دانلود جزوات دانشگاهی رشته کامپیوتر

با مدیریت سمیه عرب باقری

فصل چهارم:

برنامه نویسی پویا

فصل چهارم
برنامه‌نویسی پویا

مفهوم برنامه‌نویسی پویا (Dynamic Programming)

همان‌طور که در فصل قبل دیدید روش تقسیم و حل یک روش بالا به پائین (Top-down) بود یعنی به کمک یک رابطه بازگشتی از مسأله، یک راست به سراغ مسأله اصلی (حالت n) رفته و آن را به نمونه‌های کوچک تقسیم می‌کنیم. این نمونه‌ها را آنقدر کوچک می‌کنیم تا بتوانیم بالاخره آنها را حل کنیم، سپس با ترکیب این جواب نمونه‌های کوچک، به جواب نمونه اصلی می‌رسیم.

در تکنیک برنامه‌نویسی پویا نیز به یک رابطه بازگشتی نیاز داریم که حالات بزرگتر را بر حسب حالات کوچکتر نشان دهد و از این نظر این تکنیک مثل تقسیم و غلبه است. ولی در این روش ابتدا نمونه‌های کوچکتر را حل کرده و نتایج را ذخیره می‌کنیم و سپس هرگاه به هر کدام آنها نیاز داشتیم به جای محاسبه دوباره، کافی است آن را بازیابی کنیم. در برنامه‌نویسی پویا حل را از پائین به بالا (down-Top) در یک آرایه (یا یک سری آرایه) بنا می‌کنیم.

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

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

روش اول (بالا به پائین) :

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

روش دوم (پائین به بالا) :

$$\begin{aligned} T(1) &= 1 \\ T(2) &= T(1) + 3 = 1 + 3 = 4 \\ T(3) &= T(2) + 3 = 4 + 3 = 7 \\ T(4) &= T(3) + 3 = 7 + 3 = 10 \\ T(5) &= T(4) + 3 = 10 + 3 = 13 \\ &\vdots \\ T(n) &= 3n - 2 \end{aligned}$$

مثال اول: سری فیبوناچی

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

روش تقسیم و غلبه	روش پویا

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

همان‌طور که قبلاً توضیح دادیم مرتبه اجرایی تقسیم و غلبه فوق $O(2^{\frac{n}{2}})$ و مرتبه اجرایی روش پویای فوق $O(n)$ است. علت آنکه تقسیم و غلبه فیبوناچی مرتبه زیادی دارد این است که نمونه‌های کوچکتر با هم ارتباط دارند. مثلاً محاسبه جمله پنجم فیبوناچی به جملات سوم و چهارم نیاز دارد. ولی از آنجا که جملات سوم و چهارم هر دو به جمله دوم نیاز دارند با یکدیگر رابطه دارند و fib (2) بایستی چندین مرتبه محاسبه گردد.

ولی در مسائلی مثل مرتب‌سازی ادغامی نمونه‌های کوچکتر با هم ارتباطی ندارند و لذا روش تقسیم و غلبه برای آنها در زمان مناسبی جواب می‌دهد. در مرتب‌سازی ادغامی یک آرایه به آرایه‌های کوچکتر تقسیم می‌شود که هیچ ارتباطی با یکدیگر نداشته و هر یک به صورت مستقل مرتب می‌شوند.

پس در مسائلی که نمونه‌های کوچکتر با هم ارتباط دارند نباید از روش تقسیم و غلبه استفاده کنیم. در روش پویای حل سری فیبوناچی مشاهده می‌کنید که جواب‌های نمونه‌های کوچکتر در آرایه‌ای ذخیره می‌گردد. البته گاهی اوقات پس از طراحی با استفاده از آرایه (یا یک سری آرایه) می‌توان به گونه‌ای الگوریتم را اصلاح کرد که مصرف حافظه کمتر شود مثلاً الگوریتم پویای فوق را می‌توان به صورت زیر نیز انجام داد :

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

پس مراحل ایجاد یک الگوریتم پویا شامل مراحل زیر است :

۱ـ ایجاد یک خاصیت بازگشتی برای حل نمونه‌ای از مسأله

۲ـ حل نمونه‌ای از مسأله با روش جزء به کل از طریق حل نمونه‌های کوچکتر

مثال دوم : ضریب دوجمله‌ای

همان‌طور که می‌دانید بسط دوجمله‌ای نیوتن $(a+b)^n$ به صورت زیر می‌باشد :

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

و می‌دانیم که :

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

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

برای محاسبه ضریب دوجمله‌ای نیوتن یعنی $\binom{n}{k}$ می‌توان از فرمول مستقیم $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ استفاده کرد ولی محاسبه !n برای مقادیر نه چندان بزرگ n هم خیلی بزرگ خواهد بود، لذا باید راه‌حل بهتری پیدا کنیم.

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

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

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

پیاده‌سازی فرمول فوق به صورت الگوریتم تقسیم و غلبه به صورت زیر است:

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

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

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

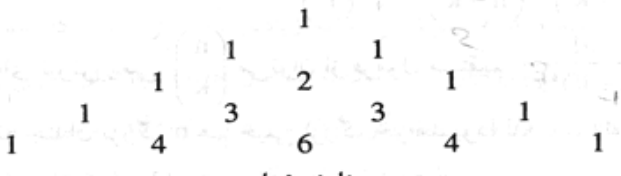
که بازدهی خیلی کمی دارد. همان‌طور که در فصل قبلی شرح دادیم روش تقسیم و غلبه برای مواردی مثل فوق که نمونه‌ای به دو نمونه، با اندازه نزدیک به نمونه اصلی، تقسیم می‌شود اصلاً مناسب نیست.

حال این مسأله را با تکنیک پویا و با پیچیدگی زمانی خیلی کمتر حل می‌کنیم.

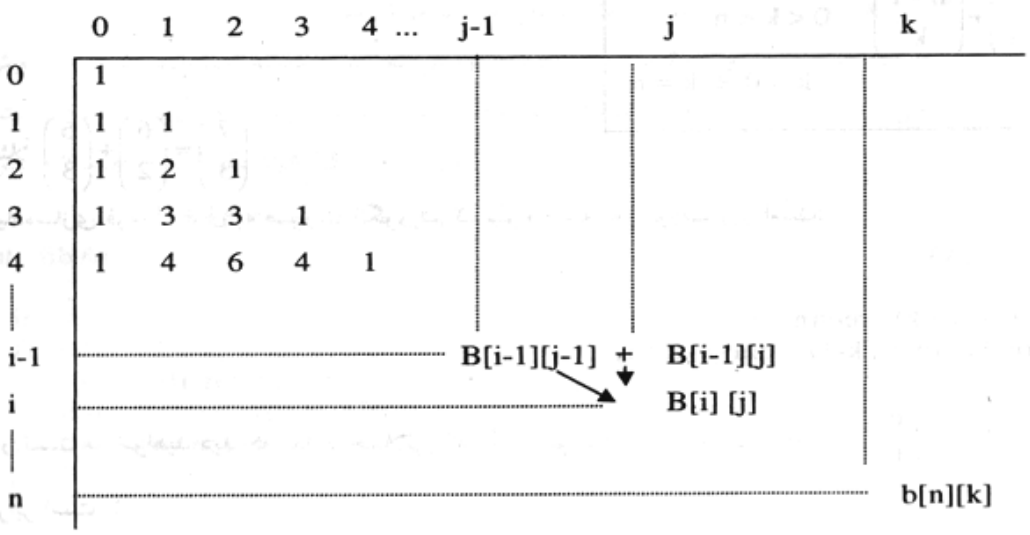
به خاطر دارید که ضرایب دو جمله‌ای را می‌توان از مثلث خیام به دست آورد. این مثلث را می‌توان بر مبنای فرمول زیر در یک ماتریس پیاده‌سازی کرد.

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

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



مثلث خیام



منظور از B[i][j] عبارت $\binom{i}{j}$ و $B[n][k]=\binom{n}{k}$ است. برنامه‌ای که ماتریس فوق را ایجاد می‌کند به صورت زیر است:

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

توجه کنید که در حلقه for مربوط به j مقدار نهایی را به جای k عبارت minimum(i,j) نوشته‌ایم چرا که بالای قطر اصلی این ماتریس را نیاز نداریم.

از آنجا که برنامه فوق دو حلقه تودرتوی for دارد روشن است که مرتبه اجرایی آن $\theta(nk)$ می‌باشد. پس:

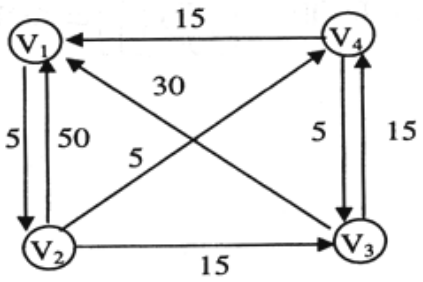
مرتبه اجرایی محاسبه ضرایب دوجمله‌ای با روش پویا $=\theta(nk)$

همان‌طور که مشاهده می‌کنید سرعت روش پویا برای محاسبه ضرایب دوجمله‌ای به مراتب بیشتر از روش تقسیم و غلبه است. همچنین مشاهده کردید که در روش پویای فوق مانند تقسیم و غلبه از یک فرمول بازگشتی استفاده کردیم. ولی به جای آنکه از ابتدا به سراغ محاسبه $\binom{n}{k}$ برویم از حل نمونه‌های کوچک شروع کرده، آنها را در آرایه‌ای ذخیره کردیم . تا ترکیب آنها (از پائین به بالا) به جواب نهایی رسیدیم. در این روش پویا هر نمونه کوچکتر فقط یک بار محاسبه شد.

همان‌طور که در ماتریس فوق مشاهده کردید برای محاسبه هر سطر فقط به سطر بالای آن نیاز است لذا می‌توان به سادگی مصرف حافظه را به یک آرایه تک‌بعدی B[0 .. k] محدود ساخت. این کار را به عنوان تمرین انجام دهید. همچنین در بعضی موارد با توجه به فرمول $\binom{n}{k}=\binom{n}{n-k}$ می‌توان برنامه فوق را بهینه‌تر کرد.

مثال سوم : الگوریتم فلوید (Floyd) جهت یافتن کوتاه‌ترین مسیر

یکی از مسائل معمول یافتن کوتاه‌ترین مسیر هوایی بین دو شهر است هنگامی که بین آنها پرواز مستقیمی وجود ندارد. جهت نمایش شهرها و طول مسافت بین آنها می‌توان از گراف وزن‌دار شبیه شکل زیر استفاده کرد:



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

گراف وزن‌دار فوق را توسط ماتریس W که به ماتریس هم‌جواری (adjacency) معروف است، طبق فرمول زیر نمایش داده‌ایم :

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

توجه کنید در تعریف فوق فرض کرده‌ایم، مشابه شکل روبه‌رو خود حلقه نداریم :

و قطر اصلی ماتریس W صفر است. W مخفف وزن (weight) است. کوتاه‌ترین مسیر (Shortest path) بین دو رأس باید یک مسیر ساده (Simple) باشد. مسیر ساده مسیری است که از یک رأس دوبار عبور نشود. مثلاً در شکل فوق $<V_1, V_2, V_3>$ یک مسیر ساده است ولی مسیر $<V_2,V_3,V_4,V_3,V_1>$ مسیر ساده نیست. بدیهی است که یک مسیر ساده هیچ‌گاه حاوی زیر مسیری که حلقه باشد، نیست. طول یک مسیر جمع وزن‌های نوشته شده بر روی یال‌هاست.

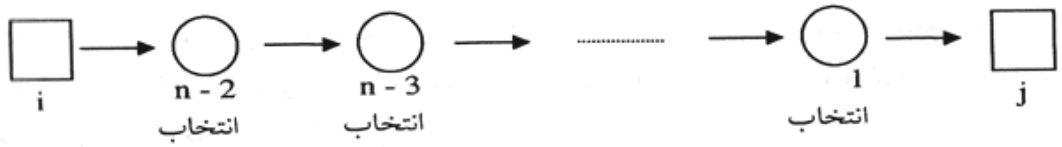
مثلاً در شکل گراف فوق دو مسیر ساده از V_2 به V_3 و با طول‌های زیر وجود دارد :

طول $< V_2, V_3 > = 15$

طول $<V_2, V_4, V_3> = 5 + 5 = 10$

پس $<V_2, V_4, V_3>$ کوتاه‌ترین مسیر از V_1 به V_3 است و ما می‌خواهیم این کوتاه‌ترین مسیر را پیدا کنیم.

یک الگوریتم ساده ولی بسیار کند آن است که تمامی مسیرهای بین گره i به j را پیدا کرده و کمترین آنها را به دست آوریم. مثلاً در حالتی که گراف با n گره کامل بوده و از هر گره به تمام گره‌های دیگر یالی وجود دارد برای رفتن از گره i به j (به شرطی که بخواهیم از همه رئوس دیگر عبور کنیم) باید از n - 2 گره بگذریم. مطابق شکل زیر هنگام حرکت از گره i تعداد n - 2 انتخاب داریم، پس از آن برای رفتن روی گره سوم n-3 انتخاب داریم و الی آخر.



پس تعداد کل راه‌های ممکن O(n!) است که بسیار زیاد است :

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

در ادامه الگوریتمی را شرح می‌دهیم که از مرتبهٔ $O(n^3)$ است.

ابتدا الگوریتمی را به دست می‌آوریم که فقط طول کوتاه‌ترین مسیرها را به ما بدهد. سپس قدری آن را اص
می‌کنیم تا کوتاه‌ترین مسیر را نیز برای ما نمایش دهد. به عبارتی دیگر هدف فعلی ما محاسبه ماتریس D ر
روی ماتریس W (برای شکل گراف قبلی) است :

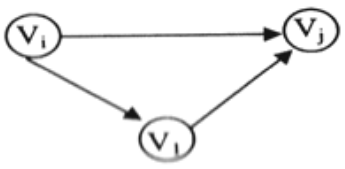
$$W = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{bmatrix} \end{array} \Longrightarrow D = \begin{array}{c} \\ 1 \\ 2 \\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

اینکه D[2][3]=10 است بدین معناست که از گره V_2 به گره V_3 کوتاه‌ترین مسیر طول 10 را دارد.

برای حل این مسأله لازم است ماتریس‌های D_0 تا D_n را به ترتیب از روی ماتریس W به دست آوریم (n
رئوس است). D_0 برابر W بوده و D_1 را از روی D_0 به دست می‌آوریم. سپس D_2 را از روی
D_3 را از روی D_2 و ... نهایت D_4 را به کمک D_3 محاسبه می‌کنیم که همان D نهایی مورد نظر ا
منظور از $D_k[i][j]$ طول کوتاه‌تر مسیر از V_i به V_j فقط با استفاده از رئوس موجود در مجم
$\{V_1, V_2, \ldots, V_k\}$ به عنوان رئوس واسطه می‌باشد.

پس $D_0[i][j]$ یعنی کوتاه‌ترین مسیر از گره V_i به V_j بدون هیچ‌گونه واسطه‌ای. لذا بدیهی است که ماتر
D_0 همان W می‌باشد.

منظور از $D_1[i][j]$ یعنی کوتاه‌ترین مسیر از V_i به V_j وقتی که گره 1 بتواند واسطه راه قرار گیرد. بد
است که در اینحالت :



$D_1[i][j] = Min(D_0[i][j], D_0[i][1] + D_0[1][j])$

پس از اینکه ماتریس D_1 را برای همه مقادیر $D_1[i][j]$ محاسبه کردیم، به سراغ محاسبه ماتریس D_2 می‌رویم. منظور از $D_2[i][j]$ یعنی کوتاه‌ترین مسیر از V_i به V_j وقتی که گره 1 و گره 2 بتوانند واسطه راه قرار گیرند. بدیهی است که در اینحالت :

$D_2[i][j] = Min(D_1[i][j], D_1[i][2] + D_1[2][j])$

منظور فرمول فوق این است که ما کوتاه‌ترین مسیر از گره i به j با واسطه‌گری شهر 1 را می‌دانیم. حال می‌خواهیم بدانیم اگر شهر 2 نیز بتواند واسطه راه باشد، کوتاه‌ترین مسیر کدام است؟ در این حال می‌گوئیم کوتاه‌ترین مسیر از شهر i به شهر 2 را با کوتاه‌ترین مسیر از شهر 2 به j جمع کن (البته این مسیرها می‌توانند حاوی شهر 1 هم باشند). سپس این حاصل جمع را با مقدار قبلی که مسیر حداقل از شهر i به j (فقط به کمک شهر 1) است مقایسه کن. مینیمم این دو مقدار جواب مورد نظر است.

توجه کنید هسته مرکزی حل این مسأله به صورت پویا همین پاراگراف فوق است که بیان بازگشتی برای حل مسأله می‌باشد. بنابراین فرمول اصلی حل مسأله به صورت زیر است :

$$D_k[i][j] = Min(D_{k-1}[i][j], D_{k-1}[i][k] + D_{k-1}[k][j])$$

با توجه به توضیحات فوق الگوریتم فلوید برای کوتاه‌ترین مسیر به صورت زیر است. اندیس آرایه‌ها را از 1 تا n در نظر گرفته‌ایم. هدف یافتن ماتریس D از روی ماتریس W است.

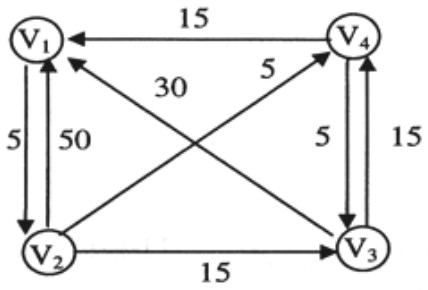
```
for (k = 1;  k <= n;  k ++)
   for (i = 1 ; i <= n ; i ++)
      for (j = 1; j <= n; j ++)
         D[i][j] = Min(D[i][j] , D[i][k] + D[k][j]);
```

توجه کنید از آنجا که پس از محاسبه ماتریس D_k نیازی به ماتریس D_{k-1} نداریم در هر مرحله جواب D_k را در ماتریس قدیمی D_{k-1} ذخیره کرده‌ایم و در نهایت ماتریس D_n جواب مورد نظر است.

از آنجا که الگوریتم فوق از سه حلقه تودرتو تشکیل شده است بدیهی است که مرتبه اجرایی آن $\theta(n^3)$ است.

مرتبه اجرایی الگوریتم فلوید $= \theta(n^3)$

برای اینکه مراحل الگوریتم را به خوبی درک کنید با روش دستی مرحله به مرحله ماتریس‌های زیر را به دست اورید :



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

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

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

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

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

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

هنگام محاسبه ماتریس D_1 ، سطر 1 و ستون 1 ماتریس D_0 تغییری نمی‌کند. هنگام محاسبه ماتریس D_2، سطر 2 و ستون 2 ماتریس D_1 تغییری نمی‌کند. هنگام محاسبه ماتریس D_3، سطر 3 و ستون 3 ماتریس D_2 تغییری نمی‌کند و ....

تا اینجا ما ماتریس D را پیدا کردیم که حداقل طول مسیر از هر گره i به j را نشان می‌دهد. در ادامه می‌خواهیم روشی نیز که در این مسیر کوتاه i به j وجود دارند را چاپ کنیم. برای این کار کافی است تغییر اندکی به برنامه محاسبه D بدهیم و ماتریس P را نیز به دست آوریم.

اتریس P یک ماتریس $n \times n$ با اندیس‌های 1 تا n است که مقدار اولیه عناصر آن صفر است :

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

دین ترتیب P[i][j] = 0 است اگر هیچ رأس واسطه‌ای بین i و j وجود نداشته باشد و در غیر این صورت اگر P[i][j] = k باشد یعنی کوتاه‌ترین مسیر از i به j از رأس k می‌گذرد. در واقع k آخرین رأس واسطه در روی کوتاه‌ترین مسیر است که i را به j وصل می‌کند.

ثلاً در گراف مثال قبل ماتریس P برابر زیر خواهد شد (بررسی کنید) :

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

P[1,3] = 4 یعنی کوتاه‌ترین مسیر از رأس 1 به 3 شامل رأس 4 است. لذا به صورت بازگشتی P[1,4] و P[4,3] را بررسی می‌کنیم :

یعنی در مسیر گره 1 به 4 باید از گره 2 بگذریم : P[1,4] = 2

یعنی در مسیر گره 4 به 3 واسطه‌ای وجود ندارد : P[4,3] = 0

پس کوتاه‌ترین مسیر از گره 1 به 3 مسیر روبه‌رو است : $1 \rightarrow 2 \rightarrow 4 \rightarrow 3$

تابع زیر رئوس واسطه بین گره i تا j را چاپ می‌کند :

```
void   path (int i, int j)
{
   if (P[i][j] ! = 0) {
       path (i , P[i][j]);
       printf(P[i][j]);
       path(P[i][j], j);
   }
}
```

تابع path در بدترین حالت از مرتبه $W(n) \in \theta(n)$ بوده و لذا الگوریتم فلوید در کل از مرتبه $\theta(n^3)$ است که برای محاسبه ماتریس‌های D و P مورد نیاز است.

روش پویا برای حل دسته مسائل بهینه‌سازی

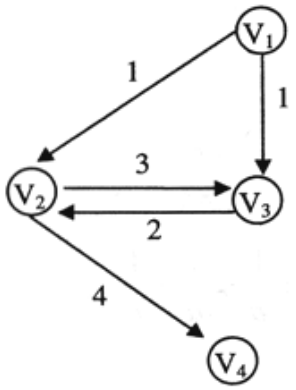
مسأله کوتاه‌ترین مسیر در یک گراف، جزو یک دسته از مسائل کلاسیک به نام مسائل **بهینه‌سازی** است. بسته به نوع مسأله مقدار بهینه حداقل یا حداکثر است. در مورد الگوریتم فلوید مسأله بهینه یافتن مقدار مینیمم بود. در بسیاری از مسائل بهینه‌سازی اگر بخواهیم همه حالات ممکن را در نظر گرفته و سپس مقدار مناسب را از بین آنها بیابیم، مرتبه زمانی از دسته نمایی یا فاکتوریل خواهد بود که بسیار زمان‌گیر است. لذا برای این دسته مسائل بهتر است که در صورت امکان از روش‌های دیگری مثل تکنیک پویا استفاده شود.

ممکن است، تصور شود که تمام مسائل بهینه‌سازی (Optimal Solution) را می‌توان با برنامه‌نویسی پویا حل کرد در حالی که چنین نیست. برای آنکه بتوان تکنیک پویا را برای یک مسأله بهینه‌سازی استفاده کرد لازم است **اصل بهینگی** (principle of optimality) برای آن مسأله صادق باشد.

تعریف : هنگامی که می‌گوئیم اصل بهینگی برای یک مسأله صادق است که یک حل بهینه برای نمونه‌ای از مسأله، همواره حاوی حل بهینه برای همه زیر نمونه‌های آن باشد.

مثلاً برای مسأله کوتاه‌ترین مسیر اصل فوق برقرار است چرا که اگر V_k یک رأس روی مسیر بهینه از V_i به V_j باشد آنگاه زیر مسیرهای V_i به V_k و V_k به V_j نیز باید بهینه باشند. ولی مثلاً در مسأله یافتن طولانی‌ترین مسیر بین دو رأس، اصل بهینگی صادق نبوده و نمی‌توان آن را از طریق برنامه‌نویسی پویا حل کرد.

جهت نمونه در شکل گراف زیر :



طولانی‌ترین مسیر ساده (مسیر بهینه) از V_1 به V_4، مسیر $< V_1, V_3, V_2, V_4 >$ می‌باشد ولی زیر مسیر $< V_1, V_3 >$ یک زیر مسیر بهینه (طولانی‌ترین) از V_1 به V_3 نیست و مسیری طولانی‌تر به صورت $< V_1, V_2, V_3 >$ بین آن دو وجود دارد.

مثال چهارم : ضرب زنجیره‌ای ماتریس‌ها

همانطور که می‌دانید ضرب ماتریس‌ها خاصیت جابه‌جایی ندارد ولی خاصیت شرکت‌پذیری را دارد. همچنین ضرب دو ماتریس A×B به شرطی تعریف شده است که بعد وسط آنها یکسان باشد، یعنی تعداد ستونهای A با تعداد سطرهای B برابر باشد.

می‌توان اثبات کرد که ضرب دو ماتریس $A_{m\times n}\times B_{n\times k}$ در کل به m×n ×k عمل ضرب نیـاز دارد. حـال بـه مثال زیر توجه کنید :

مثال : ضرب ۳ ماتریس $A_{5\times10}B_{10\times20}C_{20\times4}$ را به چه ترتیبی انجام دهیم تا سریعتر اجرا شود؟

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

$(A_{5\times10}B_{10\times20}) \Rightarrow$ تعداد ضربها $= 5\times10\times20 = 1000$

$(AB)_{5\times20}C_{20\times4} \Rightarrow$ تعداد ضربها $= 5\times20\times4 = 400$

$\Rightarrow ((AB)C)$ تعداد ضربهای $= 1000+400=1400$

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

$(B_{10\times20}C_{20\times4}) \Rightarrow$ تعداد ضربها $= 10\times20\times4 = 800$

$A_{5\times10}(BC)_{10\times4} \Rightarrow$ تعداد ضربها $= 5\times10\times4 = 200$

$\Rightarrow (A(BC))$ تعداد ضربهای $= 800+200 = 1000$

پس A(BC) سریعتر از (AB)C انجام می‌پذیرد.

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

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

اثبات :

$(MN)_{a\times c}P_{c\times d}$ تعداد ضربهای $= abc+acd$

$M_{a\times b}(NP)_{b\times d}$ تعداد ضربهای $= bcd+abd$

$$\Rightarrow abc+acd<bcd+abd \xrightarrow{\text{طرفین تقسیم بر abcd}} \frac{1}{d}+\frac{1}{b}<\frac{1}{a}+\frac{1}{c}$$

به صورت سرانگشتی می‌توان گفت هنگام ضرب چند ماتریس در همدیگر ابتدا آنهایی را درهم ضرب می‌کنیم که بعد وسطی آنها بزرگتر و بعدهای کناری آنها کوچکتر باشد تا بدین ترتیب ماتریس حاصل کوچکتر شود.

توجه کنید که قاعده سرانگشتی فوق در همه موارد صحیح نیسـت و بـاید روش سیسـتماتیک و درسـت بـرای حالتی که تعداد زیادی ماتریس در هم ضرب می‌شوند، جهت یافتن ترتیب بهینه برای این عمل بیابیم.

یک روش ساده ولی بسیار کند آن است که همه ترتیب‌های ممکن، ضرب n ماتریس را در نظر گرفتـه و بـرای هر کدام تعداد ضرب‌های لازم را محاسبه کنیم، سپس ترتیبـی را انتخـاب کنیـم کـه کمـترین تعـداد ضـرب را بخواهد. ولی این الگوریتم ساده حداقل از مرتبه نمایی می‌باشد و اثبات آن به صورت زیر است :

اگر $T(n)$ تعداد ترتیب‌های متفاوت برای ضرب n ماترس $A_1, A_2, ..., A_n$ باشد، یک زیر مجموعه از این ترتیب‌ها حالتی است که در آنها A_1 آخرین ماتریسی است که در مجموعه ضرب می‌شود یعنی :

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

بنابراین تعداد ترتیب‌های ممکن برای ضرب ماتریس‌های A_2 تا A_n برابر T_{n-1} خواهد بود. حال به پرانتز $(A_2 A_3 ... A_n)$ توجه کنید. در این پرانتز نیز یک زیر مجموعه عبارت از همه ترتیب‌هایی است که در آ A_n آخرین ماتریسی است که ضرب می‌شود، پس تعداد ترتیب‌های مختلف در این زیر مجموعه ه $T(n-1)$ بوده و لذا داریم :

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

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

رابطه بازگشتی فوق حداقل از مرتبه 2^n بوده و لذا باید راه‌حل بهتری را پیدا کنیم. خوشبختانه اصل بهینگی برای مسأله بالا صادق است و لذا می‌توانیم از روش برنامه‌نویسی پویا مسأله را حل کنیم. برای اینکه متوجه شوید چرا اصل بهینگی برای این مسأله صادق است فرض کنید برای ضرب ماتریس‌های A_1 تا A_n اولی دسته‌بندی یا پرانتزگذاری باید بین ماتریس A_i و A_{i+1} صورت گیرد:

$$(A_1 A_2 ... A_i).(A_{i+1} A_{i+2} ... A_n)$$

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

فرض کنید می‌خواهیم چهار ماتریس زیر را در یکدیگر ضرب کنیم :

A_1	A_2	A_3	A_4
5×2	2×3	3×4	4×6
d_0 d_1	d_1 d_2	d_2 d_3	d_3 d_4

تعداد ستون‌های ماتریس A_k را با d_k نمایش می‌دهیم و از آنجا که تعداد سطرهای آن می‌بایست با تعد ستون‌های ماتریس قبلی برابر باشد، تعداد سطرهای ماتریس A برابر d_{k-1} است. ابتدا مسأله را از روش مستقیم که بسیار ناکارآمد است حل می‌کنیم. یعنی تمامی حالات ممکن ضرب را یافته و بهترین آن را (ک حداقل تعداد ضرب را دارد)، انتخاب می‌کنیم. ٤ ماتریس را به ٥ طریق ممکن مشابه زیر می‌توان در یکدیگ ضرب کرد :

$$(A_2 (A_3 A_4)) = 3 \times 4 \times 6 + 2 \times 3 \times 6 + 5 \times 2 \times 6 = 72 + 36 + 60 = 168$$
$$((A_2 A_3) A_4) = 2 \times 3 \times 4 + 2 \times 4 \times 6 + 5 \times 2 \times 6 = 24 + 48 + 60 = \boxed{132}$$
$$(A_1 A_2) (A_3 A_4) = 5 \times 2 \times 3 + 3 \times 4 \times 6 + 5 \times 3 \times 6 = 30 + 72 + 90 = 192$$
$$(A_1 A_2) A_3) A_4 = 5 \times 2 \times 3 + 5 \times 3 \times 4 + 5 \times 4 \times 6 = 30 + 60 + 120 = 210$$
$$(A_1 (A_2 A_3)) A_4 = 2 \times 3 \times 4 + 5 \times 2 \times 4 + 5 \times 4 \times 6 = 24 + 40 + 120 = 184$$

کمترین تعداد ضرب مربوط به حالت $A_1((A_2A_3)A_4)$ با 132 ضرب است. دقت کنید قانون ساده سرانگشتی که می‌گفت «ابتدا آنهایی را ضرب کن که بعد وسطشان بزرگتر است» در اینجا صادق نیست. حال یک روش حل پویا برای این مسأله پی‌ریزی می‌کنیم.

همانند مثال‌های قبلی برای حل این مسأله به صورت پویا، از یک ماتریس M[n][n] استفاده می‌کنیم که n تعداد ماتریس‌هایی است که می‌خواهیم در یکدیگر ضرب شوند. خانه‌های این ماتریس با مقادیر زیر پر می‌شوند :

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

ضرب ٤ ماتریس $A_1A_2A_3A_4$ به صورت بازگشتی یکی از ۳ حالت زیر است، یعنی اولین نقطه جداکننده می‌تواند بعد از A_1 یا بعد از A_2، یا بعد از A_3 باشد :

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

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

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

عبارت بالا یعنی «حداقل تعداد ضرب‌ها را برای ضرب $(A_2A_3A_4)$ به دست بیاور و با $d_0d_1d_4$ که تعداد ضرب لازم برای انجام محاسبه $(A_1)_{d_0\times d_1}(A_2A_3A_4)_{d_1\times d_4}$ است جمع کن. این حاصل حداقل ضرب‌ها را برای محاسبه $A_1(A_2A_3A_4)$ می‌دهد. به همین ترتیب :

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

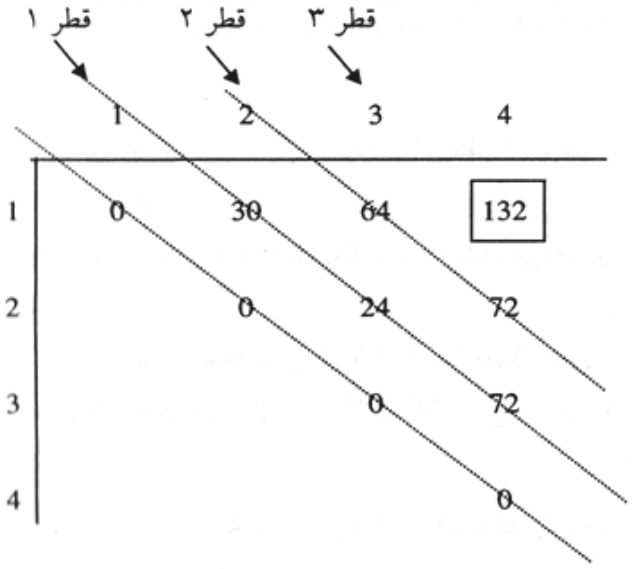
ج) $(A_1A_2A_3)_{d_0\times d_3}A_{4d_3\times d_4} \Rightarrow M[1][3] + M[4][4] + d_0d_3d_4$

توجه کنید برای $i = j$ عبارت $M[i][j] = 0$ است چرا که تعداد ضرب‌های لازم برای ایجاد یک ماتریس تنهای (A_i) را نشان می‌دهد که هیچ ضربی نیاز ندارد. پس از محاسبه عبارات الف و ب و ج جواب نهایی مینیمم بین آنها می‌باشد.

با توجه به مثال فوق می‌توان فرمول اصلی حل مسأله فوق را نوشت :

$M[i][j] = 0$	(اگر $i = j$) :
$M[i][j] = \min_{i\le k\le j-1}(M[i][k] + M[k+1][j] + d_{i-1}d_kd_j)$	(اگر $i < j$) :

تغییرات i و j از 1 تا n است و k نقطه‌ای را تعیین می‌کند که اولین پرانتز بسته برای جدا کردن ماتریس‌ها گذاشته می‌شود و در واقع نقطه شکاف را نشان می‌دهد. تغییرات k از 1 تا n-1 است. با توجه به فرمول اصلی فوق مرحله به مرحله به صورت دستی ماتریس M را محاسبه کنید تا روند الگوریتم را درک کنید. توجه کنید که ابتدا قطر اصلی ماتریس M را صفر می‌کنیم. سپس به سراغ قطر 1 (یعنی قطر بالای قطر اصلی) رفته و عناصر آن را به دست می‌آوریم. بعد به سراغ قطر ۲ و در آخر به سراغ قطر 3 می‌رویم. برای حالت n ماتریس، M یک ماتریس n × n بوده که می‌بایست (n - 1) قطر آن را محاسبه کنیم:



قطر ۱ :

$$M[1][2] = \min_{1\le k\le 1}(M[1][k]+M[k+1][2]+d_0d_kd_2)$$
$$= M[1][1]+M[2][2]+d_0d_1d_2$$
$$= 0+0+5\times2\times3 = 30$$
$$M[2][3] = \min_{2\le k\le 2}(M[2][2]+M[3][3]+d_1d_2d_3)$$
$$= 0+0+2\times3\times4 = 24$$
$$M[3][4] = 3\times4\times6 = 72$$

قطر ۲ :

$$M[1][3] = \min_{1\le k\le 2}(M[1][k]+M[k+1][3]+d_0d_kd_3)$$
$$\text{Min}((M[1][1]+M[2][3]+d_0d_1d_3),\ (M[1][2]+M[3][3]+d_0d_2d_3))$$
$$= \text{Min}((0+24+5\times2\times4),\ (30+0+5\times3\times4))$$
$$= \text{Min}(64, 90) = 64$$

$$M[2][4] = \min_{2\le k\le 3}(M[2][k]+M[k+1][4]+d_1d_kd_4)$$
$$= \text{Min}((M[2][2] + M[3][4] + d_1d_2d_4)\ ,\ (M[2][3] + M[4][4] + d_1d_3d_4))$$
$$= \text{Min}((0 + 72) + 2\times 3\times 6)\ ,\ (24 + 0 + 2\times 4\times 6))$$
$$= \text{Min}(108\ ,\ 72) = 72$$

قطر ۳ :

$$M[1][4]= \min_{1\le k\le 3}(M[1][k]+M[k+1][4]+d_0d_kd_4)$$
$$=\text{Min}((M[1][1]+M[2][4]+d_0d_1d_4),(M[1][2] + M[3][4]+d_0d_2d_4),(M[1][3] + M[4][4] +d_0d_3d_4))$$
$$= \text{Min}((0 + 72 + 5\times 2\times 6)\ ,\ (30 + 72 + 5\times 3\times 6)\ ,\ (64 + 0 + 5\times 4\times 6))$$
$$= \text{Min}(132\ ,\ 192\ ,\ 184) = \underline{\mathbf{132}}$$

توجه کنید برای قطر 1 ، مینیمم یک عبارت، برای قطر ۲، مینیمم بین دو عبارت و برای قطر ۳، مینیمم بین سه عبارت را محاسبه کردیم.

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

برای یافتن ترتیب ضرب بهینه یعنی $A_1((A_2A_3)A_4)$ کافی است کنار ماتریس M یک ماتریس P را نیز بسازیم. برای مثال فوق کار را از عنصر $M[1][4]$ شروع می‌کنیم. در هنگام محاسبه این عنصر داشتیم :

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

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

حال به سراغ $M[2][4]$ می‌رویم که ببینیم برای ضرب A_2 تا A_4 عملیات در کجا باید شکسته شود :

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

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

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

مثال : می‌خواهیم ٦ ماتریس را در هم ضرب کنیم $A_1A_2A_3A_4A_5A_6$ ماتریس P آنها طبق الگوریتم فوق به صورت زیر درآمده است. ترتیب بهینه را برای حداقل تعداد ضرب‌ها چاپ کنید.

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

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

$$(A_1)(A_2A_3A_4A_5A_6)$$

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

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

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

$$(A_1)(((A_2A_3A_4)A_5)A_6)$$

از آنجا که $P[2][4] = 3$ است یعنی :

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

کدنویسی الگوریتم فوق را به عنوان تمرین بر عهده دانشجویان قرار می‌دهیم. در کتاب نیپولیتان برنامه این الگوریتم آورده شده است.

ورودی این الگوریتم تعداد ماترس‌ها (n) و آرایه d است که ابعاد ماتریس‌ها را نشان می‌دهد (یعنی مقادیر $d_n,...,d_2,d_1,d_0$). توجه کنید به محتویات خود ماتریس‌ها نیازی نیست. خروجی این الگوریتم ماتریس M و P و در نهایت چاپ ترتیب بهینه ضرب است.

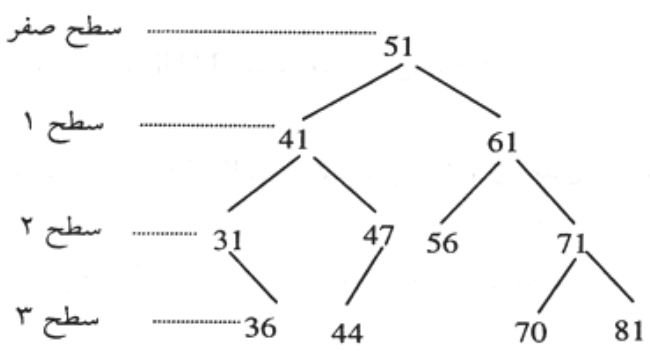
با توجه به فرمول اصلی این الگوریتم، از آنجا که می‌بایست خانه‌های ماتریس $M_{n\times n}$ محاسبه شوند و برای هر خانه $M[i][j]$ نیاز به یک مینیمم‌گیری از مرتبه $\theta(n)$ داریم لذا روشن است که مرتبه اجرایی این الگوریتم $\theta(n^3)$ می‌باشد.

مرتبه اجرایی الگوریتم ضرب زنجیره‌ای ماتریس‌ها با روش پویا $= \theta(n^3)$

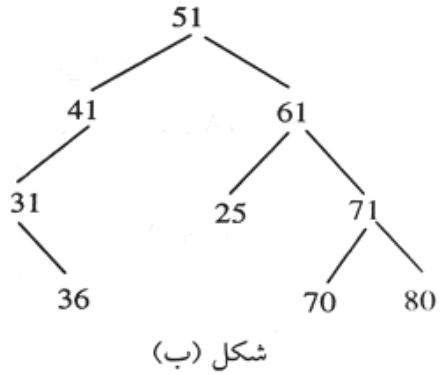
تذکر : ضرب ٤ ماتریس در یکدیگر ٥ حالت مختلف دارد که نسبتاً کم است. لذا در حل مسائل بهتر است اگر ٤ ماتریس یا کمتر داریم از همان روش معمولی رفته و تمام حالات ممکن را محاسبه و سپس حداقل آنها را به دست آوریم.

مثال پنجم : درخت‌های جستجوی دودویی بهینه

همان‌طور که در درس ساختمان داده‌ها خوانده‌اید، درخت جستجوی دودویی یا BST (Binary Search Tree) یک درخت دودویی است که مقدار هر گره بزرگتر از هر مقدار در زیر درخت چپ و کوچکتر از هر مقدار در زیر درخت راست آن می‌باشد. هر گره دارای یک کلید است و عموماً دو گره نباید دارای کلید یکسان باشند (یعنی عموماً کلیدها منحصر به فرد هستند). مثلاً شکل (الف) زیر یک درخت BST است؛ ولی شکل (ب) BST نیست؛ چرا که در زیر درخت سمت راست گره 51 عدد 25 از آن کوچکتر است :



شکل (الف)



شکل (ب)

فرض می‌کنیم ریشه در سطح صفر قرار دارد و عمق هر گره را برابر سطح آن گره می‌گیریم. مثلاً عمق گره 31 در شکل الف برابر 2 می‌باشد. عمق یک درخت برابر حداکثر عمق گره‌های آن است، مثلاً عمق درخت‌های شکل فوق برابر 3 می‌باشد. درخت متوازن درختی است که عمق دو زیر درخت از هر گره آن حداکثر یک واحد با یکدیگر اختلاف داشته باشد. مثلاً شکل درخت (الف) فوق متوازن است ولی شکل (ب) متوازن نیست چرا که عمق زیر درخت سمت چپ 41 برابر 2 و عمق زیر درخت راست آن صفر است.

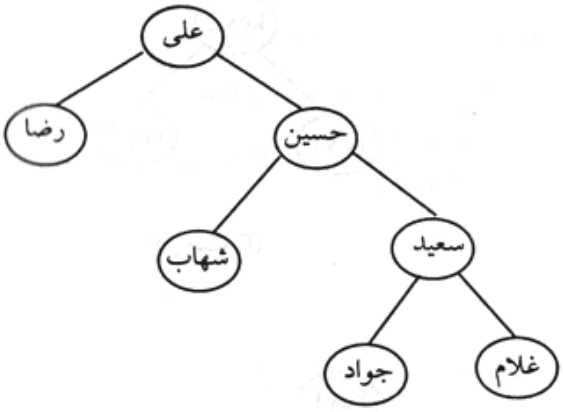
الگوریتم جستجو در درخت BST

فرض کنید دنبال کلیدی هستیم که می‌دانیم حتماً در درخت هست. جهت جستجو همواره از ریشه شروع می‌کنیم. اگر کلید مورد جستجو از کلید گره بزرگتر بود به سمت راست و اگر کوچکتر بود به سمت چپ می‌رویم این عمل را آن قدر تکرار می‌کنیم تا بالاخره داده مورد نظر پیدا شود.

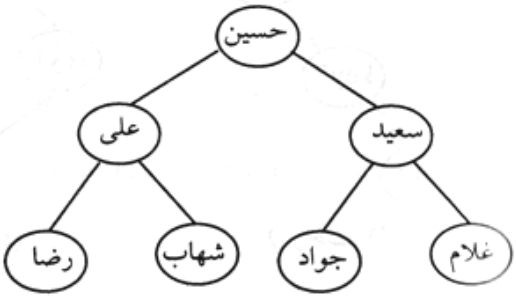
بدیهی است تعداد مقایسه‌ها برای یافتن یک کلید key برابر عبارت زیر است که منظور از depth(key) عمق آن کلید است :

depth(key) + 1

مثلاً در شکل‌های زیر :



(الف)



(ب)

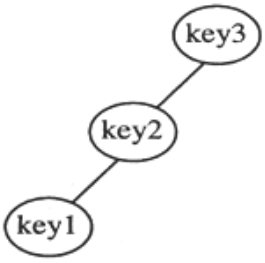
در شکل (الف) برای یافتن سعید به ۳ مقایسه و در شکل (ب) برای یافتن سعید به ۲ مقایسه نیاز داریم. هدف اصلی ما در اینجا سازماندهی کلیدها در یک درخت BST است به گونه‌ای که زمان میانگین برای یافتن همه کلیدها حداقل باشد و به چنین درختی، درخت BST بهینه (Optimal Binary Search Tree) می‌گوئیم.

بدیهی است اگر احتمال رخ دادن همه کلیدها با یکدیگر برابر باشد درخت متعادلی مثل شکل (ب) فوق درخت بهینه خواهد بود ولی اگر احتمال‌ها برابر نباشند در یک ایده کلی بهتر است کلیدهایی با احتمال بیشتر به ریشه نزدیک‌تر بوده و کلیدهایی با احتمال کمتر از ریشه دورتر باشند. بحث اصلی ما در این قسمت بر سر حالتی است که احتمال‌ها یکسان نیست.

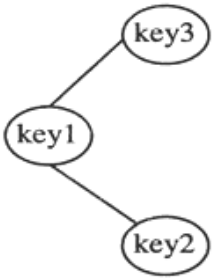
فرض کنید می‌خواهیم یکی از n کلید $key_1, key_2, ..., key_n$ را در یک درخت BST پیدا کنیم. P_i احتمال مساوی بودن کلید key_i با کلید مورد جستجو و C_i تعداد مقایسه‌های لازم برای یافتن key_i در آن درخت است. در این صورت بدیهی است که داریم :

$$\text{زمان میانگین جستجو در یک درخت BST} = \sum_{i=1}^{n} C_i P_i$$

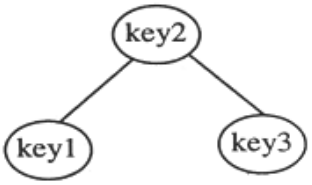
مثال : سه کلید $Key1 < key2 < key3$ با احتمال‌های $P_1 = 0.7$ ، $P_2 = 0.2$ و $P_3 = 0.1$ در ۵ درخت مختلف BST ذخیره شده‌اند. زمان جستجوی میانگین را برای هر یک از درخت‌ها محاسبه کنید :



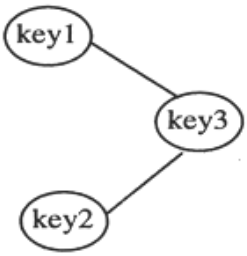
(درخت ۱)



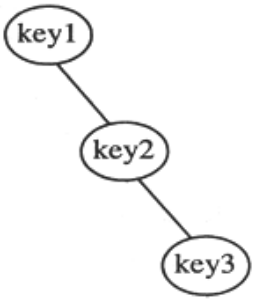
(درخت ۲)



(درخت ۳)



(درخت ٤)



(درخت ٥)

حل :

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

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

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

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

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

همان‌طور که مشاهده می‌شود درخت ٥، حداقل تعداد مقایسه را نیاز داشته و لذا از همه بهتر است. هدف ما این است که با توجه به احتمالات داده شده‌ی هر کلید شکل بهینه فوق را به دست آوریم. توجه کنید برای ۳ کلید فقط همان ٥ حالت شکل فوق امکان‌پذیر است.

یک راه ساده ولی بسیار کند مشابه فوق است یعنی تمامی حالات ممکن را ترسیم و زمان جستجوی میانگین را برای همه آنها محاسبه کنیم و در آخر حداقل آنها را انتخاب نمائیم. ولی تعداد این درخت‌ها مانند اکثر مسائل بهینه‌سازی حداقل از مرتبه نمایی است. برای اثبات فقط درخت‌هایی با عمق $n - 1$ را در نظر می‌گیریم که زیر مجموعه‌ای از کل حالات مورد نظر است یعنی هر گره فقط یک بچه داشته باشد. در این وضعیت گره فرزند

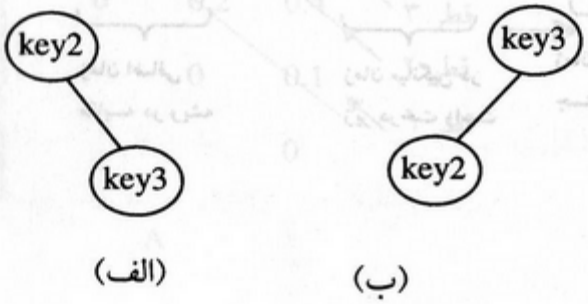
می‌تواند چپ یا سمت راست پدر خود قرار گیرد یعنی برای قرار دهی هر گره (به جز ریشه) 2 حالت انتخاب وجود دارد و لذا تعداد کل انتخاب‌ها 2^{n-1} می‌باشد یعنی تعداد درخت‌های جستجوی متفاوت با عمق n-1 برابر 2^{n-1} است که بسیار زیاد است. پس باید الگوریتمی کارآمد با برنامه‌نویسی پویا بنا کنیم.

تذکر : در کتاب هورویتز اثبات شده که با n کلید مجزا می‌توان $\frac{1}{n+1}\binom{2n}{2}$ درخت BST مجزا ساخت مثلاً برابر $n = 3$، به تعداد ۵ درخت BST می‌توان ساخت :

$$\frac{1}{3+1}\binom{6}{3}=5$$

این الگوریتم شباهت زیادی به الگوریتم ضرب زنجیری ماتریس‌ها دارد. اصل بهینگی برای این مسأله نیز صادق است. چرا که مثلاً در شکل قبلی که درخت شماره ۵ بهینه بود، زیر درخت مربوط به ریشه نیز می‌بایست بهینه باشد.

شکل‌های ممکن برای دو کلید key 2 < key 3 با احتمالات $P_3 = 0.1$ و $P_2 = 0.2$ به دو صورت زیر است:



(الف) (ب)

(الف) $\Rightarrow 1 \times 0.2 + 2 \times 0.1 = 0.4$

(ب) $\Rightarrow 2 \times 0.2 + 1 \times 0.1 = 0.5$

پس شکل (الف) شکل بهینه است.

چون اصل بهینگی در اینجا صادق است یک الگوریتم بازگشتی برای حل مسأله بنا می‌کنیم.

مشابه مسأله ضرب ماتریس‌ها از یک ماتریس A استفاده می‌کنیم که مفهوم عناصر آن به صورت زیر است:

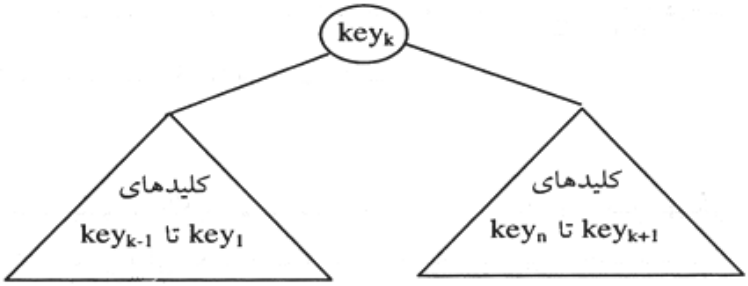
A[i] [j] =	مینیمم میانگین زمان جستجو برای یک درخت BST
$1 \le i \le j \le n$, $key_i \le key_j$	با کلیدهای key_i تا key_j که

بدیهی است که $A[i][i] = P_i$ است چرا که در این حالت درخت BST فقط یک کلید key_i به صورت روبه‌رو دارد :

که میانگین زمان جستجوی آن برابر است با :

$A[i][i] = 1 \times P_i = P_i$

بنابر اصل بهینگی اگر درخت بهینه مورد نظر شامل ریشه key_k باشد انگاه دو زیر درخت چپ و راست این ریشه نیز باید BST بهینه باشند :



اگر درخت فوق، درخت بهینه BST باشد آنگاه A[1][n] با فرض آنکه key_k ریشه باشد برابر خواهد بود با:

$$\underbrace{A[1][k-1]}_{\text{زمان میانگین در زیر درخت چپ}} + \underbrace{P_1 + \ldots + P_{k-1}}_{\text{زمان اضافی مقایسه در ریشه}} + \underbrace{P_k}_{\text{زمان میانگین جستجو برای ریشه}} + \underbrace{A[k+1][n]}_{\text{زمان میانگین در زیر درخت راست}} + \underbrace{P_{k+1} + \ldots P_n}_{\text{زمان اضافی مقایسه در ریشه}}$$

$$= A[1][k-1] + A[k+1][n] + \sum_{m=1}^{n} P_m$$

توجه کنید فرمول فوق برای وقتی است که فرض کرده‌ایم key_k ریشه باشد ولی هر یک از key ها (key_1 تا key_n) می‌توانند در ریشه باشند که آنگاه جواب نهایی مینیمم آنها خواهد بود، لذا :

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

در حالت کلی برای هر key_i تا key_j داریم :

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

$$A[i][i] = P_i \qquad (i = j)$$

فرمول اصلی فوق برای ساخت ماتریس A استفاده می‌شود. جهت سادگی ماتریس A را به صورت $(n+1)\times(n+1)$ در نظر می‌گیریم که قطر اصلی آن صفر است. همگام با ساختن ماتریس A ، ماتریس کمکی R

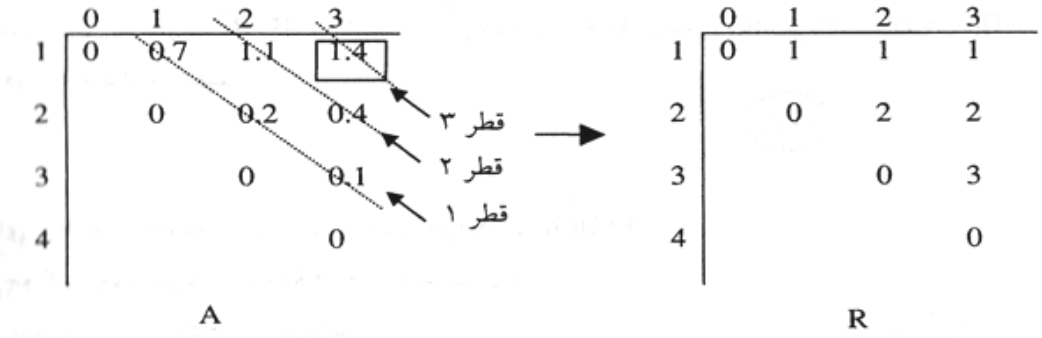
را نیز کنار آن پدید می‌آوریم. محتوای R[i][j] نمایانگر اندیس کلیدی است که به عنوان ریشه انتخاب می‌شود. مثلاً R[2][4] اندیس کلید ریشه درخت BST بهینه‌ای است که از کلیدهای key_2 ، key_3 و key_4 تشکیل شده است.

با یک مثال نحوه محاسبه ماتریس‌های A و R را نشان می‌دهیم. این ماتریس‌ها که (n+1) × (n+1) هستند اندیس سطرهایشان از 1 تا n + 1 و اندیس ستون‌هایشان از 0 تا n است که n تعداد کلیدهاست. قطر اصلی هر دو ماتریس صفر است. این ماتریس‌ها را مشابه بحث ضرب ماتریس‌ها به صورت قطر به قطر پر می‌کنیم :

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

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

حل :



قطر ۱ : از آنجا که در قطر 1 داریم $A[i][i] = P_i$ به راحتی احتمالات داده شده را در قطر مربوطه قرار می‌دهیم. به همین ترتیب R[i][i]=i می‌باشد چرا که در درخت BST بهینه از کلید key_i تا key_i بدیهی است که اندیس ریشه همان i است (چرا که این درخت، فقط یک گره دارد)

قطر ۲ :

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

پس در خانه R[1][2] عدد k = 1 را قرار می‌دهیم.

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

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

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

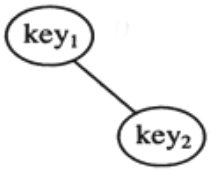
قطر ۳ :

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

و در خانه $R[1][3]$ عدد $k = 1$ را قرار می‌دهیم.

پس درخت BST بهینه با کلیدهای key_1 تا key_3 در کل به زمان جستجوی میانگین 1.4 نیاز دارد. برای ترسیم شکل به ماتریس R نگاه می‌کنیم. برای key_1 تا key_3 ریشه key_1 می‌باشد $(R[1][3] = 1)$ پس آن را در ریشه قرار می‌دهیم :

برای key_2 تا key_3 ریشه key_2 است چرا که $R[2][3]=2$ می‌باشد :
توجه کنید چون می‌دانیم $key_2 > key_1$ است آن را
در سمت راست key_1 قرار داده‌ایم.



در آخر هم key_3 را در سمت راست key_2 رسم می‌کنیم.

پیاده‌سازی الگوریتم فوق را به عنوان تمرین برعهده دانشجویان قرار می‌دهیم. البته کدهای آن را به زبان C می‌توانید در کتاب نیپولیتان مشاهده کنید.

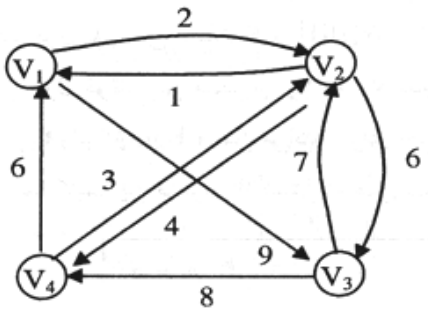
مشابه الگوریتم ضرب ماتریس‌ها این الگوریتم نیز نیاز به سه حلقه تودرتو دارد و لذا مرتبه اجرایی آن $\theta(n^3)$ است.

مثال ششم : فروشنده دوره‌گرد

فروشنده‌ای می‌خواهد از تعدادی شهر که توسط جاده‌هایی به یکدیگر متصل هستند، عبور کند و در آخر به شهر اولیه خود برگردد. هدف یافتن کوتاه‌ترین مسیر برای فروشنده است به نحوی که از تمام شهرها دقیقاً یک بار عبور کند و این مسأله استاندارد فروشنده دوره‌گرد است.

در یک گراف جهت‌دار، یک **تور** ، که به آن **مدار هامیلتون** نیز گفته می‌شود، عبارت از مسیری از یک رأس به خودش است که از تمام رئوس دیگر دقیقاً یک بار عبور کند. منظور از یک تور بهینه در گراف جهت‌دار وزن‌دار، مسیری از این نوع است که طول آن حداقل می‌باشد. بنابراین مسأله فروشنده دوره‌گرد در واقع پیدا کردن یک تور بهینه برای یک گراف جهت‌دار موزون است. توجه کنید که ممکن است گرافی اصلاً تور نداشته باشد. همچنین طول تور بهینه وابسته به انتخاب رأس آغازین نیست.

مثال : در گراف شکل زیر ۳ تور با طول‌های تعیین شده عبارتند از :



طول $[v_1, v_2, v_3, v_4, v_1] = 22$

طول $[v_1, v_3, v_2, v_4, v_1] = 26$

طول $[v_1, v_3, v_4, v_2, v_1] = 21$

در شکل فوق مسیر $[v_1, v_3, v_4, v_2, v_1]$ ، تور بهینه است.

یک راه ساده (ولی خیلی کند) برای حل مسأله فروشنده دوره‌گرد آن است که همه تورهای ممکن را به دست آورده و بهینه آنها را بیابیم. ولی این الگوریتم از مرتبه $O(n!)$ خواهد بود. فرض کنید از هر گره به تمام گره‌های دیگر یال (مسیر مستقیم) وجود داشته باشد. حال اگر از یک گره دلخواه شروع کنیم، گره دوم می‌تواند $(n-1)$ حالت داشته باشد، وقتی بر روی گره دوم قرار گرفتیم، گره سوم می‌تواند هر یک از $(n-2)$ رأس باقی‌مانده باشد و به همین ترتیب جلو می‌رویم تا اینکه گره n ام فقط یک رأس باقی مانده می‌تواند باشد. لذا تعداد کل تورها برابر است با :

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

> حل مسأله فروشنده دوره‌گرد با روش بررسی همه تورهای ممکن از مرتبه روبه‌رو است: $(n-1)!$

از آنجا که اصل بهینگی برای این مسأله نیز صادق است (چرا؟) برای حل آن می‌توانیم از تکنیک برنامه‌نویسی پویا استفاده کنیم.

برای حل مسأله فوق از تعاریف زیر استفاده می‌کنیم :

V = مجموعه همه رئوس گراف ، A = زیرمجموعه‌ای از رئوس گراف

$D[V_i][A]$ = طول کوتاه‌ترین مسیر از V_i به V_1 که از هر رأس زیر مجموعه A دقیقاً یک بار می‌گذرد

ی نشان‌دادن مجموعه‌ای از رئوس از علامت آکولاد و برای نمایش یک مسیر از علامت [ ] استفاده می‌کنیم.

ل : در گراف مثال قبلی اگر $A = \{V_3\}$ باشد، داریم : (منظور از len طول است)

$$D[v_2][A] = len[v_2, v_3, v_1] = \infty$$

گر $A = \{v_3, v_4\}$ باشد، داریم :

$$D_2[v_2][A] = D_2[v_2][\{v_3, v_4\}] = Min(len[v_2, v_3, v_4, v_1], len[v_2, v_4, v_3, v_1])$$
$$= Min(20, \infty) = 20$$

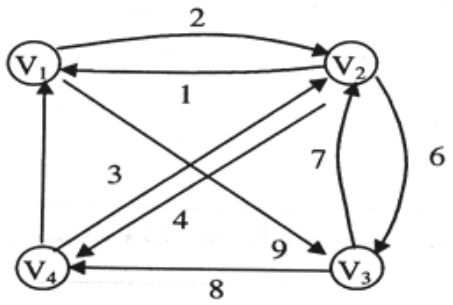
وجه به تعریف فوق بدیهی است که $D[v_i][\phi] = W[i][1]$ که W همان ماتریس هم‌جواری است.

$D[v_i][\phi]$ یعنی طول کوتاه‌ترین مسیر از v_i به v_1 بدون هیچ واسطه‌ای که برابر همان $D[v_i][1]$ یعنی وزن ی است که رأس i را به رأس 1 وصل می‌کند. برای حل مسأله فروشنده دوره‌گرد به روش پویا باید یک مول بازگشتی پیدا کنیم. این فرمول بازگشتی به صورت زیر است (چرا؟) :

$$D[v_i][A] = \min_{v_j \in A}(W[i][j] + D[v_j][A - \{v_j\}]) \quad \text{اگر } A \neq \phi$$
$$D[v_i][\phi] = W[i][1] \quad \text{اگر } A = \phi$$

مثال زیر مرحله به مرحله یافتن مقادیر فرمول فوق را تمرین کنید. در فرمول فوق نقطه آغاز تور را گره 1 رفته‌ایم که در حل مسأله این نقطه شروع اثری ندارد.

ال : در گراف جهت‌دار موزون زیر با ماتریس هم‌جواری W داده شده، طول تور بهینه را به دست آورید :



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

ل : قدم اول : مجموعه A را تهی در نظر می‌گیریم :

$$D[v_2][\phi] = W[2][1] = 1 \quad , \quad D[v_3][\phi] = W[3][1] = \infty$$
$$D[v_4][\phi] = W[4][1] = 6$$

دم دوم : مجموعه A را یک عضوی می‌گیریم :

$$D[v_3][\{v_2\}] = \min_{v_j \in \{v_2\}}(W[3][j] + D[v_j][\{v_2\} - \{v_j\}]) = W[3][2] + D[v_2][\phi] = 7 + 1 = 8$$

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

قدم سوم : مجموعه A را دو عضوی می‌گیریم :

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

قدم چهارم : در آخر مجموعه A را سه عضوی می‌گیریم :

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

پس طول تور بهینه برابر 21 می‌باشد.

نحوه پیاده‌سازی الگوریتم فوق در یک زبان برنامه‌نویسی به عنوان تمرین بر عهده دانشجویان گذاشته می‌شود. برنامه آن را به زبان C++ می‌توانید از کتاب نیپولیتان مطالعه کنید.

می‌توان اثبات کرد :

> مرتبه الگوریتم فروشنده دوره‌گرد با برنامه‌نویسی پویا $\in \theta(n^2 2^n)$

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

روش یافتن کلیه تورها $= (20-1)!\ \mu\text{sec} = 3857$ سال

روش برنامه‌نویسی پویا $\approx 20^2 \times 2^{20} \approx 7$ دقیقه

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

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

نکته ۲ : تاکنون کسی نتوانسته است برای مسأله فروشنده دوره‌گرد الگوریتمی بهتر از حالت نمایی پیدا کند، از طرف دیگر کسی هم عدم امکان یافتن چنین الگوریتمی را اثبات نکرده است.

سری کاتالان

سری معروف کاتالان به صورت زیر بوده و جواب بسیاری از مسائل شمارشی می‌باشد :

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

برخی از مقادیر این سری عبارتند از :

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

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

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

حال چند مسأله استاندارد را مطرح می‌کنیم که تعداد حالات مختلف آنها با سری فوق محاسبه می‌شود.

۱ـ مسأله ضرب ماتریس‌ها : در قسمت‌های قبلی گفتیم تعداد حالاتی که می‌توان n ماتریس را درهم ضرب کرد حداقل از مرتبه نمایی است. ولی می‌خواهیم تعداد دقیق این حالات را به دست آوریم. فرض کنید در اولین مرحله ضرب ماتریس‌ها را در نقطه i به دو دسته تقسیم کرده‌ایم یعنی : $(A_1A_2...A_i)(A_{i+1}A_{i+2}...A_n)$. اگر $T(i)$ تعداد حالات ممکن برای ضرب پرانتز سمت چپ باشد، پس $T(n-i)$ تعداد حالات ممکن برای ضرب پرانتز سمت راست بوده و در کل داریم :

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

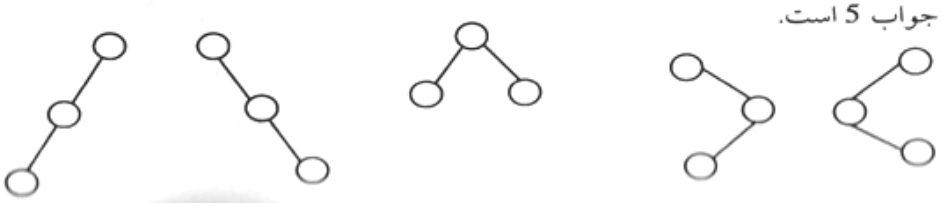
که حل رابطه فوق به فرمول $T(n)=\frac{1}{n}\binom{2n-2}{n-1}$ می‌رسد.

برای ضرب ٤ ماتریس، ٥ حالت زیر امکان‌پذیر است :

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

۲ـ مسأله تعداد درخت‌های دودویی : می‌خواهیم بدانیم با n گره چند فرم درخت دودویی می‌توان ساخت.

مثلاً برای $n=3$ جواب 5 است.



حل مسأله فوق معادل جمله $n+1$ از سری کاتالان است :

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

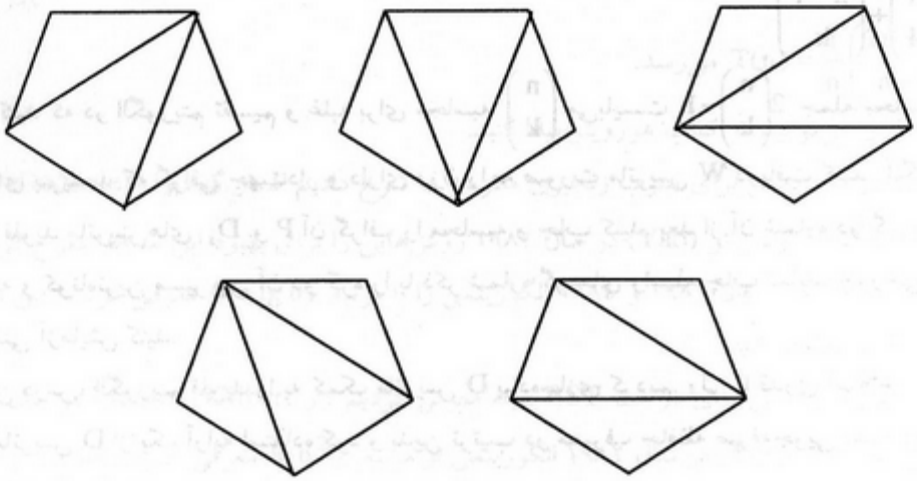
۳ـ مسأله پشته : می‌خواهیم بدانیم با n عدد ورودی که به ترتیب وارد یک پشته می‌شوند، چند حالت خروجی می‌توان داشت.

مثلاً برای اعداد ۱ و ۲ و ۳ $(n=3)$ که به ترتیب از راست به چپ وارد پشته‌ای می‌شوند ۵ حالت خروجی زیر امکان‌پذیر است (داخل پرانتزها را از چپ به راست بخوانید) :

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

پس حل مسأله فوق مشابه تعداد درخت‌های دودویی، معادل جمله $n+1$ از سری کاتالان است.

٤ـ مسأله مثلث‌بندی چند ضلعی محدب : می‌خواهیم بدانیم به چند طریق می‌توان قطرهای نامتقاطع یک چندضلعی محدب را رسم کرد و آن را به مثلث‌های مختلف تقسیم نمود. در حالت کلی یک n ضلعی همواره با n-3 قطر به $n-2$ مثلث افراز می‌شود. مثلاً برای $n=5$ به ۵ صورت مختلف می‌توان قطرها را بر طبق جملات فوق رسم کرد :



پس تعداد حالات مسأله فوق معادل (n-1) امین عدد کاتالان است :

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