فصل سوم

برنامه نویسی پویا

مقدمه

- همان طور که در فصل قبل دیدیم روش تقسیم و حل یک روش بالا به پایین (Top-Down) بود یعنی به کمک یک رابطه بازگشتی از مساله یک راست به سراغ مساله اصلی (حالت n) رفته و آن را به نمونه های کوچک تقسیم می کنیم .
- این نمونه ها را آن قدر کوچک می کنیم تا بتوانیم بالاخره آن ها را حل کنیم .سپس با ترکیب این جواب نمونه های کوچک به جواب نمونه اصلی می رسیم.

مقدمه

- برنامه نویسی پویا، از این لحاظ که نمونه به نمونه های کوچکتر تقسیم می شود ، مشابه روش تقسیم و حل است
- ولی در این روش ، نخست نمونه های کوچک تر را حل می کنیم، نتایج را ذخیره می کنیم و بعدا هر گاه به یکی از آن ها نیاز پیدا شد، به جای محاسبه دوباره کافی است آن را بازیابی کنیم.
- در برنامه نویسی پویا حل را از بالا به پایین(Down-Top) در یک آرایه (یا یک سری آرایه) بنا می کنیم.
- تشابه تکنیک تقسیم و غلبه با تکنیک پویا آن است که در هر دو رابطه بازگشتی داریم و تفاوت آن ها دراین است که تقسیم و غلبه یک روش کل به جزء و تکنیک پویا یک روش جزء به کل است.

مقدمه

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

$$T(n) = T(n-1)+3 = (T(n-2)+3)+3 = T(n-2)+2*3$$

$$= T(n-3)+3*3 = T(n-4)+4*3 = \dots$$

$$= T(n-(n-1))+(n-1)*3 = T91)+(n-1)*3 = 1+(n-1)*3$$

$$= 3n-2$$

مقدمه

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

T(1) = 1
T(2) = T(1) + 3 = 1 + 3 = 4
T(3) = T(2) + 3 = 4 + 3 = 7
T(4) = T(3) + 3 = 7 + 3 = 10
T(5) = T(4) + 3 = 10 + 3 = 13
.
.
.
T(n) = 3n - 2

مقدمه

- **مثال : محاسبه سری فیبوناچی**

روش تقسیم و غلبه

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

روش پویا

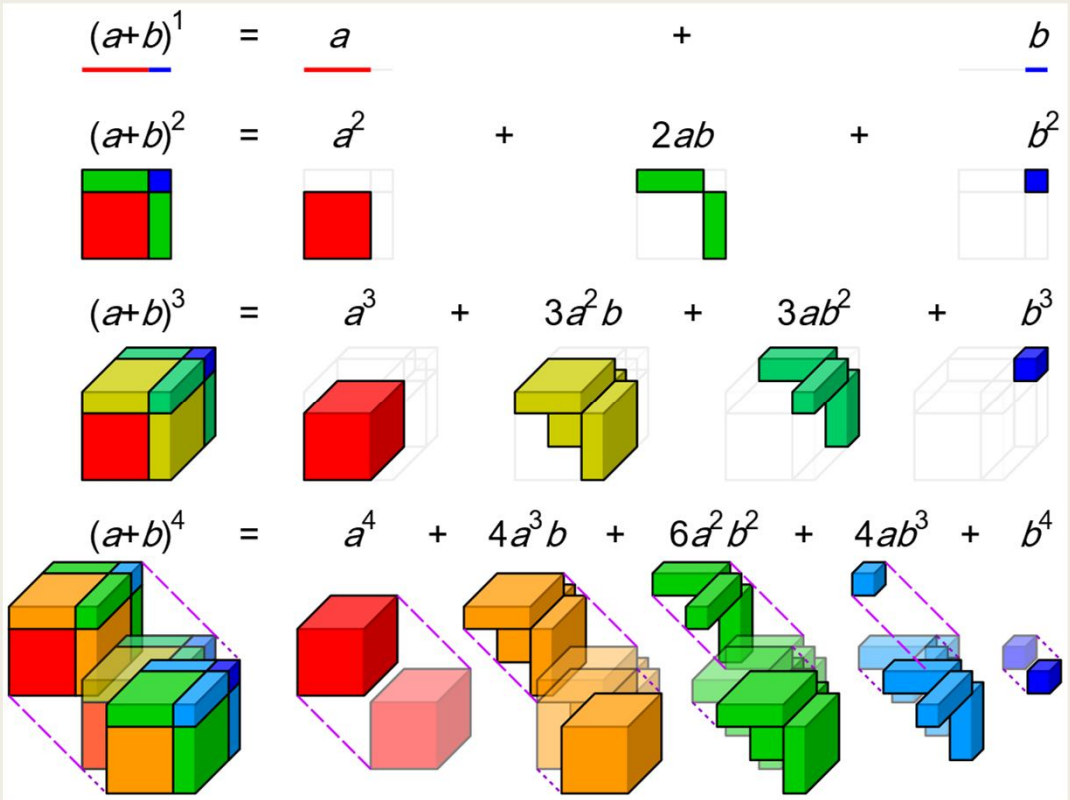
```
int fib(int n)
{
  int f[n+2];
  int i;
  f[0] = 0;
  if (n>0) {
    f[1] = 1;
    for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];
  }
  return f[n];
}
```

مراحل برنامه نویسی پویا

- مراحل بسط یک الگوریتم برنامه نویسی پویا به شرح زیر است:
  - ارائه یک ویژگی بازگشتی برای حل نمونه ای از مسئله .
  - حل نمونه ای از مسئله به شیوه جزء به کل با حل نمونه های کوچک تر

الگوریتم ضریب دوجمله ای – تقسیم و حل

- در ریاضیات هریک از اعداد صحیح که به صورت ضریب در بسط دوجمله‌ای ظاهر می‌شوند را ضریب دوجمله‌ای می‌گویند.



الگوریتم ضریب دوجمله ای – تقسیم و حل

- با تنظیم این ضرایب به صورت یک آرایه مثلثی، **مثلث خیام پاسکال** به وجود می‌آید.

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

الگوریتم ضریب دوجمله ای – تقسیم و حل

- فرمول مرسوم بدست آوردن ضریب دو جمله ای برای کلیه مقادیر $n \geq k \geq 0$ بصورت زیر است:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \quad \text{for } 0 \leq k \leq n,$$

- اما همانطور که مشاهده می کنید، بدلیل استفاده از عمل فاکتوریل ، حتی برای مقادیر کوچک متغیرهای k و n ، مقدار k! و یا n! بزرگ (و یا حتی بسیار بزرگ) خواهد بود.
- پس ما نمی توانیم ضریب دوجمله ای را مستقیماً از این روش بدست آوریم.

الگوریتم ضریب دوجمله ای – تقسیم و حل

- با استفاده از فرمول زیر نیاز به محاسبه ی !n و یا !k را به !(n-1) و یا !(k-1) تقلیل می دهیم.

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

- همانطور که ملاحظه می کنید، برای محاسبه ی این عبارت، آن را به دو عبارت کوچکتر تبدیل می کنیم و به محاسبه ی آنها می پردازیم.
- در نتیجه به الگوریتم تقسیم و حل می رسیم.

الگوریتم ضریب دوجمله ای – تقسیم و حل

- با استفاده از فرمول زیر نیاز به محاسبه ی !n و یا !k را به !(n-1) و یا !(k-1) تقلیل می دهیم.

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

- همانطور که ملاحظه می کنید، برای محاسبه ی این عبارت، آن را به دو عبارت کوچکتر تبدیل می کنیم و به محاسبه ی آنها می پردازیم.
- در نتیجه به الگوریتم تقسیم و حل می رسیم.

الگوریتم ضریب دوجمله ای – تقسیم و حل

- الگوریتم محاسبه ضریب دو جمله ای در این روش با استفاده از روش بازگشتی به صورت زیر می باشد:

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

- ورودی ها: اعداد صحیح و مثبت n و k هستند که در آن $k \leq n$ است
- خروجی ها: ضریب دو جمله ای n و k بصورت یک عدد صحیح

الگوریتم ضریب دوجمله ای – تقسیم و حل

- **مثال**

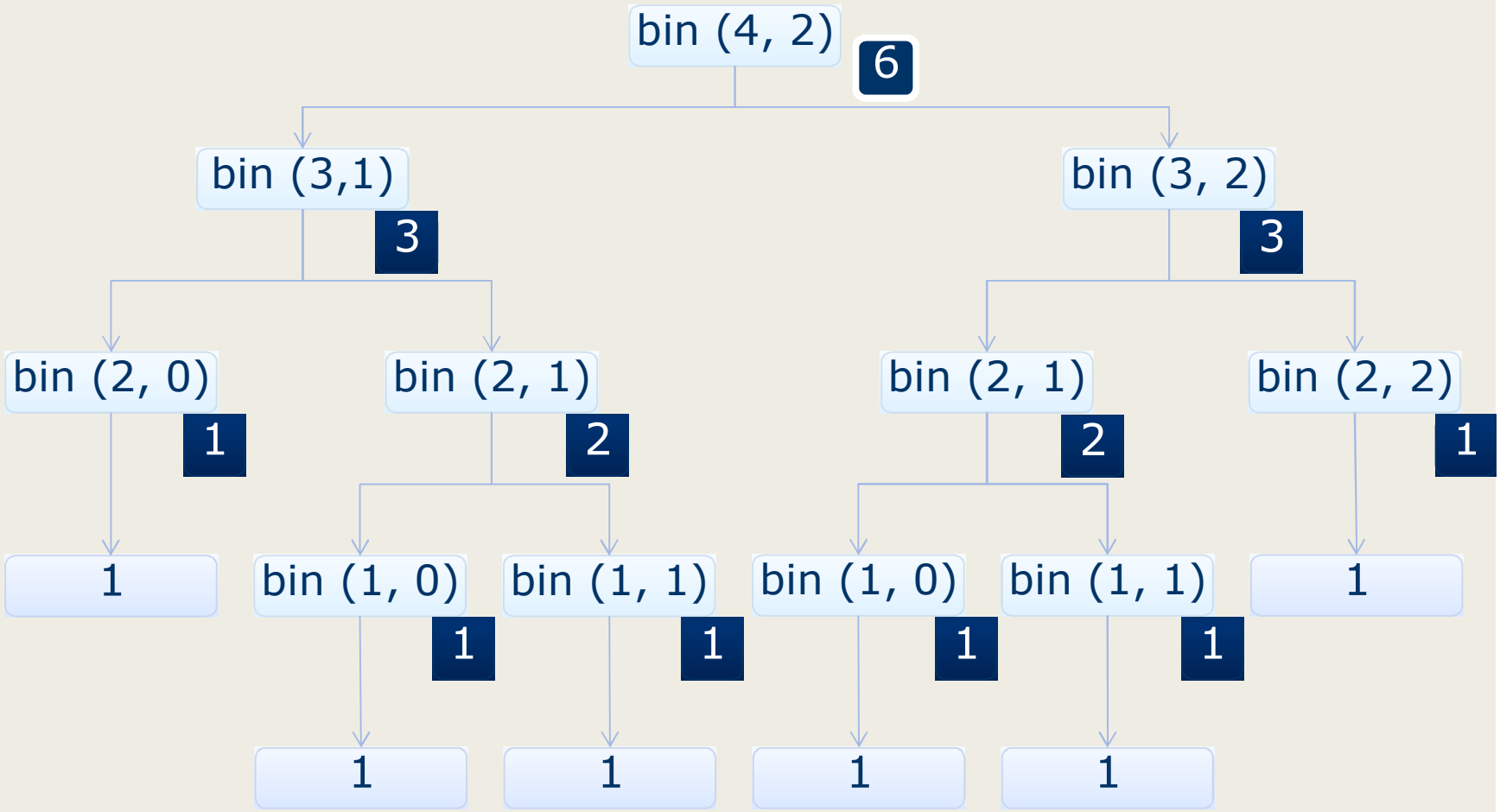
فرض کنید قرار است با استفاده از این الگوریتم ضریب دو جمله ای (۴،۲) را محاسبه کنیم.

پس $n = 4$ و $k = 2$ خواهد بود، بنابراین تابع ما به صورت زیر فراخوانی خواهد شد:

```
bin (4, 2)
```

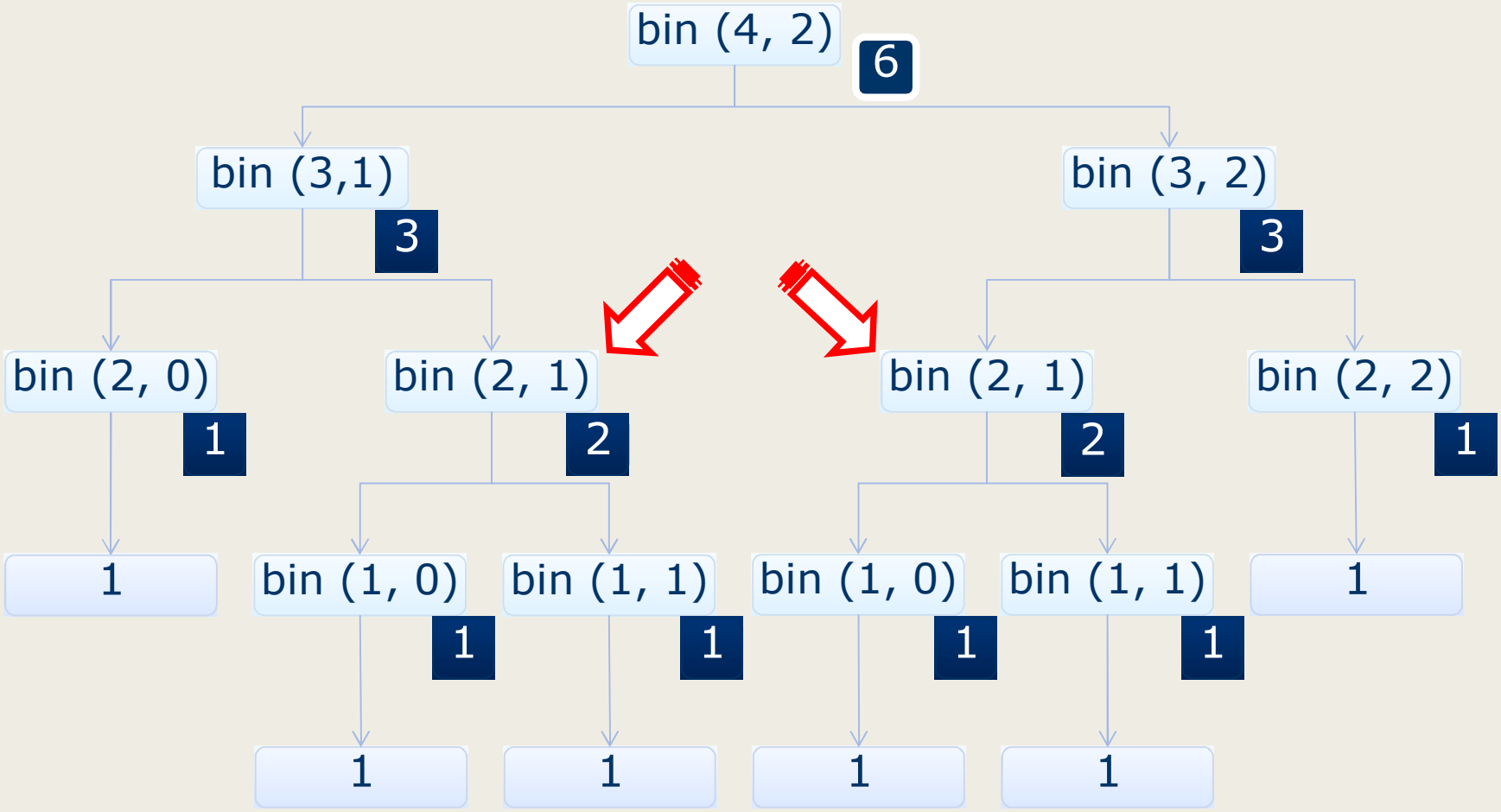
بدلیل استفاده از فراخوانی های بازگشتی و درک موضوع، درخت این سلسله فراخوانی ها را ترسیم می کنیم.

الگوریتم ضریب دوجمله ای – تقسیم و حل



الگوریتم ضریب دوجمله ای – تقسیم و حل

- **اما همانطور که مشاهده می کنید، در فراخوانی های بازگشتی، تعدادی از نمونه ها چندین بار حل می شوند و بطور جداگانه محاسبه می شوند.**



الگوریتم ضریب دوجمله ای – تقسیم و حل

- دیدیم برای محاسبه bin(n-1, k-1) و bin(n-1, k) هر دو نیاز به نتیجه ی bin(n-2, k-1) دارند و این مورد در هر بار فراخوانی به طور جداگانه ای محاسبه می شود.
- پس در روش تقسیم و حل تا زمانی که نمونه به چند نمونه کوچکتر تقسیم شود که آن نمونه ها تقریباً به بزرگی نمونه اولیه باشند، این الگوریتم کارایی ندارد، آن چنان که برای تعیین $\binom{n}{k}$ نیاز به محاسبه $2 * \binom{n}{k} - 1$ جمله خواهد بود.
- حال با استفاده از الگوریتم پویا، الگوریتمی با کارایی بیشتر و بالاتر طراحی می کنیم.

الگوریتم ضریب دوجمله ای – برنامه نویسی پویا

- با استفاده از همان فرمولی که در الگوریتم قبلی بیان کردیم:

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

- این بار با استفاده از برنامه نویسی پویا پیاده سازی میکنیم.
- با آرایه ای دو بعدی مثلاً با نام B که B[i][j] مبین $\binom{i}{j}$ است.

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

الگوریتم ضریب دوجمله ای – برنامه نویسی پویا

- مسـئله را به شیـوه ی جزء به کل حـل نموده و به پیش می رویم.
- سلول های آرایه را به ترتیب از سطر و ستون های کوچک شروع کرده و به جلو می رویم، تا آنجا که به B[n][k] که همان $\binom{n}{k}$ است برسیم.

ضریب دو جمله ای با استفاده از برنامه نویسی پویا

```
int bin2(int n ,int k )
{
  index i, j;
  int B[0..n][0..k]
  for  (i = 0;i ≤ n ;i ++)
    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]:
}
```

ضریب دو جمله ای با استفاده از برنامه نویسی پویا

- **مثال**

فرض کنید قرار است با استفاده از این الگوریتم ضریب دو جمله ای (۴،۲) را محاسبه کنیم. پس n = 4 و k = 2

n \ k	0	1	2
0	1		
1	1	1	
2	1	2	1
3	1	3	3
4	1	4	6

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

ضریب دو جمله ای با استفاده از برنامه نویسی پویا

- دیدیم که در این روش برخلاف روش قبلی سربار اضافی نداشتیم (محاسبه ی مجدد موردهای محاسبه شده) چرا که این الگوریتم جزء به کل بوده و برای محاسبه مقادیر جدید مورد نیاز، آنها را از مقادیر قبلی بدست می آورد. (نه آنکه مجدداً به محاسبه آنها بپردازد)
- در نهایت به مقایسه ی کارایی دو الگوریتم می پردازیم.

ضریب دو جمله ای با استفاده از برنامه نویسی پویا

- همانطور که قبلاً بیان شد، در الگوریتم تقسیم و حل، تعداد جملاتی که الگوریتم برای تعیین $\binom{n}{k}$ محاسبه می کند $2 * \binom{n}{k} - 1$ است.
- این در حالی است که تعداد کل گذرها در الگوریتم برنامه نویسی پویا با اثبات زیر برابر $\theta(nk)$ خواهد بود.

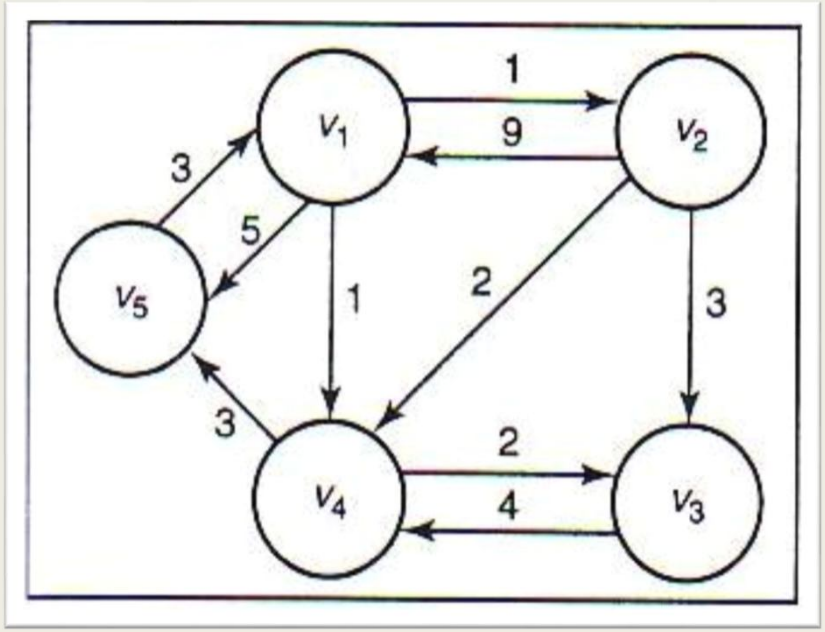
$\overbrace{\text{بار } n\text{-}k+1}$

- تعداد گذرها از حلقه j: $1+2+3+4+\ldots+k+(k+1)+(k+1)+\ldots+(k+1)$
- پس:

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

الگوریتم فلوید

- **نمونه سوال: یک مشکل متداول در سفرهای هوایی , هنگامی که پرواز مستقیم وجود نداشته باشد , تعیین کوتاه ترین مسیرپرواز از شهری به شهردیگر است.**
- **کوتاهترین مسیر بین دو گره در گراف**



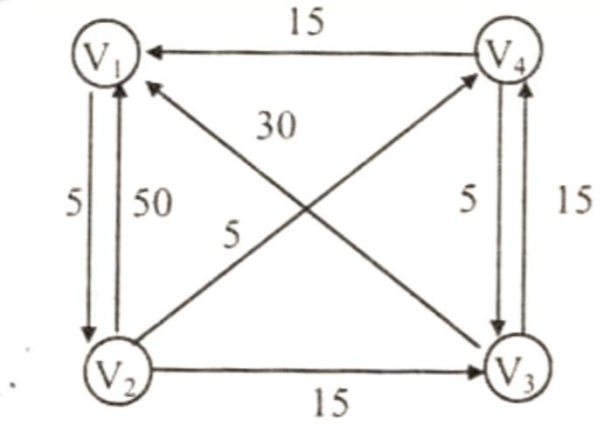
تعاریف اولیه

- **حلقه: مسیری از خود راس به خودش**
- **مسیر ساده: اگرمسیری هیچگاه از دو راس نگذرد را مسیر ساده گویند.**
- **طول مسیر: در یک گراف وزن دارحاصل جمع وزن ها را گویند و در یک گراف بدون وزن تعداد رئوس موجود**
- **مسئله کو تاه ترین مسیر که یک مسئله بهینه سازی می باشد شامل موارد زیر است:**
  - **از یک نود به نود دیگر**
  - **از یک نود به تمامی نودها**
  - **از تمامی نودها به همدیگر**

تعاریف اولیه

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

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



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

روش محاسبه تمامی مسیرها

- فرض کنید از هر راس به همه رئوس دیگر یک یال وجود دارد
- در این صورت، زیر مجموعه ای از همه مسیر ها، عبارت از مجموعه ای خواهد بود که راس نخست شروع می شود و به راسی دیگر ختم می شود و از همه رئوس دیگر عبور می کنند. چون راس دوم در چنین مسیری می تواند هر یک از n-2 راس باشد, راس سوم در چنین مسیری می تواند هر یک از n-3 راس باشد و ... و راس دومی به آخری روی چنین مسیری فقط می تواند یک راس باشد، تعداد کل مسیرها از یک راس که از همه رئوس دیگر بگذرد

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

روش محاسبه تمامی مسیرها

- ابتدا الگوریتمی را به دست می آوریم که فقط طول کوتاه ترین مسیرها را به ما می دهد . سپس قدری آن را اصلاح می کنیم تا کوتاه ترین مسیر را نیز برای ما نمایش دهد.
- به عبارتی دیگر هدف فعلی ما محاسبه ماتریس D زیر از روی ماتریس W (برای شکل گراف قبلی) است :

$$W = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{bmatrix} \end{array} \quad \Longrightarrow \quad D = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \end{array}$$

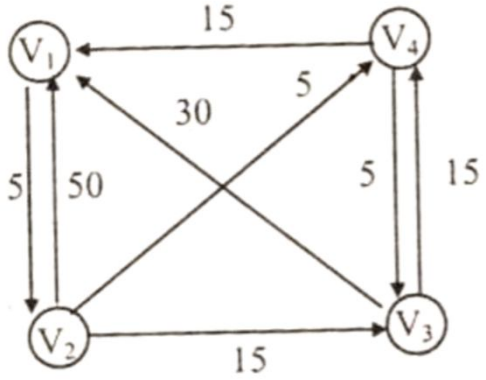
برنامه نویسی پویا و کوتاهترین مسیر

- **با استفاده از برنامه نویسی پویا، یک الگوریتم زمانی درجه سوم برای این مسئله ایجاد می کنیم**

$$\text{shortestPath}(i,j,k) = \min\Big(\text{shortestPath}(i,j,k-1),\ \text{shortestPath}(i,k,k-1) + \text{shortestPath}(k,j,k-1)\Big)$$

$$\text{shortestPath}(i,j,0) = w(i,j)$$

روش محاسبه تمامی مسیرها



$$D_0 = W = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{bmatrix} \end{array}$$

$$D_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \end{array} \Rightarrow D_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 12 & 20 & 5 & 0 \end{bmatrix} \end{array}$$

$$\Rightarrow D_3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \end{array} \Rightarrow D_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \end{array}$$

الگوریتم فلوید برای یافتن کوتاه ترین مسیر

- **با استفاده از برنامه نویسی پویا، یک الگوریتم زمانی درجه سوم برای این مسئله ایجاد می کنیم**

```
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]);
   }
```

الگوریتم فلوید برای یافتن کوتاه ترین مسیر

- تحلیل پیچیدگی زمانی در **بدترین** حالت

عمل اصلی: دستورهای موجود در حلقه for

اندازه ورودی: n ، تعداد رئوس گراف

$$T(n) = n \times n \times n = n^3 \in \theta(n^3)$$

الگوریتم فلوید دوم برای یافتن کوتاه ترین مسیر

```
void  floyd2  ( int n,
                const number W [][],
                      number D [][],
                      index    P [][])
  {
      index  i , j , k ;
      for  (  i = 1 ;  i ≤ n ; i ++)
          for  ( j = 1 ; j ≤ n ; j++)
```

الگوریتم فلوید دوم برای یافتن کوتاه ترین مسیر

```
P [i] [j] = 0;
        D = W;
  for (  k = 1 ;  k <= n ; k ++)
     for  (  i = 1 ;  i <= n ; i ++)
        for (  j = 1 ;  j <= n ; j ++)
           if ( D [i] [k] + D [k] [j] < D [i] [j] )  {
              P [i] [j] = k;
              D [i] [j] = D [i] [k] + D [k] [j];
           }
        }
```

الگوریتم نمایش کوتاه ترین مسیر

- **مسئله: چاپ رئوس واسطه روی کوتاه ترین مسیر از راسی به راس دیگر در یک گراف موزون.**

```
void path ( index q , r)
 {
    if  (P [q] [r] != 0 )  {
          path (q , P [q] [r] );
          cout << “v” << P [q] [r];
          path (P [q] [r] , r );
        }
 }
```

برنامه نویسی پویا و مسائل بهینه سازی

- **حل بهینه ، سومین مرحله از بسط یک الگوریتم برنامه نویسی پویا برای مسائل بهینه سازی است. مراحل بسط چنین الگوریتمی به شرح زیر است:**

1. **ارائه یک ویژگی بازگشتی که حل بهینه ی نمونه ای از مسئله را به دست می دهد.**
2. **محاسبه مقدار حل بهینه به شیوه ی جزء به کل.**
3. **بنا کردن یک حل نمونه به شیوه ی جزء به کل.**

برنامه نویسی پویا و مسائل بهینه سازی

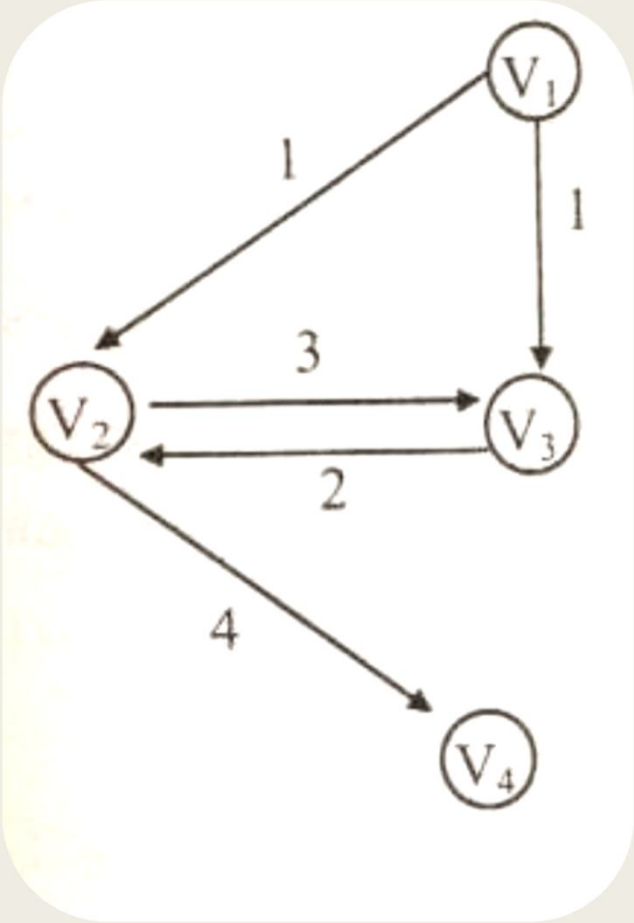
- هر مسئله بهینه سازی را نمی توان با استفاده از برنامه نویسی پویا حل کرد چرا که باید اصل بهینگی در مسئله صدق کند.
- اصل بهینگی در یک مسئله صدق می کند اگر یک حل بهینه برای نمونه ای از مسئله، همواره حاوی حل بهینه برای همه ی زیر نمونه ها باشد.

برنامه نویسی پویا و مسائل بهینه سازی

- **مثال:**
  - **برای مسئله کوتاه ترین مسیر اصل فوق برابر است چرا که اگر V_k یک راس روی مسیر بهینه از V_i به V_j باشد آنگاه زیر مسیرهای V_i به V_k و V_k به V_j نیز باید بهینه باشند.**
  - **ولی در مسئله یافتن طولانی ترین مسیر بین دو راس اصل بهینگی صادق نبوده و نمی توان آن را از طریق برنامه نویسی پویا حل کرد .**

برنامه نویسی پویا و مسائل بهینه سازی

- در شکل روبرو :
- طولانی ترین مسیر ساده (مسیر بهینه) از V_1 به V_4، مسیر $\langle V_1, V_2, V_3, V_4 \rangle$ می باشد ولی زیر مسیر $\langle V_1 , V_3 \rangle$ یک زیر مسیر بهینه(طولانی ترین) از V_1 به V_3نیست و مسیری طولانی تر به صورت $\langle V_1 , V_2 , V_3 \rangle$ بین آن دو وجود دارد.



ضرب زنجیره ای ماتریس ها

- همان طور که می دانید ضرب ماتریس ها خاصیت جابجایی ندارد ولی خاصیت شرکت پذیری را دارد .
- همچنین ضرب دو ماتریس $A \times B$ به شرطی تعریف شده است که بعد وسط آن ها یکسان باشد. یعنی تعداد ستون های A با تعداد سطرهای B برابر باشد.
- می توان اثبات کرد که عملیات $A_{m \times n} \times B_{n \times k}$ در کل به $m \times n \times k$ عمل ضرب نیاز دارد.

ضرب زنجیره ای ماتریس ها

- برای ضرب دو ماتریس باید هر سطر از ماتریس اول در ستونهای ماتریس دوم ضرب شود و حاصل هر یک با هم جمع شود. در مثال زیر ماتریس حاصلضرب یک ماتریس ۴×۲ است.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{2*3} * \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix}_{3*4}$$

$$\begin{bmatrix} 1*7+2*2+3*6 & 1*8+2*3+3*7 & 1*9+2*4+3*8 & 1*1+2*5+3*9 \\ 4*7+5*2+6*6 & 4*8+5*3+6*7 & 4*9+5*4+6*8 & 4*1+5*5+6*9 \end{bmatrix}$$

ضرب زنجیره ای ماتریس ها

- اگر بیش از دو ماتریس داشته باشیم که بین آنها علامت × است می توان با استفاده از خاصیت شرکت پذیری به شیوه های مختلف آنها را در هم ضرب کرد.
- مثال: می خواهیم ۴ ماتریس زیر را در هم ضرب کنیم و سپس تعداد اعمال ضرب را در هر حالت به دست آوریم:

A * B * C * D

20*2 2*30 30*12 12*8

ضرب زنجیره ای ماتریس ها

- A (B (CD)

تعداد اعمال ضرب: 30*12*8+2*30*8+20*2*8=3680

- (AB)(CD)

تعداد اعمال ضرب: 20*2*30+30*12*8+20*30*8=8880

- A ((BC) D)

تعداد اعمال ضرب: 2*30*12+2*12*8+20*2*8=1232

- ((AB)C)D

تعداد اعمال ضرب: 20*2*30+20*30*12+20*12*8=10320

- (A(BC))D

تعداد اعمال ضرب: 2*30*12+20*2*12+20*12*8=3120

ضرب زنجیره ای ماتریس ها

- **قضیه :**

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

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

ضرب زنجیره ای ماتریس ها

- اگر T(n) تعداد ترتیب های متفاوت برای ضرب n ماتریس A_1، A_2، ...،An باشد ، زیر مجموعه از این ترتیب ها حالتی است که در آنها A_1 آخرین ماتریسی است که در مجموعه ضرب می شود یعنی :

$$A_1\underbrace{(A_2\ A_3 \ldots A_n)}_{T(n-1)}$$

در نتیجه

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

ضرب زنجیره ای ماتریس ها

- ماتریس زیر را میخواهیم در هم ضرب کنیم:

A1 * A2 * A3 * A4 * A5 * A6

5*2 2*3 3*4 4*6 6*7 7*8

D= [5 2 3 4 6 7 8]

- برای ضرب A4 تا A6 دو ترتیب زیر را داریم:

1: (A4 A5) A6 = D3*D4*D5 + D3*D5*D6= 4*6*7+4*7*8 = 392

2: A4 (A5 A6) = D4*D5*D6 + D3*D4*D6= 6*7*8+4*6*8 = 528

M[4][6] = Min(392,528) = 392

ضرب زنجیره ای ماتریس ها

- مجموعه ماتریسهای زیر را در نظر بگیرید:

$$A1 \times A2 \times \ldots \times Ai \times \ldots \times Aj \times \ldots \times An$$

- منظور از M[i][j] حداقل تعداد ضرب برای ماتریسهای iام تا jام است بشرطی که i<j یک ماتریس بالا مثلثی است

$$M[i][i]=0$$

$$M[i][i+1]=d_{i-1}*d_i*d_{i+1}$$

ضرب زنجیره ای ماتریس ها

- مجموعه ماتریسهای زیر را در نظر بگیرید:

$$A1 \times A2 \times \ldots \times Ai \times \ldots \times Aj \times \ldots \times An$$

- منظور از M[i][j] حداقل تعداد ضرب برای ماتریسهای iام تا jام است بشرطی که i<j یک ماتریس بالا مثلثی است

$$M[i][i]=0$$

$$M[i][i+1]=d_{i-1}*d_i*d_{i+1}$$

ضرب زنجیره ای ماتریس ها

	d0	d1	d2	d3	d4	d5	d6
d	5	2	3	4	6	7	8

	1	2	3	4	5	6
1	0	30				
2		0				
3			0			
4				0		
5					0	
6						0

M[2][3]=?

ضرب زنجیره ای ماتریس ها – مثال

- در این صورت دو ترتیب برای ضرب وجود خواهد داشت:

1: (A1 A2)(A3)

2: (A1)(A2 A3)

- همانطور که در قبل اشاره کردیم حداقل تعداد ضرب دو ماتریس را با M[1][3] بیان می کنیم.

1: (A1*A2) * (A3) = A1 * A2 * A3

M[1][2] + M[3][3]+ d0*d1*d2

2: (A1) * (A2*A3) = A1 * A2 * A3

M[1][1] + M[2][3] + d0*d1*d2

ضرب زنجیره ای ماتریس ها – مثال

- حال در واقع داریم :

1: M[1][2] + M[3][3] + d0*d2*d3 = 30 + 4 + 5*3*4 =94

2: M[1][1] + M[2][3] + d0*d1*d3 = 0 + 24 + 5*2*4 =64

- باید بین دوترتیب فوق مینیمم بگیریم . مقدار مینیمم در M[1][3] قرار میگیرد.

M[1][3]= 64

- در اینصورت برای بدست آوردن درایه M[i][j] از فرمول زیر استفاده میکنیم :

M[i][j] = Minimum (M[i][k] + M[k+1][j] + di-1 dk dj)

- به شرطی که : $i < j \ , I \leq k \leq j-1$
- و داریم: $M[i][j]=0$

ضرب زنجیره ای ماتریس ها – مثال

	d6	d5	d4	d3	d1	d2	d0
d	5	2	3	4	6	7	8

	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	366
6						0

ضرب زنجیره ای ماتریس ها – الگوریتم

```
// Matrix A[i] has dimension dims[i-1] x dims[i] for i = 1..n
MatrixChainOrder(int dims[])
{
   // length[dims] = n + 1
   n = dims.length - 1;
   // m[i,j] = Minimum number of scalar multiplications (i.e., cost)
   // needed to compute the matrix A[i]A[i+1]...A[j] = A[i..j]
   // The cost is zero when multiplying one matrix
   for (i = 1; i <= n; i++)
      m[i, i] = 0;

   for (len = 2; len <= n; len++) { // Subsequence lengths
      for (i = 1; i <= n - len + 1; i++) {
         j = i + len - 1;
         m[i, j] = MAXINT;
         for (k = i; k <= j - 1; k++) {
            cost = m[i, k] + m[k+1, j] + dims[i-1]*dims[k]*dims[j];
            if (cost < m[i, j]) {
               m[i, j] = cost;
               s[i, j] = k; // Index of the subsequence split that achieved minimal cost
            }
         }
      }
   }
}
```