فصل اول :

الگوریتــم:

الگوریتم به روش حل هر دسته از مسائل گفته می‌شود. الگوریتم باید به صورت رشــته‌ای از اعمــال کــه حل دسته‌ای از مسائل را به دقت تبیین می‌نماید، سازماندهی شده باشد؛ این اعمال جزعــی بایــد بــدون ابهــام باشند و زمان اجرای متناهی داشته باشند.

ارزیابی کارایی الگوریتــم‌ها:

جهت مقایسه‌ی میزان کارایی هر الگوریتم احتیاج به معیارهایی است که دو معیار اساسی آن چنین‌اند.

I. زمان لازم برای اجرای کامل الگوریتم.

II. حداکثر میزان حافظه‌ی لازم در زمان اجرای الگوریتم.

تذکر: توجه کنید که اگر یک الگوریتم را بوسیله‌ی دو کامپیوتر متفاوت، با توانــایی‌هــا و ســرعت غیــر یکسان اجرا کنیم، دو زمان اجرای متفاوت خواهیم داشت. لذا بهتر است بجای معیارهــای فــوق از دو معیار زیر جهت ارزیابی و مقایسه‌ی کارایی الگوریتم‌ها استفاده نماییم.

I. مرتبه‌ی زمانی اجرای کامل الگوریتم.

II. مرتبه‌ی مکانی اجرای الگوریتم.

بر اساس تعریف‌های مختلف جهت مرتبه‌های زمانی و مکانی کــارایی الگــوریتم بــصورت ضــریبی از تعداد اعمال کلیدی که تکرار آن بیــشترین باشــد و بیــشترین وقــت و حافظــه کــامپیوتر را بــه خــود اختصاص دهد، سنجیده و محدود می‌گردد.

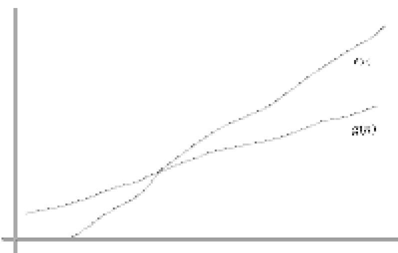
از این دسته می‌توان به موارد زیر اشاره داشت.

1- **تعریف نماد O** :

اگر $f : N \rightarrow R \geq 0$ آنگاه :

$$O(f(n)) = \{g : N \rightarrow R \geq 0 \mid \exists c > 0 \forall_n^\infty g(n) \leq cf(n)\}$$

یعنی از یک جا به بعد برای هر n داریم $g(n) = O(f(n))$



2- **تعریف نماد Ω** :

اگر $f : N \rightarrow R \geq 0$ آنگاه :

$$\Omega(f(n)) = \{g : N \rightarrow R \geq 0 \mid \exists d > 0 \forall_n^\infty df(n) \leq g(n)\}$$

نتیجه: $g(n) = \Omega(f(n)) \Leftrightarrow f(n) = O(g(n))$

3- تعریف نماد Θ :

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$
$$\Theta(f(n)) = \{g : N \rightarrow R \geq 0 \mid \exists d, c > 0 \forall_n^\infty df(n) \leq g(n) \leq cf(n)\}$$

نکته: **وقتی گفته می‌شود زمان اجرای الگوریتم $O(n^2)$ است یعنی الگوریتم هر جوری اجرا شود، مرتبه‌ی زمانی اجرای آن یا $\mathbf{n^2}$ است و یا از $\mathbf{n^2}$ کمتر است.**

$$O(n^2) = \{n^2, 5n + 6n, n, n, n^{\frac{3}{2}}, 4n^{\frac{3}{2}}, 5n, \ldots\}$$

نکته: **وقتی گفته می‌شود زمان اجرای الگوریتم $\Omega(n^2)$ است یعنی الگوریتم هر جوری اجرا شود مرتبه‌ی زمانی اجرای آن $\mathbf{n^2}$ یا بیشتر از $\mathbf{n^2}$ است.**

$$\Omega(n^2) = \{2n^2, 5n^2, 6n + n^3, \ldots\}$$

نکته: **وقتی گفته می‌شود زمان اجرای الگوریتم $\Theta(n^2)$ است یعنی الگوریتم هر جوری اجرا شود، مرتبه‌ی زمانی اجرای آن دقیقاً $\mathbf{n^2}$ خواهد بود.**

$$\Theta(n^2) = \{n^2, 2n^2, 2n^2 + 6n, \ldots\}$$

4- تعریف نماد o :

$$o(f(n)) = \{g : N \rightarrow R \geq 0 \mid \forall c > 0 \forall_n^\infty g(n) \leq cf(n)\}$$
$$o(f(n)) = \{g : N \rightarrow R \geq 0 \mid \forall c > 0 \forall_n^\infty \lim \frac{g(n)}{cf(n)} \rightarrow 1\}$$

$n = o(5n)$ **نادرست** $\Leftrightarrow \forall c > 0 \forall_n^\infty n \leq 5cn \Leftrightarrow c = \frac{1}{10}$

$5n = o(n)$ **نادرست** $\Leftrightarrow \forall c > 0 \forall_n^\infty 5n \leq cn \Leftrightarrow c = 1$

$n = o(n^2)$ **درست** $\Leftrightarrow \forall c > 0 \forall_n^\infty n \leq cn^2$

قضیه‌ی ماکسیمم‌ها :

اگر $f, g : N \rightarrow R \geq 0$ آنگاه:

$$f(n) + g(n) = O(\max\{f(n), g(n)\})$$
$$f(n) + g(n) = \Theta(\max\{f(n), g(n)\})$$
$$f(n) + g(n) = \Omega(\max\{f(n), g(n)\})$$

اثبات:

$$\left.\begin{array}{l} f(n) \leq \max\{f(n), g(n)\} \\ g(n) \leq \max\{f(n), g(n)\} \end{array}\right\} f(n) + g(n) \leq 2\max\{f(n), g(n)\}$$

$$\Rightarrow f(n) + g(n) = O(\max\{f(n), g(n)\})$$
$$\Rightarrow \max\{f(n), g(n)\} \leq f(n) + g(n) \Rightarrow f(n) + g(n) = \Theta(\max\{f(n), g(n)\})$$

5- تعریف نماد w:

$$w(f(n))=\{g:N\rightarrow R\geq 0\mid \forall d>0\forall_n^{\infty} df(n)\leq g(n)\}$$

$$w(f(n))=\{g:n\rightarrow R\geq 0\mid \forall d>0\forall_n^{\infty} \lim \frac{g(n)}{cf(n)}\rightarrow 1\}$$

$$cf(n)<g(n)$$

نکته : $g(n)=w(f(n))\Leftrightarrow f(n)=o(g(n))$

قضیه : اگر $f,g:N\rightarrow R\geq 0$ و $L=\lim_{n\rightarrow\infty}\frac{f(n)}{g(n)}$

1- اگر $L=0$ آنگاه $f(n)=O(g(n))$.

2- اگر $L=\infty$ آنگاه $g(n)=o(f(n))$.

3- اگر $0<L<\infty$ آنگاه $f(n)=\Theta(g(n))$.

آنالیـــزالگوریتم‌ها

برای آنالیز هر الگوریتم از سه اصل زیر استفاده می‌شود.

1- **اصل پایانی** : اگر از یک الگوریتم دو پیاده سازی مختلف داشته باشیم که یکی زمان t_1 و دیگری زمـان t_2 را نیاز داشته باشد در این صورت:

$$t_1=\Theta(t_2)$$

2- **اصل ترتیب‌گذاری** : اگر p_1 و p_2 دو قطعه برنامه مستقل باشند، زمان لازم برای اجرای متوالی p_1 و p_2 برابر t_1+t_2 خواهد بود.

$$t_1+t_2=\Theta(\max\{t_1,t_2\})$$

3- **اصل دستورات اتمی** : منظور از یک دستور اتمی، دستورات مقدماتی در سطح ماشین است مثل دستور *Shift*، + ، - ، *goto*، مقایسه، جایگزینی(انتساب) و ... ؛ این دستورات در سطح سخت افزار به زمانی در حد $\Theta(1)$ نیاز دارند.

آنالیز حلقه‌ی While :

$$i\leftarrow 1$$
$$while(i<=m)do$$
$$\quad p(i);$$
$$\quad i\leftarrow i+1;$$

فرض کنید یک کران بالا برای دستورات جایگزینی، مقایسه‌ای، جمع و عمل *goto* عددی چون c باشد و کران بالا برای اجراهای متمایز $p(i)$ عددی چون t باشد. اگر L زمان اجرای مجموعه دستورات فوق باشد داریم:

$L \le c$ برای عمل جایگزینی

$+(m+1)c$ برای مقایسه

$+mt$ برای $p(i)$

$+mc$ برای جایگزینی

$+mc$ برای goto

$$\Rightarrow L \le 3mc + mt + 2c \Rightarrow L = O(3mc + mt + 2c) \Rightarrow L = O(\max\{m, mt, 1\})$$

بدیهی است که اگر پراکندگی زمانی اجراهای متمایز $p(i)$ زیاد باشد در این صورت

$$L = O(\max\{m, \sum_{k=1}^{m} t_k, 1\})$$

نکته : در سطح ماشین، دستور for مانند $while$ است، لذا آنالیز یکسانی دارند.

آنالیز الگوریتم مرتب سازی حبابی (Bubble sort) :

```
Procedure BubbleSort (T[1..n])
    for i ← 1 to n−1 do
        for j ← 1 to n−i do
            if T[j] > T[j+1] then swap(T[j],T[j+1])
```

در محاسبه‌ی کران بالا، همواره فرض می‌کنیم که شرط if برقرار است و دستور $swap$ اجرا می‌شود. در این صورت زمان لازم برای مجموع این عملیات در حد زمان $O(1)$ است.

$$L \le \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1$$

$$L \le \sum_{i=1}^{n-1} (n-(i+1)+1) = \sum_{i=1}^{n-1} (n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i$$

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

$$L = O(n^2) \quad \text{I}$$

محاسبه‌ی کران پایین:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 \le L \Rightarrow \frac{n(n-1)}{2} \le L \Rightarrow L = \Omega(n^2) \quad \text{II}$$

$$I, II \Rightarrow L = \Theta(n^2)$$

چند تعریف :

$$\sum_{K=L}^{m} a = (m-L+1)a \qquad \sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{k=L}^{m} a_k = \sum_{k=L-p}^{m-p} a_{k+p} = \sum_{k=L+p}^{m+p} a_{k-p}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad \lim_{n\to\infty} \sum_{n=L}^{m} n^p \cong \lim_{n\to\infty} \frac{n^{p+1}}{p+1}$$

آنالیز الگوریتم مرتب سازی انتخابی (Selection sort) :

Procedure Select (T [1..n])

$for\ i \leftarrow 1\ to\ n-1\ do$

$\min x \leftarrow T[i]$

$\min y \leftarrow i$

$for\ j \leftarrow i+1\ to\ n\ do$

$if\ T[j] < \min x\ then$

$\min x \leftarrow T[j]$

$\min y \leftarrow j$

$T[\min y] \leftarrow T[i]$

$T[i] \leftarrow \min x$

محاسبه‌ی کران بالایی:

$$L \le \sum_{i=1}^{n-1}(1+\sum_{i=1}^{n}1)$$

$$L \le \sum_{i=1}^{n}(1+n-i) = \sum_{i=1}^{n-1}1+(n-(i+1)+1)$$

$$L \le O(n^2)$$

محاسبه‌ی کران پایین:

$$\sum_{i=1}^{n-1}(1+\sum_{j=i+1}^{n}1) \le L$$

$$L \in \Omega(n^2)$$

$\Rightarrow L \in \Theta(n^2)$ مرتبه‌ی زمانی اجرای الگوریتم انتخابی

از مقایسه‌ی دو الگوریتم مرتب سازی حبابی و مرتب سازی انتخابی، درکل فرقی با هم ندارند، ولی الگوریتم مرتب سازی انتخابی، در ضریب بدتر است.

آنالیز الگوریتم مرتب سازی درجی (Insert sort) :

Procedure insertion (T [1..n])

$for\ i \leftarrow 2\ to\ n\ do$

$i \leftarrow T[i]$

$j \leftarrow i-1$

$while(j>0\ \ \&\&\ \ x<T[j])$

$T[j+1] \leftarrow T[j]$

$j \leftarrow j-1$

$T[j+1] \leftarrow x$

محاسبه‌ی کران بالایی:

$$L \le \sum_{i=2}^{n}(i-1+1) \Rightarrow L \le O(n^2)$$

محاسبه‌ی کران پایین:

$$\sum_{i=2}^{n} 1 \le L \Rightarrow L = \Omega(n)$$

کمترین زمان وقتی است که $while$ اصلاً اجرا نشود. یعنی زمان اجرای آن $\Omega(1)$ باشد.

زمان اجرای الگوریتم درجی $\Omega(n) \le L \le O(n^2)$

در حالی که کران بالایی و پایینی الگوریتم‌ها متفاوت است معمولاً آنالیز هر الگوریتم را در سه حالت زیر انجام می‌دهیم.

1- آنالیز در بهترین حالت(Best case analysis)

یعنی ورودی‌ها بگونه‌ای باشند که الگوریتم کمترین زمان را ببرد.

2- آنالیز در بدترین حالت(Worst case analysis)

یعنی ورودی‌ها بگونه‌ای باشند که الگوریتم بیشترین زمان را سپری کند.

3- آنالیز در حالت متوسط(Average case analysis)

برای مثال در آنالیز مرتب سازی درجی اگر ورودی‌ها بصورت صعودی، مرتب شده باشند بهترین حالت رخ می‌دهد. در این حالت زمان الگوریتم $\Theta(n)$ خواهد بود. اگر داده‌ها به شکل نزولی مرتب شده باشند بدترین حالت رخ می‌دهد که در این حالت زمان اجرای الگوریتم $\Theta(n^2)$ خواهد بود. محاسبه‌ی حالت متوسط یعنی در حالت متوسط با ورودی‌های مشخص زمان اجرا چه خواهد بود.

یکی از مشکلاتی که در آنالیز الگوریتم‌ها وجود دارد، آنالیز در حالت متوسط است. در این گونه موارد در حالت کلی برای n داده، زمان لازم برای بدست آوردن ترتیب‌های مختلف داده‌هاست که می‌تواند این n داده بعنوان ورودی در نظر گرفته شود و $n!$ حالت می‌باشد.

بنابراین برای محاسبه‌ی متوسط این $n!$ حالت، زمان‌ها را محاسبه کرده و جمع نموده و بر تعداد آنها تقسیم می‌کنیم. ولی از لحاظ منطقی نشدنی است، زیرا زمان لازم برای این عملیات $O(n!)$ است و این زمان خیلی خیلی زیاد است. بنابراین روش منطقی‌تر استفاده از روش آماری می‌باشد. در روش آماری از تکنیک‌های خاص استفاده می‌کنیم. برای مثال در الگوریتم Insertion Sort از مفهوم Partial Rank (رتبه‌ی جزئی) استفاده می‌کنیم. منظور از رتبه‌ی جزئی عنصر $T[i]$ از آرایه‌ی $T[1..i]$ عبارت است از محل قرار گیری $T[i]$ در آرایه‌ی $T[1..i]$ پس از مرتب سازی زیر آرایه‌ی $T[1..i]$. در این الگوریتم حلقه‌ی while رتبه‌ی جزئی عنصر $T[i]$ را محاسبه می‌کند.

اگر c_i متوسط زمان لازم برای قرارگیری عنصر i ام در محل واقعی خود از زیر آرایه‌ی $T[1..i]$ باشد در این صورت

یک جابجایی یا دو جابجایی یا

$$c_i = \frac{1}{i}\sum_{k=1}^{i} k = \frac{1}{i}\frac{(i+1)i}{2} = \frac{i+1}{2}$$

$$\sum_{i=2}^{n} c_i = \sum_{i=2}^{n} \frac{i+1}{2} = \Theta(n^2)$$

مرتب سازی لانه کبوتری :

T

2	3	1	2	1	3	2	1	4

U:

0	0	0	0
1	2	3	4

3	3	2	1
1	2	3	4

$\Rightarrow$ 1 1 1 2 2 2 3 3 4

Procedure Pigeonhole (T [1..n])
*let $m = \max T[i]$, $for 1 \le i \le n$ and $T[i]$ are positive integer */

U $u[1..m]$

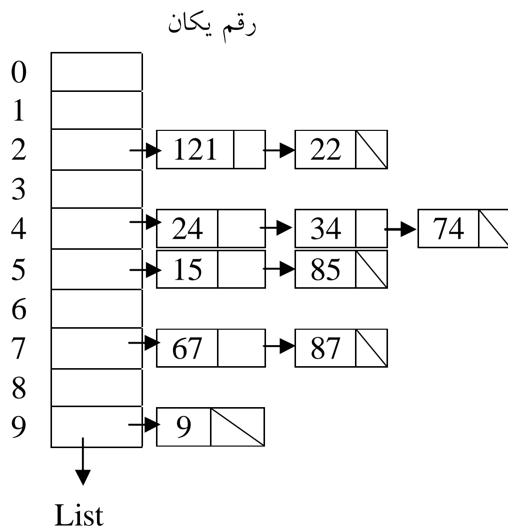
$for\ i \leftarrow 1\ to\ m\ do$ $\Theta(m)$
$\quad u[i] \leftarrow 0$
$for\ i \leftarrow 1\ to\ n\ do$
$\quad k \leftarrow T[i]$ $\Theta(n)$
$\quad u[k] \leftarrow u[k]+1$
$i \leftarrow 1$
$for\ k \leftarrow 1\ to\ m\ do$
$\quad while(u[k] <> 0) do$
$\quad\quad T[i] \leftarrow k$
$\quad\quad i \leftarrow i+1$

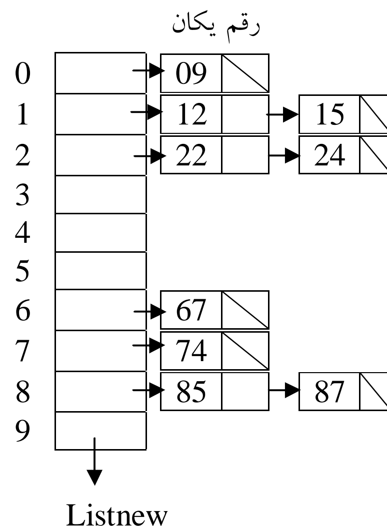
$$\Theta(\sum_{k=1}^{m} u[k]) = \Theta(n)$$

نکته : هیچ الگوریتم مرتب سازی که مبتنی بر مقایسه باشد زمان اجرای آن کمتر از $n \log n$ نیست.

مرتب سازی مبنا (Radix Sort):

24 15 67 34 12 22 87 85 74 9





مرتبه‌ی اجرا این الگوریتم $O(a(n)n)$ می‌باشد که $a(n)$ حداکثر تعداد ارقامی است که بین n عدد موجود است.

```
Procedure RadixSort (int max List[1..n])
/*let m = length max , Positive Integer */
/*let List[0..9]of Pointer */
/*let ListNew[0..9]of Pointer */
for i := 1 to n do
        {x = mainlist[i]%10
        addq(list[x], mainlist[i])}
for i = 1 to m-1 do
        {for j = 0 to 9 do
                while(list[j])
                        {x = addq(list[j]);
                        z = x% 10^(i+1);
                        y = z/10^I;
                        addq(listnew[j], x);}
        point p;
        p = listnew;
        listnew = list;
        list = p;}
```

فصل دوم : برگشت‌پذیری و الگوریتم‌های بازگشتی

برخی از زبان‌های برنامه‌سازی این امکان را دارند که بتوانیم در آنها ساختار استقرایی را پیاده سازی نمائیم. منظور از ساختارهای استقرایی ساختارهایست که درآنها جهت دستیابی به یک عنصر لازم است یک، دو و یا چند عنصر دیگر را داشته باشیم و از آنها بصورت استقرایی به داده جدید دست یابیم. عملاً جهت اجرای یک قسمت از الگوریتم، احتیاج به بازگشت و اجرای همین الگوریتم با داده‌های جدید می‌باشد. برای نوشتن الگوریتم‌های بازگشتی باید به دو نکته‌ی زیر دقت شود.

I. باید یک شرط خروج داشته باشیم. یعنی باید بر حسب یک یا چند مقدار اولیه، تابع مذکور مشخص باشد.

II. فرض شود که الگوریتم برای مرحله‌ی قبل و یا مراحل قبل کار می‌کند و فقط باید آنرا برای این مرحله ارتقاء داد.

مثال :تابع فاکتوریل را بصورت بازگشتی بنویسید.

می‌دانیم

$$n!= \begin{cases} 1 & n=0 \\ n(n-1) & else \end{cases}$$

$$\begin{array}{l} fact(\text{int} \;\; n) \\ \{ \\ \qquad if \;\; n <= 1 \;\; then \;\; return \;\; 1; \\ \qquad else \;\; return(n * fact(n-1)) \\ \} \end{array}$$

نکته :دقت شود که در مورد ساختارهای بازگشتی پیچیده، معمولاً Trace کردن برنامه هم زمان‌بر است و هم ممکن است با بی‌دقتی انجام پذیرد. لذا بهتر است ابتدا رابطه‌ی بازگشتی آنها را بدست آوریم.

توجه :یک روش جهت بررسی درستی الگوریتم‌های بازگشتی استفاده از ساختار درختی است که می‌تواند با پشته پیاده سازی شود.

جهت بررسی زمانی الگوریتم‌های بازگشتی می‌توان از رابطه‌ی بازگشتی آن‌ها به معادله‌ی بازگشتی زمان اجرا دست یافت و با حل آن معادله به مرتبه زمانی این الگوریتم‌ها رسید.

معادلات بازگشتی :

در یک معادله‌ی بازگشتی، مقدار هر جمله برحسب یک یا چند جمله‌ی قبل محاسبه می‌شود. بطور کلی علاقه‌مند هستیم معادلات بازگشتی را مورد ارزیابی قرار دهیم که ضرایب آن ثابت و حقیقی باشند. این رده از معادلات را به دو دسته معادلات بازگشتی با ضرایب ثابت همگن و معادلات بازگشتی با ضرایب ثابت ناهمگن تقسیم می‌کنیم.

در معادلات همگن $f(n)$ صفر است.

$$f(n): N \to R^{\geq 0}$$
$$\forall \;\; 0 \leq i \leq k, a_i \in R$$

$$a_0 t_n + a_1 a_{n-1} + a_2 a_{n-2} + a_3 a_{n-3} + ... + a_k a_{n-k} = f(n)$$

برای حل معادلات همگن بازگشتی با ضرایب ثابت به شکل زیر عمل می‌کنیم.

ابتدا معادله‌ی شاخص را بدست می‌آوریم. برای بدست آوردن معادله‌ی شاخص، هر t_i را به x^i تبدیل می‌کنیم، سپس ریشه‌های این معادله را بدست می‌آوریم.

$$a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots + a_kx^{n-k} = 0$$

$$x^{n-k}(a_0x^k + a_1x^{k-1} + \ldots + a_k) = 0$$

معادله‌ی شاخص یا مفسر یا سرشتنمایی

$$a_0x^k + a_1x^{k-1} + \ldots + a_k = 0$$

ممکن است یکی از شرایط زیر رخ دهد:

I. معادله دارای k ریشه‌ی حقیقی دوبه‌دو متمایز باشد. در این صورت جواب معادله به صورت زیر است.

$$t(n) = c_1(r_1)^n + c_2(r_2)^n + \ldots + c_k(r_k)^n$$

II. اگر تمام ریشه‌های معادله حقیقی باشند و مثلاً یکی از ریشه‌ها مانند r_p تکراری از مرتبه‌ی m باشد (یعنی در تجزیه‌ی معادله شاخص، عامل $(x-r_p)^m$ را داشته باشیم ولی عامل $(x-r_p)^{m+1}$ وجود نداشته باشد) در این صورت جواب معادله به فرم زیر است.

$$t(n) = c_1(r_1)^n + c_2(r_2)^n + \ldots + c_{p_0}(r_{2p})^n + c_{p_1}n(r_p)^n + c_{p2}n^2(r_p)^n + \ldots +$$
$$c_{p_{m-1}}n^{m-1}(r_p)^n + \ldots + c_t(r_t)^n$$

III. معادله دارای ریشه‌ی موهومی باشد که در اینجا بحث نمی‌شود.

نکته: غیر از روش جامع فوق دو روش دیگر نیز وجود دارد که در مورد همه مسائل قابل استفاده نیست.

1- حدس زدن جواب رابطه و اثبات آن به روش استقراء

مثال: حدس $t_n = \log n + 1$

$$\begin{cases} t_n = t_{\frac{n}{2}} + 1 \\ t_1 = 1 \end{cases}$$

استقراء

$$\begin{cases} \log 1 + 1 = 1 \\ t_{\frac{k}{2}} + 1 = \log_{\frac{k}{2}} + 2 = \log_{\frac{k}{2}} + \log_{\frac{2}{2}} + 1 = \log k + 1 \end{cases}$$

2- جایگزاری متوالی

مثال:

$$\begin{cases} t_n = 7t_{\frac{n}{2}} \\ t_1 = 1 \end{cases}$$

$$t_2 = 7t_{\frac{2}{2}} = 7t_1 = 7$$

$$t_4 = 7t_{\frac{4}{2}} = 7t_2 = 7^2$$

$$t_8 = 7t_{\frac{8}{2}} = 7t_4 = 7^3$$

$$t_{16} = 7t_{\frac{16}{2}} = 7t_8 = 7^4$$

به نظر می‌رسد $t_n = 7t^{\log n}$

مثال: مقدار $g(n)$ چیست؟

```
int g(int n)
{
if n <= 1 return (n);
else return(5g(n-1) - 6g(n-2));
}
```

$$g(n) = \begin{cases} 5g(n-1) - 6g(n-2) & else \\ n & n \le 1 \end{cases}$$

$$\Rightarrow g(n) = 5g(n-1) - 6g(n-2) \Rightarrow g(n) - 5g(n-1) + 6g(n-2) = 0$$

معادله شاخص $x^n - 5x^{n-1} + 6x^{n-2} = 0 \Rightarrow x^{n-2}(x^2 - 5x + 6) = 0 \quad r_1 = 2$

$$r_2 = 3$$

$$g(n) = c_1(2)^n + c_2(3)^n$$

$$n = 0 \Rightarrow 0 = g(0) = c_1 + c_2$$
$$n = 1 \Rightarrow 1 = g(1) = 2c_1 + 3c_2 \quad \Rightarrow \quad c_2 = 1, \; c_1 = -1 \qquad g(n) = 3^n - 2^n$$

مثال :

$$\begin{cases} t_n - 4t_{n-1} + 4t_{n-2} = 0 \\ t_1 = 4, t \;\; = 1 \end{cases}$$

$$x^2 - 4x + 4 = 0 \begin{cases} r_1 = 2 \\ r_2 = 2 \end{cases}$$

$$t_n = c_1(2^n) + c_2 n(2)^n$$
$$1 = t_0 = c_1 + c_2(0) \rightarrow c_1 = 1$$
$$4 = t_1 = 2c_1 + 2c_2 \rightarrow c_2 = 1$$
$$t_n = 2^n + n2^n$$

جهت حل معادلات بازگشتی غیر همگن با فرم عمومی زیر داریم:

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \ldots + b_m^n p_m(n)$$

که در آن هر $p_i(n)$ چند جمله‌ای از درجه‌ی d_i است.در این صورت معادله‌ی مشخصه به صورت زیر است.

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_n)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1} \ldots (x - b_m)^{d_m+1} = 0$$

حال ریشه‌های معادله‌ی فوق را بدست می‌آوریم و مانند قبل عمل می‌کنیم.

مثال:

$$t_n - 5t_{n-1} + 4t_{n-2} = 2^n + n$$
$$2^n(n)^0 + n(1)^n$$

$$(x^2-5x+4)(x-2)^{0+1}(x-1)^{1+1} \Rightarrow (x-4)(x-1)^3(x-2)=0$$
$$r_1=1, r_2=1, r_3=1, r_4=2, r_5=4$$
$$t_n=c_1(1)^n+c_2n(1)^n+c_3n^2(1)^n+c_2(2)^n+c_5(4)^n$$
$$t_n=c_1+c_2n+c_3n^2+c_4 2^n+c_4 4^n$$

جهت حل معادلات غیر همگن می‌توان از تغییر متغیر نیز استفاده کرد.

$$t_n=\begin{cases}1 & n=1\\ 2t(\frac{n}{2})+2 & n=2^k\end{cases}$$
$$n=2^k \Rightarrow t(n)=t(2^k)$$
$$t(2^k)=2t(\frac{2^k}{2})+2=2t(2^{k-1})+2$$
$$g(k)=2g(k-1)+2 \Rightarrow g(k)-2g(k-1)=2(1)^n$$
$$(x-2)(x-1)^1=0 \quad \begin{cases}r_1=1\\ r_2=2\end{cases} \quad g(k)=c_1(1)^k+c_2(2)^k$$
$$\Rightarrow g(k)=c_1+c_2 2^k \Rightarrow t(2^k)=c_1+c_2 2^k \Rightarrow t(n)=c_1+c_2 n$$
$$\begin{matrix}t(1)=c_1+c_2=1\\ t(2)=c_1+c_2=4\end{matrix} \quad \Rightarrow \quad \begin{matrix}c_1=-2\\ c_2=3\end{matrix} \quad \Rightarrow t_2=3n-2$$

مثال:

$$\begin{cases}t(n)+nt(n-1)=n!\\ t(0)=1\end{cases}$$

طرفین معادله را بر $n!$ تقسیم می‌کنیم.

$$\frac{t(n)}{n!}+\frac{nt(n-1)}{n!}=1 \Rightarrow \frac{t(n)}{n!}+\frac{nt(n-1)}{(n-1)!}=1$$
$$g(n)=\frac{t(n)}{n!} \Rightarrow g(n)+g(n-1)=1*(1)^n$$
$$(x+1)(x-1)^1=0 \quad \begin{cases}r_1=1\\ r_2=-1\end{cases} \quad \begin{matrix}g(n)=c_1(1)^n+c_2(-1)^n\\ t(n)=c_1n!+c_2(-1)^n n!\end{matrix}$$
$$\begin{matrix}t(0)=1 \Rightarrow c_1+c_2=1\\ t(1)=0 \Rightarrow c_1-c_2=0\end{matrix} \quad \Rightarrow \quad \begin{matrix}c_1=1/2\\ c_2=1/2\end{matrix} \quad t(n)=\frac{n!}{2}(1+(-1)^n)$$

مثال:

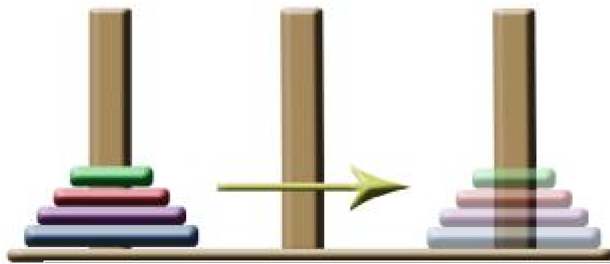
$$\left.\begin{matrix}t(n)=2t(n-1)+3^n \Rightarrow 3t(n)=6t(n-1)+3^{n+1}\\ t(n+1)=2t(n)+3^{n+1}\end{matrix}\right\}$$

$$t(n+1)-3t(n)=2t(n)-6t(n-1)$$
$$t(n+1)-5t(n)+6t(n-1)=0$$
$$x^2-5x+6=0 \Rightarrow t(n)=c_1 3^n+c_2 2^n$$

بررسی چند الگوریتم بازگشتی :

برج هانوی :

سه میله با نام‌های A،B وC داریم. فرض می‌شود درون یکی از این میله‌ها n دیسک قرار دارد. این دیسک‌ها هر-کدام دارای رنگ متمایز با دیگری می‌باشد. این n دیسک طوری قرار دارند که دیسکی با قطر بیشتر روی دیسکی با قطر کوچکتر قرار نگرفته است. می‌خواهیم این دیسک‌ها رابه میله‌ی دیگری به کمک میله‌ی باقیمانده جابجا کنیم. در هر مرحله باید یک دیسک جابجا شود و در این جابجایی نباید دیسک بزرگتر روی دیسک کوچکتر قرارگیرد. می‌خواهیم یک الگوریتم بازگشتی برای جابجایی این n دیسک بنویسیم.



برای 2 مهره: $A \to B, A \to C, B \to C$

برای 3 مهره: $A \xrightarrow{2} B, A \xrightarrow{1} C, B \xrightarrow{2} C$

برای n مهره: $A \xrightarrow{n-1} B, A \xrightarrow{1} C, B \xrightarrow{n-1} C$

```
Int tower(int n, peg A, peg B, peg C)
{                     // جابجایی مهره‌های A وB
 if(n == 1)
       move top disck on A to C
 else
       {
       tower(n-1,A,C,B);
       move top disck on A to c
       tower(n-1,B,A,C);
       }
 }
```

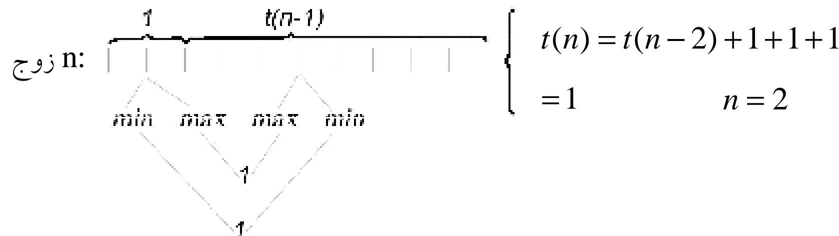
$$t(n)=t(n-1)+1+t(n-1) \quad \Rightarrow t(n)=\begin{cases}2t(n-1)+1 & n>0\\ 0 & n=0\end{cases}$$
$$t(0)=0$$

$$t(n)=2t(n-1)+1 \Rightarrow t(n)-2t(n-1)=1 \Rightarrow (x-2)(x-1)=0 \quad \begin{cases}t_1=1\\ t_2=2\end{cases}$$
$$t(n)=c_1(1)^n+c_2(2)^n$$

$$t(0) = 0 = c_1 + c_2 \quad \Rightarrow \quad c_2 = 1 \quad \Rightarrow t(n) = 2^n - 1$$
$$t(1) = 1 = c_1 + 2c_2 \qquad\qquad c_1 = -1$$

مثال: آرایه‌ای به طول n داریم می‌خواهیم عناصر max و min را در این آرایه محاسبه کنیم. حداقل تعداد مقایسه‌های لازم برای بدست آوردن عنصر max و min را بیابید.(طول آرایه را زوج فرض کنید و سپس برای طول فرد محاسبه کنید.)

زوج n:



$$\begin{cases} t(n) = t(n-2) + 1 + 1 + 1 \\ = 1 \qquad n = 2 \end{cases}$$

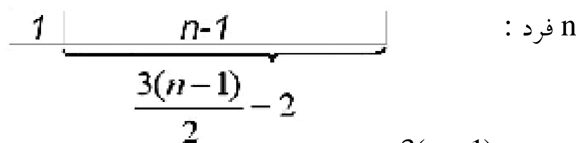
$$t(n) = t(n-2) + 3$$
$$n = 2k \Rightarrow t(2k) = t(2k-2) + 3 \Rightarrow t(2k) = t(2(k-1)) + 3 \Rightarrow g(k) = t(2k) = t(n)$$
$$g(k) = g(k-1) + 3 \Rightarrow g(k) = g(k-1) = 3 \times (1)^n$$
$$(x-1)(x-1) = 0 \qquad r_1 = 1 \quad r_2 = 1$$

$$g(k) = c_1(1)^4 + c_2 k(1)^4$$
$$g(k) = c_1 + c_2 k$$
$$g(n) = c_1 + c_2 \frac{n}{2}$$

زوج n :

$$1 = t(2) = c_1 + c_2 \quad \Rightarrow \quad c_2 = 3 \quad \Rightarrow t(n) = \frac{3n}{2} - 2$$
$$4 = t(4) = c_1 + 2c_2 \qquad\qquad c_1 = -2$$

فرد n :



$$t(n) = \frac{3(n-1)}{2} - 2 + 1 + 1 \quad = \frac{3n}{2} - \frac{3}{2}$$

1 اول مقایسه max با عنصر اول

1 دوم مقایسه min با عنصر اول

مثال: می‌خواهیم حاصلضرب $A_1 A_2 \ldots A_n$، D ماتریس را حساب کنیم. هر ماتریس $(A_i)_{d_{i-1} d_i}$ است. می‌خواهیم این ماتریس‌ها را پرانتزگذاری کنیم تا حاصل ضرب‌های مختلفی بدست آید. تعداد حالات مختلف پرانتزگذاری را بدست آورید.

$$M = A_1 A_2 \ldots A_i \; A_{i+1} \ldots A_n$$
$$t(i) \qquad t(n-i)$$

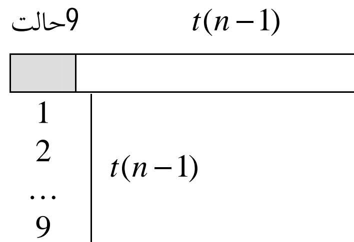
$$t(n) = \sum_{i=1}^{n-1} t(i) t(n-i) \qquad t(1) = 1 \qquad t(2) = 1 \qquad t(3) = 2$$

$$t(3)=\sum_{i=1}^{2} t(i)t(3-i)=t(1)t(2)+t(2)t(1)=1+1=2$$

$$t(n)=\frac{1}{n}\binom{2n-2}{n-1}$$ با کمک استقراء می‌توان ثابت کرد

مثال:فرض کنید کدی دهدهی با طول n زمانی معتبر است که تعداد صفرهای ظاهر شده در آن زوج باشد، می‌خواهیم رابطه‌ی بازگشتی را بدست آوریم که تمام کدهای به طول n که معتبر هستند را بدست آورد.

تعداد ارقام حالات معتبر $t(n)$است.



$b(n-1)$تعداد رشته‌های بطول $n-1$ است که تعداد صفرهایش فرد باشد.

$$t(n)=9t(n-1)+b(n-1)$$

10	10	…	10
1	2		$n-1$

$$t(n-1)+b(n-1)=10^{n-1}\Rightarrow b(n-1)=10^{n-1}-t(n-1)$$

$$t(n)=9t(n-1)+10^{n-1}-t(n-1)=8t(n-1)+10^{n-1}$$

مثال:

$$\begin{cases} t(n)=t(n-1)*t(n-1) \\ t(1)=1 \end{cases}$$

$$\log t(n)=\log t(n-1)+\log t(n-1)$$

$$\log t(n)=2\log t(n-1)$$

$$g(n)=\log t(n)\Rightarrow g(n)=2g(n-1)\Rightarrow g(n)-2g(n-1)=0$$

$$x^n-2x^{n-1}=0\Rightarrow x=2$$

$$g(n)=c_1 2^n \qquad \log_2 t(n)=c_1 2^n\Rightarrow t(n)=2^{c_1 2^n}$$

$$1=t(1)=2^{2c_1}\Rightarrow c_1=0 \qquad t(n)=1$$

$$2^0+2^1+\ldots+2^{n-1}=\frac{2^n-1}{2-1}=2^n-1 \qquad O(2^n)\in 2^{n-1}$$ ← زمان این تابع

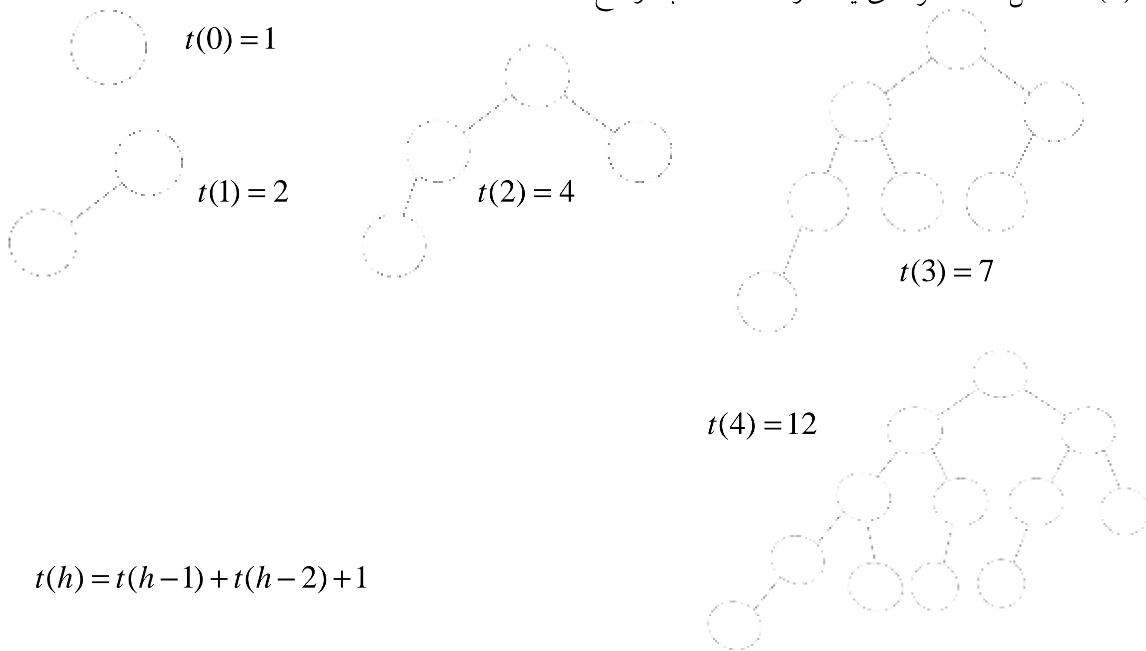
مثال: تعداد درخت‌های دودویی شامل n راس را بیابید.

$$t(n)=\sum_{i=0}^{n-1} t(i)t(n-1-i)$$

$$t(0)=1,t(1)=1,t(2)=2\Rightarrow t(n)=\frac{1}{n+1}\binom{2n}{n}$$

مثال: درخت AVL: درخت دودویی است که اختلاف زیر درختان چپ و راست هرگره آن 0،1یا1- باشد. می-خواهیم رابطه‌ای بازگشتی بدست آوریم که حداقل تعداد گره‌های یک درخت AVL به ارتفاعH را محاسبه نماید.

$T(h)$حداقل تعداد گره‌های یک درختAVL به ارتفاعh



$$t(h)=t(h-1)+t(h-2)+1$$

مثال: الگوریتم بازگشتی پیدا کنید که تمام زیر مجموعه‌های یک مجموعه‌ی n عضوی را پیدا کنید.

فرض می‌کنیم مسئله برای n-1 عضو جواب می‌دهد، آنرا به n عضو تعمیم می‌دهیم. عضو nام را از مجموعه حذف می‌کنیم و تمام زیر مجموعه‌های زیر مجموعه n-1 عضوی را دوباره می‌نویسیم. ودر بار دوم که چاپ می‌کنیم عنصر nام را به آن اضافه می‌کنیم.

تعداد حالات : $a_n = 2a_{n-1}$ $\{1,2,3\} \rightarrow \{\}\{1\}\{2\}\{1,2\}$

$\{3\}\{1,3\}\{2,3\}\{1,2,3\}$

مثال:

$$T(n)=\begin{cases} a & if\ \ n=0 \\ b & if\ \ n=1 \\ \dfrac{(1+T(n-1))}{T(n-2)} & else \end{cases}$$

$$T(0)=a$$
$$T(1)=b$$
$$T(2)=\frac{1+b}{a}$$
$$T(3)=\frac{1+\frac{1+b}{a}}{b}=\frac{a+b+1}{ab}$$

$$T(4)=\frac{1+\frac{a+b+1}{ab}}{\frac{1+b}{a}}=\frac{a(ab+a+b+1)}{(1+b)ab}=\frac{b(a+b)+1(a+1)}{(1+b)b}=\frac{a+1}{b}$$

$$T(5)=\frac{1+\frac{a+1}{b}}{\frac{a+b+1}{ab}}=\frac{\frac{a+b+1}{b}}{\frac{a+b+1}{ab}}=a$$

$$T(6)=\frac{1+a}{\frac{a+1}{b}}=b$$

$$T(n)=\begin{cases} a & n=0 \\ b & n=1 \\ \frac{1+b}{a} & n=2 \\ \frac{a+b+1}{ab} & n=3 \\ \frac{a+1}{b} & n=4 \\ T(n-\bmod 5) & else \end{cases}$$

مثال:

$$T(n)=\begin{cases} 0 & n=0 \\ \frac{1}{4-T(n-1)} & else \end{cases}$$

$$T(0)=0$$

$$T(1)=\frac{1}{4-T(0)}=\frac{1}{4}$$

$$T(2)=\frac{1}{4-\frac{1}{4}}=\frac{4}{4\times4-1}$$

$$T(3)=\frac{1}{4-\frac{15}{56}}=\frac{56}{4\times56-15}$$

$$T(n)=\frac{p_n}{q_n}$$

$$T(n+1)=\frac{p_{n+1}}{q_{n+1}}=\frac{q_n}{4q_n-q_{n-1}}$$

$$q_{n+1} = 4q_n - q_{n-1} \Rightarrow x^2 - 4x + 1 = 0 \quad \begin{cases} x_1 = 2 + \sqrt{3} \\ x_2 = 2 - \sqrt{3} \end{cases}$$

$$q_n = c_1(2+\sqrt{3})^n + c_2(2-\sqrt{3})^n$$

$$p_{n+1} = q_n \Rightarrow p_n = q_{n-1} = c_1(2+\sqrt{3})^{n-1} + c_2(2-\sqrt{3})^{n-1}$$

$$T(n) = \frac{p_n}{q_n} = \ldots$$

فصل سوم : **الگوریتم‌های حریصانه** (Greedy Algorithms) :

این دسته از الگوریتم‌ها برای بدست آوردن یک جواب از مسئله همواره سعی می‌کنند که ساده‌ترین و در عین حال پر ارزش‌ترین انتخاب‌ها را انجام دهند. انتخاب این گزینه‌ها در هر مرحله ممکن است باعث شود مسئله به بیراهه منجر شود و الگوریتم حریصانه بهترین جواب را برای مسئله اراده نکند یا اینکه ممکن است نتواند حتی یک جواب از مسئله را بدست آورد.

ساختار یک الگوریتم حریصانه مطابق شبه‌کد زیر است.

```
Function greedy(c: set): set;
{c is the set of condidates}
s ← 0 {we constract the solution in the set S}
while c ≠ 0 and NOT solution(S) do
        x ← select(c)
        c ← c − {x}
        if feasible(SU{x})then S ← SU{x}
if solution(S) then return S
else return "there is no solution"
```

بر اساس شبه‌کد مذکور هر الگوریتم حریصانه از اجزای زیر تشکیل شده است.

- **مجموعه C**: مجموعه‌ای ازتمام کاندیداهای ممکن که ممکن است به عنوان جواب مسئله انتخاب شوند.
- **مجموعه S**: مجموعه‌ایست که درنهایت قرار است به یک جواب از مسئله منتهی شود. در ابتدا تهی بوده و در نهایت در صورت وجود جواب به یک جواب از مسئله منجر می‌شود.
- **تابع Select** : ورودی این تابع مجموعه C و خروجی آن عنصری از مجموعه C می‌باشد.
- بدیهی است با انتخاب عنصری از C این عنصر از مجموعه‌ای از کاندیدها یعنی C باید حذف شود. تابع Select با توجه به نوع Greedy که در مسئله تعریف می‌شود عمل می‌نماید.
- **تابع Solution**: بررسی می‌کند که مجموعه S یک جواب از مسئله است یا خیر. ورودی آن مجموعه S و خروجی آن یک مقدار Boolean می‌باشد.
- **تابع Feasible**: ورودی آن مجموعه S و عنصر انتخاب شده از مجموعه C بوسیله‌ی تابع Select است.این تابع بررسی می‌کند که ببیند آیا امکان اضافه شدن عنصر به مجموعه S وجود دارد یا خیر. نتیجه‌ی این بررسی بصورت Boolean در خروجی تابع ظاهر می‌شود.

توجه کنید از آنجا که شبه‌کد تا زمانی تکرار می‌شود که یا به جواب برسد یا مجموعه C تهی شود، هیچگاه در حلقه‌ی بینهایت گیر نمی‌کند، مگر آنکه مجموعه‌ی انتخابی نا متناهی باشد.

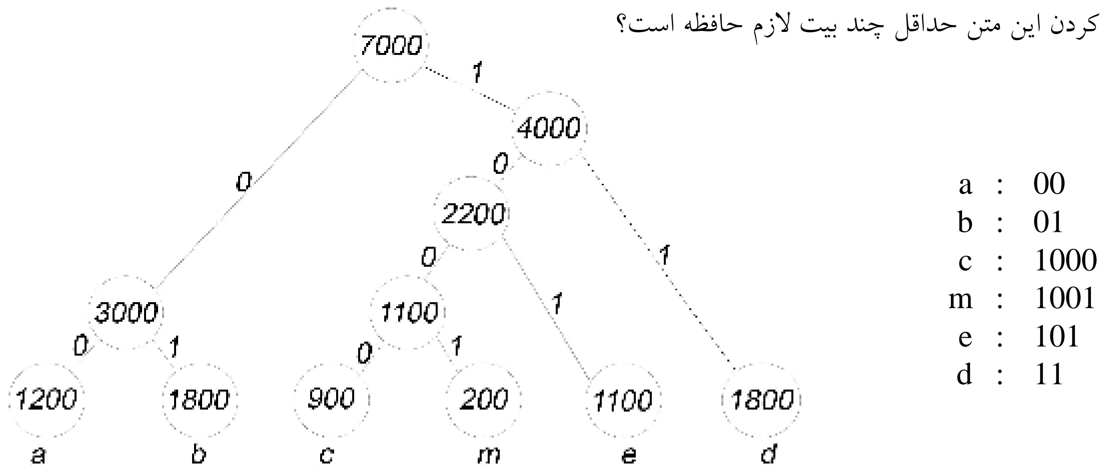
بررسی چند مثال:

الگوریتم هافمن(Huffman Algorithm)

یکی از سریعترین الگوریتم‌ها که می‌تواند یک فایل از علائم را zip کند این الگوریتم است. همچنین اجازه‌ی unzip نمودن فایل حاصل را نیز می‌دهد. برای این منظور فایل ورودی کاراکتر به کاراکتر خوانده می‌شود. فراوانی هرکدام از کاراکترهای بدست آمده در جدولی به صورت صعودی قرار می‌گیرد. از روی این جدول درختی ساخته

می‌شود که برگ‌های آن این کاراکترها باشد. حال هر دو برگی را که فرکانس کمتری دارد با هم merge می‌کنیم تا رأس جدید به عنوان ریشه آنها حاصل آید. فرکانس این نود برابر مجموع فرکانس دو نـود جمـع شـده مـی‌باشـد. اینکار را آنقدر ادامه می‌دهیم تا به ریشه‌ی درخت برسیم. سپس از ریشه‌ی درخت شـروع کـرده و بـرای فرزنـدان راست برچسب 1 و برای فرزندان چپ برچسب 0 را روی یالها در نظـر مـی‌گیـریم. بـه ایـن ترتیـب هـر یـک از کاراکترها دارای یک برچسب با کد باینری خواهد بود.

مثال: فرض کنید متنی شامل 7000 حرف الفبایی داریم که 6200تای آن حرف a، 1800تای آن حرفb، 900تـای آن حرف c، 1100تای آن e، 1800تای آن حرف d و 200 تای آن حرف m می‌باشد. می‌خواهیم بدانیم برای zip کردن این متن حداقل چند بیت لازم حافظه است؟



a : 00
b : 01
c : 1000
m : 1001
e : 101
d : 11

حداقل تعداد بیت‌های لازم : $2\times1200+2\times1800+4\times900+4\times200+3\times1100+2\times1800=17300$

بدون فشرده‌سازش : $7000\times8=56000$

الگوریتم‌های درخت پوشای مینیمال(Minimum Spanning Tree-MST):

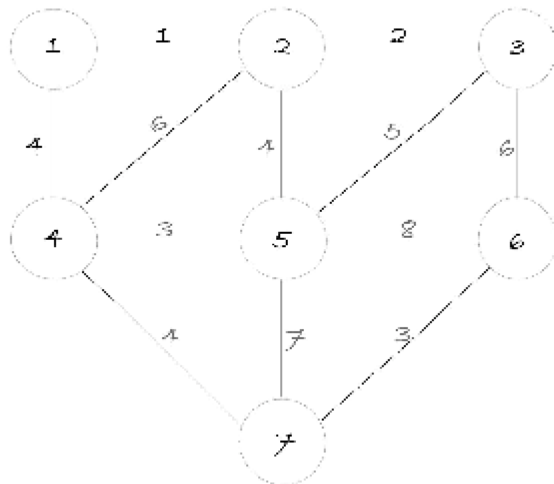
فرض کنید گراف $G=(N,A)$ یک گراف همبند باشد (یعنی بین هردو رأس متمایز آن یک مسیر وجود داشـته باشد) منظور از یک درخت پوشا از این گراف درختی است که شامل همه رئوس این گراف باشد ولی فقط بعضی از یال‌های آنرا دربر گیرد. منظور از درخت پوشای مینیمم (بـرای گـرف همبنـد وزن‌دار) درختـی اسـت کـه بـین درخت‌های پوشای آن گراف، مجموع وزن یال‌های آن، کمترین مقدار ممکن باشد.

بطور کلی دو الگوریتم برای درخت پوشای مینیمم وجود دارد که عبارتند از الگوریتم‌های Kruskal و Prim.

```
Function Kruskal(G = (N, A) : graph; length : A → R) : set of edges
        {initialization}
        sort A by increasing length
        n ← the number of nodes in N
        T ← 0 {will contain the edges of the minimum spanning tree}
        initialize n sets, each containing a different member of N
        {greedy loop}
        repeat
                e ← {u, v} ← shortest edge not yet considered
```

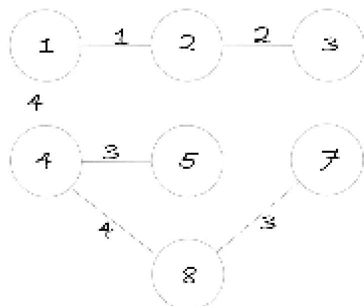
$$ucomp \leftarrow find(u)$$
$$vcomp \leftarrow find(v)$$
$$if \;\; ucomp \neq vcomp \;\; then$$
$$merge(ucomp, vcomp)$$
$$T \leftarrow T \cup \{e\}$$
$$until \;\; T \;\; contains \;\; n-1 \;\; edges$$
$$return \;\; T$$

مثال:



Step	Edge Considered	Connected Component
Initialization	---------	{1}{2}{3}{4}{5}{6}{7}
1	{1,2}	{1,2}{3}{4}{5}{6}{7}
2	{2,3}	{1,2,3}{4}{5}{6}{7}
3	{4,5}	{1,2,3}{4,5}{6}{7}
4	{6,7}	{1,2,3}{4,5}{6,7}
5	{1,4}	{1,2,3,4,5}{6,7}
6	{2,5}	*rejected*
7	{4,7}	{1,2,3,4,5,6,7}

نحوه‌ی کار الگوریتم Kruskal به این صورت است که یک جنگل از درخت‌هارا به ترتیب با هم ادغام می‌کند تا به یک درخت واحد برسد.



الگوریتم prim:

$Function\ \ prim(G=(N,A):graph; Length: A \rightarrow R^{+}): set\ \ of\ \ edges$

$\{initialization\}$

$T \leftarrow 0$

$B \leftarrow \{an\ \ arbitray\ \ member\ \ of\ \ N\}$

$while\ \ B \neq N\ \ do$

  $find\ \ e=\{u,v\}\ \ of\ \ \ minimum\ \ length\ \ such\ \ that\ \ u \in B\ \ and\ \ v \in N-B$

  $T \leftarrow T \cup \{e\}$

  $B \leftarrow B \cup \{u\}$

$return\ \ T$

Step	{u,v}	B
initialization	-----	{1}
1	{1,2}	{1,2}
2	{2,3}	{1,2,3}
3	{1,4}	{1,2,3,4}
4	{4,5}	{1,2,3,4,5}
5	{4,7}	{1,2,3,4,5,7}
6	{7,6}	{1,2,3,4,5,6,7}

- ممکن است درخت‌هایی که الگوریتم مذکور تولید می‌کنند، از لحاظ شکل ظاهری متفاوت باشند، ولی وزن همه‌ی درخت‌ها یکسان است.
- مرتبه‌ی زمانی الگوریتم $prim$ برابر $\Theta(n^2)$ است. (حلقه‌ی $while \leftarrow n$ دفعه و عمل یافتن از میان لبه‌های متصل به یک مجموعه دور خاص n دفعه اتفاق می‌افتد؛ که در مجموع برابر n^2 دفعه می‌شود).
- مرتبه‌ی زمانی الگوریتم kruskal برابر $\Theta(e \log e)$ می‌باشد.(این زمان مربوط به مرتب‌سازی لبه‌ها می-باشد).
- اگر گراف ورودی شلوغ باشد، یعنی تعداد یال‌های گراف زیاد باشد $e \leq \frac{n(n-1)}{2}$ است، در این صورت زمان الگوریتم $kruskal$ برابر $\Theta(n^2 \log n^2) = \Theta(n^2 \log n)$ است. در حالیکه زمان الگوریتم $prim$ برابر $\Theta(n^2)$ می‌باشد، پس الگوریتم $prim$ بهتر است. اگر گراف ورودی اسپارس(خلوت) باشد، یعنی تعداد یال‌ها کم باشد $(e \leftarrow n-1)$ زمان الگوریتم $kruskal$ برابر $\Theta(n \log n)$ و زمان اجرای الگوریتم $prim$ برابر $\Theta(n^2)$ می‌باشد پس الگوریتم $kruskal$ دراین حالت بهتر است.

الگوریتم دایکسترا (Dijkstra):

این الگوریتم بصورت حریصانه عمل نموده و بدنبال یافتن کوتاه‌ترین مسیر ممکن بین یک نود و هر نود دلخواه دیگر، در یک گراف وزن‌دار و جهت‌دار می‌باشد. ورودی این الگوریتم همیشه یک ماتریس دو بعدی است که همان ماتریس وزن گراف است.

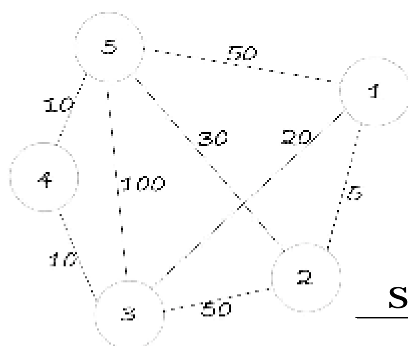
$$Lij \begin{cases} (i,j) \text{ وزن‌یال} & (i,j) \in E \\ 0 & i = j \\ +\infty & else \end{cases}$$

$Function\ Dijkstra(L[1..n,1..n]) : array[2..n]$
$array\ D[2..n]$
$\{initialization\}$
$C\{2,3,..,n\}$
$for\ i \leftarrow 2\ to\ n\ do$
    $D[i] \leftarrow l[100]$
$p[i] \leftarrow 0$**
$\{greedy\ loop\}$
$repeat\ n-2\ times$
    $v \leftarrow some\ element\ of\ C\ minimizing\ D[v]$
    $C \leftarrow C - \{v\}$
    $for\ each\ w \in C\ do$
        $D[w] \leftarrow \min\{D[w], D[v] - L[v,w]\}$**
$return\ D, L$

این الگوریتم مسیر را به ما و در صورتی که بخواهیم مسیر را بدهد بجای قسمت ** در الگوریتم، شبه کد روبـرو را قرار می‌دهیم.

$if\ \ D[w] > D[v] + L[v,w]$
    $D[w] \leftarrow D[v] + L[v,w]$
    $P[w] \leftarrow v$

مثال:



Step	V	C	D
Initialization		{2,3,4,5}	{50,30,100,10}
1	5	{2,3,4}	{50,30,20,10}
2	4	{2,3}	{40,30,20,10}
3	3	{2}	{35,30,20,10}

$$\begin{bmatrix} 0 & 50 & 30 & 100 & 10 \\ \infty & 0 & \infty & \infty & \infty \\ \infty & 5 & 0 & \infty & \infty \\ \infty & 20 & 50 & 0 & \infty \\ \infty & \infty & \infty & 10 & 0 \end{bmatrix}$$

$p:$

3	0	5	0
2	3	4	5

مرتبه‌ی زمانی الگوریتم دایکسترا $\Theta(n^2)$ است.

$$1+2+\ldots+(n-1)=\frac{(n-1)(n-2)}{2}\in\Theta(n^2)$$

مسئله‌ی زمان‌بندی (Scheduling):

مسئله‌ی زمان‌بندی یکی از پر اهمیت‌ترین مسائل در تولید سیستم عامل‌ها می‌باشد. زمان‌بندی خود به دو دسته تقسیم می‌شود.

Scheduling { Simple / With deadline

یک مثال از زمان‌بندی ساده : فرض کنید یک Server داریم که می‌خواهد به p مشتری سرویس دهد. فرض براین است که زمان لازم برای سرویس هرمشتری t_i باشد. در صورتی که تعداد مشتریان ثابت باشد نحوه سرویس دهی چگونه باشد تا متوسط زمان انتظار آنها مینیمم باشد.

$T_1=t_1$

$T_2=t_1+t_2$

$\vdots$

$T_i=\sum_{k=1}^{i}t_k$

$\vdots$

زمان لازم برای مشتری kام

متوسط زمان انتظار برای سرویس هر یک از n مشتری $=\frac{\sum_{i=1}^{n}T_i}{n}$

مثال : $3(n=3)$ مشتری داریم: $t_3=6, t_2=5, t_1=8$ می‌باشد.

$1\ 2\ 3 \quad \Rightarrow T=8+(8+5)+(8+5+6)$

$1\ 3\ 2 \quad \Rightarrow T=8+(8+6)+(8+6+5)$

$2\ 1\ 3 \quad \Rightarrow T=5+(5+8)+(5+8+6)$

$2\ 3\ 1 \quad \Rightarrow T=5+(5+6)+(5+6+8)\Rightarrow optimal$

$3\ 1\ 2 \quad \Rightarrow T=6+(6+8)+(6+8+5)$

$3\ 2\ 1 \quad \Rightarrow T=6+(6+5)+(6+5+8)$

برای این نوع زمان‌بندی راه‌حل مناسب آن است که ابتدا داده‌ها را به ترتیب صعودی بر حسب زمان مورد نیازشان مرتب کنیم، سپس از سرلیست به آنها سرویس بدهیم(مرتبه‌ی زمانی فقط زمان لازم برای مرتب سازی $O(n\log n)$ می‌شود).

زمان‌بندی مهلت دار (Schedding with deadlines):

فرض کنید n کار (job) داریم که هر یک برای انجامشان به یک واحد زمانی نیاز دارند. هر یک از این کارها دارای یک مهلت زمانی مانند d_i می‌باشند، یعنی کار iام اگر قبل از زمان d_i انجام شود، سودی به اندازه‌ی $g_i \geq 0$ به ما می‌رساند. در غیر این صورت انجام این‌کار هیچ سودی نخواهد داشت. می‌خواهیم بدانیم حداکثر سودی که می‌توان از اجرای چند job از این n، job داشت، چه مقدار است؟

i	1	2	3	4
g_i	50	10	15	30
d_i	2	1	2	1

حالت‌های ممکن:

Sequence	**Profit**
1	50
2	10
3	15
4	30
1,3	65
2,1	60
2,3	25
3,1	65
4,1	80 $\rightarrow optimal$
4,3	45

$Function\ Sequence(d[0..n]) : k, array[1..k]$

$array\ \ j[0..n]$

$d[0] \leftarrow j[0] \leftarrow 0$

$k \leftarrow j[1] \leftarrow 1\ \{job\ 1\ is\ always\ choosen\}$

$\{greedy\ loop\}$

$for\ i \leftarrow 2\ to\ n\ do$

  $r \leftarrow k$

  $while\ \ d[j[r]] > \max(d[i], r) do \qquad r \leftarrow r-1$

  $if\ \ d[i] > r\ then$

    $for\ m \leftarrow k\ \ step -1\ to\ r+1\ do$

      $j[m+1] \leftarrow j[m]$

    $j[r+1] \leftarrow i$

    $k \leftarrow k+1$

$return\ k, j[1..k]$

k: در هر مرحله شماره‌ی کاری که انجام می‌شود.

i: جهت بررسی کارهای موجودکه بترتیب نزولی مرتب شده‌اند.

i	1	2	3	4	5	6
g_i	20	15	10	7	5	3

d_i 3 1 1 3 1 3

r: از خانه‌ی kام به قبل اگر محدودیت زمانی اجازه می‌دهد، انجام آن‌ها را به تأخیر بیاندازیم.

d: لیست کارهایی که انجام می‌شود.

0	1					

j

0 1 2 3 4 5 6
0 2 1
0 2 1 4
unchanged

مرتبه‌ی زمانی این الگوریتم $O(n^2)$ می‌باشد.

ایـن الگـوریتم عناصـری کـه دارای بیـشترین سـود هـستند را در ابتـدای صـف قـرار داده، سـپس بـا توجـه بـه *deadline* آن‌ها اجازه می‌دهدتا جایی‌که ممکن است جابجا شوند و بعد از عناصر با *deadline* کمتر قرار بگیرند.

مسئله کوله پشتی (knapsack problem) :

فرض کنید کوله پشتی داریم که می‌تواند وزن w را تحمل کند. می‌خواهیم این کوله پشتی را با اشیاء $n,...,2,1$ که بترتیب وزن آن‌ها $w_1, w_2, ..., w_n$ است و ارزش آن‌ها $v_1, v_2, ..., v_n$ است پرکنیم. بطوریکه هدف زیر تامین شود.

$\sum_{i=1}^{n} x_i v_i$ ماکزیمم باشد و $\sum_{i=1}^{n} x_i w_i \le w$ باشد که در آن $\forall i \ x_i \in [0,1]$

منطقی‌تر است که ابتدا اشیائی را بریزیم که ارزش بیشتر و وزن کمتر دارند. پس بصورت زیر عمل می‌کنیم.

نسبت $\frac{v_i}{w_i}$ را برای تمام اشیاء بدست می‌آوریم. اشیاء را بترتیب نزولی مرتب می‌کنیم. از سرلیست یکی‌یکی اشـیاء را انتخاب می‌نمائیم.

تذکر: این مسئله را که در آن می‌توان کسری از یک شیء را نیز داشت مسئله کوله‌پشتی جزئی می‌نامند. در حالی کـه همه اشیاء باید یا همه آن انتخاب شوند و یا کـلاً انتخـاب نـشوند را مـسئله‌ی کولـه‌پـشتی L مـی‌نامنـد کـه در آن $x \in \{0,1\}$

```
Function Knapsack(w[1..n], v[1..n], w) : array[1..n]
{initialization}
for i = 1 to n do
        x[i] ← 0
weight ← 0
{greedy loop}
while weight < w do
        i ← the best remaining object
        if weight + w[i] ≤ w then
                x[i] ← 1
                weight ← weight + w[i]
        else
                x[i] ← (w − weight) / w[i]
                weight ← w
```

return x

مرتب‌سازی   حلقه

مرتبه‌ی زمانی این الگوریتم $O(n\log n) \leftarrow O(n) + O(n\log n)$

مثال :

$n = 5 \quad w = 100$

w	10	20	30	40	50
v	20	30	66	40	60
v/w	2	1/5	2/2	1	1/2

Step	X					i	Weight	Value
Initialization	0	0	0	0	0	-	0	0
1	0	0	1	0	0	3	30	66
2	1	0	1	0	0	1	40	86
3	1	1	1	0	0	2	60	116
4	1	1	1	0	$\frac{100-60}{50}$	5	100	164
								$\frac{8}{10}\times 60+116$

فصل چهارم :تکنیک تقسیم و حل (Divide and Conquer Technique):

D&C: یک تکنیک طراحی الگوریتم‌ها است. در این نوع الگوریتم مسئله‌ی اصلی به زیر مشئله‌هایی تقسیم می‌شود که دقیقاً شبیه مسئله‌ی اصلی هستند ولی از نظر ابعاد کوچکترند و علاوه‌برآن تعداد آن‌ها بیشتراز مسئله‌ی اصلی می‌باشد. حال به حل هریک از این زیر مسئله‌ها و ترکیب جواب آن‌ها با یکدیگر به جواب اصلی دست می‌یابیم. ساختار کلی یک الگوریتم D&C مطابق شبه کد زیر است.

```
Function DC(k)
if  x is sufficiently small as simple then
        return adhoc(x) → a simple  algorithm capable of  solving the problem
decompose x into smaller instances x_1, x_2, ..., x_p
for i ← 1 to p do
        j_i ← DC(x_i)
recombine the j_i's to obtain a solving j for x_i
return j
```

نکته : تکنیک D&C یک متد برنامه نویسی $Top-Down$ است.

مثال : یافتن یک عنصر در یک آرایه مرتب.

I. جستجوی خطی: $O(n)$ برای آرایه مرتب اصلاً بدرد نمی‌خورد.

```
Function Sequantial(T[1..n].x)
for i ← 1 to n do
        if T[i] = x then
                return i
return n+1
```

II. جستجوی باینری:

```
Function binsearch(T[1..n], x)
if n = 0 OR x > T[n] then
        return n+1
else
        return binrec(T[1..n], x)

Function binrec(T[1..j], x)
if i = j then
        return i
k ← (i+j)/2
if x ≤ T[k] then
        return binrec(T[i..k], x)
else
        return binrec(T[k+1, j], x)
```

Binary Search for $x = 12$ in $T[1..11]$

-5	-2	0	3	8	8	9	12	12	26	31
1	2	3	4	5	6	7	8	9	10	11
i					k					j
						i		k		J
						i	k	J		
						ik	J			
							ij			

$Function\ biniter(T[1..n], x)$

$\{Iterative\ binary\ search\ for\ x\ in\ array\ T\}$

$if\ \ x > T[n]\ then$

$\quad return\ n+1$

$i \leftarrow 1$

$j \leftarrow n$

$while\ i < j\ do$

$\quad k \leftarrow (i+j)/2$

$\quad if\ \ x \le T[k]\ then$

$\quad\quad j \leftarrow k$

$\quad else$

$\quad\quad i \leftarrow k+1$

$return\ i$

$$T(n) = T(\frac{n}{2}) + \Theta(1) \Rightarrow T(n) = O(\log n)$$

مثال: Merge sort :

$procedure\ merge(U[1..m+1], V[1..n+1], T[1..m+n])$

$\{merges\ sorted\ array\ U[1..m]\ and\ V[1..n]\ into\ T[1..m+n];$

$U[m+1]\ and\ V[n+1]\ are\ used\ as\ sentinels\}$

$i, j \leftarrow 1$

$U[M+1], V[n+1] \leftarrow \infty$

$for\ k \leftarrow 1\ to\ m+n\ do$

$\quad if\ \ U[i] < V[j]\ then$

$\quad\quad T[k] \leftarrow U[i]$

$\quad\quad i \leftarrow i+1$

$\quad else$

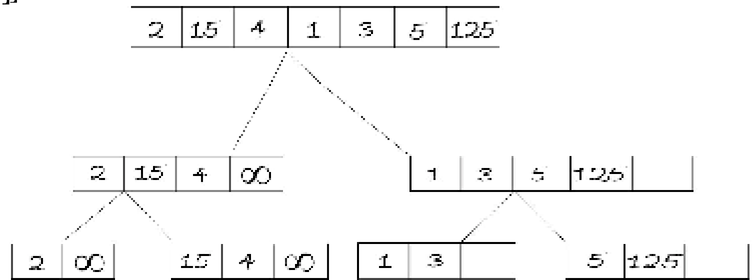
$\quad\quad T[k] \leftarrow V[j]$

$\quad\quad j \leftarrow j+1$

زمان اجرا $\Theta(m+n)$

$procedure$ mergesort(T[1..n])
if n *is sufficiently small then*
  $sort(T)$
else
  $array\ U[1..1+[n/2]], V[1..1+[n/2]]$
  $U[1..[n/2]] \leftarrow T[1..[n/2]]$
  $V[1..[n/2]] \leftarrow T[1+[n/2]..n]$
  $merge\ sort(U[1..[n/2]])$
  $merge\ sort(V[1..[n/2]])$
  $merge\ sort(U,V,T)$

$T(n) = 2T(n/2) + \Theta(n) \Rightarrow \Theta(n \log n)$



الگوریتم Quick Sort :

در این الگوریتم یک عنصر به عنوان محور درنظر گرفته می‌شود و سعی می‌شود تا آرایه اصلی به گونه‌ای به دو قسمت تقسیم شود که اعضاء قبل از عنصر محوری از آن کوچکتر و اعضاء بعد از سمت راست آن از عنصر محوری بزرگ‌تر باشند.

$T(n) = C_1(1)^n + C_2 n + C_3 n^2 \Rightarrow T(n) \in \Theta(n^2)$

آنالیز الگوریتم در حالت متوسط:

$$T(n) = \frac{1}{n}\sum_{i=1}^{n}(\Theta(n) + T(L-1) + T(n-L))$$

$$T(n) = \underbrace{\Theta(n)}_{c(n+1)} + \frac{1}{n}\sum_{L=1}^{n}(T(L-1) + T(n-L))$$

از بسط رابطه داریم.

$$nT(n) = nc(n+1) + \sum_{k=1}^{n}(T(L-1) + T(n-L))$$

$$nT(n) = cn(n+1) + 2(T(0) + T(1) + T(2) + ... + T(n-1))$$

به تبدیل n به $n-1$

$$(n-1)T(n) = cn(n-1) + 2(T(0) + T(1) + ... + T(n-2))$$

$$\Rightarrow nT(n) = (n+1)T(n-1) + 2cn$$

تمرین:

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

جایگذاری

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

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

$$= \frac{T(1)}{2} + 2c\sum_{k=3}^{n+1}\frac{1}{k}$$

$$T(1)=0 \Rightarrow T(2)=2c\sum_{k=3}^{n+1}\frac{1}{k} \le 2c\int_{2}^{n+1}\frac{1}{k}dx = 2c(\log_e^{n+1}-\log_e^2)$$
$$\Rightarrow T(n) \le 2c(n+1)(\log_e^{n+1}-\log_e^2) \Rightarrow T(n)=O(n\log_e^n)$$

آنالیز الگوریتم در بهترین حالت:

در این هر دفعه داده‌ها نصف می‌شود و مانند merge sort عمل می‌نماید و مرتبه‌ی زمانی آن $\Theta(n\log n)$ خواهد بود.

مثال: ضرب ماتریسهابه روش استراسن

ضرب در حالت عادی.

$$A_{pq}B_{qr}=C_{pr}=(C_{ij})_{pr}$$
$$C_{ij}=\sum_{k=1}^{q}a_{ik}b_{kj}$$

```
for i = 1 to p do
    for j = 1 to r do
        c_ij = 0
        for k = 1 to q do
            c_ij += a_ik b_kj
```

در حالتی که دو ماتریس هر دو مربعی $n\times n$ باشند، واضح است که زمان ضرب $O(n^3)$ خواهد بود. استراسن به دنبال این بود که این زمان را کمتر کند. روش او فقط روی ماتریس‌های مربع با توانی از 2 $(n=2)$ کار می‌کرد. مثلاً با $n=2$ تعداد ضرب‌های از 8 به 7 کاهش می‌یابد.

در این ساختار جدید زمان ضرب به میزان $\Theta(n^{\log_2^7})$ برای ماتریس‌های مربعی $n\times n\,(n=2^k)$ کاهش یافت. استراسن این کار را به روش بلاکی کردن انجام داد.

تکنیک بلاکی کردن:

$$\begin{array}{c} a_{11} \qquad a_{12} \\ \left[\begin{array}{cc|cc} O & O & O & O \\ O & O & O & O \\ \hline O & O & O & O \\ O & O & O & O \end{array}\right] \\ a_{21} \qquad a_{22} \end{array} \quad (a_{ij})_{\frac{n}{2}\frac{n}{2}}$$

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

استراسن معین کرد که اگر فرض کنیم

$$m_1=(a_{11}+a_{22})(b_{11}+b_{22})$$
$$m_2=(a_{21}+a_{22})b_{11}$$
$$m_3=a_{11}(b_{12}-b_{22})$$
$$m_4=a_{22}(b_{21}-b_{11})$$

$m_5 = (a_{11} + a_{12})b_{22}$
$m_6 = (a_{21} - a_{11})(b_{11} + b_{12})$
$m_7 = (a_{12} - a_{22})(b_{21} + b_{22})$

حال ضرب Cبصورت زیر بدست می‌آید.

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

ضرب‌ها جمع‌ها

$$T(n) = 7T(\frac{n}{2}) + 18T(\frac{n}{2})^2$$

$$T(n) = 7T(\frac{n}{2}) + \frac{9}{2}Tn^2$$

$$\Rightarrow T(n) \in \Theta(n^{\log_2^7})$$

```
procedure strassen(int n, n×n matrix A, n×n matrix B, n×n matrix c)
{
if (n ≤ threshold)
        compute C = A×B using standard algorithm
else
        {partition A into four sub matrix
        A11, A12, A21, A22
        partition B into four sub matrix
        B11, B12, B21, B22
        compute C = A×B using the strassen method;
        }
}
```

فصل پنجم : برنامه نویسی پویا (Dynamic programming) :

در این تکنیک برنامه نویسی، حل مسئله بصورت پایین به بالا (Bottom up) صورت می‌پذیرد. یعنی ابتدا مسئله در حالت ساده‌تر یا کوچک‌تر حل شده و سپس این راه‌حل‌ها را در حافظه‌ای ذخیره می‌نماییم و در مراحل بعد، از این پاسخ‌ها استفاده کرده و سعی می‌کنیم به یک جواب از مسئله با عبور از جواب‌های ابتدایی دست یابیم. برای حل یک مسئله به روش Dynamic programming اولین قدم آنست که یک فرمول بازگشتی برای آن بیابیم و سپس با توجه به فرمول بازگشتی مربوطه، یک table یا یک آرایه چند بعدی ایجاد نماییم که جواب‌های قبلی در آن ذخیره شده‌اند. سپس جواب‌های اصلی را به این شیوه بیابیم.

نکته : ایده برنامه نویسی پویا حذف عملیات تکرایی و بالا بردن زمان اجرای برنامه‌ها است.

مثال: محاسبه‌ی ضرایب بسط دو جمله‌ای $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$

می‌دانیم که ضرایب بسط دو جمله‌ای از رابطه‌ی پاسکال به دست می‌آید، که رابطه‌ی بازگشتی آن بصورت زیر است.

$$\binom{n}{k} = \begin{cases} 1 & if\ \ k=0\ \ OR\ \ k=n \\ \binom{n-1}{k}+\binom{n-1}{k-1} & if\ \ 0<k<n \\ 0 & otherwise \end{cases}$$

اگر این رابطه‌ی بازگشتی را بصورت D&C پیاده سازی کنیم داریم.

$Function \quad C(n,k)$

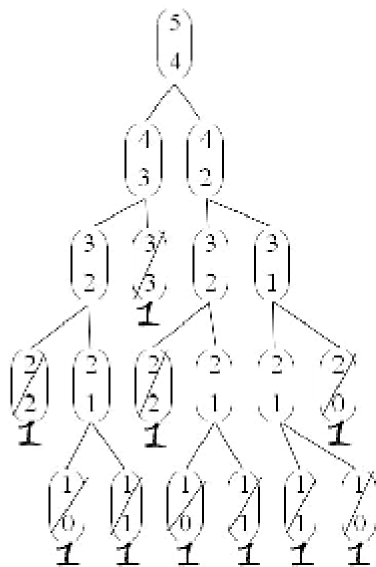
$if\ \ k=0\ \ OR\ \ k=n\ then$

$\qquad return\ 1$

$else$

$\qquad return\ C(n-1,k-1)+C(n-1,k)$`

اشکال این روش آن است که اگر درخت محاسبه‌ی رابطه‌ی بازگشتی آنرا مثلاً برای ترکیب $\binom{5}{3}$ استفاده کنیم مشاهده می‌شود که خیلی از گره‌های آن تکراری هستند و محاسبات تکراری زیادی را داریم.



	0	1	2	3	…	$k-1$	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
M							
$n-1$							
n							

$$C[n,k] = C[n-1,k-1] + C[n-1,k]$$
$$C[i,0] = 1$$
$$C[i,i] = 1$$

```
for i ← 0 to n do
    C[i,0] = 1
for j ← k to k do
    C[j, j] = 1
for i ← 2 to n do
    for j ← 1 to i−1 do
        C[i, j] ← C[i−1, j−1] + C[i−1, j]
return c[n,k]
```

سری جهانی(The world series):

فرض کنید دو تیم A وB آنقدر با هم مسابقه می‌دهندکه یک تیم به n پیروزی دست یابد. تیمی که برای اولین بار به n پیروزی دست یابد برنده است. توجه شود که نتیجه هر بازی یا پیروزی است یا شکست. بنابراین این دو تیم باید حداکثر $2n-1$بار و حد اقل n بار بازی کنند. اگر احتمال برد در هر بازی برای تیم A، p و برای تیم B، q باشد($p+q=1$) بعلاوه احتمال برد هر تیم در هر بازی مستقل از بازی دیگر باشد و اگر $P(i,j)$ احتمال برد تیم A باشد مشروط به اینکه تیم A نیازمند i پیروزی و تیم B نیازمند به j پیروزی باشد داریم:

$1 \le j \le n$ ، $P\ (0, j) = 1$

$1 \le i \le n$ ، $P\ (i,0) = 0$

$P(n,n)$ نقطه شروع بازی

$p(0,0)$ امکان پذیر نیست

$$P(i, j) = P * P(i-1, j) + q * P(i, j-1) \qquad i \ge 1, j \ge 1$$

احتمال اینکه B ببرد احتمال اینکه A ببرد

پیاده سازی بصورت D&C:

```
Function P(i, j)
if  i = 0  then
        return 1
else if  j = 0  then
        return 0
else
        return  p * p(i-1, j) + q * p(i, j-1)
```

زمان اجرا:

$$k = \begin{cases} i + j & i, j \ne 0 \\ 1 & i \text{ یا } j = 0 \end{cases} \qquad T(k) = \begin{cases} C & k = 1 \\ 2T(k-1) + h(k) & k \ne 1 \end{cases}$$

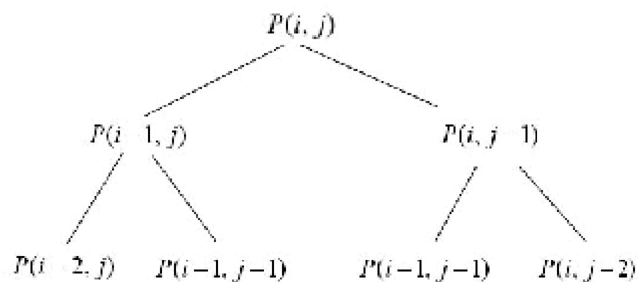
$T(k) = 2T(k-1) + h(n)$

$(x^k - 2x^{k-1})(x-1)^m = 0$

$T(k) = C_1 2^k + \ldots$

$T(i+j) = 2C^{i+j}$

کران بالا : $P(n,n) = O(2^{2n}) = O(4^n)$



درخت دودویی است و ارتفاع آن $2n-1$ میباشد

$Function\ \ Series(n, p)$
$array\ \ P[0..n,0..n]$
$q \leftarrow 1-p$
$\{fill\ \ from\ top\ left\ to\ main\ diagonal\}$
$for\ S \leftarrow 1\ to\ n\ do$

$P[0,s] \leftarrow 1;$
$P[s,0] \leftarrow 0;$
$for\ k \leftarrow 1\ to\ s-1\ do$

دربالای قطر اصلی $P[k,s-k] \leftarrow p*p[k-1,s-k]+q*p[k,s-k-1]$

$\{fill\ \ from\ below\ main\ diagonal\ to\ bottom\ rigth\}$
$for\ s \leftarrow 1\ to\ n\ do$

$for\ k \leftarrow 1\ to\ n-s\ do$

$P[s+k,n-k] \leftarrow p*p[s+k-1,n-k]+q*p[s+k,n-k-1]$

$return\ \ p[n,n]$

	0	1	2	3	…	j-1	j	…	n
0	0	1	1	1					
1	0	p	$p+pq$						
2	0	p^2	$p^2+pq=pq^2$						
M	0								
n	**M**								

زمان اجرا : $\Theta(n^2)$است. یکبار بالای قطر اصلی و یکبار پایین قطر اصلی را نگاه می‌کنیم و با هم جمع می‌کنیم. در این صورت زمان اجرای الگوریتم بدست می‌آید.

بالای قطر اصلی $\rightarrow i+j=s$ $\qquad \Theta(n^2/2)$

$\Rightarrow \Theta(n^2)$

پایین قطر اصلی $\rightarrow i+j=s+k$ $\qquad \Theta(n^2/2)$

$for\ i \leftarrow 1\ to\ n\ do$

$P(i,0)=0$
$P(0,i)=1$ $\Theta(n^2+n)$ زمان

$for\ i \leftarrow 1\ to\ n\ do$

$P[i,j]=P*P[i-1,j]+q*p[i,j-1]$

الگوریتم مسیریابی فلوید (Floyd) :

فرض کنید یک گراف جهت‌دار داریم.که هر یال آن با عددی نا منفی برچسب خورده است. می‌خواهیم طول کوتاهترین مسیر بین هردویال متمایز را بیابیم.(منظور از مسیر دنباله‌ی متناهی از رئوس است که اولاً هیچ رأس تکراری در آن نباشد و علاوه برآن بین هر دو رأس متوالی از این دنباله، یک یال در جهت مسیر وجود دارد). برای این منظور، گراف را بوسیله‌ی ماتریس وزنش(ماتریس مجاورتی) معرفی می‌کنیم سپس بوسیله‌ی رأس‌های مختلف این گراف را فیلتر می‌کنیم(یعنی بواسطه‌ی آن رأس از مبداء به مقصد می‌رویم). یعنی بررسی می‌کنیم بوسیله‌ی رأس 1 مسیر کوتاهتر می‌شود. سپس همین کار را برای رأس 2 و بقیه‌ی رئوس انجام می‌دهیم، و در نهایت ماتریس آخر ماتریسی است که کوتاهترین فاصله بین هر دو رأس گراف را به ما می‌دهد.

- تکنیک این الگوریتم، شبیه الگوریتم دایکسترا می‌باشد.

$Function\ Floyd(L[1..n,1..n])array[1..n,1..n])$
$array\ D[1..n,1..n]$
$D \leftarrow L$
$for\ k \leftarrow 1\ to\ n\ do$

$for\ i \leftarrow 1\ to\ n\ do$

$for\ j \leftarrow 1\ to\ n\ do$

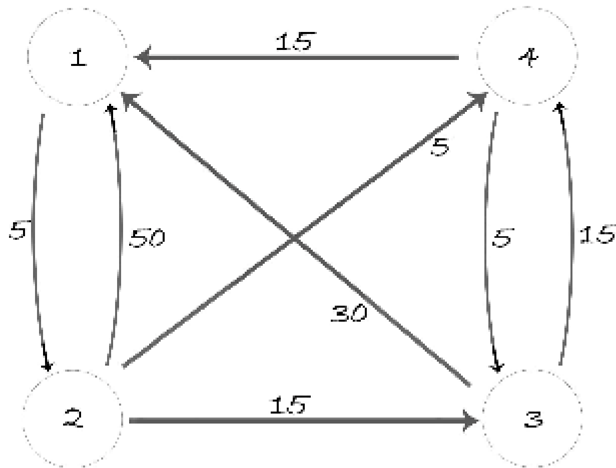
$D[i,j] \leftarrow \min(D[i,j],D[i,k]+D[k,j]);$
$if\ D[i,j] > D[i,k]+D[k,j]$
$D[i,j] \leftarrow D[i,k]+D[k,j]$
$p[i,j] \leftarrow k$

$return\ D,P$

این الگوریتم Dynamic است زیرا مرحله به مرحله(table به table پیش می‌رود.)

نکته : مرتبه‌ی زمانی برای الگوریتم فلوید $\Theta(n^3)$ است.

$$D_0 = L = \begin{bmatrix} 0 & \infty & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{bmatrix} \quad D_1 = L = \begin{bmatrix} 0 & \infty & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \quad D_2 = L = \begin{bmatrix} 0 & 5 & 20 & 10 \\ 30 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix}$$



$$D_3 = L = \begin{bmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix}$$

$$D_4 = L = \begin{bmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix}$$

- همیشه تکنیک DP بهتر از تکنیک D&C نمی‌باشد و گاهی اوقات در بعضی مسائل تکنیک D&C بهتر عمل می‌کند.

مثال : مسئله به توان رسانی :

الف) به روش D&C

$$a^n = a \times a \times \overset{\overleftrightarrow{n}}{a} \times \ldots \times a$$

$$X_n = \begin{cases} X^{n/2} X^{n/2} X & \text{اگر } n \text{ فرد باشد} \\ X^{n/2} X^{n/2} & \text{اگر } n \text{ زوج باشد} \end{cases}$$

```
Float xton(float x,int n)
{if (n==1)
        return x
j = xton(x, n/2)
if (n == n/2 × 2)
        return j × j
else
return j × j × x
}
```

$$T(n) = \begin{cases} C & n=1 \\ T(n/2)+3 & n>1 \end{cases} \Rightarrow T(n) = \Theta(\log n)$$

ب) به روش DP: جدولی در نظر گرفته می‌شود که باید کامل پر شود پس $\Theta(n^2)$

ضرب زنجیره‌ای ماتریس‌ها (Matrix-Chain-Multiplication):

فرض کنید می‌خواهیم حاصل $A = A_1A_2...A_n$ را محاسبه کنیم. فرض بر این است که همه ماتریسها $(A_i)_{d_{i-1}d_i}$ است. بنابراین ضرب مذکور خوش تعریف است. می‌خواهیم ببینیم به چه ترتیبی این n ماتریس را پرانتزگذاری نماییم تا تعداد ضرب‌های انجام شده کمترین تعداد ممکن باشد. برای این منظور ماتریسی چون $M = (m_{ij})$ را برای $j \geq i$ به شکلی می‌سازیم که m_{ij} حداقل تعداد ضرب‌های ممکن برای محاسبه $A_iA_{i+1}...A_j$ باشد. بدیهی است که مقادیر این ماتریس باید به شکل بالا مثلثی بدست آید.

یادآوری :

$$C_{pq} = A_{pq}B_{qr}$$

$$C_{ij} = \sum_{k=1}^{r} a_{ik}b_{kj} \qquad \Theta(pqr)$$

$$j = i \Rightarrow A_i \to m_{i,i} = 0$$

$$j = i+1 \Rightarrow A_iA_{i+1} \to m_{i,i+1} = d_{i-1}d_id_{i+1}$$

$$(A_i)d_{i-1}d_i \quad (A_{i+1})d_id_{i+1}$$

$$j > i+1 \Rightarrow A_iA_{i+1}...A_j \Rightarrow m_{ij} = \min(m_{ik} + m_{k+1j} + d_{i-1}d_kd_j) \qquad i \leq k \leq j$$

$$A_iA_{i+1}...A_kA_{k+1}...A_j$$

$$\Rightarrow m_{ij} = \begin{cases} 0 & & i=j \\ d_{i-1}d_id_{i+1} & & j=i+1 \\ \min(m_{ik} + m_{k+1j} + d_{i-1}d_kd_j) & 1 \leq k \leq j & j > i+1 \end{cases}$$

$j-i=0$ $j-i=1$ $j-i=2$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \qquad j-i=s \Rightarrow j=i+s$$

$$m_{i,i+s} = \begin{cases} 0 & s=0 \\ d_{i-1}d_i d_{i+1} & s=1, i=1,2,\dots,n-1 \\ \min\limits_{i \le k \le i+s}\{m_{ik}+m_{k+1,i+s}+d_{i-1}d_k d_{i+s}\} & s>1, i=1,2,\dots,n-s \quad i+s \le n \Rightarrow i \le n-s \end{cases}$$

```
int Minmult(int n, const int dl)
{
index i, j, k, diagonal;
int M[1..n][1..n];
for (i = 1; i <= n; ++i)

      M[i][0] = 0;

for(diagonal = 1; diagonal <= n-1; diagonal++)

      for(i = 1; i <= n-diagonal; i++)

         { j = i + diagonal;
         M[i][j] = min(M[i][k] + M[k+1][j] + d[i-1]×d[k]×d[j])    i ≤ k ≤ j-1
         }

      return M[1][n]          حداقل تعداد ضرب لازم برای ضرب A1...An

}
```

d	d_0	d_1	d_2	…	d_n
	0	1	2		n

ابعاد ماتریس در d ذخیره می‌شود.

$$s=0 \Rightarrow \begin{cases} m_{11} \\ m_{22} \\ m_{33} \\ m_{44} \\ \mathbf{M} \end{cases} \qquad s=1 \begin{cases} m_{12} \\ m_{23} \\ \mathbf{M} \end{cases} \qquad s=n-1 \begin{cases} m_{101} \end{cases}$$

بررسی الگوریتم فوق برروی یک مثال:

A_1	$\times$	A_2	$\times$	A_3	$\times$	A_4	$\times$	A_5	$\times$	A_6
5×2		2×3		3×4		4×6		6×7		7×8
d_0d_1		d_1d_2		d_2d_3		d_3d_4		d_4d_5		d_5d_6

$$S=1\quad\begin{cases} M[1][2]=\min\limits_{1\le k\le i}(m[1][k]+m[k+1][2]+d_0d_kd_2) \\ \qquad =M[1][1]+M[2][2]+d_0d_1d_2 \\ \qquad =0+0+5\times2\times3=30 \\ M[2][3],M[3][4],M[4][5],M[5][6] \end{cases}$$

$$S=2\quad\begin{cases} M[1][3]=\min\limits_{1\le k\le i}(m[1][k]+m[k+1][3]+d_0d_kd_3) \\ \qquad =\min(M[1][1]+M[2][3]+d_0d_1d_3\ ,\ M[1][2]+M[3][3]+d_0d_2d_3) \\ \qquad =\min(0+24+5\times2\times4\ ,\ 30+0+5\times3\times4)=64 \\ M[2][4],M[4][6],M[3][5] \end{cases}$$

$$S=3\quad\begin{cases} M[1][4]=\min\limits_{1\le k\le i}(m[1][k]+m[k+1][4]+d_0d_kd_4) \\ =\min(\quad M[1][1]+M[2][4]+d_0d_1d_4\ , \\ \qquad\qquad M[1][2]+M[3][4]+d_0d_2d_4\ , \\ \qquad\qquad M[1][3]+M[4][4])+d_0d_3d_4)= \\ \quad\min(\quad 0+72+5\times2\times6\ ,\ 30+72+5\times3\times6\ ,\ 64+0+5\times4\times6)=132 \end{cases}$$

	1	2	3	4	5	6	
1	0	30	64	132	226	348	پاسخ نهایی ←
2		0	24	72	156	268	
3			0	75	198	336	
4				0	168	392	
5					0	336	
6						0	

s=5, s=4, s=3, s=2, s=1, s=0

مرتبه‌ی زمانی الگوریتم فوق $\Theta(n^3)$ می‌باشد.

مثال : **الگوریتم Cycle** :

فرض کنید گرامر مستقل از متن $G=(V,T,S,P)$ مفروض باشد که در فرم فرمال چاسکی صدق نماید، یعنی هر قاعده آن به شکل $A \to BC$ یا $A \to a^*$ باشد. می‌خواهیم یک الگوریتم پویا ارائه کنیم که بتواند برای همه $W \in T^*$ تشخیص دهد که $W \in L(G)$ است یا خیر ؟ اگر $W = a_1 a_2 ... a_n$ باشد، W_{ij} را زیر رشته‌ی $a_i a_{i+1} ... a_j$ تعریف می‌کنیم و همچنین تعریف می‌کنیم:

$$T[i,j] = \{A \in V \mid A \xrightarrow{*} W_{ij}\}$$

پرواضح است که $W_{1n} = a_1 ... a_n = W$، بدیهی است که $S \in T[1,n] \Leftrightarrow W \in L(G)$ بنابراین یک ساختار بازگشتی برای $T[i,j]$ ارائه می‌دهیم.

$$W_{ij} = a_i a_{i+1} ... a_k a_{k+1} ... a_j$$

$$T[i,k] = \bigcup_{k=i}^{j-1} \{A \mid A \to BC, B \in T[i,k], C \in T[k+1,j]\}$$

$$T[i,i] = \{A \mid A \xrightarrow{*} a_i\}$$

$$S = 0 \Rightarrow T[1,1], T[2,2], ...$$

$$S = 1 \Rightarrow T[1,2], ..., T[n-1,n]$$

$$\vdots$$

$$S = n-1 \Rightarrow T[1,n]$$

زمان اجرا $\qquad \sum_{s=0}^{n-1} S(n,s) \in \Theta(n^3)$

مسئله کوله پشتی صفر و یک :

$\sum_{i=1}^{n} p_i x_i$ ماکزیمم شود $x_i \in \{0,1\}$

$\sum_{i=1}^{n} w_i x_i \le M$ $1 \le i \le n$

روش دکتر نقیب‌زاده :

تعریف : S^0 وضعیت اولیه مسئله $S^0 = \{(p,w)\} = \{(0,0)\}$

S^i: مجموعه کلیه حالات مسئله تا مرحله‌ی iام و شامل مرحله‌ی iام، اعم از اینکه شی iام انتخاب بشود یا نشود.

S_1^i: مجموعه کلیه حالات مسئله تا مرحله‌ی iام و شامل مرحله‌ی iام، مشروط به اینکه شی iام انتخاب بشود.

مثال : $(p_1 p_2 p_3) = (1,2,5)$ $(w_1, w_2, w_3) = (2,3,4)$ و $n = 3$ تعداد اشیاءو $M = 6$ حداکثر وزن ممکن

$S^0 = \{(0,0)\}$
$S_1^1 = \{(1,2)\}$
$S^1 = \{(0,0),(1,2)\}$
$S_1^2 = \{(2,3),(3,5)\}$
$S^2 = \{(0,0),(1,2),(2,3),(3,5)\}$
$S_1^3 = \{(5,4),(6,6)\}$
$S^3 = \{(0,0),(1,2),(2,3),(5,4),(6,6)\}$

زوج مورد نظر $(6,6)$.
زوج $(6,6)$ در S^2 نیست پس در اثر در نظر گرفته شدن جسم 3 حاصل شده است، پس $k_3 = 1$، این زوج از $(6-p_3, 6-w_3) = (1,2)$ بدست می‌آید که در S^1 وجود دارد پس $w_2 = 0$ و در نهایت $w_1 = 1$ خواهد بود.

$void\ dpknapsack(float\ p[], float\ w[], \text{int}\ n, float\ M, \text{int}\ x[])$
$\{float\ p_1, w_1;$
$set\ S[n+1], S_1[n+1];$
$S[0] = \{(0,0)\};$
$for\ (i=1; i <= n; ++i)$

$\{S_1[i] = \{(p_1, w_1) \mid (p_1 - p[i]\ ,\ w_1 - w[i]) \in S[i-1]\ \&\&\ w_1 \le M\};$
$S[i] = merge(S[i-1], S_1[i]);$
$\}$

$(p_1, w_1) = last\ tuple\ in\ S[n];$

$for\ (i = n; i > 0; i--)$

$if\ ((p_1, w_1) \in S[i-1])$

$x[i] = 0;$

$else$

$\{x[i] = 1;$

$(p_i, w_1) = (p_1\ ,\ p[i]\ ,\ w_1 - w[i]);$

$\}$

پیچیدگی زمانی:

اگر زمان لازم برای اجرای الگوریتم را تنها به زمان محاسبه S^iها محدود کنیم و تعداد مولفه‌های S^iرا با $|S^i|$ نشان دهیم، داریم:

$$\sum_{0 \le i \le n} |S_0^i| = \sum_{0 \le i \le n} 2^i = 2^{n+1} - 1 \Rightarrow T(n) = \Theta(2^n)$$

روش دوم : $V[i, j]$ ماکزیمم ارزش کوله پشتی با تحمل j می‌تواند اشیاء $1,2,...,i$ را حمل نماید.

$$V[i,j] = \begin{cases} 0 & j = 0\ ,\ i \ge 0 \\ V_1 & i = 1\ ,\ j \ge w_1 \\ V[i-1, j] & i > 1\ ,\ j < w_i \\ \max\{V_i + V[i-1\ ,\ j - w_i]\ ,\ V[i-1\ ,\ j]\} & i > 1, j \ge w_i \end{cases}$$

زمان اجرای الگوریتم $\Theta(n(w+1))$

Weight	**Value**	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$	$v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$	$v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5$	$v_3 = 18$	0	1	6	7	7	18	187	24	25	25	25	25
$w_4 = 6$	$v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7$	$v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

مثال : **درخت جستجوی دودویی بهینه (Optimal Binary Search Tree)**:

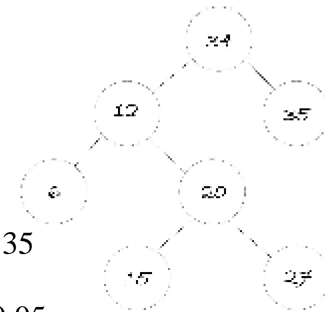
منظور از درخت جستجوی دودویی، درخت دودویی است که هر رأس آن با عددی برچسب خورده است که برچسب زیر درختان چپ یک گره، باید کوچک‌تر یا مساوی گره پدر باشد و برچسب زیر درختان سمت راست یک گره باید، بزرگتر از گره پدر باشد.

فرض کنید $c_1, c_2, \ldots, c_n$ ، n کلید دوبه‌دو متمایز باشند که بصورت زیر مرتب شده‌اند.

$$c_1 < c_2 < c_3 < \ldots < c_n$$

فرض بر این‌است که احتمال دسترسی به کلید c_i برابر p_i $(1 \le i \le n)$ باشد و همچنین جمع احتمالات ممکن برابر 1 باشد، یعنی $\sum_{i=1}^{n} p_i = 1$. بافرض اینکه ریشه در سطح صفرم می‌باشد، اگر یک کلید مانند c_i در عمق d_i باشد حداکثر $d_i + 1$ مقایسه برای یافتن آن نیاز است. بنابراین متوسط تعداد مقایسه‌های مورد نیاز برای این درخت از رابطه‌ی زیر بدست می‌آید.

$$C = \sum_{i=1}^{n} p_i (d_i + 1)$$



Node	6	12	15	20	27	34	35
Probability	0.2	0.25	0.05	0.1	0.05	0.3	0.05

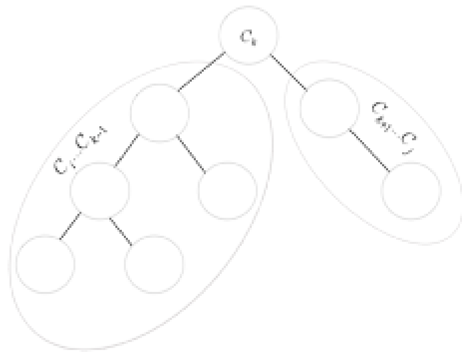
$$C = 1 \times 0.3 + 2 \times 0.25 + 2 \times 0.05 + 3 \times 0.2 + 3 \times 0.1 + 4 \times 0.05 + 4 \times 0.05 = 2.2$$

هدف، ایجاد یک درخت جستجوی دودویی بهینه است که متوسط تعداد مقایسه‌های مورد نیاز برای یافتن گره‌های آن کمترین تعداد ممکن باشد. بنابراین می‌خواهیم یک الگوریتم پویا ارائه دهیم که کمترین تعداد مقایسه‌های لازم را برای قراردادن عناصر در یک درخت دودویی بهینه مشخص نماید.

$c_{ij} =$ حداقل متوسط تعداد مقایسه‌های لازم برای قرارگیری عناصر $c_i < c_{i+1} < \ldots < c_j$ در یک درخت جستجوی دودویی می‌باشد.

$$c_{ij} = \begin{cases} 1 * P_i = P_i & i = j \\ \min\limits_{i \le k \le j} \left(c_{i,k-1} + c_{k+1,j} + \sum_{t=i}^{j} p_t\right) & j \ge i+1 \end{cases}$$

یکی در درخت پایین آمده



$$c_{i,k+1} + c_{k+1,j} + p_k + p_{k+1} ... + p_j + p_i + p_{i+1} + ... + p_{k-1} =$$
$$c_{i,k-1} + c_{k+1,j} + \sum_{t=i}^{j} p_t$$

پیاده سازی این الگوریتم شبیه به پیاده سازی الگوریتم ضرب زنجیرهای ماتریس‌ها می‌باشد.

```
int OBCT(int n, Const int P[], index Q[][])
{
index i, j, k, d;
int C[1..n][1..n];
for (i = 1; i <= n; ++i)

      C[i][i] = P[i];

for (d = 1; d <= n-1; d++)

      for(i = 1; i <= n-d; ++i)

            { j = i + d;
            C[i][j] = mnimum_{i<=k<=j} (C[i][k-1] + C[k+1][j] + sum_{t=i}^{j} P[t])
            Q[i][j] = a value of k that gave the mnimum;
            }

return C[i][n]
}
```

مثال : **فروشنده دوره گرد**:

در محدوده جغرافیایی فروشنده دوره‌گردتعدادی شهر وجود دارد که فاصله بین هر زوج از شهرها مشخص و عددی مثبت است. قرار است فروشنده دوره‌گرد از یکی از شهرها شروع کند و از کلیه شهرها هریک فقط یکبار گذر کند و در نهایت به نقطه شروع برگردد. هدف از این مسئله یافتن مسیری است با کوتاه‌ترین طول ممکن، یعنی از بین $(n-1)! = (n-1)(n-2)...1$ تور ممکن کوتاهترین آنرا انتخاب نماییم.

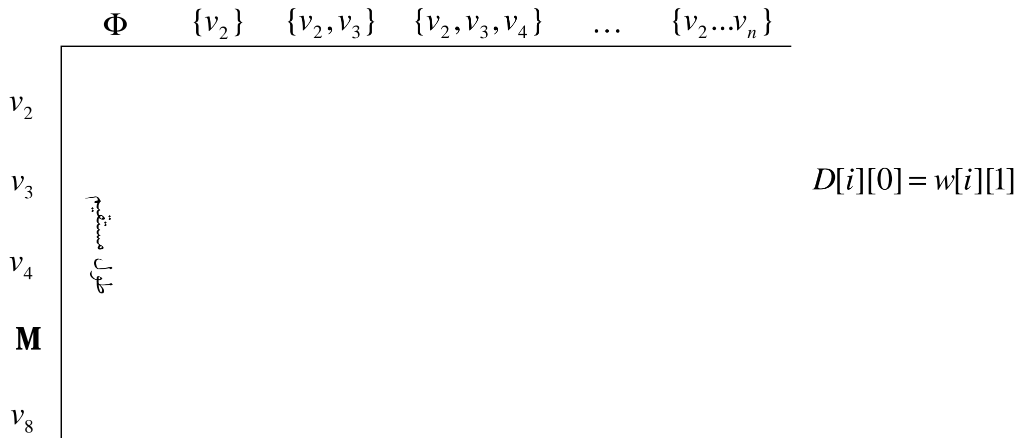
اگر v_k به v_l روی یک تور بهینه باشد، زیر مسیر آن تور از v_k به v_l باید کوتاه‌ترین مسیر از v_k به v_l باشد که از هر یک از گره‌های دیگر دقیقاً یکبار می‌گذرد.

V: مجموعه تمام گره‌ها (شهرها)

A: یک زیر مجموعه از V

$D[V_i][A]$: طول کوتاه‌ترین مسیر از v_k به v_1 با تنها یکبار عبور از گره‌های مجموعه

$D[i][v-\{v_1\}] \leftarrow$ خروجی برنامه



$D[i][0] = w[i][1]$

$void\ travel(\text{int}\ \ n, \text{int}\ \ w[][], index\ \ p[][], number\ Minlength)$
$\{index\ i, j, k;$
$number\ D[1..n][subset\ of\ v = \{v_1\}]$
$for(i = 1; k <= n; ++i)$

$D[i][\Phi] = W[i][1];$

$for(k = 1; k <= n-2; ++k)$

$for(all\ subsets\ \ A\ V - \{v_1\}\ \ containing\ \ k\ vertices)$

$for(i\ such\ that\ i \neq 1\ \ and\ \ v_i\ \ is\ not\ in\ \ A_1)$

$\{D[i][A] = mnimum(w[i][j] + D[v_j][A - \{v_j\}]);$
$p[i][A] = value\ of\ \ j\ that\ \ gave\ the\ mnimum;$
$\}$

$D[1][v - \{v_i\}] = mnimum(w[1][j] + D[v_j][v - \{v_1\}]);$
$P[i][v - \{v_i\}] = value\ of\ \ j\ that\ \ gave\ the\ mnimum;$
$mnimum = D[i][v - \{v_1\}]$
$\}$

$$T(n)=\sum_{k=1}^{n-2}\binom{n-1}{k}(n-1-k)k\ ,\ (n-1-k)\binom{n-1}{k}=(n-1)\binom{n-2}{k}$$

$$T(n)=(n-1)\sum_{k=1}^{n-2}\underbrace{k\binom{n-1}{k}}_{(n-2)\binom{n-3}{k-1}}=(n-1)(n-2)\sum_{k=1}^{n-2}\binom{n-3}{k-1}$$

$$T(n)=(n-1)(n-2)2^{n-3}\Rightarrow T(n)=\Theta(n^2 2^n)$$

مثال: **بیشترین مجموع اعداد در مسیر**

مثلثی از اعداد داریم. می‌خواهیم مسیری از راس مثلث به ضلع پایین پیدا کنیم که مجموع اعداد واقع در مسیر بین کلیه مسیرهای ممکن بیشترین باشد. در هر حرکت از بالا به پایین می‌توان به سمت چپ یا راست قدم برداشت.

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

راه حل: از پایین ترین سطح شروع کرده و در هر سطح عدد هر گره را با بیشترین عدد گره‌هایی که در سطح بالا با آن همسایه‌اند جمع می‌کنیم.

$A[i][j][0]\leftarrow$ عدد موقعیت ردیف i و ستون j

$A[i][j][1]\leftarrow$
1 : جمع با عدد سمت راست در سطح بالا
2 : جمع با عدد سمت چپ در سطح بالا

```
void diptiangle(int A[][][],int n)
{int i, j;
for(i = n-2; i >= 0; i--)
    for( j = 0; j <= i)
        if (A[i+1][j][0] < A[i+1][i+1][0])
            {A[i][j][0] = A[i][j][0] + [i+1][j+1][0];
             A[i][j][0] = 1; }
        else
            {A[i][j][0] = A[i][j][0] + A[i+1][j][0];
             A[i][j][0] = 0; }
```

// $A[0][0][0]$ مجموع اعداد واقع در مسیر را دارد//

مثال : **بزرگ‌ترین زیر رشته مشترک بین دو رشته (LCS)**:

Longest Common Subsequence

$x = (x_1, x_2, ..., x_n)$

$y = (y_1, y_2, ..., y_n)$

$LCS(x, y) : z = (z_1, z_2, ..., z_k)$

1. $if\ x_m = y_n\ then\ z_k = x_m = y_n\ and\ z_{k-1}\ is\ an\ LCS\ of\ x_{m-1}\ and\ y_{n-1}$
2. $if\ x_m \neq y_n\ then\ z_k \neq k_m\ implies\ that\ z\ is\ an\ LCS\ of\ x_{m-1} and\ y$
3. $if\ x_m \neq y_n\ then\ z_k \neq y_n\ implies\ that\ z\ is\ an\ LCS\ of\ x\ and\ y_{n-1}$

$$c[i,j] = \begin{cases} 0 & if\ i = 0\ OR\ j = 0 \\ c[i-1, j-1] + 1 & else\ if\ x_i = y_j \qquad i, j > 0 \\ \max(c[i, j-1], c[i-1, j]) & else\ x_i \neq y_j \qquad i, j > 0 \end{cases}$$

تعریف:

$c[i, j] = LCS\{(x_1, x_2, ..., x_i), (y_1, y_2, ..., y_j)\}$

$CS = length(k, j)$

$m \leftarrow length[x]$

$n \leftarrow length[y]$

$for\, i \leftarrow 1\ to\ m\ do$

$\qquad c[i, 0] \leftarrow 0$

$for\ j \leftarrow 0\ to\ n\ do$

$\qquad c[0, j] \leftarrow 0$

$for\, i \leftarrow 1\ to\ m\ do$

$\qquad for\ j \leftarrow 1\ to\ n\ do$

$\qquad\qquad if\ x_i = y_j\ then$

$\qquad\qquad\qquad c[i, j] \leftarrow c[i-1, j-1] + 1$

$\qquad\qquad\qquad b[i, j] \leftarrow$ "ã"

$\qquad\qquad else\ if\ c[i-1, j] \geq c[i, j-1]\ then$

$\qquad\qquad\qquad b[i, j] =$ "á"

$c[i, j] \leftarrow c[i-1, j]$

else

$c[i, j] \leftarrow c[i, j-1]$

$b[i, j] \leftarrow$ "ß"

return c, b

$x = (A, B, C, B, D, A, B)$

$y = (B, D, C, A, B, A)$

		0	1	2	3	4	5	6
		y_j	B	C	D	A	B	A
0	x_I	0	0	0	0	0	0	0
1	A	0	á0	á0	á0	ã1	ß1	ã1
2	B	0	ã1	ß1	ß1	á1	ã2	ß2
3	C	0	á1	á1	ã2	ß2	á2	á2
4	B	0	ã1	á1	á2	á2	ã3	ß3
5	D	0	á1	ã2	á2	á2	á3	á3
6	A	0	á1	á2	á2	ã3	á3	ã4
7	B	0	ã1	á2	á2	á3	ã4	á4

Print_LCS(b,x,I,j)

if $i = 0$ *or* $j = 0$ *then*

return 0

if $b[i, j] =$ "ã" *then*

$print_LCS(b, x, i-1, j-1)$

$print\ x_i$

else if $b[i, j] =$ "á" *then*

$print_LCS(b, x, i-1, j)$

else

$print_LCS(b, x, i, j-1)$

زمان اجرا : $O(m+n)$

فصل ششم : **تکنیک بازگشت به عقب (Back Tracking)** :

این روش یک تکنیک برنامه نویسی است، که در آن پویش فضای مسئله به شکل اول عمق صورت می‌پذیرد. یعنی با توجه به وضعیت موجود و فاکتور شاخه شاخه شدن مسئله در جهت یک شاخه‌ی خاص حرکت می‌کند تا یکی از دو حالت زیر اتفاق بیافتد:

یا اینکه به یک جواب از مسئله منجر شود یا به بن‌بست(Deadlock). در صورتی که مسئله به بن‌بست منجر شود، در این حالت، در فضای مسئله، حرکت به یک گره قبلی از گراف صورت گرفته و در جهت دیگر ادامه می-یابد. در مسائل back Tracking، تابعی به نام promissing وجود دارد، این تابع بررسی می‌کند که آیا انتخاب گره اخیر feasible است یا خیر. این تابع در صورتی که امکان انتخاب این گره وجود داشته باشد، مقدار True را بر می‌گرداند و در غیر این صورت false بر می‌گرداند. نکته حائز اهمیت این است که تابع promissing تضمین نمی‌کند که انتخاب این گره قطعاً به یک جواب از مسئله منجر شود و فقط صرفاً بررسی می‌کند که انتخاب این گره با گره‌های قبلی خودش تداخلی ندارد.

توجه: ساختار تمام الگوریتم‌های Back Tracking دقیقاً مشابه است و فقط تفاوت در تابع Promissing و فاکتور شاخه شاخه شدن است.

مثال: **مسئله n وزیر** :

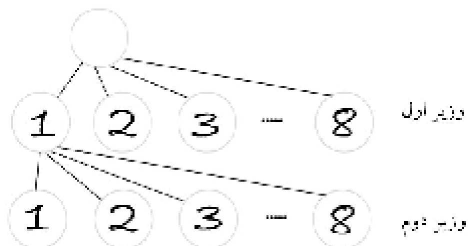
در یک صفحه شطرنج $n \times n$ می‌خواهیم n وزیررا بگونه ای قرار دهیم که هیچ دو وزیری یکدیگررا تهدید نکنند.

```
void queens(index i)
{index j
if (promissing(i))
      if( i == n)
            cout << col[1] through col[n]
      else
            for(j = 1;j <= n;j ++ )
                {
                col[i + 1] = j;
                queens(i + 1 );
                }
void promissing(index i)
{index k;
bool switch;
k = 1;
switch = true;
```

```
while (k < i && switch)
   {
    if(col[i] == col[k] || abs(col[i]-col[k]) == i-k)
          switch = false;
   k++;
   }
return switch;
}
```

در یک صفحه 2×2 نمی‌توان 2 وزیر را چنان قرار داد که همدیگر را تهدید نکنند و همچنین در صفحه 3×3 نمی توان 3 وزیر را به گونه چید که همدیگر را نزنند.

برای حل مسئله می‌توان چنان فرض کرد که وزیر iام در سطر iام واقع شده است. بنابراین مهم یافتن ستونی است که وزیر iام در آن قرار گرفته است. آنرا با $col[i]$ مشخص می‌کنیم. بنابراین وزیر iام در مختصات $(col[i],i)$ قرار می‌گیرد. فاکتور شاخه شاخه شدن در صفحه $n\times n$، n است.



مسئله Promissing در اینجا آنست که وزیر نباید با قبلی‌ها در یک سطر یا ستون باشد. اگر $m=\pm1$ باشد، promissing مقدار false را برمی‌گرداند.

$$i = m = \frac{i-k}{col[i]-col[k]} \Rightarrow i-k = |col[i]-col[k]|$$

نکته: زمان الگوریتم‌های Back Tracking را نمی‌توان محاسبه نمود، ولی با تکنیک‌های خاصی می‌توان آنرا بصورت احتمالی بدست آورد.

مثال : **حاصل جمع زیر مجموعه‌ها**:

هدف تعیین همه ترکیبات اعداد صحیح موجود در یک مجموعه n عدد صحیح، بطوریکه حاصل جمع آن‌ها مساوی مقدار معین w شود.

```
void sum-of-subsets(index i, int weigth, int total)
{if (promissing(i))
   if(weight == m)
```

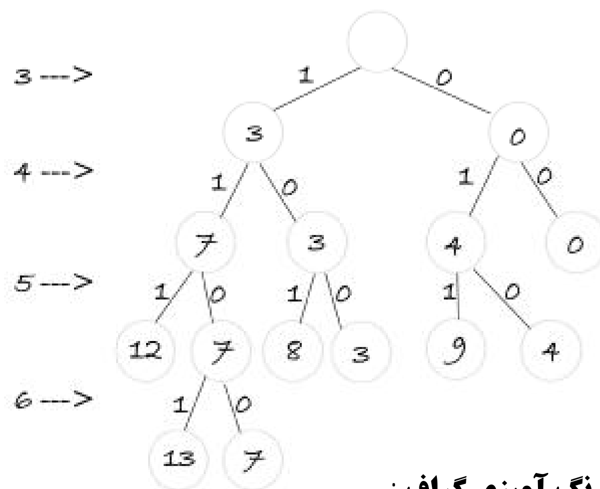
```
            cout << x[1] trough x[i]
      else
            {x[i+1] = 1;
            sum-of-sumsets(i+1,weight + w[i+1],total = w(i+1));
            x[i+1] = 0;
            sum-of-subsets(i+1,weight,total-w[i+1]);
            }
}
Bool promissing(index i)
{
return(weight + total >= m)&&(weight == m || weight + w[i+1] <= m);
}
```

$m = 13$

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

$weight_k = \sum_{i=1}^{k} k_i w_i$

$total_k = \sum_{i=k+1}^{n} w_i$



مسئله رنگ آمیزی گراف :

می‌خواهیم یک گراف غیر جهت‌دار را رنگ آمیزی کنیم. منظور از رنگ‌آمیزی گراف آن است که رئـوس گـراف را بگونه‌ای رنگ آمیزی نماییم که هیچ دو رأس مجاوری هم‌رنگ نباشند. می‌خواهیم ببینیم که آیا می‌توان یک گـراف را با mرنگ، رنگ آمیزی نمود.در اینجا چون می‌خواهیم با mرنگ گراف را رنگ آمیزی نمـاییم، فـاکتور شـاخه شاخه شدن mاست.

```
void m-coloring(index m)
{ int Color;
if (promissing(i))
   if (i == n)
      cout << VColor[i] through VColor[n]
   else
      for(color = 1;color <= m;color++)
         {VColor[i+1] = color;
         m-Coloring(i+1);
         }
}
```

```
Bool Promissing(inedex i)
{
index j;
bool switch;
switch = True;
j =1;
while(j < i && switch)
   {if (w[i][j] && VColor[i] == VColor[j])
       switch = false;
   ++j;
   }
return switch;
}
```

$w[i][j]$ ماتریس وزن گراف

مسئله یافتن دورهای همیلتون :

این چرخه برای گراف‌های مرتبط تعریف می‌گردد و گردشی کامل است که از یکی از گره‌های گراف شروع مـی-شود و کلیه‌ی گره‌ها را، هرکدام یک بار ملاقات می‌کند و به گره اولبه بازمی‌گردد.

ماتریس وزن‌های گراف 1 تا n است. و خروجی هر مسیر آرایه‌ای از شاخص‌ها بنام *vindex* است کـه از 0 الـی $n-1$ شاخص دهی شده‌است.

```
Void Hamiltonain(index i)
{index j;
if (promissing(i))
   if (i == n-1)
      cout << Vindex[0] through Vindex[n-1]
   else
      for (j = 2;j <= n;j++)
         {Vindex[i+1] = j;
          hamiltonain(i+1);
         }
}
Bool Promissing(index i)
{index j;
bool switch;
if (i == n-1 && !w[Vindex[n-1]][vindex[0]])
   switch = false;
```

```
else if (i > 0 && ! w[Vindex[i-1]][Vindex[i]])
    switch = false;
else
    {Switch = True;
    j = 1;
    while(j < i&& Switch)
        {if(Vindex[i] == Vindex[j])
            Switch = false;
        ++j;
        }
return Switch;
}
```

مثال : **مسئله کوله پشتی** 1/0:

آرایه‌های p, w به ترتیب غیر نزولی مرتب شده‌اند.

```
Void Knapsack(index i, int Profit, int Weight)
{if (weight <= w && profit > maxProfit)
    { maxProfit = profit;
    numbest = i;
    bestset = include;
    }
if ( Promissing(i))
    {include[i+1] = "Yes";
     knapsack(i+12,profit + p[i+1],weight + w[i+1]);
     include[i+1] = "No";
     Lnapsack(i+1,profit,weight);
    }
}
Bool Promissing(inedex i)
{index j,k;
int totalweight;
float bound;
if (weight >= w)
    return false;
else
    {j = i+1;
    bound = profit;
```

```
    totalweight = weight;
    while (j <= n && totalweight + w[j] <= w)
        totalweight = totalweight + w[j];
        bound = bound + p[i];
        ++j;
      }
    k = j;
    if (k = n)
        bound+ = (w-totalweight) × p[k]/w[k];
     return (bound > max profit);
    }
}
```

در برنامه نویسی پویا مرتبه‌ی زمانی $\min(2^n, w_n)$ است. الگوریتم backtracking در بدترین حالت $\Theta(2^n)$ می‌باشد. گفتن اینکه کدام‌یک بهتر است کار سختی است، زیرا همه درخت همواره در backtracking بررسی نمی‌شود.

مثال : **پویش گراف‌ها**:

بطور کلی دو روش مختلف برای پویش یک گراف به ترتیب عبارتند از جستجوی اول عمق (DFS) و جستجوی اول پهنا (BFS).

```
Procedure fd_search(G)
    for each v ∈ N do
        mark[v] ← not visited
    for each v ∈ N do
        if mark[v] ≠ visited then
            dfs(v)

procedure dfs(v)
    {Node v has not previously been visited}
    mark[v] ← visited
    for each node w adjacent to v do
        if mark[w] ≠ visited then
            dfs(w)
```

```
Procedure dfs(v)
    Q ← empty-queue
    mark[v] ← visited
    enqueue v into Q
    while Q is not empty do
        u ← first(Q)
        dequeue u from Q
        for each node w adjacent to you do
            if mark[w] ≠ visited Then
                mark[w] ← visited
                enqueue w into Q

Procedure bf_Search(G)
    for each v ∈ N do
        mark[v] ← not-visited
    for each v ∈ N do
        if mark[v] ≠ visited then
            bfs(v)
```

فصل هفتم : انشعاب و تحدید (Branch and Bound):

تکنیک B&B یک تکنیک برنامه نویسی است که در آن عملکرد تکنیک Backtracking برای برخی از مسائل می‌تواند بهبود یابد. با این مفهوم که با یک تخمین اولیه از هزینه‌ی یک راه حل، به دنبال یک راه حل بهتر هستیم. برای این منظور در هر کجای مسیر از راه حل بعدی، اگر میزان هزینه، بیشتر از هزینه تخمین اولیه باشد، آن مسیر حذف می‌شود. (می‌توان جهت تعیین یک تخمین اولیه از روش حریصانه استفاده نمود).

روش شاخه و حد، اولاً ما را به روش خاصی از پیمایش درخت محدود نمی‌کند و ثانیاً، تنها برای مسائل بهینه سازی بکار می‌رود.

جستجو بر اساس اول، بهترین انجام می‌پذیرد.

توجه : در روش شاخه و حد مرتبه‌ی زمانی در بدترین حالت تغییر نمی‌کند اما در عمل تعداد عملیات مورد نیاز کمتر خواهد بود.

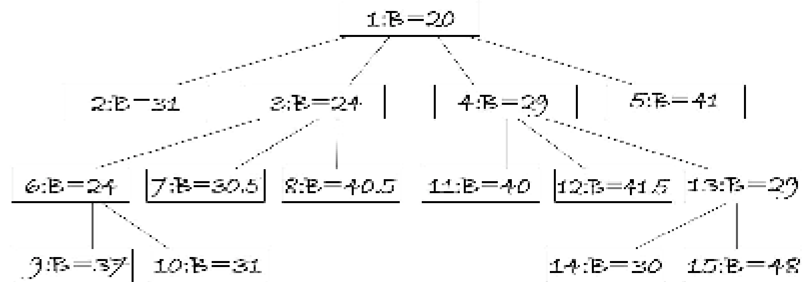
مثال: **فروشنده دوره گرد**:

تابع ارزیابی: کمترین هزینه ممکن برای یک گردش برابر می‌باشد با مجموع کمترین هزینه‌ی ورود و خروج هر گره در آن تور و تقسیم حاصل جواب بر 2.

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$

$$B = \sum_{i=1}^{n} (\min(\text{سطر } i \text{ ام}) + \min(\text{ستون } i \text{ ام}))/2$$

$$B = (4+4+7+5+4+4+4+2+2+4)/2 = 20$$



برای محاسبه‌ی هزینه‌ی تور 2 :

1- لبه‌ی (1,2) بصورت قطعی انتخاب شده است. ارزش ورودی به گره 3 و خروجی از گره 1 برابر 14 می‌باشد.
2- خروجی از گره 2 نمی‌تواند ورودی به گره 1 باشد.
3- ورودی به هیچ گره‌ای به (جز2) نمی‌تواند از گره 1 باشد.
4- خروجی از هیچ گره‌ی دیگری نباید به گره 2 باشد.

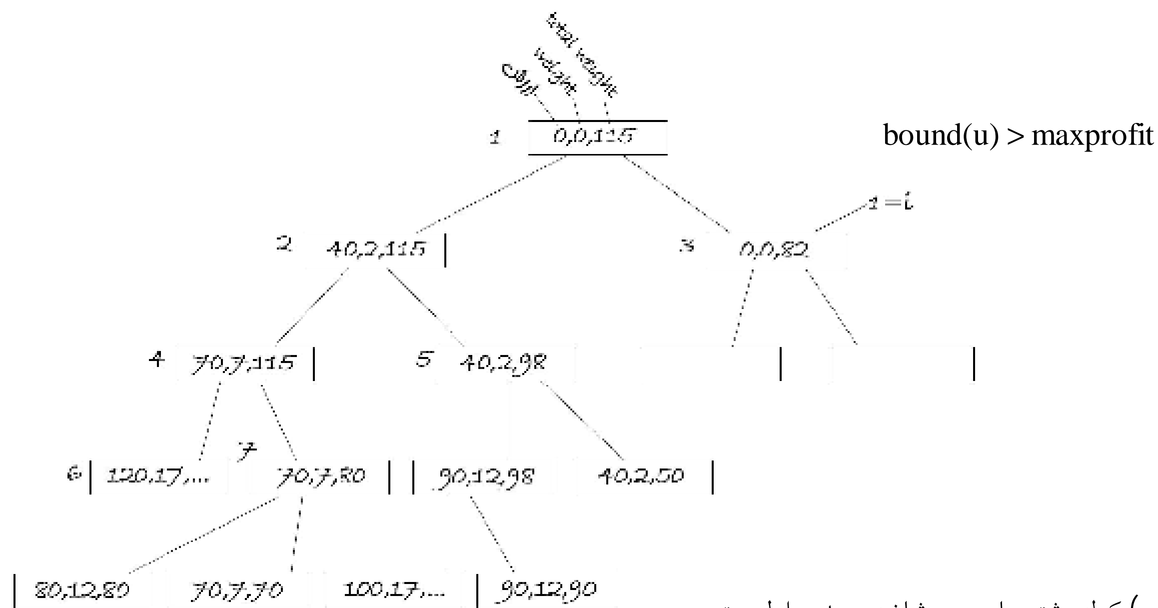
مثال: **کوله پشتی** 0/1:

i	p_i	w_i	p_i/w_i
1	20	2	20
2	30	5	6
3	50	10	5
4	10	5	2

$$Total\ weight\ =\ weight\ +\ \sum_{j=i+1}^{k-1} w_j$$

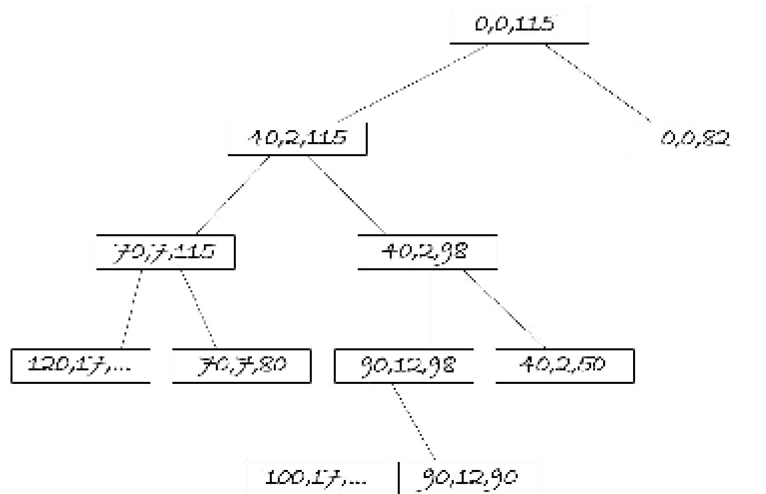
$$bound = (profit + \sum_{j=i+1}^{k-1} p_j) + (w\text{-}totalweight)\frac{p_k}{w_k}$$

الف) کوله پشتی صفرویک با هرس شاخه و حد و بصورت اول سطح



ب) کوله پشتی با هرس شاخه وحد و اول بهترین

در این روش در هر لحظه از میان نودهای باز فقط آنرا که بهترین است بسط می‌دهیم، ابتدا بررسی می‌کنیم کدام‌یک از نودها، حد را رعایت کرده(در این مثال مقدار *bound* آن‌ها از *maxprofit* بیشتر است.) و سپس از بـین آن‌هـا *max* را انتخاب می‌کنیم.



پایان