به نام یگانه معبود بخشنده مهربان

طراحی الگوریتم ها

Design and Analysis of Algorithms

گروه مهندسی کامپیوتر، دانشکده فنی و مهندسی، دانشگاه اصفهان

ترم دوم سال تحصیلی ۹۰ – ۹۱

ارائه دهنده : پیمان ادیبی

روش برنامه‌ریزی پویا

Dynamic Programming

مفاهیم

- دیدیم روش تقسیم و حل یک روش بالا به پایین است. لذا برای مسایلی مفید است که بتوان زیرنمونه ها را مستقل از یکدیگر حل نمود. در صورت وابستگی حل زیرنمونه‌ها به یکدیگر، این روش بسیار ناکارآمد خواهد بود، چون منجر به حل مکرر زیرنمونه های یکسان میشود.

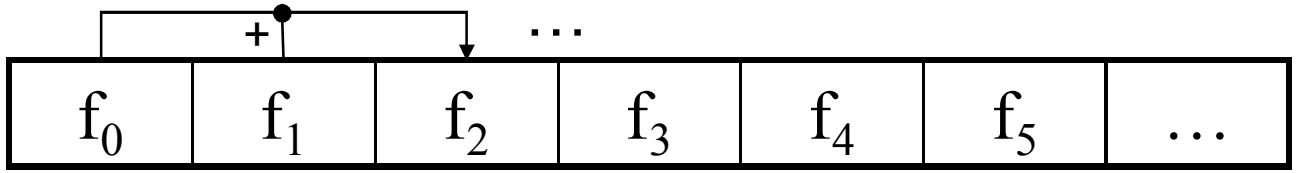
- مثال:
  - مرتب سازی ادغامی: برای آن تقسیم و حل مناسب است (بعلت مستقل بودن حل زیرنمونه ها از یکدیگر)
  - محاسبه جمله n ام فیبوناچی: برای آن تقسیم و حل نامناسب است (بعلت وابستگی حل زیرنمونه ها به یکدیگر)

مفاهیم

- روش برنامه ریزی پویا: در این روش نیز مانند تقسیم و حل یک نمونه مسأله به زیرنمونه‌های کوچکتر تقسیم میشود. اما برای پرهیز از حل مکرر زیرنمونه‌های یکسان، ابتدا نمونه های کوچکتر حل میشوند و نتایج در یک جدول دخیره میشود. در ادامه بجای حل مجدد زیرنمونه‌ها، نتایج از این جدول بازیابی شده و مورد استفاده قرار میگیرند.
- بنابراین روش برنامه ریزی پویا یک روش پایین به بالا (Bottom-Up) است که ابتدا زیرنمونه های کوچکتر را حل میکند، و سپس از پاسخ آنها برای حل زیرنمونه‌های بزرگتر استفاده میکند تا به حل نهایی برسد.

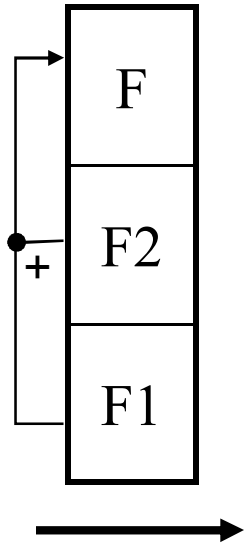
مفاهیم

- بنابراین الگوریتمها در روش برنامه ریزی پویا از نوع تکراری (iterative) هستند نه بازگشتی (recursive)، حتی اگر صورت مسأله یک راه حل بازگشتی را به ذهن آورد.
- در اغلب موارد پس از طراحی الگوریتم با استفاده از جدول، میتوان الگوریتم را بگونه ای اصلاح نمود که به بخش زیادی از فضای جدول نیاز نداشته باشد.
- مثال: الگوریتم تکراری محاسبه جمله n ام فیبوناچی یک نمونه از روش برنامه ریزی پویا محسوب میشود:



الگوریتم تکراری محاسبهٔ جملهٔ n ام فیبوناچی با کمترین حافظه مصرفی

```
int fib2 (int n)
{
    index i ;
    int f[0..n];
    f[0]=0;
    if (n>0){
      f[1]=1;
      for(i=2; i<=n; i++)
        f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```




```
int fib3 (int n)
{
    index i ;
    int F1= 0, F2 = 1, F;
    if (n==0 || n==1)
      return n;
    for(i=2; i<=n; i++){
      F = F1 + F2;
      F1 = F2;  F2=F;
    }
    return F;
}
```

مراحل

- روش برنامه ریزی پویا اغلب برای طراحی الگوریتمهایی جهت حل مسایل بهینه سازی (optimization) که اصل بهینگی (principle of optimality) در آنها صدق میکند، کاربرد دارد (یافتن دنباله بهینه‌ای از تصمیم‌گیریها).
- مراحل کلی ساخت یک الگوریتم برنامه ریزی پویا :
  - ارائه یک ویژگی بازگشتی که حل (بهینه) نمونه مسأله را بدست میدهد
  - محاسبه یک پاسخ (بهینه) برای نمونه مسأله بروش پایین به بالا، یعنی با حل نمونه های کوچکتر در آغاز
  - تشکیل یک پاسخ (بهینه) برای نمونه مسأله بروش پایین به بالا، در صورت نیاز

(مرحله آخر و موارد داخل پرانتز تنها برای مسایل بهینه سازی مطرح هستند)

مثال: محاسبه ضریب دوجمله‌ای

- بسط دو جمله ای: $(a+b)^n$
- ضریب دوجمله ای:

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

- برای n بزرگ استفاده مستقیم از رابطه فوق ناممکن است.
- حذف فاکتوریل از محاسبه ضریب دو جمله ای:

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

- الگوریتم تقسیم و حل برای محاسبه ضریب دو جمله ای:

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

تعداد جملات محاسبه شونده: $2\binom{n}{k} - 1$

بسیار ناکارآمد بدلیل انجام محاسبات افزونه

محاسبه ضریب دوجمله‌ای

- استفاده از روش برنامه ریزی پویا:
  یک آرایه دو بعدی **B** در نظر میگیریم بنحویکه **B[i][j]** حاوی $\binom{i}{j}$ باشد.
- مراحل:
  - ارائه یک ویژگی بازگشتی برای حل یک نمونه مسأله

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

  - یافتن پاسخ بروش پایین به بالا: سطرهای **B** به ترتیب با شروع از سطر اول محاسبه شود:

	0	1	2	3	4		j	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
i								
n								

$B[i-1][j-1]$ $B[i-1][j]$

$B[i][j]$

  - جواب نهایی: **B[n][k]**

الگوریتم برنامه ریزی پویا برای محاسبه ضریب دوجمله‌ای

- مساله: محاسبه ضریب دوجمله ای
- ورودی ها: اعداد صحیح مثبت n، و k که k<n
- خروجی ها: ضریب دوجمله ای bin2 ← $\binom{n}{k}$

```
int bin2 (int n, int k)
{
  index i, j;
  int B[0..n] [0..k];

  for (i = 0; i <= n; i++)
       for (j = 0; j <= 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];
}
```

تحلیل الگوریتم برنامه ریزی پویا برای محاسبه ضریب دوجمله‌ای

- عمل اصلی: مقایسه داخل حلقه j
- اندازه ورودی: تعداد بیت مورد نیاز برای ذخیره n و k
- تحلیل در هر حالت ؟ دارد ← T(k,n)=

$$1+2+3+4+\cdots+k+\underbrace{(k+1)+(k+1)\cdots+(k+1)}_{n-k+1 \text{ times}} =$$

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

```
int bin2 (int n, int k)
{
  index i, j;
  int B[0..n] [0..k];

  for (i = 0; i <= n; i++)
       for (j = 0; j <= 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];
}
```

اصلاح الگوریتم برنامه ریزی پویا برای محاسبه ضریب دوجمله‌ای

- اصلاح اول: پس از محاسبه یک سطر از B دیگر نیازی به مقادیر ذخیره شده در سطر قبل نداریم. پس میتوان مصرف حافظه را با نوشتن الگوریتم با یک آرایه تک بعدی دارای اندیسهای 0تا k کاهش داد (تمرین)

- اصلاح دوم: استفاده از رابطه زیر:

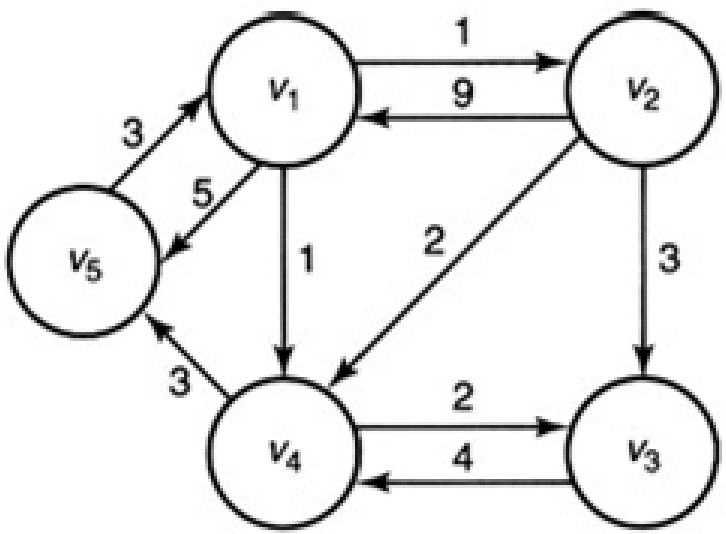
$$\binom{n}{k} = \binom{n}{n-k}$$

```
int bin2 (int n, int k)
{
  index i, j;
  int B[0..n] [0..k];

  for (i = 0; i <= n; i++)
       for (j = 0; j <= 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];
}
```

مثال: یافتن تمام کوتاهترین مسیرها در گراف بروش فلوید (Floyd-Warshall Algorithm)

- در یک گراف وزن دار جهت دار دارای n رأس، میخواهیم کوتاهترین مسیر (shortest path) از هر رأس به سایر رأسها را بدست آوریم.



- مسیرها از v_1 به v_3:

طول مسیر	نوع مسیر	مسیر از v_1 به v_3
1+3=4	ساده	v_1, v_2, v_3
1+2=3	ساده	v_1, v_4, v_3
1+2+2=5	ساده	v_1, v_2, v_4, v_3
1+3+3+1+3=11	دوری	$v_1, v_4, v_5, v_1, v_2, v_3$
…	…	…

کوتاهترین مسیر

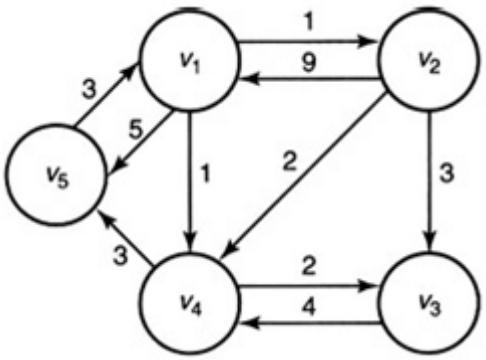
- قطعاً کوتاهترین مسیر یک مسیر ساده (بدون دور) است.

یافتن تمام کوتاهترین مسیرها در گراف

- ممکن است بین دو رأس بیش از یک مسیر با طول کمینه وجود داشته باشد. دراینجا یافتن یکی از این کوتاهترین مسیرها کافیست.
- فرض کنید گراف با ماتریس هم‌جواری (adjacency matrix) W نمایش داده می‌شود:

$$W[i][j] = \begin{cases} w_{ij} \text{ (وزن یال)} & \text{اگر بین i و j یالی وجود داشته باشد} \\ \infty & \text{اگر بین i و j یالی وجود نداشته باشد} \\ 0 & \text{اگر i=j باشد} \end{cases}$$

- مثال:



$$W = \begin{bmatrix} 0 & 1 & \infty & 1 & 5 \\ 9 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

یافتن تمام کوتاهترین مسیرها در گراف

- یک الگوریتم بدیهی:
  - محاسبه طول تمام مسیرهای ساده ممکن و انتخاب بهترین آنها
  - در بدترین حالت از مرتبهٔ بدتر از نمایی است. دلیل:
  - فرض کنید گرافی با اتصالات کامل داریم (یعنی از هر رأس به رأس دیگر یک یال وجو دارد):

تعداد کل مسیرها $>$

تعداد کل مسیرها با یک رأس شروع و پایان مشخص $>$

تعداد کل مسیرها با همان رأسهای شروع و پایان که از همه رأسهای دیگر میگذرند $= (n-2)(n-3)\ldots 1 = (n-2)!$ ← بدتر از نمایی

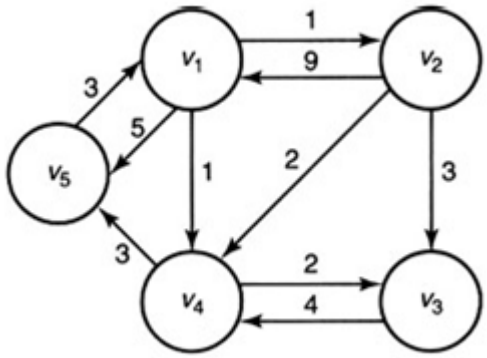
- حل با برنامه ریزی پویا :
  - ابتدا الگوریتمی برای یافتن طول کوتاهترین مسیر ارائه میشود. سپس آنرا بنحوی اصلاح میکنیم که خود کوتاهترین مسیر را هم بدهد.

یافتن کوتاهترین مسیرها در گراف بروش فلوید

- تعداد n+1 آرایه دو بعدی $D^{(k)}$ درنظر میگیریم (k=0,..,n)، که:

$D^{(k)}[i][j]=$ طول کوتاهترین مسیر از v_i به v_j تنها شامل رئوس $\{v_1,\ldots,v_k\}$ بعنوان رئوس میانی

- مثال:



$D^{(0)}[1][3]=\infty$ $D^{(1)}[1][3]=\infty$

$D^{(2)}[1][3]=4$ $D^{(3)}[1][3]=4$

$D^{(4)}[1][3]=\min(3,4,5)=3$

$D^{(5)}[1][3]=3$

- هدف: یافتن D با داشتن W ≡ یافتن $D^{(n)}$ از روی $D^{(0)}$

$D^{(0)} = W$

$D^{(n)} = D$

→ $D^0, D^1, D^2, \ldots, D^n$

پایین به بالا

مراحل برنامه ریزی پویا در الگوریتم فلوید

- **مرحله اول:** ارائه یک ویژگی بازگشتی برای حل یک نمونه مسأله:

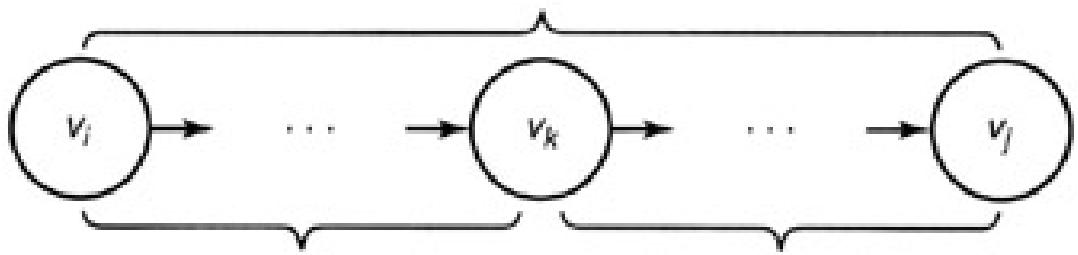
دو حالت در نظر میگیریم:

- **حالت 1 :** اقلاً یک کوتاهترین مسیر از v_i به v_j با رئوس میانی مجاز $[v_1,\ldots,v_k]$ ، از رأس v_k استفاده نمیکند. در این حالت:

$$D^{(k)}[i][j]=D^{(k-1)}[i][j]$$

- **حالت 2 :** همهٔ کوتاهترین مسیرها از v_i به v_j با رئوس میانی مجاز $[v_1,\ldots,v_k]$ ، از رأس v_k استفاده میکنند. در این حالت:

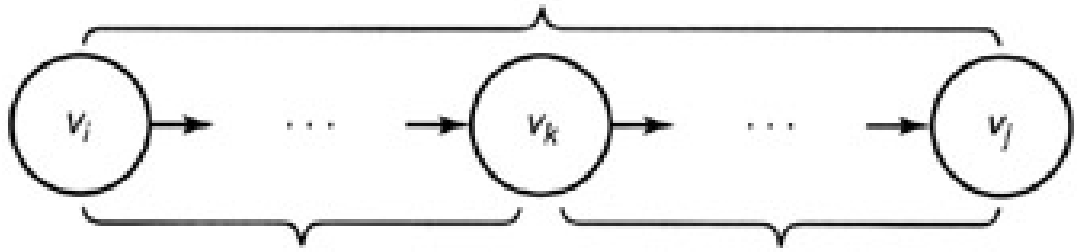
یک کوتاهترین مسیر بطول L متشکل از مسیرهای 1 و 2



مسیر 1 بطول L_1 مسیر 2 بطول L_2

مراحل برنامه ریزی پویا در الگوریتم فلوید

یک کوتاهترین مسیر بطول L متشکل از مسیرهای 1 و 2



مسیر 1 بطول L_1 مسیر 2 بطول L_2

- داریم:

$L_1 = D^{(k-1)}[i][k] \longleftarrow L_1 \not> D^{(k-1)}[i][k]$ و $L_1 \not< D^{(k-1)}[i][k]$

و به همین ترتیب $L_2 = D^{(k-1)}[k][j] \longleftarrow$

- لذا:

$L = L_1+L_2 \longrightarrow D^{(k)}[i][j]=D^{(k-1)}[i][k]+D^{(k-1)}[k][j]$

- بنابراین در حالت کلی (حالت1 یا 2) رابطه بازگشتی زیر را داریم:

$$D^{(k)}[i][j] = \min(\underbrace{D^{(k-1)}[i][j]}_{\text{حالت 1}}, \underbrace{D^{(k-1)}[i][k] + D^{(k-1)}[k][j]}_{\text{حالت 2}})$$

مراحل برنامه ریزی پویا در الگوریتم فلوید

- مرحله دوم: محاسبه پاسخ بهینه به روش پایین به بالا:
  - با افزایش دادن k از 1 تا n عناصر $D^{(k)}$ را از عناصر $D^{(k-1)}$ محاسبه میکنیم. عملاً در الگوریتم تنها به یک ماتریس D نیاز داریم.

- الگوریتم فلوید برای یافتن طول کوتاهترین مسیرها:
  - ورودی‌ها: یک گراف وزندار جهتدار نمایش داده شده با ماتریس همجواری W و n تعداد رئوس آن
  - خروجی‌ها: ماتریس D حاوی طول کوتاهترین مسیرها

```
void floyd (int n ,const number W[] [], number D[] [])
{
 index i, j, k;
 D = W;
 for (k = 1; k<=n; k++)
    for (i = 1; i <= n; i++)
       for (j = 1; j <= n; j++)
         D[i][j] = minimum(D[i][j], D[i][k] + D[k][j]);
}
```

مراحل برنامه ریزی پویا در الگوریتم فلوید

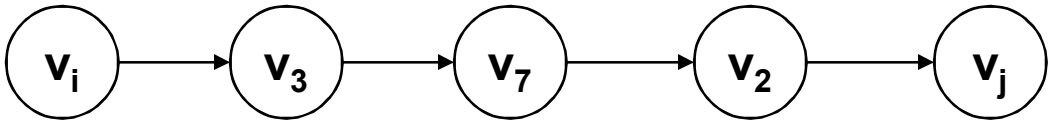
- درباره اصلاح مصرف حافظه: بخاطر آنکه سه خانه موجود در سمت راست انتساب در خط آخر الگوریتم مقادیرشان را از مرحله k-1 ام حفظ کرده اند، لذا نقش خانه‌های $D^{(k-1)}$ را دارند.

- **مرحله سوم: تشکیل پاسخ بهینه به روش پایین به بالا (یافتن خود کوتاهترین مسیرها بصورت دنباله ای از رئوس):**
  - آرایه دوبعدی P را بصورت زیر تعریف میکنیم:

$$P[i][j] = \begin{cases} j^* & \text{اگر رأس میانی در کوتاهترین مسیر از } v_i \text{ به } v_j \text{ موجود باشد} \\ 0 & \text{اگر رأس میانی در کوتاهترین مسیر از } v_i \text{ به } v_j \text{ موجود نباشد} \end{cases}$$

که در آن j^* بزرگترین اندیس رئوس میانی در کوتاهترین مسیر از v_i به v_j است.

- مثال:



$P[i][j] = j^* = 7$

الگوریتم فلوید

- **مساله**: محاسبه و ایجاد کوتاهترین مسیرها در یک گراف وزندار جهتدار
- **ورودی‌ها**: W ماتریس همجواری گراف و n تعداد رئوس آن
- **خروجی‌ها**: ماتریس D حاوی طول کوتاهترین مسیرها و ماتریس P

```
void floyd2 (int n, const number W[] [], number D[] [], index P[] [])
{
 index, i, j, k;
 for(i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
           P[i] [j] = 0;
  D = W;
  for (k = 1; k <= n; k++)
      for(i = 1; i <= n; i++)
         for(j = 1; j <= n; j++)
             if (D[i][k] + D[k][j] < D[i][j]){
                P[i][j] = k;
                D[i][j] = D[i][k] + D[k][j];
              }
}
```

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

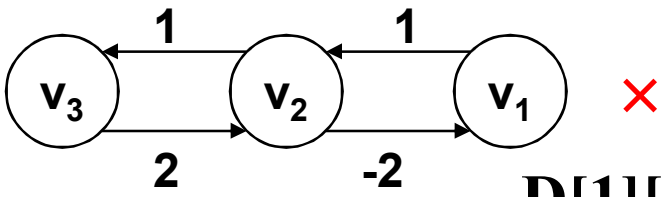
چاپ کوتاهترین مسیر حاصل از الگوریتم فلوید

- **مساله**: چاپ رئوس میانی کوتاهترین مسیر از رأس v_q به v_r
- **ورودی‌ها**: ماتریس P حاصل از الگوریتم floyd2
- **خروجی‌ها**: رئوس میانی مذکور در فوق

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

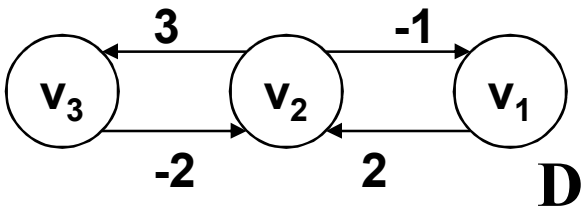
$$W(n) \in \theta(n)$$

- **نکته**: اگر وزنهای منفی در گراف وجود داشته باشد، تنها در صورتی الگوریتم فلوید جواب میدهد که دور با وزن منفی نداشته باشیم. مثال:



×

D[1][3] = -∞

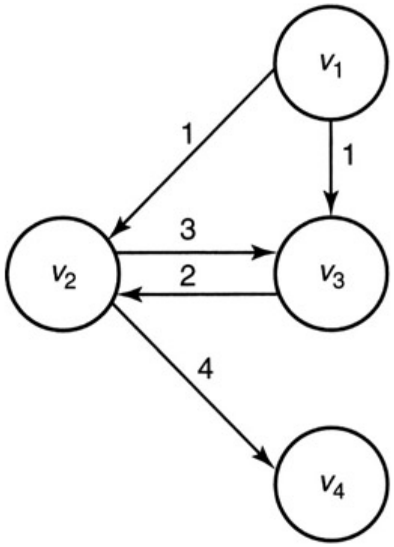


√

D[1][3] = 5

برنامه ریزی پویا در مسائل بهینه سازی

- از میان مسائل بهینه سازی، آنهایی را میتوان با برنامه ریزی پویا حل کرد که اصل بهینگی برایشان برقرار باشد.
- اصل بهینگی (principle of optimality): اگر یک پاسخ بهینه برای یک نمونه مسأله در بر دارنده پاسخ بهینه برای تمام زیر نمونه های آن مسأله باشد، اصل بهینگی برای آن مسأله صدق میکند.
- مثال: مسأله یافتن کوتاهترین مسیر در گراف:
  - اصل بهینگی برای این مسأله صدق می‌کند، زیرا هر زیر مسیر از یک کوتاهترین مسیر، خود کوتاهترین مسیر است.
- مثال: مسأله یافتن طولانی ترین مسیر ساده در گراف:
  - اصل بهینگی برای این مسأله صدق نمی‌کند:



Longest path from v_1 to v_4 = $[v_1 v_3 v_2 v_4]$

Longest path from v_1 to v_3 = $[v_1 v_2 v_3] \neq [v_1 v_3]$

مثال: مسأله ضرب بهینه زنجیره ای از ماتریسها (Chained Matrix Multiplication)

- میدانیم ضرب ماتریسها خاصیت جابجایی ندارد، اما خاصیت شرکت پذیری دارد: **A(BC)=(AB)C=ABC**
- تعداد ضرب اسکالر برای ضرب یک ماتریس **m×p** در یک ماتریس **p×r** برابر است با : **m×p×r**

دلیل:

$$A_{m\times p}\times B_{p\times r}=C_{m\times r} \longrightarrow c_{ij}=\sum_{l=1}^{p} a_{il}b_{lj}$$

- سئوال: در یک زنجیره از ضربهای ماتریسی، با چه ترتیبی ضربها را انجام دهیم که سریعتر باشد (کمترین تعداد ضرب اسکالر را نیاز داشته باشد)؟ به عبارت دیگر، پرانتز گذاری بهینه چیست؟

ضرب بهینه زنجیره ای از ماتریسها

- مثال عددی: $A_{5\times2} \times B_{2\times4} \times C_{4\times3} \times D_{3\times1}$

برای ضرب چهار ماتریس پنج ترتیب مختلف وجود دارد:

A (B (C D)) → تعداد ضرب اسکالر = $4\times3\times1+2\times4\times1+5\times2\times1 = \boxed{30}$

(A B) (C D) → تعداد ضرب اسکالر = ... = 72

A ((B C) D) → تعداد ضرب اسکالر = ... = 40

((A B) C) D → تعداد ضرب اسکالر = ... = 115

(A (B C)) D → تعداد ضرب اسکالر = ... = 69

ترتیب بهینه

- میخواهیم ترتیب بهینه ضرب n ماتریس با ابعاد داده شده را بدست آوریم : $A_1 A_2 \ldots A_n$
- الگوریتم بدیهی:
  - بررسی تمام ترتیبهای ممکن
  - زمان اجرای آن حداقل نمایی است ← دلیل: (صفحه بعد)

تعداد کل ترتیبهای ممکن ضرب زنجیره‌ای ماتریسها

تعداد کل ترتیب ها	$\geq$	تعداد ترتیبها با A_1 بعنوان آخرین ماتریس ضرب شونده	$+$	تعداد ترتیبها با A_n بعنوان آخرین ماتریس ضرب شونده
		$A_1\underbrace{(A_2\ldots A_n)}_{\text{تعداد ترتیبها}}$		$\underbrace{(A_1\ldots A_{n-1})}_{\text{تعداد ترتیبها}}A_n$
t_n	$\geq$	t_{n-1}	$+$	t_{n-1}

$$t_n \geq 2t_{n-1}$$
$$t_2 = 1$$

حل رابطه بازگشتی $\longrightarrow$ $t_n \geq 2^{n-2}$ $\longrightarrow$ حداقل نمایی

- تعداد کل ترتیبها بطور دقیق:

$$\underbrace{(A_1\ldots A_k)}_{\text{تعداد ترتیبها}\ t_k}\underbrace{(A_{k+1}\ldots A_n)}_{\text{تعداد ترتیبها}\ t_{n-k}}$$

محل آخرین ضرب

$$\longrightarrow \quad t_n = \sum_{k=1}^{n-1} t_k t_{n-k}, \quad t_1 = t_2 = 1$$

تعداد کل ترتیبهای ممکن ضرب زنجیره‌ای ماتریسها

- تعداد کل ترتیبها:

$$t_n = \sum_{k=1}^{n-1} t_k t_{n-k}, \quad t_1 = t_2 = 1 \xrightarrow[\text{دنباله کاتالان}]{\text{حل با استفاده از}} t_n = \frac{1}{n}\binom{2n-2}{n-1}$$

دلیل:

عدد n+1 ام کاتالان : $C_{n+1} \overset{\Delta}{=} \sum_{i=0}^{n} C_i C_{n-i}, \quad C_0 = 1 \xrightarrow{\text{ثابت میشود}} C_n = \frac{1}{n+1}\binom{2n}{n}$

عدد n ام کاتالان (n'th Catalan number)

$$\xrightarrow{\text{و نیز داریم}} C_n \underset{n\to\infty}{\to} \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

میتوان نشان داد $t_n = C_{n-1}$ ...

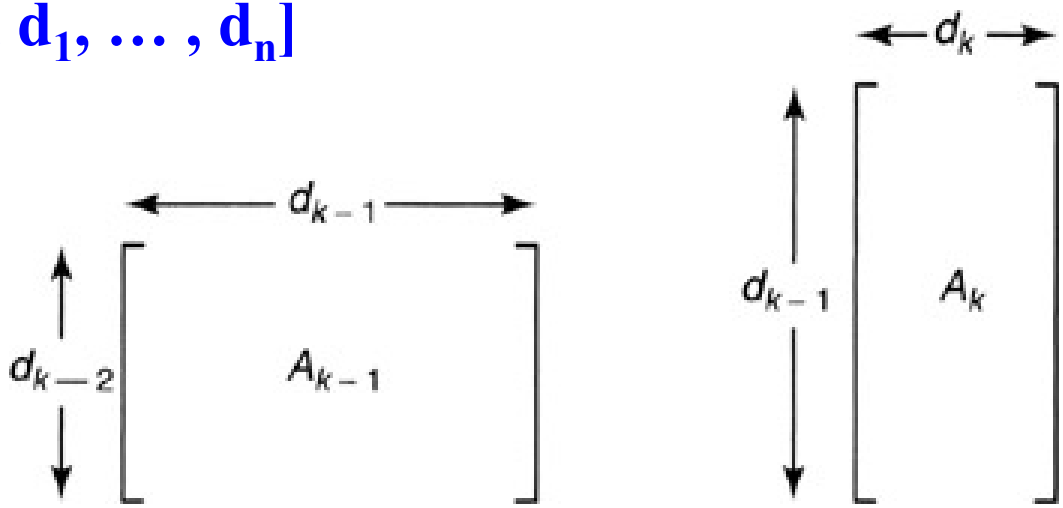
- اصل بهینگی برای این مسأله برقرار است،
چون در یک ترتیب بهینه تمام زیرترتیبها نیز باید بهینه باشند. مثال:
ترتیب بهینه: $((((A_1A_2)(A_3(A_4A_5)))A_6)A_7)(A_8A_9)$

زیر ترتیب های بهینه

برنامه ریزی پویا برای ضرب زنجیره‌ای ماتریسها

- ابعاد ماتریسها با d_0 تا d_n نمایش داده میشوند، بنحویکه ماتریس A_k دارای ابعاد $d_{k-1} \times d_k$ است $(k=1,\ldots,n)$:

$d=[d_0, d_1, \ldots, d_n]$



- ابتدا آرایه دوبعدی M که حل در آن بنا میشود را بصورت زیر تعریف میکنیم:

$$M[i][j] = \begin{cases} \text{حداقل تعداد ضرب اسکالر برای انجام ضرب زنجیره ای از } A_i \text{ تا } A_j & i < j \\ 0 & i = j \end{cases}$$

برنامه ریزی پویا برای ضرب زنجیره‌ای ماتریسها

- اگر فرض کنیم آخرین ضرب در زنجیره $A_i \ldots A_j$ بین دو بخش زیر انجام شود:

$$\underbrace{(A_i \ldots A_k)}_{\text{بخش 1}}\underbrace{(A_{k+1} \ldots A_j)}_{\text{بخش 2}}$$

- آنگاه:

$$M[i][j] = \min_k \left\{ \begin{array}{c}\text{حداقل تعداد ضرب اسکالر برای} \\ \text{محاسبه دو بخش 1 و 2}\end{array} + \begin{array}{c}\text{تعداد ضرب اسکالر برای ضرب} \\ \text{بخش 1 در بخش 2}\end{array} \right\}$$

- لذا ویژگی بازگشتی زیر را خواهیم داشت:

$$M[i][j] = \begin{cases} \min\limits_{i \le k \le j-1} (M[i][k] + M[k+1][j] + d_{i-1} d_k d_j) & i < j \\ 0 & i = j \end{cases}$$

- از رابطه فوق مشاهده میشود که مثلث بالایی ماتریس **M** باید محاسبه شود. ابتدا قطر اصلی را صفر میگذاریم. سپس عناصر قطر فرعی اول را حساب میکنیم. اینکار برای قطرهای فرعی بعد تا رسیدن به پاسخ (**M[1][n]**) ادامه می یابد.

برنامه ریزی پویا برای ضرب زنجیره‌ای ماتریسها

- مثال عددی:

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

- قطر 0 :

$$M[i][i] = 0 \qquad \text{for } 1 \le i \le 6.$$

- قطر 1 :

$$\begin{aligned} M[1][2] &= \underset{1 \le k \le 1}{minimum}(M[1][k] + M[k+1][2] + d_0 d_k d_2) \\ &= M[1][1] + M[2][2] + d_0 d_1 d_2 \\ &= 0 + 0 + 5 \times 2 \times 3 = 30. \end{aligned}$$

$\longrightarrow$ M [2] [3], M [3] [4], M [4] [5], and M [5] [6]

- قطر 2 :

$$\begin{aligned} M[1][3] &= \underset{1 \le k \le 2}{minimum}(M[1][k] + M[k+1][3] + d_0 d_k d_3) \\ &= minimum(M[1][1] + M[2][3] + d_0 d_1 d_3, \\ &\qquad M[1][2] + M[3][3] + d_0 d_2 d_3) \\ &= minimum(0 + 24 + 5 \times 2 \times 4, 30 + 0 + 5 \times 3 \times 4) = 64. \end{aligned}$$

M [2] [4], M [3] [5], and M [4] [6]

برنامه ریزی پویا برای ضرب زنجیره‌ای ماتریسها

- قطر 3 :

$$M[1][4] = \underset{1 \le k \le 3}{minimum}(M[1][k] + M[k+1][4] + d_0 d_k d_4)$$

$$= minimum(M[1][1] + M[2][4] + d_0 d_1 d_4,$$
$$M[1][2] + M[3][4] + d_0 d_2 d_4)$$
$$M[1][3] + M[4][4] + d_0 d_3 d_4)$$

M [2] [5] and M [3] [6]

$$= minimum(0 + 72 + 5 \times 2 \times 6, 30 + 72 + 5 \times 3 \times 6,$$
$$64 + 0 + 5 \times 4 \times 6) = 132.$$

- قطر 4 : ...
- قطر 5 : پاسخ نهایی: $M\,[1]\,[6] = 348.$
- تمرین:

A_1	$\times$	A_2	$\times$	A_3	$\times$	A_4	$\times$	A_5
6×2		2×5		5×4		4×8		8×7

برنامه ریزی پویا برای ضرب زنجیره‌ای ماتریسها

- ماتریس حاصل :

پاسخ نهایی

	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

Diagonal 1 Diagonal 2 Diagonal 3 Diagonal 4 Diagonal 5

- برای یافتن ترتیب بهینه ، آرایه دوبعدی P بشکل زیر تعریف میشود:

$$P[i][j]=k$$

که k محل بهینه شکستن زنجیره $A_i \ldots A_j$ به دو بخش میباشد:

$$(A_i \ldots A_k)(A_{k+1} \ldots A_j)$$

برنامه ریزی پویا برای ضرب زنجیره‌ای ماتریسها

- مساله: تعیین حداقل ضرب اسکالر برای ضرب **n** ماتریس و تعیین ترتیب بهینه
- ورودی ها: عدد صحیح مثبت **n**، و آرایه **d** حاوی ابعاد ماتریسها
- خروجی ها: **minmult** حداقل تعداد ضرب، و ماتریس **P** برای تعیین ترتیب بهینه

```
int minmult (int n, const int d [], index P [] [] )
{
   index i, j, k, diagonal;
   int M [1 .. n] [1 .. n];

   for (i = 1; i < = n; i++)
          M[i] [i] = 0;
   for (diagonal = 1; diagonal < = n - 1; diagonal ++)     // Diagonal-1 is
         for (i = 1; i <= n - diagonal; i++) {     // just above the
            j = i + diagonal;     // main diagonal
            M[i] [j] =
                     minimum (M[i] [k] + M[k + 1] [j] + d[i - 1]* d[k] * d[j]);
                     i ≤ k ≤ j - 1
            P[i] [j] = a value of k that gave the minimum;
       }
  return M[1] [n];
}
```

مثال: مسأله فروشنده دوره گرد (Traveling Salesman Problem - TSP)

- این مسأله عبارت از یافتن کوتاهترین مدار همیلتونی (تور بهینه) در یک گراف وزن دار جهت دار است .
- مدار همیلتونی یا تور: یک دور در گراف که از هر رأس دقیقاً یک بار میگذرد.
- رأس شروع را v_1 مینامیم.
- الگوریتم بدیهی: بررسی تمام تورهای ممکن ← در بدترین حالت از مرتبهٔ فاکتوریل است. ← $(n-1)(n-2)\cdots 1 = (n-1)!$
- آیا اصل بهینگی برقرار است؟

اگر v_k اولین رأس پس از مبدأ (v_1) در یک تور بهینه باشد، زیرمسیر از v_k تا v_1 باید کوتاهترین مسیر از v_k تا v_1 با عبور از تمام رئوس میانی این زیر مسیر باشد ← اصل بهینگی برقرار است.

برنامه ریزی پویا برای مسأله فروشنده دوره گرد

- گراف $G = (V , E)$ را با ماتریس همجواری **W** نمایش میدهیم.
- آرایه **D** را بصورت زیر تعریف میکنیم (با فرض $A \subset V$):

$D[v_i][A] =$ طول کوتاهترین مسیر از v_i به v_1 که از تمام رئوس **A** دقیقاً یکبار میگذرد

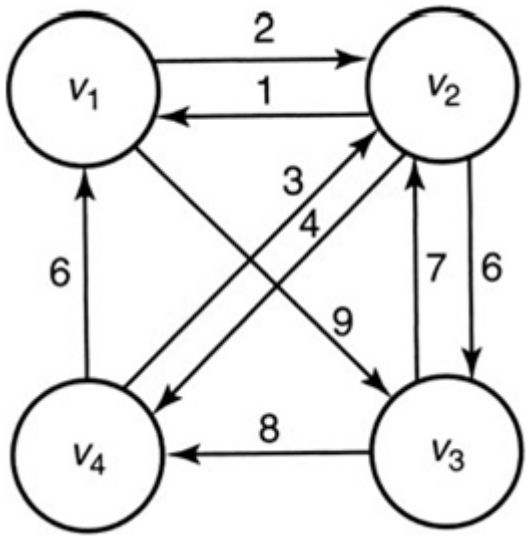
- رابطه بازگشتی (با فرض $v_1, v_i \notin A$ و $i \neq 1$):

$$D[v_i][A] = \underset{j:v_j \in A}{minimum}(W[i][j] + D[v_j][A - \{v_j\}]) if\, A \neq \varnothing$$

$$D[v_i][\varnothing] = W[i][1]. \qquad (3.7)$$

- مقدار پاسخ نهایی = طول تور بهینه = $D[v_1][V-\{v_1\}]$
- برای تشکیل پاسخ بهینه، یک آرایه **P** مشابه قبل تعریف میشود: ...

مثال عددی: برنامه ریزی پویا برای مسأله TSP



W=

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

- روش پایین به بالا:
  - ابتدا برای مجموعه تهی (|A|=0):

$$D[v_2][\varnothing] = 1$$
$$D[v_3][\varnothing] = \infty$$
$$D[v_4][\varnothing] = 6$$

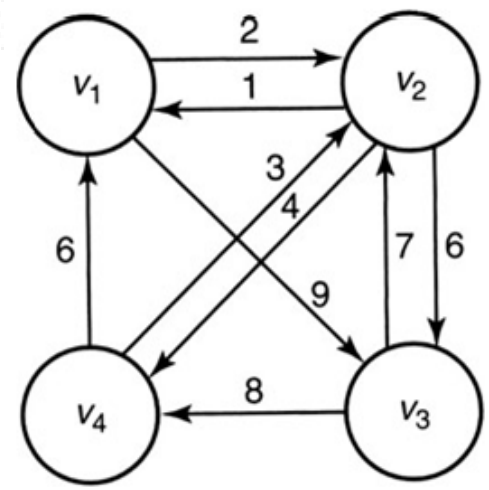
مثال عددی: برنامه ریزی پویا برای مسأله TSP

- سپس برای مجموعه های حاوی یک رأس (|A|=1):

$$D[v_3][\{v_2\}] = \underset{j:v_j \in \{v_2\}}{minimum}(W[3][j] + D[v_j][\{v_2\} - \{v_j\}])$$
$$= W[3][2] + D[v_2][\varnothing] = 7 + 1 = 8$$

Similarly,

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



W=

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

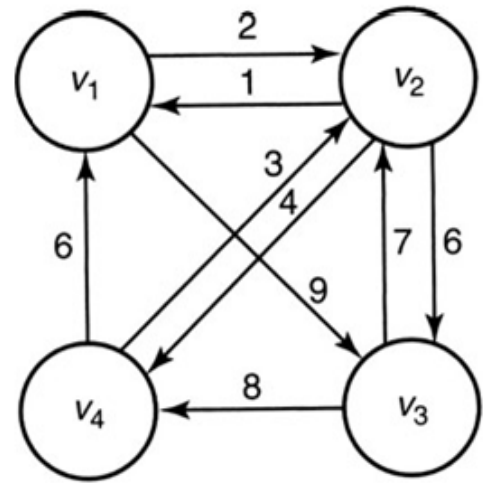
مثال عددی: برنامه ریزی پویا برای مسأله TSP

- برای مجموعه های حاوی دو رأس (|A|=2):

$$D[v_4][\{v_2, v_3\}] = \underset{j:v_j \in \{v_2,v_3\}}{minimum}(W[4][j] + D[v_j][\{v_2, v_3\} - \{v_j\}])$$
$$= minimum(W[4][2] + D[v_2][\{v_3\}], W[4][3] + D[v_3][\{v_2\}])$$
$$= minimum(3 + \infty, \infty + 8) = \infty$$

Similarly,

$$D[v_3][\{v_2, v_4\}] = minimum(7 + 10, 8 + 4) = 12$$
$$D[v_2][\{v_3, v_4\}] = minimum(6 + 14, 4 + \infty) = 20$$



W=

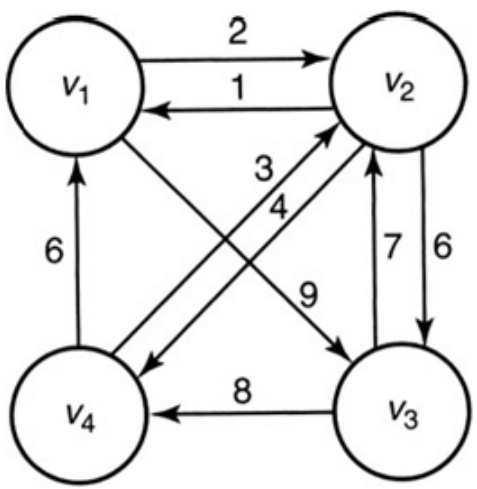
	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

مثال عددی: برنامه ریزی پویا برای مسأله TSP

- پاسخ (|A|=3):

$$D[v_1][\{v_2,v_3,v_4\}] = \underset{j:v_j\in\{v_2,v_3,v_4\}}{minimum}(W[1][j] + D[v_j][\{v_2,v_3,v_4\} - \{v_j\}])$$

$$\begin{aligned} &= minimum(W[1][2] + D[v_2][\{v_3,v_4\}], \\ &\qquad W[1][3] + D[v_3][\{v_2,v_4\}], \\ &\qquad W[1][4] + D[v_4][\{v_2,v_3\}]) \\ &= minimum(2+20, 9+12, \infty+\infty) = 21 \end{aligned}$$



W=

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

مثال عددی: برنامه ریزی پویا برای مسأله TSP

- تشکیل پاسخ :

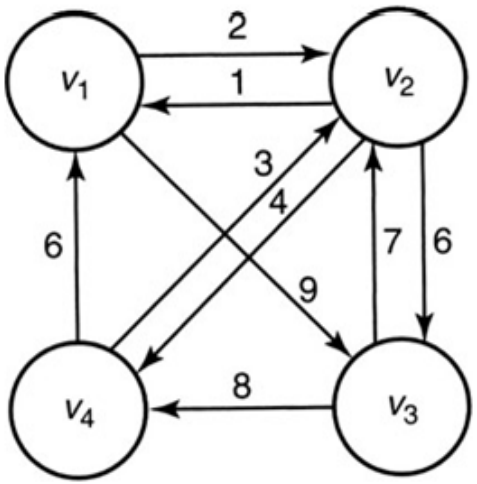
3	4	2
$P[1, \{v_2, v_3, v_4\}]$	$P[3, \{v_2, v_4\}]$	$P[4, \{v_2\}]$

We obtain an optimal tour as follows:

$$\text{Index of first node} = P\ [1][\{v_2, v_3, v_4\}] = 3$$

$$\text{Index of second node} = P\ [3][\{v_2, v_4\}] = 4$$

$$\text{Index of third node} = P\ [4][\{v_2\}] = 2$$



The optimal tour is, therefore,

$$[v_1, v_3, v_4, v_2, v_1].$$

تحلیل الگوریتم برنامه ریزی پویا برای TSP

- عمل اصلی:
  مقایسه برای یافتن
  مینیمم در حلقه j
- اندازه ورودی:
  تعداد رأسهای گراف (n)

```
void travel (int n,
                          const number W [] [],
                          index P [] [],
                          number & minlength)
{
   index i, j, k;
   number D [1 .. n] [subset of V - {v1}];

   for (i = 2; i <= n; i++)
         D [i] [∅] = W[i] [1];
   for (k = 1; k <= n - 2; k++)
               for (all subsets A ⊆ V - {v1} containing k vertices)
                         for (i such that i ≠ 1 and vi is not in A){
                           D [i] [A] = minimum (W [i] [j] + D [j][A - {vj}]);
                                     j:  vj ∈ A
                           P[i] [A] = value of j that gave the minimum;
         }
    D [1] [V - {v1}] = minimum (W[1] [j] + D[j] [V - {v1, vj}]);
                   2 ≤ j ≤ n
    P[1] [V - {v1}] = value of j that gave the minimum;
    minlength = D[1] [V - {v1}];
}
```

تحلیل الگوریتم برنامه ریزی پویا برای TSP

- زمان اجرا (در هر حالت):

$$T(n) = \sum_{k=1}^{n-2} (n-1-k)\, k \binom{n-1}{k}.$$

```
void travel (int n,
                          const number W [] [],
                          index P [] [],
                          number & minlength)
{
   index i, j, k;
   number D [1 .. n] [subset of V - {v1}];

   for (i = 2; i <= n; i++)
         D [i] [∅] = W[i] [1];
   for (k = 1; k <= n - 2; k++)
               for (all subsets A ⊆ V - {v1} containing k vertices)
                        for (i such that i ≠ 1 and vi is not in A){
                          D [i] [A] = minimum (W [i] [j] + D [j][A - {vj}]);
                                   j:  vj ∈ A
                          P[i] [A] = value of j that gave the minimum;
         }
    D [1] [V - {v1}] = minimum (W[1] [j] + D[j] [V - {v1, vj}]);
                   2 ≤ j ≤ n
    P[1] [V - {v1}] = value of j that gave the minimum;
    minlength = D[1] [V - {v1}];
}
```

تحلیل الگوریتم برنامه ریزی پویا برای TSP

- زمان اجرا (در هر حالت):

$$T(n) = \sum_{k=1}^{n-2} (n-1-k)\, k \binom{n-1}{k}.$$

$$(n-1-k)\binom{n-1}{k} = (n-1)\binom{n-2}{k}.$$

$$T(n) = (n-1)\sum_{k=1}^{n-2} k \binom{n-2}{k}.$$

$$\sum_{k=1}^{n} k \binom{n}{k} = n2^{n-1}.$$

قضیه 3.1 کتاب :

$$T(n) = (n-1)(n-2)\,2^{n-3} \in \Theta\left(n^2 2^n\right).$$

مساله NP کامل است

تحلیل الگوریتم برنامه ریزی پویا برای TSP

- تحلیل پیچیدگی حافظه:

$$M(n) = 2 \times n2^{n-1} = n2^n \in \Theta(n2^n).$$

- مقایسه با الگوریتم بدیهی با n=20 روی یک سیستم مشخص:
  - زمان اجرا :

Brute-force algorithm: $19!\ \mu s = 3857$ years

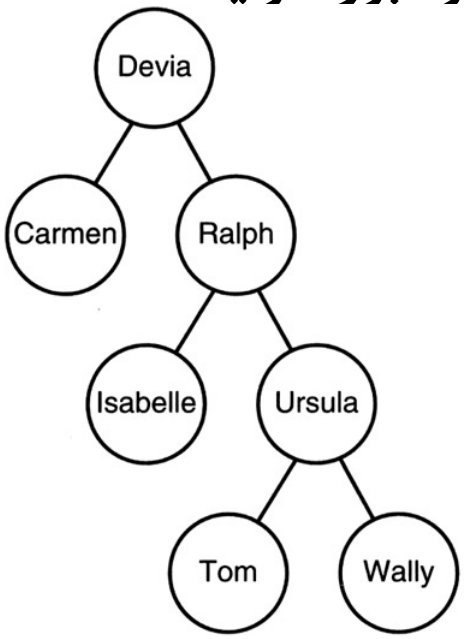
Dynamic programming algorithm: $(20-1)(20-2)\,2^{20-3}\ \mu s = 45$ seconds.

  - حافظه مورد نیاز :

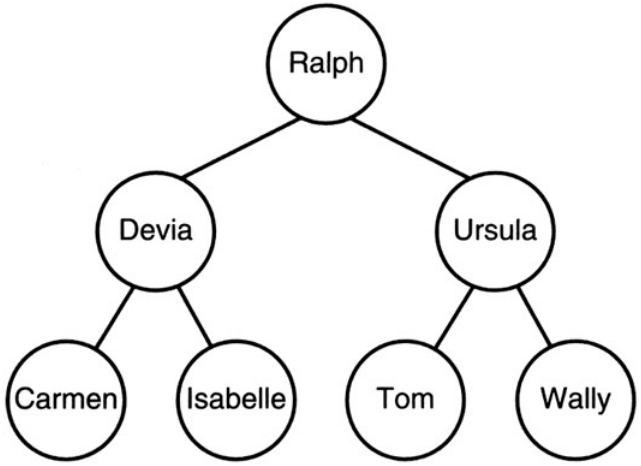
$$20 \times 2^{20} = 20,971,520 \text{ array slots.}$$

مثال: درخت جستجوی دودویی بهینه (Optimal Binary Search Tree - OBST)

- تعاریف اولیه:
  - زیر درخت چپ (راست) یک گره از درخت دودویی: درختی که ریشه آن فرزند چپ (راست) آن گره است.
  - درخت جستجوی دودویی: یک درخت دودویی از کلیدهای متعلق به یک مجموعه دارای ترتیب که کلیدهای زیر درخت چپ یک گره کوچکتر یا مساوی با کلید آن گره، و کلیدهای زیر درخت راست آن گره بزرگتر یا مساوی با کلید آن گره هستند.
  - عمق (سطح) یک گره: تعداد یالهای مسیر ریشه تا آن گره
  - عمق درخت: حداکثر عمق همه گره های آن درخت.
  - درخت متوازن: درختی که عمق دو زیر درخت از هر گره آن بیش از 1 واحد اختلاف نداشته باشد.



نامتوازن با عمق 3



متوازن با عمق 2

درخت جستجوی دودویی بهینه

- **تعریف درخت جستجوی دودویی بهینه:** یک درخت جستجوی دودویی که میانگین زمان جستجوی کلیدها در آن کمینه باشد.
- فرضها:
  - هر کلید یک احتمال وقوع دارد که میتواند با احتمال سایر کلیدها متفاوت باشد (در صورت برابری احتمالات، درخت بهینه درخت متوازن خواهد بود).
  - کلید مورد جستجو در درخت وجود دارد.
- ساختمان داده مورد استفاده برای گره‌ها:

```
struct nodetype
{
  keytype key;
  nodetype* left;
  nodetype* right;
};
typedef nodetype* node_pointer;
```

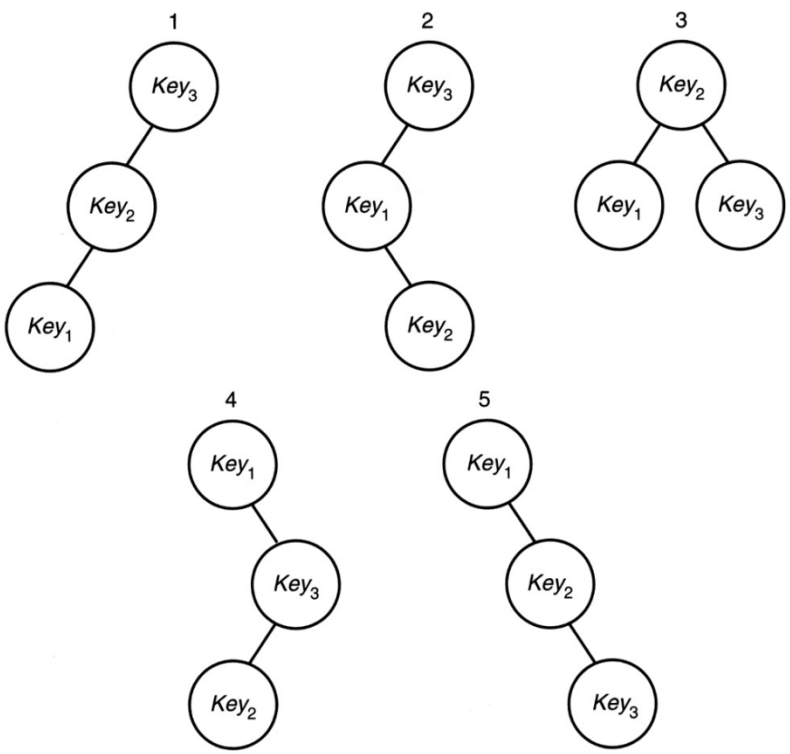
- الگوریتم جستجو در درخت جستجوی دودویی:

```
void search (node_pointer tree, keytype keyin, node_pointer& p)
{
  bool found;
  p = tree;
  found = false;
  while (! found)
      if (p->key == keyin)
                found = true;
      else if (keyin < p-> key);
          p = p-> left;        // Advance to left child.
      else
          p = p-> right;           // Advance to right child.
}
```

  - c_i : تعداد عمل اصلی (مقایسه)برای یافتن کلید iام = $depth(key_i)+1$

درخت جستجوی دودویی بهینه

- **هدف**: یافتن یک آرایش از کلیدها که زمان جستجوی متوسط را کمینه کند:
  - c_i : تعداد عمل اصلی (مقایسه)برای یافتن کلید iام = $depth(key_i)+1$
  - p_i: احتمال جستجوی کلید iام.
  - میخواهیم عبارت مقابل کمینه باشد: $\sum_{i=1}^{n} c_i p_i$
- مثال: متوسط زمان جستجو در کدام درخت کمینه است:



$$p_1 = 0.7, \quad p_2 = 0.2, \quad p_3 = 0.1$$

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

برنامه ریزی پویا برای مسأله OBST

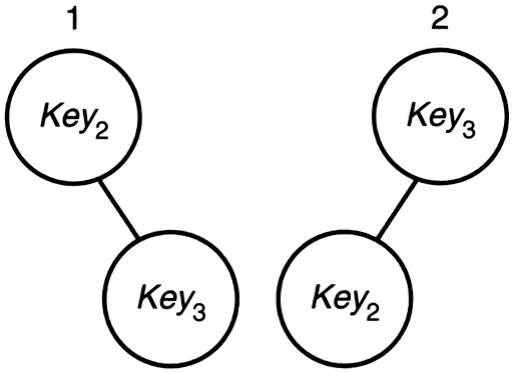
- **الگوریتم بدیهی:**
  - بررسی تمام درختهای جستجود دودویی ممکن از n کلید ← بدتر از نمایی
- **استفاده از برنامه ریزی پویا:**
  - A[i][j]: حداقل زمان جستجوی میانگین در درختی شامل کلیدهای iام تا jام.

$$A[i][j] = \sum_{m=i}^{j} c_m p_m \quad \rightarrow \quad A[i][i]=p_i$$

- **مثال:** برای سه کلید دارای احتمالات جستجوی زیر، مقدار A[2][3] را تعیین کنید: $p_1 = 0.7, \quad p_2 = 0.2, \quad p_3 = 0.1$
  - باید دو درخت زیر را در نظر گیریم:

1. $1\ (p_2) + 2\ (p_3) = 1\ (0.2) + 2\ (0.1) = 0.4$
2. $2\ (p_2) + 1\ (p_3) = 2\ (0.2) + 1\ (0.1) = 0.5$

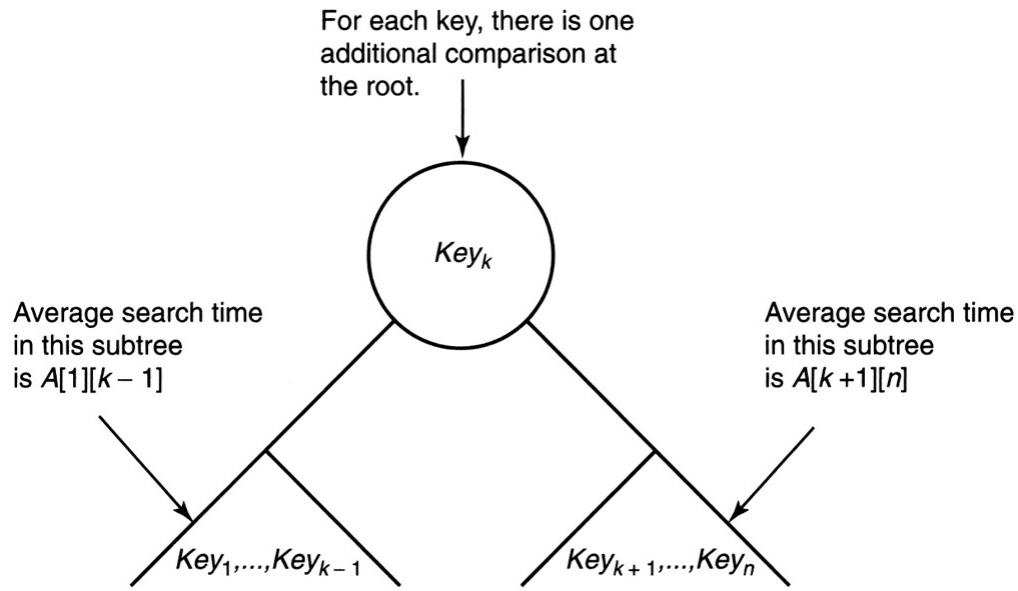
$A[2][3] = 0.4.$



√508

برنامه ریزی پویا برای مسأله OBST

- اصل بهینگی؟
  - برقرار است، چون هر زیر درخت از یک درخت بهینه باید بهینه باشد.
- درخت بهینه از میان تمام درختها با ریشه key_k :



- برای این درخت، زمان جستجوی متوسط میشود:

$$\underbrace{A[1][k-1]}_{\text{Average time in left subtree}} + \underbrace{p_1+\cdots+p_{k-1}}_{\text{Additional time comparing at root}} + \underbrace{p_k}_{\text{Average time searching for root}} + \underbrace{A[k+1][n]}_{\text{Average time in right subtree}} + \underbrace{p_{k+1}+\cdots+p_n}_{\text{Additional time comparing at root}},$$

√488

برنامه ریزی پویا برای مسأله OBST

$$\underbrace{A[1][k-1]}_{\text{Average time in left subtree}} + \underbrace{p_1+\cdots+p_{k-1}}_{\text{Additional time comparing at root}} + \underbrace{p_k}_{\text{Average time searching for root}} + \underbrace{A[k+1][n]}_{\text{Average time in right subtree}} + \underbrace{p_{k+1}+\cdots+p_n}_{\text{Additional time comparing at root}},$$

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

- هر یک از گره های 1 تا n میتواند ریشه درخت بهینه باشد ($1 \le k \le n$):

$$A[1][n] = \underset{1 \le k \le n}{minimum}(A[1][k-1] + A[k+1][n]) + \sum_{m=1}^{n} p_m, \qquad A[1][0] = A[n+1][n] = 0$$

- میتوان مطالب فوق را با داشتن کلیدهای key_i تا key_j نیز بیان کرد:

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

$$A[i][i] = p_i$$

$A[i][i-1]$ and $A[j+1][j]$ are defined to be 0.

الگوریتم و مثال از کتاب مشاهده شود

- روش پایین به بالا: محاسبه قطرهای فرعی اول تا n-1 ام ماتریس A.

√508

مثال: مساله کوله پشتی صفر و یک
(0/1 Knapsack Problem)

- یک کوله پشتی با ظرفیت وزنی M و تعداد n شیء $\{I_1,\ldots,I_n\}$ داریم.
  وزن شیء I_i برابر w_i و ارزش آن p_i است.
  میخواهیم تعدادی از اشیاء را داخل کوله پشتی قرار دهیم، بنحویکه بیشترین ارزش را درون کوله پشتی داشته باشیم و در عین حال وزن آنها از حد آستانه تحمل کوله پشتی تجاوز نکند.
- یک مسأله بهینه سازی با محدودیت است:

$$x_i = \begin{cases} 0 & \text{عدم وجود شیء } I_i \text{ در کوله پشتی} \\ 1 & \text{وجود شیء } I_i \text{ در کوله پشتی} \end{cases}$$

هدف یافتن یک دنباله بهینه $(x_1, \ldots, x_n)$ است، بنحویکه $\sum_{i=1}^{n} p_i x_i$ بیشینه بوده و در عین حال محدودیت $\sum_{i=1}^{n} w_i x_i \le M$ نیز برقرار باشد.

√488

مساله کوله پشتی صفر و یک

- الگوریتم بدیهی: بررسی تمام n تایی هایی که محدودیت را برآورده میسازند: در بدترین حالت از مرتبه نمایی است.
- آیا اصل بهینگی برقرار است؟
- knap(i, j, m): مسأله کوله پشتی با اشیاء I_i تا I_j و ظرفیت باقیمانده m درون کوله پشتی (مسأله اصلی: knap(1, n, M)).
- پاسخ بهینه مسأله knap(i, j, m) که با $x=(x_i, \ldots, x_j)$ نمایش داده میشود، دو حالت دارد:
  - حالت 1: شیء I_i انتخاب نشده است ($x_i=0$):
    در اینحالت $(x_{i+1}, \ldots, x_j)$ پاسخ بهینه زیرمسأله knap(i+1, j, m) میباشد.
  - حالت 2: شیء I_i انتخاب شده است ($x_i=1$):
    در اینحالت $(x_{i+1}, \ldots, x_j)$ پاسخ بهینه زیرمسأله $knap(i+1, j, m-w_i)$ است.
- در هر دو حالت پاسخ بهینه برای مساله‌ای با اندازه k دربردارنده پاسخ بهینه برای زیرمسأله ها با اندازه k-1 خواهد بود ($k=j-i+1$).

√524

برنامه ریزی پویا برای مساله کوله پشتی صفر و یک

- رویکرد ماتریسی: زمانی معتبر است که وزن اشیاء و ظرفیت کوله پشتی اعدادی صحیح و مثبت باشند. آرایه F را بصورت زیر تعریف میکنیم: F[i][m] = knap(1, i, m) سود بیشینه حاصل از مسأله

رابطه بازگشتی:

$$F[i][m] = \begin{cases} \max\{F[i-1][m],\ p_i + F[i-1][m-w_i]\} & w_i \le m \\ F[i-1][m] & w_i > m \end{cases}$$

مقدار سود بیشینه (پاسخ نهایی) در خانه F[n][M] خواهد بود.

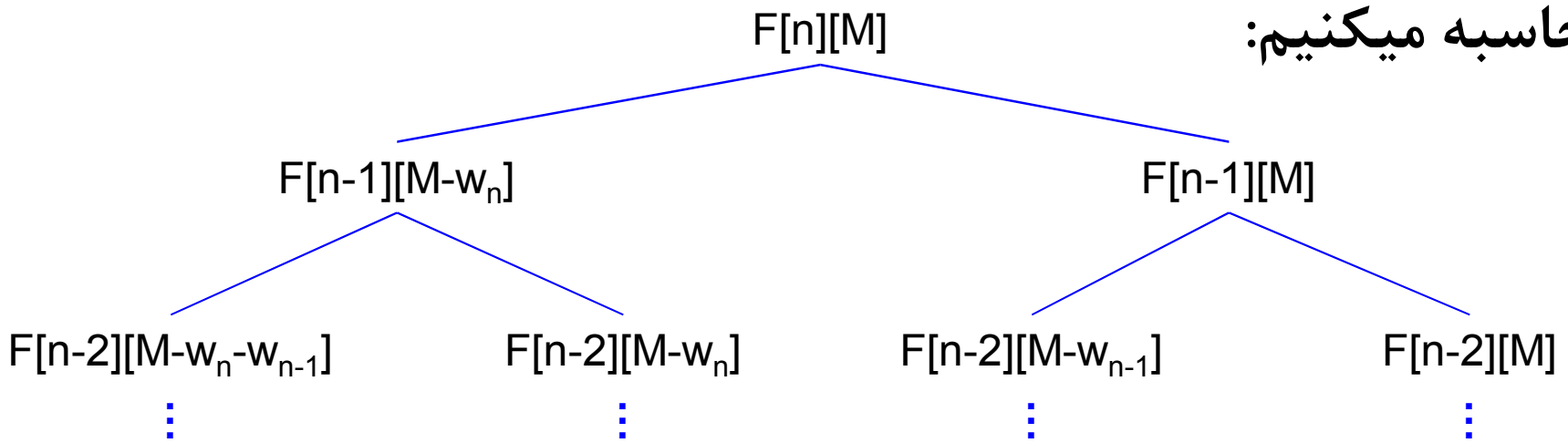
- الگوریتم:

از مرتبه $\theta(nM)$

```
number  F[0..n][0..M] = zeros[0..n][0..M];
for (i=1; i<=n ; i++)
        for (m=0; m<=M; m++){
                if(w[i]<=m)
                        F[i][m]=max(F[i-1][m], p[i]+F[i-1][m-w[i]]);
                else
                        F[i][m]=F[i-1][m];
        }
return F[n][M];
```

برنامه ریزی پویا برای مساله کوله پشتی صفر و یک

- اگر M خیلی بزرگ باشد ممکن است از الگوریتم بدیهی (که $\theta(2^n)$ بود) هم بدتر شود (مثلاً اگر $M=(n-1)!$ باشد، از مرتبه $\theta(n!)$ میشود).
- اصلاح: محاسبه همه عناصر از هر سطر مورد نیاز نیست. بنابراین ابتدا تعیین میکنیم کدام عناصر در هر سطر مورد نیاز است و تنها همانها را محاسبه میکنیم:



- در سطر $(n-i)$ ام حداکثر 2^i عنصر محاسبه میشود. لذا در بدترین حالت:
  تعداد کل عناصر محاسبه شونده = $1+2+2^2+\ldots+2^{n-1} = 2^n-1 = \theta(2^n)$
- نهایتاً الگوریتم $O(\min(2^n, nM))$ است.
- مساله NP-Complete است.

√484