فصل سوم
برنامه نویسی پویا

سید ناصر رضوی
E-mail: razavi@Comp.iust.ac.ir

۱۳۸۵

علت ناکارآمدی تقسیم و حل

- بعد از تقسیم ...
  - نمونه های کوچکتر غیر مرتبط هستند، مانند مرتب سازی ادغامی
  - نمونه های کوچکتر مرتبط هستند، مانند فیبونانچی
    - حل نمونه های مشترک به طور مکرر
- برنامه نویسی پویا
  - رهیافت پایین به بالا (bottom-up)
  - با استفاده از یک آرایه (جدول) برای ذخیره کردن راه حل نمونه های کوچکتر

مرور الگوریتم 1.7

▶ Algorithm 1.7 *n* th Fibonacci Term (Iterative)

Problem: Determine the nth term in the Fibonacci sequence.

Inputs: a nonnegative integer n.

Outputs: *fib*2, the nth term in the Fibonacci sequence.

```
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];
}
```

مراحل

- ارایه یک خاصیت بازگشتی برای حل نمونه ای از مساله

- حل نمونه ای از مساله به روش پایین به بالا با حل نمونه های کوچکتر در ابتدا

ضریب دو جمله ای

- تعریف

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

- تعریف بازگشتی

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

الگوریتم

▶ Algorithm 3.1 **Binomial Coefficient Using Divide-and-Conquer**

Problem: Compute the binomial coefficient.

Inputs: nonnegative integers n and k, where $k \leq n$.

Outputs: bin, the binomial coefficient $\binom{n}{k}$.

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

استفاده از برنامه نویسی پویا

- استفاده از یک آرایه B به منظور ذخیره کردن ضرایب
- مراحل:
  - بنا نهادن یک خاصیت بازگشتی

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

  - حل یک نمونه مساله به روش پایین به بالا با محاسبه نمودن سطرهای B به طور متوالی با شروع از سطر اول

محاسبه دنباله سطرها

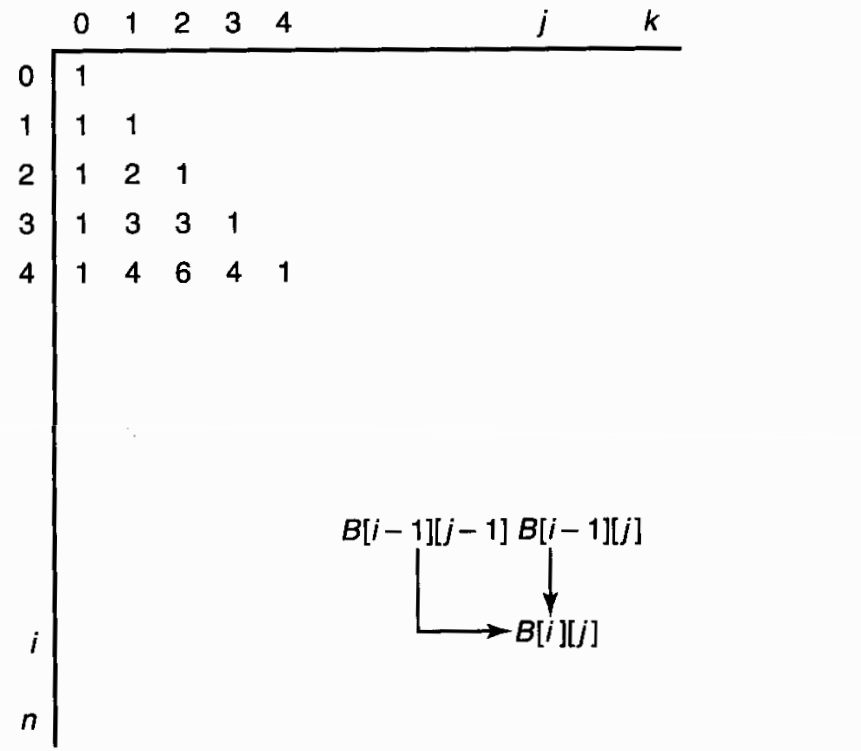


Figure 3.1 ● The array *B* used to compute the binomial coefficient.

مثال

- محاسبه $B[4][2]$

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

الگوریتم

▶ Algorithm 3.2 **Binomial Coefficient Using Dynamic Programming**

Problem: Compute the binomial coefficient.

Inputs: nonnegative integers n and k, where $k \leq n$.

Outputs: $bin2$, the binomial coefficient $\binom{n}{k}$.

```
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];
}
```

پیچیدگی زمانی

- تعداد گذر ها از حلقه j به ازاء هر مقدار از i

i	**0**	**1**	**2**	**3**	**…**	k	$k+1$	**…**	n
Number of passes	1	2	3	4	…	$k+1$	$k+1$	…	$k+1$

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

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

الگوریتم فلوید برای محاسبه کوتاه ترین مسیر ها

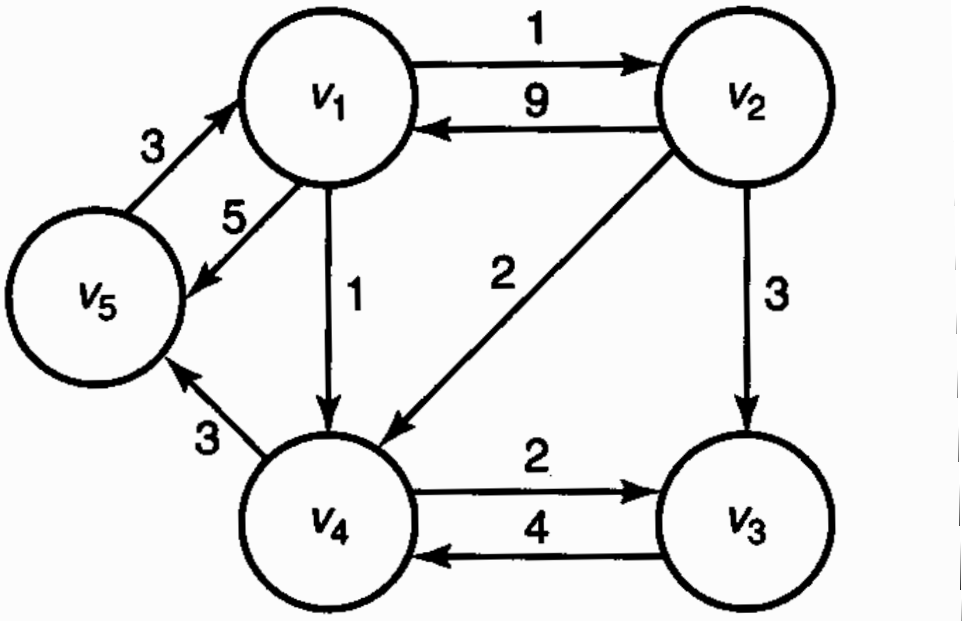


Figure 3.2 ● A weighted, directed graph.

- اصطلاحات:
  - گراف : رئوس، یال ها (کمان ها)
  - گراف جهت دار (دایگراف)
  - گراف وزن دار

اصطلاحات

- مسیر
- دور
- گراف دور دار/ بدون دور
- مسیر ساده
- طول یک مسیر

مساله کوتاه ترین مسیر

- مساله بهینه سازی
  - می تواند بیش از یک راه حل کاندیدا وجود داشته باشد
  - هر راه حل کاندیدا دارای یک مقدار می باشد.
  - راه حل، یک راه حل کاندیدا می باشد که دارای مقدار بهینه می باشد.

- الگوریتم brute force (الگوریتمی که تمام حالت های ممکن را در نظر می گیرد) دارای پیچیدگی زمانی به صورت فاکتوریل می باشد:

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

بازنمایی گراف

- ماتریس مجاورتی

$$W[i][j] = \begin{cases} \text{وزن یال} & \text{اگر یالی بین } v_i \text{ و } v_j \text{ باشد} \\ \infty & \text{اگر یالی بین } v_i \text{ و } v_j \text{ نباشد} \\ 0 & \text{اگر } i = j \text{ باشد} \end{cases}$$

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

Figure 3.3 • *W* represents the graph in Figure 3.2 and *D* contains the lengths of the shortest paths. Our algorithm for the Shortest Paths problem computes the values in *D* from those in *W*.

الگوریتم

- ایجاد دنباله ای از $n+1$ آرایه $D^{(k)}$ به طوری که $0 \le k \le n$

$D^{(k)}[i][j]$ = طول کوتاه ترین مسیر از v_i به v_j که فقط از رئوس موجود در مجموعه $\{v_1, v_2, \ldots, v_k\}$ به عنوان رئوس میانی استفاده می کند.

$$D^{(0)} = W$$

$$D^{(n)} = D$$

مثال

- محاسبه $D[2][5]$

$D^{(0)}[2][5] = length[v_2, v_5] = \infty$

$D^{(1)}[2][5] = \text{minimum}(length[v_2, v_5], length[v_2, v_1, v_5]$
$= \text{minimum}(\infty, 14) = 14$

$D^{(2)}[2][5] = D^{(1)}[2][5] = 14$

$D^{(3)}[2][5] = D^{(2)}[2][5] = 14$

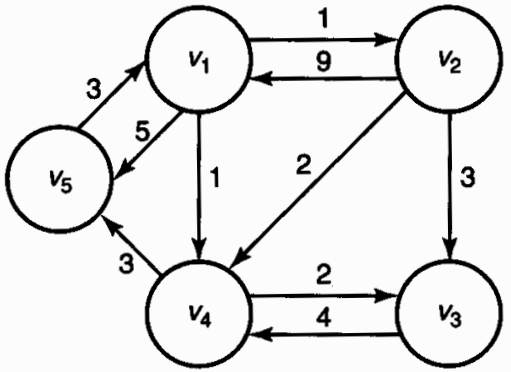


Figure 3.2 • A weighted, directed graph.

$D^{(4)}[2][5] = \text{minimum}(length[v_2, v_1, v_5], length[v_2, v_4, v_5],$
$length[v_2, v_1, v_4, v_5], length[v_2, v_3, v_4, v_5])$
$= \text{minimum}(14, 5, 13, 10) = 5$

$D^{(5)}[2][5] = D^{(4)}[2][5] = 5$

برنامه نویسی پویا برای مساله کوتاه ترین مسیر

- باید راهی برای محاسبه $D = D^{(n)}$ از روی $W = D^{(0)}$ به دست آوریم:
  - ارایه یک خاصیت بازگشتی برای محاسبه کردن $D^{(k)}$ از $D^{(k-1)}$
  - حل نمونه ای از مساله به روش پایین به بالا بازاء $k = 1$ تا n و ایجاد دنباله زیر:

$$D^{(0)}, D^{(1)}, D^{(2)}, \ldots, D^{(n)}$$

$D^{(0)} = W$ ، $D^{(n)} = D$

محاسبه $D^{(k)}$ از روی $D^{(k-1)}$

- **حالت ۱:** حداقل یک نمونه از کوتاه ترین مسیرها از v_i به v_j که فقط از رئوس موجود در مجموعه $\{v_1, v_2, \ldots, v_k\}$ به عنوان رئوس میانی استفاده می کند، از v_k عبور نکند. در این صورت:

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

- **حالت ۲:** تمام نمونه های کوتاه ترین مسیر از v_i به v_j که فقط از رئوس موجود در مجموعه $\{v_1, v_2, \ldots, v_k\}$ به عنوان رئوس میانی استفاده می کند، از v_k عبور کنند. در این صورت:

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

جمع بندی

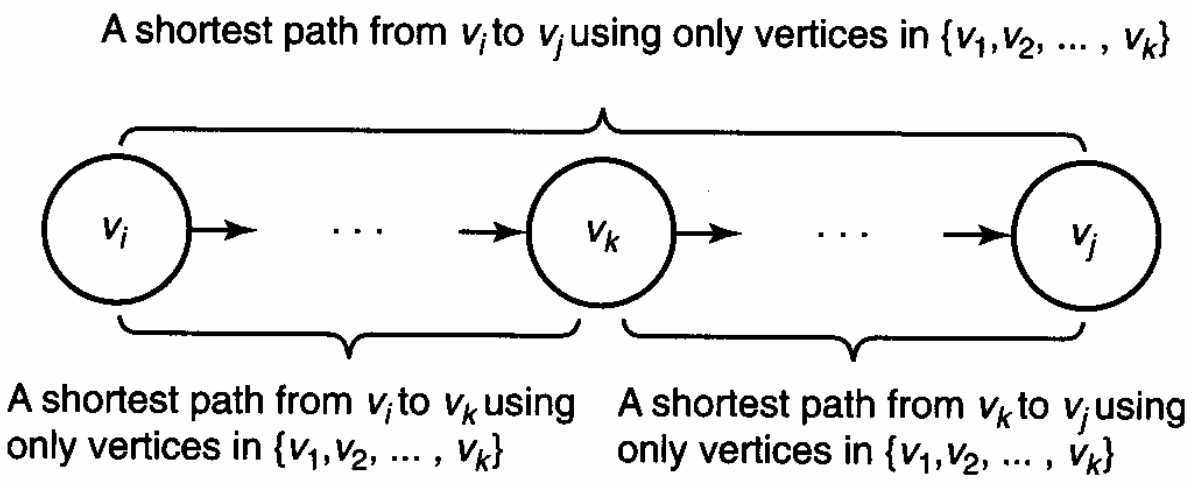


Figure 3.4 ● The shortest path uses v_k.

- $D^{(k)}[i][j] = minimum(D^{(k\text{-}1)}[i][j], D^{(k\text{-}1)}[i][k] + D^{(k\text{-}1)}[k][j])$

مثال

- محاسبه $D^{(2)}[5][4]$

$$D^{(1)}[2][4] = \text{minimum}(D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4]) = \text{minimum}(2, 9 + 1) = 2$$

$$D^{(1)}[5][2] = \text{minimum}(D^{(0)}[5][2], D^{(0)}[5][1] + D^{(0)}[1][2]) = \text{minimum}(\infty, 3 + 1) = 4$$

$$D^{(1)}[5][4] = \text{minimum}(D^{(0)}[5][4], D^{(0)}[5][1] + D^{(0)}[1][4]) = \text{minimum}(\infty, 3 + 1) = 4$$

$$D^{(2)}[5][4] = \text{minimum}(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4]) = \text{minimum}(4, 4 + 2) = 4$$

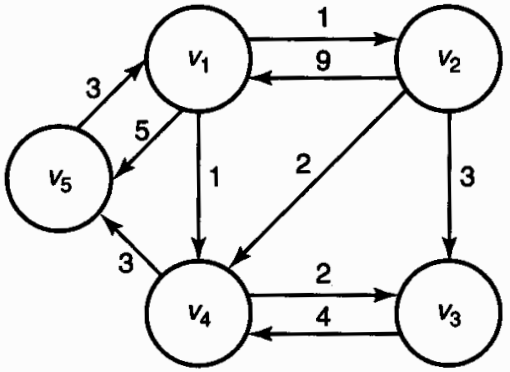


Figure 3.2 • A weighted, directed graph.

الگوریتم

► Algorithm 3.3

Floyd's Algorithm for Shortest Paths

Problem: Compute the shortest paths from each vertex in a weighted graph to each of the other vertices. The weights are nonnegative numbers.

Inputs: A weighted, directed graph and n, the number of vertices in the graph. The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where $W[i][j]$ is the weight on the edge from the ith vertex to the jth vertex.

Outputs: A two-dimensional array D, which has both its rows and columns indexed from 1 to n, where $D[i][j]$ is the length of a shortest path from the ith vertex to the jth vertex.

```
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]);
}
```

پیچیدگی زمانی برای همه حالات

- عمل اصلی: دستور واقع در حلقه for-j
- اندازه ورودی: n، تعداد رئوس گراف
- پیچیدگی زمانی:

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

▶ Algorithm 3.4

Floyd's Algorithm for Shortest Paths 2

Problem: Same as in Algorithm 3.3, except shortest paths are also created.

Additional outputs: an array P, which has both its rows and columns indexed from 1 to n, where

$$P[i][j] = \begin{cases} \text{highest index of an intermediate vertex on the shortest path} \\ \text{from } v_i \text{ to } v_j, \text{ if at least one intermediate vertex exists.} \\ 0, \text{ if no intermediate vertex exists.} \end{cases}$$

```
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];
           }
}
```

الگوریتم فلوید برای یافتن کوتاه ترین مسیر ۲

خروجی نمونه

- آرایه P در هنگام اعمال الگوریتم ۳-۴ بر گراف شکل ۳-۲ ایجاد شده است

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

Figure 3.5 • The array P produced when Algorithm 3.4 is applied to the graph in Figure 3.2.

الگوریتم چاپ کوتاهترین مسیر

▶ Algorithm 3.5

Print Shortest Path

Problem: Print the intermediate vertices on a shortest path from one vertex to another vertex in a weighted graph.

Inputs: the array P produced by Algorithm 3.4, and two indices, q and r, of vertices in the graph that is the input to Algorithm 3.4.

$$P[i][j] = \begin{cases} \text{highest index of an intermediate vertex on the shortest path} \\ \text{from } v_i \text{ to } v_j, \text{ if at least one intermediate vertex exists.} \\ 0, \text{ if no intermediate vertex exists.} \end{cases}$$

Outputs: the intermediate vertices on a shortest path from v_q to v_r.

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

برنامه نویسی پویا و مسایل بهینه سازی

- مراحل:

– ارائه یک خاصیت بازگشتی برای بدست آوردن راه حل بهینه نمونه ای از مساله

– محاسبه مقدار راه حل بهینه به روش پایین به بالا

– ساختن یک راه حل بهینه به روش پایین به بالا

اصل بهینگی

- یک راه حل بهینه برای یک نمونه از مساله همواره شامل راه حل های بهینه برای تمامی زیر نمونه ها می باشد.

- اطمینان می دهد که راه حل بهینه یک نمونه از مساله می تواند با ترکیب راه حل های بهینه زیر نمونه ها حاصل شود.

- قبل از استفاده از برنامه نویسی پویا برای بدست آوردن راه حل، لازم است که نشان دهیم اصل بهینگی برقرار است.

مساله طولانی ترین مسیرها

- طولانی ترین مسیر از v_1 به v_4 مسیر $[v_1, v_3, v_2, v_4]$ می باشد.
- زیر مسیر $[v_1, v_3]$ بهینه نیست.

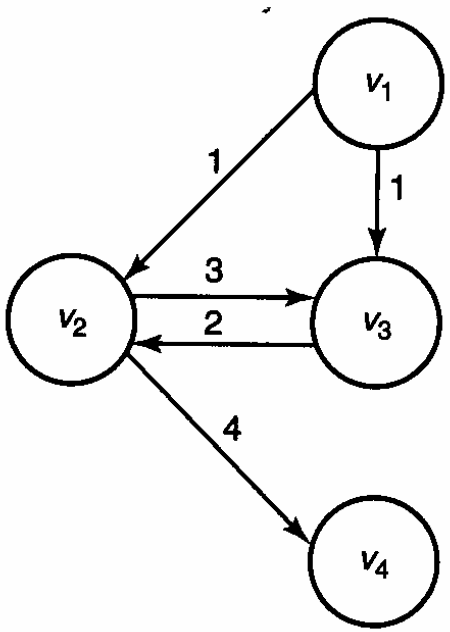


Figure 3.6 ● A weighted, directed graph with a cycle.

ضرب زنجیره ای ماتریس ها

- برای ضرب نمودن یک ماتریس $i * j$ در یک ماتریس $j * k$ تعداد ضرب های ابتدایی برابر با $i * j * k$ می باشد.
- ضرب ماتریس ها دارای خاصیت شرکت پذیری می باشد، بدین معنا که ترتیب های متفاوتی برای ضرب چند ماتریس در یکدیگر وجود دارد که هر یک منجر به تعداد متفاوتی از ضرب های ابتدایی می شود.
- مثال:

$$\underset{20\times 2}{A} \times \underset{2\times 30}{B} \times \underset{30\times 12}{C} \times \underset{12\times 8}{D}$$

مثال

- روش ۱

$$(((A \times B) \times C) \times D)$$

$$20\times2 \quad 2\times30 \quad 30\times12 \quad 12\times8$$

تعداد ضربها $= (20 * 2 * 30)$

$$+ (20 * 30 * 12)$$

$$+ (20 * 12 * 8)$$

$$= 10320$$

مثال

- روش ۲

$$((A \times B) \times (C \times D))$$

$$20\times2 \quad 2\times30 \quad 30\times12 \quad 12\times8$$

تعداد ضربها $= (20 * 2 * 30)$

$$+ (30 * 12 * 8)$$

$$+ (20 * 30 * 8)$$

$$= 8880$$

مثال

- روش ۳

$$((A \times (B \times C)) \times D)$$

$$20\times2 \quad 2\times30 \quad 30\times12 \quad 12\times8$$

تعداد ضربها $= (2 * 30 * 12)$

$$+ (20 * 2 * 12)$$

$$+ (20 * 12 * 8)$$

$$= 3120$$

مثال

- روش ۴

$$(A \times ((B \times C) \times D))$$
$$20\times2 \quad 2\times30 \quad 30\times12 \quad 12\times8$$

تعداد ضربها $= (2 * 30 * 12)$

$+ (2 * 12 * 8)$

$+ (20 * 2 * 8)$

$= 1232$

مثال

- روش ۵

$$(A \times (B \times (C \times D)))$$

$$20\times2 \quad 2\times30 \quad 30\times12 \quad 12\times8$$

تعداد ضربها $= (30 * 12 * 8)$

$+ (2 * 30 * 8)$

$+ (20 * 2 * 8)$

$= 3680$

الگوریتم Brute-force

- پیچیدگی این روش حداقل نمایی می باشد.

$t_n \geq 2\ t_{n\text{-}1}$ and $t_2 = 1$ => $t_n \geq 2^{n\text{-}2}$

- یک تحلیل بهتر:
  - تعداد روش های پرانتز گذاری n ماتریس (جمله $n-1$ در دنباله کاتالان):

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

  - و چون برای n های به اندازه کافی بزرگ

$$T(n) \approx \frac{4^{n-2}}{(n-1)^{1.5}} \in O\left(\frac{4^n}{n}\right)$$

  - بنابراین پیچیدگی الگوریتم کورکورانه نمایی می باشد.

استفاده از برنامه نویسی پویا

- در این مساله اصل بهینگی برقرار می باشد.
- برخی از خصوصیات:
  - تعداد ستون ها در A_{k-1} برابر تعداد سطرها در A_k می باشد.
  - می توان فرض نمود که d_0 تعداد سطرهای A_1 و d_k تعداد ستونهای A_k می باشد $(1 \le k \le n)$.
  - ابعاد ماتریس A_k برابر است با $d_{k-1} \times d_k$

معرفی دنباله ای از آرایه ها

- $M[i][j]$= حداقل تعداد ضرب های مورد نیاز برای ضرب A_i تا A_j ، به شرطی که $i < j$
- $M[i][i] = 0$
- مثال

$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$

$5\times 2 \quad 2\times 3 \quad 3\times 4 \quad 4\times 6 \quad 6\times 7 \quad 7\times 8$

$d_0 \; d_1 \; d_1 \; d_2 \; d_2 \; d_3 \; d_3 \; d_4 \; d_4 \; d_5 \; d_5 \; d_6$

$M[4][6] = ?$

برای ضرب ۶ ماتریس

- فاکتور گیری های ممکن

– $A_1 (A_2 A_3 A_4 A_5 A_6)$
– $(A_1 A_2) (A_3 A_4 A_5 A_6)$
– $(A_1 A_2 A_3) (A_4 A_5 A_6)$
– $(A_1 A_2 A_3 A_4) (A_5 A_6)$
– $(A_1 A_2 A_3 A_4 A_5) A_6$

- تعداد ضرب ها برای فاکتور گیری k ام

$$M[1][k] + M[k+1][6] + d_0 d_k d_6$$

- بنابراین

$$M[1][6] = \min_{1 \le k \le 5} imum(M[1][k] + M[k+1][6] + d_0 d_k d_6)$$

حالت کلی

$$\begin{cases} M[i][j] = \min_{i \le k \le j-1} imum(M[i][k] + M[k+1][j] + d_{i-1} d_k d_j), & if \quad i < j \\ M[i][i] = 0 \end{cases}$$

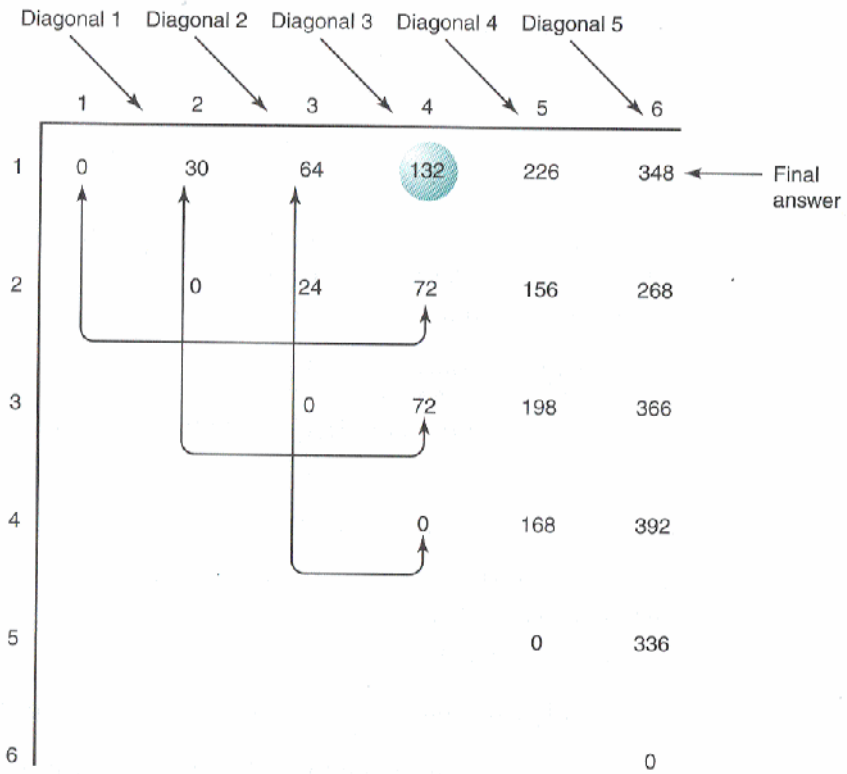


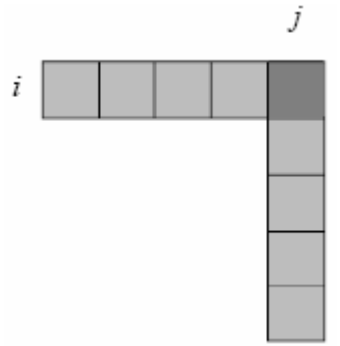
Figure 3.8 • The array M developed in Example 3.5. M [1] [4], which is circled, is computed from the pairs of entries indicated.

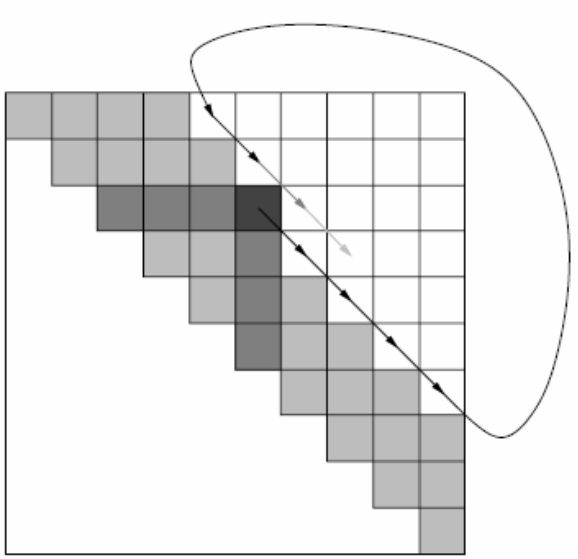
- محاسبه

$M[1][2], M[2][3]$ –

نحوه پر کردن جدول

$$m[i,j] = \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + r_{i-1} \cdot r_k \cdot r_j \}$$





مثال

$$r_0 = 10, r_1 = 20, r_2 = 50, r_3 = 1, r_4 = 100$$

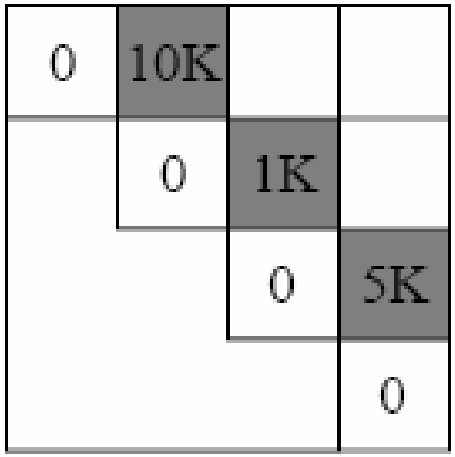
- محاسبه عناصر قطر ۱

$$m[1,2] = \min_{1 \le k < 2} \{m[1,k] + m[k+1,2] + r_0 r_k r_2\}$$

$$= r_0 r_1 r_2 = 10{,}000$$

$$m[2,3] = r_1 r_2 r_3 = 1{,}000$$

$$m[3,4] = r_2 r_3 r_4 = 5{,}000$$



مثال

$$r_0 = 10, r_1 = 20, r_2 = 50, r_3 = 1, r_4 = 100$$

- محاسبه عناصر قطر ۲

$$\begin{aligned} m[1,3] &= \min_{1 \le k < 3} \{m[1,k] + m[k+1,3] + r_0 r_k r_3\} \\ &= \min\{m[1,1] + m[2,3] + r_0 r_1 r_3, \\ &\qquad m[1,2] + m[3,3] + r_0 r_2 r_3\} \\ &= \min\{0 + 1000 + 200, 10000 + 0 + 500\} \\ &= 1200 \end{aligned}$$

$$\begin{aligned} m[2,4] &= \min_{2 \le k < 4} \{m[2,k] + m[k+1,4] + r_1 r_k r_4\} \\ &= \min\{m[2,2] + m[3,4] + r_1 r_2 r_4, \\ &\qquad m[2,2] + m[4,4] + r_1 r_3 r_4\} \\ &= \min\{0 + 5000 + 100000, 1000 + 0 + 2000\} \\ &= 3000 \end{aligned}$$

0	10K	1.2K	
	0	1K	3K
		0	5K
			0

مثال

$$r_0 = 10, r_1 = 20, r_2 = 50, r_3 = 1, r_4 = 100$$

- محاسبه عناصر قطر ۳

$$m[1,4] = \min_{1 \le k < 4} \{m[1,k] + m[k+1,4] + r_0 r_k r_4\}$$

$$= \min \{m[1,1] + m[2,4] + r_0 r_1 r_4,$$

$$m[1,2] + m[3,4] + r_0 r_2 r_4,$$

$$m[1,3] + m[4,4] + r_0 r_3 r_4\}$$

$$= 2200$$

0	10K	1.2K	2.2K
	0	1K	3K
		0	5K
			0

الگوریتم

▶ Algorithm 3.6 **Minimum Multiplications**

Problem: Determining the minimum number of elementary multiplications needed to multiply n matrices and an order that produces that minimum number.

Inputs: the number of matrices n, and an array of integers d, indexed from 0 to n, where $d[i-1] \times d[i]$ is the dimension of the ith matrix.

Outputs: *minmult*, the minimum number of elementary multiplications needed to multiply the n matrices; a two-dimensional array P from which the optimal order can be obtained. P has its rows indexed from 1 to $n-1$ and its columns indexed from 1 to n. $P[i][j]$ is the point where matrices i through j are split in an optimal order for multiplying the matrices.

```
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];
}
```

پیچیدگی زمانی برای همه حالات

- عمل اصلی: دستورالعملهای اجرا شده برای هر مقدار k
- اندازه ورودی: n، تعداد ماتریس هایی که باید ضرب شوند
- پیچیدگی زمانی:

$$\sum_{diagonal=1}^{n-1}[(n-diagonal)\times diagonal]=\frac{n(n-1)(n+1)}{6}\in\Theta(n^3)$$

برای بدست آوردن یک ترتیب بهینه از آرایه P

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

Figure 3.9 ● The array P produced when Algorithm 3.6 is applied to the dimensions in Example 3.5.

- ترتیب بهینه چیست؟

چاپ ترتیب بهینه

▶ Algorithm 3.7 Print Optimal Order

Problem: Print the optimal order for multiplying n matrices.

Inputs: Positive integer n, and the array P produced by Algorithm 3.6. $P[i][j]$ is the point where matrices i through j are split in an optimal order for multiplying those matrices.

Outputs: the optimal order for multiplying the matrices.

```
void order (index i, index j)
{
  if(i == j)
     cout << "A" << i;
  else{
     k = P[i][j];
     cout << "(";
     order(i, k);
     order(k + 1, j);
     cout << ")";
  }
}
```

درخت های جستجوی دودویی بهینه

- درخت جستجوی دودویی

تعریف

درخت جستجوی دودویی یک درخت دودویی از عناصر (کلید ها) است، که از یک مجموعه مرتب حاصل می شود، به طوری که

۱. هر گره دارای یک کلید می باشد.

۲. کلیدهای واقع در زیر درخت چپ یک گره، کوچکتر یا مساوی کلید آن گره می باشند.

۳. کلیدهای واقع در زیر درخت راست یک گره، بزرگتر یا مساوی کلید آن گره می باشند.

مثال ها

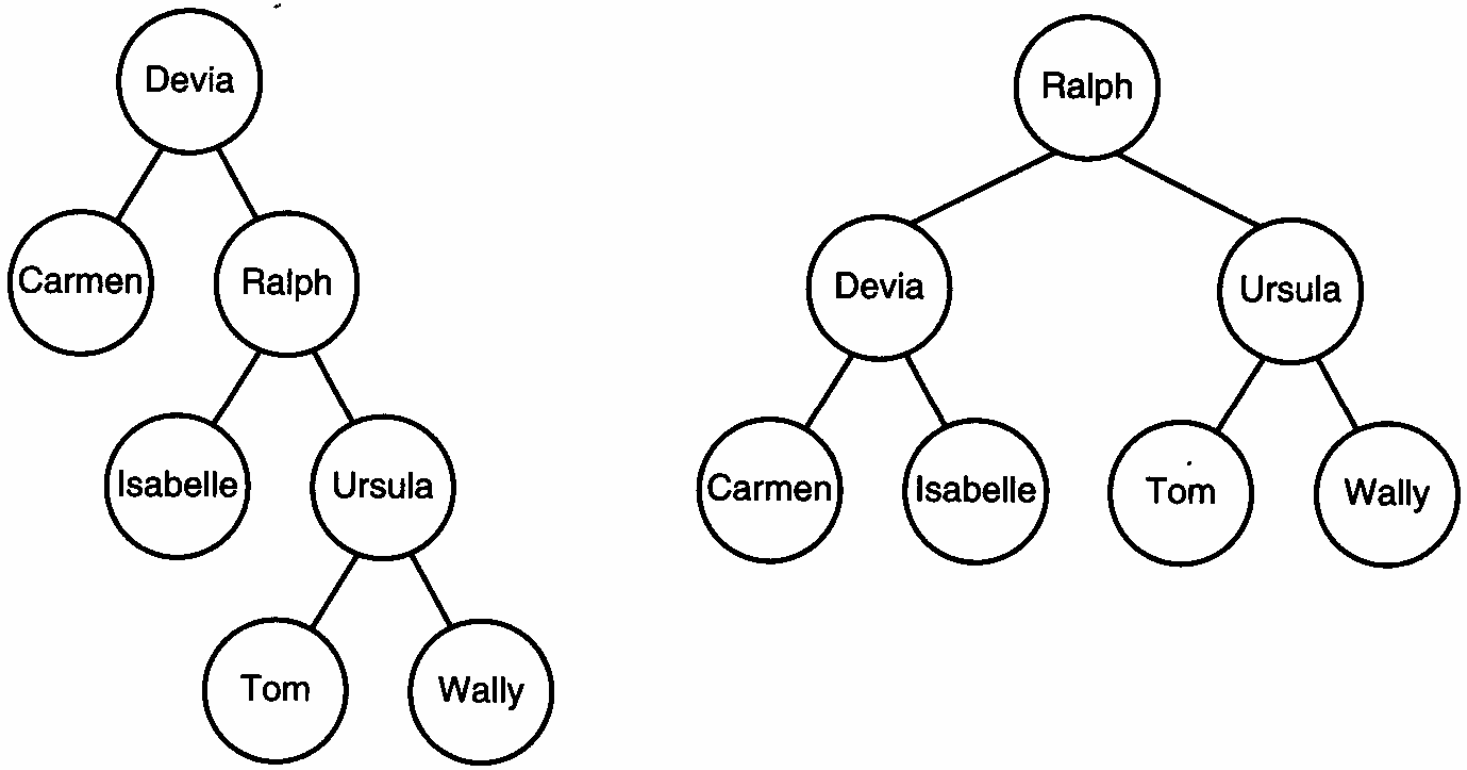


Figure 3.10 ● Two binary search trees.

درخت متوازن

- عمق (سطح) یک گره: تعداد لبه های موجود در مسیر منحصر بفرد از ریشه به آن گره.
- عمق یک درخت: حداکثر عمق تمامی گره ها در آن درخت.
- درخت دودویی متوازن: اگر عمق دو زیر درخت از هر گره بیش از یک واحد اختلاف نداشته باشند.
- درخت جستجوی دودویی بهینه: زمان متوسط برای مکان یابی یک کلید کمینه است.

ساختار داده ای

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

typedef nodetype* node_pointer
```

الگوریتم جستجو

▶ Algorithm 3.8 **Search Binary Tree**

Problem: Determine the node containing a key in a binary search tree. It is assumed that the key is in the tree.

Inputs: a pointer *tree* to a binary search tree and a key *keyin*.

Outputs: a pointer *p* to the node containing the key.

```
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.
}
```

متوسط زمان جستجو

- زمان جستجو: تعداد مقایسه های انجام شده برای مکان یابی یک کلید
- زمان جستجو برای key برابر است با:

$$depth(key) + 1$$

- متوسط زمان جستجو:

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

که در آن:

n تعداد کلید ها،

p_i احتمال آنکه key_i کلید مورد جستجو باشد،

c_i تعداد مقایسه های مورد نیاز برای یافتن key_i می باشد.

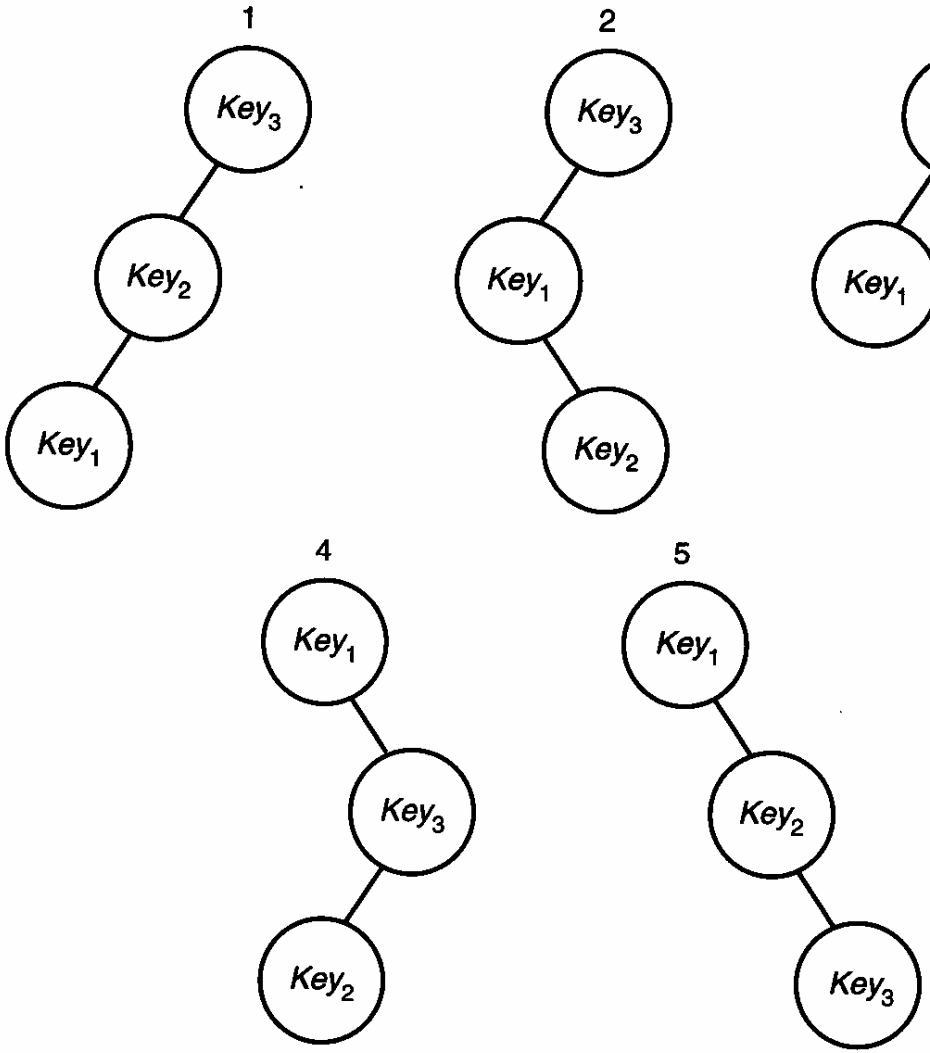


Figure 3.11 ● The possible binary search trees when there are three keys.

مثال 3.7

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

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

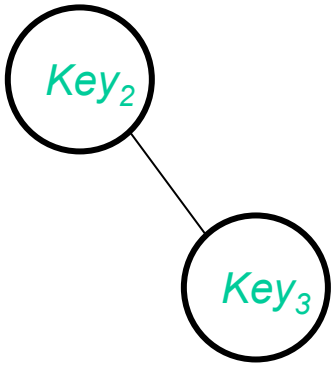
توسعه یک الگوریتم کارآ

- جستجوی Brute force حداقل نمایی است
  - تعداد درخت های جست و جوی دودویی متفاوت با عمق $n-1$ برابر است با 2^{n-1}
- اصل بهینگی برقرار است.
- اگر $A[i][j]$ برابر حداقل مقدار $\sum_{m=i}^{j} c_m p_m$ باشد.
- $A[i][i] = p_i$

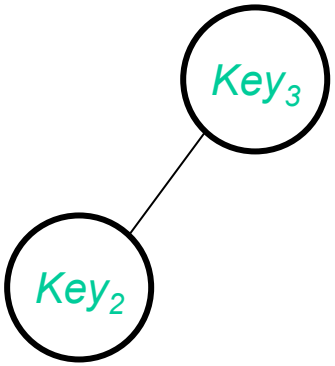
مثال 3.8

- محاسبه $A[2][3]$ در مثال قبل

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



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



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

درخت سمت چپ بهینه است و بنابراین:

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

جستجوی یک کلید در درخت k

- فرض کنید درخت k که ریشه آن برابر key_k است، بهینه می باشد.

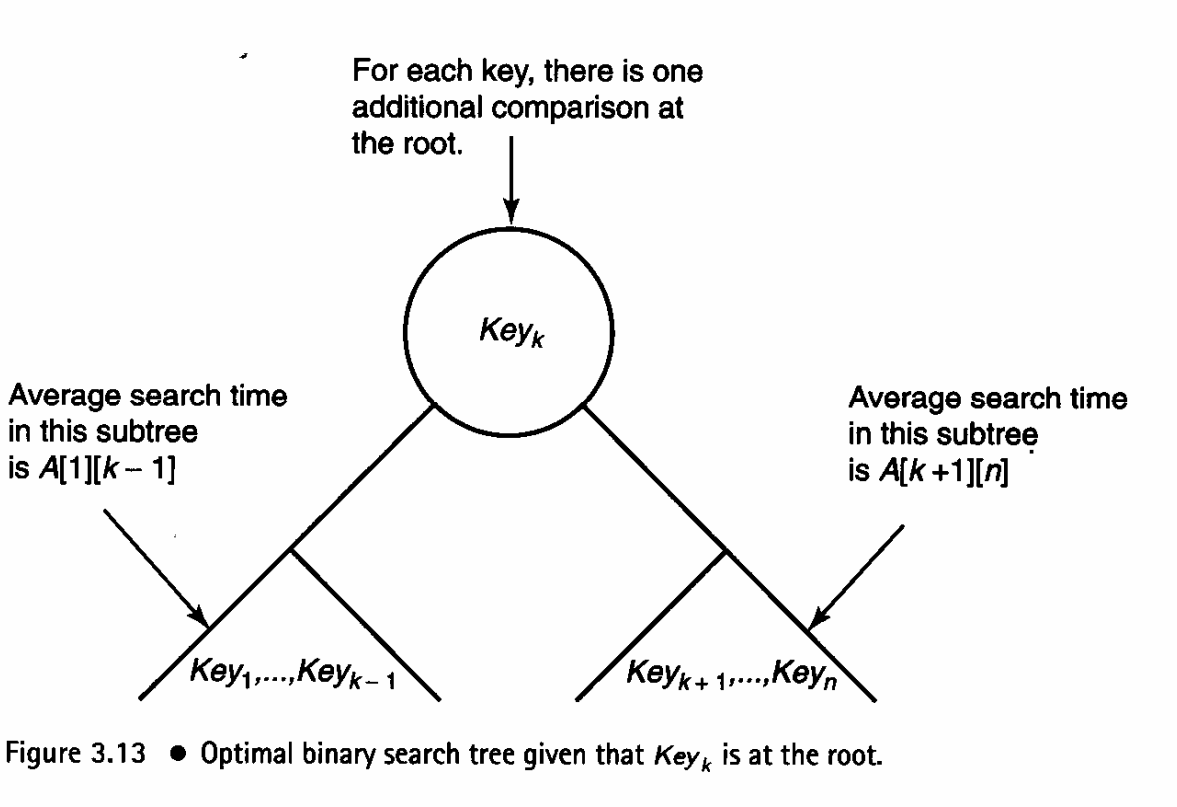


Figure 3.13 ● Optimal binary search tree given that Key_k is at the root.

متوسط زمان جستجو

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

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

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

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

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

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

الگوریتم

▶ Algorithm 3.9

Optimal Binary Search Tree

Problem: Determine an optimal binary search tree for a set of keys, each with a given probability of being the search key.

Inputs: n, the number of keys, and an array of real numbers p indexed from 1 to n, where $p[i]$ is the probability of searching for the ith key.

Outputs: A variable *minavg*, whose value is the average search time for an optimal binary search tree; and a two-dimensional array R from which an optimal tree can be constructed. R has its rows indexed from 1 to $n+1$ and its columns indexed from 0 to n. $R[i][j]$ is the index of the key in the root of an optimal tree containing the ith through the jth keys.

```
void optsearchtree (int n,
                    const float p[],
                    float& minavg,
                    index R[][])
{
  index i, j, k, diagonal;
  float A[1..n + 1][0..n];
```

الگوریتم (ادامه)

```
for (i = 1; i <= n; i++){
    A[i][i - 1] = 0;
    A[i][i] = p[i];
    R[i][i] = i;
    R[i][i - 1] = 0;
}
A[n + 1][n] = 0;
R[n + 1][n] = 0;
for (diagonal = 1; diagonal <= n - 1; diagonal++)
    for (i = 1; i <= n - diagonal; i++){                    // Diagonal-1 is
                                                            // just above the
                                                            // main diagonal.
        j = i + diagonal;
        A[i][j] = minimum(A[i][k - 1] + A[k + 1][j]) + sum_{m=i}^{j} p_m .
                  i <= k <= j
        R[i][j] = a value of k that gave the minimum;
    }
minavg = A[1][n];
}
```

پیچیدگی زمانی در همه حالات

- عمل اصلی: دستورالعمل های اجرا شده برای هر مقدار از k
- اندازه ورودی: n، تعداد کلید ها
- پیچیدگی زمانی:

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

ساختن درخت جستجوی دودویی بهینه

▶ Algorithm 3.10 **Build Optimal Binary Search Tree**

Problem: Build an optimal binary search tree.

Inputs: n, the number of keys, an array Key containing the n keys in order, and the array R produced by Algorithm 3.9. $R[i][j]$ is the index of the key in the root of an optimal tree containing the ith through the jth keys.

Outputs: a pointer *tree* to an optimal binary search tree containing the n keys.

```
node_pointer tree (index i, j)
{
  index k;
  node_pointer p;

  k = R[i][j];
  if (k == 0)
     return NULL;
  else{
     p = new nodetype;
     p-> key = Key[k];
     p-> left = tree(i, k - 1);
     p-> right = tree(k + 1, j);
     return p;
  }
}
```

مثال 3.9

- کلید ها:

Don	Isabelle	Ralph	Wally
Key[1]	Key[2]	Key[3]	Key[4]
p_1=3/8	p_2=3/8	p_3=1/8	p_4=1/8

- آرایه های ایجاد شده:

	0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

A

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

R

Figure 3.14 ● The arrays *A* and *R*, produced when Algorithm 3.9 is applied to the instance in Example 3.9.

درخت حاصل

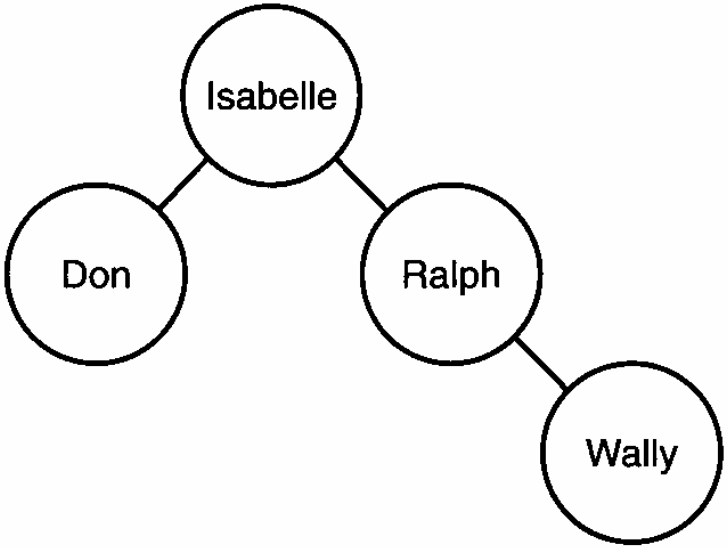


Figure 3.15 • The tree produced when Algorithms 3.9 and 3.10 are applied to the instance in Example 3.9.

مساله فروشنده دوره گرد (*TSP*)

- تور(دور هامیلتونی): مسیری از یک راس به خودش که از هر یک از رئوس دیگر دقیقا یک با عبور می کند.
- تور بهینه: مسیری با مشخصات بالا با طول حداقل.
- الگوریتم *Brute force* از مرتبه فاکتوریل می باشد:

$$(n\text{-}1)(n\text{-}2)\cdots 1 = (n\text{-}1)!$$

- اصل بهینگی برقرار است(؟).

یک مثال

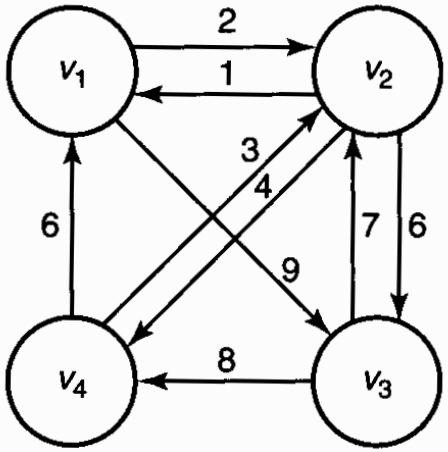


Figure 3.16 ● The optimal tour is $[v_1, v_3, v_4, v_2, v_1]$.

$length[V_1, V_2, V_3, V_4, V_1] = 22$

$length[V_1, V_3, V_2, V_4, V_1] = 26$

$length[V_1, V_3, V_4, V_2, V_1] = 21$

آماده سازی

- V = مجموعه تمام رئوس
- A = زیر مجموعه ای از V
- $D[v_i][A]$ = طول کوتاهترین مسیر از v_i به v_1 که از تمام رئوس مجموعه A دقیقا یک بار عبور می کند.

مثال ۳-۱۰

- محاسبه $D[v_2][A]$ برای گراف زیر

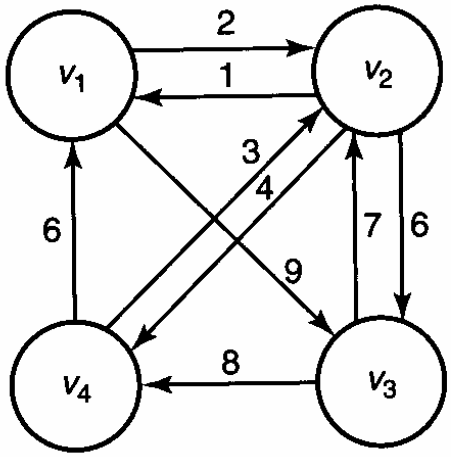


Figure 3.16 ● The optimal tour is $[v_1, v_3, v_4, v_2, v_1]$.

اگر $A=\{v_3\}$، آنگاه:

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

اگر $A=\{v_3, v_4\}$، آنگاه:

$$D[v_2][A] = \text{minimum}(\ length[v_2, v_3, v_4, v_1], length[v_2, v_4, v_3, v_1]\ )$$

$$= \text{minimum}(20, \infty) = 20$$

الگوریتم

- طول یک تور بهینه =

$$D[v_1][V-\{v_1\}] = \underset{2\le j\le n}{\text{minimum}}(W[1][j] + D[v_j][V-\{v_1, v_j\}])$$

- به طور کلی بازاء $i \neq 1$ و $v_i \notin A$

$$D[v_i][A] = \underset{j:v_j\in A}{\min imum}(W[i][j] + D[v_j][A-\{v_j\}]) \quad if \quad A \neq \phi$$

$$D[v_i][\phi] = W[i][1]$$

مثال ۳-۱۱

- محاسبه تور بهینه برای گراف زیر

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

Figure 3.17 ● The adjacency matrix representation *W* of the graph in Figure 3.16.

حل مثال ۳-۱۱

- ابتدا مجموعه تهی را در نظر می گیریم:

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

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

حل مثال ۳-۱۱

- اکنون تمام مجموعه های یک عنصری را در نظر می گیریم:

$$D[v_3][\{v_2\}] = \underset{j:v_j \in \{v_2\}}{\text{minimum}}(W[3][j] + D[v_j][\{v_2\} - \{v_j\}])$$

$$= W[3][2] + D[v_2][\varnothing] = 7 + 1 = 8$$

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

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

حل مثال ۳-۱۱

- سپس همه مجموعه های دو عنصری را در نظر می گیریم:

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

$$= \text{minimum}(W[4][2] + D[v_2][\{v_3\}],\ W[4][3] + D[v_3][\{v_2\}])$$

$$= \text{minimum}(3 + \infty,\quad \infty + 8) = \infty$$

$$D[v_3][\{v_2, v_4\}] = \text{mimimum}(7 + 10,\ 8 + 4) = 12$$

$$D[v_2][\{v_3, v_4\}] = \text{mimimum}(6 + 14,\ 14 + \infty) = 20$$

حل مثال ۳-۱۱

- اکنون طول تور بهینه را محاسبه می کنیم:

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

$$= \text{minimum}(W[1][2] + D[v_2][\{v_3, v_4\}],$$
$$W[1][3] + D[v_3][\{v_2, v_4\}],$$
$$W[1][4] + D[v_4][\{v_2, v_3\}])$$

$$=\text{minimum}(2 + 20, 9 + 12, \infty + \infty) = 21$$

```
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}];
}
```

الگوریتم ۳-۱۱ الگوریتم برنامه نویسی پویا برای مساله فروشنده دوره گرد

قضیه ۳-۱

- بازاء هر $n \geq 1$

$$\sum_{k=1}^{n} k\binom{n}{k} = n\,2^{n-1}$$

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

$$\sum_{k=1}^{n} k\binom{n}{k} = \sum_{k=1}^{n} n\binom{n-1}{k-1} = n\sum_{k=0}^{n-1}\binom{n-1}{k} = n\,2^{n-1}$$

پیچیدگی زمانی برای همه حالات

- عمل اصلی: دستورالعمل اجرا شده برای هر مقدار از v_j
- اندازه ورودی: n، تعداد رئوس در گراف
- پیچیدگی زمانی:

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

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

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

$$T(n) = (n\text{-}1)(n\text{-}2)2^{n\text{-}3} \in \Theta(n^2 2^n)$$

- پیچیدگی حافظه:

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

آیا چنین الگوریتمی می تواند مفید باشد؟

- رالف و نانسی در حال رقابت برای تصاحب یک موقعیت شغلی
- مسابقه برای تصاحب موقعیت شغلی در شرکت:
  - هرکسی که سریعتر سفر به ۲۰ شهر مشخص را انجام دهد و به شرکت برگردد برنده است. بین هر دو شهر یک جاده وجود دارد.
  - رالف : الگوریتم غیر هوشمند

$(20-1)!\ \mu s = 19!\ \mu s = 3857$ سال

  - نانسی: الگوریتم برنامه نویسی پویا

$(20-1)(20-2)2^{20-3}\ \mu s = 45$ ثانیه

$20 * 2^{20} = 20,971,520$ عنصر آرایه

اگر تعداد شهر ها 60 باشد: الگوریتم برنامه نویسی پویا نیز سال ها وقت لازم دارد.

تعیین تور بهینه

- اندیس اولین گره (گره بعد از v_1) =

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

- اندیس دومین گره =

$$P[3][\{v_2, v_4\}] = 4$$

- اندیس سومین گره =

$$P[4][\{v_2\}] = 2$$

- تور بهینه =

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

مسائل NP-Complete

- مساله فروشنده دوره گرد:
  - تاکنون هیچ کس نتوانسته برای مساله فروشنده دوره گرد الگوریتمی بیابد که پیچیدگی زمانی آن در بدترین حالت بهتر از نمایی باشد
  - همچنین، تاکنون کسی هم نتوانسته عدم امکان چنین الگوریتمی را اثبات کند.

- خانواده بزرگی از مسایل مرتبط وجود دارند که در این ویژگی مشترک می باشند که به این خانواده از مسایل، NP-Complete گفته می شود. (ر.ک. فصل ۹)

تمرین ها

- بخش ۳–۱
  - ۴
- بخش ۳–۲
  - ۵و ۶و ۷
- بخش ۳–۴
  - ۱۷و ۱۸
- بخش ۳–۵
  - ۲۰ و ۲۴و ۲۶
- تمرین های اضافی
  - ۲۹و ۳۰و ۳۱