طراحی الگوریتم

فصول ۳ و ۴

مؤلف و پشتیبان : مهندس مهدی دادبخش

فصل سوم

روش برنامه ریزی پویا (Dynamic Programming)

درروش تقسیم و حل به این ترتیب عمل می کردیم که ابتدا مسئله اصلی رابه مسائل کوچکتر تقسیم کرده سپس مسائل کوچکتر را حل می کردیم و درنهایت حل مسئله اصلی را با ترکیب حل مسائل کوچکتر بدست می آوردیم.دراین روش که اغلب به صورت بازگشتی ممکن است این تقسیمات منجر به تکرار انجام حل مسائل کوچکتر شده،که باعث افزایش مرتبه زمانی الگوریتم می شود . تقسیم و حل برای مسائلی که در آن مسائل کوچکتر تقسیم شده مستقل از هم باشند مناسب است نظیر مسئله مرتب سازی ادغامی .امادر مسائلی که در آن مسائل کوچکتر تقسیم شده باهم در ارتباط هستند نظیر مسئله فیبوناچی ،این روش مناسب نمی باشدو معمولاً باباز دهی کم روبرو می شود.برای حل اینگونه مسائل باید الگوریتم دیگری به کار ببریم.

روش برنامه نویسی پویا تکنیکی است که به آن می پردازیم .این روش از این لحاظ که نمونه رابه نمونه های کوچکتر تقسیم می کندمشابه روش تقسیم و حل است،ولی در این روش،ابتدا نمونه های کوچکتر راحل می کنیم،نتایج را ذخیره می کنیم و سپس هرگاه‌به هریک از آنهانیاز داشته باشیم،بجای محاسبه دوباره،کافی است آن رابازیابی کنیم.این روش برخلاف روش تقسیم وحل ،که روشی بالا به پایین (top -down) بود،یک روش پایین به بالا(Bottom-up) است.مراحل بسط یک الگوریتم برنامه نویسی پویا به شرح زیر است:

۱-بنا نهادن یک ویژگی بازگشتی که حل نمونه‌ای از مسئله را ارائه می دهد.

۲-حل نمونه ای از مسئله به شیوه جزء به کل ، باحل نمونه های کوچکتر.

برای نشان دادن این مراحل به ذکر مثالهایی می پردازیم :

۳-۱)ضریب دوجمله ای :

ضریب دوجمله ای به صورت زیر است:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad 0 \le k \le n \quad \text{به ازای}$$

برای مقادیری از K,n که کوچک نیستند ، نمی توانیم ضریب دوجمله ای را مستقیماً از این تعریف محاسبه کنیم زیرا n! حتی برای مقادیر نه چندان بزرگ n نیز بسیار بزرگ است.پس بهتر است از رابطه زیر استفاده کنیم:

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

الگوریتم ۳-۱) ضریب دوجمله ای بااستفاده از تقسیم وحل :

مسئله :محاسبه ضریب دوجمله ای:

ورودی :اعداد صحیح و مثبت k,n که در آن $k \le n$ است.

خروجی :bin، ضریب دوجمله ای $\binom{n}{k}$.

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

باز الگوریتم بازدهی بسیارکمی دارد.تعداد جملاتی که الگوریتم برای تعیین $\binom{n}{k}$ محاسبه می کند،برابر است با $2\binom{n}{k}-1$ مشکل اینجاست که در هر فراخوانی بازگشتی، نمونه ها چندین بار حل می شوند.برای مثال bin(n-1,k-1) و bin(n-1,k) هر دو به نتیجه (bin (n-2 ,k-1 نیاز دارند.همانطور که گفتیم روش تقسیم وحل مادامی که نمونه ای به دو نمونه کوچکتر تقسیم شود که تقریباً به بزرگی نمونه اولیه هستند ،بازدهی ندارد.

حال با استفاده از برنامه ریزی پویا الگوریتمی بابازدهی بیشتر طراحی می کنیم.مراحل بنا کردن الگوریتم برنامه نویسی پویا برای این مسئله به شرح زیر است:

۱- یک ویژگی بازگشتی بنا می کنیم(B[i][j] آرایه ای است که حاوی $\binom{i}{j}$ می باشد.)

B[i-1][j]

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

۲- نمونه ای ازمسئله رابه شیوه جزء به کل بامحاسبه سطر های B به ترتیب و با شروع از سطر اول ،حل می کنیم این مرحله درشکل زیر نشان داده شده است:

	0	1	2	3	4	J	K
0	1						
1	1	1					
2	1	2	1				
3	1	3	2	1			
4	1	4	3	2	1		
.							
.							
.							
.i						B[i-1][j-1]+B[i-1][j]=B[i][j]	
n							$B[n][k]=\binom{n}{k}$

ثال ۳-۱) $B[4][2]=\binom{4}{2}$ رامحاسبه کنید:

حاسبه سطر ۰ {این مرحله فقط برای آنکه الگوریتم بدقت دنبال شود ،انجام می شود.}

$B[0][0]=1$

	0	1	2	3	4	J	K
0	1						
1	1	1					
2	1	2	1				
3	1	3	2	1			
4	1	4	3	2	1		
.							
.							
.							
.i						B[i-1][j-1]+B[i-1][j]=B[i][j]	
n							$B[n][k]=\binom{n}{k}$

مثال ۳-۱) $B[4][2]=\binom{4}{2}$ رامحاسبه کنید:

محاسبه سطر ۰ {این مرحله فقط برای آنکه الگوریتم بدقت دنبال شود ،انجام می شود.}

$B[0][0]=1$

محاسبه سطر ۱:

$$\begin{cases} B[1][0]=1 \\ B[1][1]=1 \end{cases}$$

محاسبه سطر ۲:

$$\begin{cases} B[2][0]=1 \\ B[2][1]= B[1][0]+B[1][1]=1+1=2 \\ B[2][2]=1 \end{cases}$$

محاسبه سطر ۳:

$$\begin{cases} B[3][0]=1 \\ B[3][1]=B[2][0]+B[2][1]=1+2=3 \\ B[3][2]=B[2][1]+B[2][2]=2+1=3 \end{cases}$$

محاسبه سطر ۴:

$$\begin{cases} B[4][0]=1 \\ B[4][1]=B[3][0]+B[3][1]=1+3=4 \\ B[4][2]=B[3][1]+B[3][2]=3+3=6 \end{cases}$$

الگوریتم(۳-۲) ضریب دوجمله ای بااستفاده از برنامه نویسی پویا:

مسئله :محاسبه ضریب دو جمله ای

ورودی : اعداد صحیح و مثبت k،n که درآن $k \le n$ است.

خروجی :bin2، ضریب دوجمله ای $\binom{n}{k}$.

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

تعداد گذرها ازحلقه j به ازای مقادیر مختلف i را در جدول زیرنشان می‌دهیم:

i	0	1	2	...	K	K+1	...	n
متغیرها	1	2	3	...	K+1	K+1	...	K+1

پس تعداد کل گذرها عبارتست از:

$$1+2+3+4+\ldots\ldots+\underbrace{(K+1)+(K+1)+\ldots\ldots\ldots(K+1)}_{n-k+1 \text{ بار}}$$

و سپس داریم:

$$\frac{k(k+1)}{2}+(n-k+1)(k+1)=\frac{(2n-k+2)(k+1)}{2}\in\theta(n/k)$$

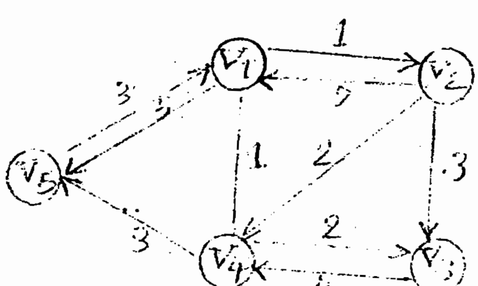
این الگوریتم بازدهی بیشتری نسبت به روش تقسیم و حل دارد.

راه دیگر برای بهتر کردن بازدهی این است که ا زاین واقعیت بهره ببریم که $\binom{n}{k}=\binom{n}{n-k}$

۳-۲)الگوریتم فلوید برای کوتاهترین مسیر:

یک مشکل متداول درسفر ،تعیین کوتاهترین مسیر ازشهری به شهردیگر است.

حالا الگوریتمی طراحی می کنیم که این مسئله ومسائل مشابه راحل کند.ابتدا لازم است نظریه گرافها رامرور کنیم.شکل زیر یک گراف جهت دارو موزون (weighted) رانشان می دهد:



گره ها یا راس هانشان دهنده شهرها،

و یا لها نشان دهنده مسیر مستقیم بین دو شهر می باشند.

دریک گراف جهت دار،مسیر عبارتست از یک سری راس ،بطوریکه از یک راس به بعدی یک یال وجود داشته باشد.مثلاً $[V_1,V_4,V_3]$ یک مسیر است.مسیری ازیک راس به خود آن راس را،چرخه گویند.

مثلا مسیر $[V_1,V_4,V_5,V_1]$ یک چرخه است.

اگر مسیری هیچگاه دوبار ازیک راس نگذرد ،آن رامسیر ساده گویند.یعنی مسیر ساده هرگز حاوی زیر مسیری که چرخه‌ای باشد نیست .

طول مسیر دریک گراف موزون ،حاصلجمع وزنهای مسیر است.

مسئله یافتن کوتاهترین مسیر ازیک راس به راس دیگر ،کاربردهای فراوانی دارد. یکی از کاربردهای آن ،تعیین کوتاهترین مسیر میان دوشهر است.مسئله کوتاهترین مسیر یک مسئله بهینه سازی است.

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

(W)

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

(D)

W،گراف رانشان می دهد و D ،حاوی طول کوتاهترین مسیر است.

تعداد کل مسیر ها از یک راس که ازهمه رئوس دیگر بگذرد عبارتست است:

$$(n-2)(n-3)\ldots\ldots\ldots 1=(n-2)!$$

ابتدا الگویتمی ارائه می دهیم که طول مسیرها را به ما بدهد سپس آن را طوری اصلاح می کنیم کـه کوتـاهترین مسـیر را بما بدهد.یک گراف موزون حاوی n راس رابایک آرایه W نشان می‌دهند که درآن

$$w[i][j]=\begin{cases} \text{وزن یال} & \text{اگریالی بین } v_j, v_i \text{ باشد} \\ \infty & \text{اگریالی بین } v_j, v_i \text{ نباشد} \\ 0 & \text{اگر } i=j \text{ باشد} \end{cases}$$

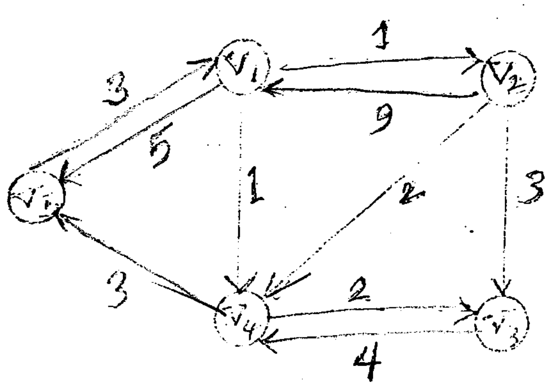
به این آرایه ،ماتریس هم جواری یک گراف می گویند.

$D^{(k)}[i][j]$: طول کوتاهترین مسیر از V_i به V_j فقط با استفاده از رئوس موجود د ر مجموعه $\{v_1, v_2, \ldots v_k\}$ بـه عنوان رئوس واسطه،

$D^{(k)}[i][j]$ را از رابطه زیر بدست می آوریم:

$$D^{(k)}[i][j]=\min imum(D^{(k)}[i][j], D^{(k)}[i][k]+D^{(k)}[k][j]$$

مثال ۳-۲) با داشتن گراف زیر ،که توسط ماتریس هم جواری W نشان داده شده است ، داریم:



	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

$D^{(0)}[2][5] = lenght[v_2, v_5] = \infty$

$D^{(1)}[2][5] = \min imum(lenght[v_2, v_5], lenght[v_2, v_1, v_5]) = \min imum(\infty, 14) = 14$

$D^{(1)}[2][4] = \min imum(D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4] = \min imum(2, 9+1) = 2$

$D^{(1)}[5][2] = \min imum(D^{(1)}[5][2], D^{(1)}[5][1] + D^{(0)}[1][] = \min imum(\infty, 3+1) = 2$

الگوریتم ۳-۳)الگوریتم فلوید برای کوتاهترین مسیر:

مسئله: محاسبه کوتاهترین مسیر ازهرراس دریک گراف موزون به رئوس دیگر.

ورودی: یک گراف جهت دار موزون، n تعداد رئوس موجود درگراف w[i][j]

خروجی :یک آرایه دو بعدی D که D[i][j] وزن کوتاهترین مسیر میان راس i ام و راس j ام است .

```
void    floyd    (int  n;
                   const  number   w[][];
                          number   D[][])

{  index   i, j, k;
   D = w;
    for   (k = 1; k <= n; k + +)
          for (i = 1;  i <= n;  i + +)
                for (j = 1; j <= n; j + +)
                      D[i][j] = minimum(D[i][j], D[i][k] + D[k][j]);
}
```

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم فلوید:

عمل اصلی:دستورهای موجود درحلقه for

اندازه ورودی: n تعداد رئوس گراف

$T(n) = n \times n \times n = n_3 \in \theta(n_3)$

۳-۳)دنباله فیبوناچی:

اعداد فیبوناچی از فرمول زیر می توان محاسبه کرد:

$$fib(n)=\begin{cases} n & n=0 \text{ یا } n=1 \\ \mathrm{fib(n-1)}+\mathrm{fib(n-2)} & n>1 \end{cases}$$

الگوریتم ۳-۴)محاسبه دنباله فیبوناچی به روش تقسیم و حل:

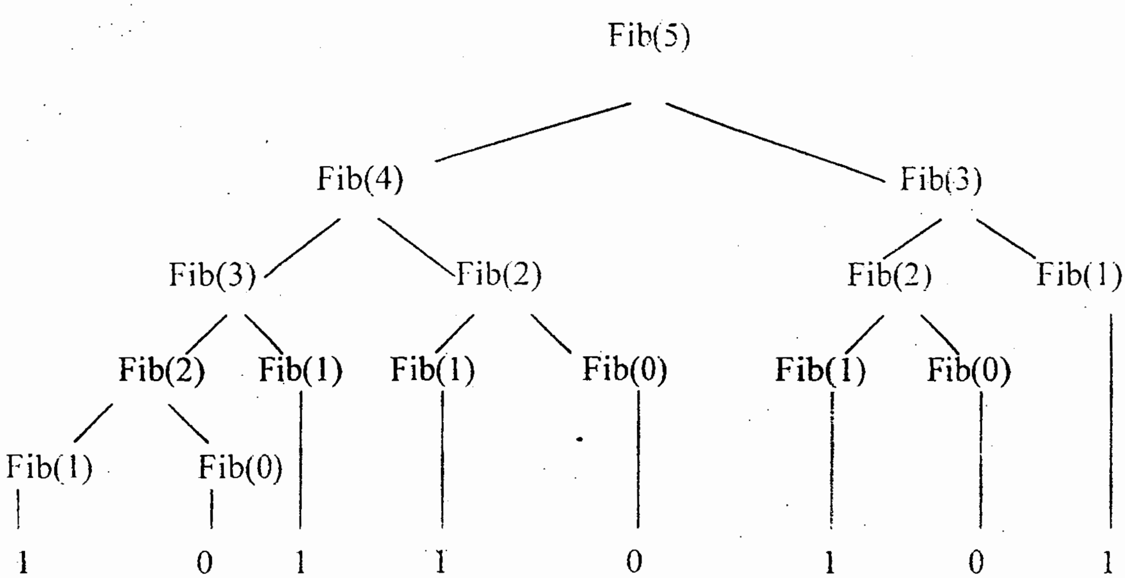
مسئله :محاسبه جمله n ام سری فیبوناچی

ورودی:n

خروجی :fib، جمله nام سری

```
int   fib (int  n)
{
    int  i :
   if  (i = 0   OR    i = 1 )    return    i; )
   else      return    (fib (i - 1)  +  fib (i - 2);
}
```

فراخوانی الگوریتم فوق به ازای n=5 طی مراحل مختلف ، بصورت زیر با یک درخت نشان داده می شود:



چنانچه ملاحظه می کنید برخی محاسبات تکراری هستند مثلاً fib(2) سه بار محاسبه می شود.

برای پرهیز ازتکرار محاسبات ،آنرا به روش برنامه سازی پویا حل می کنیم :

الگوریتم ۳-۵)محاسبه سری فیبوناچی با روش برنامه نویسی پویا:

مسئله : محاسبه جمله n ام سری فیبوناچی

ورودی: n

خروجی: fib

```
DpFib (int  n,
        const  number  Fib[])
{
    index  i;
    fib [0] ← 0;
    fib[1] ← 0;
    for(i = 2; i <= n; i + +)
        fib[i] = fib[i - 1] + fib[i - 2];
}
```

این الگوریتم دارای پیچیدگی زمانی $\theta(n)$ است درصورتی که الگوریتم (۳-۴)ازمرتبه توانی بود.

۳-۴)سریهای جهانی:

مسابقه ای بین دوتیم AوB با شرایط زیربرگزاری می گردد:

الف)حداکثر 2n-1 بر بازی صورت می گیرد.

ب) تیمی برنده است که n برد داشته باشد.

ج) بازی طوری است که تساوی ندارد.

د) نتیجه هر بازی مستقل ازبازیهای دیگر است یعنی احتمال برد درهربازی برای هرکدام ثابت است و به نتایج بازیهای قبلی ارتباطی ندارد.

ه) احتمال برد تیم A در هر بازی مقدار ثابت p و احتمال بردB برابر با q=1-p است.

فرض کنید p(i,j) احتمال برنده شدن تیم A درمسابقه باشد به شرطی که تیمA نیاز بـه iبرد دیگر وتیم B نیاز بـه j برددیگر داشته باشد.

درآغاز مسابقه احتمال اینکه A برنده نهایی باشد، p(n,n) است.

بدیهی است p(0,i)=1 و p(i,0)=0 می باشد.و p(0,0) تعریف نشده است.

طبق قانون احتمالات داریم:

$$p(i,j)=pP(i-1,j)+qP(i,j-1)$$

حل این مسئله با روش تقسیم و حل به صورت زیر است:

الگوریتم ۳-۹-۶)مسابقه بین دو تیم B,A

مسئله :تعیین برنده مسابقه به روش تقسیم و حل .

ورودی :اعداد صحیح ومثبت j,i

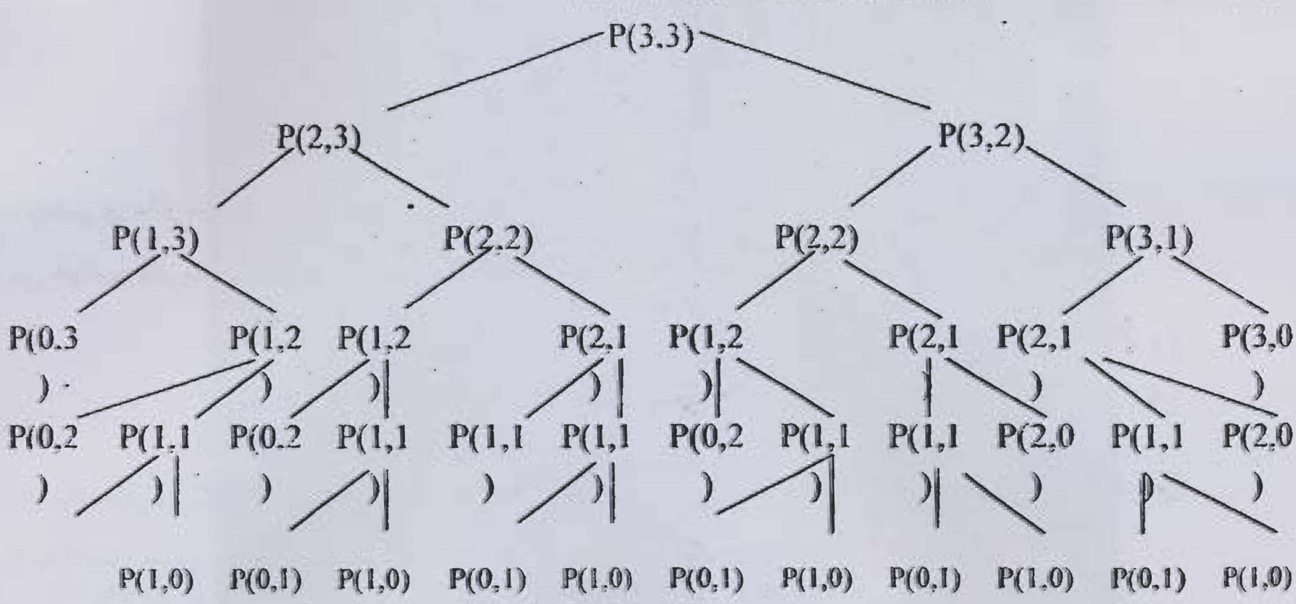
خروجی: p ،احتمال برنده شدن تیم A

```
int    (int i ,int j)
{
    if (i = 0)    return  1;
      else  if (j = 0)   return 0;
            else   return (pP(i - 1, j) + qp(i, j - 1));
}
```

فرض کنید T(k) ،زمان لازم برای محاسبه p(i,,j) دربدترین حالت باشد بطوریکه k=i+j . با این روش داریم:

$$T(1) = c$$
$$T(k) \le 2\ T(k-1) + d \qquad k > 1$$
$$T(k-1) \le 2\ T(k-2) + d$$
$$\Rightarrow T(k) \le 4\ \ T(k-1) + 2d + d \quad k > 2$$
$$\vdots$$
$$\le 2^{k-1} T(1) + (2^{k-2} + 2^{k-3} + \dots\dots\dots + 2 + 1)d$$
$$= 2^{k-1} c + (2^{k-1} - 1)d$$
$$= 2^{k} (c/2 + d/2) - d \Rightarrow T(k) \approx \theta(2^{k})$$

اگر این الگوریتم رابرای p(3,3) بااستفاده از درخت دنبال کنیم،داریم:



الگوریتم ۳-۹-۶)مسابقه بین دو تیم B,A

مسئله :تعیین برنده مسابقه به روش تقسیم و حل .

ورودی :اعداد صحیح ومثبت j,i

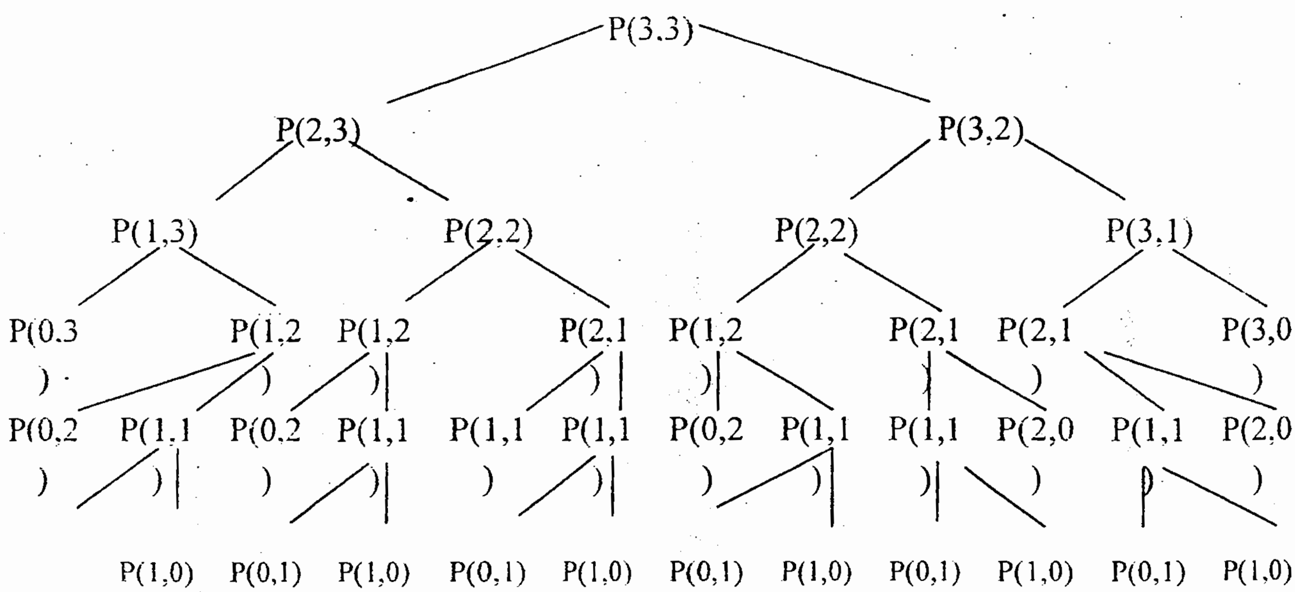
خروجی: p ،احتمال برنده شدن تیم A

```
int      (int  i ,int  j)
{
    if  (i = 0)     return   1;
       else   if (j = 0)    return 0;
              else    return (pP(i - 1, j) + qp(i, j - 1));
}
```

فرض کنید T(k) ،زمان لازم برای محاسبه p(i.,j) دربدترین حالت باشد بطوریکه k=i+j . با این روش داریم:

$$T(1) = c$$
$$T(k) \le 2\ T(k-1) + d \qquad k > 1$$
$$T(k-1) \le 2\ T(k-2) + d$$
$$\Rightarrow T(k) \le 4\ \ T(k-1) + 2d + d \quad k > 2$$
$$\vdots$$
$$\le 2^{k-1} T(1) + (2^{k-2} + 2^{k-3} + \dots\dots + 2 + 1)d$$
$$= 2^{k-1} c + (2^{k-1} - 1)d$$
$$= 2^{k}(c/2 + d/2) - d \Rightarrow T(k) \approx \theta(2^{k})$$

اگر این الگوریتم رابرای p(3,3) بااستفاده از درخت دنبال کنیم،داریم:



همانطوریکه ملاحظه می کنید بسیاری ازمحاسبات تکراری هستندمثلاً $p(1,0)$ ،۶بارمحاسبه شده است.این تکرارها باعث بالا رفتن زمان اجرای الگوریتم و کاهش بازدهی آن می شوند.

برای بهبود این وضع ازروش برنامه نویسی پویا استفاده می کنیم.

الگوریتم ۳-۷)مسابقه بین دو تیم B,A

مسئله : تعیین برنده مسابقه بروش برنامه نویسی پویا

ورودی: n

خروجی: [ ]P

```
void   series  (int p.int  n)
{    int p , q , s , k ; i
     array   p[0..n,0..n];
      q = 1 - p;
        for (s = 1, s <= n; s + +)
            p[0, s] = 1;  p[s,0] = 0;
            for    (k = 1, k <= s - 1; k + +)
                   p[k, s - k] = pP[k - 1, s - k] + q[k, s - k - 1];
        for (s = 1; s <= n; s + +)
            for (k = 0; k <= n - s; k + +)
                   p[s + k, n - k] = pP[s + k - 1, n - k] + qP[s - k, n - k - 1];
        return    p[n, n];
}
```

۳-۵)ضرب زنجیر ی ماتریسها:

فرض کنید می خواهیم ماتریس ۳×۲ را دریک ماتریس ۴×۳ به صورت زیر ضرب کنیم:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}\begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix}=\begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 84 \end{bmatrix}$$

ماتریس حاصل ،یک ماتریس ۴×۲ است.اگرازروش استاندارد ضرب ماتریسها استفاده کنیم،برای محاسبه هرعنصر ازماتریس حاصلضرب ۱ به ۳ عمل ضرب نیاز داریم.پس تعداد کل اعمال ضرب که نیازداریم برابر است با:۲۴=۳×۴×۲

بطورکلی برای ضرب ~~ماتریس~~ ماتریس $i\times j$ دریک ماتریس $j\times k$ بااستفاده از ضرب استاندارد،تعداد اعمال ضرب لازم $i\times j\times k$ می باشد.

ضرب چهار ماتریس زیر رادرنظر بگیرید:

$$\begin{array}{ccccccc} A & \times & B & \times & C & \times & D \\ 20\times 2 & & 2\times 30 & & 30\times 12 & & 12\times 8 \end{array}$$

ضرب ماتریسها .یک عمل شرکت پذیر است،یعنی ترتیب انجام ضربها اهمیت ندارد.برای مثال، A(B(CD)) و(AB)(CD)هردو یک نتیجه می دهند .چهار ماتریس رابه پنج ترتیب متفاوت می توان درهم ضرب کرد که در هر یک تعداد ضربهای انجام شده بادیگری متفاوت است:

$$A(B(CD)): 30\times 12\times 8+2\times 30\times 8+20\times 2\times 8=3680$$

$$(AB)(CD): 20\times 2\times 30+30\times 12\times 8+20\times 30\times 8=8880$$

$$A((BC)D): 2\times 30\times 12+2\times 12\times 8+20\times 2\times 8=1232$$

$$((AB)C)D: 20\times 2\times 30+20\times 30\times 12+20\times 12\times 8=10320$$

$$(A(BC))D: 2\times 30\times 12+20\times 2\times 12+20\times 12\times 8=3120$$

ترتیب سوم برای ضرب این چهار ماتریس ،ترتیب بهینه است.

هدف ما بسط الگوریتمی است که ترتیب بهینه رابرای n ماتریس معین کند.ترتیب بهینه فقط به ابعاد ماتریسها بستگی دارد.بنابراین علاوه بر n ،این ابعاد تنها ورودیهای الگوریتم هستند.

فرض کنید t_n تعداد ترتیبهای متفاوت برای ضرب n ماتریس $A_1, A_2, \ldots\ldots, A_n$ دریکدیگر باشد:

$$\left.\begin{array}{l} t_2=1 \\ t_n\ge 2t_{n-1} \end{array}\right. \Rightarrow t_n\ge 2^{n-2}$$

حداقل تعداد در ضربهای لازم برای ضرب A_i تا A_j (اگر $i<j$) برابر است با: m[i][j]

مثال ۳-۳)فرض کنید شش ماتریس زیر را داریم:

$$\begin{array}{ccccccccccc} A_1 & \times & A_2 & \times & A_3 & \times & A_4 & \times & A_5 & \times & A_6 \\ 5\times 2 & & 2\times 3 & & 3\times 4 & & 4\times 6 & & 6\times 7 & & 7\times 8 \\ d_0\ d_1 & & d_1\ d_2 & & d_2\ d_3 & & d_3\ d_4 & & d_4\ d_5 & & d_5\ d_6 \end{array}$$

برای ضرب A_6, A_5, A_4 دو ترتیب زیررا به همراه تعداد اعمال ضرب آنها داریم:

$d_3 \times d_5$

$(\overline{A_4 A_5})A_6:$ تعداد ضربها $= d_3 \times d_4 \times d_5 + d_3 \times d_5 \times d_6 = 4 \times 6 \times 7 + 4 \times 7 \times 8 = 392$

$A_4(\underline{A_5 A_6}):$ تعداد ضربها $= d_4 \times d_5 \times d_6 + d_3 \times d_4 \times d_6 = 6 \times 7 \times 8 + 4 \times 6 \times 8 = 528$

$d_4 \times d_6$

بنابراین : $M[4][6] = \text{minimum}(392,528) = 392$

ترتیب بهینه برای ضرب شش ماتریس باید یکی ازحالات زیر باشد:

$1.A_1(A_2A_3A_4A_5A_6)$

$2.(A_1A_2)(A_3A_4A_5A_6)$

$3.(A_1A_2A_3)(A_4A_5A_6)$

$4.(A_1A_2A_3A_4)(A_5A_6)$

$5.(A_1A_2A_3A_4A_5)(A_6)$

تعداد ضربها برای حالت k ام بصورت زیر است:

$m[1][k] + m[k+1][6] + d.d_k d_6$

ضرب A_1 تا A_6 $= m[1][6] = \underset{1 \le k \le 5}{\text{minimum}}(m[1][k] + m[k+1][6] + d.d_k d_6)$

$m[1][1] + m[2][6] + d \cdot d_1 d_6$ $k=1$
$m[1][2] + m[3][6] + d.d_2 d_6$ $k=2$
$m[1][3] + m[4][6] + d.d_3 d_6$ $k=3$
$m[1][4] + m[5][6] + d d_4 d_6$ $k=4$
$m[1][5] + m[6][6] + d d_5 d_6$ $k=5$

$\Rightarrow m[i][j] = \underset{i \le k \le j-1}{\text{minimum}}(m[i][k] + m[k+1][j] + d_{i-1} d_k d_j)$ اگر $i < j$

الگوریتم ۳-۸)حداقل ضربها:

مسئله : تعیین حداقل تعدادضربهای اصلی موردنیاز برای ضرب n ماتریس ،وترتیبی که حداقل تعداد رابدست می دهد.

ورودی :تعداد ماتریس n، آرایه ای ازاعداد صحیح، d که از صفر تا n اندیس گذاری شده است.

خروجی : minmult ، حداقل تعداد ضرب وآرایه دوبعدی p که ترتیب بهینه رابه ما می دهد.

```
int        minmult (int  n ,
                    const  int d[ ],
                    index  p[ ][ ])
{
  index i, j, k, diagonal;
  int  m[1..n][1..n];
  for (i = 1; i <= n; i + +)
     m[i][i] = 0;
  for (diagonal = 1; diagonal <= n - 1; diagonal + +)
     for (i = 1; i <= n - diagonal; i + +)
      {
        j = i + diagonal;
       m[i][j] = minmum(m[i][k] + m[k + 1][j] + d[i - 1] * d[k] * d[j]);
                 i≤k≤j-1
       p[i][j] = a  valve of k  that  gave the minmum;
return   m[1][n];
}
```

تعدادکل دفعاتی که عمل اصلی انجام می شود از مرتبه $\frac{n(n-1)(n+1)}{6} \in \theta(n^3)$ است.

الگوریتم ۳-۹ چاپ ترتیب بهینه

مسئله : چاپ ترتیب بهینه برای ضرب nماتریس

ورودی :عدد صحیح ومثبت n ، آرایه p ، p[i][j] نقطه ای است که ترتیب بهینه رامی دهد.

خروجی :ترتیب بهینه ضرب ماتریسها

```
void   order  (index i , index j)
{
   if (i == 1)  cout <<  "A" << i;
       else {
           k = p[i][j];
          cout <<   "(";
          order  (i.k);
          order (k + 1, j);
         cout <<  ")";
        }
   }
```

برای الگوریتم فوق $T(n) \in \theta(n)$ است.

۳-۶)درختهای جستجوی دو دویی بهینه :

تعریف: درخت جستجوی دودویی ، یک درخت دودویی ازعناصر (که معمولاً کلیدنامیده می شوند) است که از یک مجموعه مرتب حاصل می شود،به طوری که:

۱- هرگره حاوی یک کلیداست.

۲- کلیدهای موجود درزیر درخت چپ یک گره مفروض ،کوچکتر یامساوی کلید آن گره هستند.

۳- کلیدهای موجود درزیر درخت راست یک گره مفروض ،بزرگتر یامساوی کلید آن گره هستند.

عمق یک گره دردرخت برابر با تعداد یالها درمسیر منحصر به فردی از ریشه به آن گره است.این ویژگی راسطح گره نیز می نامند.

عمق درخت :عبارت ازحداکثر عمق همه گرههای درخت است.

هدف سازماندهی کلیدها دریک درخت جستجویی دودویی است بطوریکه زمان میانگین برای تعیین مکان کلیدهابه حداقل برسد.درختی که به این شیوه سازماندهی می شود، درخت بهینه نام دارد.

حالتی رابررسی می کنیم که معلوم است کلید دردرخت موجود می باشد.

الگوریتم ۳-۱۰)درخت جستجوی دودویی:

مسئله : تعیین گره حاوی یک کلید دردرخت دودویی

ورودی :یک اشاره گر tree به درخت جستجوی دودویی وکلید keyin

خروجی :اشاره گر p به گره حاوی کلید.

```
Struct nodetype
{
  keytype key;
 nodetype * left;
 nodetype *right;
};
typedef    nodetype * node-pointer;
void search (node-pointer tree,
                 keytype  keyin,
                 node - pionter & p)
{
bool found;
```

```
 p=tree;
found =false;
while(found)
   :if (p ›     key= = keyin) found =True;
   else if (keyin <p ›      key) p=p ›    left:
        else p=p › right;
}
```

تعداد مقایسه های انجام شده برای یافتن کلید رازمان جستجو گویند.هدف تعیین درختی است که زمان جستجوی میانگین برای آن حداقل باشد.زمان جستجو برای یک کلید مفروض عبارتست از: depth(key)+1 (عمق گره حاوی کنید)

فرض کنید p_i احتمال مساوی بودن کلید key_i با کلید مورد جستجو و c_i تعداد مقایسه های موردنیاز بـرای یـافتن key_i باشد زمان جستجوی میانگین برای آن درخت بصورت زیر است:

$$\sum_{i=1}^{n} c_i p_i$$

الگوریتم ۳-۱۱) درخت جستجوی دودویی بهینه :

مسئله :تعیین یک درخت جستجوی بهینه

ورودی: n تعداد کلید ها و pآرایه ای از اعداد حقیقی .

خروجی:متغیر minavg که مقدار آن زمان جستجوی میانگین برای یک درخت جستجوی دودویی بهینه است. و آرایه دو بعدی R که از روی آن یک درخت بهینه می توان ساخت.

```
Void optsearchtree (int n,
                    Const float p[ ],
                    Float & minavg,
                    Index R [ ][ ])
{
index i.j.k.diagonal;
float A[1..n+1][0…n];
for (i=1;i<=n;i++)
{
     A [i][i-1]=0;
     A[i][i]=p[i];
    R[i][i]=i;
    R[i][i-1]=0;
}
  A[n+1][0]=0;
  R[n+1][n]=0;
```

```
For (diagonal=1;diagonal <=n-1;diagonal++)
    For (i=1;i<=n-diagonal;i++)
    {
    j=i +diagonal;
    A[i][j]=minimum(A[i][k-1]+A[k+1][j])+ Σ(m=i..j) p_m;
            i≤k≤j
    R[i][j]=a value of k that gave the minimum;
 }
    minavg =A[i][j];
}
```

پیچیدگی زمانی این الگوریتم درهر حالت چنین است:

$$T(n)=\frac{n(n-1)(n+4)}{6}\in\theta(n^3)$$

الگوریتم ۳-۱۲)ساخت درخت جستجوی دودویی بهینه:

مسئله : ساخت درخت جستجوی دودویی بهینه

ورودی : n آرایه key، آرایه R

خروجی: اشاره گر tree

```
node-pointer tree(index i,j)
{
index k;
node-pointer p;
k=R[i][j];
if (k= =0) return Null;
else {
     p=new nodetype;
      p› key=key[k];
     p› left =tree (i,k-1);
     p› right=tree (k+1,j);
   return p;
  }
}
```

فصل چهارم :

روش حریصانه

روش حریصانه شاید سرراست ترین روش طراحی الگوریتم باشد.این الگوریتم به شیوه اسکروج عمل می کند.اسکروج،شخصیت داستانی چارلز دیکنز،شاید حریصترین فردی است که تاکنون دیده ایم.او هرگز به آینده یا گذشته نمی اندیشید.هرروز تنهاانگیزه ای که داشت به چنگ آوردن طلای بیشتر بود.الگوریتم حریصانه نیز چنین عمل می کند . یعنی به ترتیب عناصر داده راگرفته ، هربار آن عنصری راکه طبق معیاری معین "بهترین" به نظر می رسد، بدون توجه به انتخابهایی که قبلا انجام داده یا درآینده انجام خواهد داد ،برمی دارد.این الگوریتمها غالبا به راه حلهایی بسیار ساده و کارآمد منجر می شوند.

الگوریتم های حریصانه ،مانندبرنامه نویسی پویا ، برای حل مسائل بهینه سازی بکار می روند،ولی روش حریصانه صراحت بیشتری دارد، در برنامه نویسی پویا، از یک ویژگی پویا برای تقسیم نمونه ای به نمونه های کوچکتر استفاده می شود.درروش حریصانه ، تقسیم به نمونه های کوچکتر صورت نمی گیرد.الگوریتم حریصانه با انجام یک سری انتخاب ،که هر یک درلحظه ای خاص ، بهترین به نظر می رسد، عمل می کند.یعنی انتخاب درجای خود بهینه است.امید این است که یک حل بهینه سراسری یافت شود.با یک مثال ساده روش حریصانه رانشان می دهیم.

۴-۱)مسئله خرد کردن پول:

فروشنده یک فروشگاه،غالبا برای دادن بقیه پول به خریدار ، دچار مشکل می شود.مشتریان معمولا مایل نیستند مقدارزیادی پول خرد بگیرند،بنابراین هدف فروشنده نه تنها دادن بقیه پول به میزان صحیح ، بلکه انجام این کار با حداقل تعداد سکه ممکن است.

یک حل برای نمونه ای از این مسئله عبارت است ازمجموعه ای از سکه ها که جمع آنها معادل بقیه پول مشتری شده است.حل بهینه همین مجموعه ، با حداقل تعداد سکه ها انجام می شود.الگوریتم حریصانه برای این مسئله چنین عمل می کند:

در آغاز هیچ سکه ای در مجموعه نداریم .فروشنده بزرگترین سکه (ازلحاظ ارزش) راپیدا می کند.یعنی ملاک وی برای اینکه کدام سکه بهترین است (بهینه محلی)ارزش سکه است.این رادر الگوریتم حریصانه ،"روال انتخاب" می نامند.سپس باید دیدکه آیا باافزودن این سکه به بقیه پول ،جمع کل آنها از چیزی که بایدباشد ، بیشترمی شود یا خیر .این را در الگوریتم حریصانه ،" تحقیق عملی بودن" می نامند.اگر با افزودن این سکه بقیه پول از میزان لازم بیشتر نشود،این سکه به مجموعه اضافه می شود .سپس تحقیق می کند تا ببیند که آیا مقداربقیه پول با میزان لازم برابر شده است یا خیر .این موضوع را در الگوریتم حریصانه ،"تحقیق حل شدن "می گویند.اگر برابر نبودند ، با استفاده از روال انتخاب،یک سکه دیگر انتخاب می کند

و فرایند تکرار می شود. او چندین بار این کار را انجام می دهد تا مقدار بقیه پول با میزان لازم برابر شود یا اینکه دیگر سکه ای باقی نماند.

یک الگوریتم سطح بالا برای این روال در زیر خواهد آمد:

```
while (there are more coins and the instance is not  solved)
{
Grab the largest remaining coin;                "selection procedure
   If (adding the coin makes the change exceed  the amount owed)
          Reject the coin;                        "feasibility check
Else  add the coin to the  change;
 If ( the total value of the change equals the amount owed)        "solution check
         The instance  is solved;
}
```

مثال ۴-۱)مسئله خرد کردن پول به روش حریصانه :

فرض کنید فروشنده سکه های زیر را دارد:

سکه ۲۵ تومانی ۱ عدد

سکه ۱۰ تومانی ۲ عدد

سکه ۵ تومانی ۱ عدد

سکه ۱ تومانی ۲ عدد

حال می خواهد باقی مانده پول مشتری که ۳۶ تومان است را بپردازد.

برای حل این مسئله به روش حریصانه چنین عمل می کنیم:

۱- یک سکه ۲۵ تومانی می گیریم ، کمتر از ۳۶ است پس به مجموعه اضافه می کنیم .

۲- یک سکه ۱۰ تومانی می گیریم ،مجموع ۳۵ است و کمتر از ۳۶ ، پس به مجموعه اضافه می کنیم.

۳- سکه ۱۰ تومانی دوم را می گیریم ،چون مجموع ۴۵ می شود و از ۳۶ بزرگتر است ، آن سکه راکنار می گذاریم .

۴- سکه ۵ تومانی رامی گیریم ،چون مجموع ۴۰ می شود واز ۳۶ بزرگتر است ،آن سکه را کنار می گذاریم .

۵- سکه ۱ تومانی رامی گیریم ،مجموع ۳۶ می شودو بیشتر از ۳۶ نیست ،پس آن را به مجموعه اضافه می کنیم و چون مجموع سکه ها برابر با میزان لازم است ،این مسئله حل شده و کارتمام است.

پس برای پرداخت ۳۶ تومان،به یک سکه ۲۵ تومانی،یک سکه ۱۰تومانی و یک سکه ۱ تومانی نیاز است .

به طور خلاصه ، الگوریتم حریصانه، کار را با یک مجموعه تهی شروع کرده، به ترتیب عناصری به مجموعه اضافه می کندتا این مجموعه حلی برای نمونه ای از یک مسئله را نشان دهد.هر دور تکرار شامل مولفه های زیر است:

۱- روال انتخاب :عنصر بعدی را که باید به مجموعه اضافه شود، انتخاب می کند.انتخاب طبق یک ملاک حریصانه اجـرا مـی شود که یک شرط بهینه را درهمان برهه برآورده می سازد.

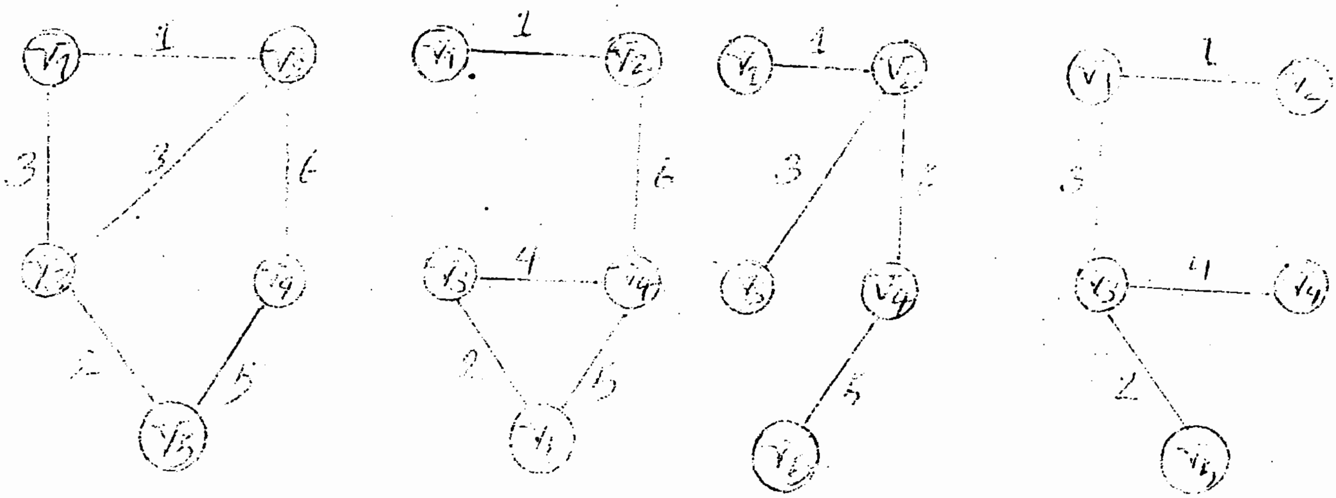
۲- تحقیق عملی بودن: تعیین می کند که آیا مجموعه جدید برای رسیدن به حل عملی است یاخیر .

۳- تحقیق حل :تعیین می کند که آیا مجموعه جدید،حل نمونه را بدست می دهدیاخیر .

۴-۲)درختهای پوشای کمینه

فرض کنید طراح شهری می خواهد چندشهر معین را با جاده به هم متصل کند.به طوری که مردم بتوانند ازهر شهر به شـهر دیگر بروند.اگر محدودیت های بودجه ای درکار باشد،ممکن است طراح بخواهد این کار را باحداقل مقدار جاده کشـی انجـام دهد.حال الگوریتم بسط می دهیم که این مسئله و مسائل مشابه راحل کند.

ابتدا نظریه گراف ها را مرور می کنیم .شکل (الف)گراف متصل ،بدون جهت و موزون G رانشان مـی دهـد.مسیر ، درگـراف بدون جهت عبارت است از یک سری راس که بین هر کدام از آنها و راس مجاور، یک یال وجود داشته باشد.چون یالـها فـاقد جهت هستند،از راس u به راس v یک مسیر وجوددارد اگر و فقط اگر مسیری از v بـه u وجـود داشـته باشـد. گـراف بـدون جهت را **متصل** گوییم ،اگر بین هر جفت از رئوس آن یک مسیر وجود داشته باشد.



(الف)گراف متصل،موزون و بدون جهت G (ب)گراف متصل (ج)درخت پوشا برای G (د)درخت پوشای کمینه برای G

درشکل (ب) اگر یال میان v_5 و v_4 را حذف کنیم گراف متصل باقی می ماند ولی اگر یال بین v_4 و v_2 حذف کنیـم دیگـر گراف، متصل نخواهد بود.

دریک گراف بدون جهت ،همانند گراف جهت دار، مسیری از یک راس به خودش ، چرخه نامیده می شود.

گراف های ج و د بی چرخه هستند ،درحالی که گراف های الف و ب بی چرخه نیستند.

درخت یک گراف بدون جهت متصل و بی چرخه است.

درخت پوشا برای گراف G ، یک زیر گراف متصل است .که حاوی همه رئوس موجود در G بوده و یک درخت می باشد.درختهای ج و د ، درخت پوشا هستند . یک زیر گراف متصل با وزن کمینه باید یک درخت پوشا، باشد ولی هر درخت پوشا دارای وزن کمینه نیست.

درخت پوشای کمینه ،درخت پوشایی است که دارای وزن کمینه باشد.هدف ما طراحی الگوریتمی برای ایجاد درخت پوشای کمینه است

تعریف : گراف بدون جهت G شامل یک مجموعه متناهی V ، که اعضای آن را رئوس G می نامند ،ویک مجموعه از جفت رئوس در V است . این جفت ها را یالهای G می گویند .G را به صورت زیر نشان می دهیم :

G=(V,E)

اعضای V [illegible] و (V,E) [illegible] نشان می دهیم .

مثال [illegible] برای گراف [illegible] داریم:

$\nu = \{\nu_1, \nu_2, \nu_3, \nu_4, \nu_5\}$

$E = \{(\nu_1, \nu_2), (\nu_1, \nu_4), (\nu_2, \nu_3), (\nu_2, \nu_4), (\nu_3, \nu_5), (\nu_4, \nu_5)\}$

ترتیب ذکر کردن رئوس برای مشخص کردن یالها در یک گراف بدون جهت اهمیتی ندارد.

یک درخت پوشای T برای G ، دارای همان مجموعه رئوس V است که در G داریم ، ولی مجموعه یالهای T زیر مجموعه F از E است درخت پوشا را با T=(V,F) نشان می دهیم . مسئله ما یافتن زیر مجموعه F از E است به طوری که T=(V,F) یک درخت پوشای کمینه برای G باشد یک الگوریتم حریصانه سطح بالا برای چنین مسئله ای می تواند به صورت زیر باشد.

```
F=ф;                                        // initialize set of edges to empty.
While  (the  instance is not solved)
{
select an edge according to some  locally optimal consideration;  // selection procedure
if  (adding the edge of F does  not  create a cycle)add it        // feasibility  check
if (T=(V,F) is a spaning tree) the  instance is solved           // solution check
}
```

برای این مسئله دو الگوریتم حریصانه متفاوت را بررسی خواهیم کرد،الگوریتم پریم و الگوریتم کروسکال .

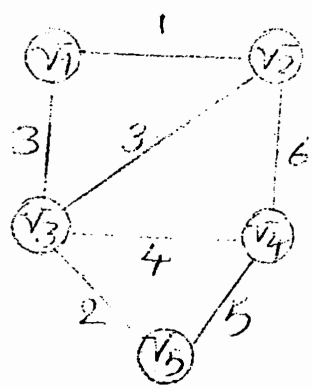
هر یک از این الگوریتم ها از یک ویژگی بهینه محلی استفاده می کند.الگوریتمهای کروسکال و پریم ،همواره درخت های پوشای کمینه را ایجاد می کند.

۴-۲-۱)الگوریتم پریم (Prime Algorithm)

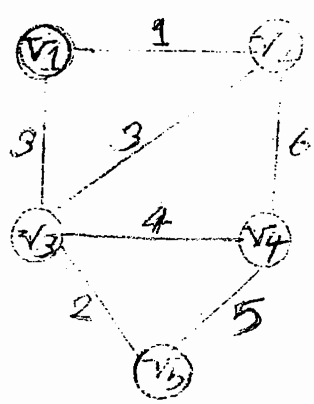
الگوریتم پریم با زیر مجموعه ای تهی از یالهای F و زیر مجموعه از رئوس Y آغاز می شود، زیر مجموعه Y حاوی یک راس دلخواه است. به عنوان مقدار اولیه،$\{v_1\}$ رابه Y می دهیم . نزدیکترین راس به Y ، راسی در V-Y است که توسط یالی با وزن کمینه به راسی در Y متصل است.راسی که از همه به Y نزدیکتر است ،رابه Y و یال مربوط را به F اضافه می کنیم .این فرایند را آنقدر ادامه می دهیم تا Y=V شود ، یک الگوریتم سطح بالا برای این روال به شرح زیر است.

```
F=φ;                                          //initialize set of edges to empty.
Y={v1};                                       //initialize set of vertics to
                                              //contain only the first one.
While(the instance is not solved)
{
  select a vertex in V-Y that is nearest to Y;   //selection procedure & feasiblity check.
Add the vertex to Y;
Add the edge to F;
If (Y=V) the instance is solved.                 //selection check.
}
```

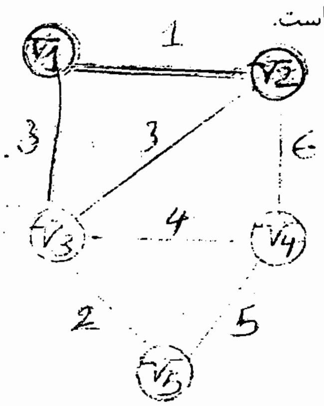
شکل زیر الگوریتم پریم را نشان می دهد:(درهر مرحله از شکل ،Y حاوی رئوس سایه زده شده و F حاوی یالهای ضخیم شده است.



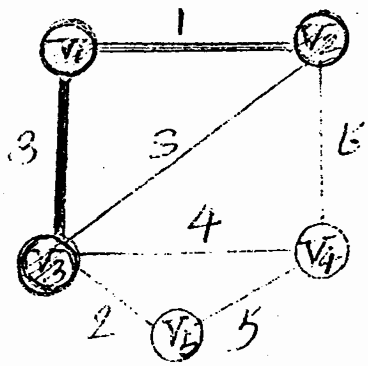
مسئله :تعیین درخت پوشای کمینه



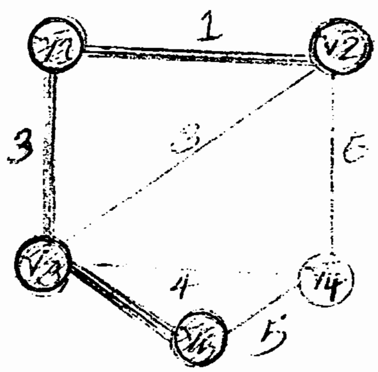
۱- ابتدا راس v_1 انتخاب می شود



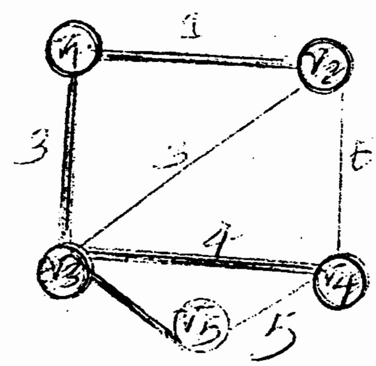
۲- راس انتخاب می شود زیرانزدیکترین راس به $\{v_1\}$ است.



۳- راس v_3 انتخاب می شود زیرا نزدیکترین راس به $\{v_1, v_2\}$ است.



۴- راس v_5 انتخاب می شود زیرا نزدیک ترین به $\{v_1, v_2, v_3\}$ است



۵- راس v_4 انتخاب می شود

گراف موزون را توسط ماتریس هم جواری آن نشان می دهیم، یعنی آن را با یک آرایه $n \times n$ از اعداد نشان می دهیم که در آن :

$$w[i][j] = \begin{cases} \text{وزن یال} & \text{اگر بین } v_j , v_i \text{ یک یال باشد} \\ \infty & \text{اگر بین } v_j , v_i \text{ یک یال نباشد} \\ 0 & \text{اگر } i = j \text{ باشد} \end{cases}$$

	1	2	3	4	5
1	0	1	3	∞	∞
2	1	0	3	6	∞
3	3	3	0	4	2
4	∞	6	4	0	5
5	∞	∞	2	5	0

دو آرایه نیاز داریم :

$nearest[i] = v_i$ اندیس نزدیکترین راس در Y به

distance $[i]$ = وزن یال میان vi و راسی که توسط $nearest[i]$ اندیس شده است

الگوریتم ۴-۱) الگوریتم پریم :

مسئله : تعیین یک درخت پوشای کمینه

ورودی : عدد صحیح $n \geq 2$ و یک گراف متصل ، موزون و بدون جهت حاوی n راس، آرایه W.

خروجی : مجموعه یالهای F در یک درخت پوشای کمینه برای گراف.

```
Void prim (int n, const number w [ ] [ ], set- of – edges & F)
{
        index    i, vnear ;
        number min ;
        edge e ;

        index nearest [2..n] ;
        number distance [ 2..n] ;
        F = Φ ;
        For (i = 2 ; i<n; i++)
        {
        nearest [i] = 1;
        distance [i] = w [1][1] ;
        }
        repeat (n-1 times)
        {
                min = ∞ ;
                for (i = 2; i <= n; i ++)
                        if (0 ≤ distance [i]<min)
                        {
                                min = distance [i];
                                vnear = i ;
                        }
                        e = edge connecting vetrics indexed by vnear and nearest [vnear] ;
                        add e to f ;
                        distance [vnear] = -1;
                        for (i=2;i<=n;i++)
                        if (w[i] [vnear] <distance [i])
                        {
                        distance [i] = w [i] [vnear] ;
                        nearest [i] = vnear :
                }
        }
}
```

ـ پیچیدگی زمانی این الگوریتم عبارتست از :

$T(n) = ۲(n-۱)(n-۱) \in \theta(n^۲)$

قضیه ۴-۱) الگوریتم پریم همواره یک درخت پوشای کمینه تولید می کند.

۴-۲-۲) الگوریتم کروسکال :

این الگوریتم ، برای مسئله درخت پوشای کمینه ، با ایجاد زیرمجموعه های مستقلی از V آغاز می شود که برای هر راس یک زیر مجموعه و هر زیر مجموعه حاوی یک راس است. سپس یالها را طبق ترتیب نزولی وزن بررسی می کند. اگر یالی دو راس را در مجموعه های مستقل به هم متصل کند، آن یال اضافه می شود و دو زیرمجموعه با هم ادغام می شوند . این فرآیند آنقدر تکرار می شود تا همه زیرمجموعه ها در هم ادغام شوند. یک الگوریم سطح بالا برای این روال چنین است :

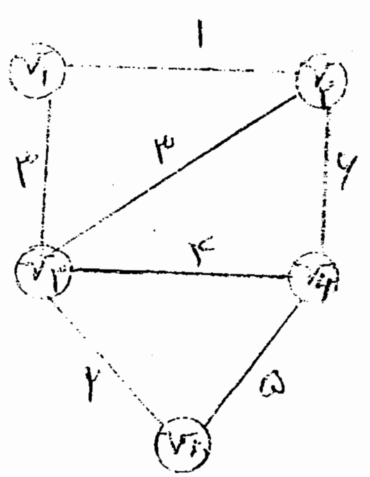
```
F = ϕ;          //initialize set of edges to empty.
Create disjoint of V, one for each vertex and containing only that vertex ;
Sort the edges in E in non decreasing order ;
While (The instance is not solred)
{
        Select next edge ;                    // selection procedure
        If (the edge connects two vertices in disjoint subsets)    //feasibility check
        {
                merge the subsets;
                add the edge to F;
        }
        if (all the subsets are merged)                    // solution check
             the instance is solved ;
}
```

حال با ذکر مثالی ، این الگوریتم را بهتر درک خواهید کرد.

مثال : برای گراف زیر درخت پوشای کمینه را به روش الگوریتم کروسکال بدست آورید.

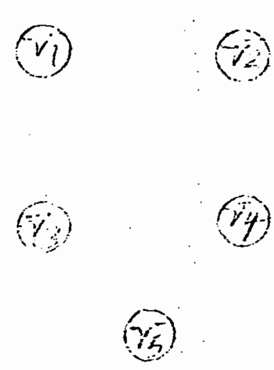
مراحل کاربصورت زیر است :

۱- یالها برحسب طول مرتب می شوند :

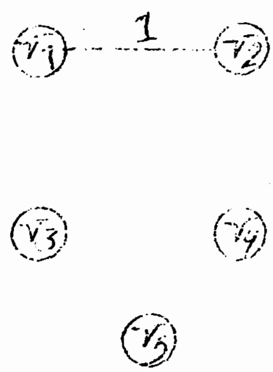


طول	یال
۱	$(v_{۱}, v_{۲})$
۲	$(v_{۳}, v_{۵})$
۳	$(v_{۱}, v_{۳})$
۳	$(v_{۲}, v_{۳})$
۴	$(v_{۳}, v_{۴})$
۵	$(v_{۴}, v_{۵})$
۶	$(v_{۲}, v_{۴})$

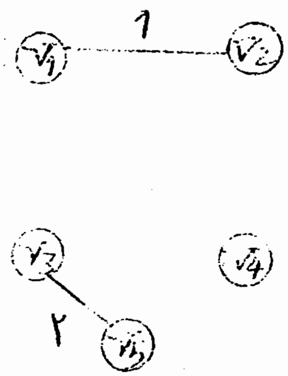
۲- مجموعه های مجزا ساخته می شوند .



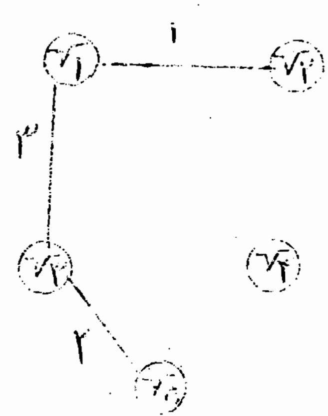
۳- یال (v_1, v_2) انتخاب می شود.



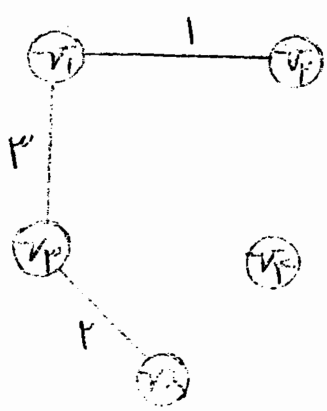
۴- یال (v_3, v_5) انتخاب می شود.



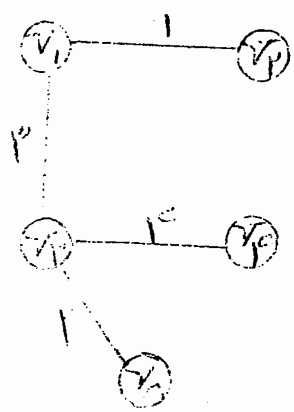
۵- یال (v_1, v_3) انتخاب می شود



۶- یال (v_2, v_3) انتخاب می شود.



۷- یال (v_3, v_4) انتخاب می شود.



مراحل را به صورت زیر نشان می دهند :

مرحله	یال مورد بررسی	مولفه های متصل
۰	———	$\{v_1\}$ $\{v_2\}$ $\{v_3\}$ $\{v_4\}$ $\{v_5\}$
۱	(v_1, v_2)	$\{v_1, v_2\}$ $\{v_3\}$ $\{v_4\}$ $\{v_5\}$
۲	(v_3, v_5)	$\{v_1, v_2\}$ $\{v_3, v_5\}$ $\{v_4\}$
۳	(v_1, v_3)	$\{v_1, v_2, v_3, v_5\}$ $\{v_4\}$
۴	(v_2, v_3)	ادغامی صورت نمی گیرد
۵	(v_3, v_4)	$\{v_1, v_2, v_3, v_4, v_5\}$

وقتی الگوریتم متوقف می شود، مجموعه F شامل همه یالها خواهد بود وطول درخت پوشای کمینه برابر با ۱۰ است.

الگوریتم ۴-۲) الگوریتم کورسکال :

مسئله : تعیین یک درخت پوشای کمینه :

ورودی : عدد صحیح $n \geq 2$ ، عدد صحیح و مثبت m ویک گراف بدون جهت، موزون و متصل حاوی n راس و m یال.

خروجی : F ، مجموعه ای از یالها در یک درخت پوشای کمینه

```
Void kruskal (int n , int m,
            Set – of – edges E,
            Set – of – edges & F)
{
     index  i,j;
     set – pointer p,q;
     edge e;
     sort the m edges in E by weight in nondecreasing order;
     F = Φ;
     Initial (n);          // initialize n disjoint subsets.
     While (number of edges in F is less than n-1)
     {
     e = edge with least weight not yet considered;
     i,j = indices of vertics connected by e ;
     p = find (i) ;
     q = Find (j) ;
     if (!equal (p,q))
     {
        merge (p,q);
        add e to F;
     }
   }
}
```

- پیچیدگی زمانی الگوریتم کروسکال فوق در بدترین حالت برابر است با $\theta(n^2 Log n)$

✓ قضیه ۴-۲) الگوریتم کروسکال همواره یک درخت پوشای کمینه تولید می کند.

۴-۲-۳) الگوریتم پریم در مقایسه با الگوریتم کروسکال :

پیچیدگی زمانی زیر را به دست آوردیم :

الگوریتم پریم : $T(n) \in \theta(n^2)$

الگوریتم کروسکال : $w(m,n) \in \theta(n^2 \lg n)$

تعداد یال (m)   تعداد راس (n)

همچنین نشان دادیم که در یک گراف متصل : $n-1 \le m \le \frac{n(n-1)}{2}$

برای گرافی که تعداد یالهای آن (m) نزدیک به کرانه پایینی این بازه باشد (گرافی که بسیار متراکم است)، الگوریتم کروسکال $\theta(n \lg n)$ است، یعنی الگوریتم کروسکال باید سریعتر باشد. ولی، برای گرافی که تعداد یالهای آن نزدیک به کرانه بالایی باشد(گراف بسیار متصل باشد) ، الگوریتم کروسکال $\theta(n^2 \lg n)$ است، یعنی الگوریتم پریم باید سریعتر باشد.

۴-۳) یافتن کوتاهترین مسیر توسط الگوریتم دیکسترا :

در فصل قبل یک الگوریتم $\theta(n^3)$ برای یافتن کوتاهترین مسیر از هر راس به همه رئوس دیگر در یک گراف موزون وبدون جهت ارائه دادیم. حال از روش حریصانه استفاده کرده، یک الگوریتم $\theta(n^2)$ برای مسئله طراحی می کنیم این الگوریتم دیکسترا نام دارد. الگوریتم را با این فرض ارائه می کنیم که از راس مورد نظر به هر یک از رئوس دیگر ، مسیری وجود داشته باشد.

این الگوریتم همانند الگوریتم پریم (برای تعیین وایجاد درخت پوشای کمینه) می باشد. برای مقدار دهی اولیه به مجموعه Y راسی را در آن قرار می دهیم که کوتاهترین مسیرهای آن باید تعیین شود، آن راس را v_1 در نظر می گیریم. به مجموعه F مقدار اولیه تهی را می دهیم. نخست یک راس v را انتخاب می کنیم که از همه به v_1 نزدیک تر است و آن را به Y و $< v_1, v >$ را به F اضافه می کنیم. واضح است که آن یال کوتاهترین مسیر میان v, v_1 است. سپس مسیرهایی از v_1 به رئوس موجود در V-Y را چک می کنیم که فقط رئوس موجود در Y را به عنوان رئوس واسط مجاز می شمارند. یکی از این مسیرها ، کوهترین مسیر است، راسی که در انتهای چنین مسیری باشد به Y ویالی که آن راس را در بردارد به F افزوده می شود. این روال ادامه می یابد تا Y با V برابر شود. در این نقطه F حاوی یالهای موجود در کوتاهترین مسیر است. یک الگوریتم سطح بالا برای این روال چنین است :

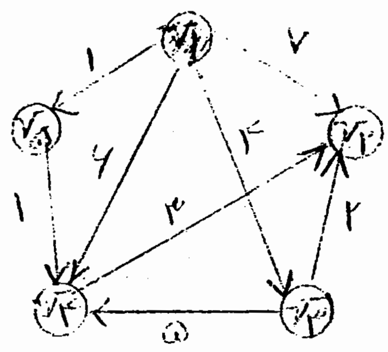
```
Y = { v1 } ;
F = Φ ;
While (the instance is not solved)
{
      select a vertex v form V-Y, that has a          // selection prcedure
      shortest path form v1 , using only vertices.    // and feasibility check
      in Y as intermediates;
      add the new vertex v to Y;
      add the edge (on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

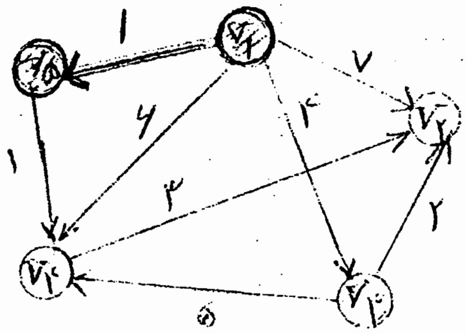
حال با ذکر مثالی ، این الگوریتم را بهتر درک خواهید کرد.

مثال : در گراف زیر کوتاهترین مسیر از راس v_1 به رئوس دیگر را بیابید.

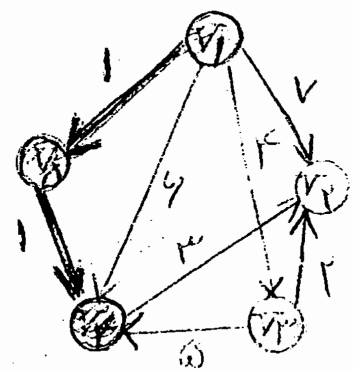
مراحل کار بصورت زیر است :

۱- راس v_5 به دلیل نزدیکتر بودن به v_1 انتخاب می شود.

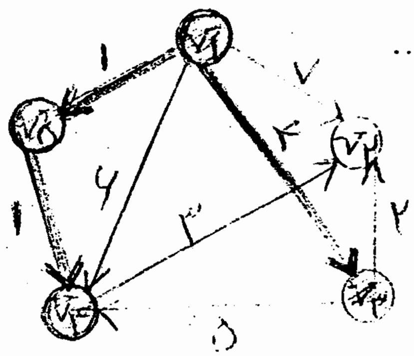




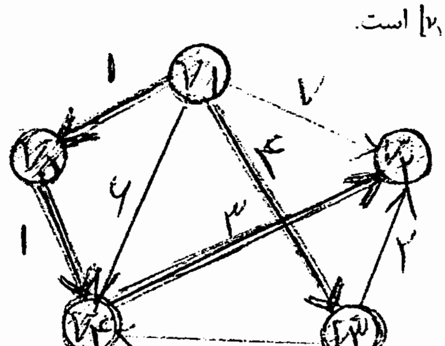
۲- راس v_4 به این دلیل انتخاب می شود که با استفاده از راسهای موجود در $\{v_5\}$ به عنوان واسط، کوتاهترین مسیر را از v_1 دارد.



۳- راس v_3 به این دلیل انتخاب می شود که با استفاده از راسهای موجود در $\{v_4, v_5\}$ به عنوان واسط، کوتاهترین مسیر را از v_1 دارد.



۴- کوتاهترین مسیر از v_1 به v_2 ، $[v_1, v_5, v_4, v_2]$ است.



رئوس واقع در y و یالهای موجود در F در هر مرحله سایه زده شده اند.

الگوریتم ۴-۳) الگوریتم دیکسترا :

مسئله : تعیین کوتاهترین مسیر از v_1 به همه رئوس دیگر در یک گراف موزون و جهت دار.

ورودی : عدد صحیح $n \geq 2$ و یک گراف جهت دار، موزون و متصل حاوی n راس . این گراف توسط آرایه دو بعدی W نشان داده می شود که سطرها و ستونهای آن از یک تا n اندیس گذاری شده اند و در آن w[i] [j] وزن یال راس i ام به راس j ام است.

خروجی . مجموعه یالهای F حاوی یالهای موجود در کوتاهترین مسیر.

```
Void dijkstra (int n,
                  Const number w [ ] [ ] ,
                  Set – of – edyes & F)
{
   index i, vnear;
   edge  e;
   index touch [2..n];
   number Length [2..n];
   F = Φ
   For (i=2 ; i < = n; i ++)            //for all vertices, initialize v1
   {
      touch [i] = 1 ;                   //to be the last vertex on the
      length [i] = w [1] [i] ;          // current shortest path from v1 ,
   }                                    //and initialize length of that
                                        //path to be the weight
                                        //on the edge from v1 ,
repeat (n-1     times)                  //Add all n-1 vertices to y.
{
  min = ∞ ;
  for (i=2 ; i <=n;i++)                 //check each vertex for
   if (0 ≤ length [i] < min)            // having shortest path.
   {
    min = length [i];
    vnear = i;
   }
e = edge from vertex indexed by touch [vnear] to vertex indexed by vnear;
```

```
add e to F;
for (i = 2 ; i<=n; i++)
   if (length[vnear]+w[vnear] [i] < length [i])
   {
      length [i] = length [vnear] + w [vnear] [i] ;
      touch [I] = vnear;                          //for each vertex not in y,
}
                                                 // up date its shortest path.
   Length [vnear] = -1;
    }
}
```

این الگوریتم فقط یالهای موجود در کوتاهترین مسیرها را تعیین می کند وطول این یالها را بدست نمی دهد. این طولها را می توان از یالها بدست آورد.

- پیچیدگی زمانی الگوریتم دیکسترا برابر است با $T(n) \in \Theta(n^2)$

۴-۴) زمانبندی :

آرایشگری را در نظر بگیرید که چند مشتری دارد و هر یک از مشتریان امر متفاوتی دارد (مثل اصلاح ساده ،اصلاح و شستشو، فردائم، رنگ کردن مو و...) ، این امور به یک اندازه زمان نمی برند، اما آرایشگر می داند که هر کدام چقدر زمان لازم دارد . هدف ، زمانبندی برای مشتریان است به نحوی که کمترین زمان انتظار را داشته باشند. زمان انتظار و نیز زمان ارائه سرویس را زمان در سیستم می نامند. مسئله کمینه سازی زمان کل در سیستم ،کاربردهای متعدد وزیادی دارد از قبیل دستیابی برنامه ها ($task$ ها) به cpu برای انجام پردازش مورد نیاز.

نوع دیگر زمانبندی، زمانبندی با تعیین مهلت معین است یعنی مشتریان (یا برنامه ها) برای انجام امور یک بازه زمانی مشخص را فرصت دارند. این روش برای رسیدن به حداکثر بهره ارائه می شود. به این روش ، زمانبندی با مهلت معین گویند.

۴-۴-۱) کمینه سازی زمان کل در سیستم :

یک روش برای این منظور اینست که همه زمانبندی های ممکن را در نظر گرفته ، آن را که کمینه است انتخاب کنیم. مثال زیر را در نظر بگیرید.

مثال) فرض کنید زمانهای لازم برای سرویس سه کار مختلف به قرار زیر است :

$t_1 = 5$ ، $t_2 = ۱۰$ ، $t_3 = ۴$

زمانهای صرف شده در سیستم برای این سه کار به قرار زیر است :

کار	زمان صرف شده در سیستم
۱	۵ (زمان ارائه سرویس)

۲	۵ (انتظار برای انجام کار ۱) + ۱۰ (زمان ارائه سرویس)
۳	۵ (انتظار برای انجام کار ۱) + ۱۰ (انتظار برای انجام کار ۲) +۴ (زمان ارائه سرویس)

زمان کل در سیستم برای این زمانبندی عبارتست از : ۳۹=(۴+۱۰+۵)+(۱۰+۵)+۵

تمام زمانبندی های موجود برای این مسئله به قرار زیر هستند :

ترتیب زمانبندی	زمان صرف شده کل در سیستم
[۱,۲,۳]	۵+(۵+۱۰)+(۵+۱۰+۴)=۳۹
[۱,۳,۲]	۵+(۵+۴)+(۵+۴+۱۰)=۳۳
[۲,۱,۳]	۱۰+(۱۰+۵)+(۱۰+۵+۴)=۴۴
[۲,۳,۱]	۱۰+(۱۰+۴)+(۱۰+۴+۵)=۴۳
[۳,۱,۲]	۴+(۴+۵)+(۴+۵+۱۰)=۳۲
[۳,۲,۱]	۴+(۴+۱۰)+(۴+۱۰+۵)=۳۷

ملاحظه می کنید که زمانبندی [۲و۱و۳] ، با زمان کل ۳۲ بهینه است. این زمانبندی بهینه زمانی حاصل می شود که کاری با کوچکترین زمان لازم جهت ارائه سرویس ، اول از همه انجام شود وسپس کارها به ترتیب صعودی زمان سرویس لازم انجام شوند. بنابراین کاری که آخر از همه اجرا می شود دارای بیشترین زمان لازم خواهد بود.

یک الگوریتم حریصانه سطح بالا برای این روش به صورت زیر می باشد :

```
sort the jobs by service time in nondecreasing order ;
while (the instance is not solved)
{
   schedule the next Job ;                 // selection procedure and feasibility check
   if (there are no more jobs)
        the instance is solved ;           // solution check
}
```

پیچیدگی زمانی الگوریتم فوق برابر است با $w(n) \in \theta(n \lg n)$:

قضیه : ۴-۳) تنها زمان بندیی که زمان کل را کمینه می کند، زمانبندیی است که در آن کارها به ترتیب و برحسب افزایش زمان ارائه سرویس ارائه می شوند.

۴-۴-۲) زمانبندی با مهلت معین :

در این روش ، هر کاری در یک واحد زمانی به پایان می رسد و دارای یک مهلت معین است، اگر کار پیش از مهلت معین یا در آن مدت انجام شود، بهره بدست می آید.

هدف ، زمانبندی کارها به نحوی است که بهره بیشینه بدست آید. برخی زمانبندی ها غیر ممکن هستند. مثال زیر را درنظر بگیرید.

مثال) فرض کنید کارها ، مهلت ها وبهره های زیر را داریم :

کار	مهلت	بهره
۱	۲	۳۰
۲	۱	۳۵
۳	۲	۲۵

وقتی می گوییم کار ۱ دارای مهلت ۲ است یعنی این کار را می توان در زمان ۱ یا ۲ آغاز کرد زمانبندی های ممکن عبارتند از :

زمانبندی	بهره کل
[۱,۳]	۳۰+۲۵=۵۵
[۲,۱]	۳۵+۳۰=۶۵
[۲,۳]	۳۵+۲۵=۶۰
[۳,۱]	۲۵+۳۰=۵۵

در جدول فوق زمانبندی های غیر ممکن ذکر نشده اند، بعنون مثال، زمانبندی [۲و۱] غیر ممکن است زیر کار ۱ ابتدا در زمان ۱ آغاز شده ۱ واحد لازم دارد تا به پایان برسد وبعد کار ۲ آغاز می شود ولی مهلت کار ۲ ، زمان ۱ است.

زمانبندی [۱و۲] با بهره ۶۵ بهینه است.

پیش از نوشتن یک الگوریتم سطح بالا به چند تعریف نیاز داریم :

ترتیب امکان پذیر : ترتیبی است که همه کارها در آن به ترتیب در مهلت مقرر خود آغاز شوند.

مجموعه امکان پذیر : مجموعه ای از این کارها را مجموعه امکان پذیر گویند.

ترتیب بهینه : ترتیبی که امکان پذیر بوده وبیشترین بهره کل را بدهد ترتیب بهینه است.

مجموعه بهینه کارها : مجموعه کارها در ترتیب بهینه را گویند.

حال الگوریتم حریصانه سطح بالای زیر را برای مسئله زمانبندی با مهلت ارائه می کنیم :

```
sort the jobs is nonincreasing order by profit ;
s = Φ
while (the instance is not solved)
{
  select next job ;                          // selection procedure
  if (s if feasible with this job added)     // feasibility check
        add this job to s ;
  if (there are no more jobs)                // solution check
     the instance is solved ;
}
```

مثال) فرض کنید کارها، مهلت ها و بهره های زیر موجود هستند :

کار	مهلت	بهره
۱	۳	۴۰
۲	۱	۳۵
۳	۱	۳۰
۴	۳	۲۵
۵	۱	۲۰
۶	۳	۱۵
۷	۲	۱۰

الگوریتم حریصانه فوق چنین عمل می کند :

۱- S برابر با Φ می باشد.

۲- S را برابر {۱} قرار می دهد چون ترتیب [۱] امکان پذیر است.

۳- S را برابر با {۱و۲} قرار میدهد چون ترتیب [۱و۲] امکان پذیر است.

۴- {۱و۲و۳} رد می شود چون هیچ ترتیب امکان پذیری برای این مجموعه وجود ندارد.

۵- S را برابر {۱و۲و۴} قرار می دهد چون ترتیب [۲و۱و۴] امکان پذیر است.

۶- {۱و۲و۴و۵} رد می شود چون هیچ ترتیب امکان پذیری برای این مجموعه موجود نیست.

۷- {۱و۲و۴و۶} رد می شود چون هیچ ترتیب امکان پذیری برای این مجموعه موجود نیست.

۸- {۱و۲و۴و۷} رد می شود چون هیچ ترتیب امکان پذیری برای این مجموعه موجود نیست.

مقدار نهایی S ، {۱و۲و۴} است ویک ترتیب امکان پذیر برای آن [۲و۱و۴] است چون کارهای ۱و۴ هر دو دارای مهلت ۳ هستند می توانیم از ترتیب [۲و۴و۱] نیز استفاده کنیم.

الگوریتم ۴-۴) زمانبندی با مهلت

مسئله : تعیین زمابندی با بهره کل بیشینه .

ورودی : n تعدادکارها ، آرایه ای از اعداد صحیح deadline که از ۱ تا n مرتب شده اند و deadline [i] مهلت مقرر برای کار i 'م را نشان می دهد. این آرایه به ترتیب غیر نزولی مرتب شده است.

خروجی : یک ترتیب بهینه J برای کارها .

```
Void schedule (int n,
          Const int deadline[ ],
          Sequence – of – integer & J)
{
     index    i ;
     sequence – of – integer  k;
     j = [1];
     For (i = 2 ; i <=n;i++)
  {
     k = j with  i   added according to nondecreasing values of deadline [i] ;
     if (k is feasible)
          j = k ;
     }
}
```

مثال : فرض کنید کارهای زیر داده شده اند الگوریتم فوق عملیات زیر را انجام می دهد :

کار	۱	۲	۳	۴	۵	۶	۷
مهلت	۳	۱	۱	۳	۱	۳	۲

مراحل کار چنین است :

۱- j مساوی با [۱] قرار داده می شود.

۲- k مساوی با [۱و۲] قرار داده شده ، مشخص می شود که امکان پذیر است.

J مساوی با [۱و۲] قرار داده شده ، زیرا k امکان پذیر است.

۳- k مساوی با [۱و۳و۲] قرار داده شده و رد می شود زیرا امکان پذیر است.

۴- k مساوی با [۴و۱و۲] قرار داده شده و مشخص می شود که امکان پذیر نیست.

J مساوی با [۴و۱و۲] قرار داده می شود زیرا k امکان پذیر است.

۵- k مساوی با [۴و۱و۵و۲] قرار داده شده و رد می شود زیرا امکان پذیر نیست.

۶- k مساوی با [۴و۶و۱و۲] قرار داده شده و رد می شود زیرا امکان پذیر نیست.

۷- k مساوی با [۴و۱و۷و۲] قرار داده شده و رد می شود زیرا امکان پذیر نیست.

بنابراین مقدار نهایی J برابر است با [۴و۱و۲]

پیچیدگی زمانی این الگوریتم در بدترین حالت برابر است با $w(n)\varepsilon\theta(n^2)$

قضیه ۴-۴) الگوریتم زمانبندی با مهلت معین همواره یک مجموعه بهینه تولید می کند.

۴-۵) روش حریصانه در مقابل روش برنامه نویسی پویا : مسئله کوله پشتی

روشهای حریصانه و برنامه نویسی پویا هردو برای حل مسائل بهینه سازی بکار می روند.

اصولاً یک مسئله را می توان با هر روش حل کرد مثلاً مسئله کوتاهترین مسیر ، هم با استفاده از برنامه نویسی پویا و هم با کمک روش حریصانه قابل حل است. اما برنامه نویسی پویا از آن جهت طاقت فرساست که کوتاهترین مسیر را از روی همه منابع تولید می کند بنابراین پیچیدگی زمانی آن $\theta(n^3)$ است. حال آنکه روش حریصانه دارای پیچیدگی زمانی $\theta(n^2)$ می باشد.

از طرف دیگر تعیین اینکه روش حریصانه حل بهینهٔ را تولید می کند یا خیر، دشوار است.

برای روشن تر شدن اختلاف وتفاوت میان این دو روش ، مسئله کوله پشتی صفر و ۱ را ارائه می دهیم. یک الگوریتم حریصانه طراحی می کنیم که قادر به حل مسئله کوله پشتی صفر و ۱ نمی باشد. سپس مسئله کوله پشتی صفر و ۱ را با موفقیت وبه کمک برنامه نویسی پویا حل می کنیم.

۴-۵-۱) روش حریصانه در حل مسئله کوله پشتی صفر و یک :

نمونه ای از این مسئله چنین است که دزدی با کوله پشتی وارد یک جواهر فروشی می شود. اگر وزن کل قطعات از یک حد بیشنه W بالاتر رود، کوله پشتی پاره خواهد شد. هر قطعه دارای ارزش و وزن معین می باشد. مسئله ای که دزد با آن مواجه

است، تعیین حداکثر ارزش قطعات است بطوریکه وزن کل آنها از حد معین W بالاتر نرود این مسئله را مسئله کوله پشتی صفر و یک می نامند. می توان مسئله را بصورت زیر فرموله کرد :

فرض کنید n قطعه داریم :

$s = \{item, item_2, ..., item_n\}$

Wi= وزن آیتم i ام

Pi = ارزش آیتم i ام

w= حداکثر وزنی که کوله پشتی قادر به تحمل آن است .

w, pi, wi همگی اعداد صحیح هستند. باید زیرمجموعه A از s طوری تعیین شود که $\sum_{item\, i \in A} Pi$ نسبت به $\sum_{item\, i \in A} wi \le w$ بیشینه شده باشد.

- راه حل غیر هوشمندانه اینست که تمام زیرمجموعه های این n قطعه را در نظر بگیریم، زیر مجموعه هایی را که وزن کل آنها از W بالاتر رود، کنار می گذاریم و از میان آنهایی که باقی مانده اند، آن را که بیشترین ارزش را دارد انتخاب می کنیم. مجموعه ای خاوی n قطعه دارای 2^n زیر مجموعه است. بنابراین زمان الگوریتم غیر هوشمندانه ، نمایی است.
- یک راه حل حریصانه این است که قطعاتی با بیشترین ارزش زودتر از همه برداشته شوند. یعنی آنها را به ترتیب کاهش ارزش، برداریم. ولی اگر با ارزش ترین قطعه در مقایسه با ارزشی که دارد، وزن بالایی داشته باشد این شیوه به خوبی کار نمی کند.

برای مثال فرض کنید سه قطعه داریم که اولی دارای وزن ۲۵ کیلوگرم و ارزش ۱۰۰ تومان، دومی و سومی هر یک به وزن ۱۰ کیلوگرم و ارزش ۹۰ تومان است. اگر W یعنی ظرفیت کوله پشتی ۳۰ کیلوگرم باشد، این روش حریصانه فقط ۱۰۰ تومان بهره دهی دارد، حال آنکه حل بهینه ۱۸۰ تومان می باشد.

- یک راهبرد حریصانه دیگر ، قرار دادن سبکترین قطعه در ابتداست . هنگامی که قطعات سبک در مقایسه با وزنی که دارند، کم ارزش باشند این روش نیز به شکست می انجامد.
- راهبرد حریصانه پیچیده تر این است که ابتدا قطعاتی با بزرگترین ارزش به ازای واحد وزن برداشته شوند یعنی قطعات را به تریبت ارزش آنها مرتب کرده و انتخاب می کنیم . فرض کنید سه قطعه با وزن و ارزشهای معین را داریم :

وزن

$$item_1 = \begin{cases} \text{وزن} : & 5kg \\ \text{ارزش} : & 50 \end{cases}$$

$$item_2 : \begin{cases} \text{وزن} : ۱۰\ Kg \\ \\ \text{ارزش} : ۶۰ \end{cases} \qquad item_3 : \begin{cases} \text{وزن} : ۲۰\ kg \\ \\ \text{ارزش} : ۱۴۰ \end{cases}$$

حال نسبت $\frac{\text{ارزش}}{\text{وزن}}$ را برای هر سه آیتم (قطعه) مشخص می کنیم :

$$item_1 : \frac{۵۰}{۵} = ۱۰ \quad , \quad item_2 : \frac{۶۰}{۱۰} = ۶ \quad , \quad item_3 = \frac{۱۴۰}{۲۰} = ۷$$

وبه ترتیب از بیشترین به کمترین ، داریم : $item_2$ و $item_3$ و $item_1$

الگوریتم حریصانه $item_1$ و $item_3$ را انتخاب می کند. بهره کل ۱۹۰ می شود. درحالیکه حل بهینه، انتخاب $item_2$ و $item_3$ می باشد که بهره ای برابر با ۲۰۰ را دارد.

۴-۵-۲) روش برنامه نوسی پویا برای مسئله کوله پشتی صفر ویک :

اگر بتوانیم نشان دهیم که اصل بهینگی برقرار است می توانیم مسئله کوله پشتی صفر ویک را با استفاده از برنامه نوسی پویا حل کنیم. فرض کنید A یک زیر مجموعه بهینه از n قطعه باشد. دو مورد وجود دارد :

یا A حاوی $item_n$ هست یا نیست.

اگر A حاوی $item_n$ نباشد، A با زیر مجموعه های بهینه از n-1 قطعه نخست برابر است ، اگر A حاوی $item_n$ باشد، بهره کل حاصل از قطعات موجود در A برابر P_n بعلاوه بهره بهینه است، زمانی که قطعات را بتوان از n-1 قطعه نخست انتخاب کرد. بنابراین ، اصل بهینگی برقرار است.

ننیجه بدست امده را می توان بصورتی که در ادامه خواهد آمد، تعمیم داد. اگر به ازای $p[i][w], w > ۰, i > ۰$ بهره بهینه حاصل از انتخاب i قطعه اول وتحت این محدودیت باشد که وزن کل نباید از W تجاوز کند L چنین است :

$$p[i][w] = \begin{cases} \max imun(p[i-۱][w], Pi + P[i-۱][w - wi] & \text{اگر } wi \le w\ ) \\ \\ P[i-۱][w] & \text{اگر } wi > w \end{cases}$$

بهره بیشینه برابر است با $p[n][w]$ ، می توانیم این مقدار را با استفاده از آرایه دو بعدی P که سطرهای آن از صفر تا n و ستونهای آن از صفر تا W اندیس گذاری شده اند، تعیین کنیم . مقادیر درون سطرها و ستونهای آرایه را به ترتیب با استفاده از رابطه قبلی برای $p[i][w]$ محاسبه می کنیم. مقادیر $p[0][w], p[i][0]$ را مساوی قرار می دهیم. واضح است تعداد عناصری که محاسبه می شود عبارت است از $nw \in \Theta(nw)$

نکات کلیدی فصل چهارم :

۱- الگوریتم حریصانه ، همانند برنامه نویسی پویا غالباً برای حل مسائل بهینه سازی بکار می رود ولی صراحت بیشتری دارد.

۲- الگوریتم حریصانه با انجام یکسری انتخاب ، که هر یک در لحظه ای خاص، بهترین به نظر می رسد عمل می کند.

۳- هر الگوریتم حریصانه شامل سه بخش روال انتخاب ، تحقیق عملی بودن وتحقیق حل است بطوریکه :

(selection proceeure) روال انتخاب، عنصر بعدی را که باید به مجموعه اضافه شود، انتخاب می کند.

(feasibility check) تحیق عملی بودن، تعیین می کند که آیا مجموعه جدید برای رسیدن به حل ، عملی است یا خیر.

(solution check) تحقیق حل، تعیین می کند که آیا مجموعه جدید، حل نمونه را بدست می دهد یا خیر.

۴- برای بدست آوردن درخت پوشای کمینه، روشهایی نظیر الگوریتم پریم، الگوریتم کروسکال وجود دارند. بطوریکه در گرافهایی که متراکم تر هستند الگوریتم کروسکال سریعتر عمل می کند و در گرافهایی که متصل تر هستند الگوریتم پریم بهتر عمل می کند.

۵- الگوریتم دیکسترا برای یافتن کوتاهترین مسیر بکار می رود و دارای پیچیدگی زمانی $\theta(n^2)$ می باشد.

۶- دو نوع الگوریتم زمانبندی ساده و با مهلت معین داریم که پیچیدگی زمانی اولی $\theta(n \log n)$ و دومی $\theta(n^2)$ است.