# 15.Dynamic Programming

Hsu, Lih-Hsing



- ***Dynamic programming*** is typically applied to optimization problems. In such problem there can be ***many solutions***. Each solution has a value, and we wish to find *a* ***solution*** with the optimal value.

- The development of a dynamic programming algorithm can be broken into a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom up fashion.
4. Construct an optimal solution from computed information.





# 15.1 Assembly-line scheduling

An automobile chassis enters each assembly line, has parts added to it at a number of stations, and a finished auto exits at the end of the line.

Each assembly line has n stations, numbered j = 1, 2,...,n. We denote the jth station on line j ( where i is 1 or 2) by $S_{i,j}$. The jth station on line 1 ($S_{1,j}$) performs the same function as the jth station on line 2 ($S_{2,j}$).

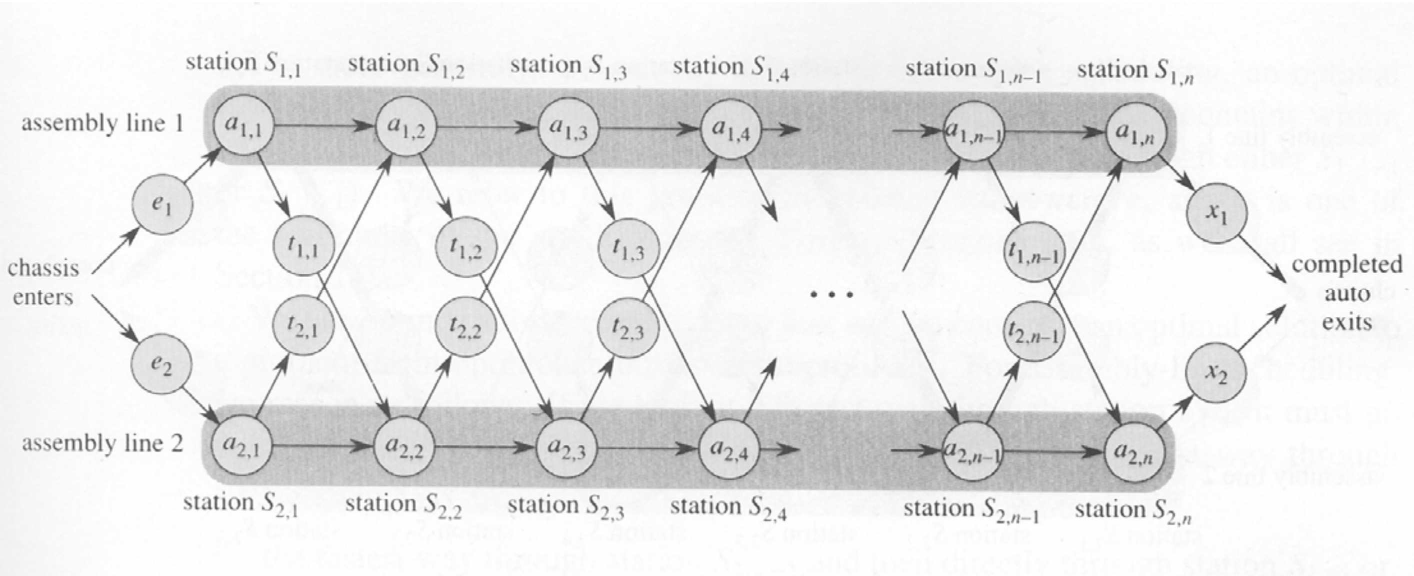
The stations were built at different times and with different technologies, however, so that the time required at each station varies, even between stations at the same position on the two different lines. We denote the assembly time required at station $S_{i,j}$ by $a_{i,j}$.

As the coming figure shows, a chassis enters station 1 of one of the assembly lines, and it progresses from each station to the next. There is also an entry time $e_i$ for the chassis to enter assembly line i and an exit time $x_i$ for the completed auto to exit assembly line i.





## a manufacturing problem to find the fast way through a factory



Normally, once a chassis enters an assembly line, it passes through that line only. The time to go from one station to the next within the same assembly line is negligible.

Occasionally a special rush order comes in, and the customer wants the automobile to be manufactured as quickly as possible.

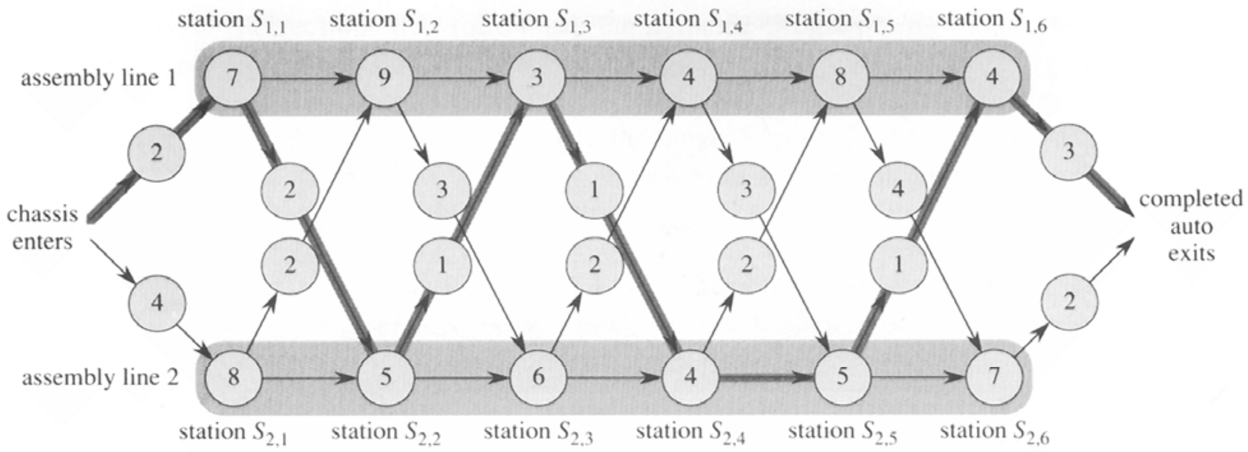
For the rush orders, the chassis still passes through the n stations in order, but the factory manager may switch the partially-completed auto from one assembly line to the other after any station.





The time to transfer a chassis away from assembly line i after having gone through station $S_{ij}$ is $t_{i,j}$, where i = 1, 2 and j = 1, 2, …, n-1 (since after the nth station, assembly is complete). The problem is to determine which stations to choose from line 1 and which to choose from line 2 in order to minimize the total time through the factory for one auto.

# An instance of the assembly-line problem with costs




(a)

| $j$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $f_1[j]$ | 9 | 18 | 20 | 24 | 32 | 35 |
| $f_2[j]$ | 12 | 16 | 22 | 25 | 30 | 37 |

$f^* = 38$

| $j$ | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| $l_1[j]$ | 1 | 2 | 1 | 1 | 2 |
| $l_2[j]$ | 1 | 2 | 1 | 2 | 2 |

$l^* = 1$

(b)





# Step1 The structure of the fastest way through the factory

- the fast way through station $S_{1,j}$ is either
  - the fastest way through Station $S_{1,j-1}$ and then directly through station $S_{1,j}$, or
  - the fastest way through station $S_{2,j-1}$, a transfer from line 2 to line 1, and then through station $S_{1,j}$.
- Using symmetric reasoning, the fastest way through station $S_{2,j}$ is either
  - the fastest way through station $S_{2,j-1}$ and then directly through Station $S_{2,j}$, or
  - the fastest way through station $S_{1,j-1}$, a transfer from line 1 to line 2, and then through Station $S_{2,j}$.

## Step 2: A recursive solution

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$$

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if} \quad j = 1, \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if} \quad j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if} \quad j = 1, \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if} \quad j \geq 2 \end{cases}$$



Computer Theory Lab.

## step 3: computing the fastest times

- Let $r_i(j)$ be the number of references made to $f_i[j]$ in a recursive algorithm.

  $r_1(n)=r_2(n)=1$

  $r_1(j) = r_2(j)=r_1(j+1)+r_2(j+1)$
- The total number of references to all $f_i[j]$ values is $\Theta(2^n)$.
- We can do much better if we compute the $f_i[j]$ values in different order from the recursive way. Observe that for $j \geq 2$, each value of $f_i[j]$ depends only on the values of $f_1[j-1]$ and $f_2[j-1]$.

# FASTEST-WAY procedure

FASTEST-WAY$(a, t, e, x, n)$

$f_1[1] \leftarrow e_1 + a_{1,1}$

$f_2[1] \leftarrow e_2 + a_{2,1}$

**for** $j \leftarrow 2$ to $n$

    **do if** $f_1[j\text{-}1] + a_{1,j} \leq f_2[j\text{-}1] + t_{2,j\text{-}1} + a_{1,j}$

        **then** $f_1[j] \leftarrow f_1[j\text{-}1] + a_{1,j}$

            $l_1[j] \leftarrow 1$

        **else** $f_1[j] \leftarrow f_2[j\text{-}1] + t_{2,j\text{-}1} + \mathrm{a}_{1,j}$

            $l_1[j] \leftarrow 2$

      **if** $f_2[j\text{-}1] + a_{2,j} \leq f_1[j\text{-}1] + \mathrm{t}_{1,j\text{-}1} + \mathrm{a}_{2,j}$





        **then** $f_2[j] \leftarrow f_2[j-1] + \mathrm{a}_{2,j}$

            $l2[j] \leftarrow 2$

        **else** $f_2[j] \leftarrow f_1[\mathrm{j}-1] + t_{1,j\text{-}1} + a_2,j$

            $l_2[j] \leftarrow 1$

**if** $f_1[n] + x_1 \leq f_2[n] + x_2$

    **then** $f^* = f_1[n] + x_1$

        $l^* = 1$

    **else** $f^* = f_2[n] + x_2$

        $l^* = 2$

## step 4: constructing the fastest way through the factory

PRINT-STATIONS($l$, $n$)

i ← l*
print "line" i ",station" n
for $j$ ← n downto 2
    do i ← $l_i[j]$
        print "line" i ",station" j – 1

output

line 1, station 6
line 2, station 5
line 2, station 4
line 1, station 3
line 2, station 2
line 1, station 1





## 15.2 Matrix-chain multiplication

- A product of matrices is **fully parenthesized** if it is either a single matrix, or a product of two fully parenthesized matrix product, surrounded by parentheses.

- How to compute $A_1 A_2 \ldots A_n$ where $A_i$ is a matrix for every *i*.
- Example: $A_1 A_2 A_3 A_4$

$$(A_1(A_2(A_3A_4))) \quad (A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4)) \quad ((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$





# MATRIX MULTIPLY

MATRIX MULTIPLY(*A*,*B*)

**if** columns[*A*] $\neq$ column[*B*]
**then error** "incompatible dimensions"
**else for** $i \leftarrow 1$ **to** rows[*A*]
**do for** $j \leftarrow 1$ **to** columns[*B*]
**do** $c[i, j] \leftarrow 0$
**for** $k \leftarrow 1$ **to** columns[*A*]
**do** $c[i, j] \leftarrow c[i, j] + A[i,k]B[k, j]$
**return** *C*

# Complexity:

- Let $A$ be a $p \times q$ matrix, and $B$ be a $q \times r$ matrix. Then the complexity is $p \times q \times r$.



Computer Theory Lab.

# Example:

- $A_1$ is a $10 \times 100$ matrix, $A_2$ is a $100 \times 5$ matrix, and $A_3$ is a $5 \times 50$ matrix. Then $((A_1 A_2) A_3)$ takes $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ time. However $(A_1 (A_2 A_3))$ takes $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$ time.

# The matrix-chain multiplication problem:

- Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of $n$ matrices, where for $i=0,1,\ldots,n$, matrix $A_i$ has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.





# Counting the number of parenthesizations:

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)p(n-k) & \text{if } n \geq 2 \end{cases}$$

- $P(n) = C(n-1)$ [Catalan number]

$$= \frac{1}{n+1}\binom{2n}{n} = \Omega(\frac{4^n}{n^{3/2}})$$

# Step 1: The structure of an optimal parenthesization

Optimal

$$((A_1 A_2 \ldots A_k)(A_{k+1} A_{k+2} \ldots A_n))$$

Combine





# Step 2: A recursive solution

- Define $m[i, j]$= minimum number of scalar multiplications needed to compute the matrix $A_{i..j} = A_i A_{i+1} .. A_j$
- goal $m[1, n]$
- $m[i, j] =$

$$\begin{cases} 0 & i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1, j] + p_{i-1} p_k p_j\} & i \ne j \end{cases}$$

# Step 3: Computing the optimal costs



# MATRIX_CHAIN_ORDER

```
MATRIX_CHAIN_ORDER(p)
n ← length[p] –1
for  i ← 1 to n
        do m[i, i] ← 0
for  l ← 2 to n
        do for  i ← 1 to n – l + 1
                      do  j ← i + l – 1
                            m[i, j] ← ∞
                            for  k ← i  to  j – 1
                                   do  q ← m[i, k] + m[k+1, j]+ p_{i-1}p_k p_j
                                         if  q < m[i, j]
                                               then m[i, j] ← q
                                                      s[i, j] ← k
 return m and s
```

**Complexity:** $O(n^3)$

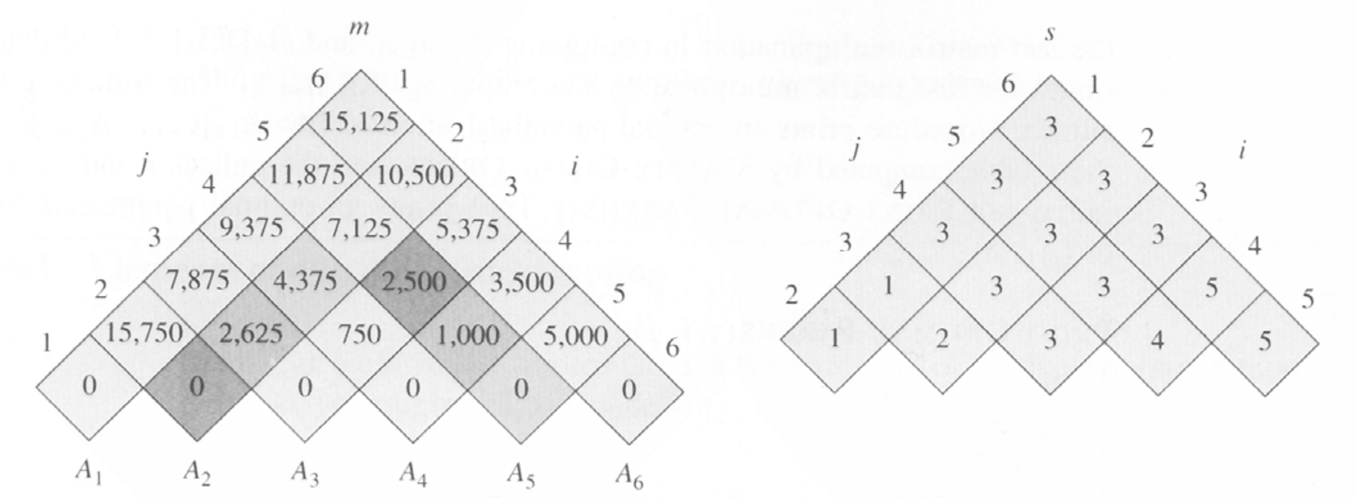
# Example:

$$
\begin{array}{lll}
A_1 & 30 \times 35 & = p_0 \times p_1 \\
A_2 & 35 \times 15 & = p_1 \times p_2 \\
A_3 & 15 \times 5 & = p_2 \times p_3 \\
A_4 & 5 \times 10 & = p_3 \times p_4 \\
A_5 & 10 \times 20 & = p_4 \times p_5 \\
A_6 & 20 \times 25 & = p_5 \times p_6
\end{array}
$$





# the m and s table computed by MATRIX-CHAIN-ORDER for n=6



$m[2,5]=$
**min{**
$m[2,2]+m[3,5]+p_1p_2p_5=0+2500+35\times15\times20=13000,$
$m[2,3]+m[4,5]+p_1p_3p_5=2625+1000+35\times5\times20=7125,$
$m[2,4]+m[5,5]+p_1p_4p_5=4375+0+35\times10\times20=11374$
**}**
$=7125$



# Step 4: Constructing an optimal solution

# MATRIX_CHAIN_MULTIPLY

MATRIX_CHAIN_MULTIPLY($A, s, i, j$)

1 **if** $j > i$

2 **then** $x \leftarrow MCM(A, s, i, s[i, j])$

3 $y \leftarrow MCM(A, s, s[i, j] + 1, j)$

4 **return** MATRIX-MULTIPLY($X, Y$)

5 **else return** $A_i$

- example: $((A_1(A_2A_3))((A_4A_5)A_6))$



# 16.3 Elements of dynamic programming

# Optimal substructure:

- We say that a problem exhibits **optimal substructure** if an optimal solution to the problem contains within its optimal solution to subproblems.

- Example: Matrix-multiplication problem





1. You show that a solution to the problem consists of making a choice, Making this choice leaves one or more subproblems to be solved.
2. You suppose that for a given problem, you are given the choice that leads to an optimal solution.
3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within the optimal solution to the problem must themselves be optimal by using a "*cut-and-paste*" technique.

Optimal substructure varies across problem domains in two ways:

1. how many subproblems are used in an optimal solutiion to the original problem, and

2. how many choices we have in determining which subproblem(s) to use in an optimal solution.





## Subtleties

- One should be careful not to assume that optimal substructure applies when it does not. consider the following two problems in which we are given a directed graph $G = (V, E)$ and vertices $u, v \in V$.
  - Unweighted shortest path:
    - Find a path from $u$ to $v$ consisting of the fewest edges. Good for Dynamic programming.
  - Unweighted longest simple path:
    - Find a simple path from $u$ to $v$ consisting of the most edges. Not good for Dynamic programming.

# Overlapping subproblems:

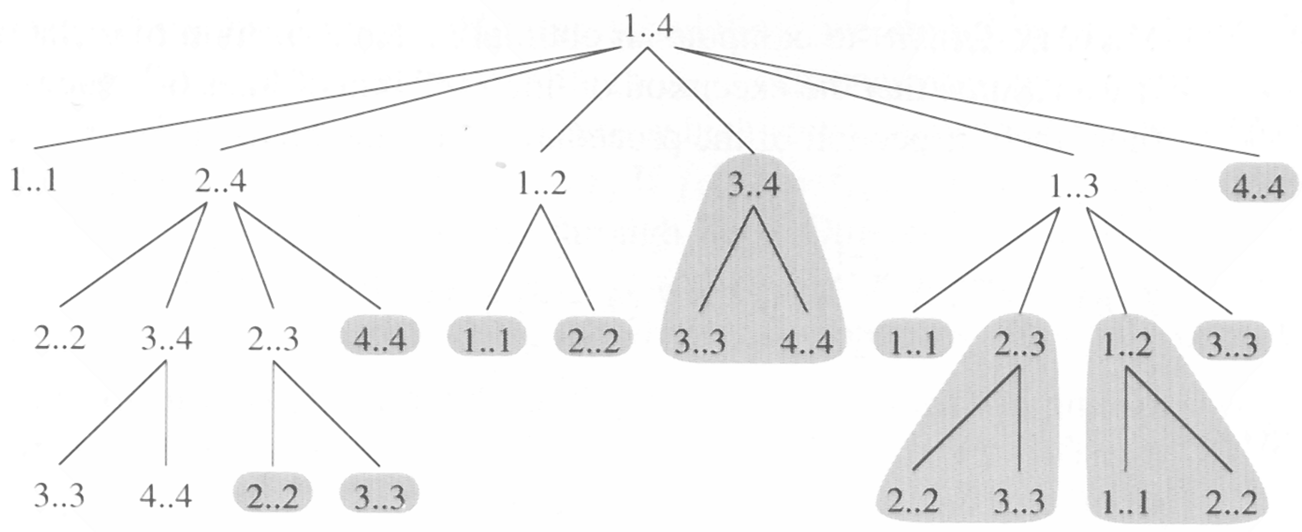
example:
MAXTRIX_CHAIN_ORDER



# RECURSIVE_MATRIX_CHAIN

RECURSIVE_MATRIX_CHAIN($p$, $i$, $j$)
**if** $i = j$
  **then return** 0
$m[i, j] \leftarrow \infty$
**for** $k \leftarrow i$ **to** $j - 1$
  **do** $q \leftarrow$ RMC($p$,$i$,$k$) + RMC($p$,$k$+1,$j$) + $p_{i-1}p_kp_j$
  **if** $q < m[i, j]$
    **then** $m[i, j] \leftarrow q$
**return** $m[i, j]$

# The recursion tree for the computation of RECURSUVE-MATRIX-CHAIN(P, 1, 4)








$$\begin{cases} T(1) \geq 1 \\ T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad for\ n > 1 \end{cases}$$

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

- We can prove that $T(n) = \Omega(2^n)$ using substitution method.

$$T(1) \geq 1 = 2^0$$

$$T(n) \geq 2\sum_{i=1}^{n-1} 2^{i-1} + n = 2\sum_{i=0}^{n-2} 2^i + n$$

$$= 2(2^{n-1} - 1) + n = (2^n - 2) + n \geq 2^{n-1}$$

Solution:

1. bottom up
2. memorization (memorize the natural, but inefficient)





# MEMORIZED_MATRIX_CHAIN

MEMORIZED_MATRIX_CHAIN(*p*)

1 $n \leftarrow length[p] - 1$

2 **for** $i \leftarrow 1$ **to** $n$

3 **do for** $j \leftarrow 1$ **to** $n$

4 **do** $m[i, j] \leftarrow \infty$

5 **return** $LC(p, 1, n)$

# LOOKUP_CHAIN

LOOKUP_CHAIN($p$, $i$, $j$)

```
if m[i, j] < ∞
    then return m[i, j]
if i = j
    then m[i, j] ← 0
    else for k ← i to j – 1
        do q ← LC(p, i, k) +LC(p, k+1, j)+p_{i-1}p_k p_j
           if q < m[i, j]
               then m[i, j] ← q
return m[i, j]
```

**Time Complexity:** $O(n^3)$





# 16.4 Longest Common Subsequence

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

- $\langle B, C, A \rangle$ is a common subsequence of both $X$ and $Y$.
- $\langle B, C, B, A \rangle$ or $\langle B, C, A, B \rangle$ is the longest common subsequence of $X$ and $Y$.

# Longest-common-subsequence problem:

- We are given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a maximum length common subsequence of $X$ and $Y$.
- We Define $X_i = \langle x_1, x_2, \ldots, x_i \rangle$.



Computer Theory Lab.

# **Theorem 16.1.** (Optimal substructure of LCS)

- Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be the sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of $X$ and $Y$.

1. If $x_m = y_n$
   then $z_k = x_m = y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$
   then $z_k \neq x_m$ implies $Z$ is an LCS of $X_{m-1}$ and $Y$.
3. If $x_m \neq y_n$
   then $z_k \neq y_n$ implies $Z$ is an LCS of $X$ and $Y_{n-1}$.

# A recursive solution to subproblem

- Define $c[i, j]$ is the length of the LCS of $X_i$ and $Y_j$.

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





# Computing the length of an LCS

LCS_LENGTH(X,Y)

$m \leftarrow length[X]$
$n \leftarrow length[Y]$
**for** $i \leftarrow 1$ **to** $m$
    **do** $c[i, 0] \leftarrow 0$
**for** $j \leftarrow 1$ **to** $n$
    **do** $c[0, j] \leftarrow 0$
**for** $i \leftarrow 1$ **to** $m$
    **do for** $j \leftarrow 1$ **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 c and b
```





| | $j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| $i$ | | $y_j$ | B | D | C | A | 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 |

Complexity: O($mn$)

# PRINT_LCS

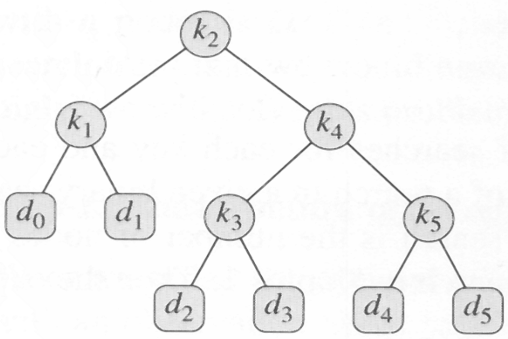
```
PRINT_LCS(b, X, c, j )
if i = 0 or j = 0
        then return
if b[i, j] = "↖"
        then PRINT_LCS(b, X, i-1, j-1)
             print x_i
else if b[i, j] = "↑"
        then PRINT_LCS(b, X, i-1, j)
then PRINT_LCS(b, X, i, j-1)
```

**Complexity: O($m+n$)**

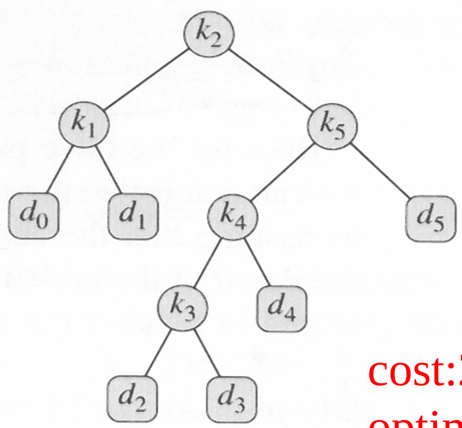




# 15.5 Optimal Binary search trees




cost:2.80




cost:2.75
optimal!!

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $p_i$ | | 0.15 | 0.10 | 0.05 | 0.10 | 0.20 |
| $q_i$ | 0.05 | 0.10 | 0.05 | 0.05 | 0.05 | 0.10 |

# expected cost

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

- the expected cost of a search in T is

$$\text{E[search cost in T]} = \sum_{i=1}^{n} (\text{depth}_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i)+1)\cdot q_i$$

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i)\cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i)\cdot q_i$$





| node | depth | probability | contribution |
|---|---|---|---|
| $k_1$ | 1 | 0.15 | 0.30 |
| $k_2$ | 0 | 0.10 | 0.10 |
| $k_3$ | 2 | 0.05 | 0.15 |
| $k_4$ | 1 | 0.10 | 0.20 |
| $k_5$ | 2 | 0.20 | 0.60 |
| $d_0$ | 2 | 0.05 | 0.15 |
| $d_1$ | 2 | 0.10 | 0.30 |
| $d_2$ | 3 | 0.05 | 0.20 |
| $d_3$ | 3 | 0.05 | 0.20 |
| $d_4$ | 3 | 0.05 | 0.20 |
| $d_5$ | 3 | 0.10 | 0.40 |
| Total | | | 2.80 |

- For a given set of probabilities, our goal is to construct a binary search tree whose expected search is smallest. We call such a tree an **optimal binary search tree**.

## Step 1: The structure of an optimal binary search tree

- Consider any subtree of a binary search tree. It must contain keys in a contiguous range $k_i, \ldots, k_j$, for some $1 \le i \le j \le n$. In addition, a subtree that contains keys $k_i, \ldots, k_j$ must also have as its leaves the dummy keys $d_{i-1}, \ldots, d_j$.
- If an optimal binary search tree T has a subtree $T'$ containing keys $k_i, \ldots, k_j$, then this subtree $T'$ must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$.





## Step 2: A recursive solution

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1, \\ \min\limits_{i \le r \le j} \{ e[i, r-1] + e[r+1, j] + w(i,j) \} & \text{if } i \le j. \end{cases}$$

## Step 3:computing the expected search cost of an optimal binary search tree

OPTIMAL-BST(*p*,*q*,*n*)

```
for i ← 1 to n + 1
   do e[i, i – 1] ← q_{i-1}
      w[i, i – 1] ← q_{i-1}
for l ← 1 to n
   do for i ← 1 to n – l + 1
         do j ← i + l – 1
            e[i, j] ← ∞
            w[i, j] ← w[i, j – 1] + p_j + q_j
```

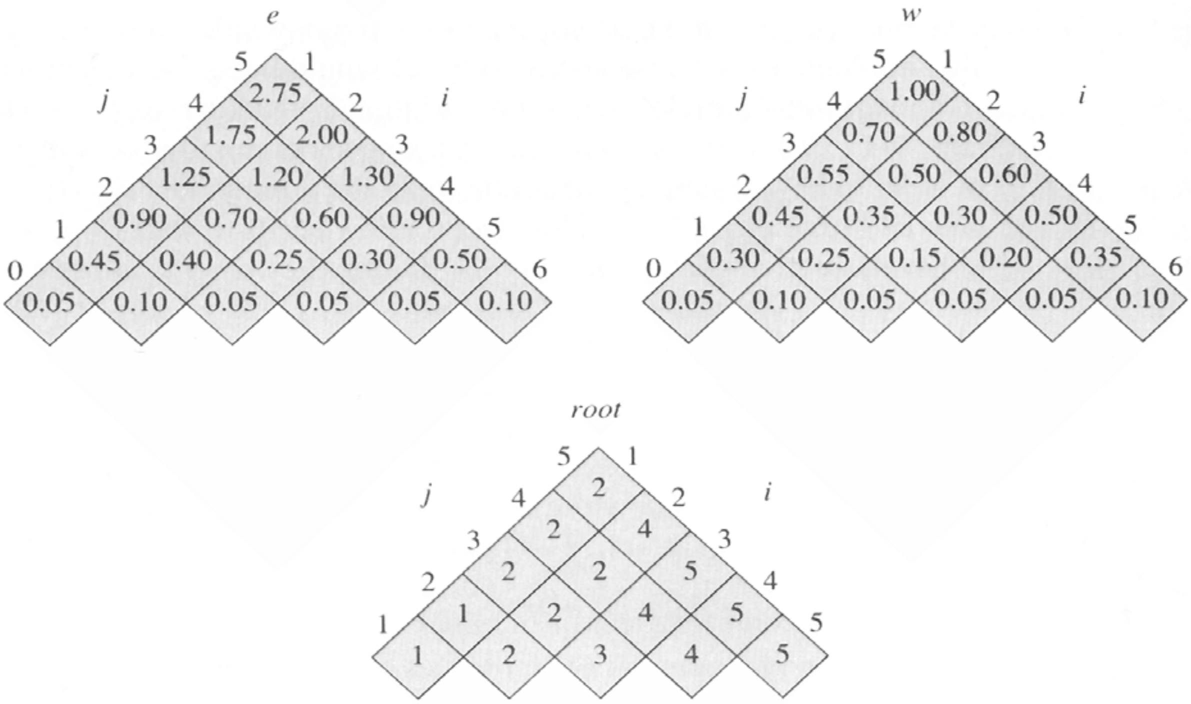




```
            for r ← i to j
               do t ← e[i, r –1]+e[r +1, j]+w[i, j]
                  if t < e[i, j]
                     then e[i, j] ← t
                          root[i, j] ← r
return e and root
```

- the OPTIMAL-BST procedure takes $\Theta(n^3)$, just like MATRIX-CHAIN-ORDER

## The table e[i,j], w[i,j], and root[i,j] computer by OPTIMAL-BST on the key distribution.








- *Knuth* has shown that there are always roots of optimal subtrees such that $root[i, j-1] \leq root[i+1, j]$ for all $1 \leq i \leq j \leq n$. We can use this fact to modify Optimal-BST procedure to run in $\Theta(n^2)$ time.