DYNAMIC PROGRAMMING

Omid R. B. Speily

Reference:

Dr. M. Ghodsi, Sharif University of Technology

Dr. N. Razavi, Iran University of Science & Technology

ضرب ماتریس‌ها

n ماتریس را می‌خواهیم درهم ضرب کنیم:

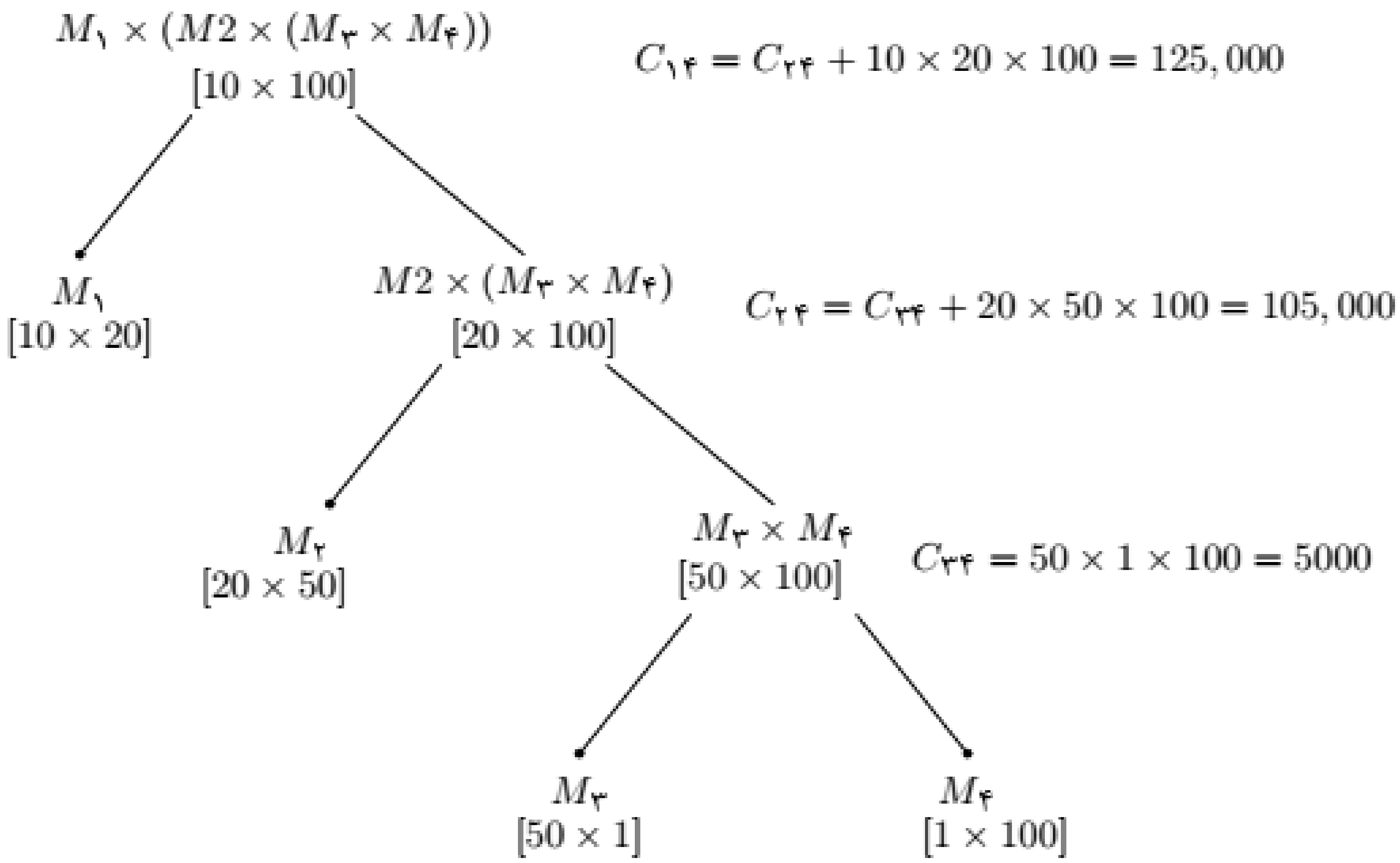
$$M_۱ \times M_۲ \times \cdots \times M_n$$

ابعاد M_i برابر است با: $d_{i-۱} \times d_i$

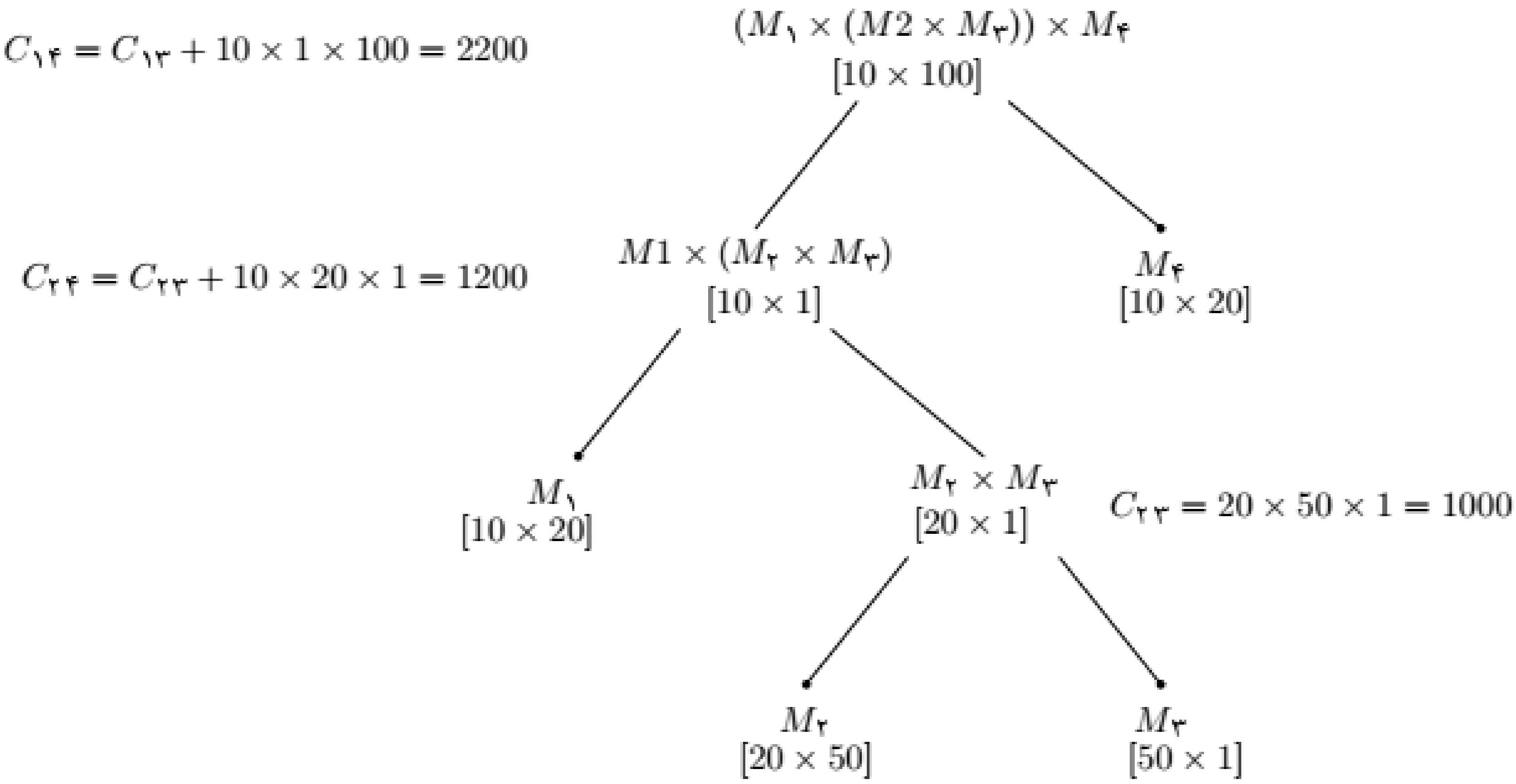
می‌خواهیم این ضرب‌ها را به‌ترتیبی انجام دهیم که هزینه‌ی کل کمینه شود.

مثال

$$M = \underset{[۱۰ \times ۲۰]}{M_۱} \times \underset{[۲۰ \times ۵۰]}{M_۲} \times \underset{[۵۰ \times ۱]}{M_۳} \times \underset{[۱ \times ۱۰۰]}{M_۴}$$



رتیب و هزینه‌ی ضرب ماتریس‌ها در محاسبه‌ی $M_{1} \times (M_{2} \times (M_{3} \times M_{4}))$



ترتیب و هزینه‌ی ضرب ماتریس‌ها در محاسبه‌ی $(M_1 \times (M_2 \times M_3)) \times M_4$.

راه‌حل بازگشتی

زیر مسئله:

$$M_{ij} = M_i \times M_{i+1} \times \cdots \times M_j$$

هزینه‌ی بهینه برای M_{ij} : C_{ij}

$$C_{ij} = \begin{cases} 0 & if \ \ i = j \\ \min\limits_{i \leq k < j} \{C_{ik} + C_{k+1,j} + d_{i-1} d_k d_j\} & if \ \ i < j \end{cases}$$

راه‌حل بازگشتی

<u>Recursive-Matrix-Multiplication</u>(d, i, j)

▷ Computes $C[i, j]$, the optimal cost of $M_i \times M_{i+1} \times \cdots \times M_j$

if $i = j$
  then return 0
$C[i, j] \leftarrow \infty$
for $k \leftarrow i$ **to** $j - 1$
    do $q \leftarrow$ Recursive-Matrix-Multiplication(d, i, k) $+$
        Recursive-Matrix-Multiplication$(d, k + 1, j)$ $+$
$d_{i-1} d_k d_j$
      if $q < C[i, j]$
        then $C[i, j] \leftarrow q$
return $C[i, j]$

تحلیل

$T(n)$ زمان اجرای $\textsc{Recursive-Matrix-Multiplication}(d, ۱, n)$

$$
\begin{aligned}
T(۱) &\geq ۱ \\
T(n) &\geq ۱ + \sum_{k=۱}^{n-۱} [T(k) + T(n-k) + ۱], \ \text{for} \ n > ۱
\end{aligned}
$$

در این رابطه، هر $T(i)$ دو بار تکرار می‌شود، پس

$$
T(n) \geq ۲ \sum_{i=۱}^{n-۱} T(i) + n
$$

$$
\begin{aligned}
T(n) &\geq ۲\sum_{i=۱}^{n-۱} ۲^{i-۱} + n \\
&= ۲\sum_{i=۰}^{n-۲} ۲^{i} + n \\
&= ۲(۲^{n-۱} - ۱) + n \\
&= (۲^{n} - ۲) + n \\
&\geq ۲^{n-۱}
\end{aligned}
$$

از طرفی دیگر، این الگوریتم در واقع همه‌ی حالات ضرب n ماتریس را بررسی می‌کند و بین آن‌ها حالت بهینه را به‌دست می‌آورد.

هزینه‌ی آن متناسب با تعداد حالات ممکن ضرب و یا پرانتزگذاری ضرب این ماتریس‌هاست که آن‌را با $T(n)$ نمایش می‌دهیم.

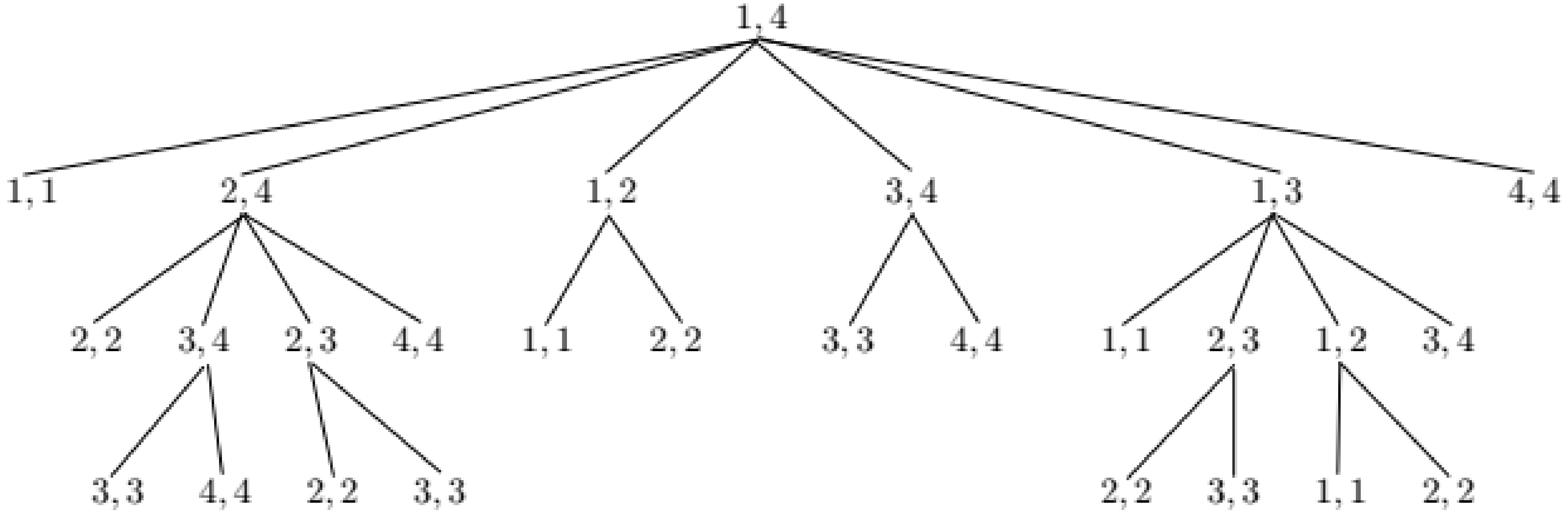
داریم،

$$T(n) = \begin{cases} ۱ & n = ۱ \\ \sum\limits_{i=۱}^{n-۱} T(i)T(n-i) & n > ۱ \end{cases}$$

می‌توان نشان داد که $T(n) = C(n-۱)$ که $C(n)$ nامین عدد کاتالان و برابر است با

$$C(n) = \frac{۱}{n+۱}\binom{۲n}{n} = \Omega(۴^n/n^{\frac{۳}{۲}})$$

زیرمسئله‌های تکراری



درخت فراخوانی‌ها برای $\textsc{Recursive-Matrix-Multiplication}(d, ۱, ۴)$

راه‌حل پویا

مسئله راه‌حل پویا دارد، چرا؟

زیرمسئله:

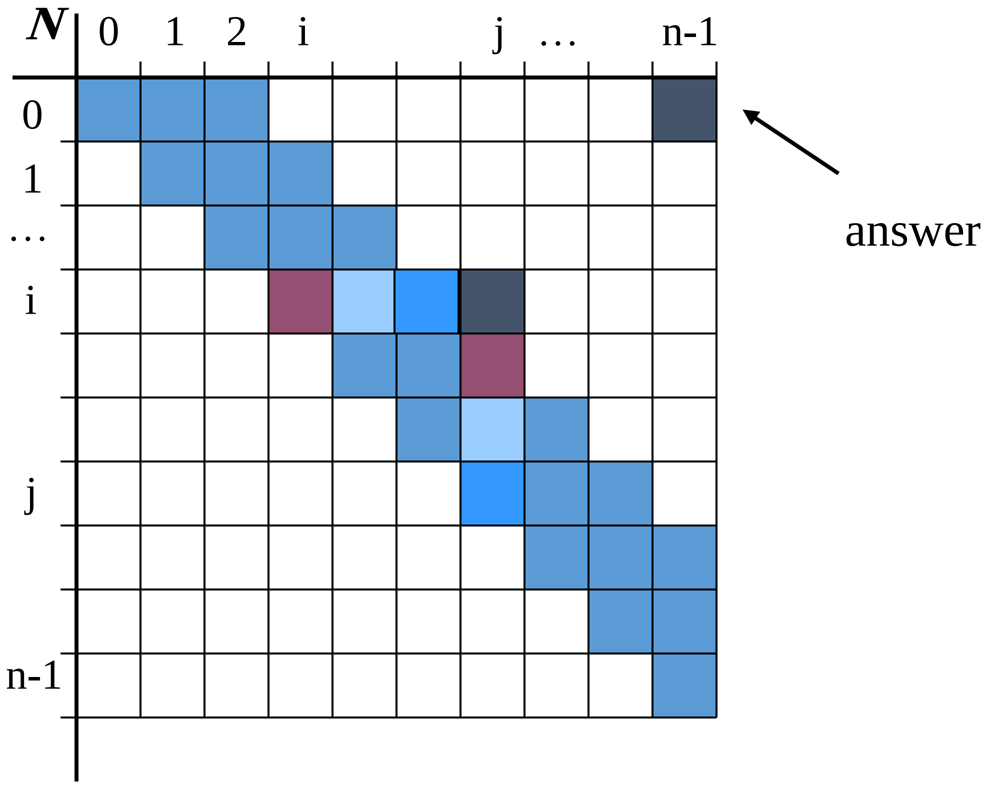
$$M_{ij} = M_i \times M_{i+1} \times \cdots \times M_j$$

بدیهی است که برای حل بهینه‌ی مسئله همه‌ی زیرمسئله‌ها هم باید بهینه حل شوند!

اگر C_{ij} هزینه‌ی بهینه‌ی M_{ij} باشد، داریم: $C_{ii} = 0$ و

$$C_{ij} = \min_{i \le k < j} \{C_{ik} + C_{k+1,j} + d_{i-1} d_k d_j\}$$

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$



الگوریتم پویا

```
DYBNAMIC-MATRIX-MULTIPLICATIONI(d)
for i ← 1 to n
    do C[i, i] ← 0
for l ← 2 to n
    do
        ▷ l = number of multiplied matrices
        for i ← 1 to n − l + 1
            do j ← i + l − 1
                ▷ solving M_ij
                for k ← i to j − 1
                    do C[i, j] ← min{C[i, k] + C[k + 1, j]} + d[i − 1] * d[k] * d[j]
                       R[i, j] ← the value of k that makes the minimum
```

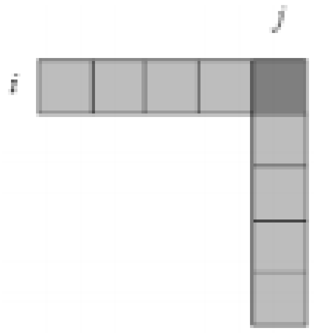
نوشتن پرانتزگذاری بهینه

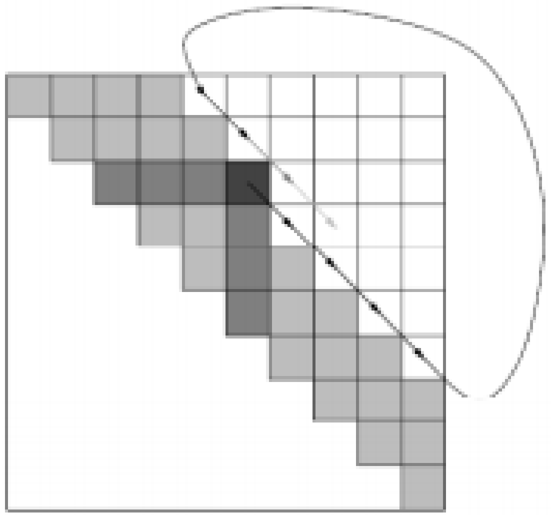
```
PRINTRESULTS(i, j)
if i ≠ j
  then k ← R[i, j]
       PRINT '('
       PRINTRESULTS(i, k)
       PRINT '×'
       PRINTRESULTS (k + 1, j)
       PRINT ')'
```

تحلیل: زمان $\mathcal{O}(n^3)$ و حافظه‌ی $\mathcal{O}(n^2)$

مرحله دوم) پر نمودن ماتریس M

$$M[i,\ j] = \min_{i \le k < j} \{M[i, k] + M[k+1,\ j] + d_{i-1} \cdot d_k \cdot d_j\}$$





مثال: محاسبه عناصر قطر ۱

$$d_0 = 10, d_1 = 20, d_2 = 50, d_3 = 1, d_4 = 100$$

قطر ۱

0	10K		
	0	1K	
		0	5K
			0

$$M[1,2] = \min_{1 \le k < 2}\{M[1,k] + M[k+1,2] + d_0 d_k d_2\}$$
$$= d_0 d_1 d_2 = 10{,}000$$

$$M[2,3] = d_1 d_2 d_3 = 1{,}000$$

$$M[3,4] = d_2 d_3 d_4 = 5{,}000$$

مثال: محاسبه عناصر قطر ۲

$$d_0 = 10, d_1 = 20, d_2 = 50, d_3 = 1, d_4 = 100$$

$$M[2,4] = \min_{2 \le k < 4}\{M[2,k] + M[k+1,4] + d_1 d_k d_4\}$$

$$= \min\{M[2,2] + M[3,4] + d_1 d_2 d_4,$$

$$M[2,3] + M[4,4] + d_{\cdot} d_3 d_4\}$$

$$= \min\{0 + 5000 + 100000, 1000 + 0 + 2000\}$$

$$= 3000$$

قطر ۲

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

مثال: محاسبه عناصر قطر ۳

$$d_0 = 10, d_1 = 20, d_2 = 50, d_3 = 1, d_4 = 100$$

$$\begin{aligned} M[1,4] &= \min_{1 \le k < 4}\{M[1,k] + M[k+1,4] + d_0 d_k d_4\} \\ &= \min\{M[1,1] + M[2,4] + d_0 d_1 d_4, \\ &\qquad M[1,2] + M[3,4] + d_0 d_2 d_4, \\ &\qquad M[1,3] + M[4,4] + d_0 d_3 d_4\} \\ &= 2200 \end{aligned}$$

قطر ۳

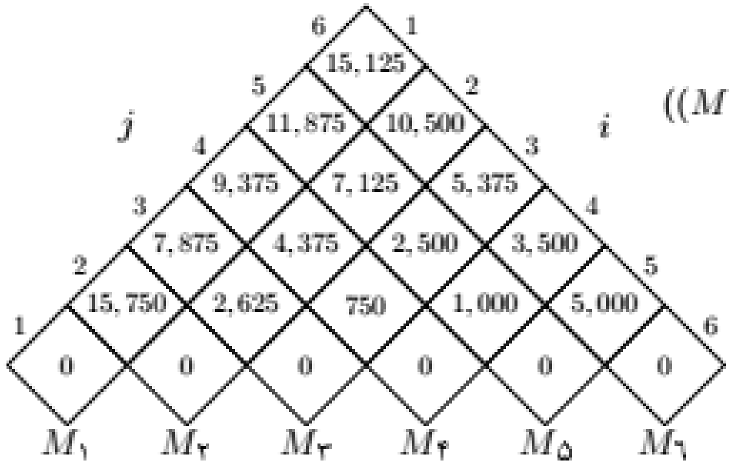
0	10K	1.2K	2.2K
	0	1K	3K
		0	5K
			0

مثال

ضرب شش ماتریس با اندازه‌های زیر:

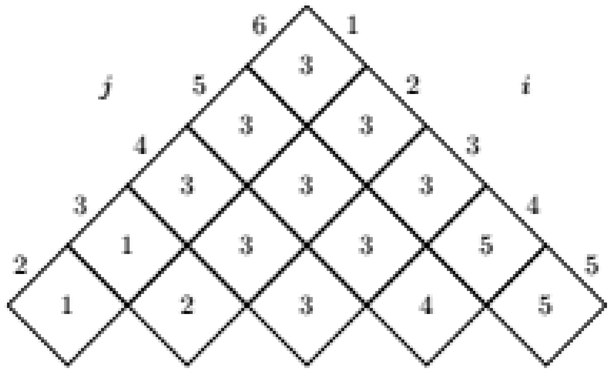
$$\begin{array}{ccccccccccc} M_{۱} & \times & M_{۲} & \times & M_{۳} & \times & M_{۴} & \times & M_{۵} & \times & M_{۶} \\ [۳۰ \times ۳۵] & & [۳۵ \times ۱۵] & & [۱۵ \times ۵] & & [۵ \times ۱۰] & & [۱۰ \times ۲۰] & & [۲۰ \times ۲۵] \end{array}$$

C[i,j]



$$((M_۱ \times (M_۲ \times M_۳)) \times ((M_۴ \times M_۵) \times M_۶))$$

R[i,j]



$$
\begin{aligned}
C[2,5] &= \min \begin{cases} C[2,2] + C[3,5] + d_۱ d_۲ d_۵ = 0 + 2500 + 35 * 15 * 20 & = 13000, \\ C[2,3] + C[4,5] + d_۱ d_۳ d_۵ = 2625 + 1000 + 35 * 5 * 20 & = 7125, \\ C[2,4] + C[5,5] + d_۱ d_۴ d_۵ = 4375 + 0 + 35 * 10 * 20 & = 11375 \end{cases} \\
&= 7125.
\end{aligned}
$$

روش به‌خاطر سپاری (Memoization)

- بازگشتی : از بالا به پایین و کورکورانه
- پویا : از پایین به بالا و ذخیره‌ی حاصل زیرمسئله‌ها
- به‌خاطرسپاری : از بالا به پایین ولی انجام ندادن کار تکراری

```
MEMOIZED-MATRIX-MULTIPLICATION (d)
n ← length[d] − 1
for i ← 1 to n
    do for j ← i to n
           do C[i, j] ← ∞
```

```
LOOKUP-MATRIX (p, 1, n)
if C[i, j] < ∞
  then return C[i, j]
if i = j
  then C[i, j] ← 0
  elsefor k ← i to j − 1
          do q ← LOOKUP-MATRIX(d, i, k) +
                 LOOKUP-MATRIX(d, k + 1, j) + d_{i−1} d_k d_j
             if q < C[i, j]
               then C[i, j] ← q
return C[i, j]
```

این روش نیز مانند روش پویا از مرتبه‌ی $\Theta(n^3)$ است، چرا که $\Theta(n^2)$ درایه‌ی ماتریس C هر یک فقط یک‌بار و هر بار با مرتبه‌ی $\Theta(n)$ حساب می‌شود.

درخت های جستجوی دودویی بهینه

- درخت جستجوی دودویی

تعریف

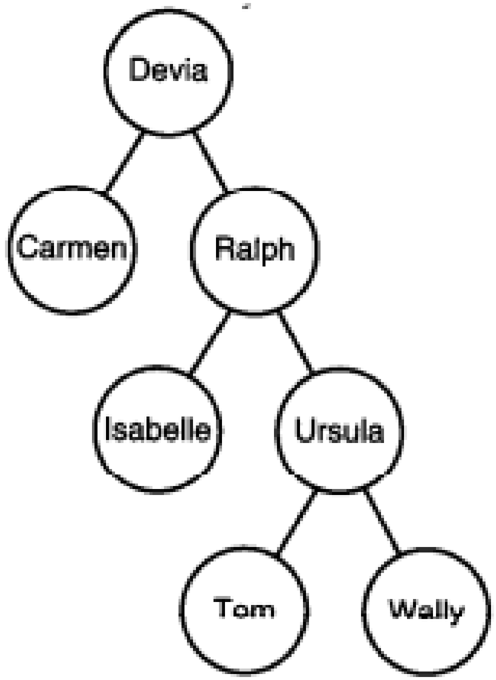
درخت جستجوی دودویی یک درخت دودویی از عناصر (کلید ها) است، که از یک مجموعه مرتب حاصل می شود، به طوری که

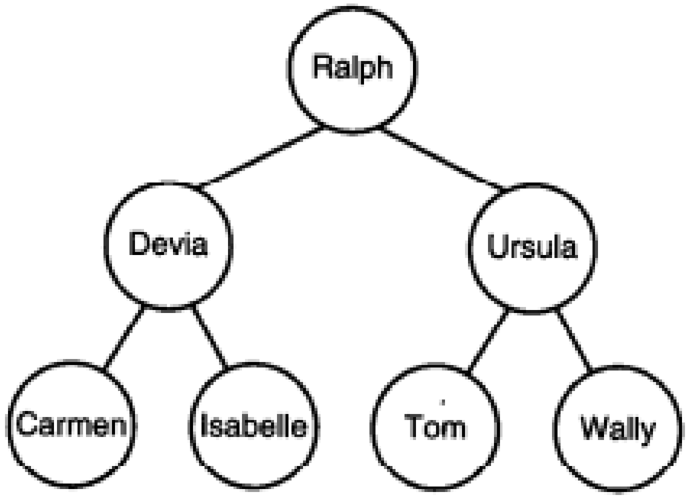
۱. هر گره دارای یک کلید می باشد.

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


Devia
Carmen
Ralph
Isabelle
Ursula
Tom
Wally


Ralph
Devia
Ursula
Carmen
Isabelle
Tom
Wally

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.

مثال 3.7

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

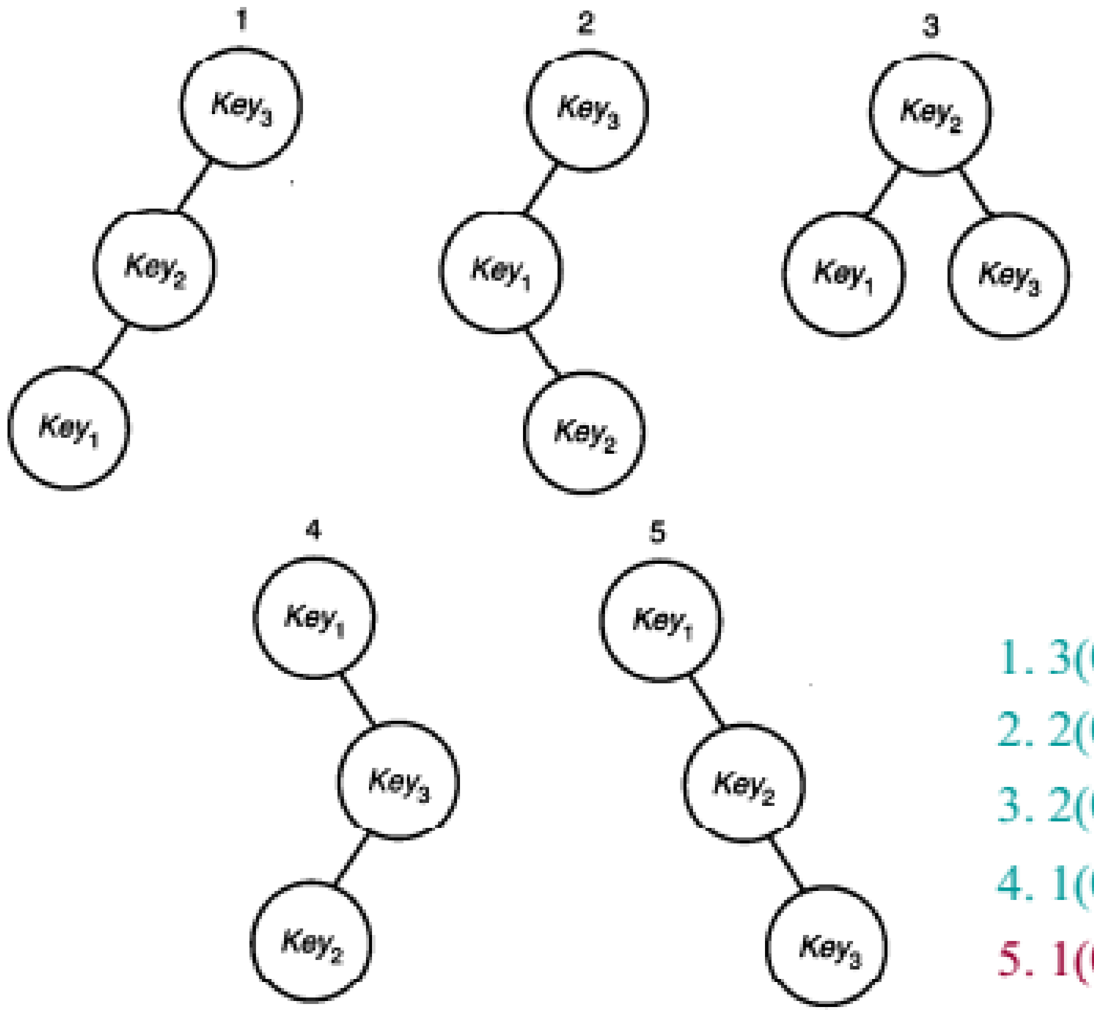


Figure 3.11 • The possible binary search trees when there are three keys.

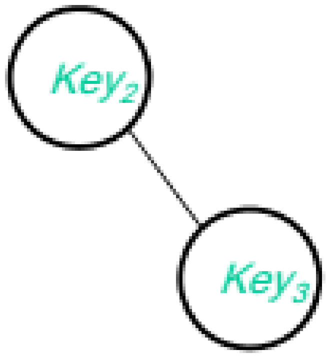
1. $3(0.7) + 2(0.2) + 1(0.1) = 2.6$
2. $2(0.7) + 3(0.2) + 1(0.1) = 2.1$
3. $2(0.7) + 1(0.2) + 2(0.1) = 1.8$
4. $1(0.7) + 3(0.2) + 2(0.1) = 1.5$
5. $1(0.7) + 2(0.2) + 3(0.1) = 1.4$

توسعه یک الگوریتم کارآ

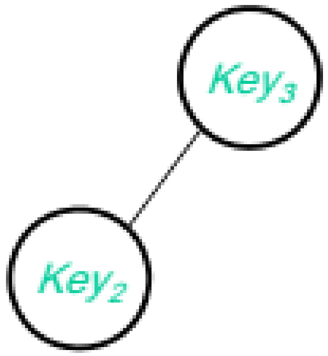
- جستجوی Brute force حداقل نمایی است
  - تعداد درخت های جست و جوی دودویی متفاوت با عمق $n-1$ برابر است با 2^{n-1}
- اصل بهینگی برقرار است.
- اگر $A[i][j]$ برابر حداقل مقدار $\sum_{m=i}^{j} c_m p_m$ باشد.
- $A[i][i] = p_i$

مثال 3.8

- محاسبه $A[2][3]$ در مثال قبل
- $p_1 = 0.7, \quad p_2 = 0.2, \quad p_3 = 0.1$



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



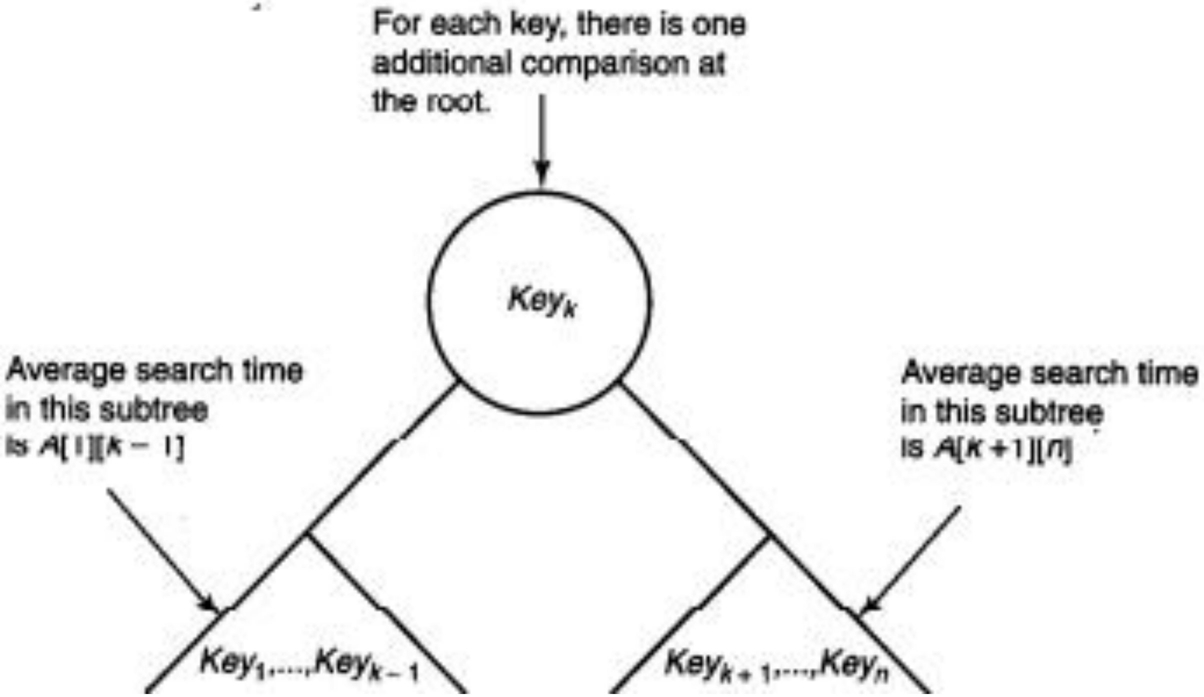
$2(p_2) + 1(p_3) = 2(0.2) + 1(0.1) = 0.5$

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 باشا



متوسط زمان جستجو

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

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

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

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

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

$A[i][i-1]$ 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_{i<=k<=j}(A[i][k - 1] + A[k + 1][j]) + sum_{m=i}^{j} p_m;
      R[i][j] = a value of k that gave the minimum;
    }
minavg = A[1][n];
}
```

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

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:

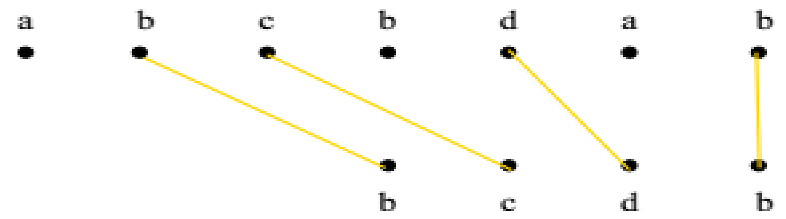
	0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

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

بزرگ‌ترین زیردنباله‌ی مشترک



$Z =< z_1, z_2, \cdots, z_k >$ زیردنباله‌ی $X =< x_1, x_2, \cdots, x_m >$ است اگر دنباله‌ی اکیداً صعودی $< i_1, i_2, \cdots, i_k >$ از اندیس‌های عناصر X وجود داشته باشد به‌طوری‌که برای $j = 1, \ldots, k$ داشته باشیم: $z_i = x_{i_j}$.

مثلاً $Z =< b, c, d, b >$ یک زیردنباله از $X =< a, b, c, b, d, a, b >$ است که دنباله‌ی اندیس‌های مربوط $< 2, 3, 5, 7 >$ می‌باشد.

راه‌حل پویا

اگر $c[i,j]$ طول LCS برای X_i و Y_j باشد،

$$c[i,j] = \begin{cases} ۰ & \text{if } i = ۰ \text{ or } j = ۰, \\ c[i-۱, j-۱] + ۱ & \text{if } i,j > ۰ \text{ and } x_i = y_j, \\ \max(c[i, j-۱], c[i-۱, j]) & \text{if } i,j > ۰ \text{ and } x_i \neq y_j. \end{cases}$$

راه‌حل پویا

اگر $c[i,j]$ طول LCS برای X_i و Y_j باشد،

$$c[i,j] = \begin{cases} ۰ & \text{if } i = ۰ \text{ or } j = ۰, \\ c[i-۱, j-۱] + ۱ & \text{if } i,j > ۰ \text{ and } x_i = y_j, \\ \max(c[i, j-۱], c[i-۱, j]) & \text{if } i,j > ۰ \text{ and } x_i \neq y_j. \end{cases}$$

```
for i := 1 to m do c[i,0] := 0;
for j := 1 to m do c[0,j] := 0;
for i := 1 to m do
    for j := 1 to n do
        if x_i = y_j then begin
            c[i,j] := c[i-1, j-1] + 1;
            b[i,j] := '↖'
        end
        else if c[i-1, j] ≥ c[i, j-1]
            then begin
                c[i,j] := c[i-1, j];
                b[i,j] := '↑'
            end
            else begin
                c[i,j] := c[i, j-1];
                b[i,j] := '←'
            end
```

الگوریتم ارائه شده از مرتبه‌ی زمانی $O(mn)$ و میزان حافظه‌ی مصرفی آن نیز $O(mn)$

مثال: $X =< A, B, C, B, D, A, B >$ و $Y =< B, D, C, A, B, A >$

	j	0	1	2	3	4	5	6
i		y_j	b	d	c	b	b	a
0	x_i	0	0	0	0	0	0	0
1	a	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	b	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	c	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	b	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	d	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	a	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	b	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

```
PRINT-LCS(b, X, i, j)
if (i = 0 or j = 0)
   then return;
if b[i, j] = '↖'
   then begin
         PRINT-LCS (b, X, i − 1, j − 1);
         print x_i
   end
   else if b[i, j] = '↑'
        then PRINT-LCS (b, X, i − 1, j)
        else PRINT-LCS (b, X, i − 1, j)
```

این الگوریتم از مرتبه‌ی $O(m+n)$ است.