طراحی الگوریتم

درس چهارم: برنامه نویسی پویا

مدرس:

فرشید شیرافکن

دانشجوی دکتری دانشگاه تهران

(کارشناسی و کارشناسی ارشد : کامپیوتر نرم افزار) (دکتری: بیو انفورماتیک)

برنامه نویسی پویا

(Dynamic Programming)

در روش پویا ابتدا نمونه های کوچکتر را حل کرده و نتایج را ذخیره می کنیم و بعداً هرگاه به یکی از آن ها نیاز پیدا شد به جای محاسبه دوباره، کافی است آن را بازیابی کنیم.

در برنامه نویسی پویا از آرایه‌ای (جدولی) استفاده می شود.

برنامه نویسی پویا از این لحاظ که نمونه را به نمونه های کوچکتر تقسیم می کند مشابه روش تقسیم و حل است.

مراحل بسط یک الگوریتم برنامه نویسی پویا

۱- ارائه یک ویژگی بازگشتی برای حل نمونه ای مسئله

۲- حل نمونه ای از مسئله به شیوه پایین به بالا با حل نمونه های کوچکتر

هر مسئله بهینه سازی را نمی توان با استفاده از برنامه نویسی پویا حل کرد. اصل بهینگی باید در مسئله صدق کند.

گفته می شود اصل بهینگی در یک مسئله صدق می کند اگر یک حل بهینه برای نمونه ای مسئله، همواره حاوی حل بهینه برای همه زیرنمونه ها باشد.

تعدادی از الگوریتم هایی که به روش برنامه نویسی پویا حل می شوند:

۱- دنباله فیبوناچی

۲- ضریب دو جمله ای

۳- ضرب زنجیره ای ماتریس ها

۴- درخت های جستجوی دودویی بهینه

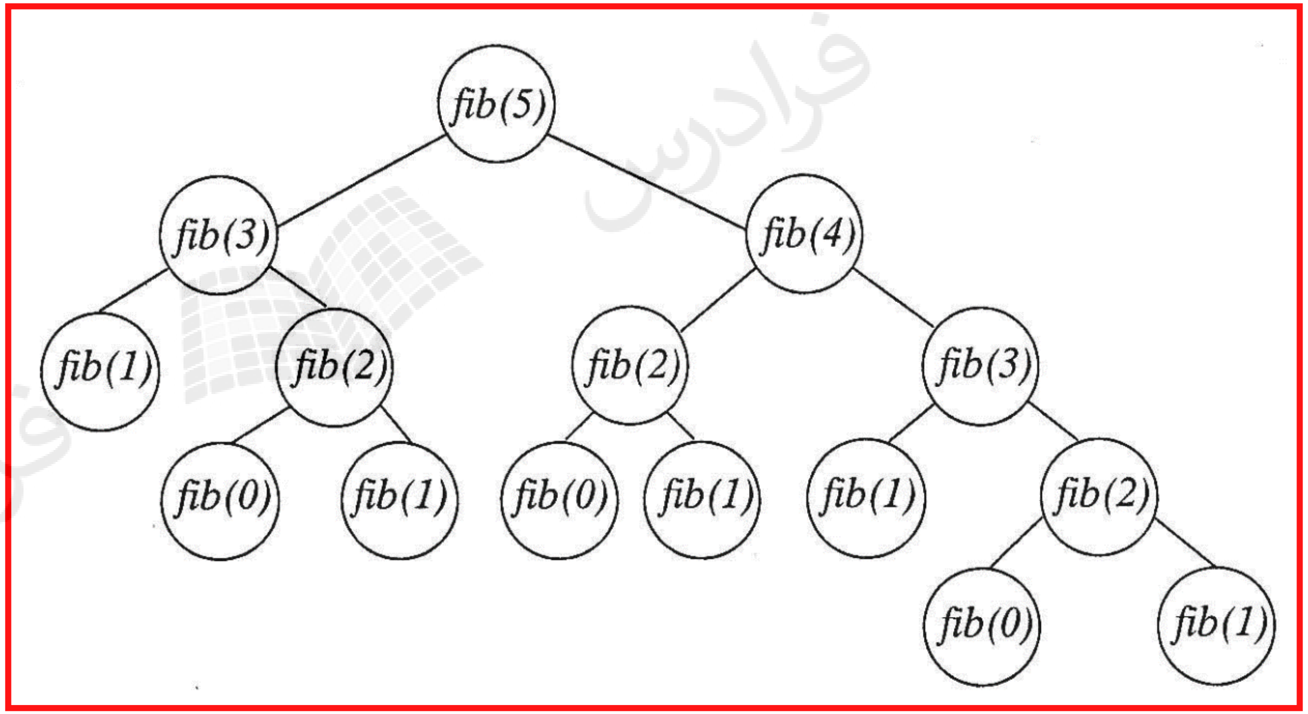
۵- کوله پشتی صفر و یک

۶- فلوید

۷- فروشنده دوره گرد

الگوریتم بازگشتی برای محاسبه جمله n ام دنباله فیبوناچی

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



تعداد جملات محاسبه شده توسط الگوریتم بازگشتی بالا، برای تعیین جمله nاُم فیبوناچی، بزرگتر از $2^{n/2}$ است.

با توجه به درخت، الگوریتم برای تعیین fib(n) تعداد جملات زیر را محاسبه می‌کند:

n	0	1	2	3	4	5	6	…
تعداد جملات محاسبه شده	1	1	3	5	9	15	25	…

تعداد جملات محاسبه شده برای محاسبه فیبوناچی n :

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

الگوریتم تکرار برای محاسبه جمله nاُم دنباله فیبوناچی

هنگام محاسبه یک مقدار، آن را در آرایه‌ای ذخیره می کنیم تا هرگاه در آینده به آن نیاز بود، لازم نباشد دوباره محاسبه شود.

```
fib(n){
    int  f[0..n];
    f[0] = 0;
    if (n > 0){
        f[1] = 1;
        for (i = 2; i <= n; i++)
            f[i] = f[i–1] + f[i–2];
    }
    return   f[n];
}
```

الگوریتم تکرار از مرتبه $O(n)$ و الگوریتم بازگشتی از مرتبه $O(2^n)$ است.

ضریب دو جمله ای

محاسبه ضریب جمله k+1 ام در بسط $(a+b)^n$:

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

ضریب جمله سوم بسط $(a+b)^3$:

$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

$$\binom{3}{2} = \frac{3!}{2! \times (3-2)!} = 3$$

ضریب جمله ششم از بسط $(x+y)^8$

$$\binom{8}{5} = \frac{8!}{5! \times (8-5)!} = \frac{8 \times 7 \times 6 \times 5!}{5! \times 3!} = \frac{8 \times 7 \times 6}{3!} = 56$$

می دانیم که :

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

با استفاده از این ویژگی بازگشتی دیگر نیازی به محاسبه !n یا !k نیست و به الگوریتم تقسیم و حل می رسیم.

محاسبه ضریب دو جمله ای (تقسیم و حل)

n و k اعداد صحیح و مثبت می باشند.($k \leq n$)

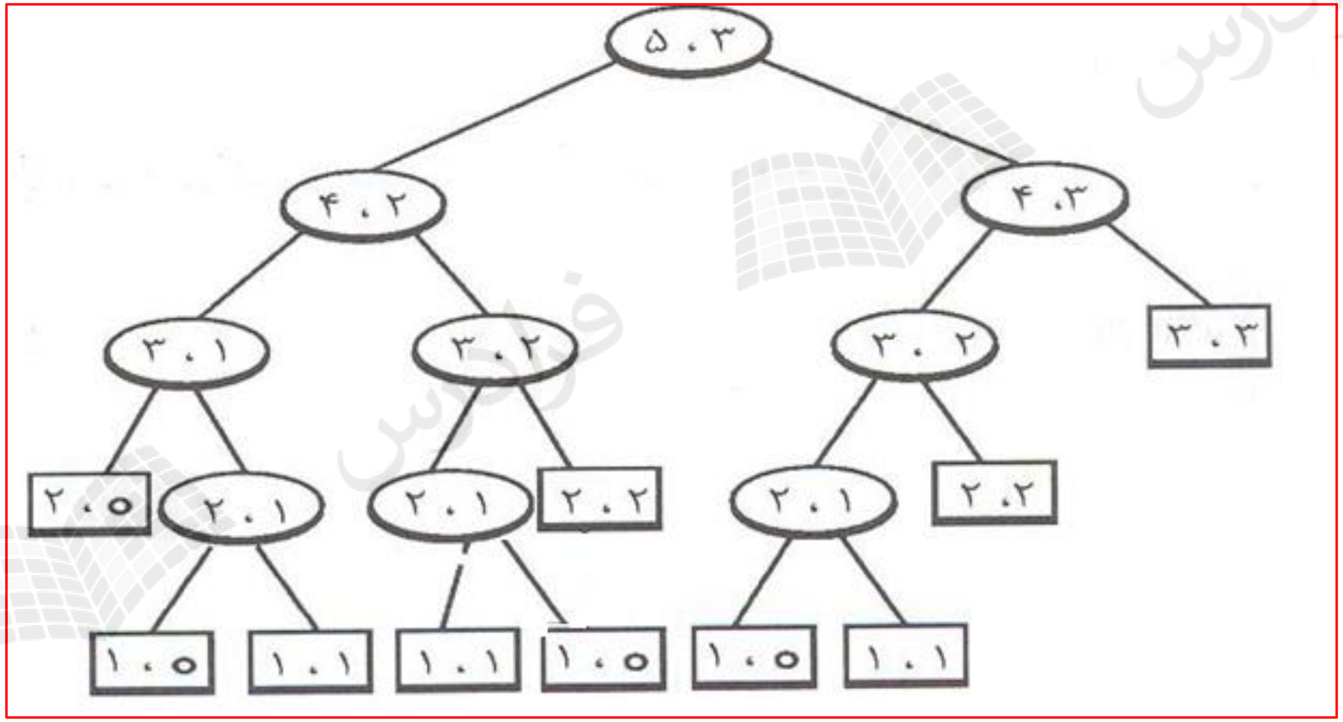
$$\binom{n}{k}$$

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

مثال

درخت بازگشتی $\binom{5}{3}$

$$bin(n,k) = bin(n-1,k-1) + bin(n-1,k)$$



این درخت دارای ۱۹ گره است.

روش تقسیم و حل برای این مسئله کارایی ندارد. چون در هربار فراخوانی بازگشتی، نمونه ها چندین بار حل می شوند و نمونه به دو نمونه کوچکتر تقسیم می شود که تقریباً به بزرگی نمونه اولیه هستند.در نتیجه الگوریتمی با استفاده از برنامه نویسی پویا طراحی می کنیم.

ویژگی بازگشتی :

یا

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

مثال

محاسبه مقدار $B[4][2] = \binom{4}{2}$

$B[0..4][0..2]$

	0	1	2
0	1		
1	1	1	
2	1		1
3	1		
4	1		

خانه های بعدی ماتریس را به کمک رابطه زیر ، سطر به سطر پر می کنیم:

$$B[i][j] = B[i-1][j-1] + B[i-1][j] \qquad i > j > 0$$

$$B[2][1] = B[1][0] + B[1][1] = 1 + 1 = 2$$

$$B[3][1] = B[2][0] + B[2][1] = 1 + 2 = 3$$

$$B[3][2] = B[2][1] + B[2][2] = 2 + 1 = 3$$

$$B[4][1] = B[3][0] + B[3][1] = 1 + 3 = 4$$

$$B[4][2] = B[3][1] + B[3][2] = 3 + 3 = 6$$

	0	1	2
0	1		
1	1	1	
2	1	2	1
3	1	3	3
4	1	4	6

الگوریتم محاسبه ضریب دو جمله ای (برنامه نویسی پویا)

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

تعداد گذرهای انجام شده از حلقه j :

i	0	1	2	3	...	k	k+1	...	n
تعداد گذرها	1	2	3	4	...	k+1	k+1	...	k+1

تعداد کل گذرها :

$$1+2+3+4+\ldots+k+\overbrace{(k+1)+(k+1)+\ldots+(k+1)}^{n-k+1 \text{ بار}}$$

بنابراین :

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

مثال

تعداد گذرهای انجام شده از حلقه j، برای محاسبه bin(6,2) به صورت زیر است:

i	0	1	2	3	4	5	6
تعداد گذر	1	2	3	3	3	3	3

$$\frac{(2\times 6-2+2)\times(2+1)}{2}=\frac{12\times 3}{2}=18$$

ضرب زنجیره ای ماتریس‌ها

(Matrix-chain Multiplication)

یک زنجیره از n ماتریس $< A_1, A_2, \ldots, A_n >$ وجود دارد که هدف محاسبه حاصل ضرب $A_1 \times A_2 \times \ldots \times A_n$ است به طوری که تعداد ضرب های عددی مینیمم شود.

مثال

تعیین بهترین حالت ضرب ماتریس های زیر :

A	$\times$	B	$\times$	C	$\times$	D
20×2		2×30		30×12		12×8

تعداد ضرب های مورد نیاز برای ضرب ماتریس با ابعاد $i\times j$ در ماتریس با ابعاد $j\times k$:

$$i\times j\times k$$

پنج حالت ممکن:

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

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

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

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

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

مثال

کمترین تعداد عمل ضرب برای ضرب چهار ماتریس زیر کدام است؟

$$A_{10\times100}\times B_{100\times1}\times C_{1\times20}\times D_{20\times5}$$

ترتیب ضرب $A\times B$: $10\times100\times1=1000$

تعداد ضرب مورد نیاز برای محاسبه $C\times D$: $1\times20\times5=100$

تعداد ضرب مورد نیاز برای محاسبه $((A\times B)\times(C\times D))$: $10\times1\times5=50$

مجموع ضرب ها:

$$1000+100+50=1150$$

تحلیل

$$< A_1, A_2, ..., A_n >$$

تعداد پرانتز گذاریها برای حاصل ضرب پرانتز اول را با $T(k)$ و برای پرانتز دوم را با $T(n-k)$ نشان می دهیم،

$$(A_1 \times A_2 \times ... \times A_k)(A_{k+1} \times A_{k+2} \times ... \times A_n)$$

رابطه بازگشتی تعداد حالات ممکن برای پرانتز گذاری در ضرب n ماتریس:

$$T(n) = \sum_{k=1}^{n-1} T(k)T(n-k) \qquad n \geq 2$$

$$T(1) = 1$$

$$\Rightarrow \frac{1}{n}\binom{2n-2}{n-1}$$

از مرتبه $\Omega(2^n)$ است.

مثال

تعداد حالت های پرانتز گذاری در ضرب ۳ ماتریس $< A_1, A_2, A_3 >$:

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

$$T(3) = T(1)T(2) + T(2)T(1) = 1\times1 + 1\times1 = 2$$

این دو حالت :

$$(A_1 \times A_2) \times A_3$$

$$A_1 \times (A_2 \times A_3)$$

ویژگی بازگشتی برای ضرب n ماتریس

$$M[i][j] = \underset{i \le k \le j-1}{\operatorname{minimum}}(M[i][k] + M[k+1][j] + d_{i-1}d_k d_j)$$

$$M[i][j] = 0 \qquad i = j$$

مثال

تعیین عناصر ماتریس M :

A	$\times$	B	$\times$	C	$\times$	D
20×2		2×30		30×12		12×8

	1	2	3	4
1	0			
2		0		
3			0	
4				0

محاسبه عناصر قطر اول:

$$M[1][2]=M[1][1]+M[2][2]+d_0d_1d_2\ =0+0+20\times 2\times 30=1200$$

$$M[2][3]=M[2][2]+M[3][3]+d_1d_2d_3\ =0+0+2\times 30\times 12=720$$

$$M[3][4]=M[3][3]+M[4][4]+d_2d_3d_4\ =0+0+30\times 12\times 8=2880$$

	1	2	3	4
1	0	1200		
2		0	720	
3			0	2880
4				0

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

محاسبه عناصر قطر دوم:

$$M[1][3]=\begin{cases} K=1 \Rightarrow M[1][1]+M[2][3]+d_0 d_1 d_3 = 0+720+20\times 2\times 12 = 1200 \\ K=2 \Rightarrow M[1][2]+M[3][3]+d_0 d_2 d_3 = 1200+0+20\times 30\times 12 = 8400 \end{cases}$$

$$M[2][4]=\begin{cases} k=2 \Rightarrow M[2][2]+M[3][4]+d_1 d_2 d_4 = 0+2880+2\times 30\times 8 = 3360 \\ k=3 \Rightarrow M[2][3]+M[4][4]+d_1 d_3 d_4 = 720+0+2\times 12\times 8 = 912 \end{cases}$$

	1	2	3	4
1	0	1200	1200	
2		0	720	912
3			0	2880
4				0

$$M[i][j] = \underset{i\le k\le j-1}{\min imum}(M[i][k]+M[k+1][j]+d_{i-1}d_k d_j)$$

محاسبه عناصر قطر سوم:

$$M[i][j] = \min_{i \le k \le j-1} imum(M[i][k] + M[k+1][j] + d_{i-1} d_k d_j)$$

$$M[1][4] = \begin{cases} k=1 \Rightarrow M[1][1] + M[2][4] + d_0 d_1 d_4 = 0 + 912 + 20 \times 2 \times 8 = 1232 \\ k=2 \Rightarrow M[1][2] + M[3][4] + d_0 d_2 d_4 = 1200 + 2880 + 20 \times 30 \times 8 = 8880 \\ k=3 \Rightarrow M[1][3] + M[4][4] + d_0 d_3 d_4 = 1200 + 0 + 20 \times 12 \times 8 = 3120 \end{cases}$$

	1	2	3	4
1	0	1200	1200	1232
2		0	720	912
3			0	2880
4				0

	1	2	3	4
1		1	1	1
2			2	3
3				3
4				

ماتریس p

مثال

تعیین نحوه ضرب ماتریس ها

	1	2	3	4
1		1	1	1
2			2	3
3				3
4				

(A)(BCD)

(A((BC)D))

مثال

تعیین نحوه ضرب ماتریس ها

	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						

$(A_1A_2A_3)(A_4A_5A_6)$

$(A_1(A_2A_3))$

$((A_4A_5)A_6)$

$((A_1(A_2A_3))((A_4A_5)A_6))$

الگوریتم حداقل ضرب‌ها

```
minmult (n , d[ ] , P[ ][ ] ){
    int  M [1..n][1..n];
    for ( t =1 ; t <= n–1 ; t++)
        for ( i=1 ; i<= n–t ; i++)
        {
            j= i+t;
            M[i][j] =  minimum   (M[i][k] + M[k+1][j] + d [i-1] * d[k] * d[j]);
                      i≤k≤j–1
            P[i][j] = a value of k that gave the minimum;
        }
    return   M[1][n];
}
```

تحلیل الگوریتم

$$\sum_{t=1}^{n-1}\sum_{i=1}^{n-t}\sum_{k=i}^{j-1} 1$$

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

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

$$= \sum_{t=1}^{n-1}\sum_{i=1}^{n-t} t = \sum_{t=1}^{n-1} t(n-t) = \frac{n(n-1)(n+1)}{6}$$

مثال

در ضرب چهار ماتریس تعداد دفعاتی که دستور **Minimum** اجرا می شود، یعنی تعداد دفعاتی که **k** مقدار می گیرد، چقدر است؟

t	i	j	k
1	1	2	1
	2	3	2
	3	4	3
2	1	3	1,2
	2	4	2,3
3	1	4	1,2,3

$$\frac{4(4-1)(4+1)}{6} = 10$$

بررسی برقرار بودن اصل بهینگی

اصل بهینگی در مسئله ضرب زنجیره ای ماتریس ها صدق می کند. یعنی ترتیب بهینه برای ضرب $\mathbf{n}$ ماتریس، در برگیرندۀ ترتیب بهینه برای ضرب هر زیر مجموعه ای از این $\mathbf{n}$ ماتریس است.

برای مثال، اگر ترتیب بهینه برای ضرب شش ماتریس برابر $A_1(((A_2A_3)A_4)A_5)A_6)$ باشد، در این صورت $(A_2A_3)A_4$ حتما ترتیب بهینه برای ضرب ماتریس های A_2 تا A_4 است.

چون اصل بهینگی برقرار است، می توان از برنامه نویسی پویا در این مسئله استفاده کرد.

الگوریتم بازگشتی ضرب زنجیره ای ماتریس‌ها

```
RMC(p , i , j )
{
   if (i == j)   return  0;
   m[i,j] = ∞ ;
   for( k = i ; k<= j-1 ; k++)
    {
      q = RMC( p , i , k ) + RMC( p , k+1 , j )+  d_{i-1}d_k d_j
      if (q < M[i,j] )
           M[i,j]=q;
     }
   return  M[i,j];
}
```

کران پایین:

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n$$

$$T(n) = \Omega(2^n)$$

زمان اجرای هر یک از if ها ، حداقل یک واحد زمانی باشد.

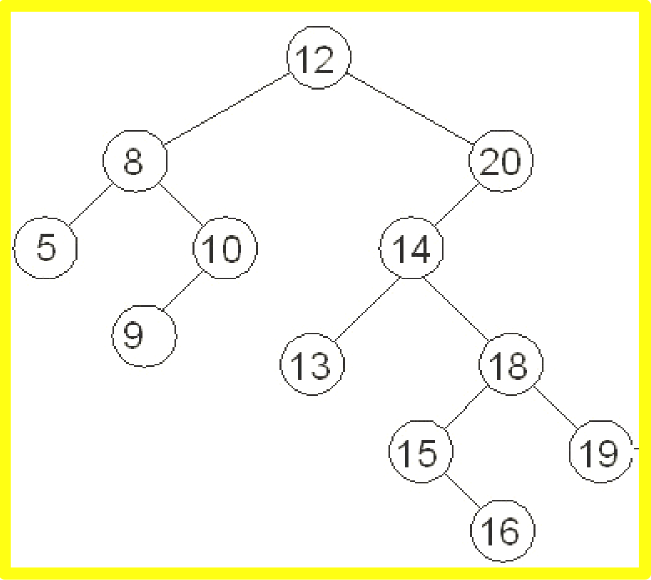
کران بالا:

$$T(n) \leq 2 \sum_{i=1}^{n-1} T(i) + cn$$

$$T(n) = O(n3^{n-1})$$

زمان اجرای هر یک از ifها ، حداکثر c واحد زمانی باشد.

درخت جستجوی دودویی بهینه



درخت جستجوی دودویی(BST)، یک درخت دودویی از عناصر (کلید) است که :

۱- هر گره حاوی یک کلید است.

۲- کلیدهای موجود در زیر درخت چپ یک گره مفروض، کوچکتر از کلید آن گره هستند.

۳- کلیدهای موجود در زیر درخت راست یک گره مفروض، بزرگتر از کلید آن گره هستند.

ما می خواهیم کلیدها را در یک BST طوری سازماندهی کنیم که زمان میانگین برای تعیین مکان کلیدها به حداقل برسد.

به این درخت، درخت بهینه می گویند.

مثال

n عدد متمایز داده شده است. تعداد درخت های جستجوی دودویی که با این اعداد می توان ساخت از رابطه زیر محاسبه می شود:

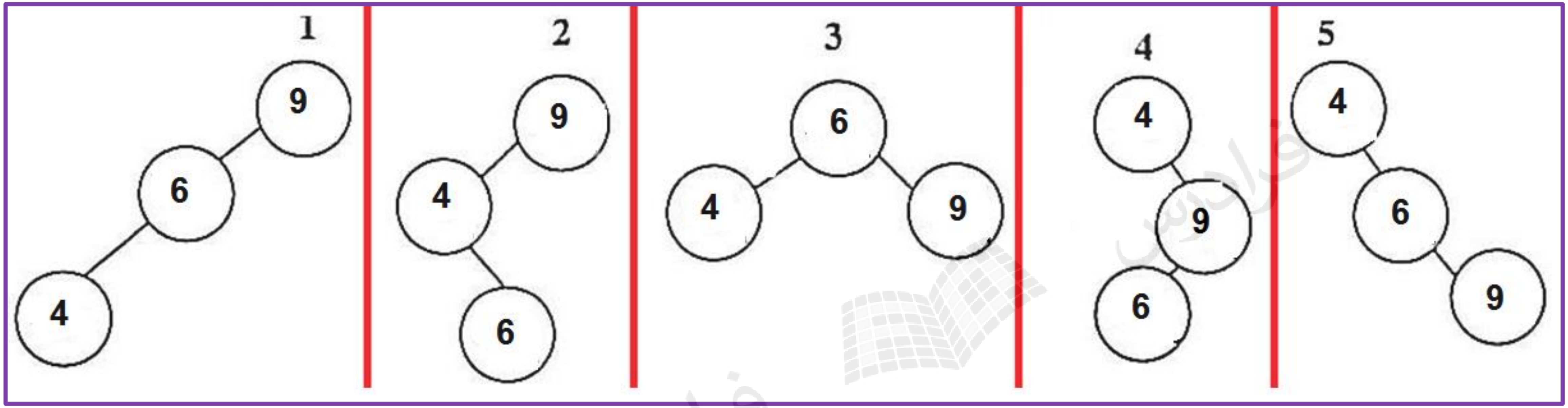
$$T(n) = \sum_{k=1}^{n-1} T(k)T(n-k)$$

مثال

با فرض اینکه سه کلید مرتب شده داشته باشیم، $(key_1 < key_2 < key_3)$ به طوری که احتمال مساوی بودن هر کلید با کلید مورد جستجو به ترتیب برابر 0.7 ، 0.2 و 0.1 باشد، زمان های جستجوی میانگین را مشخص کنید.

key1=4
key2=6
key3= 9

درخت های قابل ایجاد با ۳ کلید



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

$P_1 = 0.7$

$P_2 = 0.2$

$P_3 = 0.1$

1. $3(0.7) + 2(0.2) + 1(0.1) = 2.6$
2. $2(0.7) + 3(0.2) + 1(0.1) = 2.1$
3. $2(0.7) + 1(0.2) + 2(0.1) = 1.8$
4. $1(0.7) + 3(0.2) + 2(0.1) = 1.5$
5. $1(0.7) + 2(0.2) + 3(0.1) = 1.4$

- به طور کلی، یک **BST** بهینه را نمی توان با در نظر گرفتن همهٔ **BST**ها یافت، زیرا تعداد چنین درخت‌هایی، حداقل رابطه نمایی با **n** دارد.

- در این مسئله، اصل بهینگی برقرار است. چون هر زیر درخت بهینه‌ای از یک درخت بهینه، برای کلیدهای موجود در آن زیر درخت، بهینه است.

رابطه بازگشتی تعیین BST بهینه

$$A[i][j] = \underset{i \le k \le j}{\text{minimum}}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \qquad i < j$$

$$A[i][i] = p_i \qquad A[i][i-1] = 0 \qquad A[j+1][i] = 0$$

مثال

key1=4
key2=6
key3= 9

$P_1 = 0.7$
$P_2 = 0.2$
$P_3 = 0.1$

	0	1	2	3
1	0	0.7		
2		0	0.2	
3			0	0.1
4				0

ماتریسی با ابعاد A[1..n+1][0..n] در نظر گرفته و قطر اصلی آن را صفر قرار داده و احتمال ها را در قطر یک قرار می دهیم.

محاسبه A[1][2]:

$$A[i][j] = \underset{i \le k \le j}{\text{minimum}}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \qquad i < j$$

	0	1	2	3
1	0	0.7	1.1	
2		0	0.2	
3			0	0.1
4				0

$$k = 1 \Rightarrow A[1][2] = A[1][0] + A[2][2] + P_1 + P_2 = 0 + 0.2 + 0.7 + 0.2 = 1.1$$

$$k = 2 \Rightarrow A[1][2] = A[1][1] + A[3][2] + P_1 + P_2 = 0.7 + 0 + 0.7 + 0.2 = 1.6$$

محاسبه A[2][3]

$$A[i][j] = \underset{i \le k \le j}{\text{minimum}}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \qquad i < j$$

	0	1	2	3
1	0	0.7	1.1	
2		0	0.2	0.4
3			0	0.1
4				0

$$k = 2 \Rightarrow A[2][3] = A[2][1] + A[3][3] + P_2 + P_3 = 0 + 0.1 + 0.2 + 0.1 = 0.4$$

$$k = 3 \Rightarrow A[2][3] = A[2][2] + A[4][3] + P_2 + P_3 = 0.2 + 0 + 0.2 + 0.1 = 0.5$$

محاسبه A[1][3]:

$$k=1 \Rightarrow A[1][3]=A[1][0]+A[2][3]+P_1+P_2+P_3=0+0.4+0.7+0.2+0.1=1.4$$

$$k=2 \Rightarrow A[1][3]=A[1][1]+A[3][3]+P_1+P_2+P_3=0.7+0.1+0.7+0.2+0.1=1.8$$

$$k=3 \Rightarrow A[1][3]=A[1][2]+A[4][3]+P_1+P_2+P_3=1.1+0+0.7+0.2+0.1=2.1$$

	0	1	2	3
1	0	0.7	1.1	1.4
2		0	0.2	0.4
3			0	0.1
4				0

	0	1	2	3
1	0	1	1	1
2		0	2	2
3			0	3
4				0

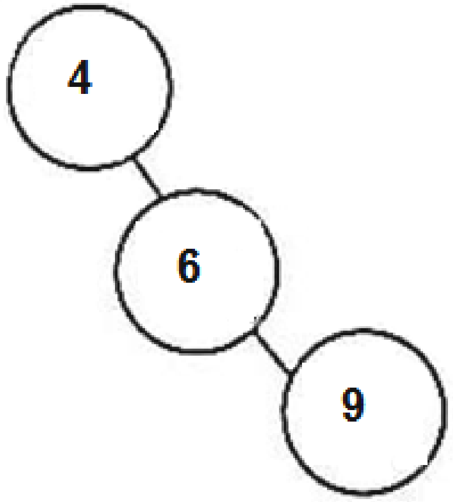
ماتریس R

رسم درخت به کمک ماتریس R :

چون R[1][3] برابر 1 می باشد، پس key1 ریشه اصلی است.

حال چون R[2][3] برابر 2 است، پس بین این دو کلید ، key2 ریشه است.

	0	1	2	3
1	0	1	1	1
2		0	2	2
3			0	3
4				0



key1=4 , key2=6 , key3= 9

رسم BST بهینه:

مثال

$key1 = 2$

$key2 = 5$

$key3 = 6$

$key4 = 9$

$p_1 = \frac{3}{8}$

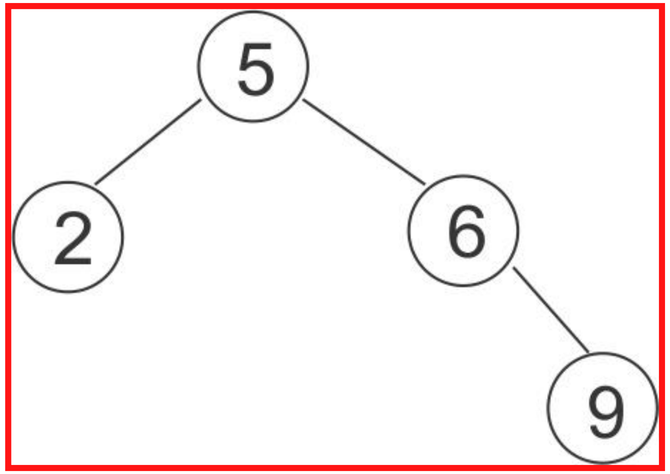
$p_2 = \frac{3}{8}$

$p_3 = \frac{1}{8}$

$p_4 = \frac{1}{8}$

ماتریس R

	0	1	2	3	4
1	0	1	1	2	2
2		0	2	2	2
3			0	3	3
4				0	4
5					0



الگوریتم درخت جستجوی دودویی بهینه

```
optsearchtree (n , p[ ] , minavg ,R[ ][ ] ){
   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][n]= 0;   R[n+1][n] = 0;
for (t = 1; t <= n–1 ; t ++)
      for (i =1; i <= n–t ; i++) {
            j = i + t;
```

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

```
            R[i][j] = a value of  k  that  gave the minimum;
         }
   minavg= A[1][n];
}
```

تحلیل الگوریتم

$$\sum_{t=1}^{n-1}\sum_{i=1}^{n-t}\sum_{k=i}^{j}1$$

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

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

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

مثال

تعداد دفعات انجام عمل اصلی (مقدار دهی به k)

t	i	j	k
1	1	2	1,2
	2	3	2,3
2	1	3	1,2,3

$$\frac{3\times 2\times 7}{6}=7$$

مسئله کوله پشتی صفر و یک

(0-1 knapsack)

دزدی وارد یک جواهر فروشی شده و می خواهد قطعه هایی که دارای ارزش و وزن معینی هستند را طوری در کوله پشتی خود قرار دهد که بیشترین سود حاصل شود.

البته وزن قطعه ها از یک حد مشخص نباید بیشتر شود، چون کوله پشتی پاره خواهد شد.

اگر قطعه ها به گونه ای باشند که یا انتخاب می شوند و یا نه، به آن مسئله کوله پشتی صفر و یک می گویند. و اگر دزد بتواند هر کسری از قطعه ها را بردارد، به آن مسئله کوله پشتی کسری می گویند.

کوله پشتی صفر و یک : شمش های طلا و نقره

کوله پشتی کسری: کیسه های حاوی خاک طلا و نقره .

روش حل کوله پشتی:

صفر و یک : پویا

کسری : حریصانه

روش پویا برای حل مسئله کوله پشتی 0/1

$$P[n][w] = \begin{cases} \text{maximum}(P[n-1][w], P_n + P[n-1][w-w_n]) & w_n \leq W \\ P[n-1][W] & w_n > W \end{cases}$$

وزن $item_i$ با w_i و ارزش آن با p_i نشان داده می شود.

تنها عناصر مورد نیاز در سطر $(n-1)$ اُم، آنهایی هستند که برای محاسبه $P[n][w]$ به کار می روند. که عبارتند از: $P[n-1][w-w_n]$ و $P[n-1][w]$

مثال : با فرض $w = 30$ ، سود بهینه را بدست آورید.

وزن(پوند)	ارزش(دلار)	قطعه
5	50	1
10	60	2
20	140	3

باید $P[3][30]$ را بدست آوریم.

$$P[n][w] = \text{maximum}(P[n-1][w], P_n + P[n-1][w-w_n])$$

$$P[3][30] = \text{maximum}\begin{cases} P[2][30] \\ P_3 + P[2][30-w_3] = 140 + P[2][10] \end{cases}$$

$$P[2][10] = \text{maximum}\begin{cases} P[1][10] = 50 \\ P_2 + P[1][10 - w_2] = 60 + P[1][0] = 60 \end{cases} \Rightarrow P[2][10] = 60$$

$$P[2][30] = \text{maximum}\begin{cases} P[1][30] = 50 \\ P_2 + P[1][30 - w_2] = 60 + P[1][20] = 110 \end{cases} \Rightarrow P[2][30] = 110$$

بنابراین:

$$p[3][30] = \text{maximum}\begin{cases} P[2][30] = 110 \\ 140 + P[2][10] = 140 + 60 = 200 \end{cases} \Rightarrow p[3][30] = 200$$

پس سود بهینه ۲۰۰ دلار است. (انتخاب قطعه های ۲ و ۳)

تعداد عناصری که توسط الگوریتم برنامه نویسی پویا برای مسئله کوله پشتی صفر و یک محاسبه می شود در بدترین حالت برابر است با:

$$O(\text{minimum}(2^n, nW))$$

تاکنون کسی برای مسئله کوله پشتی صفر و یک الگوریتمی نیافته است که بدترین حالت آن بهتر از زمان نمایی باشد و در عین حال هنوز کسی عدم امکان آن را نیز اثبات نکرده است.

مثال روش پویا برای حل مسئله کوله پشتی 0/1

$n = 4$

$S = 5$

Elements (size, value) =

{ (2, 3), (3, 4), (4, 5), (5, 6) }

i\s	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

i\s	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

مسئله نصب پوستر

در یک راهرو **n** تابلو پشت سرهم برای نصب پوستر آماده شده است (تابلوهای b_1 تا b_n).

طبق مقررات، یک پوستر نباید در دو تابلوی پشت سرهم نصب شود.

و در یک تابلو نباید بیش از یک عدد از یک پوستر نصب شود.

برای هر تابلو یک " ضریب دید " w_i تعیین شده که نشان دهنده ی میزان دید آن تابلو است

(هر چه عدد بزرگ تر ، به این معنی است که پوستر این تابلو بیش تر از بقیه دیده می شود)

با داشتن **W** ها برای همه ی تابلوها، می خواهیم یک پوستر را در تعدادی از این تابلوها نصب کنیم که مجموع ضریب دید آن بیشینه شود.

W1	W2	W3
W1	W2	W3

$f(i)$: مجموع ضریب دید(وزن) تابلوهای با شماره ۱ تا i است که بر روی آنها پوستر نصب می شود.

پوستری را در نظر بگیرید. دو حالت می توان رخ دهد:

الف- این پوستر روی تابلوی i نصب شود:

که دیگر نمی تواند بر روی تابلوی i-1 نصب شود. مجموع ضریب دیدها : $w_i + f(i-2)$

ب- این پوستر بر روی تابلوی i نصب نشود.

مجموع ضریب دیدها : $f(i-1)$

$$f(i) = \max\{f(i-1), w_i + f(i-2)\}$$

طولانی ترین زیر رشته مشترک

Longest Common Subsequence(LCS)

X = A B C B D A B
Y = B D C A B A
Z = B C B A

ACTGAACTCTGTGCACT

TGACTCAGCACAAAAAC

ACTGAACTCTGTGCACT

TGACTCAGCACAAAAAC

طولانی ترین زیر رشته مشترک

$X = \langle x_1, x_2, \cdots, x_m \rangle$
$Y = \langle y_1, y_2, \cdots, y_n \rangle$
$Z = \langle z_1, z_2, \cdots, z_k \rangle$

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1} = LCS(X_{m-1}, Y_{n-1})$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z = LCS(X_{m-1}, Y)$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z = LCS(X, Y_{n-1})$.

X = A B C B A
Y = C D A
Z = C A

X = A B C B A B
Y = B C A B A
Z = B C B A

X = A B C
Y = B C A
Z = B C

طولانی ترین زیر رشته مشترک

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1} = LCS(X_{m-1}, Y_{n-1})$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z = LCS(X_{m-1}, Y)$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z = LCS(X, Y_{n-1})$.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ C[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

$X = A\ B\ C\ B\ D\ A\ B$

$Y = B\ D\ C\ A\ B\ A$

i	x_i \ y_j	j=0	1: B	2: D	3: C	4: A	5: B	6: A
0		0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

$Z = B\ C\ B\ A$

طولانی ترین زیر رشته مشترک

$$B[i,j] = \begin{cases} \leftarrow \text{ or } \uparrow & \text{if } i = 0 \text{ or } j = 0, \\ \nwarrow & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \leftarrow \text{ or } \uparrow & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

الگوریتم طولانی ترین زیر رشته مشترک

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
      do c[i, 0] ← 0
for j ← 0 to n
      do c[0, j] ← 0
for i ← 1 to m
      do for j ← 1 to n
             do if x_i = y_j
                   then c[i, j] ← c[i − 1, j − 1] + 1
                        b[i, j] ← "↖"
                   else if c[i − 1, j] ≥ c[i, j − 1]
                           then c[i, j] ← c[i − 1, j]
                                b[i, j] ← "↑"
                           else c[i, j] ← c[i, j − 1]
                                b[i, j] ← "←"
return c and b
```

```
PRINT-LCS(b, X, i, j)
if i = 0 or j = 0
    then return
if b[i, j] = "↖"
    then PRINT-LCS(b, X, i − 1, j − 1)
         print x_i
elseif b[i, j] = "↑"
    then PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

این اسلاید ها بر مبنای نکات مطرح شده در فرادرس

«آموزش طراحی الگوریتم»

تهیه شده است.

برای کسب اطلاعات بیشتر در مورد این آموزش به لینک زیر مراجعه نمایید

faradars.org/fvsft1092