیم بگوییم Yes و در غیر اینصورت بگوییم No :



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

قضیه اصلی :

اگر $f(n) = a_m n^m + \cdots . + a_1 n + a_0$ باشد آنگاه

$$f(n) = O(n^m)$$

مثال ۶ :

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

مثال ۷ : مرتبه اجرایی برنامه های زیر را به دست 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
        x:= x + 1 ;
```
$\Rightarrow O(n^2)$

مثال ۸ :

تعداد دقیق اجرا شدن دستور write (OK) و مرتبه اجرایی ان را در تکه برنامه زیر به دست آورید.

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

حل : حلقه های فوق وابسته به یکدیگر بوده و همانطور که قبلاً دیدید تعداد دقیق اجرا شدن دستور write برابر جمع تصاعد عددی از ۱ تا n می باشد یعنی $\frac{n(n+1)}{2}$ و بنابر قضیه اصلی مرتبه اجرایی آن $O(n^2)$ می باشد.

مثال ۹ :

مرتبه اجرایی زیر $O(1)$ میباشد چرا که حلقه به تعداد معین و محدودی اجرا می شود :

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

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

مثال ۱۰ :

مرتبه اجرایی برنامه زیر چیست ؟

فرض کنید n= 32 آنگاه

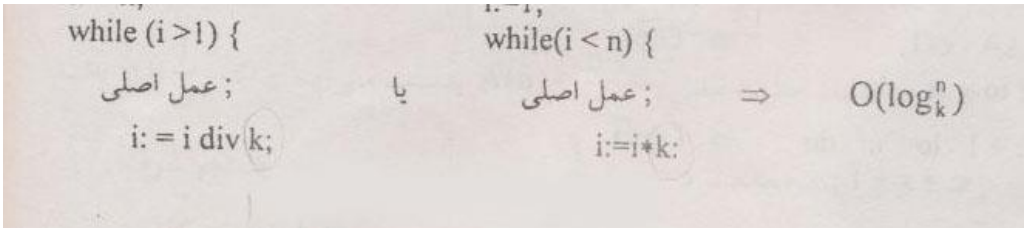
```
x : = 0;
i : = n ;
while (i > 1) do begin
        x : = x + 1;
        i : = i div 2;
end;
```

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 می شود ($5 = \log_2 32$) . پس در حالت کلی مرتبه اجرایی الگوریتم فوق برابر O(log) می باشد.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 ۱ تغییر می کند .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 ۱ تا n تغییر کند باز هم مرتبه اجرایی آن ($O(\log_k n)$ می باشد:



نکته ۳ :

در جدول زیر مرتبه اجرایی 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(logn)	O(n)	O(nlogn)	$O(n^2)$	$O(2^n)$	O(n!)

نکته ۴ :

نماد O زمان اجرای الگوریتم را برای حد بالای n نشان می دهد و برای n های کمتر از حد خاصی ممکن است اطلاعات مناسبی را به ما ندهد.مثلاش زمان اجرای یک الگوریتم $1000n^2$ و زمان اجرای الگوریتمی دیگر $10n^3$ می باشد.

با نماد O الگوریتم اول از مرتبه $O(n^2)$ و الگوریتم از مرتبه $O(n^3)$ است.ولی برای n های کمتر از ۱۰۰ الگوریتم دوم سریعتر از الگوریتم اول خواهد بود :

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

لذا هنگام مقایسه دقیق سرعت اجرای الگوریتم ها به محدوده n نیز باید توجه داشته باشیم.

نکته ۵ :

برنامه هایی با پیچیدگی نمایی تنها برای مقادیر کوچک n (اغلب $n \leq 40$) سودمند هستند.

نکته ۶ :

از نظر عملی برای n های بزرگ $(n \geq 100)$ تنها برنامه هایی با پیچیدگی کم (مانند n^2 ,nlogn, n و n^3) سودمند می باشند.

نکته ۷ :

برای اعداد صحیح a , b,r بزرگتر از صفر از نظر مرتبه داریم

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

نکته ۸ :

اگر $O(n_2)$ ، t_1 ، 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 ، v_2 به ترتیب مرتبه اجرایی و زمان اجرایی الکوریتمی 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 ۱۱ :

الگوریتمی با مرتبه زمانی $o(n \log n)$ در کامپیوتری در مدت زمان ۱ ثانیه اجرا می شود . همان الگوریتم روی کامپیوتر دیگری با سرعت ۱۰۰ برابر در چه مدت زمانی اجرا خواهد شد؟

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

مثال ۱۲ :

برنامه ای با اندازه ۱۰ و مرتبه اجرایی $O(n^2)$ روی سیستمی در مدت زمان 1 msec اجرا می شود.

همان مسئله با اندازه ۱۰۰ روی همان کامپیوتر در چه مدت زمان اجرا می شود؟

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

نمادهای Ω و θ (امگای بزرگ و تتا)

به صورت صوری می توان عبارت زیر را برای تعریف O در نظر گرفت :

$$f \le g \approx f(n) = O(g(n))$$

عکس تعریف O ، تعریف Ω می باشد که به صورت صوری معادل عبارت زیر است :

$$f \geq g \approx f(n)=\Omega(g(n))$$

یعنی Ω حد پایین را برای تابع مشخص می کند.

مثال ۱۳ :

برای $f(n) = 5n^3 + 2n$ عبارات زیر همگی درست هستند :

$$f(n)=\Omega(n)\ ,\quad f(n)=\Omega(n^2)\ ,\quad f(n)=\Omega(n^3)$$

ولی اغلب منظور از Ω در مثال فوق همان($f(n) = \Omega(n^3$ است.بنابراین تعاریف O و Ω چندان دقیق نبوده و به همین دلیل اغلب از مفهوم θ استفاده می شود که مطابق تعریف صوری زیر است :

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

تذکر : تعاریف Ω و θ در کتاب ساختمان داده آورده شده است ولی تا همین حد که در اینجا بیان شد برای درس طراحی الگوریتم کفایت می کند.

مثال ۱۴ :

برای $f(n) = 5n^3 + 5n$ عبارات زیر درست هستند :

$f(n)=O(n^3)$, $f(n)=\Omega(n^3)$, $f(n)=\theta(n^3)$

$f(n)=O(n^5)$, $f(n)=\Omega(n)$

ولی عبارات زیر غلط هستند :

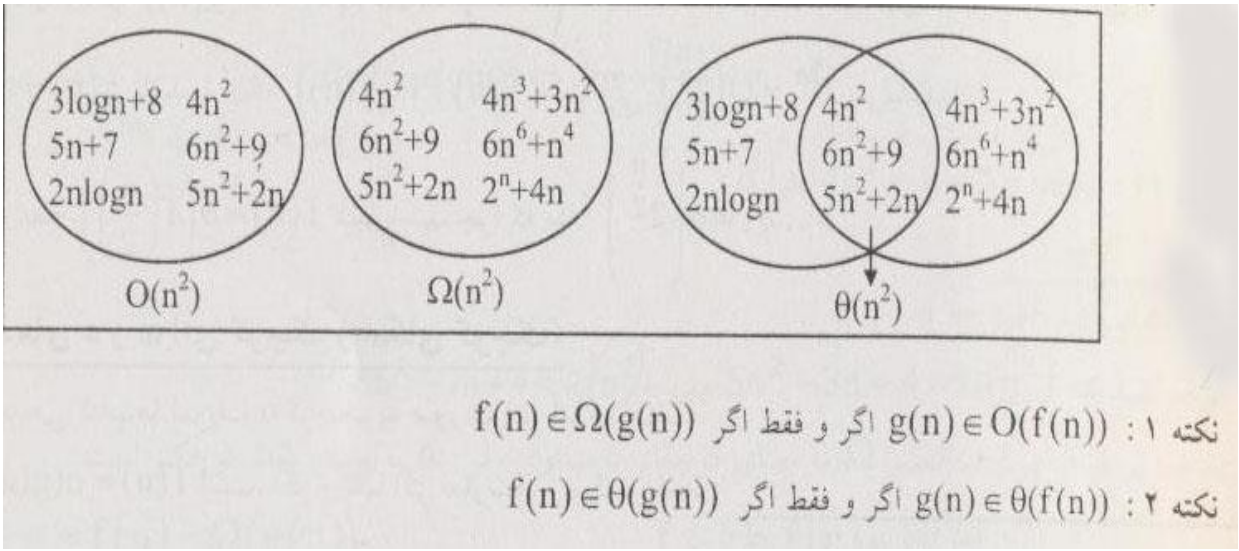
$f(n)=\theta(n^5)$, $f(n)=\theta(n)$

تذکر : اغلب در تست های کنکور منظور از نماد O همان θ می باشد.

در واقع با توجه به تعاریف فوق داریم :

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

مثال ۱۵ :



نمادهای o و ω (ای کوچک و امگای کوچک)

در برخی از کتاب ها 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 ۱۹ :

$3n+2 = O(n^2)$ است چرا که :

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

در واقع می توان گفت اگر O بزرگ به معنای بزرگ تر یا مساوی ولی ای کوچک به معنای فقط بزرگ تر است.

$$f \le g \approx f(n) = O(g(n)) \qquad f < g \approx f(n) = o(g(n))$$

مثال ۲۰ :

مشابه فوق می توان 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 ۲۱ :

$$5n^2 - 3n + 4 = O(n^2) \quad \text{ولی} \quad 5n^2 - 3n + 4 \neq o(n^2)$$

نکته : همان طور که قبلا گفتیم دسته های پیچیدگی معروف به ترتیب از چپ به راست عبارت اند 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) ϵ o(f(n))

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

زیرا $\frac{a}{b} < 1$

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

بر عکس o کوچک , می توان 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

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

مقایسه خواص $\omega , o, \theta, , \Omega, O,$:

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 :

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

مثال ۲۳ :

مرتبه اجرایی الگوریتم بازگشتی محاسبه فاکتوریل را به دست آورید ؟

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

همان طور که مشاهده می شود به ازای *n=4* تابع بوده تعداد ۳ بار خودش را فراخوانی می کند که با احتساب بار اول یعنی *fact(4)* تابع مذکور ۴ بار فراخوانی می شود .

پس در حالت کلی به ازای *n* , تابع *n* بار فراخوانی می شود . لذا مرتبه اجرایی این الگوریتم *T(n) = O(n)* می باشد و در واقع مرتبه تعداد گره های درخت بازگشتی , همان مرتبه اجرایی الگوریتم است.

```
int fact (int n)
 {
    if (n == 1) return 1;
      return (n*fact(n-1));
 }
    fact(4)
      |
 4*fact(3)
      |
 3*fact(2)
      |
 2*fact(1)
```

$$\begin{cases} T(n) = 1 & \text{اگر } n = 1 \\ T(n) = T(n-1) + C & \text{در غیر اینصورت} \end{cases}$$

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

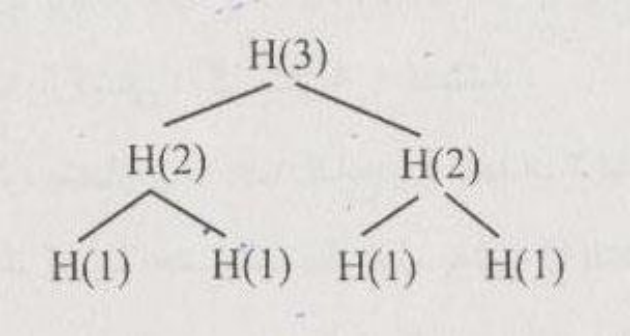
$$T(n) = T(n-1) + C = T(n-2) + 2C = T(n-3) + 3C = \dots$$
$$= T(1) + (n-1)C = 1 + (n-1)C = O(n)$$

مثال ۲۴ : مرتبه اجرایی الگوریتم بازگشتی برج های هانوی را به دست 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 گره های درخت ۷ عدد است :



به همین ترتیب اگر $n=4$ باشد تعداد کل گره های درخت بازگشتی ۱۵ می شود و در حالت کلی برای n تعداد کل گره ها $2^n - 1$ یعنی از مرتبه $O(2^n)$ است.

تذکر : تعداد کل گره های درخت دودویی پر با n سطح , از مرتبه $O(2^n)$ می باشد.

روش دوم :

اگر *n=1* باشد فقط یک انتقال صورت می گیرد . در غیر اینصورت علاوه بر انتقال تابع دو بار با مقدار *n-1* فراخوانی می گردد.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 می کنیم :

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

مثال ۲۵ :

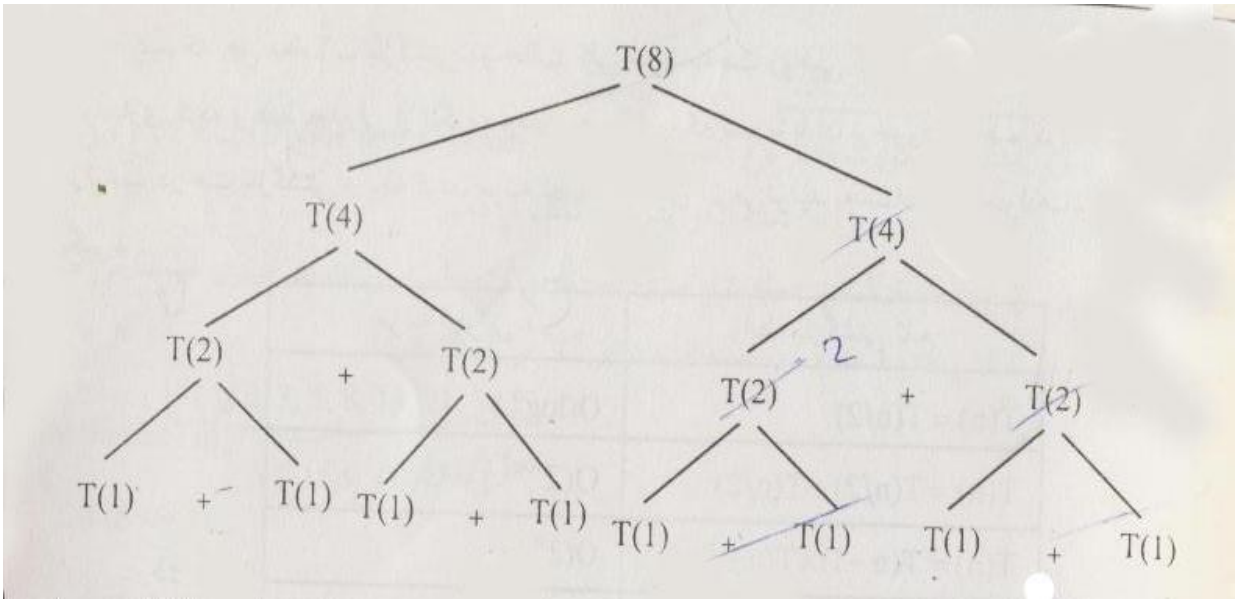
مرتبه اجرایی تابع زیر کدام است؟

```
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 :



روش تقسیم و غلبه و روش پویا

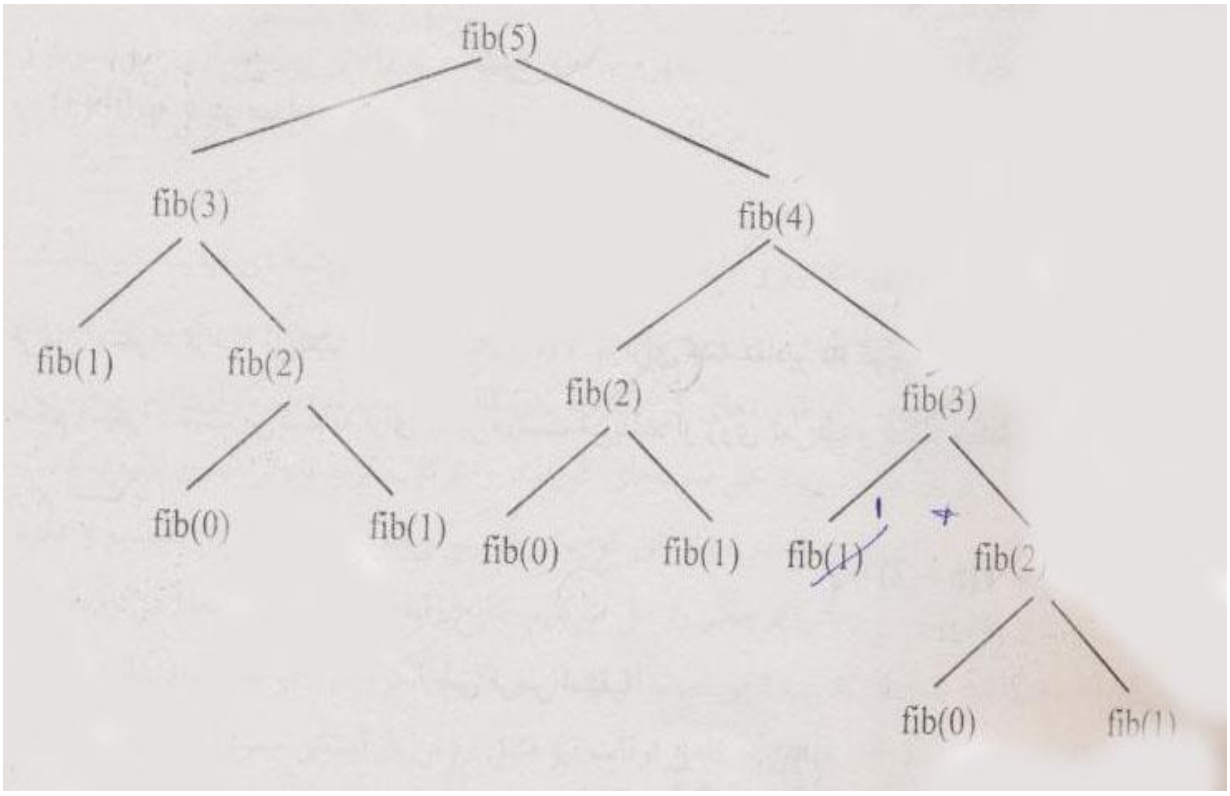
مثال ۲۶ :

تابع زیر به صورت بازگشتی جمله *n* ام سری فیبونانچی را محاسبه می کند . در سری فیبونانچی هر جمله جمع دو جمله قبلی خود است و دو جمله اولیه برابر *"1"* می باشد یعنی

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

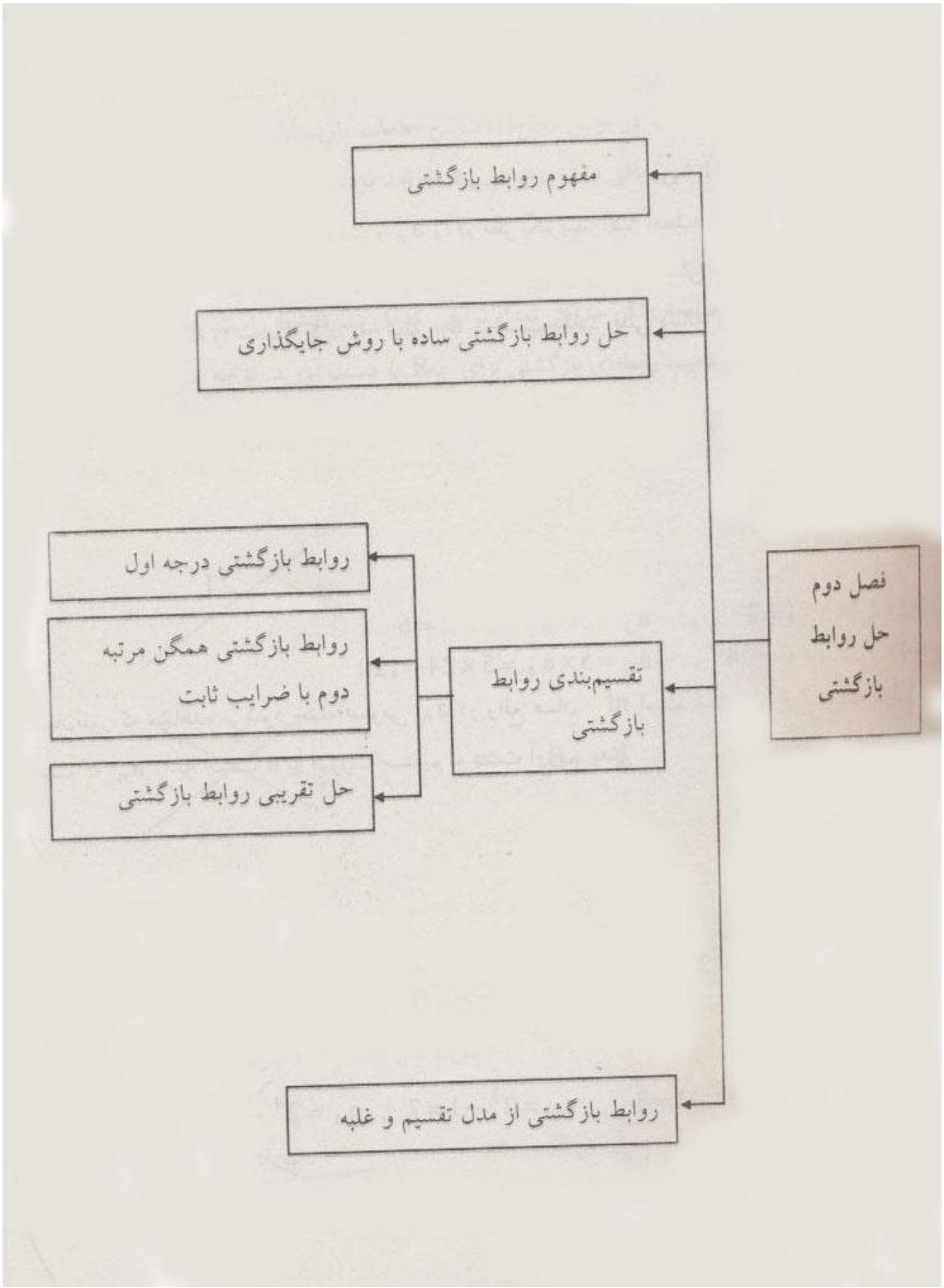
```
int fib (int  n)
{
   if (n <= 1)
         return n;
   else
      return fib (n-1) + fib(n-2);
```

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


fib(5)
fib(3)
fib(4)
fib(1)
fib(2)
fib(2)
fib(3)
fib(0)
fib(1)
fib(0)
fib(1)
fib(1)
fib(2
fib(0)
fib(1)

فصل دوم

روابط بازگشتی



مفهوم روابط بازگشتی

مثال ۱ :

در رابطه بازگشتی زیر مقادیر a_2 تا a_5 را به دست آورید:

$$\begin{cases} a_n = n a_{n-1} \\ a_1 = 1 \end{cases}$$

حل :

$$a_2 = 2 \times a_1 = 2 \times 1 = 2 \quad , \quad a_3 = 3 \times a_2 = 3 \times 2 = 6$$
$$a_4 = 4 \times a_3 = 4 \times 6 = 24 \quad , \quad a_5 = 5 \times a_4 = 5 \times 24 = 120$$

همان طور که مشاهده می شود جمله عمومی a_n در واقع همان $n!$است. منظور از حل رابطه بازگشتی آن است که a_n را بر حسب n به صورت مستقیم به دست آوریم یعنی :

$$\begin{cases} a_n = n a_{n-1} \\ a_1 = 1 \end{cases} \Rightarrow a_n = n!$$

مثال ۲ :

با مثال نشان دهید که :

$$\begin{cases} a_n = 2a_{n-1} + 1 \\ a_1 = 1 \end{cases} \Rightarrow a_n = 2^n - 1$$

حل :

از روی رابطه مستقیم $a_n = 2^n - 1$ داریم :

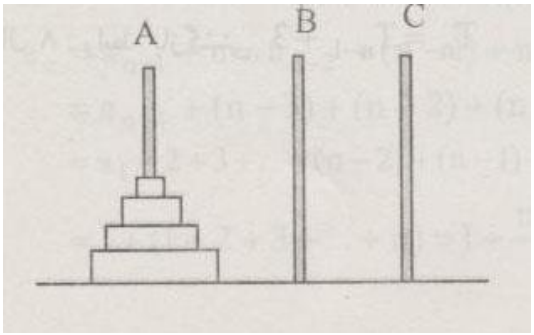
$$a_1 = 2^1 - 1 = 1 \ , \ a_2 = 2^2 - 1 = 3 \ , \ a_3 = 2^3 - 1 = 7 \ , \ a_4 = 2^4 - 1 = 15$$

از روی رابطه بازگشتی داریم :

$$a_2 = 2a_1 + 1 = 2 \times 1 + 1 = 3 \ , \ a_3 = 2 \times a_2 + 1 = 2 \times 3 + 1 = 7$$
$$a_4 = 2a_3 + 1 = 2 \times 7 + 1 = 15$$

مثال۶ :

مساله استاندارد برج های هانوی : این مسئله را در درس ساختمان داده ها خوانده اید که می خواهیم n دیسک با قطرهای متمایز را از روی میله A به میله C و به کمک میله B انتقال دهیم.



در این جا دو محدودیت داریم یکی آن که در هر بار انتقال فقط یک دیسک را می توان منتقل ساخت و دوم اینکه یک دیسک با قطر بزرگ تر نباید روی دیسکی با قطر کوچک تر قرار گیرد .

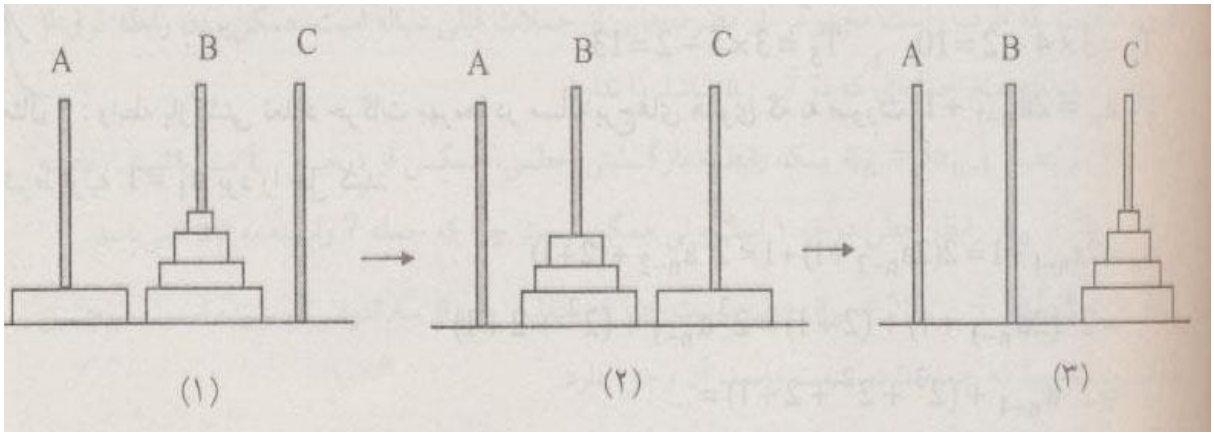
الگوریتم بازگشتی این مسئله به صورت زیر است :

۱- ابتدا (n-1) دیسک را با رعایت دو شرط گفته شده از میله A به B انتقال می دهیم.

۲- سپس دیسک آخر را از میله A به C می بریم.

۳- (n-1) دیسک موجود را در میله B را با رعایت دو شرط گفته شده از میله B به C منتقل می کنیم.

۴-



پس بدیهی است که اگر تعداد حرکات لازم برای انتقال n دیسک را با H_n نمایش دهیم , این میزان برابر است با :

$$H_n = H_{n-1} + 1 + H_{n-1} = 2H_{n-1} + 1$$

و شرط اولیه $H_1 = 1$ می باشد.

حل روابط بازگشتی ساده با روش جایگذاری

برای یک دسته از مسائل ساده مثل حالتی که a_n فقط به a_{n-1} بستگی داشته باشد می توان از روش جایگذاری روابط بازگشتی را حل کرد.

مثال ۷ :

رابطه بازگشتی $a_n = na_{n-1}$ را با شرط اولیه $a_1 = 1$ حل کنید.

$$a_2 = 2a_1 = 2\times(1) = 2! \quad , \quad a_3 = 3a_2 = 3\times(2\times1) = 3!$$
$$a_4 = 4a_3 = 4\times(3\times2\times1) = 4!$$

$$a_5 = 5a_4 = 5\times 4! = 5!$$

پس در حالت کلی داریم $a_n = n!$

مثال ۸ : رابطه بازگشتی $T_n = T_{n-1} + 3 \quad (n \geq 2)$ با شرط اولیه $T_1 = 1$

$a_5 = 5a_4 = 5\times4! = 5!$

پس در حالت کلی داریم $a_n = n!$

مثال ۸ : رابطه بازگشتی $T_n = T_{n-1} + 3 \quad (n \geq 2)$ با شرط اولیه $T_1 = 1$ را حل کنید.

$$T_n = T_{n-1} + 3 = (T_{n-2} + 3) + 3 = T_{n-2} + 2\times3$$
$$= (T_{n-3} + 3) + 2\times3 = T_{n-3} + 3\times3$$
$$= (T_{n-4} + 3) + 3\times3 = T_{n-4} + 4\times3$$
$$= \ldots = T_{n-(n-1)} + (n-1)\times3 = T_1 + (n-1)3$$
$$= 1 + 3(n-1) = 3n - 2$$

تا به کمک رابطه بازگشتی داریم :

$$T_2 = T_1 + 3 = 1 + 3 = 4$$
$$T_3 = T_2 + 3 = 4 + 3 = 7$$
$$T_4 = T_3 + 3 = 7 + 3 = 10$$
$$T_5 = T_4 + 3 = 10 + 3 = 13$$

همین مقادیر فوق به کمک رابطه مستقیم $T_n = 3n - 2$ بدست می‌آیند :

$$T_2 = 3\times2 - 2 = 4 \quad , \quad T_3 = 3\times3 - 2 = 7$$
$$T_4 = 3\times4 - 2 = 10 \quad , \quad T_5 = 3\times5 - 2 = 13$$

مثال ۹ :

رابطه بازگشتی تعداد حرکات مهره ها در مسئله برج های هانوی که به صورت $a_n = 2a_{n-1} + 1$ و با شرط اولیه $a_1 = 1$ بوده را حل کنید.

$$a_n = 2a_{n-1} + 1 = 2(2a_{n-2} + 1) + 1 = 2^2 a_{n-2} + (2+1)$$
$$= 2^2(2a_{n-3} + 1) + (2+1) = 2^3 a_{n-3} + (2^2 + 2 + 1)$$
$$= 2^4 a_{n-4} + (2^3 + 2^2 + 2 + 1) = \ldots$$
$$= 2^{n-1} a_1 + (2^{n-2} + 2^{n-3} + \ldots + 2 + 1)$$

می‌دانیم که $a_1 = 1$ است :

$$= 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2 + 1$$
$$\frac{t_1(q^n - 1)}{q - 1}$$

می دانیم جمله تصاعد هندسی n جمله , با جمله اول t_1 و با ضریب تصاعد q برابر است با : $\frac{t_1(q^n-1)}{q-1}$

در تصاعد هندسی فوق t_1= ۱ , $q=2$ و تعداد جملات n می باشد , لذا :

$$a_n = \frac{1 \times (2^n - 1)}{2 - 1} = 2^n - 1 \Rightarrow a_n = 2^n - 1$$

مثلاً برای انتقال ۴ مهره به 2^4-1=15 *حرکت نیاز داریم.*

مثال ۱۰ :

تعداد نواحی ایجاد شده توسط n خط متقاطع که با رابطه بازگشتی $a_n = a_{n-1} + n \quad (n \geq 2)$ و شرط اولیه $a_1 = 2$ (مطابق مثال ۴) بیان می شد را به صورت مستقیم محاسبه کنید.

$$a_n = a_{n-1} + n = a_{n-2} + (n-1) + n = a_{n-3} + (n-2) + (n-1) + n$$
$$= a_{n-4} + (n-3) + (n-2) + (n-1) + n = \dots$$
$$= a_1 + 2 + 3 + \dots + (n-2) + (n-1) + n = 2 + 2 + 3 + 4 + \dots + n$$
$$= 1 + (1 + 2 + 3 + \dots + n) = 1 + \frac{n(n+1)}{2}$$

تقسیم بندی روابط بازگشتی

در حالت کلی یک روش عمومی جهن حل تمام روابط بازگشتی وجود ندارد ولی در ادامه حل چند دسته از روابط بازگشتی معروف و پر استفاده را شرح می دهیم.

تعریف : رابطه زیر را یک رابطه بازگشتی خطی از درجه (یا مرتبه) k با ضرایب ثابت می نامیم :

$$a_n = C_1 a_{n-1} + C_2 a_{n-2} + \dots + C_k a_{n-k}$$

که $C_k, \dots \dots, C_1, C_1$ اعداد ثابت حقیقی بوده و $C_k \neq 0$ است . توجه کنید که C_iها وابسته به n نیستند . این رابطه از درجه k است چرا که a_n بر حسب k جمله قبلی خود بیان شده است.

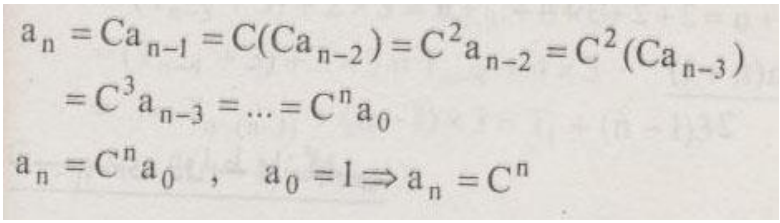
خطی بودن رابطه فوق از آن جهت است که طرف راست مجموعی از مضرب هایی از جملات قبلی دنباله است . همگن بودن رابطه فوق از آن جهت است که هیچ دو جمله ای که در آن a_j نباشد را ندارد.

حل رابطه بازگشتی درجه اول :

عموماً روابط بازگشتی درجه اول با روش جایگذاری به سادگی حل می شوند. این جایگذاری می تواند از آخر به اول یا از اول به آخر باشد.

مثال ۱۳ :

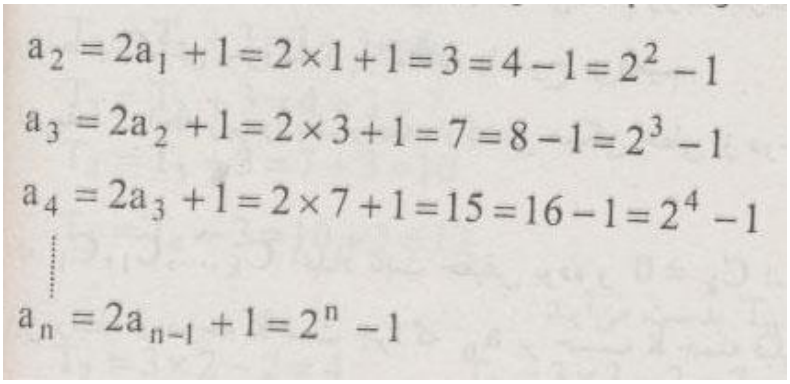
رابطه $a_n = Ca_{n-1}$ را حل کنید که C عدد ثابتی است و شرط اولیه $a_0=1$ می باشد.



$$a_n = Ca_{n-1} = C(Ca_{n-2}) = C^2 a_{n-2} = C^2(Ca_{n-3})$$
$$= C^3 a_{n-3} = \ldots = C^n a_0$$
$$a_n = C^n a_0 \quad , \quad a_0 = 1 \Rightarrow a_n = C^n$$

مثال ۱۴:

رابطه $a_n = 2a_{n-1} + 1$ که در آن $n \geq 2$ و $a_1 = 1$ را حل کنید.



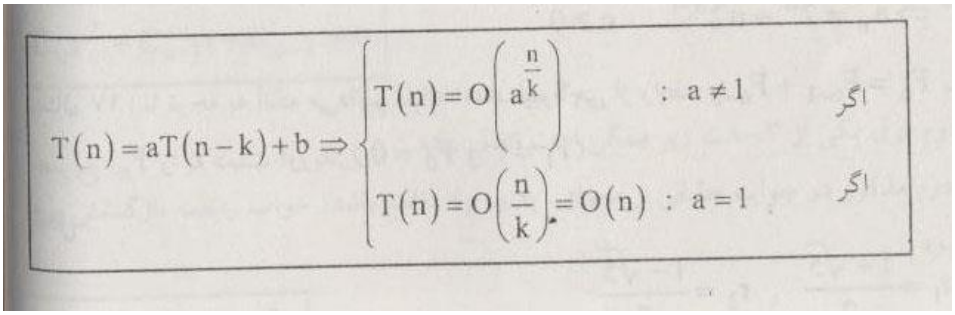
$$a_2 = 2a_1 + 1 = 2 \times 1 + 1 = 3 = 4 - 1 = 2^2 - 1$$
$$a_3 = 2a_2 + 1 = 2 \times 3 + 1 = 7 = 8 - 1 = 2^3 - 1$$
$$a_4 = 2a_3 + 1 = 2 \times 7 + 1 = 15 = 16 - 1 = 2^4 - 1$$
$$\vdots$$
$$a_n = 2a_{n-1} + 1 = 2^n - 1$$

حل تقریبی روابط بازگشتی

حل دقیق روابط بازگشتی در درس ساختمان گسسته مطرح می شود و ما از ذکر مجدد آن در اینجا خودداری می کنیم .

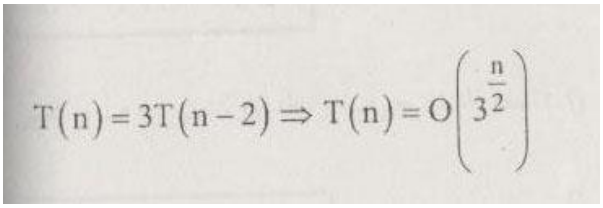
در اینجا تنها به حل تقریبی دسته ای از روابط بازگشتی مهم به کمک نمادهای O یا θ می پردازیم .

در حالت کلی داریم :



$$T(n)=aT(n-k)+b \Rightarrow \begin{cases} T(n)=O\left(a^{\frac{n}{k}}\right) & : a\neq 1 \quad \text{اگر} \\ T(n)=O\left(\frac{n}{k}\right)=O(n) & : a=1 \quad \text{اگر} \end{cases}$$

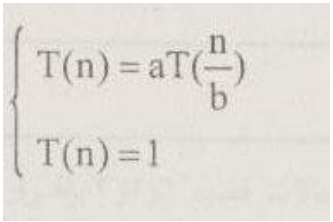
مثال ۱۹ :



$$T(n)=3T(n-2) \Rightarrow T(n)=O\left(3^{\frac{n}{2}}\right)$$

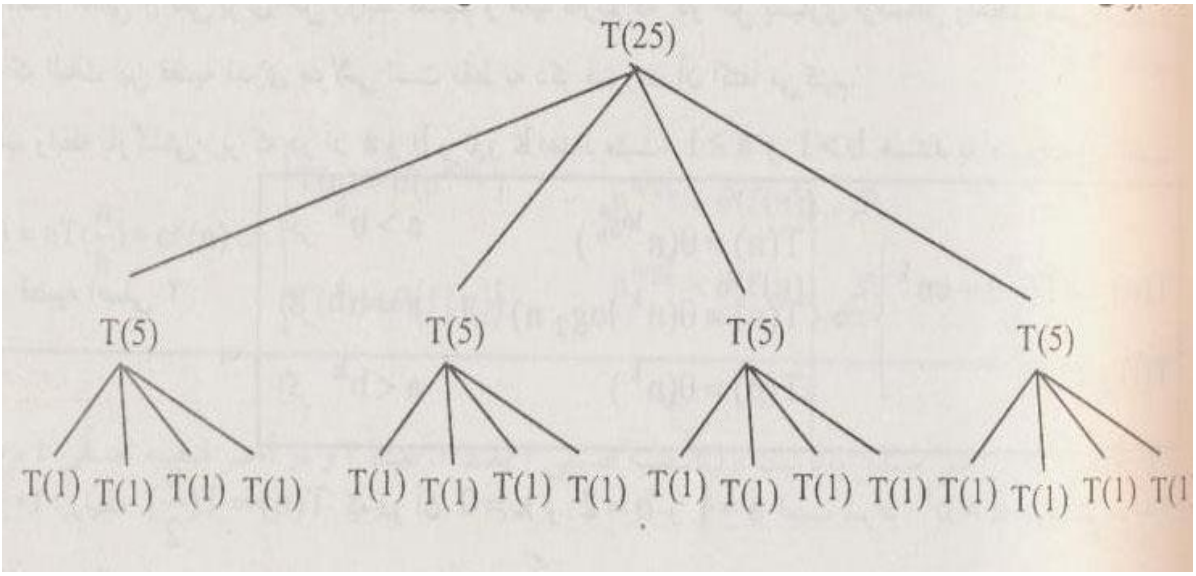
روابط بازگشتی از مدل تقسیم و غلبه

پیچیدگی زمانی بسیاری از الگوریتم های بازگشتی به فرم کلی رابطه بازگشتی زیر است :



$$\begin{cases} T(n)=aT(\frac{n}{b}) \\ T(n)=1 \end{cases}$$

مثلاً برای a = 4 و n = 25 تعداد صدا زدن ها به شکل زیر است :



ارتفاع درخت شکل فوق برابر ۲ است یعنی $\log_5 25 = 2$ که در حالت کلی $\log_b n = 2$ می شود . در هر سطح هر گره به ۴ گره تقسیم می شود که در حالت کلی هر گره به a انشعاب تقسیم می گردد. پس :

تعداد کل صدا زدن‌ها در درخت درجه a به ارتفاع h $= 1 + a + a + a^2 + \ldots + a^h$

$$= \frac{1 \times (a^{h+1} - 1)}{a - 1} \quad (a \neq 1)$$

توجه کنید که h را برابر $\log_b^n$ در نظر گرفته‌ایم و ارتفاع، تعداد سطوح منهای یک است.

تعداد کل صدازدن‌ها $= \frac{a^{h+1} - 1}{a - 1} = \frac{a^{1+\log_b^n} - 1}{a - 1}$

توجه کنید که h را برابر $\log_b n$ در نظر گرفته ایم و ارتفاع , تعداد سطوح منهای یک است.

پس در حالت کلی داریم :

قضیه اصلی ۱ :

$$\begin{cases} T(n) = aT(\frac{n}{b}) \\ T(1) = 1 \end{cases} \Rightarrow \begin{cases} \theta\left(a^{\log_b^n}\right) = \theta(n^{\log_b^a}) & (a \neq 1) \\ \theta(\log_b^n) & (a = 1) \end{cases}$$

قضیه اصلی ۲ :

$$\left.\begin{aligned} T(n) &= aT(\frac{n}{b}) + cn^k \\ T(1) &= c \end{aligned}\right\} \Rightarrow \begin{cases} T(n) = \theta(n^{\log_b^a}) & \text{اگر } a > b^k \\ T(n) = \theta(n^k \log_2 n) & \text{اگر } a = b^k \\ T(n) = \theta(n^k) & \text{اگر } a < b^k \end{cases}$$

قضیه اصلی ۳ : در رابطه بازگشتی $T(n) = aT(\frac{n}{b}) + cf(n)$ اگر $f(n)$ به صورت $\theta(n^k)$ باشد مشابه قضیه اصلی ۲ حل می‌شود ولی اگر $f(n)$ به صورت $f(n) = \theta\left(n^{\log_b^a} \times (\log n)^k\right)$ باشد آن‌گاه $T(n) = \theta\left(n^{\log_b^a} \times (\log n)^{k+1}\right)$ می‌شود.

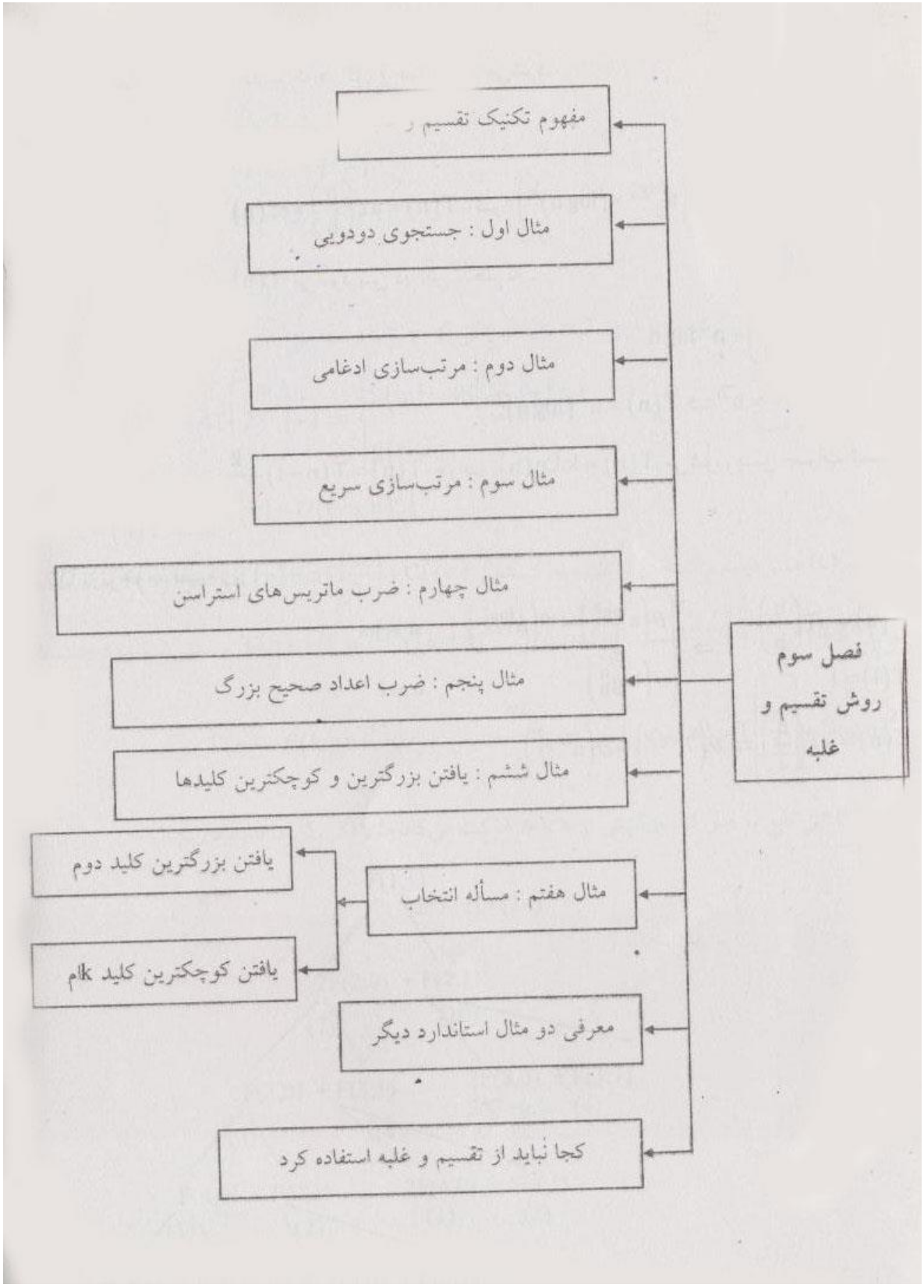
قضیه اصلی ٤ : در رابطه بازگشتی $T(n) = aT\left(\frac{n}{b}\right) + cf(n)$ اگر هیچکدام از حالات قضیه ۱و۲و۳ برقرار نباشد آن‌گاه :

$$T(n) = aT(\frac{n}{b}) + cf(n) \Rightarrow \begin{cases} T(n) = \theta(n^{\log_b^a}) & \text{اگر } n^{\log_b^a} > \theta(f(n)) \\ T(n) = \theta\left(f(n)\right) & \text{اگر } n^{\log_b^a} < \theta(f(n)) \end{cases}$$

فصل سوم

روش تقسیم و غلبه

(Divide and Conquer)


فصل سوم
روش تقسیم و
غلبه
مفهوم تکنیک تقسیم و
مثال اول : جستجوی دودویی
مثال دوم : مرتب‌سازی ادغامی
مثال سوم : مرتب‌سازی سریع
مثال چهارم : ضرب ماتریس‌های استراسن
مثال پنجم : ضرب اعداد صحیح بزرگ
مثال ششم : یافتن بزرگترین و کوچکترین کلیدها
مثال هفتم : مسأله انتخاب
یافتن بزرگترین کلید دوم
یافتن کوچکترین کلید nام
معرفی دو مثال استاندارد دیگر
کجا نباید از تقسیم و غلبه استفاده کرد

مفهوم تکنیک تقسیم و غلبه

نامگذاری این روش از یک تکنیک جنگی که ناپلئون به کار برد گرفته شده است. در سال ۱۸۰۵ ارتشی از سربازان روسی و اتریشی با بیش از ۱۵ هزار نفر به جنگ ناپلئون آمدند . در این حال ناپلئون به قلب سپاه آنان حمله کرده و با تقسیم نیروهای دشمن به دو بخش بر آنها پیروز شد.

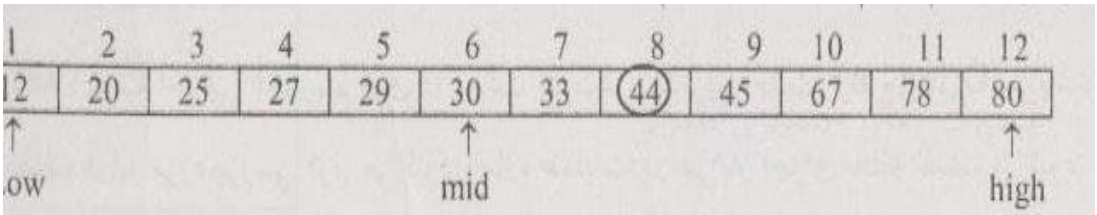
در واقع ناپلئون با تقسیم (Divide) سپاهی بزرگ به دو سپاه کوچک تر و پیروز شدن بر تک تک آنها موفق شد بر آن سپاه بزرگ غلبه یافته و آن را فتح کند . (Conquer)

مثال اول : جستجوی دودویی

جستجوی دودویی فقط در آرایه های مرتب استفاده می شود . در این روش عنصر مورد نظر با خانه وسط آرایه مقایسه می شود. اگر با این خانه برابر بود جستجو تمام می شود .

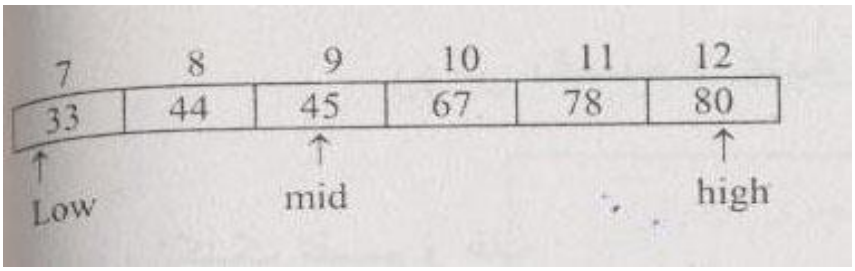
اگر عنصر مورد جستجو از خانه وسط بزرگ تر بود جستجو در بخش بالایی آرایه و در غیر اینصورت جست و جو در بخش پایینی آرایه انجام می شود (فرض کرده ایم که آرایه به صورت صعودی مرتب شده است) این رویه تا یافتن عنصر مورد نظر یا بررسی کل خانه های آرایه ادامه می یابد.

مثال : در لیست زیر می خواهیم ببینیم عدد 44 در کدام خانه قرار دارد ؟

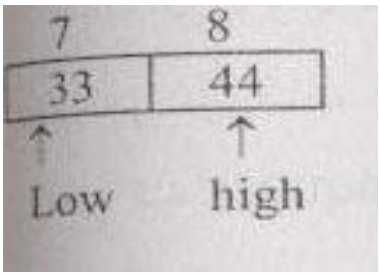


ابتدا خانه وسطی(6) را با عدد ۴۴ مقایسه می کنیم . چون ۳۰ کمتر از ۴۴ است پس نیمه بالایی آرایه یعنی از خانه ۷تا ۱۲ را فقط نگاه می کنیم.

برای این منظور low را برابر mid + 1 قرار می دهیم :



حال خانه وسط یعنی خانه ۹ را که حاوی عدد ۴۵ است را با ۴۴ مقایسه می کنیم . چون ۴۴ کمتر از ۴۵ است پس نیمه پایین این آرایه را فقط نگاه می کنیم . برای این کار high را برابر mid -1 قرار می دهیم.



حال خانه شماره ۷ سپس خانه شماره ۸ را با کلید ۴۴ مقایسه می کنیم و درمی یابیم که عدد ۴۴ در خانه شماره ۸ قرار دارد. برنامه مربوط به جستجوی باینری به صورت زیر است :

تابع زیر جستجوی باینری را (به صورت غیر بازگشتی) انجام می دهد و مشخص می سازد عدد m در کدام خانه آرایه x وجود دارد.

N طول آرایه مرتب شده x است.اگر عدد m در آرایه نباشد تابع مقدار 1- برمی گرداند.

```
int  bsearch (int  x [  ] , int  n , int m)
{
   int   low, high , mid;
   low = 0 ;    high  = n - 1 ;
   while  (low < = high ) {
           mid = (low + high) / 2 ;
           if  ( m < x [ mid ] )
                high  = mid - 1 ;
           else  if  (m > x [mid] )
                   low = mid + 1 ;
           else   return  mid
    }
    return   - 1 ;
}
```

حال نسخه بازگشتی الگوریتم جستجوی دودویی را بیان می کنیم :

```
int bsearch (int low, int high, int x, int A[ ])
{
  int  mid;
  if (low > high)    return  -1;
  else {
     mid = (low+  high) / 2;
     if (x = = A[mid]   return   mid;
     else  if (x < A[mid])
             return bsearch (low, mid -1, x, A);
    else return bsearch (mid+1, high, x, A);
  }
}
```

تحلیل پیچیدگی زمانی جستجوی دودویی

جستجوی دودویی پیچیدگی زمانی در هر حالت ندارد.لذا باید حالت های خوب , بد و متوسط آن را جداگانه حساب کنیم.

یکی از بدترین حالت ها هنگامی رخ می دهد که عدد مورد جستجو (x) آخرین (بزرگترین) ذعنصر آرایه باشد. در این حالت آرایه باید مرتباً نصف شود تا هنگامی که در نهایت آرایه یک خانه داشته باشد . آرایه ای با یک خانه تنها به یک عمل مقایسه نیاز دارد . لذا :

$$W(n) = W(\frac{n}{2}) + 1$$
$$W(1) = 1$$

پیچیدگی زمانی جستجوی دودویی :

با توجه به قضیه اصلی زیر که قبلاً بیان کردیم :

$$\begin{cases} T(n) = aT(\frac{n}{b}) + cn^k \\ T(1) \end{cases} \Rightarrow \begin{cases} T(n) = \theta(n^{\log_b^a}) & \text{اگر } a > b^k \\ T(n) = \theta(n^k \log_2 n) & \text{اگر } a = b^k \\ T(n) = \theta(n^k) & \text{اگر } a < b^k \end{cases}$$

$$a = 1 , b = 2 , c = 1 , k = 0 \Rightarrow a = b^k \Rightarrow 1 = 2^0 \Rightarrow$$
$$W(n) = \theta(n^0 \log_2 n) = \theta(\log n)$$

جواب رابطه بازگشتی w(n) برای جستجوی دودویی به صورت دقیق برابر است با

$$W(n) = \lfloor \log n \rfloor + 1 \Rightarrow W(n) = \theta(\log n)$$

در تست ها علت فرمول$\lfloor \log n \rfloor + 1$ را می بینید.

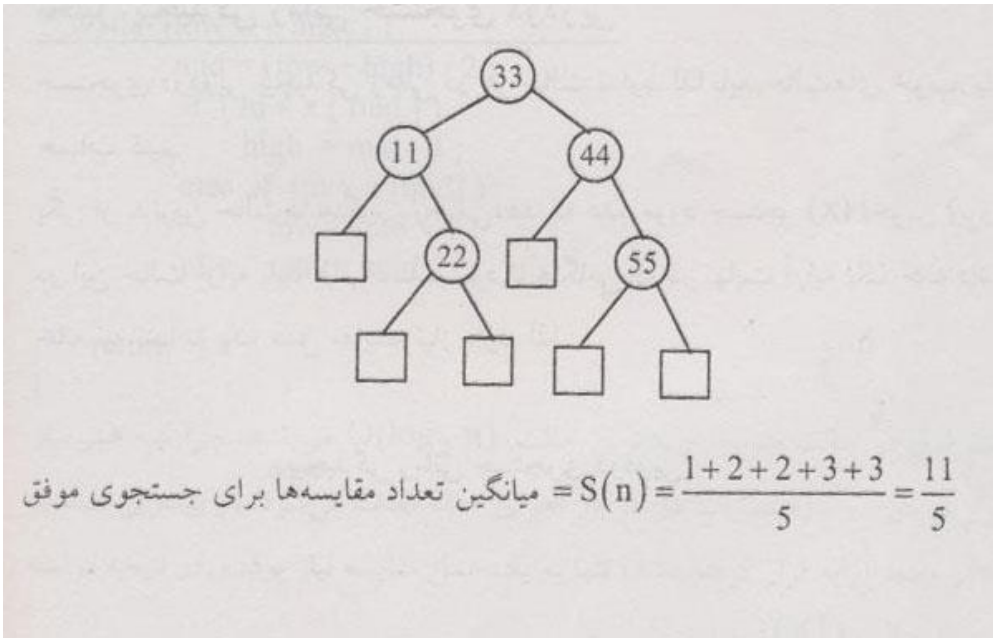
مثال : فرض کنید آرایه ای ۵ خانه ای و مرتب داریم و می خواهیم میانگین تعداد مقایسه ها را برای جست و جوی موفق و نیز میاتگین تعداد مقایسه ها را برای جستجوی ناموفق به دست آوریم :

1	2	3	4	5
11	22	33	44	55

برای یافتن عدد ۱۱ به ۲ مقایسه , عدد ۲۲ به ۳ مقایسه , عدد ۳۳ به یک مقایسه و عدد ۴۴ به ۲ مقایسه و عدد ۵۵ به ۳ مقایسه نیاز است . پس :

$$\text{میانگین تعداد مقایسه‌ها برای جستجوی موفق} = S(n) = \frac{2+3+1+2+3}{5} = \frac{11}{5}$$

این روش شبیه یافتن اعداد در درخت جست و جوی زیر است که متوسط سطح هر گره داخلی را به دست می آوریم.



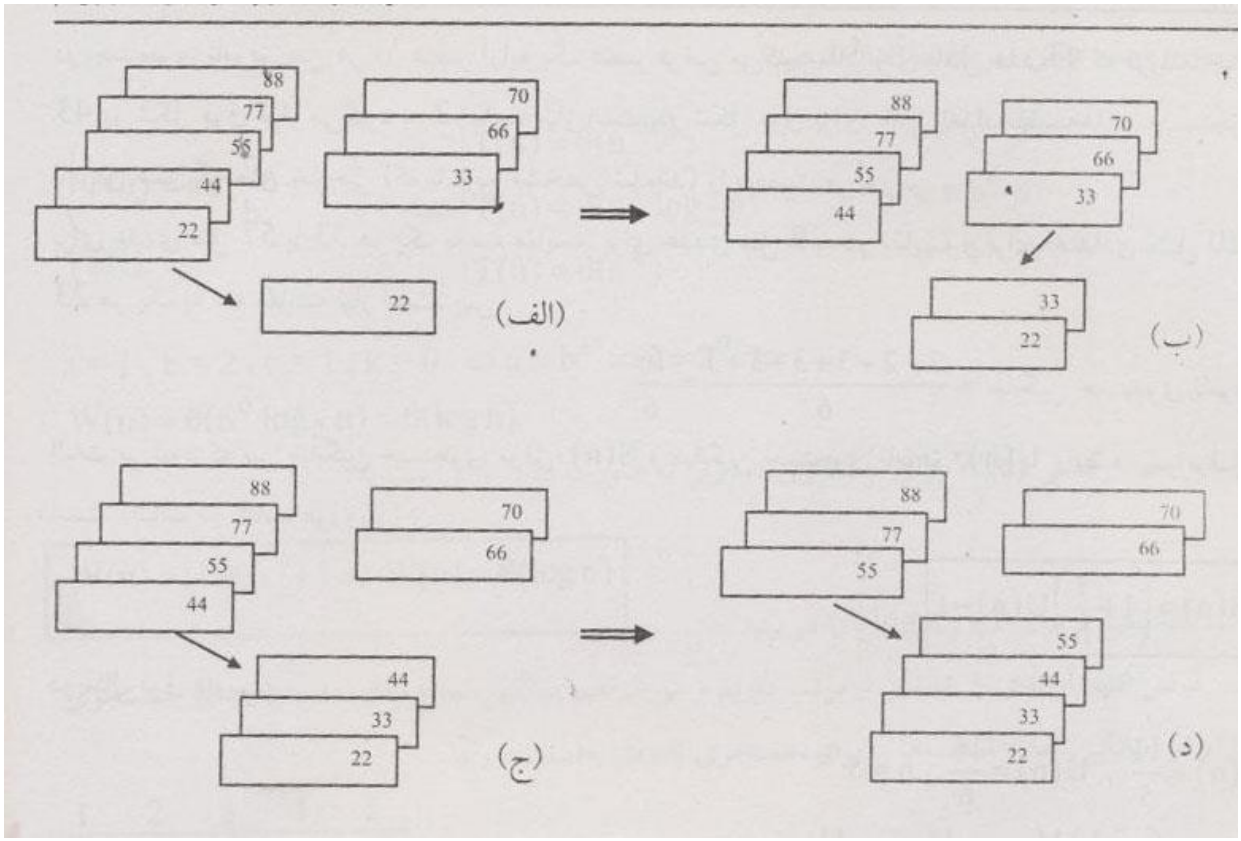
مثال دوم : مرتب سازی ادغامی

قبل از توضیح مرتب سازی ادغام لازم است نحوه ترکیب دو لیست مرتب را در یک لیست مرتب جدید بیان کنیم .

مثال :

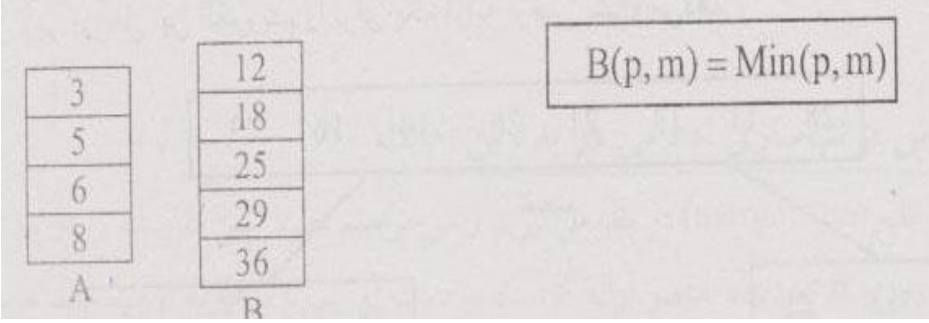
فرض کنید دو دسته کارت مرتب شده به شکل زیر داریم . در هر مرحه کارت جلویی هر دسته را با کارت جلویی دسته دیگر مقایسه می کنیم و کارت با شماره کوچک تر را در دسته جدید قرار می دهیم .

هر گاه یکی از دسته ها خالی شد تمام کارت های باقی مانده در دسته دیگر را در انتهای دسته کارت جدید قرار می دهیم.

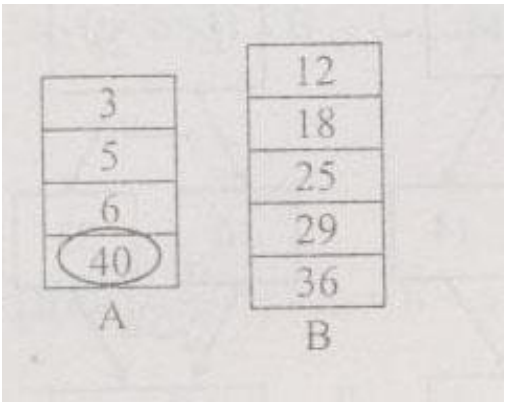


تابع زیر دو آرایه مرتب A با P عنصر و B با m عنصر را گرفته و یک آرایه S با m+p عنصر مرتب شده را برمی گرداند.(شروع اندیس آرایه ها را ۱ فرض کرده ایم) :

```
void  merge (int p, int m, int A[ ], int B[ ], int S[ ])
{
   int  i,j,k;
   i = j = k = 1;
   while (i <= p  && j <= m) {
       if (A[i] < B[j]) {    S[k] = A[i], i++; }
      else {S[k] = B[j];  j++}
      k ++;
}
if (i > p)
          while (j <= m)
          {   S[k] = B[j]; k++; j++;}
else if (j>m)
       while (i <= p)
         {   S[k] = A[i] ; k ++; i++;}
}
```



بدترین حالت نیز زمانی رخ می دهد که همه عناصراز آرایه اول (به جز آخرین عضو آن) از اولین عنصر آرایه دوم کوچک تر باشند . ولی این آخرین عضو از تمام عناصر آرایه دوم بزرگ تر باشد. مثل شکل زیر



که در این حالت به ۸ مقایسه نیاز داریم . یعنی در بدترین حالت تعداد مقایسه ها را برابر است با :

$$W(p,m) = p + m - 1$$

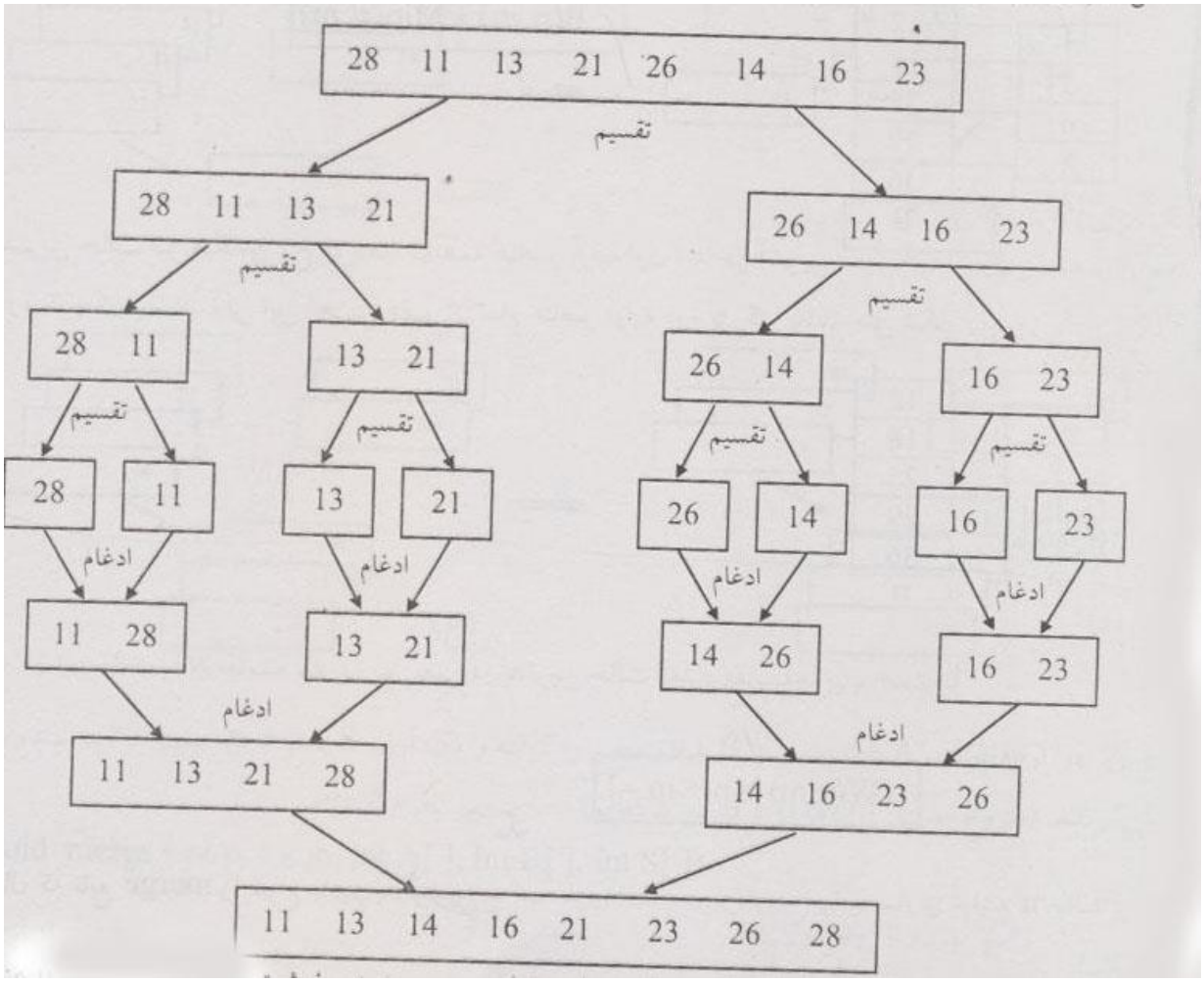
حال که تابع mergo را شرح دادیم به توضیح مرتب سازی ادغام می پردازیم . مرتب سازی آرایه n خانه ای با روش ادغام شامل مراحل زیر است :

۱- آرایه را به دو قسمت هر یک با $\frac{n}{2}$ عنصر تقسیم کن.

۲- با روش ادغام هر زیر آرایه را مرتب کن .اگر آرایه به قدر کافی کوچک نمی باشد به صورت بازگشتی آن را کوچک تر و حل کن.

۳- از طریق الگوریتم ادغام زیر آرایه های مرتب شده را در یک آرایه مرتب , ترکیب کن.

شکل زیر مراحل این الگوریتم را برای یک آرایه فرضی نشان می دهد :



تحلیل پیچیدگی زمانی مرتب سازی ادغامی در بدترین حالت

عمل اصلی را مقایسه در تابع ادغام (merge) در نظر می گیریم و می خواهیم تعداد این مقایسه را در بدترین حالت بر حسب اندازه ورودی n که تعداد عناصر آرایه s است به دست می آوریم. با توجه به توضیحات مربوط به این الگوریتم داریم :

$$W(n) = W(p) + W(m) + (p+m-1)$$

W(p) زمان لازم برای مرتب سازی آرایه A و W(m) زمان لازم برای مرتب سازی B و (P+M-1) زمان لازم برای ادغام A و B در بدترین حالت هستند.

در ابتدا فرض می کنیم n توانی از ۲ باشد , در این صورت :

$$P = \lfloor n/2 \rfloor = \frac{n}{2} \quad , \quad m = n - p = n - \frac{n}{2} = \frac{n}{2}$$

$$W(n) = W(\frac{n}{2}) + W(\frac{n}{2}) + (\frac{n}{2} + \frac{n}{2} - 1)$$

$$\Rightarrow W(n) = 2W(\frac{n}{2}) + n - 1$$

اگر اندازه ورودی ۱ باشد (یعنی آرایه یک خانه ای باشد)نیازی به ادغام نبوده و W(l)=0 است.پس رابطه پیچیدگی زمانی برای مرتب سازی ادغام هنگامی که n>1 بوده و n توانی از ۲ باشد به صورت زیر است :

$$W(n) = 2W(\frac{n}{2}) + n - 1$$
$$W(1) = 0$$

با توجه به قضیه اصلی زیر :

$$T(n) = aT(\frac{n}{b}) + Cn^k = \begin{cases} T(n) = \theta(n^{\log_b^a}) & \text{اگر } a > b^k \\ T(n) = \theta(n^k \log_2 n) & \text{اگر } a = b^k \\ T(n) = \theta(n^k) & \text{اگر } a < b^k \end{cases}$$

$$a = 2,\ b=2,\ k=1 \Rightarrow a = b^k \Rightarrow 2 = 2^1$$

پس در حالت داریم :

$$W(n) = \theta(n \log n)$$

```
void mergesort2 (int  low, int high, int S[ ])
{
  int  mid;
  if (low < high) {
       mid = ⌊ (low+  high) / 2 ⌋;
       mergesort2 (low, mid, S);
       mergesort2(mid +1, high, S);
       merge2 (low, mid, high, S);
  }
}
/ * * * * * * * * * * * * * * * * * * * * * * * * */
void merge2 (int low, int mid, int high, int S[ ])
{
 int i,j,k;
 int T[low .. high] ;     //A local arrray needed for the merging
 i = low;   j = mid +1;   k = low;
 while (i <= mid   && j <= high)  {
       if (S[i] < S[j]) {  T[k] = S[i];   i++;  }
       else  {   T[k] = S[j];   j++ }
       k ++;
 }
 if (i > mid) move  S[j] through S[high]  to T[k] through T[high];
 else move S[i] through S[mid] to T[k] through  T[high];
 move T[low] through T[high] to S[low] through S[high];
}
```

```
procedure Partition (Var x: Arraylist; left, right : integer; Var pivotpoint : integer);
Var i,j,pivot : integer;
begin
        i := left ;    j := right+1;    pivot := x[left];
        repeat
                repeat
                        i := i + 1;
                until  x[i] >= pivot;
                repeat
                        j := j - 1;
                until  x[j] <= pivot;
                if i<j  then  swap (x[i], x[j]);
        until  i>= j;
       swap (x[left], x[j]);
       pivotpoint := j;
end;
{ * * * * * * * * * * * * * * * * * * * }
procedure Quicksort (Var  x: Arraylist; left, right : integer) ;
var   pivotpoint : integer;
begin
       if (left  < right)   then  begin
              partition (x , left, right, pivotpoint); {عنصر لولا در محل واقعی خود قرار می گیرد}
              QuickSort(x, left, pivotpoint - 1); {مرتب‌سازی زیر لیست چپ}
              QuickSort (x, pivotpoint+1, right); { مرتب‌سازی زیر لیست راست}
       end;   {if}
end;
```

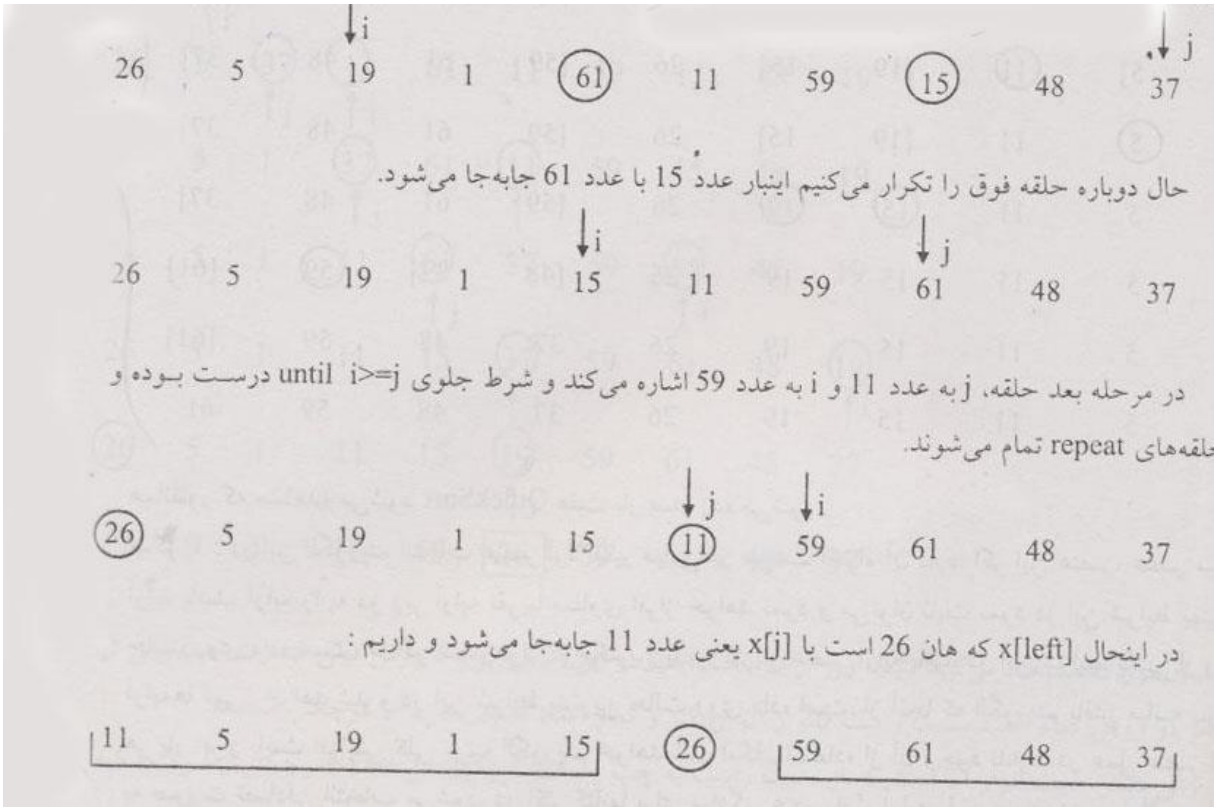
مثال ۳ :

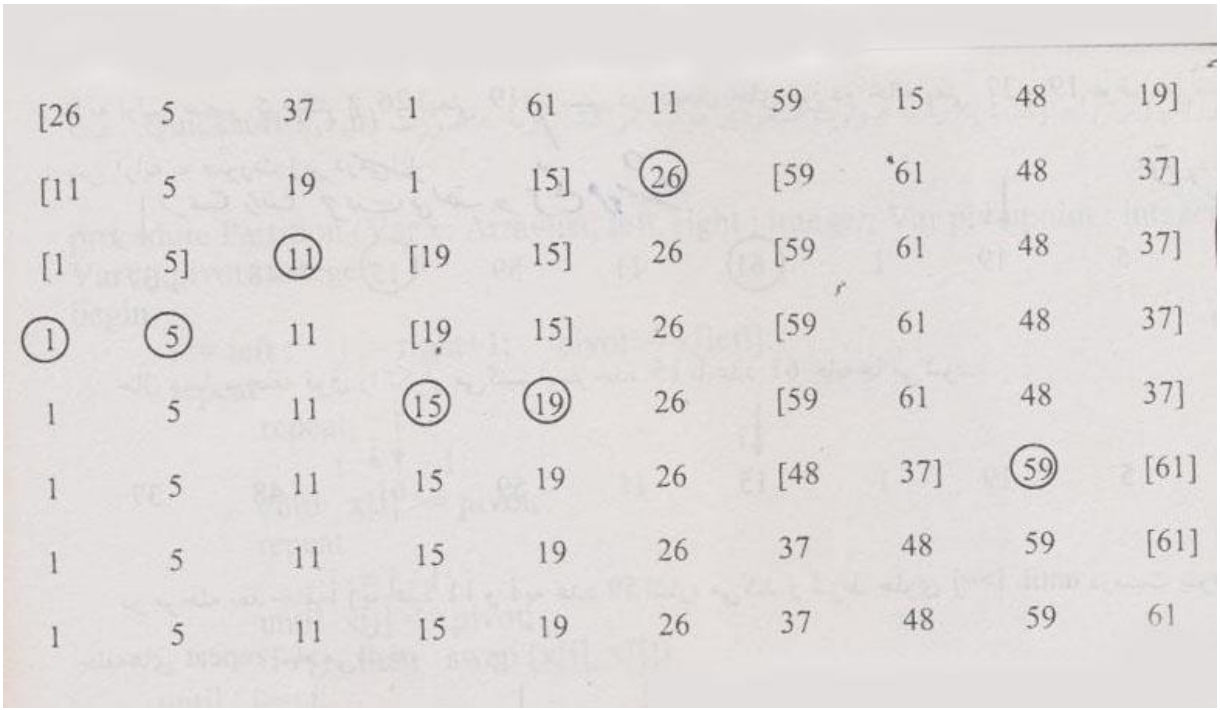
فرض کنید ارایه اولیه به صورت زیر باشد انجام یک مرحله از الگوریتم فوق به صورت زیر است :

x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]	x[10]
26	5	37	1	61	11	59	15	48	19

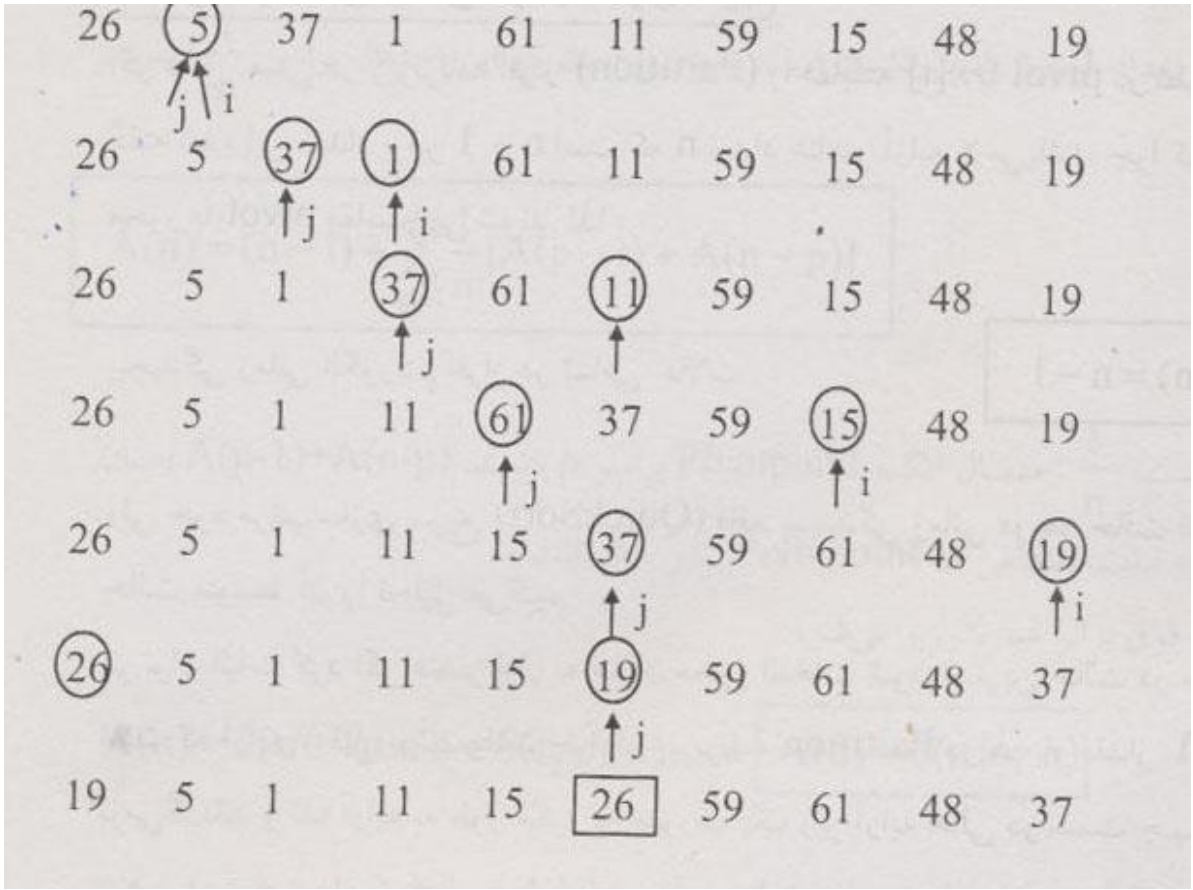
در ابتدا left = 1 و right = 10 و l =1 و j=11 و pivot = 26 می شود . سپس از سمت چپ مرتباً i زیاد شده تا هنگامی که به اولین عنصر بزرگ تر از ۲۶ (یعنی ۳۷) برسیم.به همین ترتیب از سمت راست مرتباً j کم می شود.

تا به اولین عنصر کوچک تر از ۲۶ (یعنی ۱۹) برسیم.در این حال جای این دو خانه یعنی ۳۷ و ۱۹ عوض می شود . پس آرایه به صورت زیر درمی آید :





نمونه زیر مراحل کار افراز را مطابق الگوریتم فوق نشان می دهد . عنصر لولا عدد ۲۶ است.



26	5	37	1	61	11	59	15	48	19
19	5	37	1	61	11	59	15	48	26
19	5	26	1	61	11	59	15	48	37
19	5	15	1	61	11	59	26	48	37
19	5	15	1	26	11	59	61	48	37
19	5	15	1	11	26	59	61	48	37

بهترین حالت هنگامی است که عنصر محور (افراز) همواره وسط آرایه قرار می گیرد . در این حالت اگر برای سادگی فرض کنیم $n = 2^m$ آن گاه پیچیدگی در بهترین زمان برابر است با :

$$B(n) = 2B\left(\frac{n}{2}\right) + \theta(n) \Rightarrow B(n) = \theta(n \log n)$$

نکته : پیچیدگی روش های ادغام و سریع به ورت زیر است :

	بهترین حالت	حالت متوسط	بدترین حالت
مرتب‌سازی سریع	O (n logn)	O (n logn)	$O(n^2)$
مرتب‌سازی ادغام	O (n logn)	O (n logn)	O (n logn)

مثال چهارم : ضرب ماتریس های استراسن (Strassen)

همان طور که می دانید الگوریتم ساده ضرب ماتریس ها به صورت زیر است :

```
for (i=1;  i<= n; i++)
    for (j=1;   j <= n; j++) {
        C[i][j] = 0;
        for (k=1; k <=n;  k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

تکه برنامه فوق دو ماتریس مربعی A و B را ضرب کرده و حاصل را در ماتریس C می ریزیم . هر سه ماتریس n*n هستند.

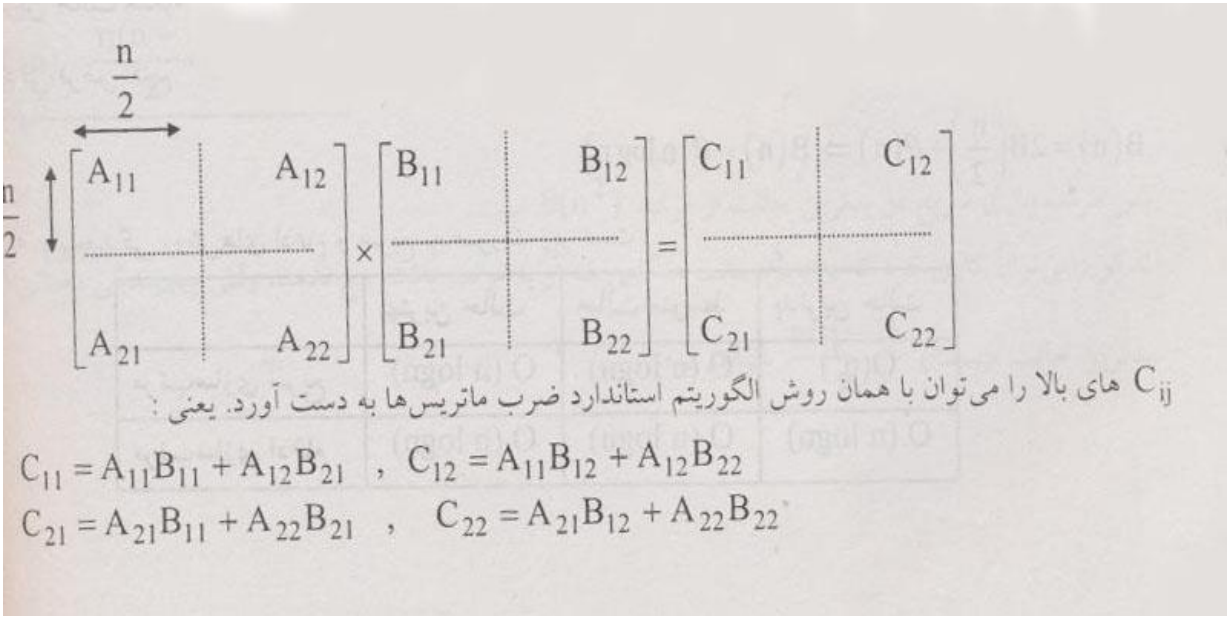
در الگوریتم ساده فوق تعداد ضرب ها همواره $T(n) = n^3$ است. تعداد جمع ها نیز در برنامه ساده فوق $T(n) = n^3$ می باشد.با تغییر ساده زیر می توان تعداد جمع ها را کاهش داد و به $T(n) = n^3-n^2$ رساند :

```
for (i=1;  i<= n; i++)
    for (j=1;   j <= n; j++) {
        C[i ,j] = A[i][1] * B[1][j]
        for (k=2; k <=n;  k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

پس خلاصه در الگوریتم استاندارد ضرب ماتریس ها را داریم :

تعداد ضرب‌ها $= n^3 = \theta(n^3)$
تعداد جمع‌ها $= n^3 - n^2 = \theta(n^3)$

حال روش ضرب ماتریس ها را به صورت تقسیم و غلبه بیان می کنیم.اگر n توانی از ۲ باشد می توان ماتریس های A و B را به چهار ماتریس کوچکتر که هر یک $\frac{n}{2} \times \frac{n}{2}$ هستند تقسیم نمود.



C_{ij} های بالا را می‌توان با همان روش الگوریتم استاندارد ضرب ماتریس‌ها به دست آورد. یعنی :

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \quad , \quad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \quad , \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

اگر در الگوریتم بالا عمل اصلی را تعداد ضرب ها در نظر بگیریم , آنگاه تعداد ضرب ها برای ماتریس های n*n به ۸ عمل ضرب ماتریس های $\frac{n}{2} \times \frac{n}{2}$ نیاز خواهند داشت.

از طرف دیگر بدیهی است که ضرب دو ماتریس 1*1 فقط به یک عمل ضرب اسکالر نیاز دارد . لذا برای الگوریتم ساده تقسیم و غلبه ضرب ماتریس ها داریم :

$$\left.\begin{array}{l} T(n) = 8T(\frac{n}{2}) \\ T(1) = 1 \end{array}\right\} \Rightarrow \theta(n^3)$$

یادآوری : نتیجه فوق از قضیه زیر به دست آمد که در فصل قبلی بیان کرده بودیم :

$$\begin{cases} T(n) = aT(\frac{n}{b}) \\ T(1) = 1 \end{cases} \Rightarrow \begin{cases} \theta(n^{\log_b^a}) & (a \neq 1) \\ \theta(\log_b^n) & (a = 1) \end{cases}$$

همان طور که مشاهده می شود روش تقسیم و غلبه مانند روش مستقیم از مرتبه $\theta(n^3)$ و مزیتی ندارد. ولی در سال ۱۹۶۹ استراسن (strassen) الگوریتمی را معرفی کرد که تعداد ضرب های آن از n^3 کمتر بوده و تقریباً $n^{2.81}$ بود که آن را در ادامه شرح می دهیم.

استراسن اثبات کرد که حاصلضرب دو ماتریس A*B یعنی ماتریس C را می توان از روابط زیر به دست آورد (برای حالت n=2 به سادگی می توانید آن را اثبات کنید.)

$$C = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

مثلاً برای ماتریس زیر :

$$A=\begin{bmatrix}1&2&3&4\\5&6&7&8\\9&1&2&3\\4&5&6&7\end{bmatrix},\quad B=\begin{bmatrix}8&9&1&2\\3&4&5&6\\7&8&9&1\\2&3&4&5\end{bmatrix}$$

$$M_1=(A_{11}+A_{22})(B_{11}+B_{22})=$$
$$\left(\begin{bmatrix}1&2\\5&6\end{bmatrix}+\begin{bmatrix}2&3\\6&7\end{bmatrix}\right)\times\left(\begin{bmatrix}8&9\\3&4\end{bmatrix}+\begin{bmatrix}9&1\\4&5\end{bmatrix}\right)=\begin{bmatrix}3&5\\11&13\end{bmatrix}\times\begin{bmatrix}17&10\\7&9\end{bmatrix}=\begin{bmatrix}86&75\\278&227\end{bmatrix}$$

بدیهی است که برای حالت n=1 یعنی ضرب ماتریس های L*L تنها به یک عمل ضرب و صفر عمل جمع و تفریق نیاز داریم.

با توجه به توضیحات فوق :

$$\text{تعداد ضرب‌ها در استراسن} = \begin{cases} T(n)=7T(\frac{n}{2}) \Rightarrow T(n)=n^{\log_2^7}=n^{2.81} \\ T(1)=1 \end{cases}$$

و به همین ترتیب تعداد جمع ها برابر است با :

$$\text{تعداد جمع‌ها در استراسن} = \begin{cases} T(n)=7T(\frac{n}{2})+18(\frac{n}{2})^2 \\ T(1)=0 \end{cases}$$

جدول زیر روش استراسن را با روش استاندارد مقایسه می کند :

عمل	روش استاندارد	روش استراسن
ضرب	n^3	$n^{2.81}$
جمع / تفریق	$n^3 - n^2$	$6n^{2.81} - 6n^2$

بنابراین روش استراسن کارآمدترین از روش استاندارد است.

نکته : می توان اثبات کرد ضرب ماتریس ها حداقل به زمان $\theta(n^2)$ نیاز دارد و لی هنوز کسی نتوانسته است الگوریتمی از مرتبه ۲ برای ضرب ماتریس ها ابداع کند و از طرف دیگر تاکنون نیز کسی نتوانسته اثبات کند نوشتن چنین الگوریتمی غیر ممکن است.

مثال پنجم : ضرب اعداد صحیح بزرگ

همانطور که می دانید اعداد صحیح بزرگ را می توان توسط آرایه ذخیره کرد . مثلاً عدد بزرگ 8,173,963,563 را به صورت زیر ذخیره کنیم:

a[10]	a[9]	a[8]	a[7]	a[6]	a[5]	a[4]	a[3]	a[2]	a[1]
8	1	7	3	9	6	3	5	6	3

برای تعیین مثبت یا منفی بودن عدد هم کافی است آخرین خانه سمت چپ را به عنوان علامت در نظر بگیریم که اگر مثلاً صفر باشد یعنی مثبت و اگر ۱ باشد یعنی منفی.

در حالت کلی عدد صحیح u با n رقم را می توان به صورت زیر نشان داد :

$$u = x \times 10^m + y$$

که x حاوی $\left\lfloor \frac{n}{2} \right\rfloor$ رقم، y حاوی $\left\lfloor \frac{n}{2} \right\rfloor$ رقم و $m = \left\lfloor \frac{n}{2} \right\rfloor$ است. مثلاً :

$$967345 = 967 \times 10^3 + 345$$
$$8967345 = 8967 \times 10^3 + 345$$

ضرب این دو عدد به صورت زیر خواهد بود :

$$uv = (x \times 10^m + y)(w \times 10^m + z) = xw \times 10^{2m} + (xz + wy) \times 10^m + yz$$

پس ضرب u در V به چهار عمل ضرب روی اعداد صحیح با نیمی از ارقام نیاز دارد. توجه کنید که ضرب یک عدد در 10^m یا 10^{2m} به سادگی در زمان O(n) صورت می پذیرد . مثلاً :

$$567,832 \times 9\,,\,423,723 = (567 \times 10^3 + 832)(9423 \times 10^3 + 723)$$
$$= 567 \times 9423 \times 10^6 + (567 \times 723 + 9423 \times 832) \times 10^3 + 832 \times 723$$

پیاده سازی الگوریتم ضرب فوق به صورت زیر است :

```
Large_int  prod (Large_int  u, Large_int  v)
  {
  Large_int  x,y,w,z;
  int  n,m ;
  n = Maximum (number of digits in u, number of digits in v)
  if (u == 0 || v == 0)   return  0;
  else  if(n <= threshold)
      return   u × v;
  else  {
      m = ⌊n / 2⌋ ;
      x = u  divide  10^m ;    y = u  rem  10^m ;
      w = v  divide  10^m ;  z = v  rem  10^m;
      return   prod (x,w) × 10^2m + (prod(x,z) + prod(w,y)) × 10^m + prod (y,z);
    }
```

جواب فوق از قضیه اصلی در فصل قبل به دست آمده است. مقدار S همان مقدار آستانه است که به ازای آن دیگر نمونه تقسیم نمی شود.

الگوریتم فوق از درجه n^2 بوده و در نتیجه کند است با یک تغییر می توان مرتبه اجرای الگوریتم را پایین تر آورد . همان طور که در بالا مشاهده کردید تابع prod باید موارد زیر را محاسبه کند:

Xw ,xz +yw ,yz

یعنی تابع prod بایستی چهار بار برای محاسبه , xz,yw,yz xw صدا زده شود.حال روابط مذکور را قدری دستکاری می کنیم.

$$xz + yz = (x + y)(w + z) - xw - yz$$

یعنی ابتدا xw و yz را به دست آورده و سپس به کمک عبارت سمت راست رابطه فوق مقدار xz+yz را با یک عمل ضرب به دست می آوریم . پس در کل به جای ۴ ضرب به ۳ ضرب نیاز خواهیم داشت . لذا :

$$\left.\begin{array}{l} W(n) = 4W(\frac{n}{2}) + Cn \\ W(s) = 0 \end{array}\right\} \Rightarrow W(n) = \theta(n^{\log_2^4}) = \theta(n^2)$$

حال اگر بخواهیم به شیوه مستقیم بزرگ ترین و کوچک ترین عنصر یک آرایه را پیدا کنیم یک الگوریتم ساده به صورت زیر است :

```
min = S[1];
max = S[1] ;
for (i = 2  ;  i <= n ; i ++)
{
      if (S[i] < min)
            min = S[i] ;
      else   if (S[i] > max)
            max = S[i];
}
```

الگوریتم زیر به صورت مستقیم با فرض آنکه n عددی زوج است کوچکترین و بزرگ ترین عنصر را با تکنیک جفت کردن کلیدها به دست می آورد :

```
if (S[1] < S[2])   {   min  = S[1]; max = S[2];}
else  { min=S[2];   max=S[1]; }
for (i=3;  i <= n-1 ;  i = i + 2)   {
     if  (S[i] > S[i+1])
            swap (S[i] , S[i+1]);
    if (S[i] < min)     min = S[i];
    if (S[i+1] > max )    max = S[i+1];
}
```

برای نمونه یک آرایه ۱۰ خانه ای در نظر گرفته و الگوریتم فوق را روی آن آزمایش کنید . در برنامه فوق حلقه for برای n های زوج به تعداد $(\frac{n}{2} - 1)$ بار اجرا می شود که در هر بار اجرا سه مقایسه صرت می

گیرد . یک مقایسه نیز در ابتدای کار و بیرون حلقه انجام می پذیرد لذا مرتبه اجرایی برنامه فوق در تمامی حالات برابر است با :

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

به عنوان تمرین برنامه فوق را برای nهای فرد بازنویسی کرده و نشان دهید که برای nهای فرد پیچیدگی زمانی برابر است با

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

لذا در کل داریم :

$$T(n)=\begin{cases}\frac{3n}{2}-2 & \text{اگر n زوج باشد}\\ \frac{3n}{2}-\frac{3}{2} & \text{اگر n فرد باشد}\end{cases}$$

در برنامه زیر آرایه S حاوی اندیس های high.... Low بوده و خروجی تابع توسط پارامترهای min و max برگردانده می شود :

```
find  (low, high, S, max, min)
{
    if (low == high)  { max=min=S[low];   return;}
        if (low == high - 1)
        {
            if (S[low] < S[high])
                  { max = S[high];   min = S[low]; }
            else  { max = S[low];   min = S[high]; }
          return ;
        }
```

```
mid = (low+ high) / 2 ;
find (low, mid, S, max1, min1);
find (mid+1, high , S, max2, min2);
max = Maximum (max1, max2);
min = Minimum (min1, min2);
}
```

پیچیدگی زمانی الگوریتم فوق از رابطه بازگشتی زیر به دست می آید :

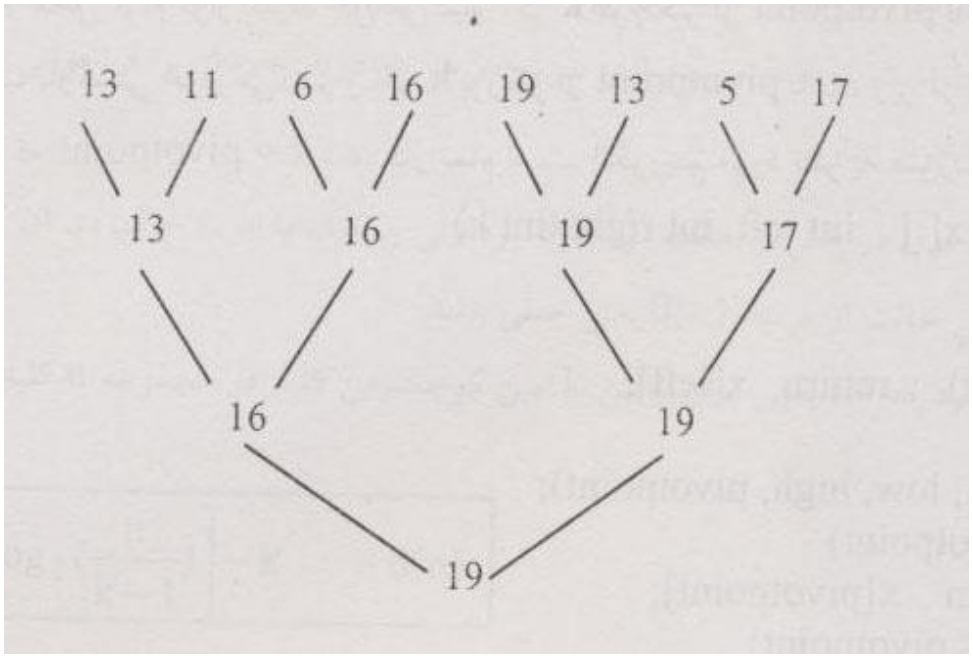
$$T(n)=\begin{cases}0 & \text{اگر } n=1\\ 1 & \text{اگر } n=2\\ T\left(\left\lceil \frac{n}{2}\right\rceil\right)+T\left(\left\lfloor \frac{n}{2}\right\rfloor\right)+2\end{cases}$$

با حل رابطه فوق به نتیجه زیر می رسیم :

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

یافتن بزرگ ترین کلید دوم :

روش تورنمنت همان روشی است که در مسابقات ورزشی حذفی استفاده می شود . فرض می کنیم ۸ عدد به صورت درخت زیر با یکدیگر مسابقه دهند و هر بار برنده عدد بزرگ تر باشد :




```
{
    int  pivotpoint;
    if (left == right)    return  x[left];
    else {
        partition (x, low, high, pivotpoint);
        if(k == pivotpoint)
                return   x[pivotpoint];
        else  if(k < pivotpoint)
                return  selection (x, left, pivotpoint-1, k);
        else
                return  selection(x, pivotpoint +1 , right, k);
    }
}
void  partition (int x[ ], int left, int right, int & pivotpoint)
{
   int  i,j,pivot;
   j=left;   pivot = x[left];
   for (i=left+1;  i <= right ;  i++)
          if (x[i] < pivot)  {
                j ++;
                swap (x[i] , x[j]);
        }
   swap (x[left], x[j]);
   pivotpoint = j;
}
```

کجا نباید از تقسیم و غلبه استفاده کرد

در صورت امکان در موارد زیر ازروش تقسیم و غلبه استفاده نمی کنیم چرا که مرتبه الگوریتم نمایی می شود :

۱- هنگامی که نمونه ای با اندازه n به دو یا چند نمونه تسیم می شود که اندازه آن ها نیز تقریباً برابر با n است .مثلاً :

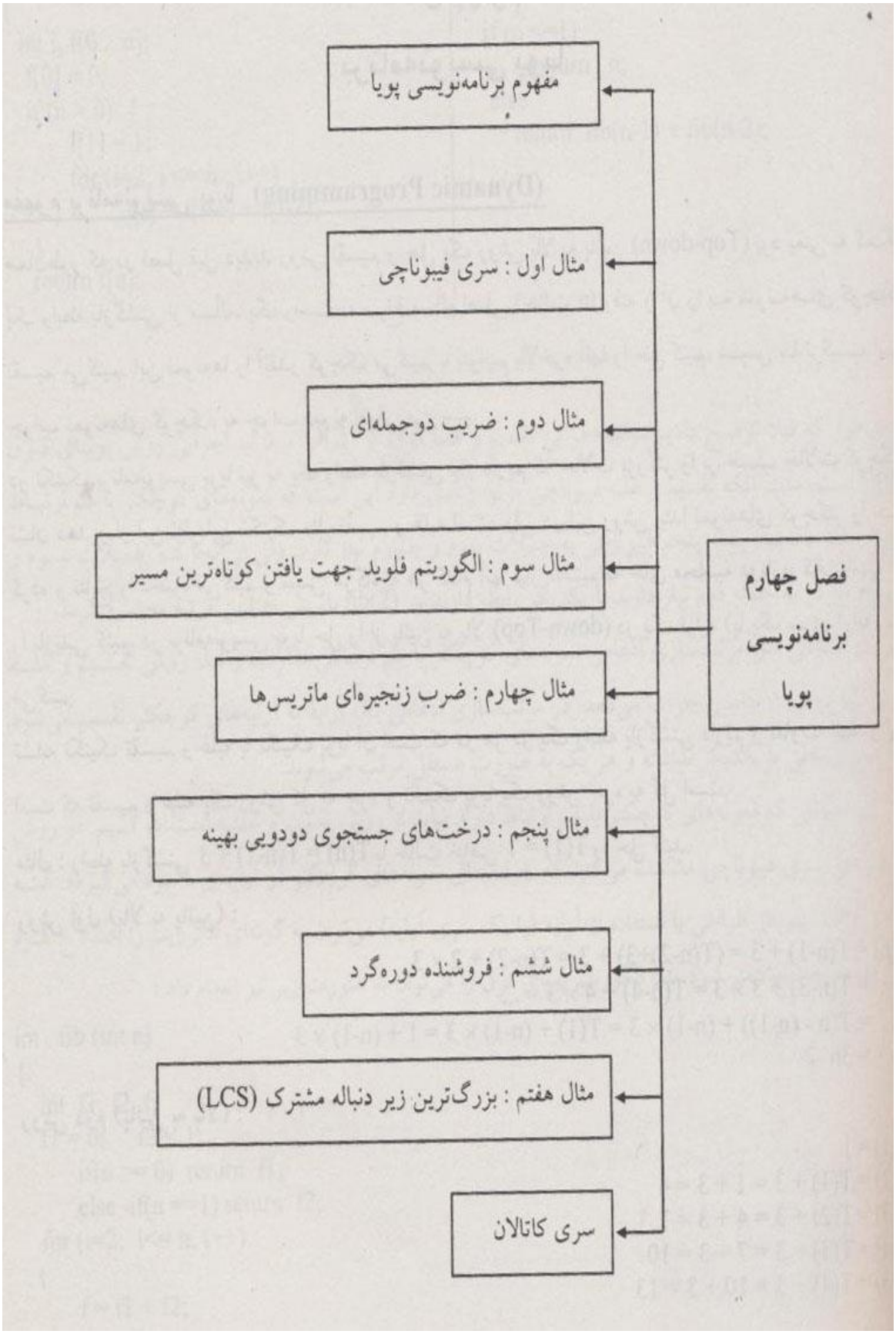
$$T(n) = 2T(n-1) \Rightarrow T(n) = \theta(2^n)$$

۲- هنگامی که نمونه ای با اندازه n , تقریباً به n نمونه با اندازه $\frac{n}{c}$ تقسیم شود که c یک مقدار ثابت اسن.

توجه کنید که همه مسائل را نمی توان به نمونه های کوچک تر تقسیم کرد و سپس نتایج حاصله را با هم ترکیب نمود. مثلاً اگر بخواهیم بزرگ ترین عدد بین ۲۰ عدد را پیدا کنیم می توان آن را به دو دسته ۱۰ تایی تقسیم کرد , سپس ماکزیمم هر دسته را یافته و با مقایسه این دو بزرگ ترین را پیدا کرد.ولی مثلاض در مورد یک دستگاه ۲۰ معادله ۲۰ مجهولی نمی توان آن را به دو دستگاه کوچک تر ۱۰ معادله و ۱۰ مجهولی تقسیم کرد.

فصل چهارم

برنامه نویسی پویا


فصل چهارم
برنامه‌نویسی
پویا
مفهوم برنامه‌نویسی پویا
مثال اول : سری فیبوناچی
مثال دوم : ضریب دوجمله‌ای
مثال سوم : الگوریتم فلوید جهت یافتن کوتاه‌ترین مسیر
مثال چهارم : ضرب زنجیره‌ای ماتریس‌ها
مثال پنجم : درخت‌های جستجوی دودویی بهینه
مثال ششم : فروشنده دوره‌گرد
مثال هفتم : بزرگ‌ترین زیر دنباله مشترک (LCS)
سری کاتالان

مفهوم برنامه نویسی پویا |(Dynamic Programming)

همان طور که در فصل قبل دیدید روش تقسیم و حل یک روش بالا به پایین (Top-down) بود یعنی به کمک یک رابطه بازگشتی از مساله یک راست به سراغ مساله اصلی (حالت n) رفته و آن را به نمونه های کوچک تقسیم می کنیم .

این نمونه ها را ان قدر کوچک می کنیم تا بتوانیم بالاخره آن ها را حل کنیم .سپس با ترکیب این جواب نمونه های کوچک به جواب نمونه اصلی می رسیم.

در تکنیک برنامه نویسی پویا نیز یک رابطه بازگشتی نیاز داریم که حالات بزرگ تر را بر حسب حالات کوچک تر نشان می دهد و از این این تکنیک مثل تقسیم و غلبه است.ولی در این روش ابتدا نمونه های کوچک تر را حل کرده و نتایج را ذخیره می کنیم و سپس هر گاه به هر کدام آنها نیاز داشتیم به جای محاسبه دوباره کافی است آن ها را بازیابی کنیم .

در برنامه نویسی پویا حل را از بالا به پایین (down-Top) در یک آرایه (یا یک سری آرایه) بنا می کنیم.

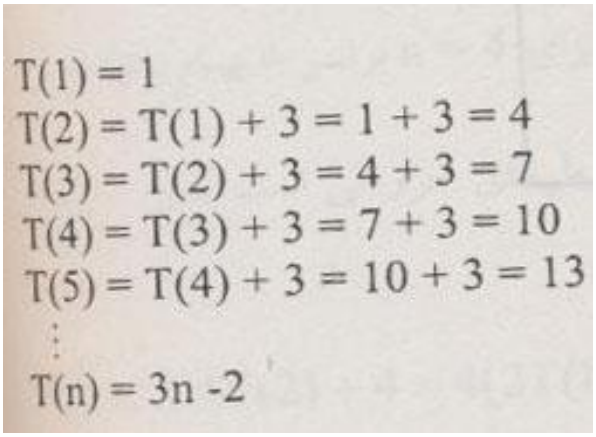
تشابه تکنیک تقسیم و غلبه با تکنیک پویا آن است که در هر دو رابطه بازگشتی داریم و تفاوت آن ها در این است که تقسیم و غلبه یک روش کل به جزء و تکنیک پویا یک روش جزء به کل است.

مثال : رابطه بازگشتی $T(n) = T(n-1)+3$ با حالت خاص $T(1) =1$ را حل کنید.

روش اول (بالا به پایین) :

$$
\begin{aligned}
T(n) &= T(n-1) + 3 = (T(n-2)+3) + 3 = T(n-2) + 2 \times 3 \\
&= T(n-3) + 3 \times 3 = T(n-4) + 4 \times 3 = \dots \\
&= T(n - (n-1)) + (n-1) \times 3 = T(1) + (n-1) \times 3 = 1 + (n-1) \times 3 \\
&= 3n - 2
\end{aligned}
$$

روش دوم (پایین به بالا) :



مثال اول : سری فیبونانچی

مساله سری فیبونانچی را در انتهای فصل اول به دو صورت تقسیم و غلبه و پویا به صورت زیر حل کردیم :

روش پویا

```
int  fib(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];
}
```

روش تقسیم و غلبه

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

```
int  fib (int n)
{
   int  f1, f2, f;
   f1 = 0;     f2 = 1;
        if(n == 0)  return  f1;
        else  if(n ==1) return  f2;
   for (i=2;  i<= n; i++)
   {
        f = f1 + f2;
        f1 = f2;
        f2 = f;
   }
   return f;
}
```

مثال دوم : ضریب دو جمله ای

همان طور که می دانید بسط دو جمله ای نیوتن $(a+b)^n$ به صورت زیر می باشد :

و می دانیم که

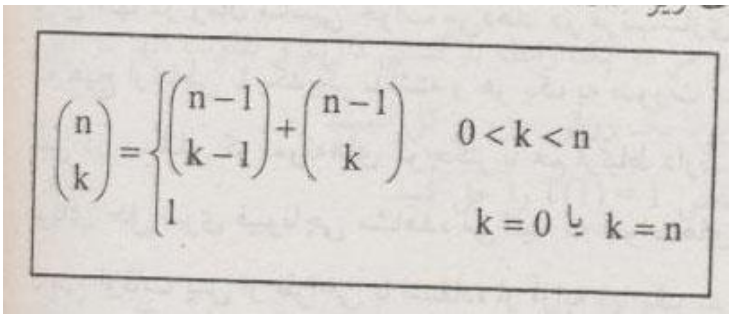
$$(a+b)^n = \binom{n}{0}a^n + \binom{n}{1}a^{n-1}b + \binom{n}{2}a^{n-2}b^2 + \ldots + \binom{n}{k}a^{n-k}b^k + \ldots + \binom{n}{n}b^n$$

$$C(n,k) = \binom{n}{k} = \frac{n!}{(n-k)!k!}$$

$$\binom{n}{0} = \binom{n}{n} = 1 \quad , \quad \binom{n}{k} = \binom{n}{n-k} \quad , \quad \binom{n}{1} = n$$

برای محاسبه ضریب دو جمله ای نیوتن یعنی $\binom{n}{k}$ می توان از فرمول مستقیم $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ استفاده کرد ولی محاسبه $n!$ برای مقادیر نه چندان بزرگ n هم خیلی بزرگ خواهد بود ، لذا باید راه حل بهتری پیدا کنیم.

یک فرمول بازگشتی برای محاسبه ترکیب k از n به صورت زیر است :



$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ یا } k = n \end{cases}$$

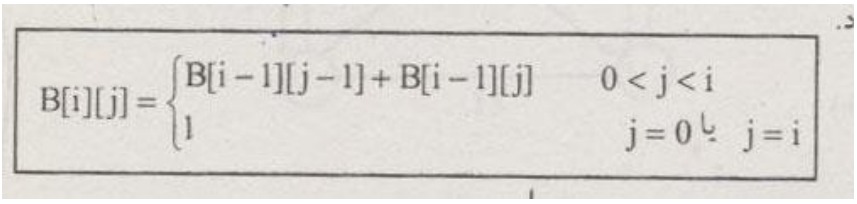
مثلاً $\binom{7}{3} = \binom{6}{2} + \binom{6}{3}$

پیاده سازی فرمول فوق به صورت الگوریتم تقسیم و غلبه به صورت زیر است :

```
int bin(int n, int k)
{
  if (k ==0 || n == k)  return 1 ;
  else return bin (n-1 , k-1) + bin (n-1, k);
}
```

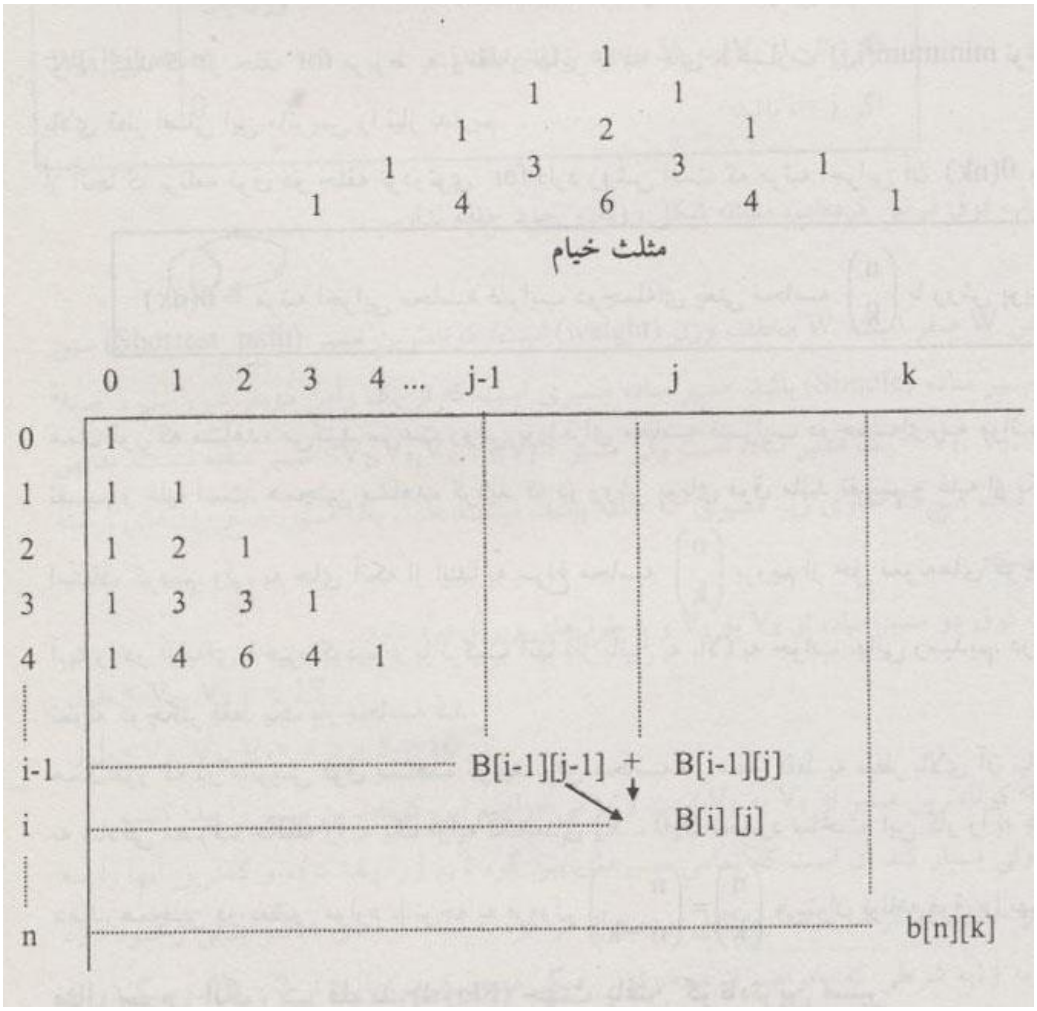
در تست ها خواهید دید که تعداد جملاتی که الگوریتم فوق برای به دست آوردن $\binom{n}{k}$ لازم است محاسبه کند برابر است با :

$$2\binom{n}{k} - 1$$



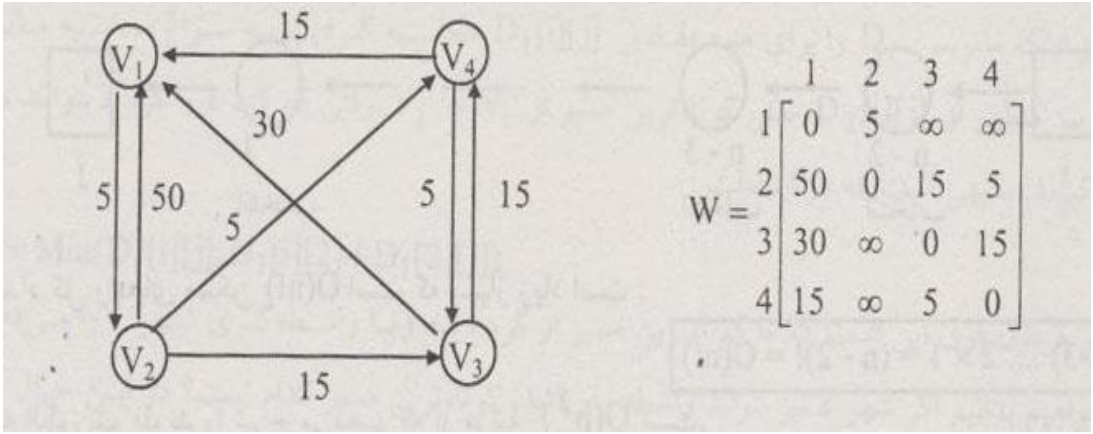
$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0 \text{ یا } j = i \end{cases}$$

توجه کنید که فرمول فوق از همان فرمول بازگشتی $\binom{n}{k}$ به دست آمده است .



منظور از $B[i][j]$ عبارت $\binom{i}{j}$ ، $B[n][k] = \binom{n}{k}$ است. برنامه ای که ماتریس فوق را اجرا می کند به صورت زیر است :

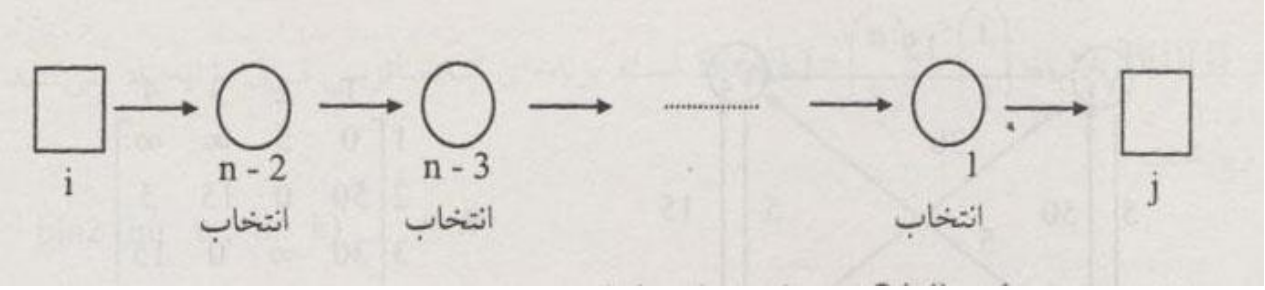
```
int   bin2 (int   n ,  int  k)
{
   int  i,j ;    int  B[0 .. n] [0 .. k];
   for (i=0;  i <= n ; i ++)
     for (j=0; j <=minimum(i,k); j++)
     {
          if (j == 0 || j == i)
                B[i][j] = 1 ;
          else  B[i][j] = B[i-1][j-1] + B[i-1][j];
     }
   return B[n][k];
}
```



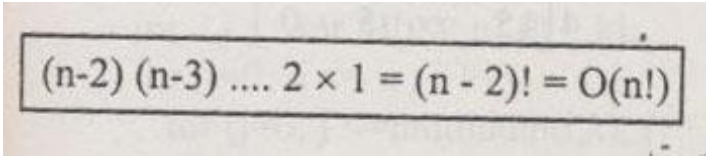
$$W = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{bmatrix} \end{array}$$

گراف وزن دار فوق را توسط ماتریس w که به ماتریس هم جواری (adjacency) معروف است ، طبق فرمول زیر نمایش داده ایم :

$$W[i][j] = \begin{cases} \text{وزن یال} & \text{اگر یالی از } V_i \text{ به } V_j \text{ وجود داشته باشد :} \\ \infty & \text{اگر یالی از } V_i \text{ به } V_j \text{ وجود نداشته باشد :} \\ 0 & \text{اگر } i = j \text{ باشد :} \end{cases}$$



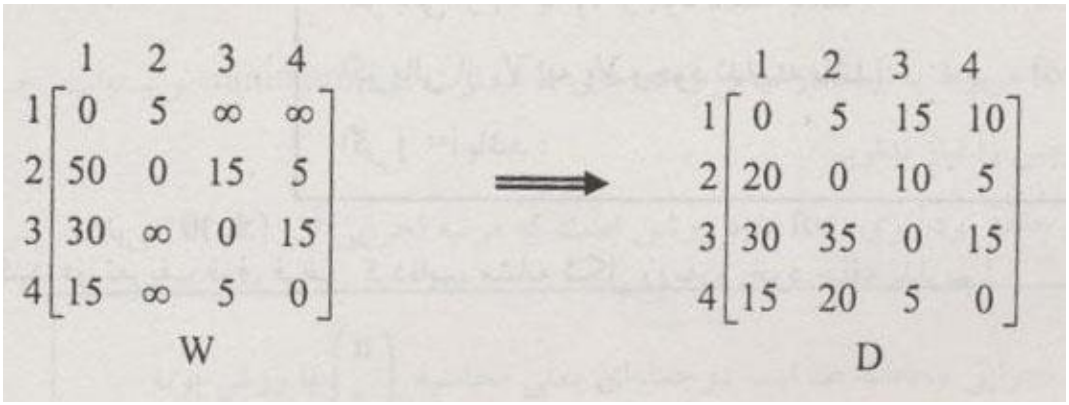
پس تعداد کل راه حل های ممکن $O(n\ !)$ است که بسیار زیاد است.



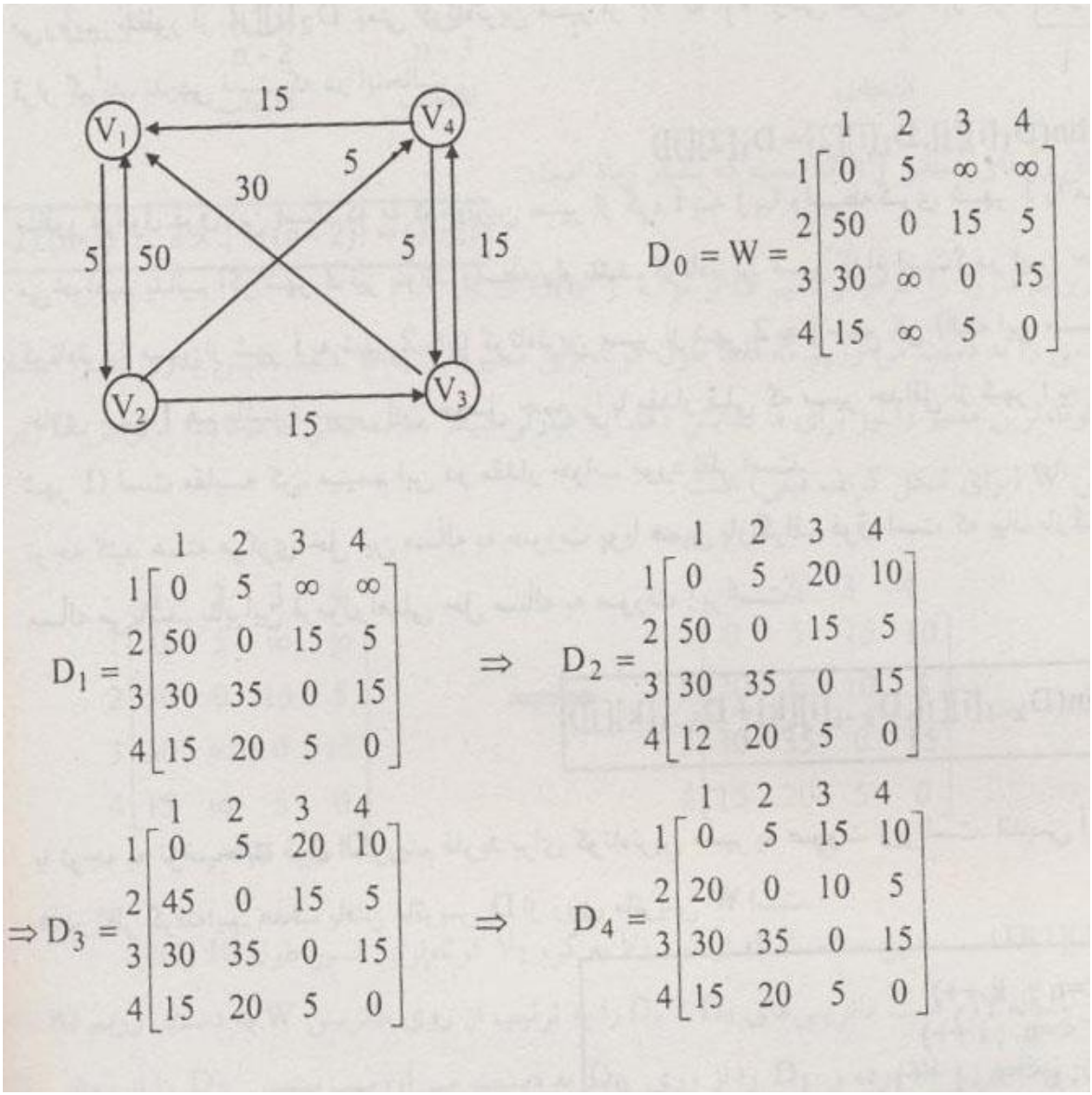
در ادامه الگوریتم فلوراید را شرح می دهیم که از مرتبه $O(n^3)$ است.

ابتدا الگوریتمی را به دست می آوریم که فقط طول کوتاه ترین مسیرها را به ما می دهد . سپس قدری آن را اصلاح می کنیم تا کوتاه ترین مسیر را نیز برای ما نمایش دهد.

به عبارتی دیگر هدف فعلی ما محاسبه ماتریس D زیر از روی ماتریس W (برای شکل گراف قبلی) است :



برای اینکه مراحل الگوریتم را به خوبی درک کنید با روش دستی مرحله به مرحله ماتریس های زیر را به دست آورید :



مثلاً برای محاسبه $D_1[3][2]$ ، این گونه عمل شده است :

$$D_1[3][2] = \text{Min}(D_0[3][2], D_0[3][1] + D_0[1][2])$$
$$D_1[3][2] = \text{Min}(\infty, 30 + 5) = 35$$

ماتریس P یک ماتریس $n \times n$ با اندیس های ۱ تا n است که مقدار اولیه عناصر ان صفر است :

```
for (k = 1;   k <= n; k++)
   for (i = 1 ; i <= n; i++)
        for (j=1; j <=n; j++)
          if (D[i][k] + D[k][j] < D[i][j]) {
               P[i][j] = k;
               D[i][j] = D[i][k] + D[k][j];
          }
```

مثلاً در گراف مثال قبل ماتریس p برابر زیر خواهد شد :

$$P = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

روش پویا برای حل دسته مسائل بهینه سازی

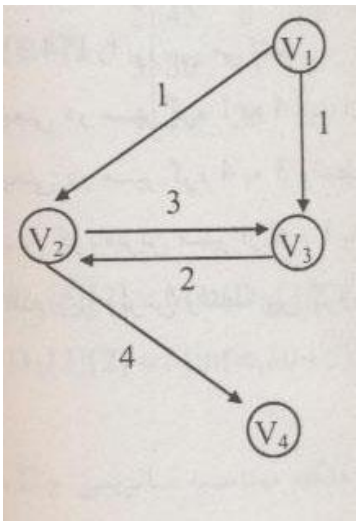
تعریف : هنگامی که می گوییم اصل بهینگی برای یک مسئله صادق است که یک حل بهینه برای نمونه ای از مسئله همواره حاوی حل بهینه برای همه زیر نمونه های زیر باشد.

مثلاً برای مسئله کوتاه ترین مسیر اصل فوق برابر است چرا که اگر V_k یک راس روی مسیر بهینه از V_i به V_j باشد آنگاه زیر مسیرهای V_i به V_k و V_k به V_j نیز باید بهینه باشند.

ولی مثلاً در مسئله یافتن طولانی ترین مسیر بین دو راس اصل بهینگی صادق نبوده و نمی توان آن را از طریق برنامه نویسی پویا حل کرد .

جهت نمونه در شکل زیر :

طولانی ترین مسیر ساده (مسیر بهینه) از V_1 به V_4 ، مسیر $\langle V_1, V_2, V_3, V_4 \rangle$ می باشد ولی زیر مسیر $\langle V_1, V_3 \rangle$ یک زیر مسیر بهینه (طولانی ترین) از V_1 به V_3 نیست و مسیری طولانی تر به صورت $\langle V_1, V_2, V_3 \rangle$ بین آن دو وجود دارد.



مثال چهارم : ضرب زنجیره ای ماتریس ها

همان طور که میدانید ضرب ماتریس ها خاصیت جابه جایی ندارد ولی خاصیت شرکت پذیری را دارد . همچنین ضرب دو ماتریس $A \times B$ به شرطی تعریف شده است که وسط آن ها یکسان باشد . یعنی تعداد ستون های A با تعداد سطرهای B برابر باشد.

می توان اثبات کرد که دو ماتریس $A_{\mathrm{m\times n}} \times \mathrm{B}_{\mathrm{n\times k}}$ در کل به $k \times n \times m$ عمل ضرب نیاز دارد .

حال به مثال زیر توجه کنید :

مثال : ضرب ۳ ماتریس $A_{5\times10}\mathrm{B}_{10\times20}\mathrm{C}_{20\times4}$ را به چه ترتیبی انجام دهیم تا سریع تر انجام شود ؟

حل :

اگر A و B را در هم ضرب کنیم و بعد در C ضرب کنیم :

$$(A_{5\times10}B_{10\times20}) \Rightarrow \text{تعداد ضربها} = 5\times10\times20 = 1000$$
$$(AB)_{5\times20}C_{20\times4} \Rightarrow \text{تعداد ضربها} = 5\times20\times4 = 400$$
$$\Rightarrow ((AB)C) \text{ تعداد ضربهای} = 1000+400=1400$$

ولی اگر B را در C ضرب کرده و سپس حاصل را در A ضرب کنیم :

$$(B_{10\times20}C_{20\times4}) \Rightarrow \text{تعداد ضربها} = 10\times20\times4 = 800$$
$$A_{5\times10}(BC)_{10\times4} \Rightarrow \text{تعداد ضربها} = 5\times10\times4 = 200$$
$$\Rightarrow (A(BC)) \text{ تعداد ضربهای} = 800+200 = 1000$$

پس A(BC) سریعتر از (AB)C انجام می پذیرد.

قضیه :

برای محاسبه ضرب سه ماتریس $M_{a\times b}N_{b\times c}P_{c\times d}$ برای آن که ترتیب (M.N).P زمان کمتری نسبت به M.(N.P) صرف کند می بایست داشته باشیم :

$$\frac{1}{b}+\frac{1}{d}<\frac{1}{a}+\frac{1}{c}$$

اثبات :

$$(MN)_{a\times c}P_{c\times d} \text{ تعداد ضربهای} = abc + acd$$
$$M_{a\times b}(NP)_{b\times d} \text{ تعداد ضربهای} = bcd + abd$$
$$\Rightarrow abc + acd < bcd + abd \xrightarrow{\text{abcd طرفین تقسیم بر}} \frac{1}{d} + \frac{1}{b} < \frac{1}{a} + \frac{1}{c}$$

اگر T(n) تعداد ترتیب های متفاوت برای ضرب n ماتریس A_1وA_2 .... A_n باشد ، زیر مجموعه از این ترتیب ها حالتی است که در آنها A_1 آخرین ماتریسی است که در مجکوعه ضرب می شود یعنی :

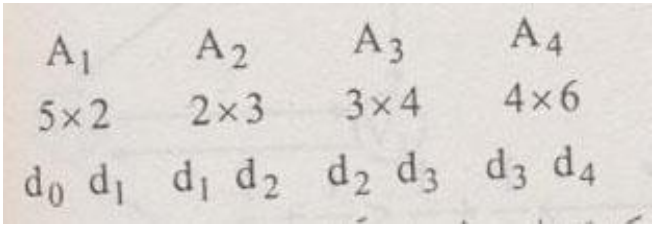
$$A_1\underbrace{(A_2A_3...A_n)}_{T(n-1)}$$

بنابراین تعداد ترکیب های ممکن برای ضرب ماتریس های A_2 تا A_n برابر T_{n-1} خواهد بود. حال به پرانتز $(A_2A_3 \ldots . A_n)$ توجه کنید در این پرانتز نیز یک زیر مجموعه عبارت از همه ترتیب هایی است که در آن A_n آخرین ماتریسی است که ضرب می شود .پس تعداد ترتیب های مختلف در این زیر مجموعه هم *T(n-1)* بوده و لذا داریم :

$$T(n) \geq T(n-1) + T(n-1) = 2T(n-1)$$
$$T(n) \geq 2T(n-1)$$

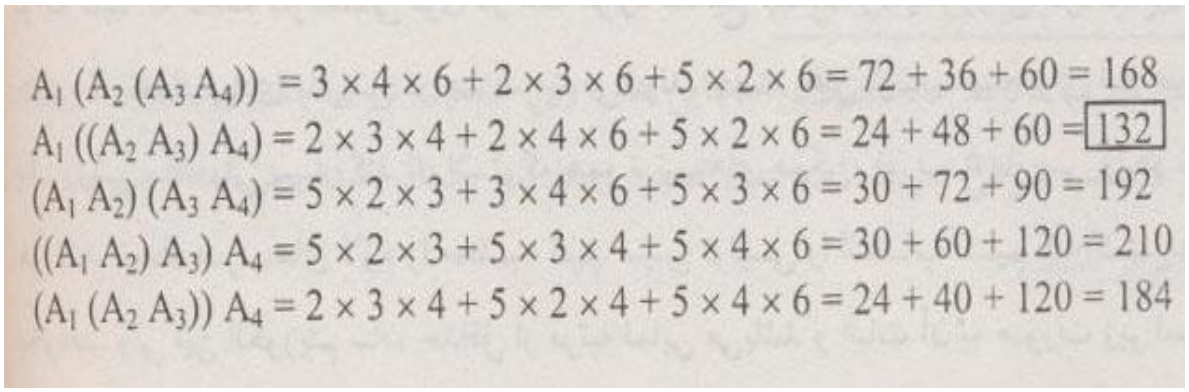
در این صورت روشن است که هریک از دو پرانتز فوق باید حداقل تعداد ضرب را داشته باشند تا ضرب دو پرانتز نیز حداقل تعداد ضرب را شامل باشند.

فرض کنید می خواهیم چهر ماتریس زیر را در یکدیگر ضرب کنیم :



تعداد ستون های ماتریس A_k را با d_k نمایش می دهیم . و از آن جا که تعداد سطرهای آن می بایست با تعداد ستون های ماتریس قبلی برابر باشد تعداد سطرهای ماتریس A برابر d_{k-1} است.

ابتدا مساله را از روش مستقیم که بسیار ناکارآمد است حل می کنیم . یعنی تمتمی حالات ممکن ضرب را یافته و بهترین ان را (که حداقل تعداد ضرب را دارد) انتخاب می کنیم . ۴ ماتریس را به ۵ طریق ممکن مشابه زیر می توان در یکدیگر ضرب کرد :



کمترین تعداد ضرب مربوط به حالت $A_1((A_2A_3)A_4)$ با ۱۳۲ ضرب است .دقت کنید قانون ساده سرانگشتی که می گفت " ابتدا آنهایی را ضرب کن که بعد وسطشان بزرگتر است "

در اینجا صادق نیست . حال یک روش حل پویا برای ایم مسئله پی ریزی می کنیم.

همانند مثال های قبلی برای حل این مسئله به صورت پویا از یک ماتریس *M[n][n]* استفاده میکنیم که *n* تعداد ماتریس هایی است که می خواهیم در یکدیگر ضرب شوند.

خانه های این ماتریس با مقادیر زیر پر می شوند :

$M[i][j] = A_j$ تا A_i حداقل تعداد ضرب‌های لازم برای ضرب	(برای $i < j$)
$M[i][j] = 0$	(برای $i = j$)

ضرب ۴ ماتریس $A_1A_2A_3A_4$ به صورت بازگشتی یکی از سه حالت زیر است ، یعنی اولین نقطه جداکننده می تواند بعد از A_1 یا بعد از A_2 یا بعد از A_3 باشد :

$$A_1(A_2A_3A_4)$$
$$(A_1A_2)(A_3A_4)$$
$$(A_1A_2A_3)A_4$$

تعداد حداقل ضرب ها برای هر یک از حالات فوق به صورت زیر است :

الف) $A_{1_{d_0\times d_1}}(A_2A_3A_4)_{d_1\times d_4} \Rightarrow \underbrace{M[1][1]}_{0} + M[2][4] + d_0d_1d_4$

عبارات بالا یعنی " حداقل تعداد ضرب ها را برای ضرب $(A_2A_3A_4)$ به دست بیاور و با $(d_0d_1d_4)$ که تعداد ضرب لازم برای انجام محاسبه $(A)_{d_0\times d_1}$ $(A_2A_3A_4)_{d_1\times d_4}$ است جمع کن . این حاصل حداقل ضرب ها را برای محاسبه$(A_2A_3A_4)$ A_1 می دهد . به همین ترتیب :

ب) $(A_1A_2)_{d_0\times d_2}(A_3A_4)_{d_2\times d_4} \Rightarrow M[1][2]+M[3][4]+d_0d_2d_4$

ج) $(A_1A_2A_3)_{d_0\times d_3}A_{4\,d_3\times d_4} \Rightarrow M[1][3]+M[4][4]+d_0d_3d_4$

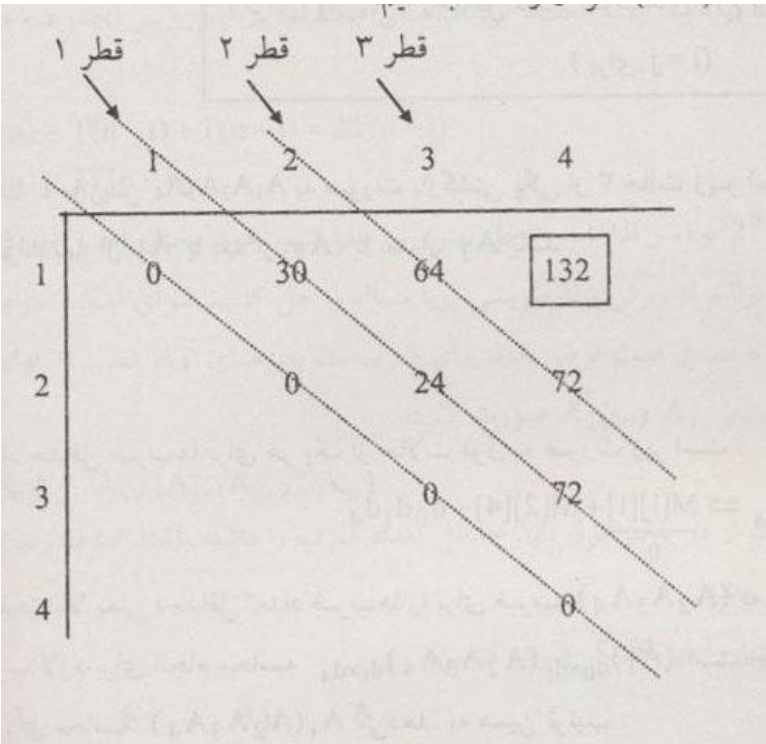
توجه کنید برای $i = j$ عبارت $M[i][j] = 0$ است چرا که تعداد ضرب های لازم برای ایجاد یک ماتریس تنهای (A_i) را نشان می دهد که هیچ ضربی نیاز ندارد .

پس از محاسبه عبارات الف و ب و ج جواب نهایی مینیمم بین آنها می باشد.

با توجه به مثال فوق می توان فرمول اصلی حل مساله فوق را نوشت :

$M[i][j] = 0$: (اگر $i = j$)

$M[i][j] = \min_{i\le k\le j-1}(M[i][k]+M[k+1][j]+d_{i-1}d_kd_j)$: (اگر $i < j$)



قطر ١ :

$$M[1][2] = \min_{1 \le k \le 1} (M[1][k] + M[k+1][2] + d_0 d_k d_2)$$

$$= M[1][1] + M[2][2] + d_0 d_1 d_2$$

$$= 0 + 0 + 5 \times 2 \times 3 = 30$$

$$M[2][3] = \min_{2 \le k \le 2} (M[2][2] + M[3][3] + d_1 d_2 d_3)$$

$$= 0 + 0 + 2 \times 3 \times 4 = 24$$

$$M[3][4] = 3 \times 4 \times 6 = 72$$

قطر ٢ :

$$M[1][3] = \min_{1 \le k \le 2} (M[1][k] + M[k+1][3] + d_0 d_k d_3)$$

$$\text{Min}((M[1][1] + M[2][3] + d_0 d_1 d_3),\ (M[1][2] + M[3][3] + d_0 d_2 d_3))$$

$$= \text{Min}((0 + 24 + 5 \times 2 \times 4),\ (30 + 0 + 5 \times 3 \times 4))$$

$$= \text{Min}(64, 90) = 64$$

$$M[2][4] = \min_{2\le k\le 3}(M[2][k]+M[k+1][4]+d_1 d_k d_4)$$
$$= \text{Min}((M[2][2] + M[3][4] + d_1d_2d_4)\ ,\ (M[2][3] + M[4][4] + d_1d_3d_4))$$
$$= \text{Min}((0 + 72) + 2 \times 3 \times 6)\ ,\ (24 + 0 + 2\times 4 \times 6))$$
$$= \text{Min}(108\ ,\ 72) = 72$$

قطر ۳ :

$$M[1][4] = \min_{1\le k\le 3}(M[1][k]+M[k+1][4]+d_0 d_k d_4)$$
$$=\text{Min}((M[1][1]+M[2][4]+d_0d_1d_4),(M[1][2] + M[3][4]+d_0d_2d_4),(M[1][3] + M[4][4] +d_0d_3d_4))$$
$$= \text{Min}((0 + 72 + 5 \times 2 \times 6)\ ,\ (30 + 72 + 5\times 3 \times 6)\ ,(64 + 0 + 5 \times 4 \times 6))$$
$$= \text{Min}(132\ ,\ 192\ ,\ 184) = \underline{\mathbf{132}}$$

توجه کنید برای قطر ۱ ، مینیمم یک عبارت ، برای قطر ۲ مینیمم بین ۲ عبارت و برای قطر ۳ ، مینیمم بین ۳ عبارت را محاسبه کردیم.

جواب نهایی مورد نظر M[1][4] = 132 است که نشان می دهد برای ضرب A_1 تا A_4 به حداقل ۱۳۲ ضرب نیاز است.

برای یافتن ترتیب ضرب بهینه یعنی($A_1((A_2A_3)A_4$ کافی است کنار ماتریس M یک ماتریس P را نیز بسازیم.

برای مثال فوق کار را از عنصر M[1][4] شروع می کنیم.

در هنگام محاسبه این عنصر داشتیم :

$$M[1][4] = \text{Min}\ (\underbrace{132}_{k=1}, \underbrace{192}_{k=2}, \underbrace{184}_{k=3}) = 132$$

یعنی حداقل مربوط به K = 1 است . این بدان معناست که برای ضرب A_1 تا A_4 ابتدا باید ترتیب را در A_1 بشکنیم : $(A_1)(A_2A_3A_4)$

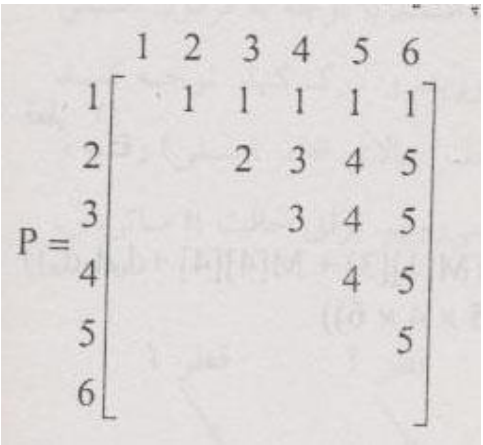
حال به سراغ M[2][4] می رویم که ببینیم برای ضرب A_2 تا A_4 عملیات در کجا باید شکسته شود.

$$M[2][4] = \text{Min}\,(\underbrace{108}_{k=2},\ \underbrace{72}_{k=3}) = 72$$

پس باید پرانتز بسته در نقطه K=3 گذاشته شود یعنی $(A_2A_3)A_4$ بدین ترتیب ماتریس P برابر زیر خواهد شد که از روی آن به راحتی میتوان ترتیب بهینه را چاپ کرد :

$$P = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & \\ & & 2 & 3 \\ & & & 3 \\ & & & \end{bmatrix} \end{array}$$

مثال : میخواهیم ۶ ماتریس را در هم ضرب کنیم $A_1A_2A_3A_4A_5A_6$ ماتریس P آنها طبق الگوریتم فوق به صورت زیر درآمده است . ترتیب بهینه را برای حداقل تعداد ضرب ها چاپ کنید .



$$P = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ & & 2 & 3 & 4 & 5 \\ & & & 3 & 4 & 5 \\ & & & & 4 & 5 \\ & & & & & 5 \\ & & & & & \end{bmatrix} \end{array}$$

حل : عنصر $P[1][6] = 1$ است یعنی اولین جداسازی بعد از A_1 صورت می گیرد :

$$(A_1)(A_2A_3A_4A_5A_6)$$

سپس به سراغ $P[2][6] = 5$ می رویم . یعنی جداسازی بعدی از A_5 است :

$$(A_1)((A_2A_3A_4A_5)A_6)$$

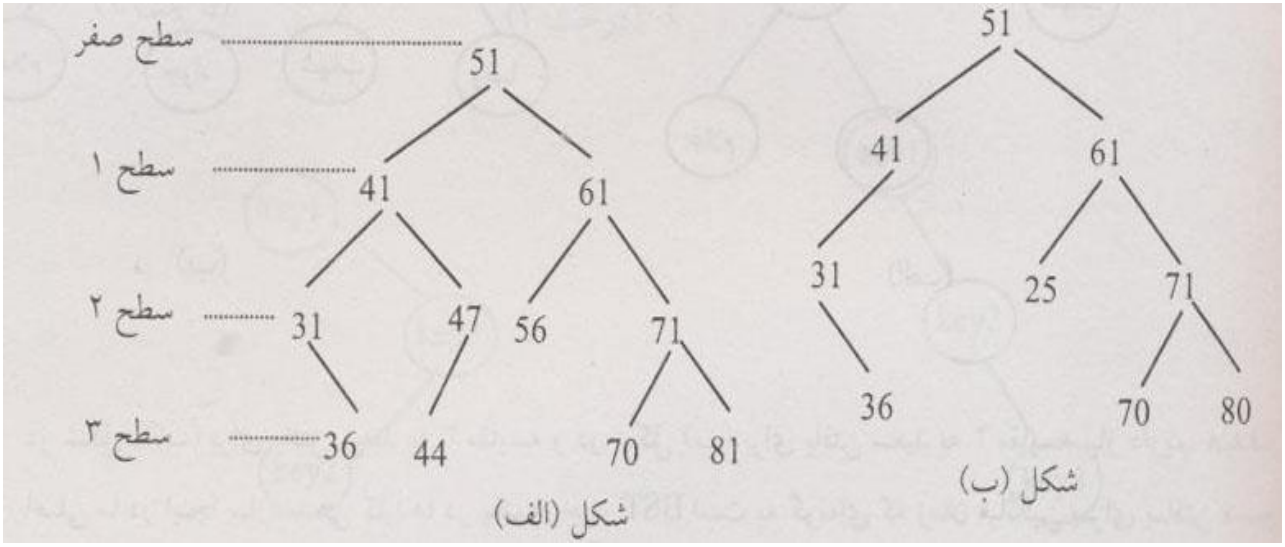
حال به سراغ $P[2][5] = 4$ می رویم ، یعنی جداسازی بعدی از A_4 است :

$$(A_1)((((A_2A_3)A_4)A_5)A_6)$$

مثال پنجم : درخت های جست و جوی دودویی بهینه

همان طور که در درس ساختمان داده ها خوانده اید درخت جستجوی دودویی یا BST (Binary Search Tree) یک درخت دودویی است که مقدار هر گره بزرگ تر از هر مقدار زیر درخت چپ و کوچک تر از هر مقدار در زیر درخت راست آن می باشد . هر گره دارای یک کلید است و عموماً دو گره نباید دارای کلید

یکسان باشند (یعنی عموماً کلیدها منحصر به فرد هستند) . مثلاً شکل (الف) زیر یک درخت BST است ولی شکل (ب) BST نیست . چرا که در زیر درخت سمت راست گره ۵۱ عدد ۲۵ از آن کوچک تر است :



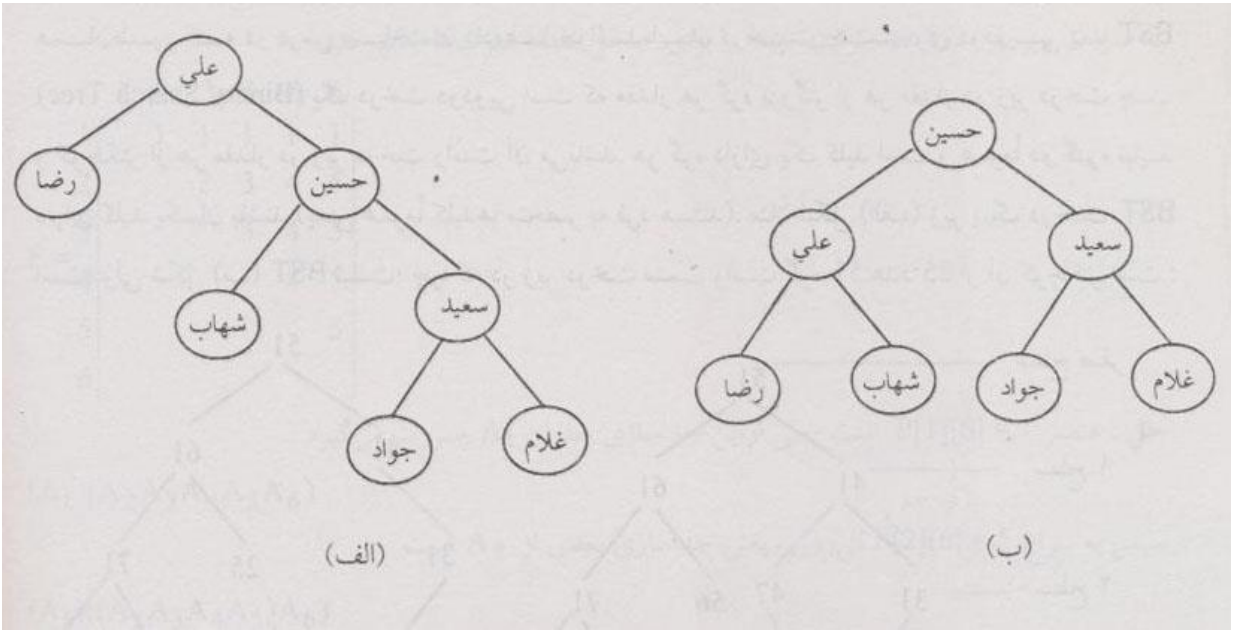
الگوریتم جستجو در درخت BST

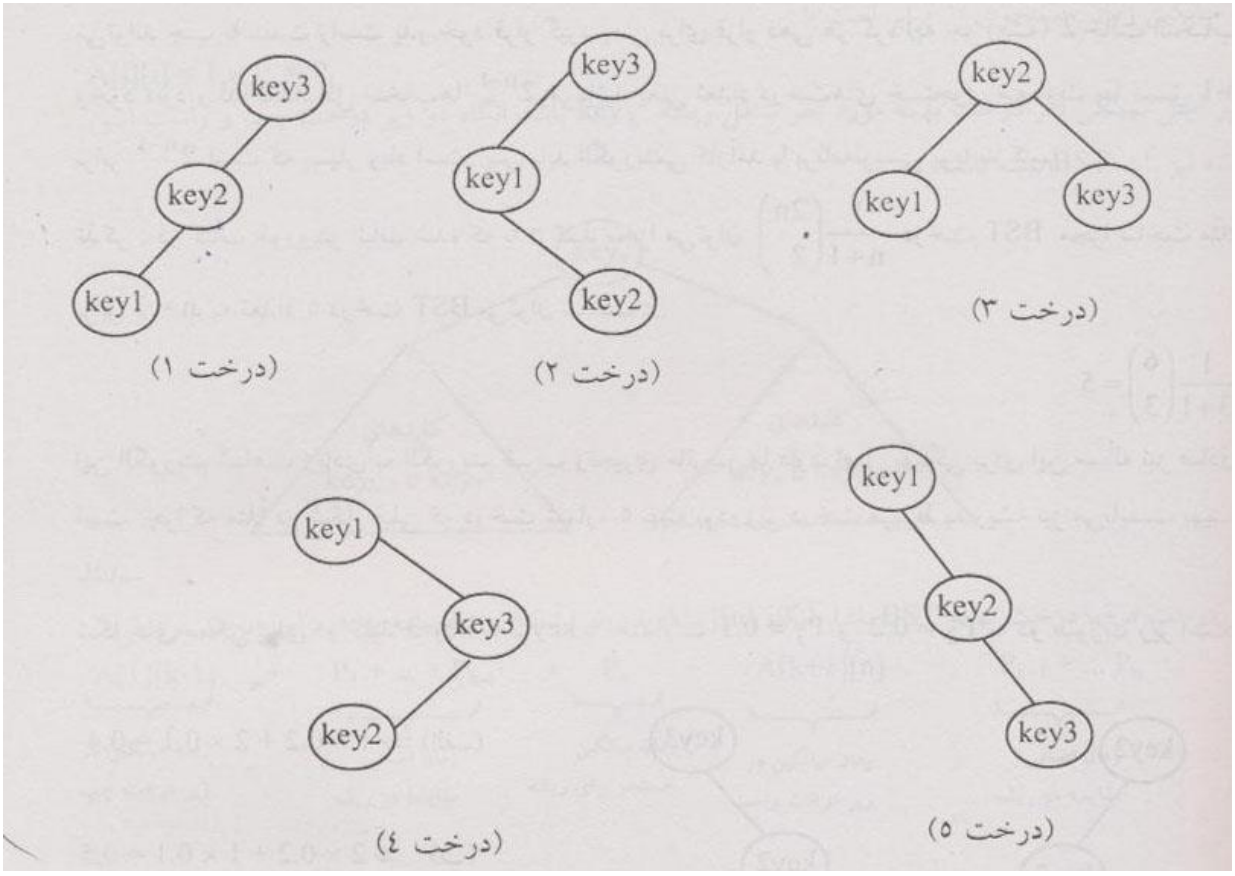
فرض کنید دنبال کلیدی هستیم که می دانیم حتماً در درخت است . جهت جستجو همواره از ریشه شروع می کنیم . اگر کلید مورد جستجو از کلید گره بزرگ تر بود به سمت راست و اگر کوچک تر بود به سمت چپ می رویم . این عمل را آن قدر تکرار می کنیم تا بالاخره داده مورد نظر پیدا شود.

بدیهی است تعداد مقایسه ها برای یافتن یک کلید Key برابر عبارت زیر است که منظور از depth(key) عمق آن کلید است.

depth(key)+1

مثلاً در شکل های زیر :





حل :

درخت ۱ $\Rightarrow 3 \times 0.7 + 2 \times 0.2 + 1 \times 0.1 = 2.6$

درخت ۲ $\Rightarrow 2 \times 0.7 + 3 \times 0.2 + 1 \times 0.1 = 2.1$

درخت ۳ $\Rightarrow 2 \times 0.7 + 1 \times 0.2 + 2 \times 0.1 = 1.8$

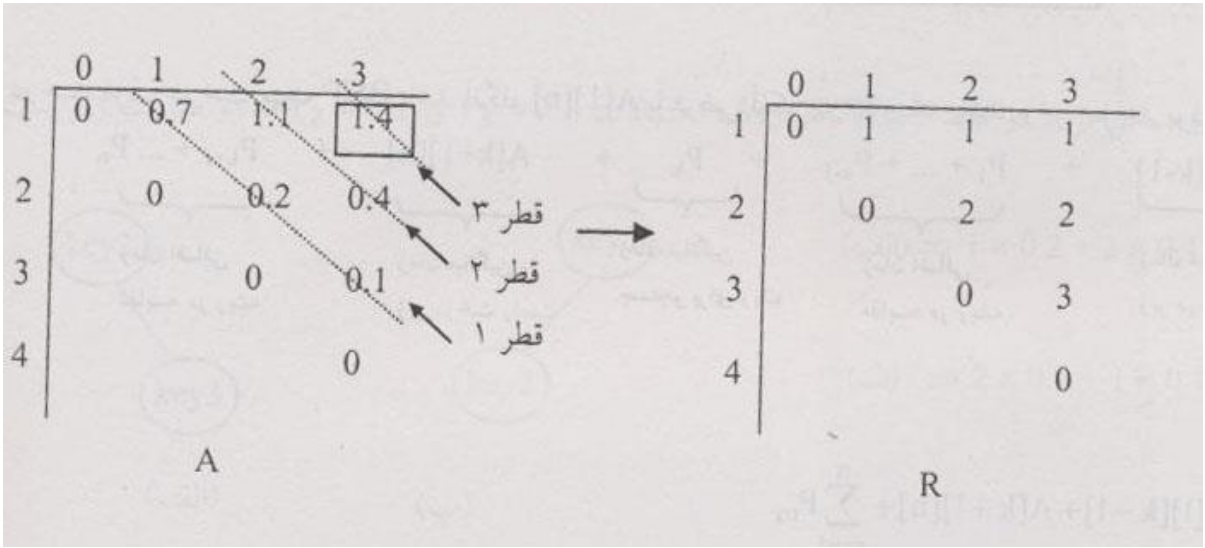
درخت ٤ $\Rightarrow 1 \times 0.7 + 3 \times 0.2 + 2 \times 0.1 = 1.5$

درخت ٥ $\Rightarrow 1 \times 0.7 + 2 \times 0.2 + 3 \times 0.1 = 1.4$

مثال : درخت BST بهینه مربوط به کلیدهای زیر را با احتمالات داده شده ترسیم کنید :

key_1	key_2	key_3
$P_1=0.7$	$P_2=0.2$	$P_3=0.1$

حل :



قطر ۱ : از انجا که در قطر ۱ داریم $A[I][I] = P_I$ به راحتی احتمالات داده شده را در قطر مربوطه قرار می دهیم .

به همین ترتیب $R[i][i] = i$ می باشد چرا که در درخت BST بهینه از کلید های Key_i تا key_i بدیهی است که اندیس ریشه همان i است. (چرا که این درخت فقط یک گره دارد)

$$A[1][2] = \min_{1\le k\le 2}(A[1][k-1] + A[k+1][2]) + \sum_{m=1}^{2} P_m$$
$$= \text{Min}(A[1][0] + A[2][2], A[1][1] + A[3][2]) + (P_1 + P_2)$$
$$= \text{Min}(\underbrace{0+0.2}_{k=1}, \underbrace{0.7+0}_{k=2}) + (0.7+0.2) = 1.1$$

پس در خانه $R[1][2]$ عدد $k=1$ را قرار می دهیم .

$$A[2][3] = \min_{2\le k\le 3}(A[2][k-1] + A[k+1][3]) + \sum_{m=2}^{3} P_m$$

$$= \text{Min}(A[2][1] + A[3][3], A[2][2] + A[4][3]) + P_2 + P_3$$
$$= \text{Min}(\underbrace{0+0.1}_{k=2}, \underbrace{0.2+0}_{k=3}) + 0.2 + 0.1 = 0.4$$

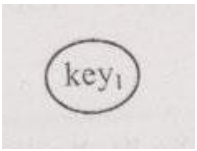
و در خانه $R[2][3]$ عدد $k=2$ را قرار می دهیم .

قطر ۳

$$A[1][3] = \min_{1\le k\le 3}(A[1][k-1] + A[k+1][3]) + \sum_{m=1}^{3} P_m$$
$$= \text{Min}(A[1][0] + A[2][3], A[1][1] + A[3][3], A[1][2] + A[4][3]) + P_1 + P_2 + P_3$$
$$= \min(\underbrace{0 + 0.4}_{k=1}, \underbrace{0.7 + 0.1}_{k=2}, \underbrace{1.1 + 0}_{k=3}) + 0.7 + 0.2 + 0.1 = 1.4$$

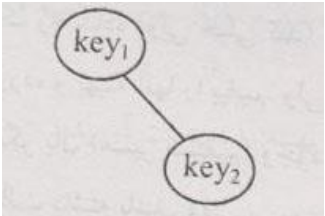
و در خانه R[1][3] عدد k =1 را قرار می دهیم.

پس درخت BST بهینه باکلیدهای Key_1 تا key_3 در کل به زمان جستجوی میانگین 1.4 نیاز دارد.برای ترسیم شکل به ماتریس R نگاه می کنیم برای Key_1 تا key_3 ریشه Key_1 می باشد. (R[1][3]=1) پس آن را در ریشه قرار می دهیم :



برای Key_2 تا key_3 ریشه Key_2 است چرا که R[2][3] می باشد.

توجه کنید چون می دانیمKey_1 < Key_2 است ان را در سمت راست Key_1 قرار داده ایم .



در آخر هم key_3 را در سمت راست Key_2 رسم می کنیم .

پیاده سازی الگوریتم فوق را به عنوان تمرین بر عهده دانشجویان قرار می دهیم . البته کدهای آن را به زبان C می توانید در کتاب بیپولیتان مشاهده کنید.

مشابه الگوریتم ضرب ماتریس ها این الگوریتم نیز نیاز به سه حلقه تو در تو دارد و لذا مرتبه اجرایی آن $\theta(n^3)$ است.

مثال ششم : فروشنده دوره گرد

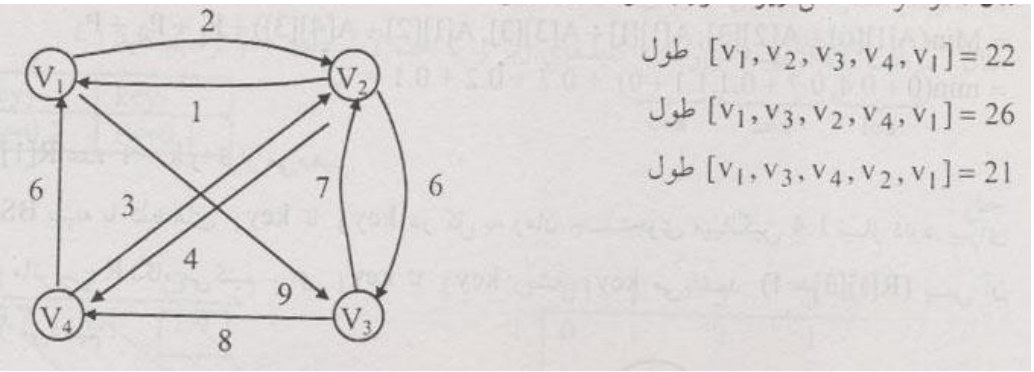
فروشنده ای می خواهد از تعدادی شهر که توسط جاده هایی به یکدیگر متصل هستند عبور کند و در آخر به شهر اولیه خود برگردد .

هدف یافتن کوتاه ترین مسیر برای فروشنده است به نحوی که از تمام شهرها دقیقاً یک بار عبور کند و این مساله استاندارد فروشنده دوره گرد است.

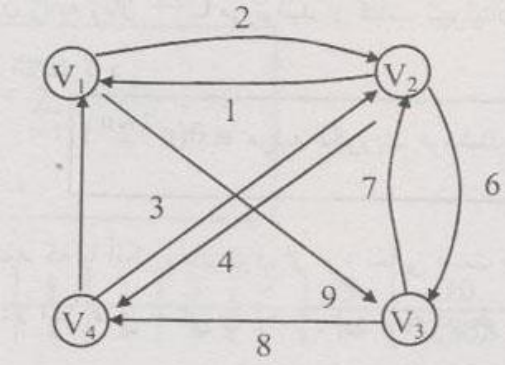
در یک گراف جهت دار یک تور ، که به آن مدار هامیلتون نیز گفته می شود عبارت از مسیری از یک راس به خودش است که از تمام رئوس دیگر دقیقاً یک بار عبور می کند . منظور از یک تور بهینه در گراف جهت دار وزن دار مسیری از یک نوع است که طول آن حداقل می باشد .

بنابراین مسئله فروشنده دوره گرد در واقع پیدا کردن یک تور بهینه برای یک گراف جهت دار موزون است . توجه کنید که ممکن است گرافی اصلاً تور نداشته باشد.همچنین طول تور بهینه وابسته به انتخاب راس آغازین نیست.

مثال : در گراف شکل زیر ۳ تور با طول های تعیین شده عبارت اند از :



مثال : در گراف جهت دار موزون زیر با ماتریس هم جواری W داده شده ، طول تور بهینه را به دست آورید.



$$W = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 2 & 9 & \infty \\ 1 & 0 & 6 & 4 \\ \infty & 7 & 0 & 8 \\ 6 & 3 & \infty & 0 \end{bmatrix} \end{array}$$

$$D[v_4][\{v_2\}] = W[4][2] + D[2][\phi] = 3 + 1 = 4$$
$$D[v_2][\{v_3\}] = W[2][3] + D[v_3][\phi] = 6 + \infty = \infty$$
$$D[v_4][\{v_3\}] = W[4][3] + D[v_3][\phi] = \infty + \infty = \infty$$
$$D[v_2][\{v_4\}] = W[2][4] + D[v_4][\phi] = 4 + 6 = 10$$
$$D[v_3][\{v_4\}] = W[3][4] + D[v_4][\phi] = 8 + 6 = 14$$

قدم سوم : مجموعه A را دو عضوی می گیریم :

$$D[v_4][\{v_2, v_3\}] = \min_{v_i \in \{v_2, v_3\}} (W[4][j] + D[v_j][\{v_2, v_3\} - \{v_j\}])$$
$$= \text{Min}(W[4][2] + D[v_2][\{v_3\}] \ , \ W[4][3] + D[v_3][\{v_2\}])$$
$$= \text{Min}(3 + \infty, \infty + 8) = \infty$$
$$D[v_3][\{v_2, v_4\}] = \text{Min}(w[3][2] + D[v_2][\{v_4\}] \ , \ W[3][4] + D[v_4][\{v_2\}])$$
$$= \text{Min}(7 + 10{,}8 + 4) = 12$$
$$D[v_2][\{v_3, v_4\}] = \text{Min}(W[2][3] + D[v_3][\{v_4\}] \ , \ W[2][4] + D[v_4][\{v_3\}])$$
$$= \text{Min}(6 + 14{,}4 + \infty) = 20$$

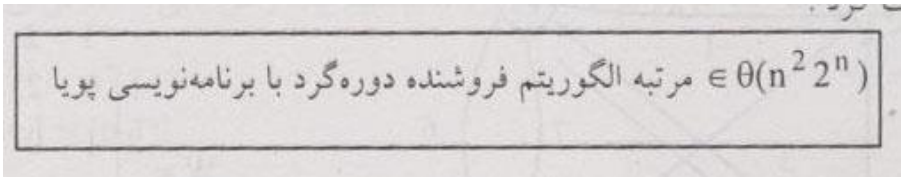
قدم چهارم : در آخر مجموعه A را سه عضوی می گیریم :

$$D[v_1][\{v_2, v_3, v_4\}] = \min_{v_j \in \{v_2, v_3, v_4\}} (W[1][j] + D[v_j][\{v_2, v_3, v_4\} - \{v_j\}])$$
$$= \text{Min}(W[1][2] + D[v_2][\{v_3, v_4\}], W[1][3] + D[v_3][\{v_2, v_4\}], W[1][4] + D[v_4][\{v_2, v_3\}])$$
$$= \text{Min}\ (2 + 20\ ,\ 9 + 12\ ,\ \infty + \infty) = 21$$

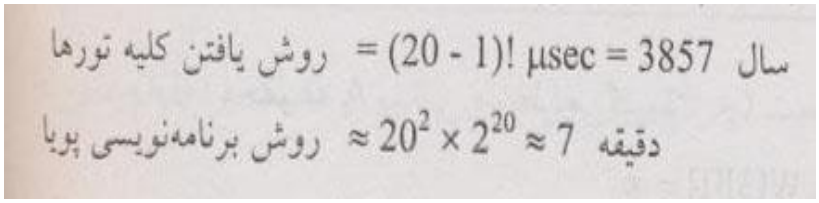
پس طول تور بهینه برابر ۲۱ می باشد.

نحوه پیاده سازی الگوریتم فوق در یک زبان برنامه نویسی به عنوان تمرین بر عهده دانشجویان گذاشته می شود .

برنامه آن را به زبان C++ می توانید از کتاب نیپولیتان مطالعه کنید.



توجه کنید با آنکه زمان فوق از نوع نمایی است ولی به مراتب بهتر از زمان الگوریتم استاندارد یعنی $(n-1)!$ است . مثلاً اگر عمل اصلی را یک میکرو ثانیه در نظر بگیریم برای گرافی با n =20 راس داریم :



البته الگوریتم $\theta(n^2 2^n)$ نیز هنگامی عملی است که n نسبتاً کوچک باشد چرا که مثلاً برای $n=$ *70* نیز این الگوریتم سال ها وقت نیاز دارد.

نکته ۱ : می توان اثبات کرد حافظه مورد نیاز این الگوریتم از مرتبه($\theta(n2^n$ است.

مثال هفتم : بزرگ ترین زیر دنباله (رشته) مشترک *(LCS)*

دو رشته *X= <A,B,C,B,D,A,B>* و *Y=<B,D, C, A,B,A>* را در نظر بگیرید. *Z= <B,C,A>* .یک زیر رشته مشترک برای این دو رشته است .

توجه کنید که *<B,C,A>* لازم نیست دقیقاً پشت سر هم در دو رشته *X* و *Y* ظاهر شده باشد بلکه تنها لازم است توالی ظاهر شدن (اول *B* ، بعد *C* و بعد *A*) در دو رشته وجود داشته باشد.

به عبارت دقیق تر رشته $Z = <z_1 , z_2, ...z_k>$ یک زیر رشته برای رشته $X=<x_1 , x_2,...,x_n>$ ، $k \geq n$ است هر گاه یک رشته اکیداً صعودی مثل $< i_1 i_2, ... , i_n >$

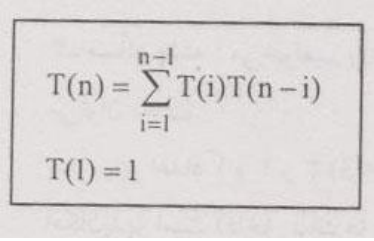
از اندیس های *x* وجود داشته باشد به گونه ای که برای همه *j= 1,2,....,k* رابطه $x_{i_j} = z_j$ برقرار باشد.

در مثال فوق زیر رشته *<B, C,A>* به طول ۳طولانی ترین زیر رشته مشترک در *X* و *Y* نیست . بلکه رشته *<B,C,B,A>* به طول ۴ طولانی ترین زیر دنباله مشترک آن ها است.

در کتاب آقای قلی زاده یک روش پویا برای حل این مساله ارایه شده است.که از مرتبه $\theta(n,m)$ است.n و m طول دو رشته مورد نظر هستند. *X=<x1 , x2,...,xn>* و $Y = < y_1, y_2, ... y_n >$. بررسی این روش را در قسمت مطالعه و تحقیق بر عهده دانشجویان گذتشته ایم.

سری کاتان

سری معروف کاتان به صورت زیر بوده و جواب بسیاری از مسائل شمارشی می باشد :



برخی ار مقادیر این سری عبارت اند از :

n	1	2	3	4	5	10	15
T(n)	1	1	2	5	14	4,862	2,674,440

می توان اثبات کرد (به کتاب هورویتز رجوع کنید) که سری فوق معادل فرمول زیر است :

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i) = \frac{1}{n}\binom{2n-2}{n-1}$$

حال چند مساله استاندارد را مطرح می کنیم که تعداد حالات مختلف آنها با سری فوق محاسبه می شود.

۱- **مساله ضرب ماتریس ها :** در قسمت های قبلی گفتیم تعداد حالات مختلفی که می توان n ماتریس را در هم ضرب کرد حداقل از مرتبه نمایی است.

ولی می خواهیم تعداد دقیقاین حالات را به دست آوریم . فرض کنید در اولین مرحله ضرب ماتریس ها را در نقطه i به دو دسته تقسیم کرده ایم یعنی : $(A_1A_1 \ldots . A_i)(A_{i+1}A_{i+2} \ldots . A_n)$ اگر T(i) تعداد حالات ممکن برای ضرب پرانتز سمت چپ باشد.پس T(n –i) تعداد حالات ممکن برای ضرب پرانتز سمت راست بوده و در کل داریم :

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i) \quad , \quad T(1) = 1$$

که حل رابطه فوق به فرمول زیر می رسد.

$$\Gamma(n) = \frac{1}{n}\binom{2n-2}{n-1}$$

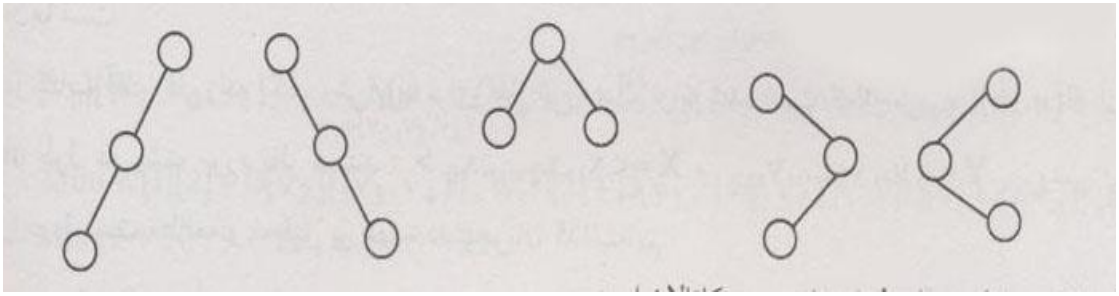
ای ضرب ۴ ماتریس , ۵ حالت زیر امکان پذیر است :

A((BC)D) , A(B (CD)) , ((AB) C) D , (A(BC)) D , (AB) (CD)

۲- مساله تعداد درخت های دودویی :

می خواهیم بدانیم با n گره چند فرم درخت دودویی می توان ساخت .

مثلاً برای n=3 جواب ۵ است.



۳- مساله پشته :

می خواهیم بدانیم با n عدد ورودی که به ترتیب وارد یک پشته می شوند , چند حالت مختلف می توان داشت .

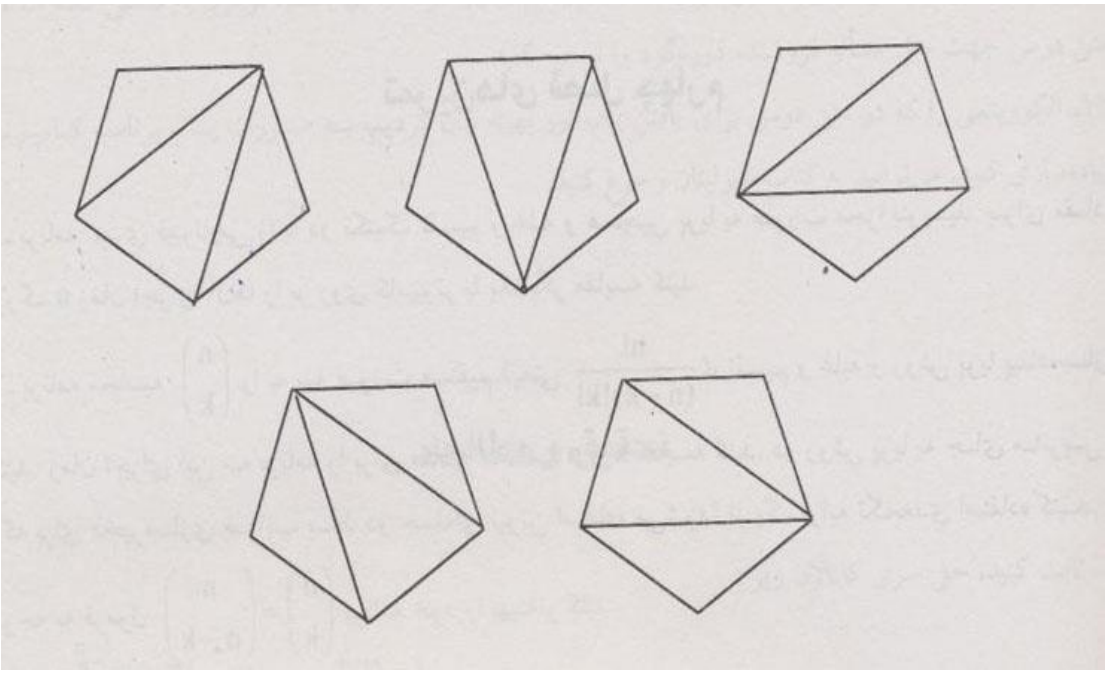
مثلاً برای اعداد ۱و ۲و ۳ (n=3) که به ترتیب از راست به چپ وارد پشته می شوند ۵ حالت خروجی زیر امکان پذیر است (داخل پرانتزها را از چپ به راست بخوانید) :

(1,2,3) , (1,3,2) , (2,1,3) , (2,3,1), (3,2,1)

پس حل مساله فوق مشابه تعداد درخت های دودویی معادل جمله n+1 از سری کاتان است.

۴- مساله مثلث بندی چند ضلعی محدب :

می خواهیم بدانیم به چند طریق می توان قطرهای نامتقاطع یک چند ضلعی محدب را رسم کرد و آن را به مثلث های مختلف تقسیم نمود.در حالت کلی یک n ضلعی همواره با n-3 قطر به n-2 مثلث افراز می شود.مثلاً برای n= 5 به ۵ صورت مختلف می توان قطرها را بر طبق جملات فوق رسم کرد :

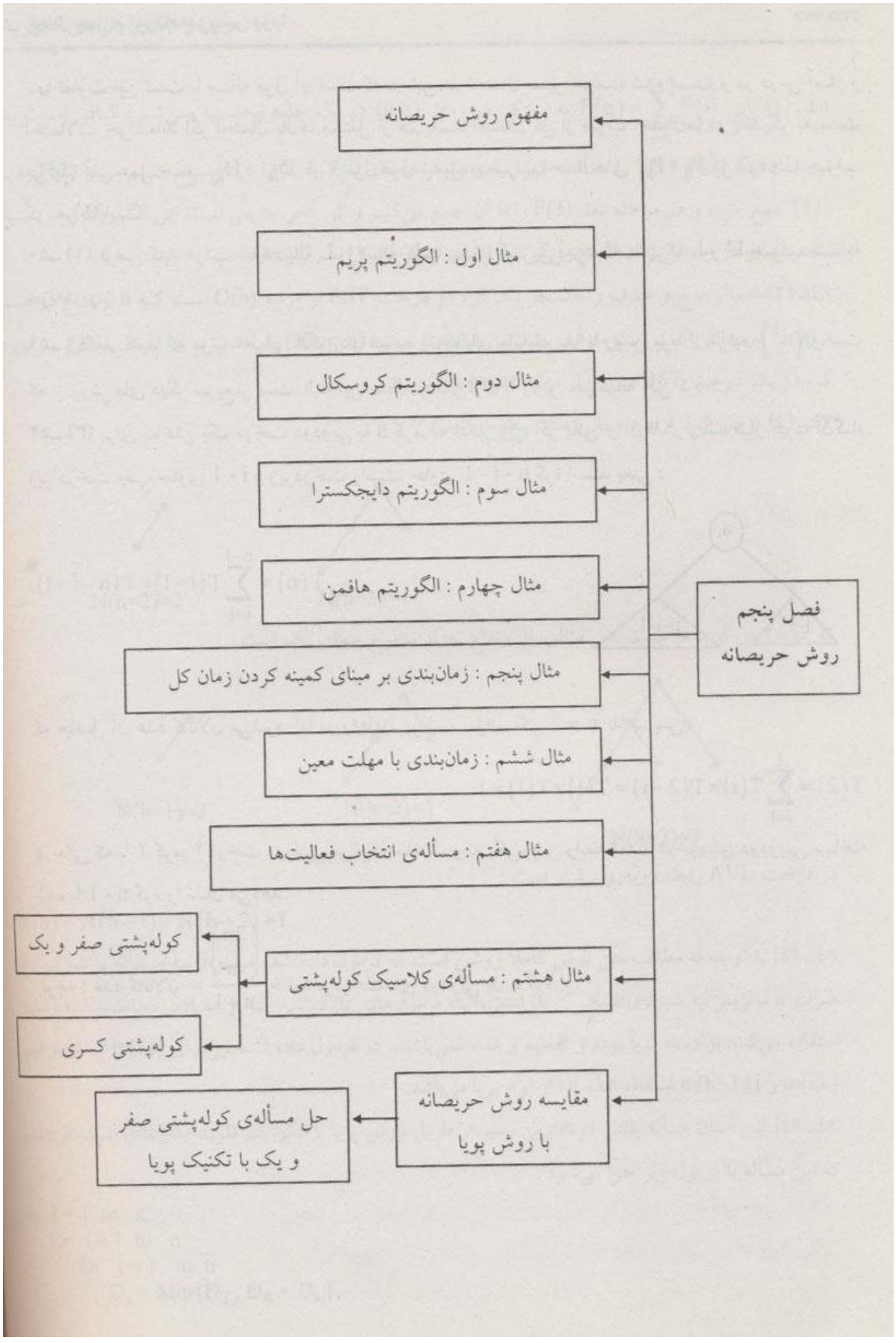


پس تعداد حالات مساله فوق معادل (n-1) امین عدد کاتان است :

$$T(n-1)=\sum_{i=1}^{n-2}T(i)T(n-1-i)=\frac{1}{n-1}\binom{2n-4}{n-2}$$

فصل پنجم

روش حریصانه


فصل پنجم
روش حریصانه
مفهوم روش حریصانه
مثال اول : الگوریتم پریم
مثال دوم : الگوریتم کروسکال
مثال سوم : الگوریتم دایجکسترا
مثال چهارم : الگوریتم هافمن
مثال پنجم : زمان‌بندی بر مبنای کمینه کردن زمان کل
مثال ششم : زمان‌بندی با مهلت معین
مثال هفتم : مسأله‌ی انتخاب فعالیت‌ها
مثال هشتم : مسأله‌ی کلاسیک کوله‌پشتی
کوله‌پشتی صفر و یک
کوله‌پشتی کسری
مقایسه روش حریصانه
با روش پویا
حل مسأله‌ی کوله‌پشتی صفر
و یک با تکنیک پویا

مفهوم روش حریصانه :

نام این روش از شخصیت معروف اسکروج گرفته شده است . اسکروج به جای آن که به گذشته و آینده فکر کند تنها انگیزه هر روز او به دست آوردن طلای بیشتر بود . الگوریتم حریصانه ***(Greedy)*** نیز مانند شیوه اسکروج می باشد.

الگوریتم های حریصانه یا آزمند شبیه روش های پویا اغلب برای حل مسائل بهینه سازی استفاده می شوند . در شیوه حریصانه در هر مرحله عنصری که بر مبنای معیاری معین "بهترین " به نظر می رسد بدون توجه به انتخاب هایی که قبلاً انجام شده یا در آینده انجام خواهد شد , انتخاب می شود .الگوریتم های حریصانه اغلب راه حل هایی ساده هستند .در روش حریصانه بر خلاف روش پویا , مساله به نمونه های کوچک تر تقسیم نمی شود.

مثال ۲ : فرض کنید به فردی قرار است ۱۶ تومان پس بدهیم سکه های موجود ***1, 5,10 , 12 , 25*** تومانی است.(با اینکه ما در ایران سکه ۱۲ تومانی نداریم ولی این فرض را بکنید که داریم)

با الگوریتم حریصانه فوق به این نتیجه می رسیم که باید به آن فرد یک سکه ۱۲ تومانی و ۴ سکه یک تومانی بدهیم یعنی جمعاً ۵ سکه . در حالی که حل بهینه مساله فوق یک سکه ۱۰ تومانی , یک سکه ۵ تومانی و یک سکه یک تومانی است.یعنی در کل سه سکه.

از مثال فوق نتیجه می گیریم که هر الگوریتم حریصانه الزاماً حل بهینه را نمی دهد و برای هر مساله خاص باید اثبات کنیم که آیا الگوریتم حریصانه برای آن , جواب بهینه می دهد یا خیر و این موضوع اغلب سخت ترین مرحله کار است.

با توجه به مثال های فوق می توان مراحل روش حریصانه را به این صورت بیان کرد :

کار با یک مجموعه تهی شروع شده و به ترتیبی خاص عناصری به مجموعه اضافه می شوند .هر دور تکرار الگوریتم شامل بخش های زیر است :

۱- **روال انتخاب :** این روال عنصر بعدی را طبق یک معیار حریصانه انتخاب می کند . این انتخاب یک شرط بهینه را در همان برهه برآورده می سازد.

۲- **تحقیق عملی بودن :** در این مرحله مشخص می شود که آیا مجموعه جدید به دست آمده برای رسیدن به حل عملی است یا خیر.

۳- **تحقیق حل** : مشخص می سازد که آیا مجموعه جدید نمونه مورد نظر را حل کرده است یا خیر .

در ادامه مثال هایی را بیان می کنیم که با این روش حل می شوند و در آخر مثالی را می آوریم که با روش حریصانه قابل حل نبوده و باید روش پویا برای حل آن استفاده کرد وبدین ترتیب روش پویا و حریصانه را با یکدیگر مقایسه می کنیم تا دریابیم در هر مورد بهتر است از کدام روش استفاده کرد.

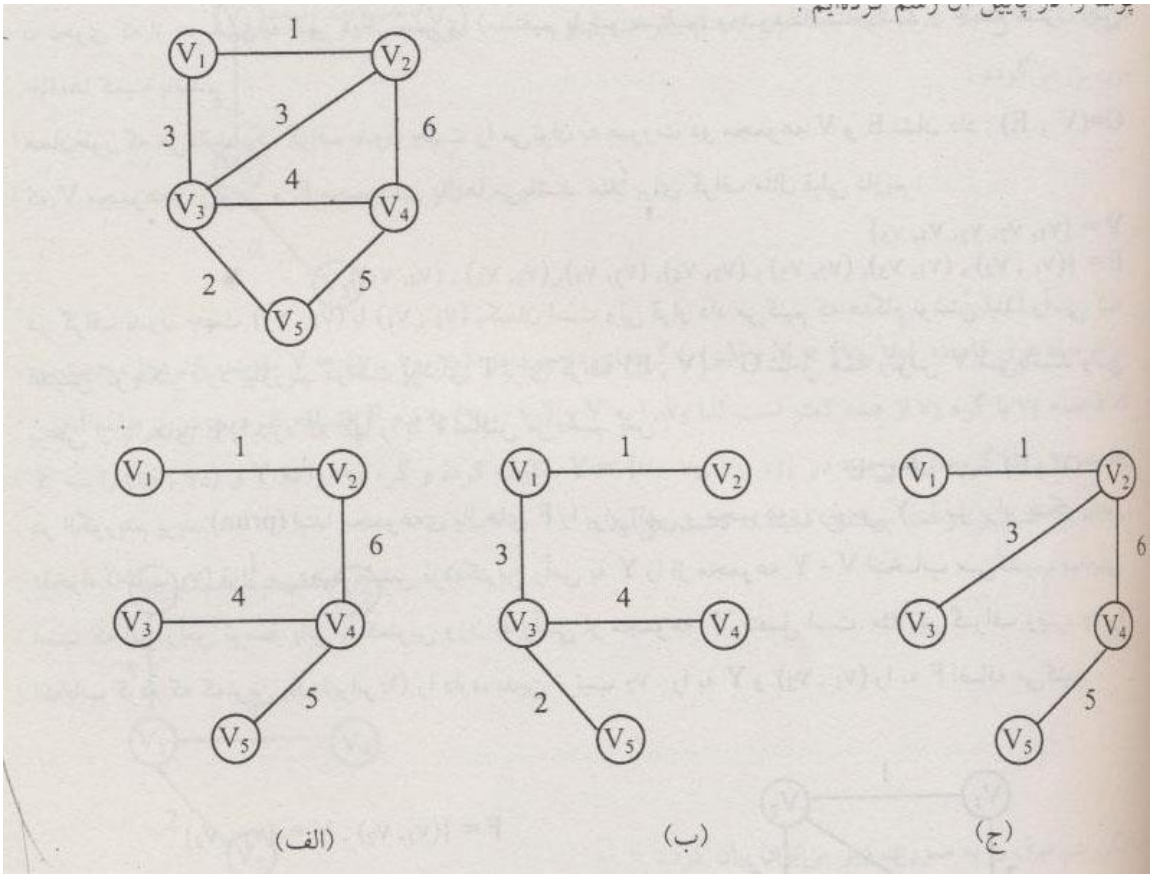
مثال اول : الگوریتم پریم ***(prim)***

در درس ساختمان داده ها خوانده ایک که درخت یک گراف بدون جهت , متصل و بی چرخه است . به عبارتی دیگر درخت یک گراف بدون جهت است که در آن بین هر جفت از رئوس فقط و فقط یک مسیر وجود دارد.

توجه کنید در این تعریف هیچ گره ای را به عنون ریشه در نظر نمی گیریم ولی در درخت ریشه دار ***(rooted tree)*** یکی از راس ها ریشه می باشد و اغلب منظور از درخت , درخت ریشه دار است . البته در این قسمت فرض می کنیم با درخت هایی سر و کار داریم که ریشه آن ها برای ما مهم نیست .

درخت پوشا ***(spanning tree)*** برای یک گراف G , یک زیر گراف متصل می باشد که حاوی همه راس های گراف G بوده و همچنین یک درخت است به عبارت دیگر درخت پوشا شامل همه رئوس و برخی یال های گراف است به نحوی که متصل بوده و چرخه نیز ندارد .

مثلاً برای گراف بدون جهت وزن دار زیر سه درخت پوشا را در پایین آن رسم کرده ایم.



همان طور که مشاهده می کنید یک گراف ممکن است چندین درخت پوشا داشته باشد . گراف فوق درخت های پوشای بیشتری از آن چه در بالا رسم کرده ایم دارد . وزن کلی درخت های فوق عبارت اند از :

1+4+5+6 = 16

1+2+3+4 = 10

1+3+5+6 = 15

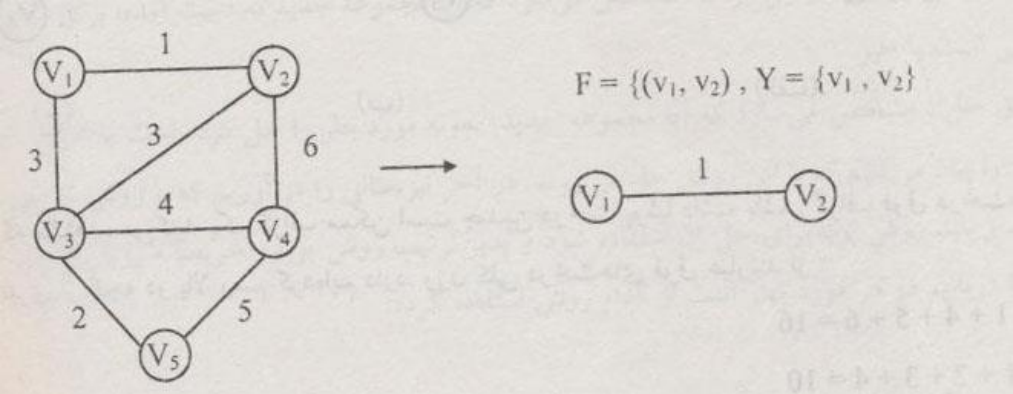
همان طور که می دانید یک گراف بدون جهت را می توان به صورت دو مجموعه V , E نشان داد : $G=(V , E)$که V مجموعه رئوس و E مجموعه یال ها می باشند . مثلاً برای گراف قبلی داریم :

$V = \{v_1, v_2, v_3, v_4, v_5\}$
$E = \{(v_1 , v_2) , (v_1, v_3), (v_2, v_3) , (v_2, v_4), (v_3, v_4), (v_3, v_5) , (v_4, v_5)\}$

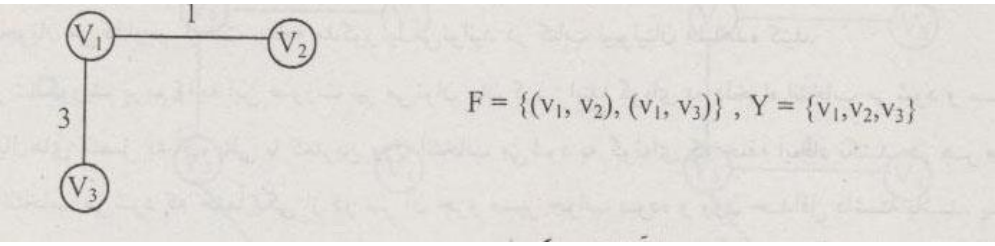
در گراف بدون جهت (V_1, V_2) با (V_2, V_1) یکسان است ولی قرارداد می کنیم که هنگام نوشتن ابتدا راسی که اندیس کوچک تر دارد بیاوریم . درخت پوشای T برای گراف $G= (V , E)$ شامل همه رئوس V می باشدولی برخی از یال های E را دارد که آنها را با F نمایش می دهیم . پس :

$T = (V , F) , F \in \dot{\subseteq} E$

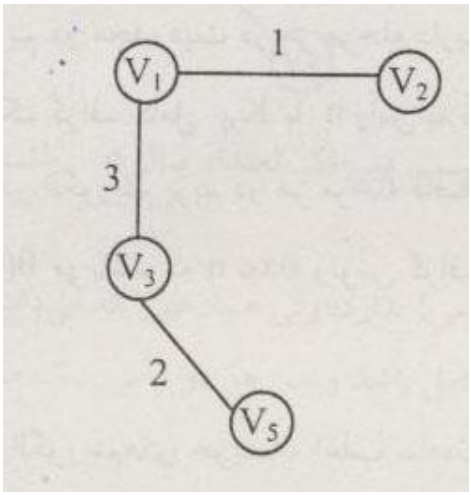
در الگوریتم پریم ابتدا مجموعه یال های F را برابر تهی و مجموعه رئوس Y را نیز برابر یک راس دلخواه قرار می دهیم.سپس نزدیک ترین راس به Y را از مجموعه V-Y انتخاب می کنیم. بدیهی است که این راس توسط یالی با کمترین وزن به راسی از مجموعه Y متصل است.مثلاً در گراف زیر V_2 را انتخاب کرده که کمترین یال (برابر ۱) را دارد . بدین ترتیب V_2 را به Y و (V_1, V_2) را به F اضافه می کنیم:



در قدم بعدی V_3 را به مجموعه Y و (V_1, V_3) را به مجموعه F اضافه می کنیم :

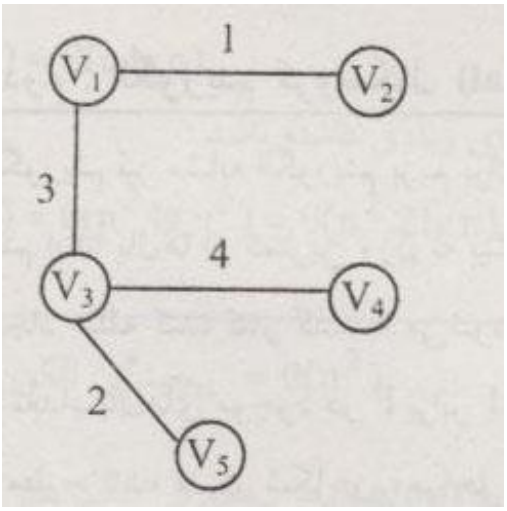


این عملیات افزودن نزدیک ترین رئوس را آن قدر تکرار می کنیم تا $Y=V$ شود. البته باید عدم ایجاد چرخه نیز بررسی می گردد :



در مرحله فوق فاصله $V-Y = \{V_4 , V_5\}$ را با گره های وجود در $Y=\{V_1, V_2, V_3\}$ مقایسه کردیم و دیدیم که فاصله V_5 با گره V_3 از همه کمتر است لذا V_5 را به Y و (V_3, V_5) را به F اضافه کردیم .

در مرحله آخر فاصله V_4 را با گره های موجود در $Y = \{V_1, V_2, V_3, V_5\}$ مقایسه کرده و گره V_4 را به Y و (V_3, V_4) را به F اضافه می کنیم و درخت پوشای کمینه به دست می آید.



الگوریتم فوق را به صورت زیر می توان بیان کرد :

```
F = ϕ
y = {v1} ;
while  (the instance is not solved)
{
   select a vertex in V - Y that is nearest to Y;   // selection procedure and feasibility check
   add the vertex to Y;
   add the edge to F;
   if  (Y == V)   the instance is solved;    //solution check
}
```

ا

لبته در هنگام انتخاب راس از V- Y باید دقت کرد که حلقه ایجاد نشود.

نکته : در یک گراف کامل K_n با n راس به تعداد n^{n-2} درخت پوشا وجود دارد .

از آنجا که در الگوریتم پریم در هر مرحل فاصله هر گره با گره های قبلی مقایسه می شود پس بدیهی است که از مرتبه $\theta(n^2)$ می باشد که n تعداد رئوس گراف است.

مرتبه اجرای الگوریتم پریم:$\theta(\mathrm{n}^2)$

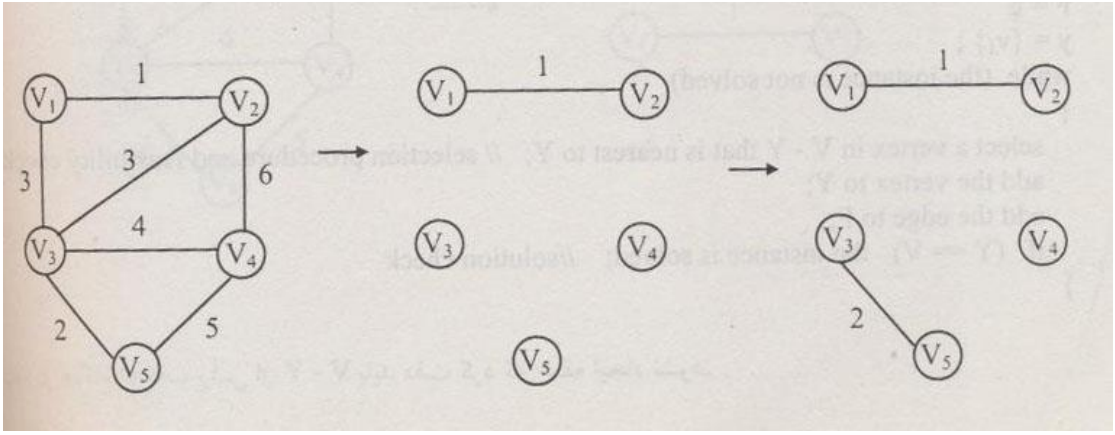
هر چند که الگوریتم حریصانه اغلب ساده تر از الکوریتم های پویا هستند ولی معمولاً مشخص کردن این که آیا این الگوریتم حریصانه همواره جواب بهینه را می دهد دشوار بوده و نیاز ه اثبات رسمی دارد .در کتاب نیپولیتان قضیه ای اثبات شده است که نشان می دهد الگوریتم پریم همواره یک درخت پوشای کمینه را تولید می کند.

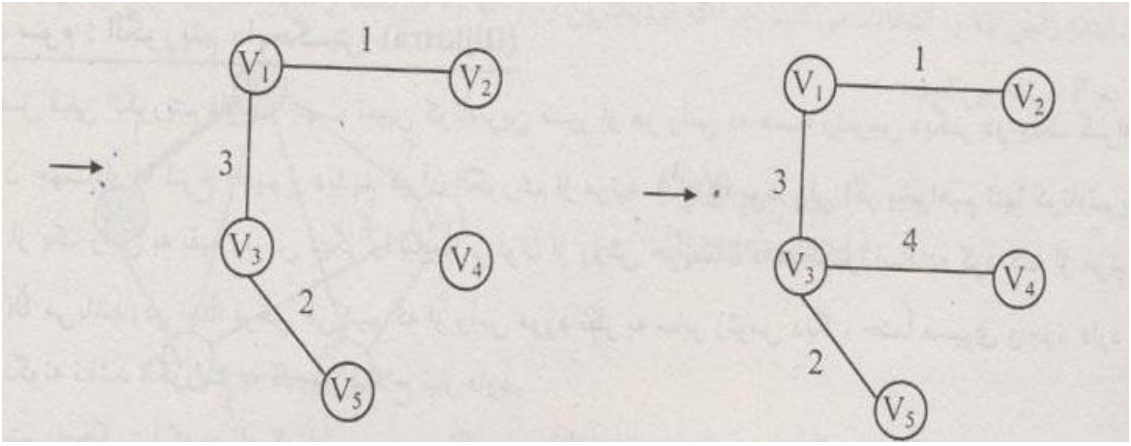
مثال دوم : الگوریتم کروسکال *(kruskal)*

این الگوریتم نیز مشابه الگوریتم پریم برای یافتن درخت پوشای کمینه یک گراف به کار می رود.در این الگوریتم ابتدا یال ها از کمترین وزن به بیشترین وزن مرتب می شوند سپس یال ها به ترتیب انتخاب شده و اگر یالی ایجاد حلقه کند کنار گذاشته می شود .

عملیات هنگامی خاتمه می یابد که تمام راس ها به هم وصل شوند یا این که تعداد یال های موجود در f برابر $n-1$ شود که تعداد راس ها است.

در برخی کتاب ها این روش با نام راشال مطرح شده است . شکل زیر مراحل کار برای یک گراف شخصی را نشان می دهد.





بیشتر زمان در الگوریتم کروسکال مربوط به مرتب سازی یال هاست پس اگر تعداد یال e باشد زمان این الگوریتم از مرتبه $\theta(\text{elge})$ خوهد بود.

ممکن است به نظر برسد که این زمان بستگس به n (تعداد رئوس) ندارد . ولی همان طور که می دانید در بهترین حالت تعداد یال ها برابر $n\text{-}1$ و در بدترین حالت که گراف کامل باشد و بین هر دو راس یک مسیر مستقیم داشته باشیم تعداد یال ها برابر $\frac{n(n-1)}{2}$ خواهد بود یعنی :

$$n-1 \le e \le \frac{n(n-1)}{2}$$

پس اگر e نزدیک به کران پایین باشد یعنی گراف نسبتاً خلوت بوده و یال کمی داشته باشد : الگوریتم کروسکال از مرتبه رو به رو است :

$$\theta(e \lg e) = \theta(n \lg n)$$

و اگر e نزدیک به کران بالا باشد یعنی گراف نسبتاً پر باشد و یال های زیادی داشته باشد :

$$\theta(e \lg e) = \theta(n^2 \lg n^2) = \theta(n^2 . 2 \lg n) = \theta(n^2 \lg n)$$

خلاصه آنکه

پیچیدگی الگوریتم پریم $= \theta(n^2)$

پیچیدگی الگوریتم کروسکال $= \theta(e \lg e) = \begin{cases} \theta(n \lg n) & \text{برای گراف خلوت} \\ \theta(n^2 \lg n) & \text{برای گراف شلوغ} \end{cases}$

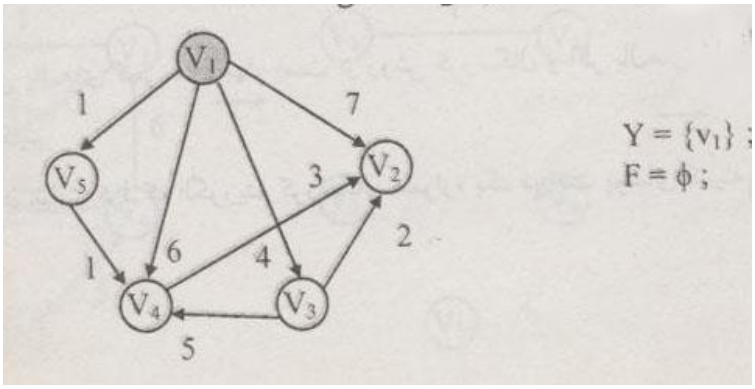
پس اگر گرافی یال های کمی دارد بهتر است از روش کروسکال و اگر یال های زیادی دارد بهتر است از روش پریم استفاده کنیم.

تذکر : می توان اثبات کرد که الگوریتم کروسکال همواره یک درخت پوشای کمینه ایجاد می کند.

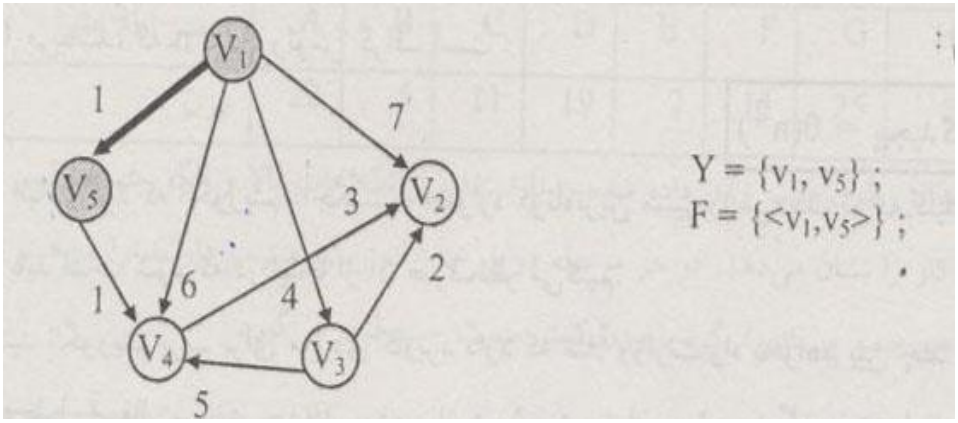
```
Y = {v1};
F = ϕ
while (the instance is not solved) {
      select a vertex v from V - Y, that has a // selection procedure
      shortest path from v1, using only vertices // and feasibility check
      in Y as intermediates;
      add the new vertex v to Y;
      add the edge (on the shortest path) that touches v to F;
      if (Y == V)
            the instance is solved; //solution check
```

شکل زیر مراحل الگوریتم فوق را برای یک گراف فرضی نشان می دهد :

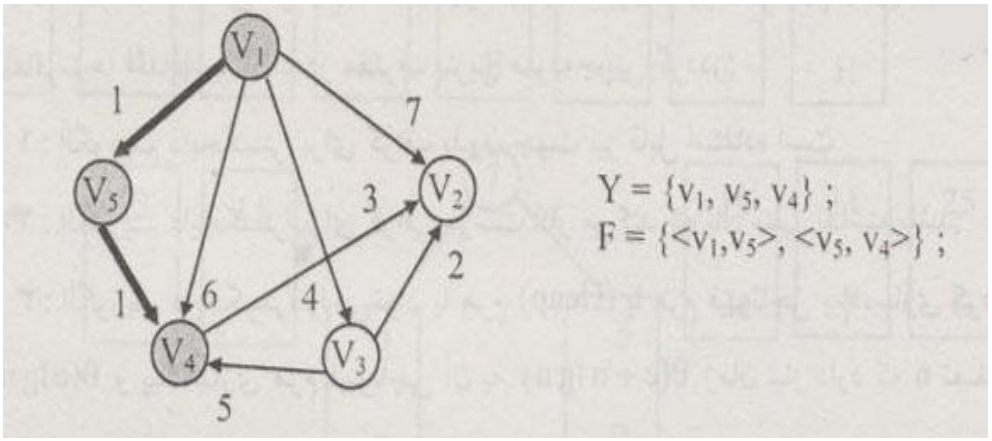
۱- محاسبه کوتاه ترین مسیر از V_1



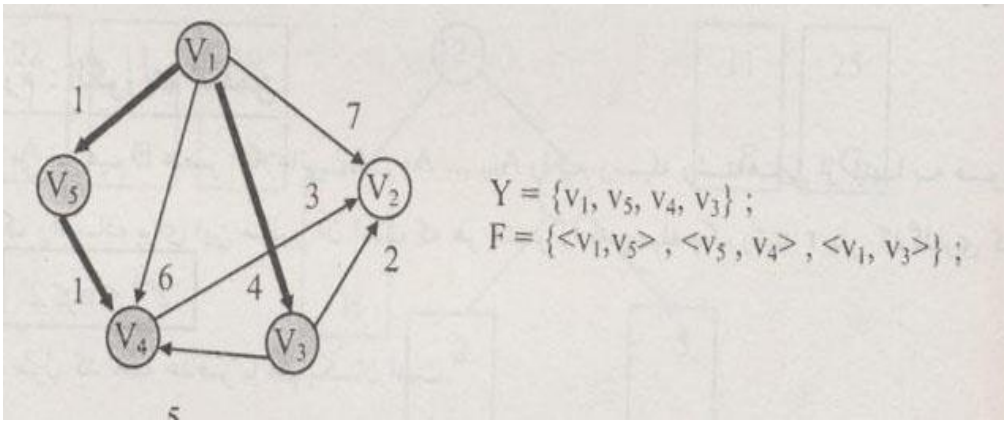
۲- ابتدا راس V_5 را انتخاب ی کنیم چرا که نزدیک ترین راس به V_1 می باشد و بدین دلیل V_5 را به Y و $< V_1, V_2 >$ را به F اضافه ی کنیم.



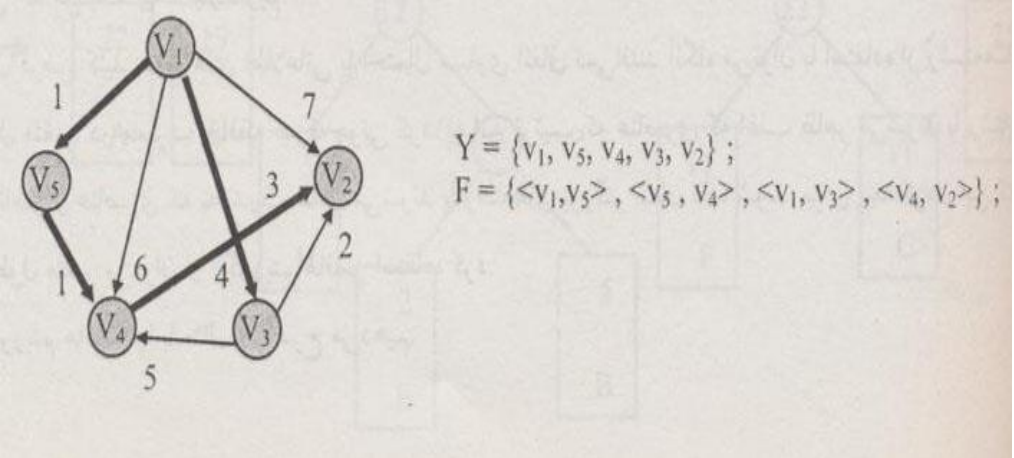
۳- حال با در نظر گرفتن گره V_5 به عنوان واسطه گره V_4 را انتخاب می کنیم چرا که V_4 کوتاه ترین مسیر تا V_1 را (به کمک گره V_5) دارد.



۴- راس V_5 را انتخاب می کنیم چرا که با واسطه گری $\{V_4, V_5\}$ کوتاه ترین مسیر از V_1 را دارد.



۵- در آخر نیز V_2 را انتخاب می کنیم و می بینیم که کوتاه ترین مسیر از V_1 به V_2 مسیر $[V_1, V_5, V_4, V_2]$ است .لذا V_2 را به Y و آخرین یال مسیر مذکور یعنی $< V_4, V_2 >$ را به F اضافه می کنیم.



مثال چهارم : الگوریتم هافمن :

فرض می کنید می خواهید n عنصر اطلاعاتی $A_n, \ldots . A_2, A_1$ را به وسیله رشته هایی از بیت ها به صورت کد درآوریم . یک راه ساده برای این منظور آن است که هر عنصر را به وسیله یک رشته r بیتی کد گذاری کنیم که در آن

$$2^{r-1} < n < 2^r$$

در این حالت طول کد همه عناصر با هم یکسان است.

مثال : برای کد گذاری ۴۸ کاراکتر حداقل به چند بیت نیاز داریم ؟

$$32 = 2^5 < 48 \leq 2^6 = 64 \rightarrow r = 6$$

پس حداقل به ۶ بیت نیاز داریم .

حال فرض کنید عنصر های اطلاعاتی با احتمال مساوی اتفاق نمی افتد آنگاه می توان با استفاده از رشته های با طول متغیر در مصرف حافظه صرفه جویی کرد .

به این ترتیب که عناصری که اغلب ظاهر می شوند با رشته های کوتاهتر و عناصری که به ندرت ظاهر می شوند با رشته های بزرگ تر نشان داده شوند.برای پیدا کردن این کدهای با طول متغیر می توان از الگوریتم های هافمن استفاده کرد.

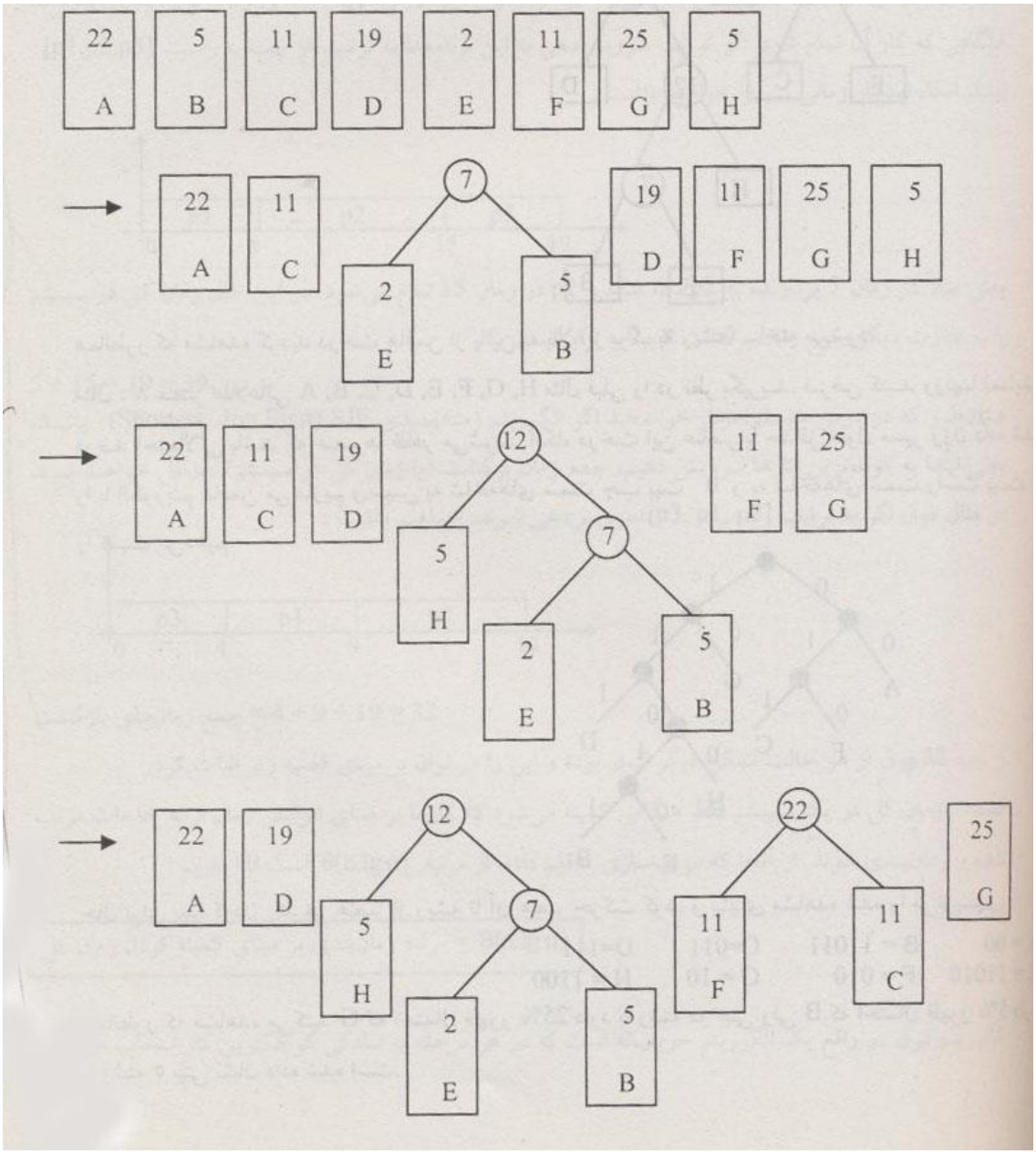
الگوریتم هافمن را با مثال زیر شرح می دهیم.

فرض کنید *A,H,G,F,E,D,C,B,A* عنصر اطلاعاتی با وزن های مشخص شده هستند.

عنصر اطلاعاتی	A	B	C	D	E	F	G	H
وزن	22	5	11	19	2	11	25	5

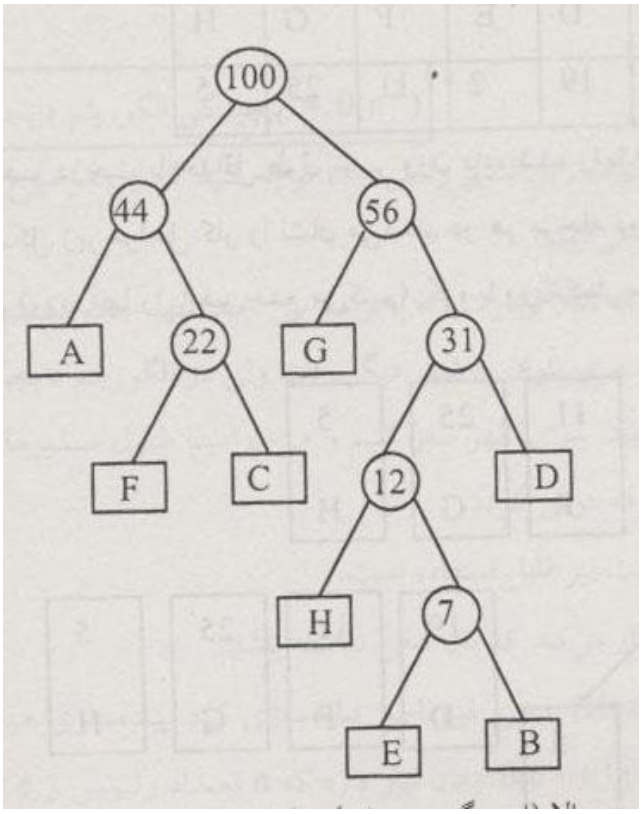
می خواهیم درخت با حداقل طول مسیر وزن داده شده را با استفاده از اطلاعات بالا و الگوریتم هافمن ترسیم کنیم.

شکل زیر مراحل کار را نشان می دهد . در هر مرحله دو درختی که ریشه مینیمم دارند را با هم ترکیب می کنیم (وزن آن ها را با هم جمع می کنیم) . گره با وزن کمتر سمت چپ قرار می گیرد.



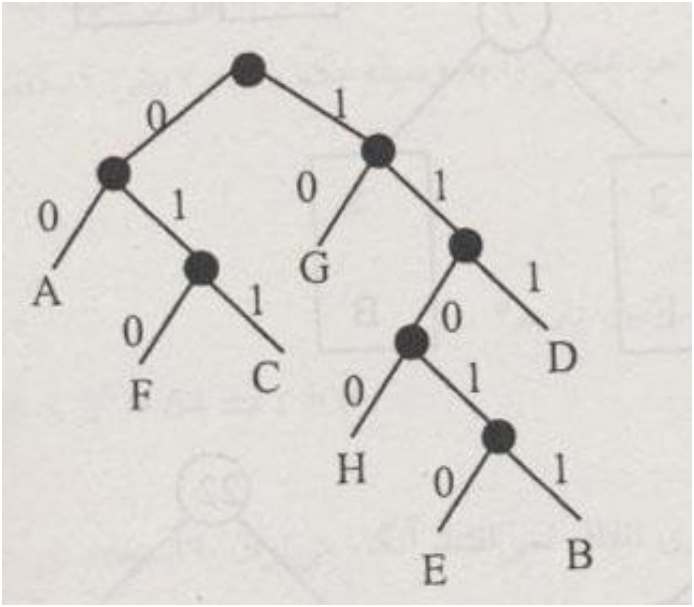
اگر همین طور الگوریتم را ادامه دهید شکل نهایی به صورت زیر می شود :

درخت هافمن



همان طور که مشاهده کردید درخت عافمن از پایین به بالا (از برگ ه ریشه) ساخته می شود.

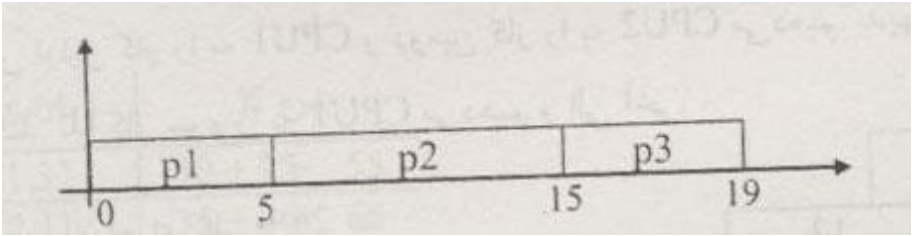
مثال ۸ : عنصر اطلاعاتی *H, G, F, E , D, C ,B, A* مثال قبلی را در نظر بگیرید . فرض کنید وزن ها نمایش درصد احتمالی باشند که عنصرها ظاهر می شوند .آنگاه درخت این عناصر با حداقل طول مسیر وزن داده شده را با الگوریتم هافمن می سازیم و سپس به شاخه های سمت چپ بیت *0* و به شاخه های سمت راست *1* را نسبت می دهیم.



حال برای پیدا کردن کد هر عنصر از ریشه تا آن عنصر حرکت کرده و بیت های مشاهده شده را می نویسیم:

A = 00	B = 11011	C=011	D=111
E = 11010	F = 010	G = 10	H = 1100

همان طور که مشاهده می کنید *G* که احتمال ظهور *25%* دارد با رشته دو بیتی ولی *B* که احتمال ظهور *5%* دارد با رشته ۵ بیتی نشان داده شده است.

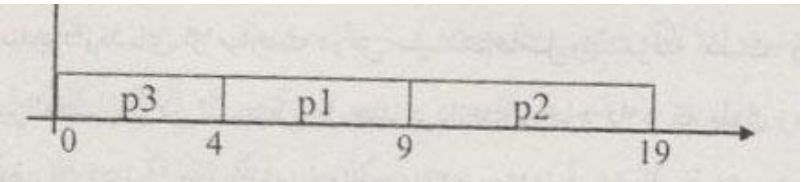


یعنی مثلاً در زمان ۵ پردازنده به P^2 داده شده و P^2 در زمان *15* تمام می شود . در این حالت زمان کل در سیستم یا به عبارت دیگر جمع زمان برگشت ۳ برنامه برابر است با :

5 + 15 +19 = 39

همان طور که سیستم عامل خوانده اید اگر عامل زمانبندی ***(Shortest job First) SJF*** باشد , یعنی ابتدا به کوتاه ترین کارها سرویس دهیم , جمع زمان برگشت (یا زمان کل در سیستم) حداقل خواهد بود .

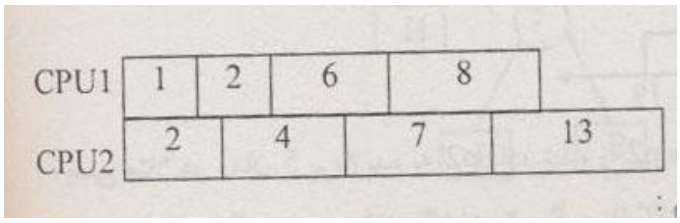
در مثال فوق اگر به ترتیب $[P^3,P^1,P^2]$ سرویس دهی شوند , خواهیم داشت :



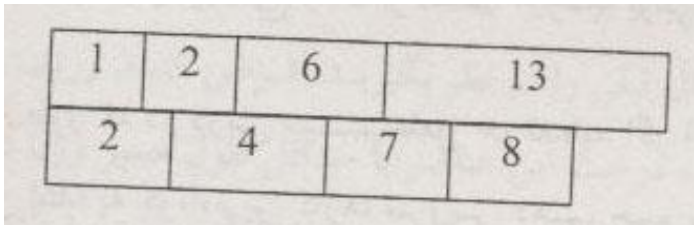
ابتدا برنامه ها را به صورت صعودی مرتب می کنیم :

{1,2,2,4,7,8,13}

سپس اولین کار را به ***CPU1*** و دومین کار را به ***CPU2*** می دهیم.بدیهی است که ***CPU1*** کارش را زودتر تمام می کند . لذا کار سوم را به ***CPU1*** می دهیم و الی آخر .



ولی اگر ترتیب ۸ و ۱۳ را در مرحله آخر عوض کنیم :



مثال : چهار کار زیر که همگی طول اجرای ۱ داشته با بهره و مهلت داده شده را در نظر بگیرید :

کار	مهلت	بهره	زمان اجرا
1	2	30	1
2	1	35	1
3	2	25	1
4	1	40	1

اینکه مهلت کار ۳ برابر ۲ است یعنی اگر کا ۳ در زمان ۱ یا زمان ۲ شروع شود , بهره ۲۵ معادل آن به دست می آید و اگر بخواهد در زمان ۳ یا بیشتر آغاز شود غیر قابل قبول بوده و بهره ای را نمی دهد در مثال فوق فرض می کنیم زمان صفر نداریم و شروع زمان ۱ می باشد .

برای مثال فوق انواع زمانبندی های ممکن و نیز بهره های به دست آمده عبارت اند از

زمان‌بندی	بهره کل
[1,3]	30 + 25 = 55
[2,1]	35 + 30 = 65
[2,3]	35+25=60
[3,1]	25 + 30 = 55
[4,1]	40 + 30 = 70
[4,3]	40 + 25 = 65

در جدول فوق زمانبندی های غیرممکن نوشته نشده اند . مثلاض زمانبندی *[1, 4]* امکان پذیر نیست زیرا اگر کار ۱ اجرا شود دیگر مهلت کار ۴ تمام شده و نمی تواند اجرا گردد.هدف از این مثال به دست آوردن زمانبندی *[4,1]* با بالاتریت بهره کل ۷۰ است.

اگر بخواهیم همه زمانبندی های ممکن را در نظر بگیریم الگوریتمی ار مرتبه ی فاکتوریل خواهد بود که خیلی زمان بر است لذا الگوریتمی حریصانه و بسیار سریع تر معرفی می کنیم.

مثال : بهترین زمان بندی برای جدول زیر را جهت به دست آوردن حداکثر بهره بیان کنید.

کار	مهلت	بهره
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

حل : جدول فوق بر اساس بهره های نزولی مرتب شده است.

۱ـ ابتدا S برابر ϕ می‌شود.

۲ـ S برابر {1} شده و بررسی می‌گردد که ترتیب [1] امکان‌پذیر است.

۳ـ S برابر {1,2} شده و قبول می‌شود چرا که ترتیب [2,1] امکان‌پذیر می‌باشد.

٤ـ {1,2,3} رد می‌شود چرا که ترتیب [3 , 2 , 1] با مهلت‌هایی که زیر آن با فلش رسم کرده‌ایم، امکان‌پذیر نمی‌باشد. پس هیچ ترتیب دیگری از {1,2,3} نیز امکان‌پذیر نمی‌باشد.

3 1 1
↑ ↑ ↑

٥ـ S را برابر {1,2,4} قرار می‌دهیم و این مجموعه امکان‌پذیر است چرا که ترتیب [2 , 1 , 4] امکان‌پذیر است.

3 3 1
↑ ↑ ↑

٦ـ {1,2,4,5} رد می‌شود چرا که ترتیب [2 , 5 , 1 , 4] امکان‌پذیر نیست.

3 3 1 1
↑ ↑ ↑ ↑

۷ـ {1,2,4,6} رد می‌شود چرا که ترتیب [2 , 1 , 4 , 6] امکان‌پذیر نیست.

3 3 3 1
↑ ↑ ↑ ↑

۸ـ {1,2,4,7} رد می‌شود چرا که ترتیب [2 , 7 , 1 , 4] امکان‌پذیر نیست (کار 4 در مهلت 3 تمام نمی‌شود.)

قضیه : الگوریتم حریصانه ای که در بالا شرح دادیم , همواره یک مجموعه بهینه را تولید می کند . اثبات قضیه فوق الذکر در کتاب نیپولیتان آمده است.

کارهایی که در الگوریتم فوق صورت می گیرد یکی مرتب کردن n کار بر اساس بهره است که به زمان θnlgn نیاز دارد.

دیگری بررسی امکان پذیر بودن مجموعه پدید آمده در همه مرحله ها که از مرتبه $\theta(n^2)$ بوده و بر طمان $nlgn$ برتری دارد لذا

زمان الگوریتم حریصانه زمان‌بندی با مهلت معین $= \theta(n^2)$

مثال هشتم : مساله کلاسیک کوله پشتی *(Knapsack)*

دزدی با کوله پشتی خود وارد جواهر فروشی می شود . کوله پشتی او تحمل حداکثر وزن W را دارد و بیشتر از آن را پاره می شود.

هر قطعه جواهر وزن (W_i) و ارزش (P_i) معینی دارد. مساله مهم دزد انتخاب قطعاتی است که حداکثر ارزش را داشته باشد و نیز جمع وزن آنها از حداکثر W بیشتر نشود و این مساله کلاسیک کوله پشتی است.

یک راه حل ساده و غیر هوشمندانه آن است که همه زیر مجموعه های ممکن از n قطعه را در نطر بگیریم و با مقایسه کردن همه آنها حالتی را در نظر بگیریم که بیشترین ارزش را داشته و جمع وزن آن ها W باشد . از آنجا که تعداد این زیر مجموعه ها 2^n است لذا این الگوریتم از مرتبه نمایی است و ما دنبال راه حل سریع تری هستیم.

ممکن است به نظر برسد یک راه حل حریصانه آن است که قطعاتی با ارزش بیشتر را زودتر برداریم تا هنگامی که جمع وزن آن ها به W برسد . ولی با مثال های ساده ای می توان نشان داد که این روش غلط است .مثلاً سه قطعه با وزن و ارزش های زیر را در نظر بگیرید :

قطعه	1	2	3
وزن قطعه (کیلوگرم) = w_i	25	10	10
ارزش قطعه (میلیون تومان) = p_i	10	9	9

فرض کنید حداکثر $W=30$ کیلوگرم باشد . در این حالت دزد فقط قطعه اول را برمی دارد که ۱۰ میلیون تومان به دست می آورد . در حالی که اگر قطعات دوم و سوم را برمی داشت صاحب ۱۸ میلیون تومان می شد.

یک راه حل حریصانه دیگر آن است که قطعات را به ترتیب افزایش ارزش آنها به ازای واحد وزن مرتب کرده و سپس آنها را به ترتیب انتخاب می کنیم.

برای جدول زیر و برای حداکثر $W=30$

قطعه	اول	دوم	سوم
w_i = وزن قطعه	5	10	20
p_i = ارزش قطعه	50	60	140
$\frac{p_i}{w_i}$ نسبت	$\frac{50}{5}=10$	$\frac{60}{10}=6$	$\frac{140}{20}=7$

فصل ششم

تکنیک عقبگرد

مفهوم تکنیک عقبگرد *(Backtracking)*

فرض کنید در غار علی صدر قایقرانی می کنید و در ۱۰۰۰ توی این غار هنگامی که به بن بست می رسید باید از همان راهی که آمده اید بازگردید و بر سر دوراهی ها مسیر خود را عوض کنید . حال اگر بر سر برخی دو راهی ها تابلویی وجود داشته باشد که به شما بگوید انتهای این مسیر بن بست است آنگاه صرفه جویی زیادی در وقت خوذ خواهید کرد.

مثلا در مورد مسله کوله پشتی *0, 1* , یک راه غیر هوشمندانه تولید تمام 2^n زیرمجموعه برای *n* قطعه بود که برای *n* های بزرگ بسیار زمان بر است. ولی اگر هنگام تولید این زیرمجموعه ها بتوانیم از علائمی استفاده کنیم که به ما بگویید تولید بسیاری از این زیر مجموعه ها لازم نیست , آنگاه در وقت صرفه جویی زیادی می کنید.

این تکنیک همان الگوریتم عقبگرد , پس گرد , پی جویی به عقب یا *Backtracking* است. الگوریتم های عقبگرد اغلب برای بسیاری از نمونه های بزرگ موثر می باشد اما نه برای همه نمونه ها.

روش عقبگرد برای حل مسائلی به کار می رود که در آن دنباله ای از اشیا از یک مجموعه مشخص انتخاب می شود. به گونه ای که این دنباله ملاک و قانونی خاصی رعایت کند.

مشهورترین مساله کلاسیک برای این تکنیک , مساله *n*وزیر *(n-queen problem)* است.در این مساله می خواهیم *n* وزیر را در یک صفحه شطرنج *n*n* به گونه ای قرار دهیم که هیچ وزیری نتواند دیگری را تهدید کند . یعنی هیچ دئ مهره ای نباید در یک سطر , ستون یا قطر یکسان قرار گیرند.

در این مسئله *n* وزیر منظور از دنباله *n* مکانی است که وزیرها قرار می گیرند و مجموعه n^2 موقعیت در صفحه شطرنج و ملاک و قانون *(criterion)* آن است که هیچ دو وزیری همدیگر را تهدید نکنند.

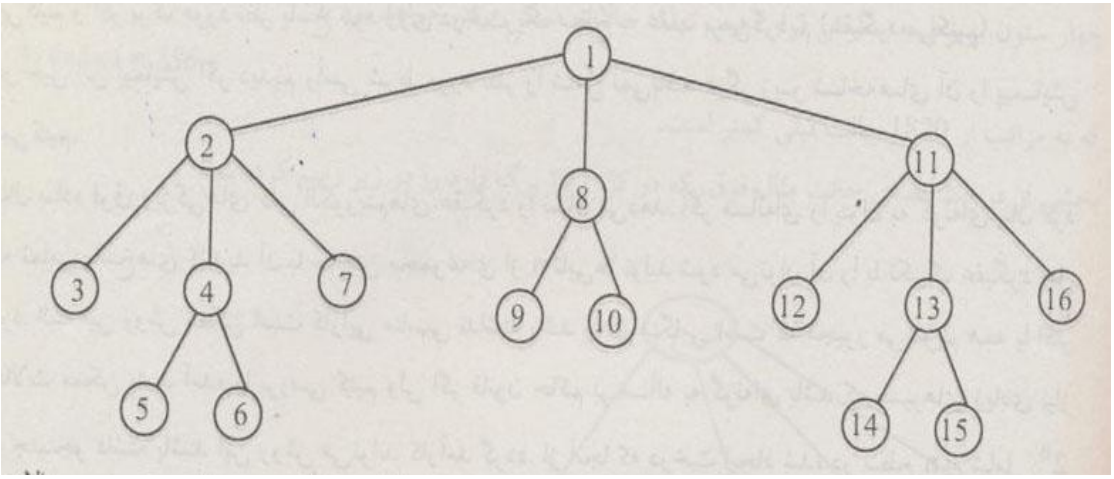
تکنیک عقبگرد در واقع حالت اصلاح شده ای از جستجوی عمقی یک درخت است.بحث جستجو و پیمایش درخت ها را به طور مفصل در درس ساختمان داده ها خوانده ایم. در اینجا جهت یادآوری پیمایش

preorder یک درخت (که نوعی پیمایش عمقی است) را یادآوری می کنیم.در این پیمایش ابتدا ریشه و سپس فرزندان آن (عموما به ترتیب از چپ به راست) پیمایش می شوند.

تابع بازگشتی زیر این پیمایش را انجام می دهد.

```
void dfs(node v)
{
  node u;
  visit v;
  for (each child u of v)
      dfs (u);
}
```

مثلاً در درخت شکل زیر گره ها به ترتیب پیمایش عمقی ***preorder*** شماره گذاری شده اند :



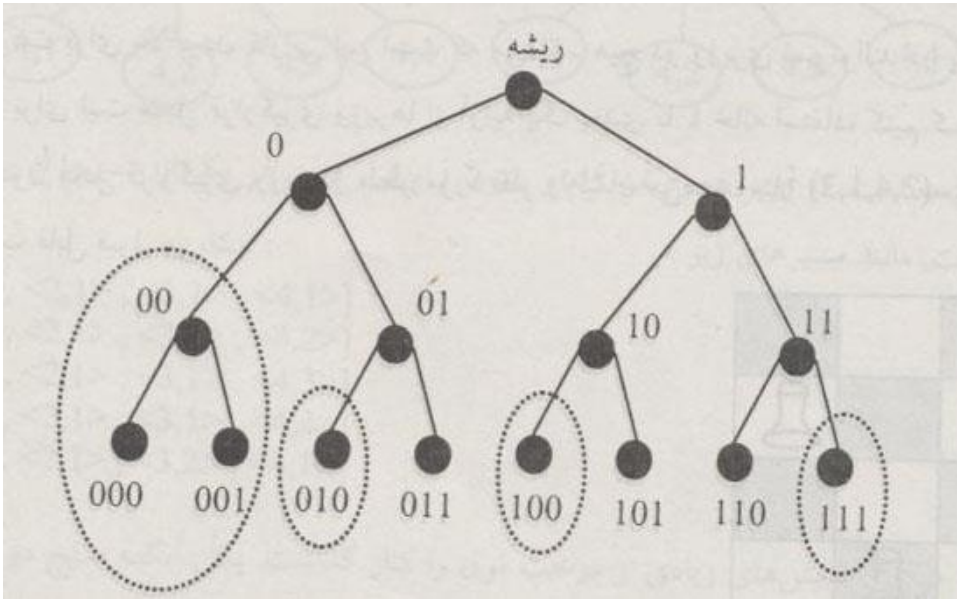
نکته مهم در جستجوی فوق آن است که اغلب از جستجوی همه حالات ممکن خودداری شده و حالات و شاخه هایی که بنا به دلایلی منجر به پاسخ مورد نظر نمی شوند , حذف می گردند.

مثال اول : قفل رمزی

یک قفل رمزی شامل n کلید دیجیتالی است که هر کلید فقط دو حالت بسته *(1)* و یا باز *(0)* دارد. می دانیم که n کلید دیجیتالی تعداد 2^n حالت مختلف را پدید می آورند و این قفل تنها با یک حالت که همان رمز است باز می شود.

فرض کنید در یک مساله ساده $n = 3$ باشد و ما رمز را فراموش کرده باشیم. در حالت معمولی باید ۸ حالت ممکن را بررسی کنیم ولی مثلاً اگر یادمان باشد که این رمز شامل دو رقم *1* بوده می توان سریع تر به جواب رسید .

درخت زیر که به درخت فضای حالت *(state space tree)* معروف است تمام حالت های مختلف را نشان می دهد:



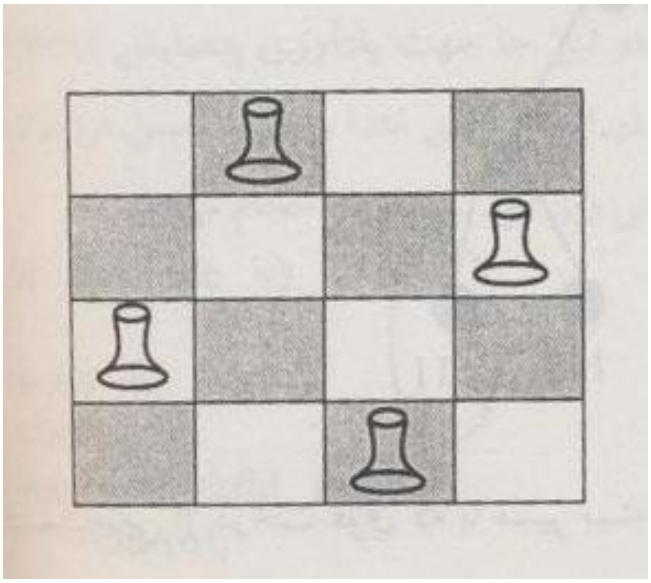
قسمت هایی از درخت فوق با مسیر بسته نقطه چین مشخص ساخته ایم را بررسی نمی کنیم و بدین ترتیب قلمرو و فضای پاسخ را کوچک تر و محدودتر می کنیم.

مساله دوم : مساله n وزیر

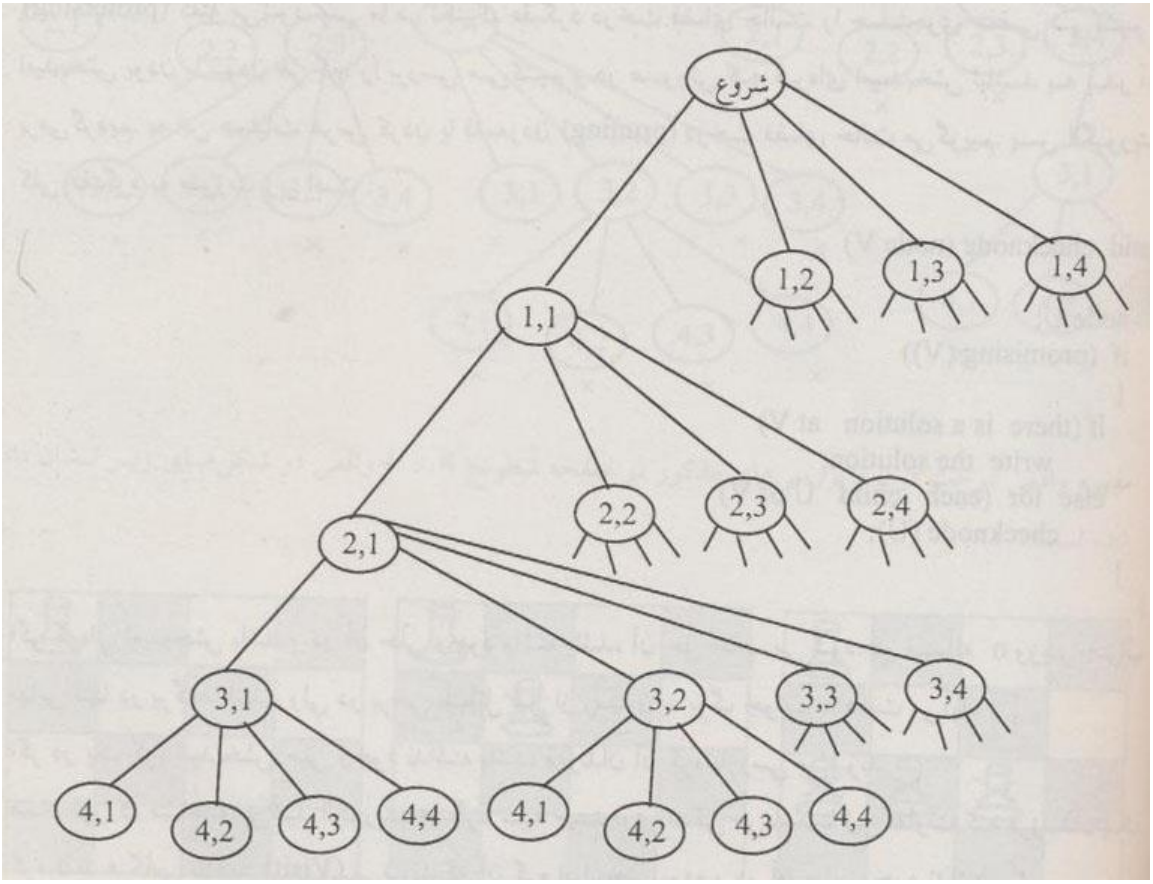
همان طور که قبلاً گفتیم در این مسئله می خواهیم تمامی حالاتی را پیدا کنیم که بتوان n وزیر را در صفحه شطرنج $n*n$ قرار دارد به نحوی که هیچ دو وزیری یکدیگر را تهدید نکنند . می دانید که صفحه شطرنج استاندارد $8*8$ می باشد. در ابتدا جهت سادگی n را برابر 4 در نظر می گیریم.

یک راه حل روشن ولی غیر هوشمندانه آن است که مثلاً برای حالت $n=4$ تمامی حالات ممکن چیدن مهره ها را آزمایش کنیم . ولی تعداد این حالات $\binom{16}{4}$ بوده که ۱۸۲۰ می شود.

اولین حرکت برای بالا بردن کارایی این است که می دانیم هیچ دو وزیری نمی توانند در یک سطر قرار بگیرند . لذا می توانیم برای ثبت محل قرارگیری وزیرها از آرایه یک بعدب با ۴ خانه استفاده کنیم که محتویات هر خانه شماره ستون محل قرار گیری وزیر در سطر مورد نظر را نشان می دهد.مثلاً $(2,4,1,3)$ متناظر شکل زیر بوده و یک جواب قابل قبول می باشد :



بخشی از درخت فضای حالت مثال فوق , که در کل ۲۵۶ برگ دارد را در زیر رسم کرده ایم :

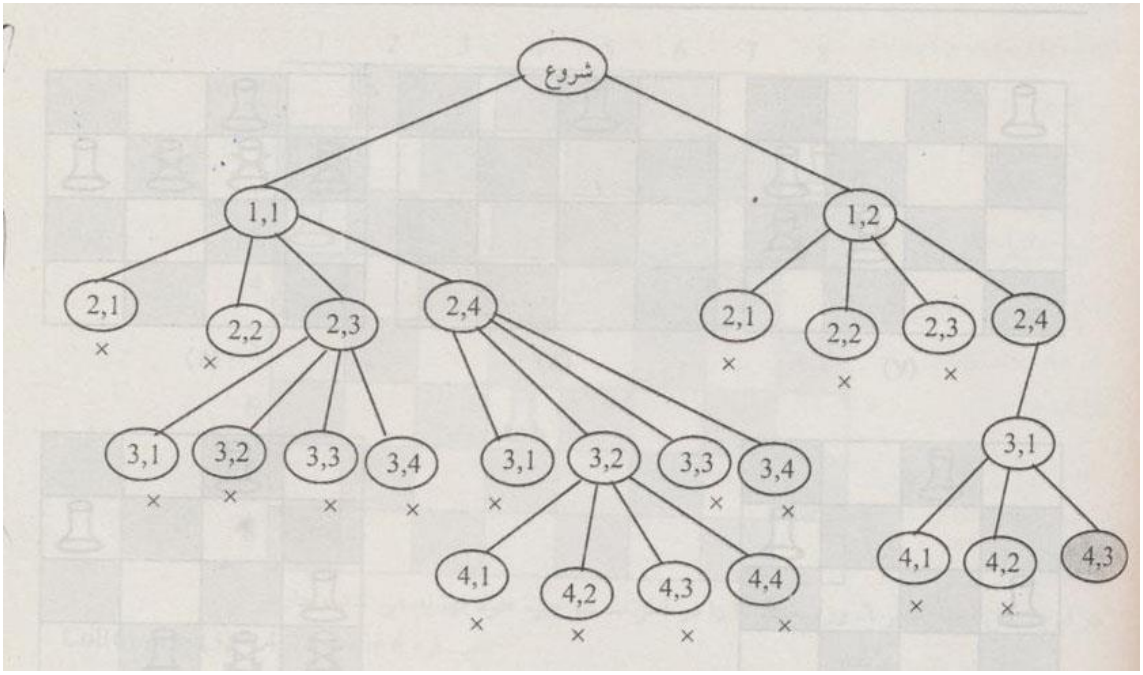


زوج مرتب $<i , j>$ به این معناست که وزیر در ردیف i ام و ستون j ام قراردارد .هر یک از مسیرها از ریشه به برگ یک حل کاندید است مانند مسیرهای زیر :

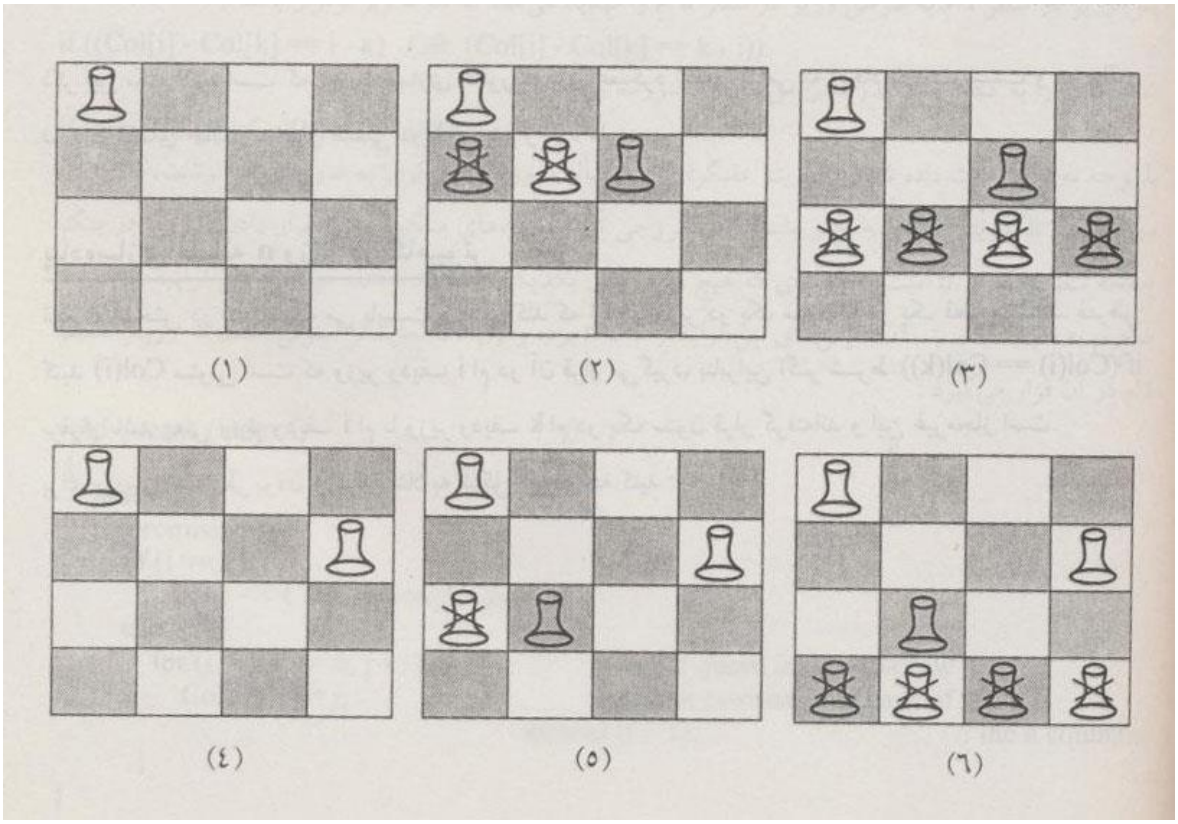
[<1,1>, <2,1> , <3,1> , <4,1>]
[<1,1>, <2,1> , <3,1> , <4,2>]
[<1,1>, <2,1> , <3,1> , <4,3>]
[<1,1>, <2,1> , <3,1> , <4,4>]
[<1,1>, <2,1> , <3,2> , <4,1>]

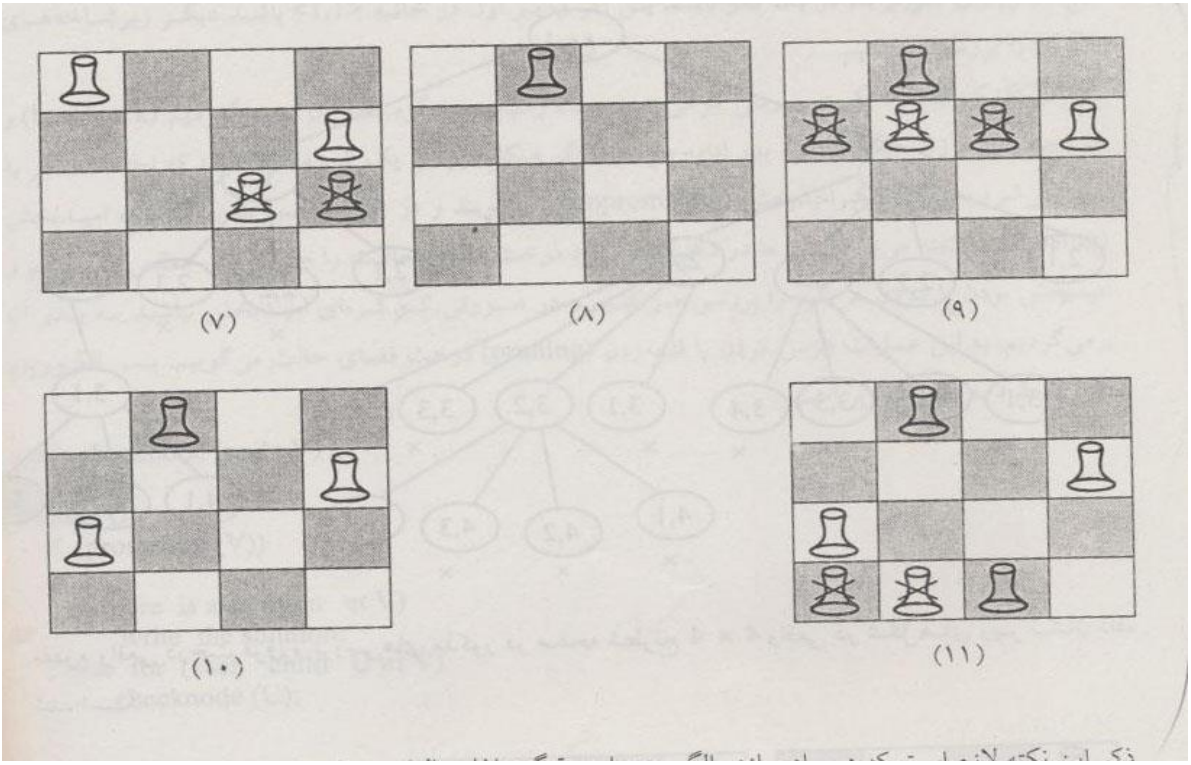
ولی با توجه به دو نکته می توان بخش های زیادی از درخت فوق را کنار گذاشت.یکی آنکه هیچ دو وزیری نمی توانند در یک ستون باشند لذا مثلاً در درخت فوق زیر شاخه های < *2 ,1*> را بررسی نمی کنیم . دوم آنکه هیچ دو وزیری نمی توانند در یک قطر باشند پس اگر وزیر اول در خانه < *1,1*> باشد دیگر زیر شاخه های <*2 ,2*> را بررسی نمی کنیم.

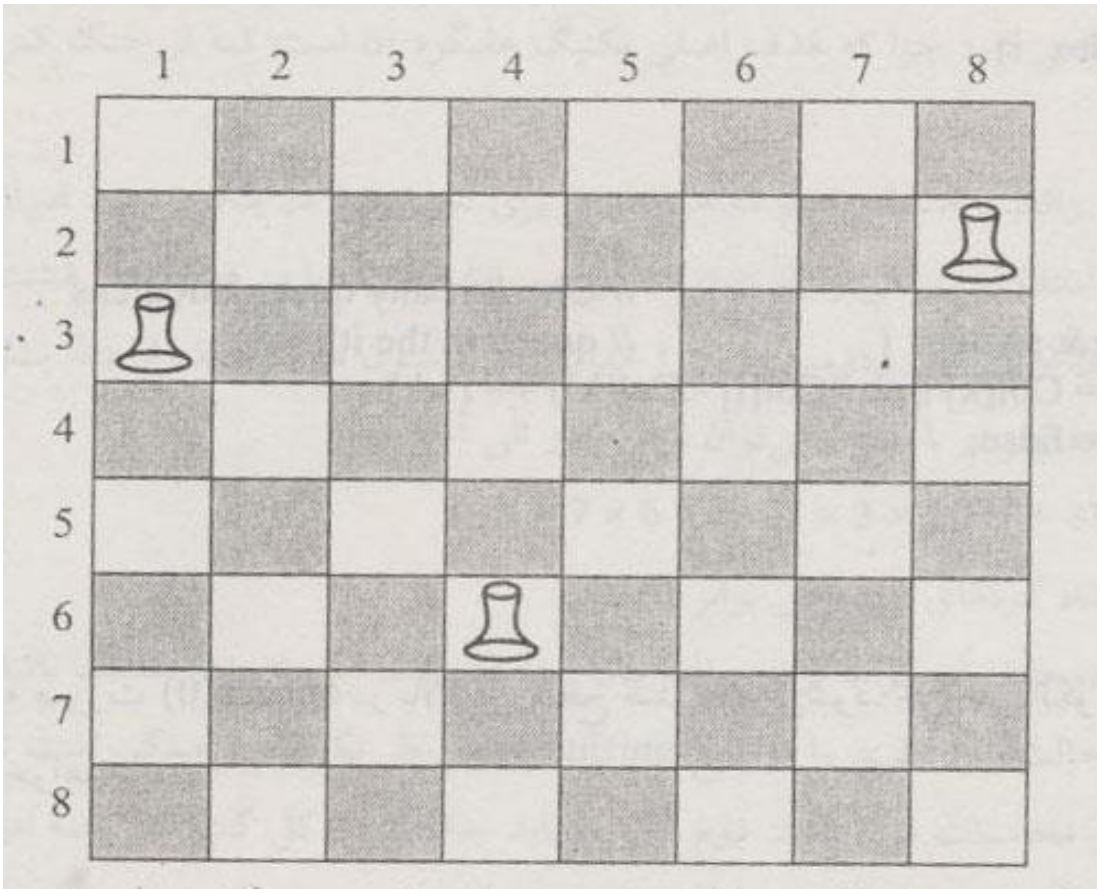
```
void checknode (node V)
{
  node U;
  if (promising (V))
  {
    if (there is a solution at V)
        write the solution;
    else for (each child U of V)
        checknode (U);
  }
}
```



مفهوم واقعی درخت فوق و بررسی های مذکور در صفحه شطرنج *4*4* واقعی در شکل های زیر نشان داده شده است.







در این شکل وزیر سطر ۶ , سطر ۳ را در قطر سمت چپ خود تهدید می کند و داریم :

Col (6) – Col(3) = 4-1 = 3 = 6 -3

همچنین وزیر سطر ۶ , وزیر سطر ۲ را در قطر سمت راست خود تهدید می کند و داریم :

Col(6) – col(2) = 4-8 = -4 = 2-6

پس وزیر در سطر ***i*** ام به شرطی وزیر در سطر ***k*** ام را تهدید می کند که شرط زیر برقرار باشد .

```
if ((Col[i] - Col[k] == i - k)  OR  (Col[i] - Col[k] == k - i))
```

شرط فوق را به کمک تابع قدر مطلق می توان به صورت زیر نوشت :

با توجه به توضیحات داده شده الگوریتم عقبگرد برای مساله n وزیر را می توان به صورت زیر نوشت :

ورودی این الگوریتم عدد صحیح و مثبت n و خروجی آن همه راه های ممکن جهت قرار دادن n وزیر در یک صفحه شطرنج $n*n$ است به گونه ای که هیچ دو وزیری یکدیگر را تهدید نکنند.

هر خروجی آرایه از اعداد صحیح به نام col است که اندیس های این آرایه از 1 تا n بوده و $col[i]$ نمایانگر ستونی است که وزیر سطر i ام در آن قرار می گیرد.

```
void queens (index i)
{
  index  j ;
  if (promising (i))
     if (i == n)
        cout  << Col[1] through Col[n];
     else
       for (j = 1; j <= n; j ++) {          // See if queen in (i + 1)st row
         Col[i + 1] = j;                    // can be positioned in each of
                                    queens (i + 1);                  // the n columns.
       }
}
```

```
bool  promising (index  i)
{
   index  k ;
   bool switch ;
   k = 1;
   switch = true;                                  // Check if any queen threatens
      while  (k < i && switch) {                   // queen in the ith row.
         if (Col[i] == Col[k] || abs(Col[i] - Col[k]) == i - k)
            switch = false;
         k++;
      }
return switch;
}
```

الگوریتم فوق به صورت $queens(0)$ در بالاترین سطح صدا زده می شود . خروجی الگوریتم فوق برای $n=4$ به صورت زیر خواهد بود :

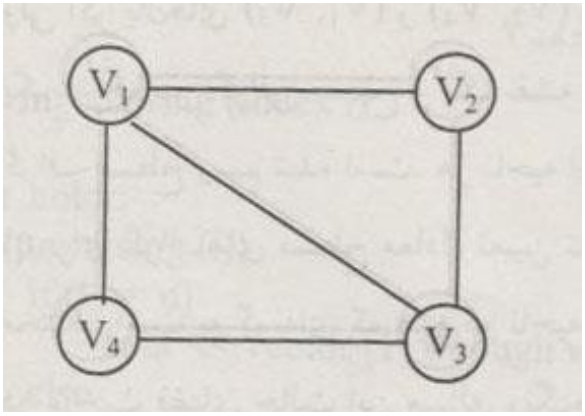
۳ ۱ ۴ ۲

۲ ۴ ۱ ۳

مثال سوم : رنگ آمیزی گراف :

در این مساله می خواهیم تمام راه های ممکن جهت رنگ آمیزی گره های یک گراف بدون جهت را با استفاده از حداکثر m رنگ متفاوت پیدا کنیم به گونه ای که هیچ دو راس مجاوری هم رنگ نباشند . توجه کنید هدف رنگ کردن داخل گره ها است و نه فضای بین یال ها.

مثال : در گراف شکل زیر :



نمی توان با دو رنگ متفاوت گراف را به گونه ای رنگ آمیزی کرد که رئوس مجاور هم رنگ نباشند.ولی با ۳ رنگ متفاوت این کار امکان پذیر است و یکی از راه های مکن عبارت است از :

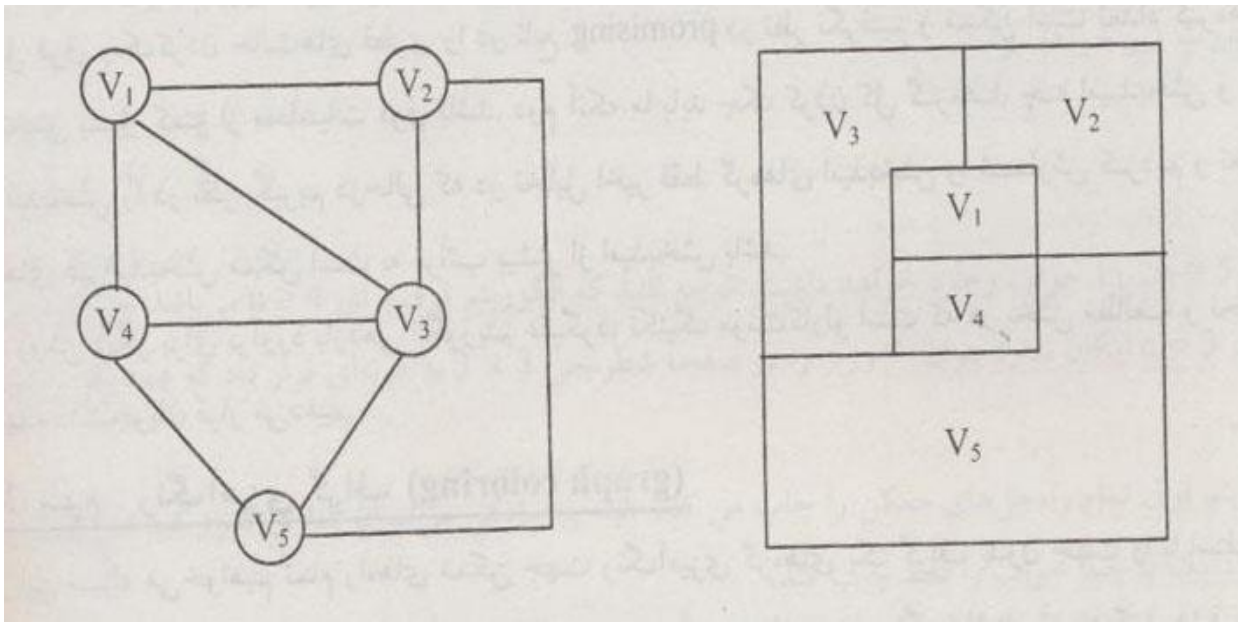
رأس	V_1	V_2	V_3	V_4
رنگ	رنگ ۱	رنگ ۲	رنگ ۳	رنگ ۲

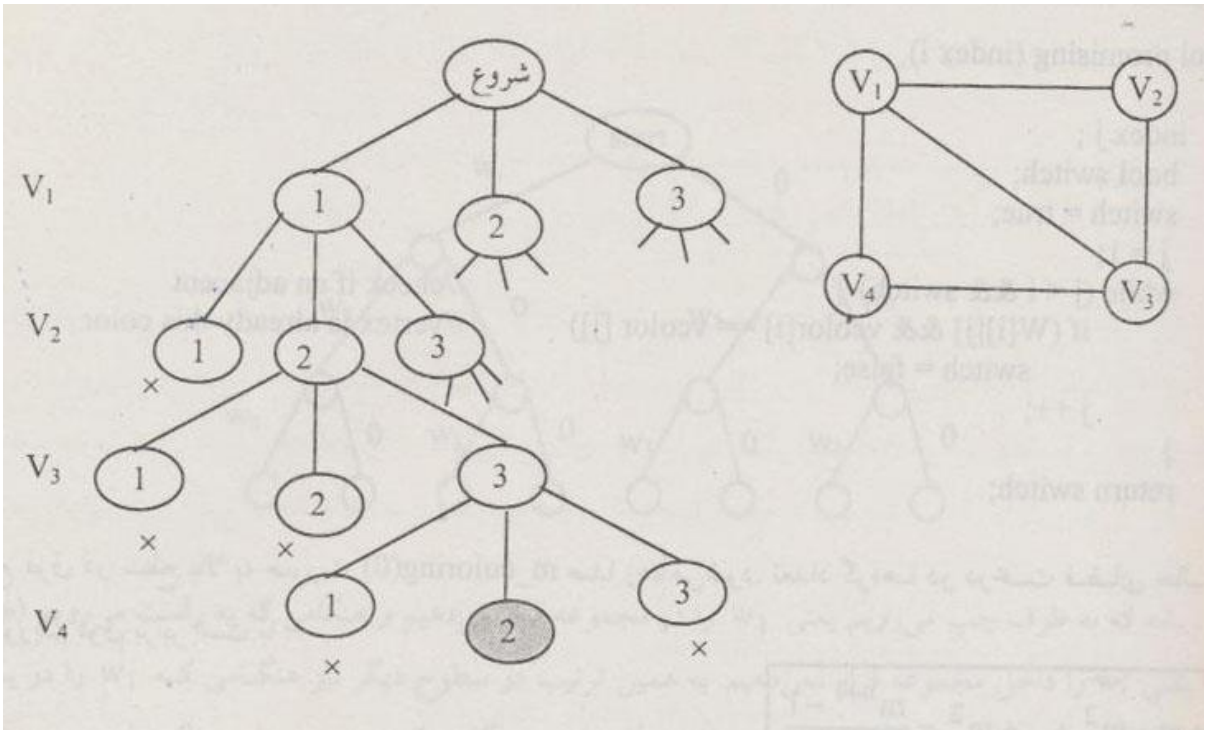
البته برای حل مسئله فوق ۶ حل وجود دارد که فقط در شیوه جابه جایی رنگ ها با هم فرق دارند.مثلاً یک حل دیگر عبارت است از :

رأس	V_1	V_2	V_3	V_4
رنگ	رنگ ۲	رنگ ۱	رنگ ۳	رنگ ۱

یک کاربرد مهم حل این مساله , رنگ آمیزی نقشه ها است ولی قبل از آن لازم است مفهوم گراف مسطح شرح داده شود .

گرافی را مسطح می گوییم به شرطی که آن را بتوان در یک صفحه به گونه ای رسم کرد که هیچ دو یالی همدیگر را قطع نکنند . مثلاً گراف شکل زیر (سمت چپ مسطح است) :






```
void  m_coloring (index i)
{
   int  color;
   if (promising (i))
        if (i == n)
             cout << vcolor [1] through vcolor[n] ;
        else
             for (color = 1; color <= m; color ++) {        // Try every color for
                  vcolor [i + 1] = color;                   // next vertex.
                   m_coloring (i + 1);
              }
}
```

```
bool promising (index i)
{
    index j ;
     bool switch;
     switch = true;
      j = 1;
     while (j < i && switch) {                    //check if an adjacent
           if (W[i][j] && vcolor[i] == vcolor [j])  //vertex is already this color
                switch = false;
           j ++;
     }
     return switch;
}
```

تابع فوق در سطح بالا به صورت m_coloring(0) صدا زده می شود . تعداد گره ها در درخت فضای حالت الگوریتم فوق برابر است با :

$$1+m+m^2+\ldots+m^2=\frac{m^{n+1}-1}{m-1}$$

مثال چهارم : مساله حاصل جمع زیر مجموعه ها

در این مساله n عدد صحیح مثبت W_i و یک عدد صحیح مثبت W داریم.هدف پیدا کردن تمامی زیر مجموعه هایی از این اعداد صحیح می باشد که حاصل جمع آنها برابر W است.

مثال برای n=5 و w=21 مقادیر زیر $w_1 = 5\,,\quad w_2 = 6\,,\quad w_3 = 10\,,\quad w_4 = 11$

$w_5 = 16$

حل :

مسئله شامل جواب های زیر است :

$w_1 = 5$, $w_2 = 6$, $w_3 = 10$, $w_4 = 11$, $w_5 = 16$

حل مسأله شامل جواب‌های زیر است :

$w_1 + w_2 + w_3 = 5 + 6 + 10 = 21$

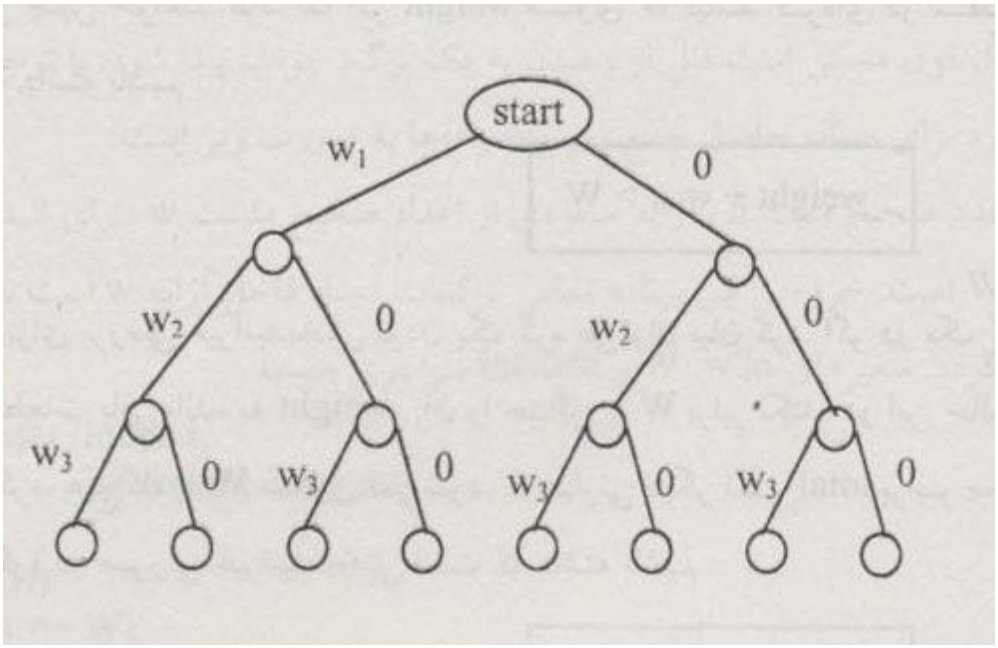
$w_1 + w_5 = 5 + 16 = 21$

$w_3 + w_4 = 10 + 11 = 21$

پس الگوریتم باید مجموعه های $\{w_1, w_2, w_3\}$و $\{w_1, w_5\}$و$\{w_3, w_4\}$ را به عنوان خروجی بدهد .

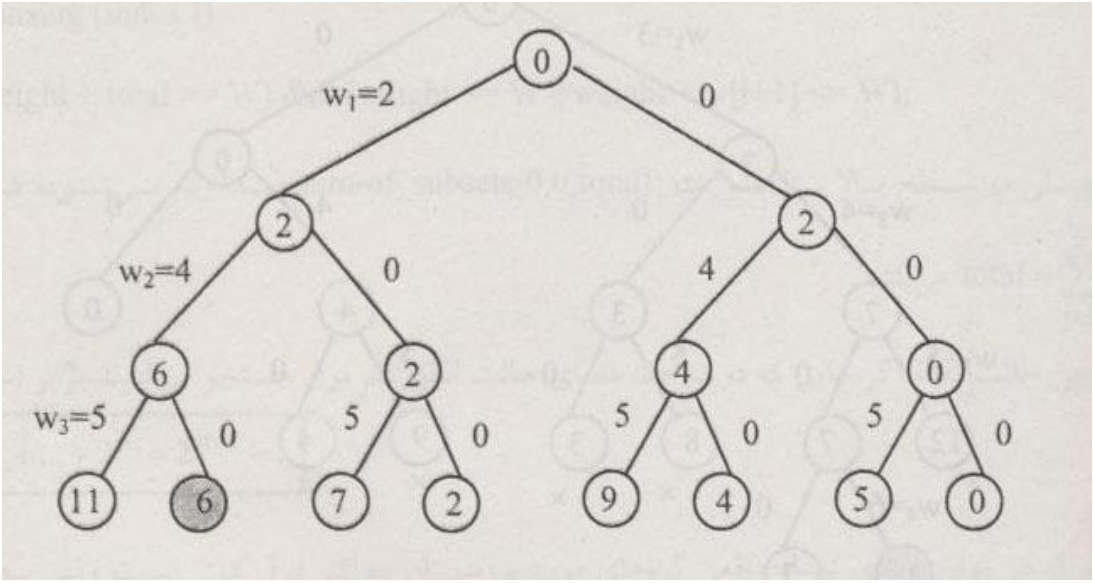
مسائلی مانند فوق را می توان به کمک ترسیم درخت فضای حالت و تکنیک عقبگرد حل کرد.

مثال : برای n=3 و w=6 ومقادیر $w_3 = 5$ و w_1=1, $w_2 = 4$ درخت فضای حالت به شکل زیر است :



اگر داخل گره های درخت فوق جمع اوزان انتخاب شده تا آن گره را بنویسیم با توجه به مقادیر $w=6$

w_1=1، $w_2 = 4$ و $w_3 = 5$ درخت فضای حالت زیر را به دست خواهیم آورد :



مسئله فوق یک جواب $\{w_1, w_2\}$ دارد.

وزن های بعدی آن نیز چنین خواهند بود. لذا اگر ***weight*** مساوی ***W*** نباشد گره ای در سطح ***i*** ام به شرطی غیر امید بخش است که داشته باشیم :

$$weight + w_{i+1} > w$$

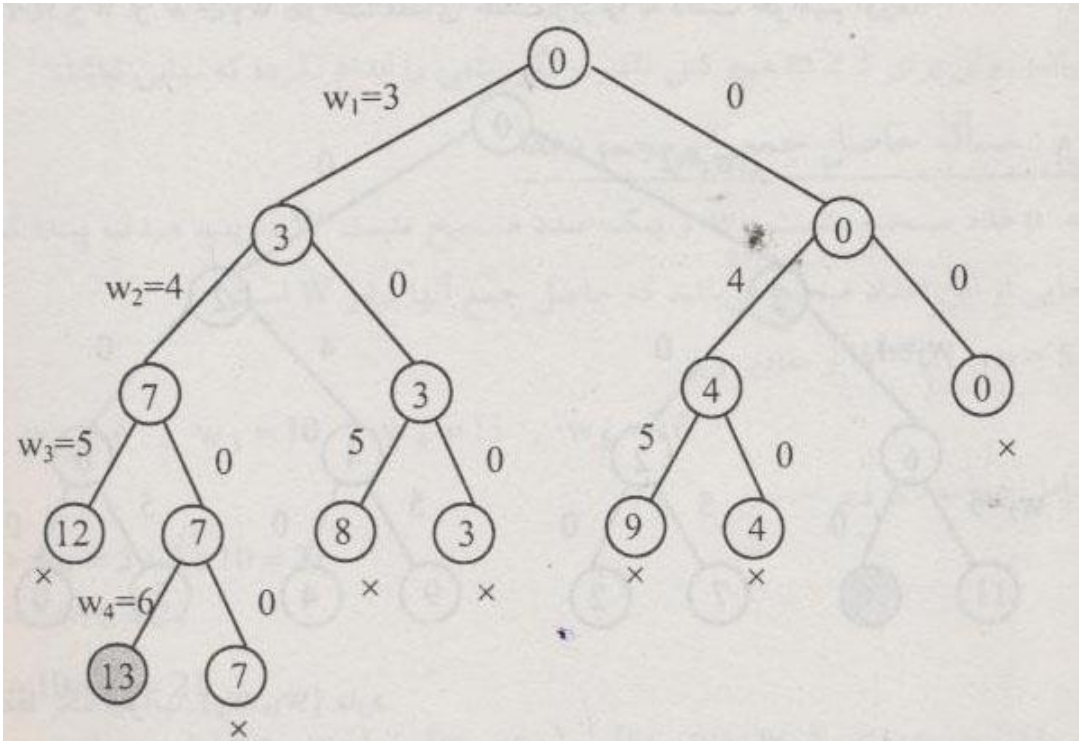
یک ویژگی دیگر نیز برای بررسی غیر امید بخش بودن یک گره می توان بیان کرد.اگر در یک گره معین اضافه کردن همه وزن های قطعات باقی مانده به $weight$ ، آن را حداقل با ***w*** برابر نکند در این حالت مقدار $weight$ در سطح بعدی آن گره هیچ گاه با ***W*** مساوی نمی شود. به عبارتی دیگر اگر ***Total*** برابر مجموع w_i های باقیمانده باشد یک گره در صورتی غیرامید بخش است که داشته باشیم.

$$weight + total < w$$

مثال : برای ***n=4*** و ***w=13*** و مقادیر $w_5 = 16$

$$w_1 = 3\,, \quad w_2 = 4\,, \quad w_3 = 5\,, \quad w_4 = 6$$

درخت فضای حالت هرس شده با در نظر گرفتن دو شرط گفته شده در فوق (جهت بررسی امیدبخش بودن گره ها)به صورت زیر خواهد بود. توجه کنید که کل درخت فضای حالت ۳۱ گره دارد و درخت هرس شده زیر دارای ۱۵ گره است :



گره های حاوی اعداد *12 , 8 , 9* به خاطر شرط $weight + w_{i+1} > w$ غیر امید بخش هستند . گره های حاوی اعداد *7,3,4,0* به خاطر شرط $weight + total < w$ غیر امید بخش می باشند.

ورودی این برنامه عدد صحیح مثبت n آرایه صعودی از اعداد صحیح مثبت w دارای اندیس های 1 تا n و عدد صحیح مثبت w است خروجی این برنامه تمامی ترکیبات اعداد داخل آرایه w است به گونه ای که حاصل جمع آن ها برابر w گردد . متغیرهای *include , W,w,n* سراسری هستند.

```
void sum_of_subset (index i,
                    int weight, int total)
{
   if (promising(i))
        if (weight == W)
              cout << include[1] through include[i];
        else {
              include[i + 1] = "yes";                    // Include w[i+1].
              sum_of_subsets (i + 1, weight + w[i+1], total - w[i+1]);
              include [i+1] = "nö ;                      //Do not include w[i+1].
              sum_of_subsets(i+1, weight, total - w[i+1]);
        }
}
```

تابع اصلی در سطح بالا به صورت sum_of_subsets (0,0,total) صدا زده می شود که در ابتدا $total = \sum_{j=1}^{n} w[j]$ می باشد.

در بدترین حالت تعداد گره هایی که در درخت فضای حالت الگوریتم فوق جستجو می شوند برابر است با :

$$1+2+2^2+\ldots+2^n = 2^{n+1}-1$$

هر چند که در موارد زیادی تعداد واقعی گره های مورد بررسی از حداکثر فوق کمتر است ولی به هر حال الگوریتم فوق از مرتبه نمایی است و ممکن است در شرایطی لازم باشد همه گره ها بررسی شوند. مثلاً در حالتی که داشته باشیم :

$$\sum_{j=1}^{n} w_i < W \qquad , \qquad w_n = W$$

آنگاه مساله فقط یک جواب $\{w_n\}$ داشته و برای یافتن آن باید همه گره ها بررسی شوند.

حل مساله کوله پشتی صفر و یک با تکنیک عقبگرد

در کتاب نیپولیتان مساله کوله پشتی صفر و یک به شیوه عقبگرد نیز حل شده است که مطالعه آن را بر عهده دانشجویان قرار می دهیم.

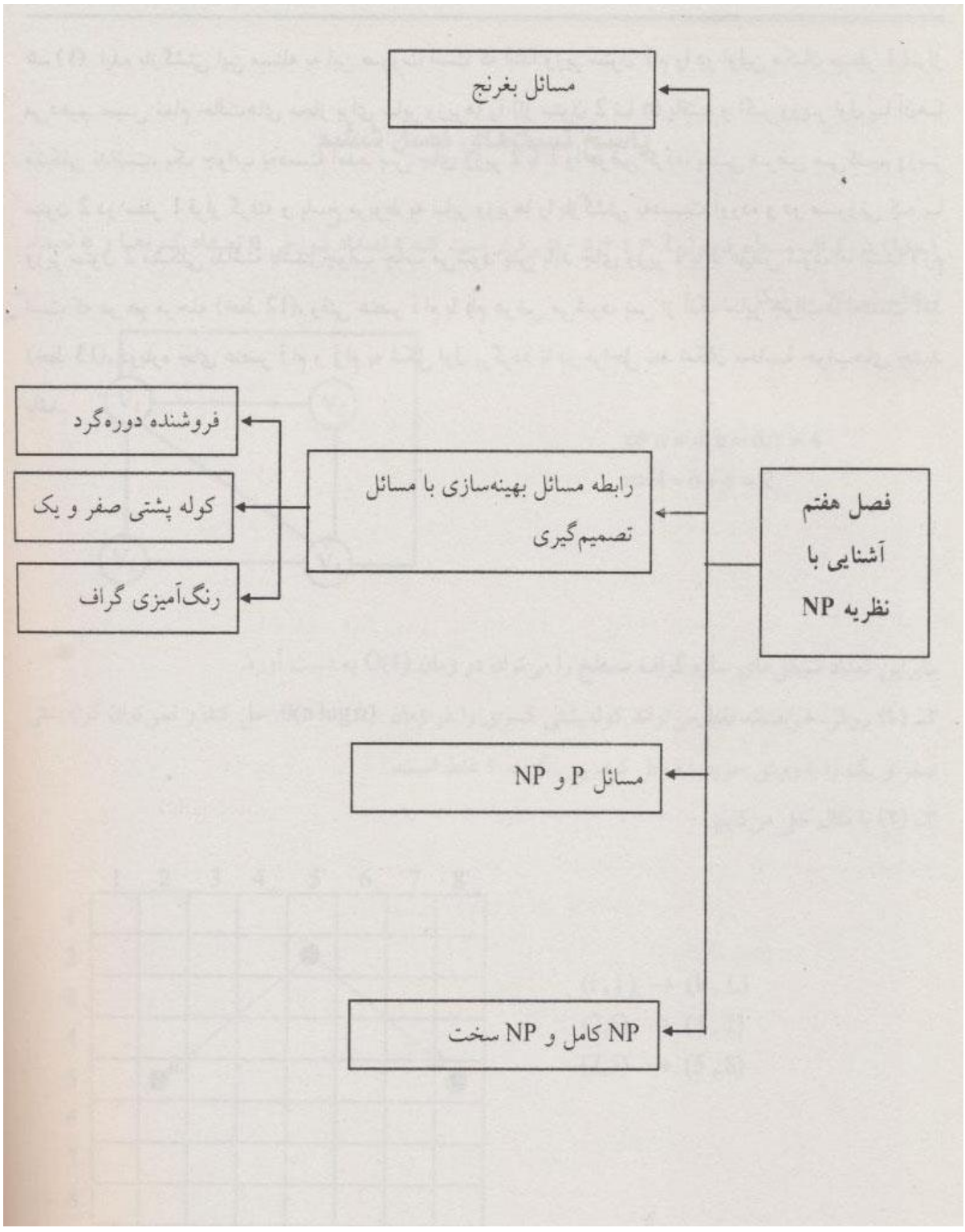
در اینجا فقط این نکته را یادآوری می کنیم که حل این مساله با روش برنامه نویسی پویا از مرتبه $O(Min\ (2^n\ , nW)$ بوده و در روش عقبگرد در بدترین حالت $\theta(2^n)$ گره را چک می کند.

ممکن است به خاطر nW تصور شود که تکنیک پویا به تکنیک به تکنیک عقبگرد در این مساله ارجحیت دارد ولی با اجرای واقعی روی کامپیوتر هورویتز و شانی در سال ۱۹۷۸ نشان دادند که در مساله کوله پشتی صفر و یک روش عقبگرد معمولاً بازدهی بیشتری نسبت به روش پویا دارد.

همچنین هورویتز و شانی در سال ۱۹۷۴ با ترکیب روش تقسیم و حل و روش برنامه نویسی پویا الگوریتمی را مطرح کردند که در بدترین حالت به $\theta(2^{n/2})$ تعلق داشت و نشان دادند این الگوریتم اغلب بازدهی بیشتری نسبت به الگوریتم عقبگرد دارد.

فصل هفتم

آشنایی با نظریه NP


فصل هفتم
آشنایی با
نظریه NP
مسائل بغرنج
رابطه مسائل بهینه‌سازی با مسائل تصمیم‌گیری
فروشنده دوره‌گرد
کوله پشتی صفر و یک
رنگ‌آمیزی گراف
مسائل P و NP
NP کامل و NP سخت

مسائل P و NP

P عبارت از تمامی مسائل تصمیم گیری می باشد که به کمک الگوریتم هایی از مرتبه چند جمله ای قابل حل هستند . P مخفف Polynomial (چند جمله ای) است.

به عنوان مثال تعیین کنید اینکه آیا کلید خاص در آرایه ای وجود دارد یا خیر از نوع مساله p هستند.مسائل زیادی جزو p هستند.از طرف دیگر مسائل اندکی مثل مساله halting و حساب پرزبرگر اثبات شده اندکه جزو p نمی باشند.

نکته جالب آن است که مسائل زیاد دیگری وجود دارد که نمی دانیم آیا جزو دسته p هستند یا خیر مانند مساله تصمیم گیری فروشنده دوره گرد.

مثلاً در مورد مساله تصمیم گیری فروشنده دوره گرد می توان الگوریتمی شبیه زیر نوشت که تصدیق کند آیا تور پیشنهادی وزن کوچکتر یا مساوی d را دارد یا خیر . ورودی این الگوریتم ، گراف G ، عدد d و رشته S حاوی تور ادعا شده است :

```
boolean  Verify (graph G, number  d,   tour  S)
 {
   if (S  is  a tour  and the total weight  in S is <= d)
      return  true;
   else
      return false;
 }
```

تعریف : الگوریتم چند جمله ای غیرقطعی یا NP یک الگوریتم غیرقطعی است که مرحله تصدیق آن زمان چند جمله ای می باشد.

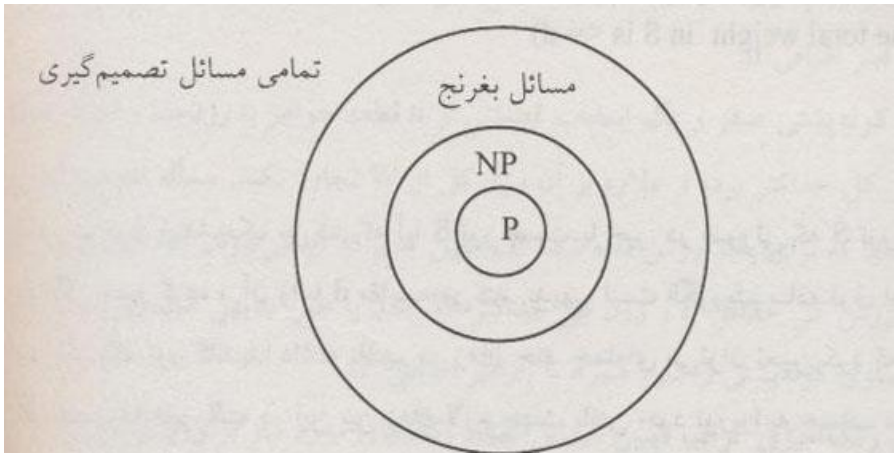
منظور از NP (Nondeterministic Polynomial) مجموعه تمامی مسائل تصمیم گیری است که توسط الگوریتم های چند جمله ای غیرقطعی قابل حل هستند.

تذکر مهم : NP مخفف Nonpolynomial (غیر چند جمله ای) نیست ، بلکه مخفف Nondeterministic Polynomial (چند جمله ای غیرقطعی) است.

مساله تصمیم گیری فروشنده دوره گرد یک مساله NP است چرا که عمل تصدیق آن را در زمان چند جمله ای صورت می گیرد ، البته ذکر این نکته بسیار ضروری است که معنای جمله فوق این نیست که الزاماً الگوریتمی با زمان چند جمله ای برای حل مساله داریم.

هدف از تعاریف P و NP طبقه بندی کردن الگوریتم ها است.

می توان اثبات کرد که مسائل تصمیم گیری فروشنده دوره گرد ، کوله پشتی صفر و یک و رنگ آمیزی گراف همگی از نوع NP هستند .همچنین هزاران مساله دیگر وجود دارند که اثبات شده جزو NP بوده ولی هیچکس تاکنون نتوانسته برای حل آن ها الگوریتم هایی چند جمله ای بیان کند . بدیهی است که مسائل P را می توان در زمان چند جمله ای حل کرد و لذا مرحله تصدیق آن ها نیز به سادگی چند جمله ای خواهد بود .بنابراین تمامی مسائل تصمیم گیری را به صورت شکل زیر می توان طبقه بندی کرد :



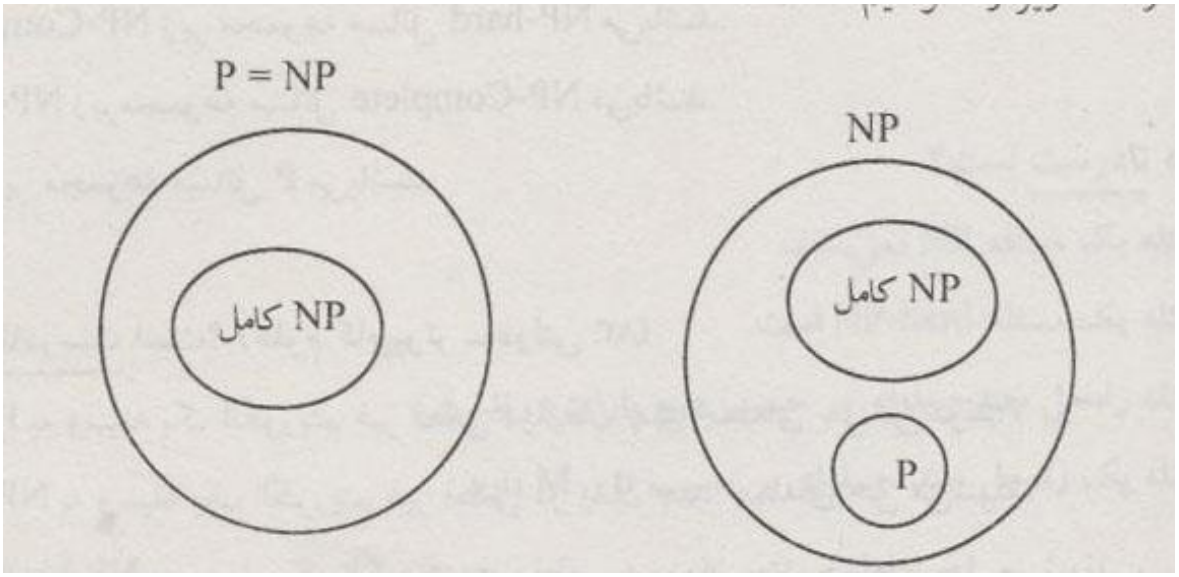
مسائلی که بغرنج بودن آنها به اثبات رسیده نظیر مساله halting و حساب پرزبرگر در NP نیستند.نکته جالب دیگر آن است که تاکنون کسی نتوانسته است اثبات کند که آیا P = NP است و یا برعکس $NP \neq P$ می باشد و این در واقع یکی از مهم ترین پرسش های بی پاسخ در علم کامپیوتر است.

این سوال از آن جهت اهمیت دارد که اکثر مسائل تصمیم گیری جزو NP می باشند و بنابراین اگر P = NP باشد برای اکثر مسائل تصمیم گیری ، می توان الگوریتم های چند جمله داشت . البته بسیاری از دانشمندان شک دارند که NP = P باشد.

NP-hard , NP-Complete

شرح کامل مسائل NP کامل (NP-Complete) و NP سخت (NP_hard) در کتاب نیپولیتان آمده است که به عنوان تحقیق و مطالعه بر عهده دانشجویان قرار می دهیم.

در اینجا خیلی خلاصه بیان می کنیم که اگر P = NP باشد شکل سمت چپ زیر و اگر $P \neq NP$ باشد شکل سمت راست زیر را خواهیم داشت :



یعنی اگر $P \neq NP$ باشد آنگاه NP کامل ∩ P = ∅ است. همچنین هر مساله NP کامل ، NP سخت نیز می باشد یعنی NP کامل زیر مجموعه NP سخت است.