فصل چهارم

برنامه‌نویسی پویا

مقدمه

روش تقسیم و غلبه یک روش کل به جز یا بالا به پایین (top – down) بود. ولی روش برنامه نویسی پویا یک روش جزء به کل یا پایین به بالا (bottom- up) می‌باشد. در تکنیک تقسیم و غلبه از مسئله اصلی شروع می‌کردیم و سپس آن را به اجزای کوچکتر تقسیم می‌کردیم و تقسیم را تا زمانی ادامه می‌دادیم که به مسائل کوچک قابل حل برسیم و سپس با حل آنها به تدریج مسئله اصلی حل می‌شد.

در برنامه نویسی پویا، ابتدا مسائل کوچک حل می‌شوند و در یک مکان ذخیره می‌شوند و سپس به تدریج به حل مسئله اصلی می‌رسیم. در این الگوریتم‌ها یک مسئله کوچک فقط یک بار محاسبه می‌شود.

شباهت تقسیم و غلبه و برنامه نویسی پویا این است که هر دو نیاز به یک رابطه بازگشتی دارند و تفاوت آن‌ها این است که تقسیم و غلبه مسئله را از بالا به پایین حل می‌کند، در صورتی که برنامه نویسی پویا آن را از پایین به بالا حل می‌کند.

مثال: رابطه بازگشتی $T(n) = T(n-1) + 2$ را با فرض $T(1)=1$ را حل کنید.

حل: روش اول (بالا به پایین):

$$T(n) = T(n-1) + 2 = \big(T(n-2) + 2\big) + 2$$
$$= T(n-2) + 2 + 2 + 2 = T(n-2) + 3\times 2 = \ldots =$$
$$= T\big(n-(n-1)\big) + (n-1)\times 2 = T(1) + (n-1)\times 2 = 2n-1$$
$$T(1) = 1$$

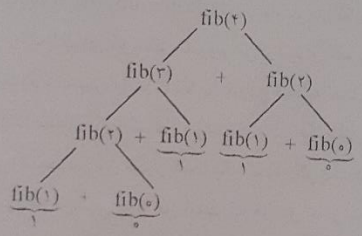
روش دوم (پایین به بالا):

$$T(2) = T(1) + 2 = 1 + 2 = 3$$
$$T(3) = T(2) + 2 = 3 + 2 = 5$$
$$T(4) = T(3) + 2 = 5 + 2 = 7$$
$$\vdots$$
$$T(n) = 2n - 1$$

مثال: اگر بخواهیم جمله n ام سری فیبوناچی را با روش تقسیم و غلبه بیابیم در این صورت الگوریتم آن به صورت زیر است:

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

با توجه به الگوریتم فوق اگر بخواهیم fib(۴) را بیابیم به شکل زیر عمل می‌کنیم:



همانطور که ملاحظه می‌شود، محاسبات تکراری در درخت وجود دارد و مثلاً fib(۲)، ۲ بار محاسبه می‌شود. مرتبه اجرایی این الگوریتم بیش از $۲^{\frac{n}{۲}}$ می‌باشد.

اگر بخواهیم جمله nام سری فیبوناچی را با روش برنامه نویسی پویا بیابیم، الگوریتم زیر را ارائه می‌دهیم:

```
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];
}
```

البته می‌توان جمله nام سری فیبوناچی را به صورت پویا و با مصرف حافظه کمتر با توجه به الگوریتم زیر بدست آورد:

```
int fib (int n)
  {
    int f۱ , f۲ , f;
    f۱ = ۰; f۲ = ۱;
    if (n ==۰) return f۱;
    else if (n ==۱) return f۲;
    for (i = ۲;i <= n;i ++)
    {
      f = f۱ + f۲;
      f۱ = f۲;
      f۲ = f;
    }
      return f;
}
```

با توجه به دو الگوریتم نوشته شده به شیوه پویا مشاهده می‌شود که مرتبه اجرایی $\theta(n)$ می‌باشد.

دلیل اصلی اینکه فیبوناچی بازگشتی دارای مرتبه نمایی و فیبوناچی غیر بازگشتی دارای مرتبه خطی است این است که در فیبوناچی بازگشتی زیر مسائل به هم ارتباط دارند. مثلاً fib(3) و fib (4) هر دو دارای زیر مسائل مشترک هستند.

تمرین: روشی وجود دارد که می‌تواند جمله n ام فیبوناچی را با مرتبه $O(Lgn)$ بیابد. آن روش را ارائه دهید!

اصولاً روش تقسیم و غلبه برای الگوریتم‌هایی مناسب است که زیر مسائل آنها با هم اشتراک نداشته باشند مثل مرتب سازی ادغامی که نمونه‌های تقسیم شده ارتباطی به هم ندارند.

به طور کلی می‌توان گفت مسائلی که به روش برنامه سازی پویا حل می‌شوند دارای ویژگی‌های زیر هستند:

۱) بهینه‌سازی. در اکثر الگوریتم‌های برنامه سازی پویا، فقط بدست آوردن جواب کافی نیست، بلکه جواب باید بهینه نیز باشد.

۲) مسائل را از پایین‌ترین سطح به طرف بالاترین سطح حل می‌کنیم (پایین به بالا)

۳) برخلاف روش تقسیم و حل که برای هر مساله در سطح L تنها از مسائل سطح L-۱ استفاده می‌کنیم، در روش برنامه سازی پویا، برای حل هر مساله در سطح L می‌توانیم از کلیه مسائل سطوح پایین‌تر که لازم باشد استفاده کنیم.

۴) در هر سطح کلیه مسائل موجود آن سطح حل می‌گردند و نتایج نگهداری می‌شود.

تذکر: در کلیه مسائل بهینه سازی که با روش برنامه سازی پویا قابل حل هستند باید اصل بهینگی (Principle of optimality) برقرار باشد. اصل بهینگی در صورتی برقرار است که زیر مسائل یک مسائل بهینه خود بهینه باشند. به عنوان مثال اگر کوتاهترین مسیر از a به b حتماً از c می‌گذرد آنگاه باید مسیر از a به c و از c به b خود کوتاهترین باشند.

ولی در مسئله یافتن طولانی‌ترین مسیر بین دو راس اصل بهینگی صادق نیست و نمی‌توان آن را از طریق برنامه‌ریزی پویا حل کرد. به عنوان مثال در گراف زیر، طولانی‌ترین مسیر ساده (مسیر بهینه) از a به d مسیر <a , c, b, d> می‌باشد ولی زیر مسیر <a , c> یک مسیر بهینه (طولانی‌ترین) از a به c نیست و مسیر طولانی‌تر به صورت <a, b , c> می‌باشد.



۱- ضرب زنجیره‌ای ماتریس‌ها

همانطور که می‌دانیم ضرب ماتریس‌ها دارای خاصیت شرکت پذیری می‌باشد ولی جابجایی ندارد. علاوه بر این دو ماتریس A و B فقط به شرطی به صورت A×B قابل ضرب هستند که بعد وسط آنها یکسان باشد یعنی تعداد ستون‌های A با تعداد سطرهای B مساوی باشد. علاوه بر این می‌دانیم که ضرب $A_{m\times n} \times B_{n\times p}$ نیاز به $m\times n\times p$ عمل ضرب دارد.

مثال: ضرب سه ماتریس $A_{۲\times۵}$ و $B_{۵\times۲}$ و $C_{۲\times۲}$ به صورت ABC حداقل چند عمل ضرب نیاز دارد؟

پاسخ: سه ماتریس را می‌توان به ۲ صورت (AB)C و A(BC) در هم ضرب کرد. بنابراین هر دو حالت را بررسی می‌کنیم:

$(A_{۲\times۵}\times B_{۵\times۲})\times C_{۲\times۲} \Rightarrow$ تعداد ضرب $= ۲\times۵\times۲ + ۲\times۲\times۲ = ۱۴۷$

$A_{۲\times۵}\times(B_{۵\times۲}\times C_{۲\times۲}) \Rightarrow$ تعداد ضرب $= ۵\times۲\times۲ + ۲\times۵\times۲ = ۱۰۰$

بنابراین جواب ۱۰۰ می‌باشد.

مثال: چه رابطه‌ای بین w و x و y و z برقرار باشد تا ضرب ۳ ماتریس $A_{w\times x}$ و $B_{x\times y}$ و $C_{y\times z}$ به صورت (AB)C از A(BC) بهتر باشد یعنی تعداد عملیات ضرب کمتری داشته باشد.

پاسخ:

$A_{wxx}\times B_{xxy} \Rightarrow$ تعداد ضرب $= wxy$

$(AB)_{wxy}\times C_{yxz} \Rightarrow$ تعداد ضرب $= wyz$

$(AB)C \Rightarrow$ تعداد ضرب $= wxy+wyz$

به همین ترتیب تعداد ضرب A(BC) برابر xyz+wxz می‌باشد بنابراین

$wxy+wyz<xyz+wxz$

طرفین رابطه را به wxyz تقسیم می‌کنیم:

$$\frac{1}{z}+\frac{1}{x}<\frac{1}{w}+\frac{1}{y}$$

حال می‌خواهیم روش ضرب بهینه n ماتریس را پیدا کنیم:

$M = M_1 \times M_2 \times ... \times M_n$

می دانیم تعداد حالاتی که می‌توان $n+1$ ماتریس را درهم ضرب کرد برابر عدد کاتالان یعنی $\frac{1}{n+1}\binom{2n}{n}$ می‌باشد. اگر بخواهیم تمام حالات ضرب را آزمایش کنیم تا به ضرب بهینه برسیم بسیار زمانگیر است و می‌توان ثابت کرد تابع رشد کاتالان به صورت $\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$ می‌باشد که بیشتر از نمایی است.

می خواهیم با مرتبه چند جمله ای و با استفاده از برنامه‌ریزی پویا، ضرب بهینه را بیابیم. روشن است که اصل بهینگی برای ضرب ماتریس‌ها برقرار است زیرا اگر مثلاً ضرب $(M_1 M_2)(M_3 M_4)$ بهینه باشد باید M_1M_2 و M_3M_4 بهینه باشند.

در ادامه بحث فرض می‌کنیم ماتریس M_i دارای ابعاد $r_{i-1} \times r_i$ $(i = 1,2,...,n)$ می‌باشد مثلاً ماتریس M_1 یک ماتریس $r_0 \times r_1$ می‌باشد. همچنین فرض می‌کنیم m_{ij} حداقل تعداد عملیات ضرب عددی برای محاسبه حاصل $M_i \times M_{i+1} ... \times M_j$ می‌باشد.

مثلاً m_{13} یعنی حداقل تعداد ضرب ماتریس‌های $M_1 \times M_2 \times M_3$. واضح است که $m_{ii} = 0$ می‌باشد.

برای محاسبه m_{ij} فرض کنید ضرب $M_i \times ... \times M_j$ را به صورت زیر نوشته ایم:

$(M_i \times ... \times M_k)(M_{k+1} \times ... \times M_j)$ (۱)

بنابراین m_{ij} از رابطه زیر بدست می‌آید:

$$m_{ij} = \min_{i \le k < j} \left(m_{ik} + m_{k+1,j} + r_{i-1} \times r_k \times r_j\right) \quad (۲)$$

که در این رابطه m_{ik} تعداد حداقل اعمال ضرب لازم برای محاسبه

$M_i \times M_{i+1} \times ... \times M_k$ (۳)

می باشد و $m_{k+1,j}$ حداقل ضرب لازم برای محاسبه

$M_{k+1} \times M_{k+2} + ... + M_j$ (۴)

می‌باشد و جمله $r_{i-1} \times r_k \times r_j$ تعداد ضرب لازم برای ضرب دو ماتریس حاصل از هر قسمت (۳) و (۴) می‌باشد.

با توجه به فرمول (۲) مراحل زیر را انجام می‌دهیم تا به m_{1n} یعنی جواب برسیم:

مرحله ۰: هیچ عمل ضرب ماتریسی. این مرحله معادل یادداشت $m_{ii} = 0$ ، $i = 1,2,...,n$ می‌باشد.

مرحله ۱: یک عمل ضرب ماتریسی. در این مرحله کلیه $m_{i,i+1}$ و $i = 1,2,...,n-1$ را محاسبه و در جدول می‌گذاریم.

مرحله ۲: دو عمل ضرب ماتریسی. در این مرحله کلیه $m_{i,i+2}$ و $i = 1,2,\ldots,n-2$ را محاسبه می‌کنیم.

مرحله ۱-n : n-۱ عمل ضرب ماتریسی. در این مرحله m_{1n} را محاسبه می‌کنیم.

مثال: فرض کنید ماتریس‌های M_1 و M_2 و M_3 و M_4 به ترتیب ۱۰×۲۰ و ۲۰×۵۰ و ۵۰×۱ و ۱×۱۰۰ باشند. کمیت‌های گفته شده را محاسبه می‌کنیم:

$$m_{11} = m_{22} = m_{33} = m_{44} = 0$$

$$\begin{cases} m_{12} = \min(m_{11} + m_{22} + 10\times20\times50) = 10000 \\ m_{23} = \min(m_{22} + m_{33} + 20\times50\times1) = 1000 \\ m_{34} = \min(m_{33} + m_{44} + 50\times1\times100) = 5000 \end{cases}$$

$$\begin{cases} m_{13} = \min(m_{11} + m_{23} + 10\times20\times1, m_{12} + m_{33} + 10\times50\times1) = 1200 \\ m_{24} = \min(m_{22} + m_{34} + 20\times50\times100, m_{23} + m_{44} + 20\times1\times100) = 3000 \end{cases}$$

$$m_{14} = \min(m_{11} + m_{24} + 10\times20\times100, m_{12} + m_{34} + 10\times50\times100, m_{13} + m_{44} + 10\times1\times100) = 2200$$

خلاصه عملیات فوق در جدول زیر آمده است:

مرحله ۰	$m_{11} = 0$	$m_{22} = 0$	$m_{33} = 0$	$m_{44} = 0$
مرحله ۱	$m_{12} = 10000$	$m_{23} = 1000$	$m_{34} = 5000$	
مرحله ۲	$m_{13} = 1200$	$m_{24} = 3000$		
مرحله ۳	$m_{14} = 2200$			

الگوریتم optmm تعداد حداقل ضرب را برای n ماتریس بدست می‌آورد:

```
int optmm (int m[100][100] , int n)
{
  int i, j, L;
  for (i = 1;i <= n ; i++) m[i][i] = 0;
  for (L = 1;L <= n;L++)
   for (i = 1; i <= n - L; i++)
   {
     j = i + L;
     m[i][j] = min {m[i][k] + m[k+1][j] + r_{i-1} * r_k * r_j}
               i≤k<j
   }
  return m [1][n];
}
```

الگوریتم فوق از مرتبه $O(n^3)$ می‌باشد. (مینیمم‌گیری خود o(n) است، و دو حلقه for دیگر مرتبه $O(n^2)$ می‌کند) در واقع آرایه m در مثال قبل به شکل زیر است:

$$m = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 10000 & 1200 & 2200 \\ & 0 & 1000 & 3000 \\ & & 0 & 5000 \\ & & & 0 \end{bmatrix} \end{array}$$

که $m[۱][۴] = ۲۲۰۰$ تعداد حداقل ضرب ۴ ماتریس گفته شده را نشان می‌دهد.

برای تعیین ترتیب بهینه ضرب ماتریس‌ها آرایه دیگری به نام p که ابعاد آن برابر ابعاد m است تعریف می‌کنیم و هر بار مقدار k را که به ازای آن m_{ij} مینیمم شد در محل P_{ij} ذخیره می‌کنیم. در مثال ضرب $M_۱ \times M_۲ \times M_۳ \times M_۴$ دیدیم که $m_{۱۴}$ از رابطه زیر بدست آمد:

$$m_{۱۴} = \min\left(\underbrace{۲۳۰۰۰}_{k=۱} , \underbrace{۶۵۰۰۰}_{k=۲} , \underbrace{۲۲۰۰}_{k=۳}\right) = ۲۲۰۰$$

یعنی مینیمم ضرب $m_{۱۴}$ به ازای $k=۳$ بدست آمد یعنی: $(M_۴)$. $(M_۱ \times M_۲ \times M_۳)$ حال برای آنکه ببینیم نقطه شکست $M_۱ \times M_۲ \times M_۳$ کجاست، به $m_{۱۳}$ نگاه می‌کنیم:

$$m_{۱۳} = \min\left(\underbrace{۱۲۰۰۰}_{k=۱} , \underbrace{۱۰۵۰۰}_{k=۲}\right) = ۱۲۰۰$$

پس نقطه شکست $M_۱ M_۲ M_۳$ ، $k=۱$ $(M_۲ M_۳)$. $(M_۱)$ می‌باشد یعنی ضرب بهینه به صورت $(M_۱.(M_۲.M_۳))M_۴$ می‌باشد.

ماتریس p برای این مثال به شکل زیر است:

$$p = \begin{array}{c} \\ ۱ \\ ۲ \\ ۳ \\ ۴ \end{array} \begin{array}{c} \begin{array}{cccc} ۱ & ۲ & ۳ & ۴ \end{array} \\ \begin{bmatrix} ۱ & ۱ & ۳ \\ & ۲ & ۲ \\ & & ۳ \\ & & \end{bmatrix} \end{array}$$

مثال: می‌خواهیم ۶ ماتریس $M_۱M_۲M_۳M_۴M_۵M_۶$ را در هم ضرب کنیم. ماتریس p به صورت زیر بدست آمده است. ترتیب بهینه ضرب را مشخص کنید.

$$p = \begin{array}{c} \\ ۱ \\ ۲ \\ ۳ \\ ۴ \\ ۵ \\ ۶ \end{array} \begin{array}{c} \begin{array}{cccccc} ۱ & ۲ & ۳ & ۴ & ۵ & ۶ \end{array} \\ \begin{bmatrix} ۱ & ۱ & ۱ & ۱ & ۱ & ۱ \\ & ۲ & ۳ & ۴ & ۵ & ۵ \\ & & ۳ & ۴ & ۵ & \\ & & & ۴ & ۵ & \\ & & & & ۵ & \\ & & & & & \end{bmatrix} \end{array}$$

حل: عنصر $p[۱,۶] = ۱$ یعنی نقطه شکست برای ضرب $M_۱...M_۶$، نقطه ۱ است پس: $(M_۱)(M_۲ M_۳ M_۴ M_۵ M_۶)$

حال باید $p[۲,۶]$ را بررسی کنیم که مقدار آن ۵ می‌باشد پس نقطه شکست $M_۲...M_۶$ نقطه ۵ است یعنی:

$(M_۱)((M_۲ M_۳ M_۴ M_۵)(M_۶))$

حال $p[۲,۵]=۴$ پس:

$$(M_۱)((M_۲\,M_۳\,M_۴)(M_۵)(M_۶))$$

و با توجه به اینکه $p[۲,۴]=۳$ در نتیجه:

$$M_۱(((M_۲\,M_۳)M_۴)M_۵)M_۶$$

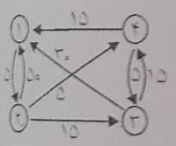
ترتیب بهینه ضرب است.

۲- الگوریتم فلوید برای یافتن تمام کوتاهترین مسیرها

فرض کنید $G=(V,E)$ یک گراف جهت دار با n راس است که $V=\{۱,۲,...,n\}$ مجموعه رئوس گراف و E مجموعه یال‌ها است. همچنین فرض کنید C ماتریس هزینه‌های گراف است و به صورت زیر تعریف می‌شود:

$$C[i,j]=\begin{cases}۰ & \text{اگر } i=j \text{ باشد}\\ \infty & \text{اگر از راس } i \text{ به } j \text{ یال نباشد.}\\ \text{وزن یال} & \text{اگر یالی از راس } i \text{ به } j \text{ باشد}\end{cases}$$

به عنوان مثال برای گراف مقابل ماتریس C عبارت است از:



$$\Rightarrow C=\begin{array}{c}\\ ۱\\ ۲\\ ۳\\ ۴\end{array}\begin{array}{c}\begin{array}{cccc}۱ & ۲ & ۳ & ۴\end{array}\\ \begin{bmatrix}۰ & ۵ & \infty & \infty\\ ۵۰ & ۰ & ۱۵ & ۵\\ ۳۰ & \infty & ۰ & ۱۵\\ ۱۵ & \infty & ۵ & ۰\end{bmatrix}\end{array}$$

می خواهیم تمامی کوتاهترین مسیرها بین رئوس گراف را بیابیم. به عبارتی می‌خواهیم یک ماتریس $A_{n\times n}$ بیابیم که درایه $A[i,j]$ حاوی طول کوتاهترین مسیر از راس i به راس j می‌باشد. به عنوان مثال برای گراف شکل فوق ماتریس A به صورت زیر بدست خواهد آمد:

$$A=\begin{array}{c}\\ ۱\\ ۲\\ ۳\\ ۴\end{array}\begin{array}{c}\begin{array}{cccc}۱ & ۲ & ۳ & ۴\end{array}\\ \begin{bmatrix}۰ & ۵ & ۱۵ & ۱۰\\ ۲۰ & ۰ & ۱۰ & ۵\\ ۳۰ & ۳۵ & ۰ & ۱۵\\ ۱۵ & ۲۰ & ۵ & ۰\end{bmatrix}\end{array}$$

مثلاً درایه $A[۱,۳]=۱۵$ به این معنی است که کوتاهترین مسیر بین راس ۱ و راس ۳ که مسیر $<۱,۲,۴,۳>$ است دارای طول ۱۵ می‌باشد و به همین ترتیب.

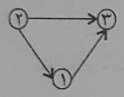
اگر بخواهیم این مساله را با بررسی همه حالات حل کنیم مرتبه زمانی، بدتر از نمایی $(O(n!))$ خواهد بود ولی با برنامه‌ریزی پویا الگوریتمی ارائه می‌دهیم از مرتبه $O(n^۳)$.

لازم به ذکر است که در این مساله، اصل بهینگی برقرار است. زیرا اگر مسیر i به j که از k می‌گذرد کوتاهترین باشد آنگاه مسیر i به k و k به j نیز کوتاهترین است.

برای حل مساله، ماتریس‌های $A_۰$ تا A_n را بدست می‌آوریم. $A_۰$ همان ماتریس هزینه یعنی C می‌باشد و A_k از روی $A_{k-۱}$ بدست می‌آید و A_n همان A یعنی جواب می‌باشد.

درایه‌های A_k طول کوتاهترین مسیرهایی را می‌دهند که فقط رئوس $\{1,2,..,k\}$ به عنوان رئوس میانی می‌توانند باشند.

مثلاً $A_1[2,3]$ یعنی طول کوتاهترین مسیر از راس ۲ به راس ۳ به شرطی که این مسیر فقط بتواند از راس ۱ عبور کند. می‌توان گفت



$$A_1[2,3] = \min\left(A_0[2,3]\ ,\ A_0[2,1]+A_0[1,3]\right)$$

زیرا کوتاهترین مسیر از راس ۲ به راس ۳، به شرطی که فقط راس ۱ بتواند در مسیر حضور داشته باشد یا مسیر مستقیم $\langle 2,3\rangle$ می‌باشد که هزینه آن $A_0[2,3]$ $(C[2,3])$ است یا مسیر $<2,1,3>$ می‌باشد که هزینه آن مجموع $A_0[1,3]+A_0[2,1]$ است.

پس از اینکه همه درایه‌های ماتریس A_1 محاسبه شد، ماتریس A_2 را محاسبه می‌کنیم یعنی تاثیر راس ۲ را روی تمام مسیرها محاسبه می‌کنیم. بنابراین $A_2[i,j]$ یعنی کوتاهترین مسیر از راس i به راس j به شرطی که رئوس $\{2$ و $1\}$ بتوانند رئوس واسط باشند.

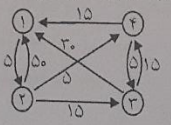
واضح است که

$$A_2[i,j] = \min\left(\underbrace{A_1[i,j]}_{\substack{\text{کوتاهترین مسیر از } i\\ \text{به } j \text{ با واسطه راس ۱}}}\ ,\ \underbrace{A_1[i,2]}_{\substack{\text{کوتاهترین از } i \text{ به ۲}\\ \text{به واسطه راس ۱}}} + \underbrace{A_1[2,j]}_{\substack{\text{کوتاهترین از ۲ به}\\ j \text{ با واسطه راس ۱}}}\right)$$

به طور کلی می‌توان فرمول بهینه‌سازی زیر را ارائه داد:

$$A_k[i,j] = \min\left(A_{k-1}[i,j]\ ,\ A_{k-1}[i,k]+A_{k-1}[k,j]\right)$$

مثال: مراحل الگوریتم فلوید را روی گراف زیر نشان دهید.



حل:

$$A_0=C=\begin{array}{c} \\ 1\\2\\3\\4\end{array}\begin{array}{c}\begin{array}{cccc}1&2&3&4\end{array}\\ \begin{bmatrix}0&5&\infty&\infty\\50&0&15&5\\30&\infty&0&15\\15&\infty&5&0\end{bmatrix}\end{array} \Rightarrow A_1=\begin{array}{c} \\ 1\\2\\3\\4\end{array}\begin{array}{c}\begin{array}{cccc}1&2&3&4\end{array}\\ \begin{bmatrix}0&5&\infty&\infty\\50&0&15&5\\30&35&0&15\\15&20&5&0\end{bmatrix}\end{array} \Rightarrow A_2=\begin{array}{c} \\ 1\\2\\3\\4\end{array}\begin{array}{c}\begin{array}{cccc}1&2&3&4\end{array}\\ \begin{bmatrix}0&5&20&10\\50&0&15&5\\30&35&0&15\\15&20&5&0\end{bmatrix}\end{array}$$

$$A_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \end{array} \Rightarrow A_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \end{array}$$

مثلاً برای محاسبه $A_1[3,2]$ این گونه عمل شده است:

$$A_1[3,2] = \min\left(A_0[3,2], A_0[3,1] + A_0[1,2]\right)$$
$$= \min(\infty, 30+5) = 35$$

به همین ترتیب با توجه به فرمول بهینه سازی می‌توان سایر درایه‌ها را محاسبه کرد.

توجه: سطر k، ستون k از ماتریس A_k با A_{k-1} مساوی است.

الگوریتم را می‌توان به صورت زیر نوشت:

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

توجه کنید از آنجا که پس از محاسبه ماتریس A_k نیاز به A_{k-1} نداریم، در هر مرحله جواب A_k را در ماتریس قدیمی A_{k-1} ذخیره کرده ایم و در نهایت A (k=n)، بدست آمده، جواب است.

توجه: مرتبه الگوریتم نوشته شده چون سه حلقه تو در تو دارد $O(n^3)$ است.

تاکنون ماتریس A را پیدا کردیم که حداقل طول مسیر از هر گره i به هر گره j را نشان می‌دهد. حال می‌خواهیم خود مسیر را مشخص کنیم. برای این منظور ماتریس p را بدست می‌آوریم. P یک ماتریس n×n با مقدار اولیه صفر است:

```
for(k = 1;k <= n;k++)
  for(i = 1;i <= n;i++)
    for(j = 1; j <= n; j++)
      if (A[i,k]+ A[k,j] < A[i,j])
       {
         p[i,j] = k;
         A [i , j]=A[i , k]+A[k , j] ;
       }
```

برای گراف مثال قبل داریم:

$$P_0 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

پس از تاثیر گره ۱ (k=۱):

$$\Rightarrow \quad p_1 = \begin{array}{c} \\ ۱ \\ ۲ \\ ۳ \\ ۴ \end{array} \begin{array}{c} \begin{array}{cccc} ۱ & ۲ & ۳ & ۴ \end{array} \\ \begin{bmatrix} ۰ & ۰ & ۰ & ۰ \\ ۰ & ۰ & ۰ & ۰ \\ ۰ & ۱ & ۰ & ۰ \\ ۰ & ۱ & ۰ & ۰ \end{bmatrix} \end{array}$$

پس از تاثیر گره ۲ (k=۲):

$$\Rightarrow \quad p_2 = \begin{bmatrix} ۰ & ۰ & ۲ & ۲ \\ ۰ & ۰ & ۰ & ۰ \\ ۰ & ۱ & ۰ & ۰ \\ ۰ & ۱ & ۰ & ۰ \end{bmatrix}$$

پس از تاثیر گره ۳ (k=۳):

$$\Rightarrow \quad p_3 = \begin{bmatrix} ۰ & ۰ & ۲ & ۲ \\ ۳ & ۰ & ۰ & ۰ \\ ۰ & ۱ & ۰ & ۰ \\ ۰ & ۱ & ۰ & ۰ \end{bmatrix}$$

پس از تاثیر گره ۴ (k=۴):

$$\Rightarrow p = \quad p_4 = \begin{array}{c} \\ ۱ \\ ۲ \\ ۳ \\ ۴ \end{array} \begin{array}{c} \begin{array}{cccc} ۱ & ۲ & ۳ & ۴ \end{array} \\ \begin{bmatrix} ۰ & ۰ & ۴ & ۲ \\ ۴ & ۰ & ۴ & ۰ \\ ۰ & ۱ & ۰ & ۰ \\ ۰ & ۱ & ۰ & ۰ \end{bmatrix} \end{array}$$

درایه $p[۱,۳] = ۴$ یعنی کوتاهترین مسیر از راس ۱ به ۳ شامل راس ۴ است. لذا باید $p[۱,۴]$ و $p[۴,۳]$ را بررسی کنیم. $p[۱,۴] = ۲$ یعنی در مسیر راس ۱ به راس ۴ راس ۲ وجود دارد. $p[۴,۳] = ۰$ یعنی در مسیر ۴ به ۳ راس واسطی وجود ندارد. پس کوتاهترین مسیر از راس ۱ به راس ۳ مسیر: ۳←۴←۲←۱ می‌باشد.

تابع زیر رئوس مسیر i به 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);
   }
 }
```

نکته: *می‌توان با روش ساده دیگری (ولی بد) که شبیه ضرب ماتریس‌هاست، نیز طول کوتاه‌ترین مسیر بین هر دو رأس را یافت. اگر C ماتریس هزینه باشد، L_1 و L_2 و ... L_{n-1} را محاسبه می‌کنیم:*

$$L_1 = C \;,\; L_2 = L_1\ C = C^2,\ L_3 = L_2\ C = C^3,\ldots,\ L_{n-1} = L_{n-2}C = C^{n-1}$$

این روابط همان ضرب ماتریس‌هاست[1] با این تفاوت که به جای عمل ضرب (•) عمل + و به جای عمل جمع (Σ)، عمل مینیمم‌گیری قرار دهید.

روی گراف مثال قبل نشان می‌دهیم:

$$L_1 = C = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{bmatrix} \end{array} \Rightarrow L_2 = C \cdot C = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & 20 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \end{array} \Rightarrow L_4 = L_2 \cdot C = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{bmatrix} \end{array}$$

که L_3 جواب است. دقت کنید مثلاً درایه سطر ۱ ستون ۴ از L_2 از عمل روی سطر ۱ در ستون ۴ از L_1 به دست می‌آید:

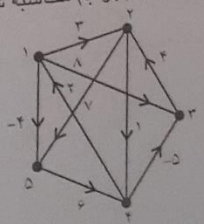
$$\min(0+\infty,\ 5+5,\ \infty+15,\ \infty+0) = 10$$

مرتبه این الگوریتم $\theta(n^4)$ است ولی می‌توان با کمک روابط زیر با مرتبه $\theta(n^3 Lgn)$ الگوریتم را پیاده‌سازی کرد:

$$L_1 = C,\ L_2 = C^2 = C \cdot C,\ L_4 = C^2 \cdot C^2,\ L_8 = C^4 \cdot C^4, \ldots$$

همانطور که ملاحظه می‌شود به صورت لگاریتمی به L_{n-1} می‌رسیم.

مثال: برای گراف مقابل با روش فوق، L_4 (جواب) محاسبه شده است:



شکل ۱۰

۱- در ضرب ماتریس‌ها $A_{n\times n} \times B_{n\times n} = C_{n\times n}$، درایه C_{ij} از ماتریس C به صورت $C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$ محاسبه می‌شود.

$$L_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \end{array} \Rightarrow L_2 \begin{bmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{bmatrix} \Rightarrow L_3 = \begin{bmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

$$\Rightarrow L_4 = \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

البته می‌توان L_3 را محاسبه نکرد. $L_4 = L_2 \cdot L_2$

۳- کوله پشتی ۱-۰ با ارزش ماکزیمم

مساله کوله پشتی را در بخش قبل مطرح کردیم و یک راه حل بهینه به روش حریصانه ارائه نمودیم.

در آنجا مجاز بودیم که یک شی را کامل انتخاب کنیم و یا کسری از شی را انتخاب کنیم. چنانچه مجاز نباشیم بخشی از شی را انتخاب کنیم یعنی شی یا باید انتخاب شود و یا اصلاً انتخاب نشود، در این صورت مسئله را کوله پشتی ۱-۰ می‌گویند. با روش حریصانه نمی‌توان جواب بهینه برای کوله پشتی ۱-۰ بدست آورد.

راه حل‌های ارائه شده برای کوله پشتی ۱-۰ دارای پیچیدگی محاسباتی بالایی هستند و در واقع این مساله یک مساله رام نشدنی (intractable) است. برای تعداد اشیاء محدود و کم با برنامه نویسی پویا می‌توان یک جواب بهینه برای مساله پیدا کرد.

فرضیات:

M= حداکثر ظرفیت کوله

$(w_1, w_2, ..., w_n)$ = جرم اشیاء

$(p_1, p_2, ..., p_n)$ = ارزش اشیاء

$(x_1, x_2, ..., x_n)$ = میزان انتخاب از هر شی ، $x_i = 0$ یا ۱

هدف ماکزیمم سازی $\sum_{i=1}^{n} p_i x_i$ به شرطی که $\sum_{i=1}^{n} w_i x_i \le M$

مثال: اگر $(w_1, w_2, w_3) = (5, 10, 20)$ و $(p_1, p_2, p_3) = (50, 60, 140)$ و M=30 باشد

در این صورت $\left(\frac{p_1}{w_1}, \frac{p_2}{w_2}, \frac{p_3}{w_3}\right) = (10, 6, 7)$ و حال اگر طبق روش حریصانه اشیاء را انتخاب کنیم (یا انتخاب نکنیم ۱ و $x_i = 0$ آنگاه اشیاء ۱ و ۳ انتخاب می‌شوند و سود حاصل ۱۹۰ می‌باشد. حال آنکه بهینه آن است که شی ۲ و ۳ انتخاب شوند تا سود حاصل ۲۰۰ باشد. ملاحظه می‌شود که روش حریصانه برای کوله پشتی ۱-۰ حل بهینه تولید نمی‌کند.

برای مساله کوله پشتی ۰-۱، اصل بهینگی برقرار است که نشان می‌دهیم:

اگر A یک زیر مجموعه بهینه از n قطعه باشد، دو حالت امکان پذیر است: یا A حاوی قطعه nام هست یا نیست. اگر A حاوی قطعه nام نباشد، A با زیر مجموعه‌ای بهینه از (n-۱) قطعه اول برابر است. اگر A حاوی قطعه nام باشد، منفعت کل برابر p_n به علاوه منفعت بهینه است هنگامی که قطعات را از n-۱ قطعه اول انتخاب می‌کنیم (البته با رعایت این اصل که وزن کل از $M - w_n$ بیشتر نشود) لذا اصل بهینگی برقرار است.

اگر c[i][m] سود بهینه حاصل از انتخاب i قطعه اول باشد به گونه‌ای که وزن کل آن از m تجاوز نکند داریم:

$$c[i][m] = \begin{cases} 0 & \text{if } i = 0 \text{ or } m = 0 \\ \max\left(c[i-1][m],\ p_i + c[i-1][m-w_i]\right) & \text{if } w_i \le m \\ c[i-1][m] & \text{if } w_i > m \end{cases}$$

و جواب مسئله c[n][M] می‌باشد.

الگوریتم زیر c[0..n][0..M] را به ترتیب سطری محاسبه می‌کند و c[n][M] جواب است.

```
p[1..n]values , w[1..n]weights
for  m ← 0  to  M  do  c[0][m] ← 0
for  i ← 1  to  n  do
{
  c[i][0] ← 0
  for  m ← 1 to M do
    if  w[i] ≤ m
      then  if  p[i] + c[i-1][m - w[i]] > c[i-1][m]
              then  c[i][m] ← p[i] + c[i-1][m - w[i]]
              else  c[i][m] ← c[i-1][m]
      else  c[i][m] ← c[i-1][m]
}
```

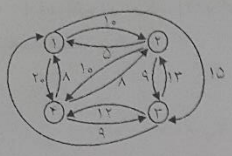
مرتبه این الگوریتم $\theta(nM)$ است و اگر M از 2^n بیشتر باشد، مرتبه الگوریتم از روش عادی با مرتبه $\theta(2^n)$ نیز بدتر می‌شود ولی می‌توان طوری آن را بهبود بخشید که مرتبه کوله پشتی صفر و یک $O(\min(2^n, nM))$ باشد.

تمرین: در مسئله کوله پشتی صفر و یک، فرض کنید عناصر را به ترتیب صعودی وزن که مرتب کردیم، به ترتیب نزولی ارزش مرتب شده‌اند، یک الگوریتم کارا برای یافتن جواب بهینه بیابید.

۴- فروشنده دوره گرد

در یک گراف وزن‌دار، می‌خواهیم یک سیکل بهینه‌هامیلتونی (تور) بیابیم. یعنی می‌خواهیم از یک راس شروع کنیم، همه رئوس را یک بار و فقط یک بار ملاقات کنیم و به راس شروع باز گردیم به شرطی ک جمع وزن یال‌ها مینیم باشد.

مثال: در گراف مقابل سیکل [۱ و ۳ و ۴ و ۲ و ۱] سیکل بهینه‌هامیلتونی است که هزینه آن ۳۵=۶+۹+۱۰+۱۰ می‌باشد.



در یک گراف n راسی اگر بخواهیم با روش معمولی کل سیکل‌ها را بررسی کنیم و سیکل بهینه را بیابیم به $(n-1)!$ بررسی نیاز است. ولی با روش برنامه سازی پویا می‌توان الگوریتمی بیان کرد از مرتبه $\theta(n^2 2^n)$ که بسیار بهتر از $(n-1)!$ است. مثلاً اگر هر عمل اصلی یک میکروثانیه به طول انجامد برای گراف n=۲۰ راسی داریم:

روش معمولی یافتن کلیه تورها = $(۲۰-۱)!\mu sec$ = ۳۸۵۷ سال

روش برنامه نویسی پویا = $۲۰^۲ \times ۲^{۲۰}\ \mu sec \simeq$ دقیقه ۷

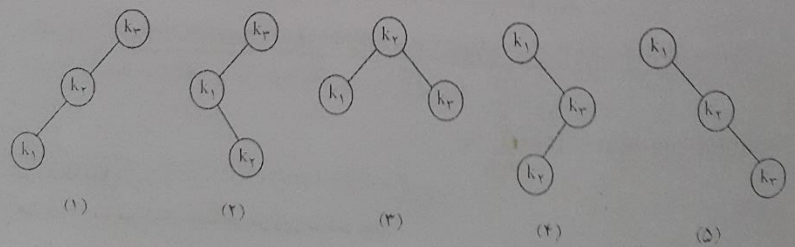
نکته: *حافظه مصرفی الگوریتم با روش برنامه سازی پویا از مرتبه* $\theta(n2^n)$ *است.*

نکته: *تاکنون کسی نتوانسته است برای مساله فروشنده دوره گرد الگوریتمی بهتر از حالت نمایی پیدا کند، و همچنین کسی هم عدم امکان یافتن چنین الگوریتمی را اثبات نکرده است. (همانند مساله کوله پشتی ۱- ۰)*

۵- درخت جستجوی دودویی بهینه (Optimal BST)

فرض کنید n تا کلید متمایز $k_1 < k_2 < \ldots < k_n$ وجود دارد. و احتمال آنکه کلید k_i را جستجو کنیم برابر p_i است. می‌خواهیم با این کلیدها یک BST بسازیم به طوریکه عبارت $\sum_{i=1}^{n} c_i\, p_i$ مینیمم شود. c_i برابر تعداد مقایسه‌ها برای یافتن k_i در BST است. عبارت $\sum_{i=1}^{n} c_i\, p_i$ زمان جستجوی میانگین برای جستجوی کلیدهاست که می‌خواهیم مینیمم شود.

مثال: با ۳ کلید $k_1 < k_2 < k_3$، ۵ درخت جستجوی دودویی وجود دارد. با فرض اینکه $p_3 = ۰/۱$ و $p_2 = ۰/۲$ و $p_1 = ۰/۷$، میانگین زمان جستجو را برای هر درخت محاسبه کنید.



(دقت کنید در درخت (۱) برای یافتن $k_۳$، یک مقایسه برای یافتن $k_۲$، دو مقایسه و برای $k_۱$، سه مقایسه انجام می‌شود)

۱) $۳(۰/۷)+۲(۰/۲)+۱(۰/۱)=۲/۶$

۲) $۲(۰/۷)+۳(۰/۲)+۱(۰/۱)=۲/۱$

۳) $۲(۰/۷)+۱(۰/۲)+۲(۰/۱)=۱/۸$

۴) $۱(۰/۷)+۳(۰/۲)+۲(۰/۱)=۱/۵$

۵) $۱(۰/۷)+۲(۰/۲)+۳(۰/۱)=۱/۴$

در این مثال درخت ۵، بهینه است. درواقع کلیدهایی که خیلی به ندرت جستجو می‌شوند (احتمال‌شان کم است) بهتر است در سطوح آخر و کلیدهایی که احتمال جستجوی آنها زیاد است، در سطوح بالایی باشند.

برای یافتن BST بهینه، شاید یک راه آن باشد که همه حالات را بررسی کنیم. ولی مرتبه این عمل از نمایی بدتر است. می‌خواهیم با روش برنامه‌نویسی پویا، BST بهینه را بیابیم.

زیر درختان یک درخت BST بهینه، خود BST بهینه هستند. یعنی اصل بهینگی برقرار است. در مثال قبل درخت ۵ بهینه بود، می‌توان بررسی کرد که  نیز بهینه است.

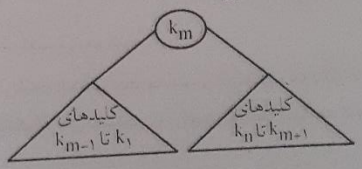
برای رسیدن به الگوریتم، ماتریس A را تعریف می‌کنیم که مفهوم هر درایه آن به صورت زیر است:

مینیمم زمان جستجو برای BST با کلیدهای k_i تا k_j $A[i][j]=k_j$

به طوریکه $k_i \le k_j$ و $۱\le i\le j\le n$

بدیهی است که $A[i][i]=p_i$ چرا که در این حالت BST فقط شامل k_i است و یک مقایسه $(c_i=۱)$ لازم است و احتمال آن نیز p_i است پس $A[i][i]=۱\times p_i=۱$

جواب ما $A[۱][n]$ است. فرض کنید BST بهینه را یافته‌ایم و ریشه آن k_m است. بنابر اصل بهینگی زیر درختان چپ و راست نیز خودشان BST بهینه هستند:



$$A[۱][n]=\underbrace{A[۱][m-۱]}_{(۱)}+\underbrace{p_۱+\ldots+p_{m-۱}}_{(۲)}+\underbrace{p_m}_{(۳)}+\underbrace{A[m+۱][n]}_{(۴)}+\underbrace{p_{m+۱}+\ldots+p_n}_{(۵)}$$

$$=A[۱][m-۱]+A[m+۱][n]+\sum_{t=۱}^{n}p_t$$

در رابطه فوق،

عبارت ۱: مینیمم زمان جستجو در زیر درخت چپ.

عبارت ۲ : هر یک از نودهای زیر درخت چپ را که بخواهیم جستجو کنیم، بخاطر مقایسه با ریشه، یک مقایسهٔ اضافی دارد.

عبارت ۳: خود ریشه یک مقایسه دارد و احتمال آن نیز p_m است.

عبارت ۴: مینیمم زمان جستجو در زیر درخت راست.

عبارت ۵: مشابه عبارت ۲، هر یک از نودهای زیر درخت راست، یک مقایسه اضافی با ریشه دارند.

رابطه فوق با این فرض بود که k_m ریشه باشد ولی با توجه به اینکه هر یک از کلیدهای ۱ تا n می‌توانند ریشه باشند:

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

بنابراین فرمول کلی برای $A[i][j]$ عبارت است از:

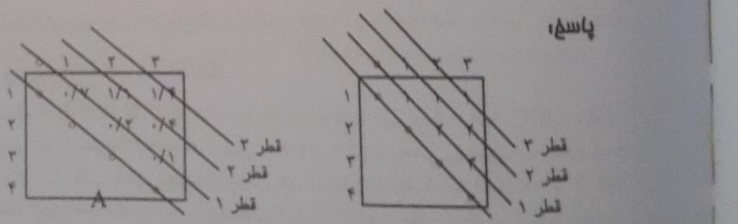
$$A[i][j] = \min_{i \le m \le j} (A[i][m-1] + A[m+1][j]) + \sum_{t=i}^{j} p_t \quad , \quad (i < j)$$
$$A[i][i] = p_i$$

از این فرمول برای ساخت ماتریس A استفاده می‌شود. جهت سهولت ماتریس را یک ماتریس $(n+1)\times(n+1)$ به شکل $A[1..n+1, 0..n]$ در نظر می‌گیریم که قطر اصلی آن صفر است.

ماتریس دیگری به نام R نیز تعریف می‌کنیم که $R[i][j]$ بیانگر اندیس کلیدی است که به عنوان ریشه انتخاب شده (درخت شامل k_i تا k_j). از ماتریس R برای ساخت خود درخت کمک خواهیم گرفت. ابعاد ماتریس R نیز مشابه A است و این ماتریسها، قطر به قطر پر می‌شوند.

مثال: درخت BST بهینه و هزینه آن را برای ۳ کلید $k_1 < k_2 < k_3$ با احتمال $p_3 = ۰/۱$ و $p_2 = ۰/۲$ و $p_1 = ۰/۷$ بیابید.

پاسخ:



قطر ۱: با توجه به اینکه $A[i][i] = p_i$ پس در قطر ۱ $(A[3][3], A[2][2], A[1][1])$ احتمالات (p_3, p_2, p_1) را قرار می‌دهیم. در ماتریس R نیز قرار می‌دهیم $R[i][i] = i$ چرا که با کلید k_i ریشه خود k_i است.

قطر ۲:

$$A[1][2] = \min_{1 \le m \le 2} (A[1][m-1] + A[m+1][2]) + \sum_{t=1}^{2} p_t$$

$$= \min\left(\underbrace{A[1][0]}_{0} + \underbrace{A[2][2]}_{0.2},\ \underbrace{A[1][1]}_{0.7} + \underbrace{A[3][2]}_{0}\right) + (p_1 + p_2)$$

$$= \min\left(\underbrace{0.2}_{m=1},\ \underbrace{0.7}_{m=2}\right) + (0.7 + 0.2) = 1.1$$

پس در $R[1][2]$ عدد $m = 1$ قرار می‌گیرد.

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

$$= \min\left(\underbrace{A[2][1]}_{0} + \underbrace{A[3][3]}_{0.1},\ \underbrace{A[2][2]}_{0.2} + \underbrace{A[4][3]}_{0}\right) + (p_2 + p_3)$$

$$= 0.1 + (0.2 + 0.1) = 0.4 \ \Rightarrow\ R[2][3] = 2$$

$$A[1][3] = \min_{1 \le m \le 3} (A[1][m-1] + A[m+1][3]) + \sum_{t=1}^{3} p_t$$

قطر ۳:

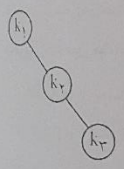
$$= \min(A[1][0] + A[2][3],\ A[1][1] + A[3][3],\ A[1][2] + A[4][3]) + (p_1 + p_2 + p_3)$$

$$= \min\left(\underbrace{0 + 0.4}_{k=1},\ \underbrace{0.7 + 0.1}_{k=2},\ \underbrace{1.1 + 0}_{k=3}\right) + (0.7 + 0.2 + 0.1) = 1.4$$

$$\Rightarrow\ R[1][3] = 1$$

بنابراین متوسط زمان جستجو ۱/۴ است. برای رسم درخت، با توجه به اینکه $R[1][3] = 1$ پس k_1 ریشه درخت شامل کلیدهای k_1 تا k_3 است: (k_1)

و چون $R[2][3] = 2$ پس، k_2 ریشه درخت شامل کلیدهای k_2 و k_3 است و چون $k_2 > k_1$ پس k_2 را سمت راست k_1 می‌کشیم و در نهایت چون $k_3 > k_2$، k_3 را نیز سمت راست k_2 می‌کشیم:



برای پیاده‌سازی الگوریتم، نیاز به ۳ حلقه تو در توست و مرتبه آن $\theta(n^3)$ است.

۶- بزرگترین زیردنباله مشترک (LCS)[1]

فرض کنید دنباله $X = <A,B,C,B,D,A,B>$ داده شده است زیر دنباله $Z = <B,C,A>$ از X زیردنباله‌ای است که متناظر با دنباله اندیسهای $<2,3,6>$ است. یعنی زیردنباله Z، به ترتیب اندیسهای ۲ و ۳ و ۶ دنباله X را شامل شده است. فرض کنید دنباله $Y = <B,D,C,A,B,A>$ در این صورت Z زیردنباله Y نیز هست که متناظر با اندیسهای $<1,3,4>$ است. یعنی Z یک زیردنباله مشترک برای X و Y است که طولش ۳ است. زیردنباله $M = <B,C,B,A>$ نیز زیردنباله مشترک برای X و Y است. که طولش ۴ است. می‌توان بررسی کرد که زیردنباله مشترک با طول بیشتر از ۴ برای X و Y وجود ندارد، پس M بزرگترین زیردنباله مشترک (LCS) برای X و Y است. البته $N = <B,D,A,B>$ نیز یک LCS برای X و Y است. ما می‌خواهیم طول LCS(X,Y) و همچنین خود LCS(X,Y) را بیابیم. مثلاً طول LCS(X,Y) برابر ۴ است و خود LCS(X,Y) برابر $<B,D,A,B>$ یا $<B,C,B,A>$ است.

1- Longest Common Subsequence

فرض کنید $X=<x_1,x_2,...,x_m>$ و $Y=<y_1,y_2,...,y_n>$. یک روش برای یافتن $LCS(X,Y)$ این است که تمام زیردنباله‌های X که 2^m است را بیابیم و بررسی کنیم کدام، زیردنباله Y نیز هست و بزرگترین آن را انتخاب کنیم که این از مرتبه نمایی است.

ولی می‌توان با روش برنامه‌ریزی پویا، LCS را یافت. فرض کنید $Z=<z_1,z_2,...,z_k>$، LCS دنباله‌های X و Y باشد؛ و همچنین فرض کنید X_i پیشوند i ام X باشد. مثلاً اگر $X=<A,B,C,B,D,A,B>$ در این صورت $X_4=<A,B,C,B>$ یا $X_0=<>$ و $X_7=X$. جملات بازگشتی زیر قابل بیان است.

۱) if $x_m=y_n \Rightarrow (z_k=x_m=y_n)$ and $Z_{k-1}=LCS(X_{m-1},Y_{n-1})$

۲) if $x_m\neq y_n \Rightarrow \begin{cases}\text{if } z_k\neq x_m \Rightarrow Z=LCS(X_{m-1},Y)\\ \text{if } z_k\neq y_n \Rightarrow Z=LCS(X,Y_{n-1})\end{cases}$

دقت کنید که Z، LCS دنباله‌های X و Y است. پس جمله ۱ فوق صحیح است زیرا اگر آخرین کاراکتر دنباله‌های X و Y مساوی باشد $(x_m=y_n)$ آنگاه حتماً آخرین کاراکتر دنباله Z یعنی z_k با x_m یا y_n مساوی است و $Z_{k-1}=<z_1,z_2,...,z_{k-1}>$ یک LCS برای X_{m-1} و Y_{n-1} است.

بررسی کنید که جمله ۲ فوق نیز صحیح است.

فرض کنید $C[i,j]$ برابر طول LCS دنباله‌های X_i و Y_j باشد واضح است که اگر $i=0$ یا $j=0$ در این صورت $C[i,j]=0$ است. با توجه به بحث‌های انجام شد. فرمول بازگشتی زیر بدست می‌آید:

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

الگوریتم زیر با روش پویا و مرتبه $\theta(mn)$، سطر به سطر C را می‌سازد تا به $C[m,n]$ برسیم که طول LCS را می‌دهد. آرایه C به صورت $C[0..m,0..n]$ تعریف می‌شود.

همچنین در این الگوریتم آرایه $b[1..m,1..n]$ مقداردهی می‌شود که با کمک آن می‌توان خود LCS را یافت:

```
LCS-LENGTH (X,Y,m,n)
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 b and C
```

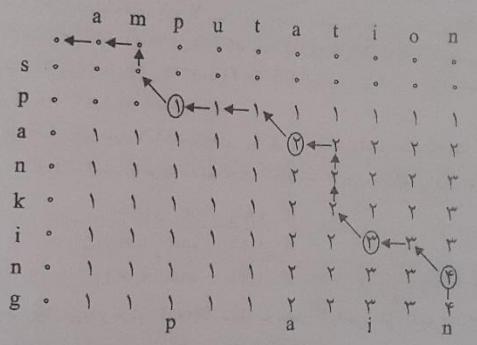
برای چاپ خود LCS می‌توان الگوریتم زیر را با مرتبه $O(m+n)$ ارائه داد:

```
PRINT(b,X,i,j)
if  i = ۰  or  j = ۰  then return
if  b[i, j] = "↖"  then PRINT(b, X, i − ۱, j − ۱), write x_i
else if  b[i, j] = "↑"  then PRINT(b, X, i − ۱, j)
else PRINT(b, X, i, j − ۱)
```

فراخوانی اولیه این تابع PRINT(b,X,m,n) است.

در تابع فوق هرگاه "↖" = b[i, j] یعنی یک کاراکتر مشترک در X و Y دیده شده است پس LCS در واقع مکانهایی هستند که علامت ↖ دارند.

مثال:



۷- ضریب دو جمله‌ای

ضریب دو جمله‌ای $\binom{n}{k}$ دارای فرمول بازگشتی مقابل است:

$$\binom{n}{k}=\begin{cases}\binom{n-۱}{k-۱}+\binom{n-۱}{k} & ۰<k<n \\ ۱ & k=۰ \text{ or } k=n\end{cases}$$

اگر بخواهیم با روش تقسیم و غلبه $\binom{n}{k}$ را محاسبه کنیم، تعداد فراخوانیها $۲\binom{n}{k}-۱$ است که بسیار زیاد است. با برنامه‌نویسی پویا می‌توان از پایین به بالا نتایج را محاسبه کرده و در یک آرایه ذخیره کرد و به $\binom{n}{k}$ رسید.

مطابق الگوریتم زیر:

```
int  bin(int n, int k)
{
   int i , j;
   int  B[0…n][0…k];
   for (i=0;i<=n;i++)
      for (j=0;j<=min(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 اجرای if برابر است با:

$$۱+۲+۳+....+k+\underbrace{(k+۱)+(k+۱)+....+(k+۱)}_{\text{بار } n-k+۱}$$

$$=\frac{k(k+۱)}{۲}+(n-k+۱)(k+۱)=\frac{(۲n-k+۲)(k+۱)}{۲}\in \theta(nk)$$

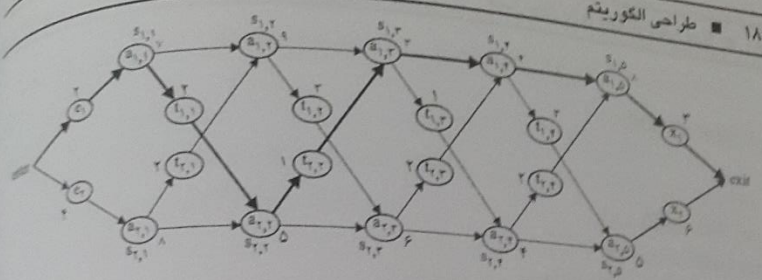
که خیلی از روش تقسیم و غلبه سریعتر است.

لازم به ذکر است که الگوریتم فوق را می‌توان با کمک یک آرایه یک بعدی B[0...k] نوشت که به عهده شما.

۸- زمانبندی خط تولید (assembly-line scheduling)

در یک کارخانه اتومبیل‌سازی، دو خط تولید وجود دارد، هر خط دارای n ایستگاه است ($S_{۱,۱}, S_{۱,۲}, ..., S_{۱,n}$ برای خط ۱ و $S_{۲,۱}, ..., S_{۲,n}$ ایستگاه‌های خط ۲). ایستگاه i ام خط ۱ و ۲ یک کار یکسان انجام می‌دهند ($S_{۱,۱}$ با $S_{۲,۱}$ و ... یکسان هستند) ولی هزینه آنها لزوماً یکسان نیست. هزینه ایستگاه $S_{i,j}$ $(i=۱,۲)$ برابر $a_{i,j}$ فرض می‌شود. هزینه ورود بدنه اتومبیل به خط ۱ برابر $e_۱$ و به خط ۲ برابر $e_۲$ و هزینه خروج از خطوط ۱ و ۲ به ترتیب $x_۱$ و $x_۲$ است. بدنه می‌تواند از یک خط به خط دیگر نیز منتقل شود. هزینه انتقال از $S_{۱,j}$ به $S_{۲,j+۱}$ برابر $t_{۱,j}$ است و هزینه انتقال از ایستگاه $S_{۲,j}$ به $S_{۱,j+۱}$ برابر $t_{۲,j}$ است.

می‌خواهیم ایستگاه‌هایی از خطوط ۱ و ۲ را انتخاب کنیم که کل هزینه‌ها مینیمم شود. در شکل زیر مسیر پررنگ نشان‌دهنده مسیر بهینه است که هزینه آن $۳۵=۳+۸+۴+۳+۱+۵+۲+۷+۲$ است.



برای حل مسئله، $f_i[j]$ را برابر کمترین هزینه رسیدن به ایستگاه $S_{i,j}$ $(j=1,\dots,n\ ,\ i=1,2)$ فرض می‌کنیم (هزینه خود $S_{i,j}$ نیز در نظر گرفته می‌شود) مثلاً در شکل فوق $f_1[1]=2+7=9$ و $f_2[1]=4+8=12$. f^* را برابر کمترین هزینه کل فرض می‌کنیم. در شکل فوق $f^*=35$ واضح است که $f^*=\min(f_1[n]+x_1\ ,\ f_2[n]+x_2)$ به راحتی می‌توان فرمولهای بازگشتی زیر را نوشت:

$$f_1[1]=e_1+a_{1,1}\quad ,\quad f_2[1]=e_2+a_{2,1}$$

$$f_1[j]=\min(f_1[j-1]+a_{1,j}\ ,\ f_2[j-1]+t_{2,j-1}+a_{1,j})\qquad j=2,\dots,n$$

$$f_2[j]=\min(f_2[j-1]+a_{2,j}\ ,\ f_1[j-1]+t_{1,j-1}+a_{2,j})\qquad j=2,\dots,n$$

با تعاریف می‌توان هزینه بهینه را یافت (f^*) ولی برای آنکه مشخص شود، چه ایستگاه‌هایی باید انتخاب شوند، $L_i[j]$ را برابر شماره خطی می‌گیریم که ایستگاه $j-1$ آن قبل از $S_{i,j}$ استفاده شده است یعنی ایستگاه $S_{L_i[j],j-1}$ قبل از ایستگاه $S_{i,j}$ است. $(i=1,2, j=2,\dots,n)$ و L^* را شماره خطی می‌گیریم که ایستگاه n از آن انتخاب شده است. در شکل قبل، $L^*=1$ و $L_1[2]=1$ زیرا برای رسیدن به $S_{1,2}$ بهترین مسیر، عبور از $S_{1,1}$ است. و یا $L_2[2]=1$ زیرا بهترین مسیر برای رسیدن به $S_{2,2}$ عبور از $S_{1,1}$ است:

$f_1[1]$	$f_1[2]$	$f_1[3]$	$f_1[4]$	$f_1[5]$
۹	۱۸	۲۰	۲۴	۳۲

$f_2[1]$	$f_2[2]$	$f_2[3]$	$f_2[4]$	$f_2[5]$
۱۲	۱۶	۲۲	۲۵	۳۰

$f^*=35$

$L_1[1]$	$L_1[2]$	$L_1[3]$	$L_1[4]$	$L_1[5]$
۱	۲	۱	۱	

$L_2[2]$	$L_2[3]$	$L_2[4]$	$L_2[5]$
۱	۲	۱	۲

$L^*=1$

الگوریتم یافتن بهترین مسیر، با روش برنامه‌ریزی پویا به قرار زیر است:

```
FASTEST-WAY(a,t,e,x,n)
f1[1] ← e1 + a1,1
f2[1] ← e2 + a2,1
for j ← 2 to n
    do {if f1[j−1] + a1,j ≤ f2[j−1] + t2,j−1 + a1,j
          then f1[j] ← f1[j−1] + a1,j , L1[j] ← 1
          else f1[j] ← f2[j−1] + t2,j−1 + a1,j , L1[j] ← 2
        if f2[j−1] + a2,j ≤ f1[j−1] + t1,j−1 + a2,j
          then f2[j] ← f2[j−1] + a2,j , L2[j] ← 2
          else f2[j] ← f1[j−1] + t1,j−1 + a2,j , L2[j] ← 1
        }
if f1[n] + x1 ≤ f2[n] + x2
    then f* = f1[n] + x1 , L* = 1
    else f* = f2[n] + x2 , L* = 2
```

و برای چاپ مسیر بهینه می‌توان تابع زیر را نوشت:

```
PRINT-STATIONS(L,n)
i ← L*
Print "line" i "station" n
for j ← n downto 2
do i ← Li[j], print "line" i "station" j−1
```

خروجی این الگوریتم برای شکل قبل عبارت است از:

```
Line 1 station 5
Line 1 station 4
Line 1 station 3
Line 2 station 2
Line 1 station 1
```

مرتبه هر دو تابع فوق $\theta(n)$ است.

تمرین: جداول $L_i[j]$ و $f_i[j]$ جمعاً شامل $4n-2$ خانه هستند. نشان دهید که چگونه می‌توان فضا را به $2n+2$ خانه کاهش داد به شرطی که بتوان f^* را محاسبه کرد و همچنین مسیر را چاپ کرد.

۹- خرد کردن سکه

دیدیم که این مسئله با روش حریصانه لزوماً جواب درستی نمی‌دهد، ولی می‌توان با برنامه‌نویسی پویا، این مسئله را حل کرد. مثلاً فرض کنید سکه‌های موجود ۱ و ۴ و۶ واحدی هستند و باقیمانده پولی که باید پرداخت شود ۸ واحد است. با روش حریصانه، یک سکه ۶ واحدی و دو سکه ۱ واحدی یعنی جمعاً ۳

سکه پرداخت می‌شود حال آنکه جواب درست ۲ سکه ۴ واحدی است که با روش پویا می‌توان به جواب درست دست یافت.

فرض کنید n نوع سکه مختلف با ارزش‌های $d_1, d_2, \ldots, d_n$ موجود است (و از هر نوع بی‌شمار سکه) و باقیمانده پولی که باید پرداخت شود M است. برای حل مسئله جدول $c[1..n, 0..M]$ در نظر می‌گیریم که درایه $c[i, j]$ برابر تعداد سکه‌های پرداختی است به طوری که جمع مبلغ آنها برابر j باشد و فقط از سکه‌های ۱ تا i (با ارزش‌های $d_1, \ldots, d_i$) استفاده شود. پس جواب $c[n, M]$ است که برابر تعداد سکه‌های پرداختی به مشتری است.

به راحتی می‌توان تساوی زیر را نشان داد:

$$c[i, j] = \min(c[i-1, j], 1 + c[i, j - d_i])$$

زیرا دو حالت وجود دارد، اگر از سکه‌های i استفاده نشود آنگاه $c[i, j] = c[i-1, j]$ و اگر یک سکه i پرداخت شود آنگاه $c[i, j] = 1 + c[i, j - d_i]$ که باید بین این دو مقدار، می‌نیمم انتخاب شود.

اگر $i = 1$ آنگاه $c[0, j]$ تعریف نشده است که برابر $+\infty$ تعریف می‌کنیم به همین ترتیب اگر $j < d_i$ آنگاه $c[i, j - d_i]$ که تعریف نشده است برابر $+\infty$ تعریف می‌شود.

فرض کنید سکه‌های موجود $d_1 = 1$ و $d_2 = 4$ و $d_3 = 6$ واحدی هستند و باقیمانده پولی که باید پرداخت شود $M = 8$ است. ماتریس $c[1..3, 0..8]$ مطابق شکل زیر مقدار می‌گیرد:

		0	1	2	3	4	5	6	7	8
$d_1 = 1$	1	0	1	2	3	4	5	6	7	8
$d_2 = 4$	2	0	1	2	3	1	2	3	4	2
$d_3 = 6$	3	0	1	2	3	1	2	1	2	2

ستون اول ماتریس c همیشه صفر است و سایر درایه‌ها طبق فرمول گفته شده محاسبه می‌شوند مثلاً

$$c[2,8] = \min(c[2,8], 1 + c[3, 8 - d_2]) = \min(2, 1+2) = 2$$

طبق ماتریس c، جواب ۲ سکه است ($c[3,8]$). الگوریتم به شکل زیر است:

```
d[1..n]
c[1..n, 0..M]
for i ← 1 to n do c[i, 0] ← 0
for i ← 1 to n do
    for j ← 1 to M do
        if i = 1 and j < d[i] then c[i, j] ← +∞
        else if i = 1 then c[i, j] ← 1 + c[1, j - d[1]]
        else if j < d[i] then c[i, j] ← c[i - 1, j]
        else c[i, j] ← min(c[i - 1, j], 1 + c[i, j - d[i]])
return c[n, M]
```

مرتبه این الگوریتم $\theta(nM)$ است.

الگوریتم فوق فقط تعداد سکه پرداختی رامشخص می‌کند. برای تشخیص خود سکه‌ها می‌توان از ماتریس c کمک گرفت برای این منظور از $c[n,M]$ (در این مثال از $c[۳,۸]$) شروع می‌کنیم. اگر $c[n,M]=c[n-۱,M]$ یعنی از سکه n ام هیچ پرداخت نشده و به $c[n-۱,M]$ حرکت می‌کنیم. و اگر $c[n,M]=۱+c[n,M-d_n]$ یعنی یک سکه از نوع n ام پرداخت شده که ارزش آن d_n است و به چپ و به $c[n,M-d_n]$ حرکت می‌کنیم برای بررسی بعدی تا در نهایت به $c[۰,۰]$ برسیم.

الگوریتمی که این عملیات را انجام دهد از مرتبه $\theta(n+c[n,M])$ است.

در این مثال $c[۳,۸]=۲$ که با $c[۲,۸]$ مساوی است پس از سکه $d_۲=۶$ پرداخت نشده، حال $c[۲,۸]=۱+c[۲,۸-۴]=۱+۱=۲$ پس یک سکه $d_۲=۴$ پرداخت شده و به $c[۲,۴]$ منتقل می‌شویم و چون $c[۲,۴]=۱+c[۲,۴-۴]=۱+۰=۱$ پس یک سکه دیگر $d_۲=۴$ پرداخت شده است.