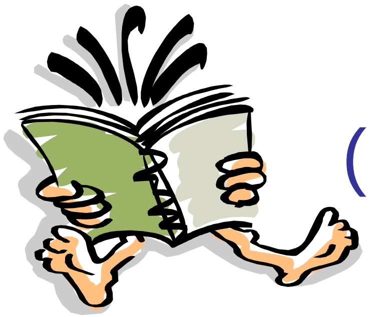
# Analysis of Algorithms
# CS 477/677

Dynamic Programming

Instructor: George Bebis



(Chapter 15)

# Dynamic Programming

- An algorithm design technique (like divide and conquer)
- Divide and conquer
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem

# Dynamic Programming

- Applicable when subproblems are **not** independent
  - Subproblems share subsubproblems

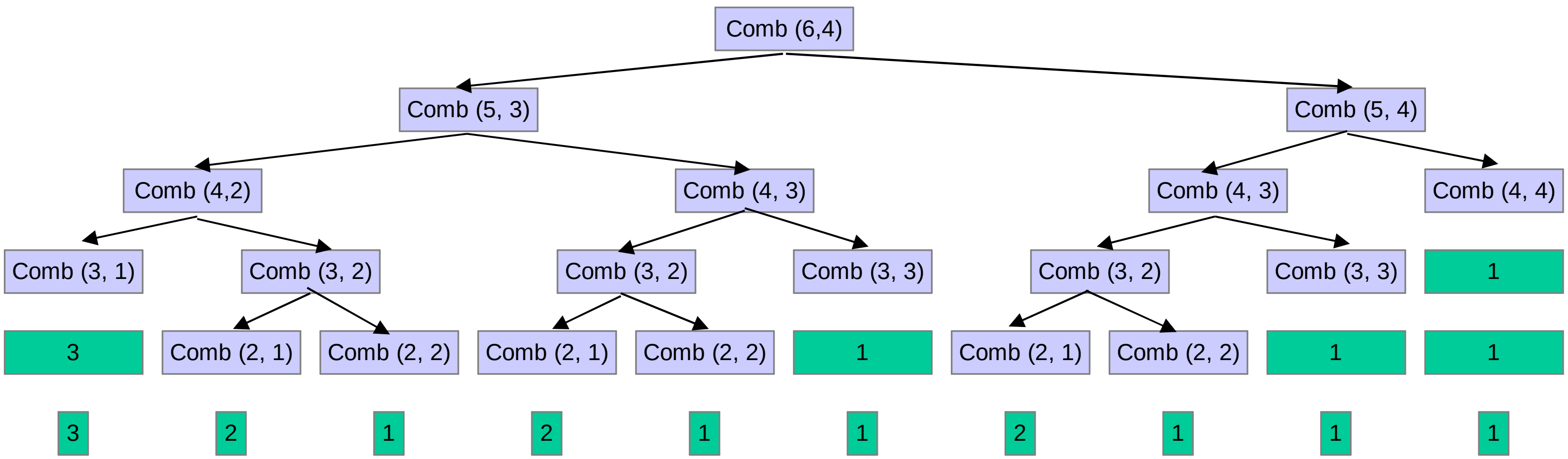
*E.g.:* Combinations:

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

$$\binom{n}{1} = 1 \qquad \binom{n}{n} = 1$$

  - A divide and conquer approach would repeatedly solve the common subproblems
  - Dynamic programming solves every subproblem just once and stores the answer in a table

# Example: Combinations




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

# Dynamic Programming

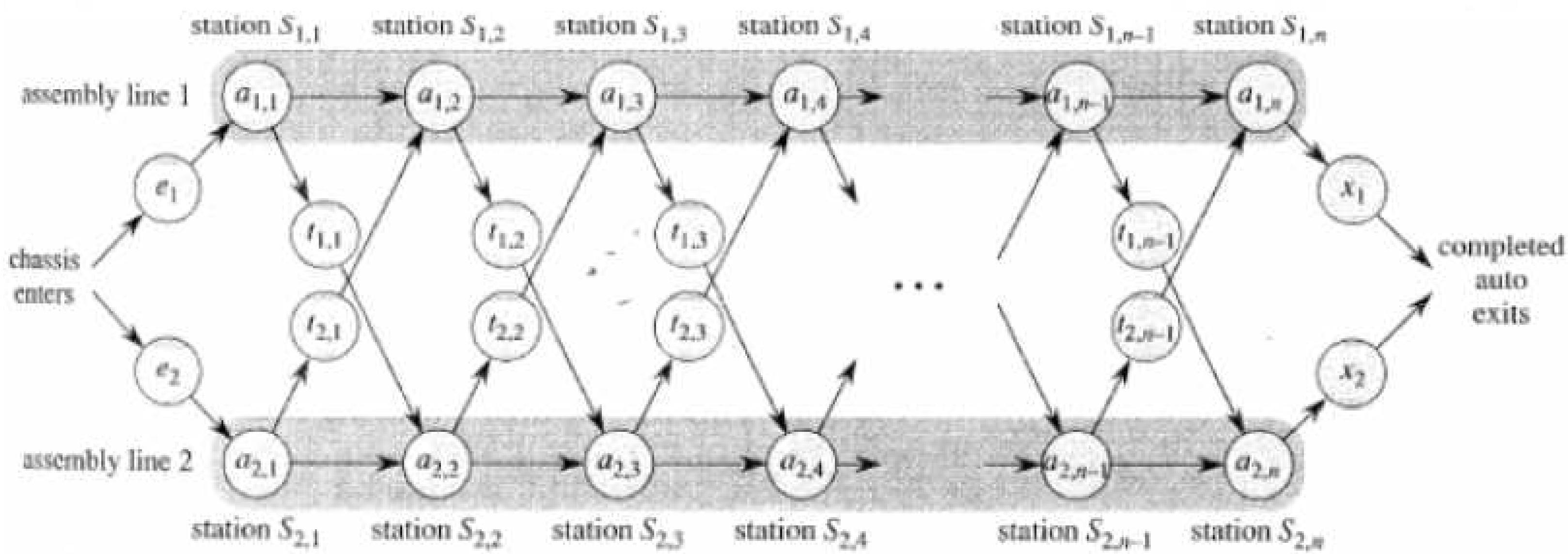
- Used for **optimization problems**
  - A set of choices must be made to get an optimal solution
  - Find a solution with the optimal value (minimum or maximum)
  - There may be many solutions that lead to an optimal value
  - Our goal: **find an optimal solution**

# Dynamic Programming Algorithm

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 (not always necessary)

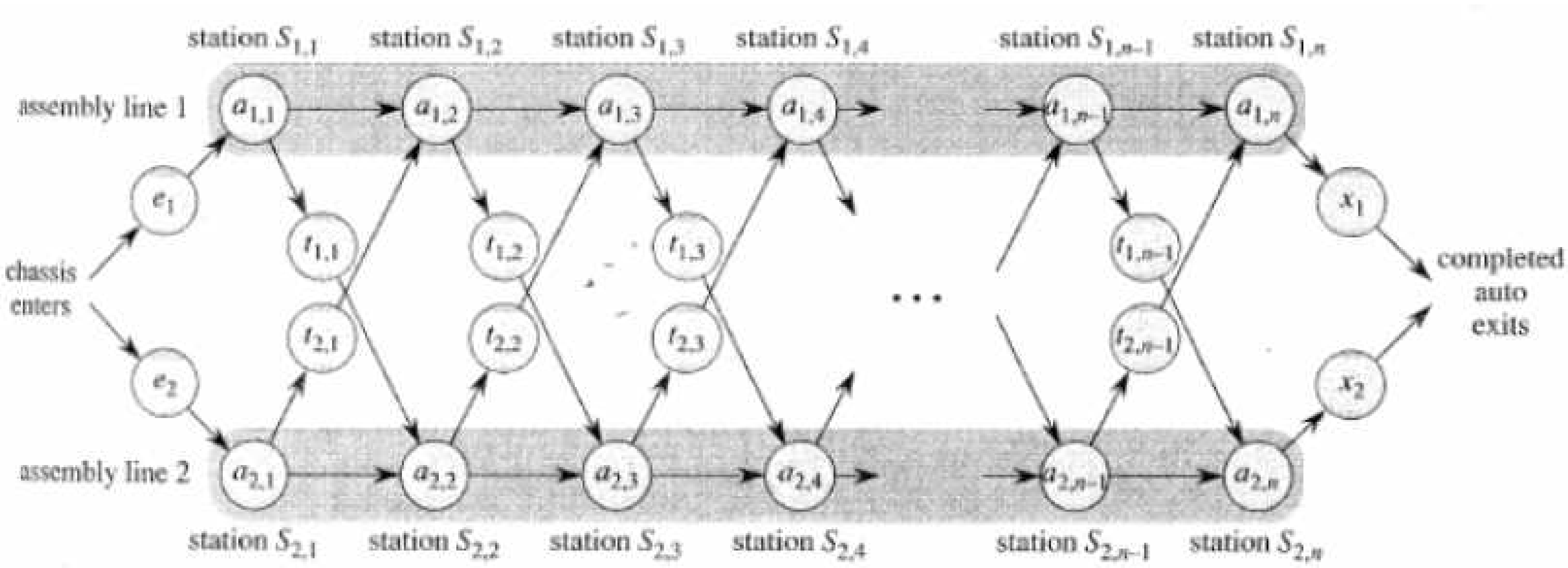
# Assembly Line Scheduling

- Automobile factory with two assembly lines
  - Each line has $n$ stations: $S_{1,1}, \ldots, S_{1,n}$ and $S_{2,1}, \ldots, S_{2,n}$
  - Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amounts of time $a_{1,j}$ and $a_{2,j}$
  - Entry times are: $e_1$ and $e_2$; exit times are: $x_1$ and $x_2$



# Assembly Line Scheduling

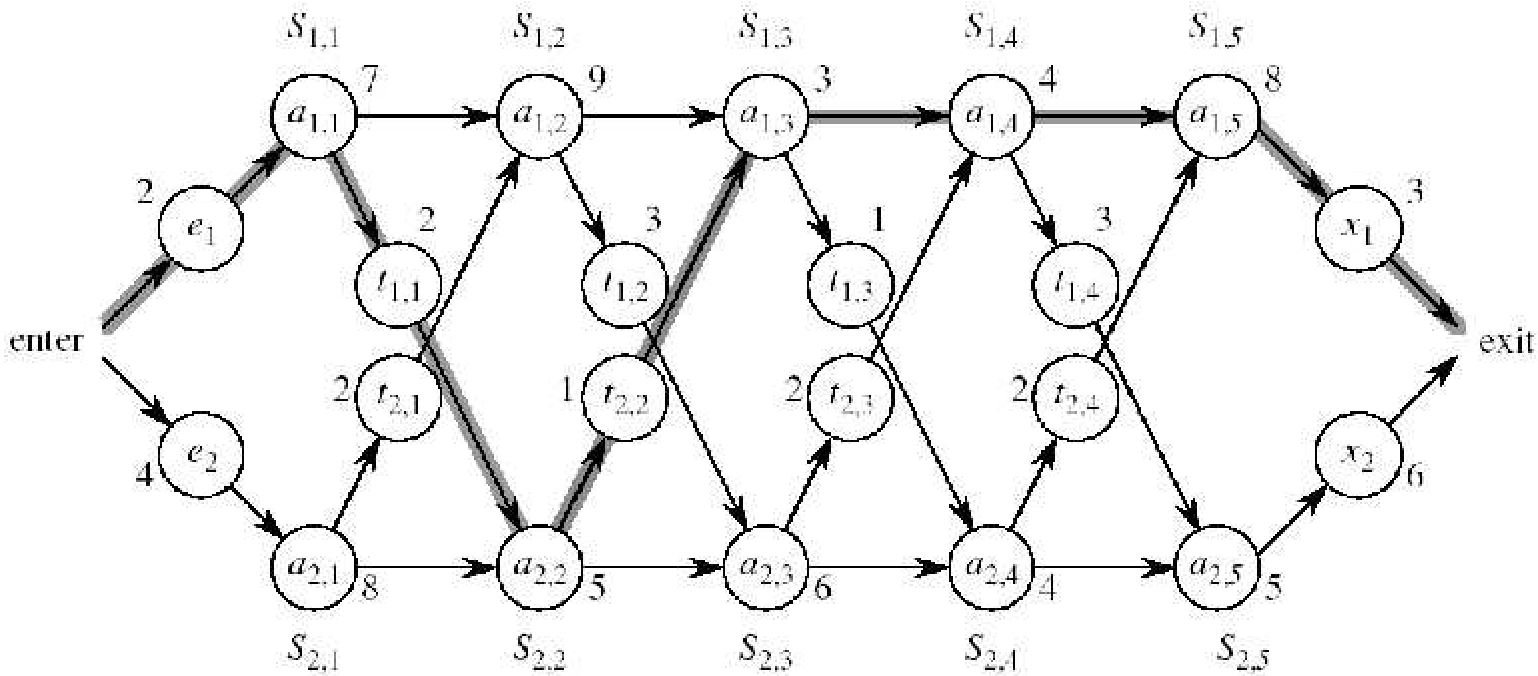
- After going through a station, can either:
  - stay on same line at no cost, or
  - transfer to other line: cost after $S_{i,j}$ is $t_{i,j}$ , $j = 1, \ldots, n - 1$



# Assembly Line Scheduling

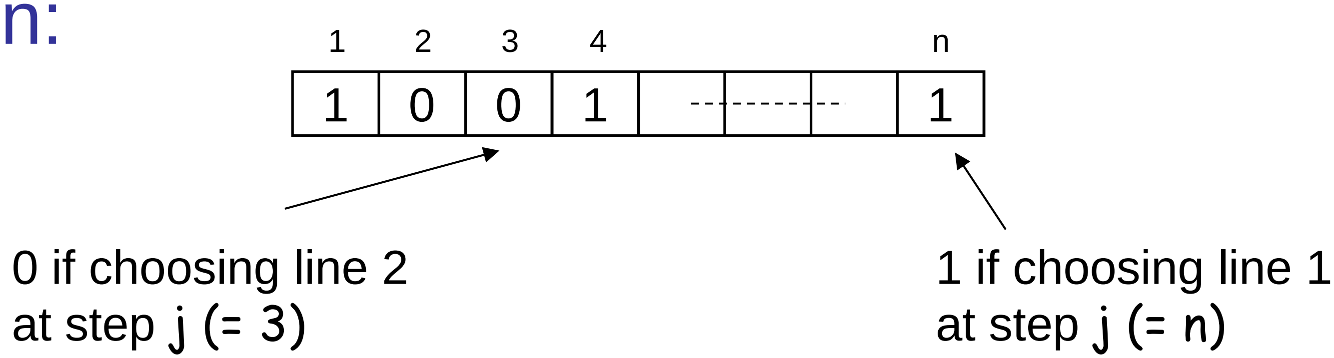
- Problem:

  what stations should be chosen from line 1 and which from line 2 in order to minimize the total time through the factory for one car?



# One Solution

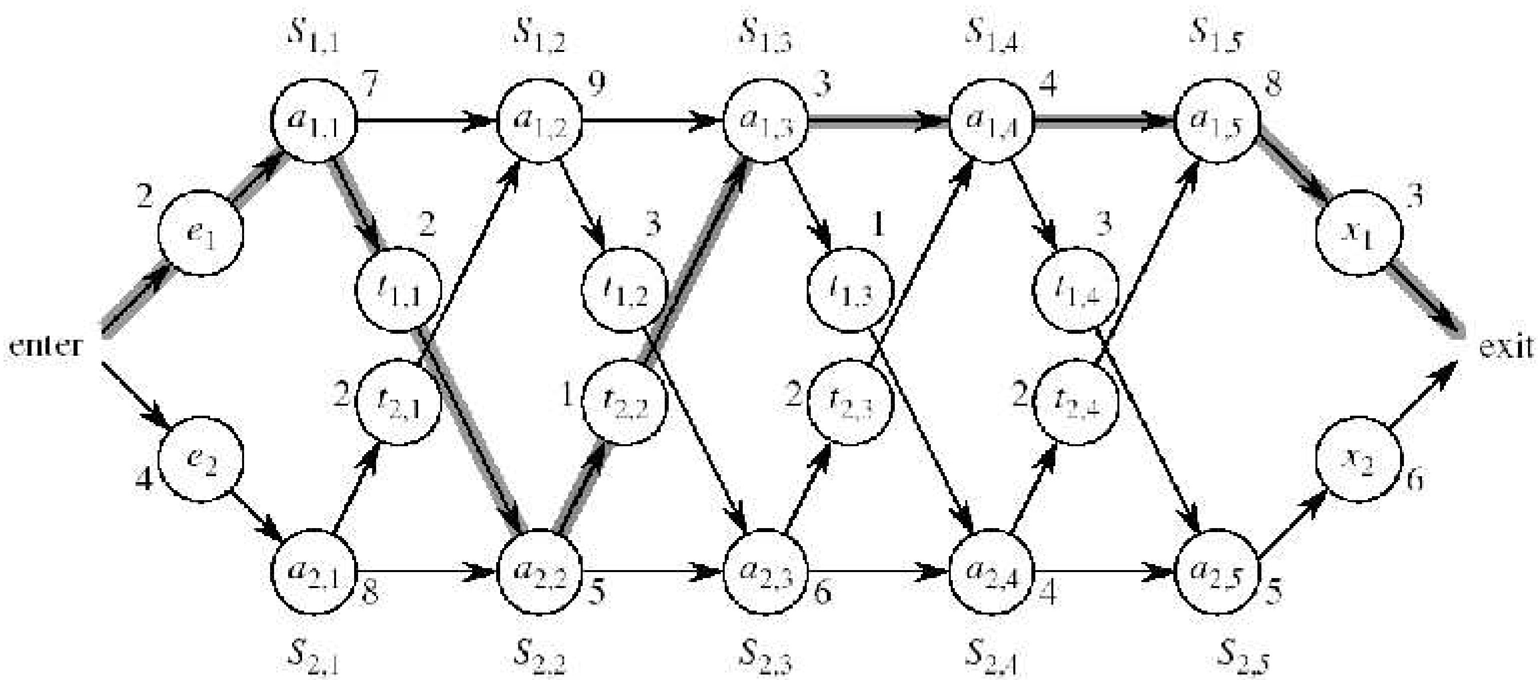
- Brute force
  - Enumerate all possibilities of selecting stations
  - Compute how long it takes in each case and choose the best one
- Solution:




  - There are $2^n$ possible ways to choose stations
  - Infeasible when $n$ is large!!

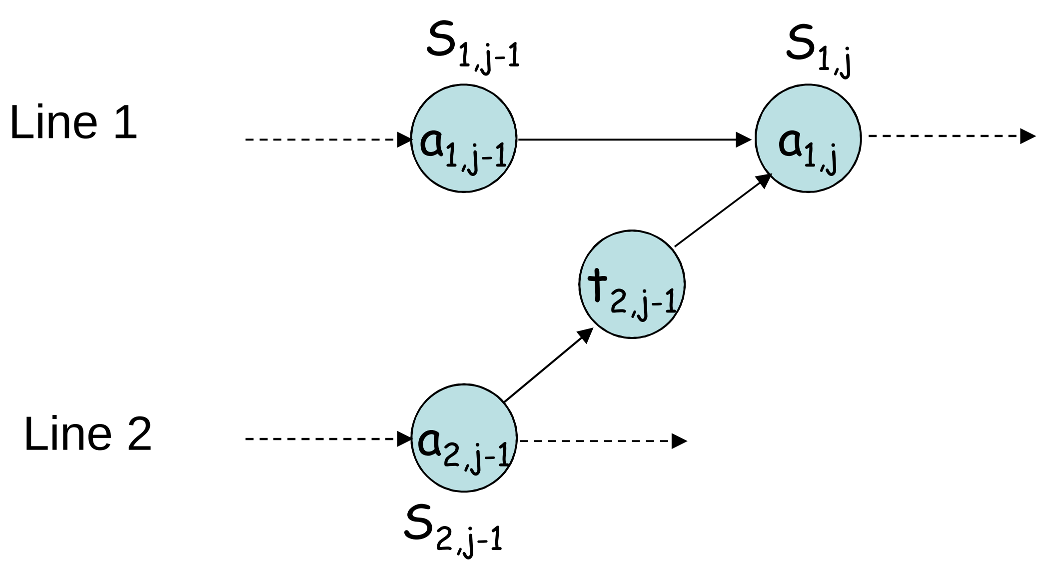
# 1. Structure of the Optimal Solution

- How do we compute the minimum time of going through a station?



# 1. Structure of the Optimal Solution

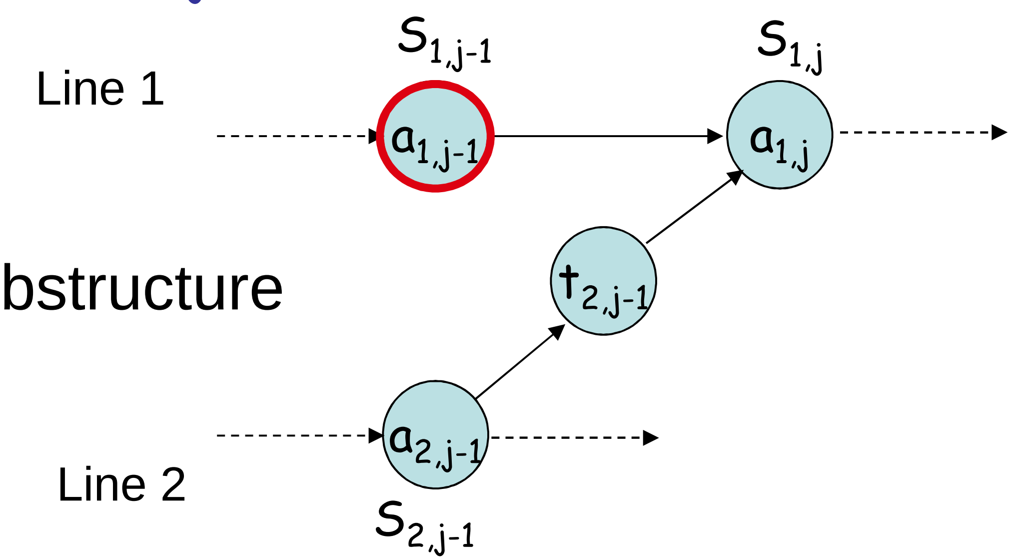
- Let's consider all possible ways to get from the starting point through station $S_{1,j}$
  - We have two choices of how to get to $S_{1,j}$:
    - Through $S_{1,j-1}$, then directly to $S_{1,j}$
    - Through $S_{2,j-1}$, then transfer over to $S_{1,j}$



# 1. Structure of the Optimal Solution

- Suppose that the fastest way through $S_{1,\,j}$ is through $S_{1,\,j-1}$
  - We must have taken a fastest way from entry through $S_{1,\,j-1}$
  - If there were a faster way through $S_{1,\,j-1}$, we would use it instead
- Similarly for $S_{2,\,j-1}$

Optimal Substructure



# Optimal Substructure

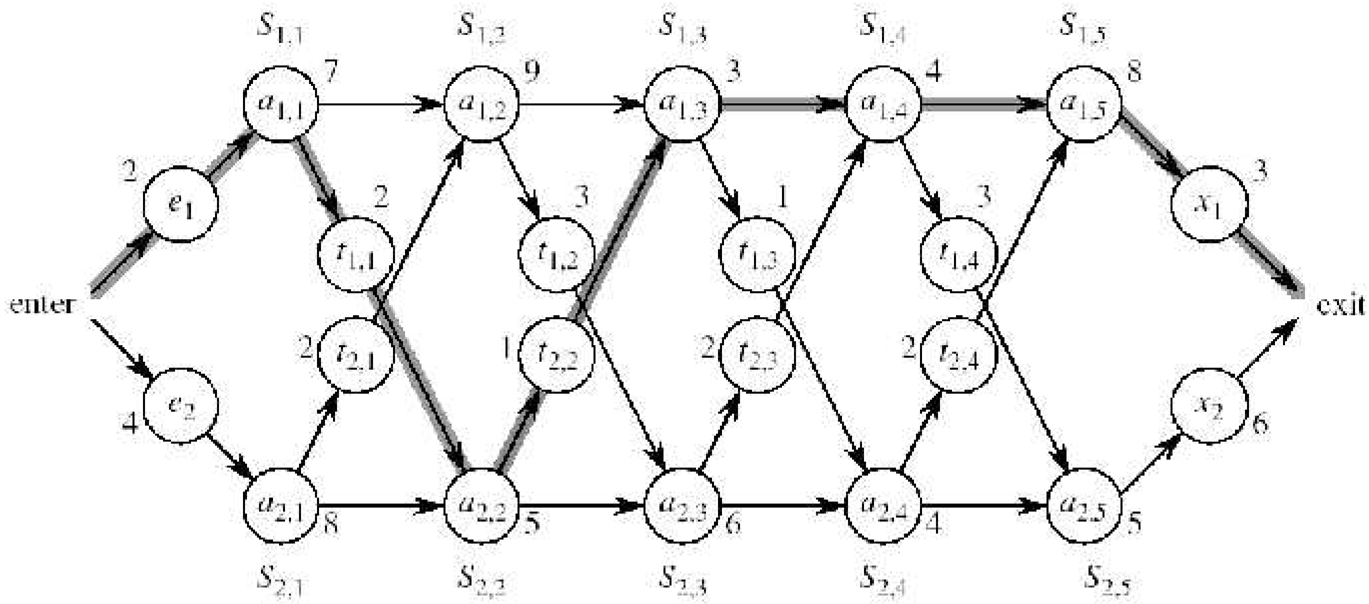
- **Generalization**: an optimal solution to the problem "*find the fastest way through* $S_{1,j}$" contains within it an optimal solution to subproblems: "*find the fastest way through* $S_{1,j-1}$ *or* $S_{2,j-1}$".

- This is referred to as the **optimal substructure** property

- We use this property to construct an optimal solution to a problem from optimal solutions to subproblems

# 2. A Recursive Solution

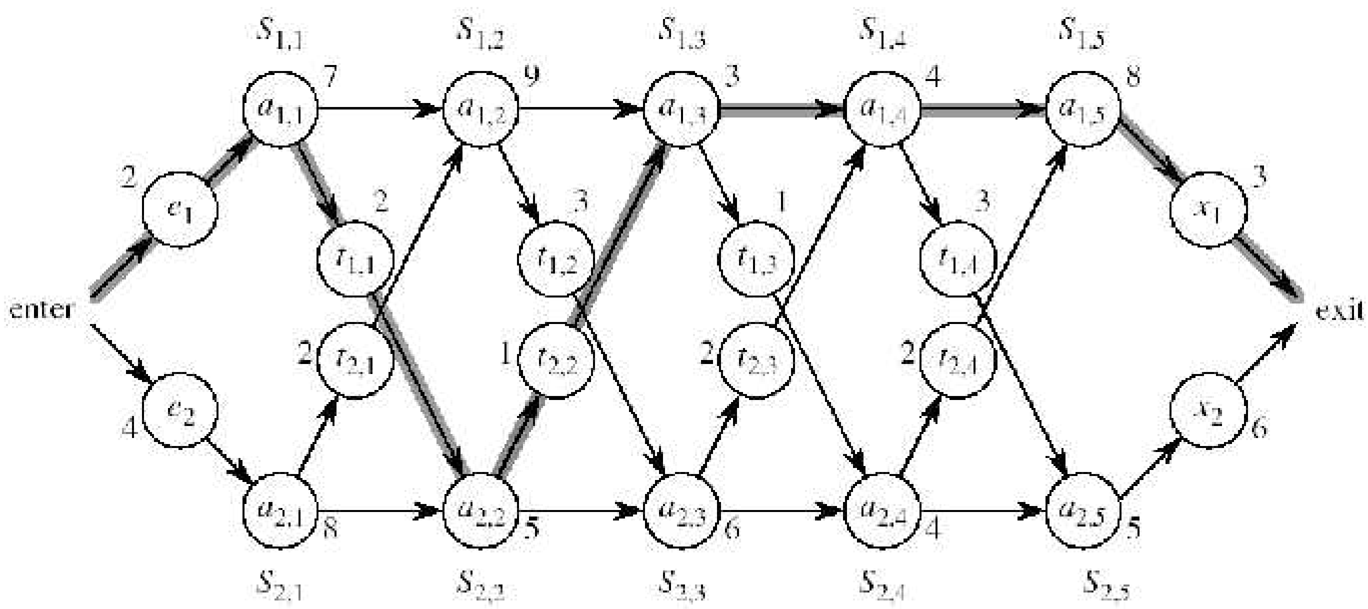
- Define the value of an optimal solution in terms of the optimal solution to subproblems



# 2. A Recursive Solution (cont.)

- Definitions:
  - f* : the fastest time to get through the entire factory
  - $f_i[j]$ : the fastest time to get from the starting point through station $S_{i,j}$

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

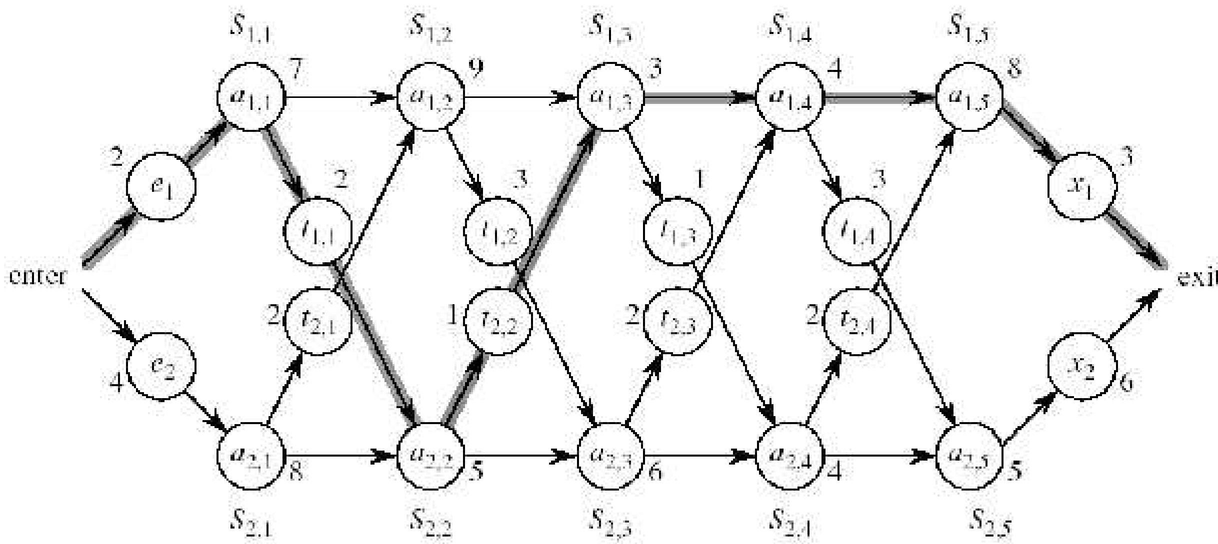


# 2. A Recursive Solution (cont.)

- Base case: $j = 1$, $i=1,2$ (getting through station 1)

$$f_1[1] = e_1 + a_{1,1}$$

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

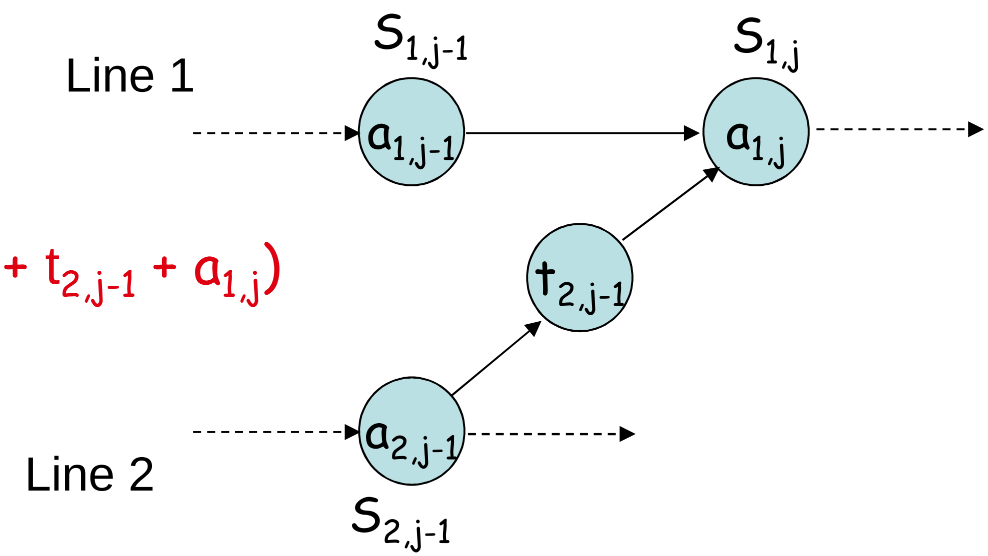


# 2. A Recursive Solution (cont.)

- General Case: $j = 2, 3, \ldots, n$, and $i = 1, 2$
- Fastest way through $S_{1,j}$ is either:
  - the way through $S_{1,j-1}$ then directly through $S_{1,j}$, or

    $f_1[j-1] + a_{1,j}$
  - the way through $S_{2,j-1}$, transfer from line 2 to line 1, then through $S_{1,j}$

    $f_2[j-1] + t_{2,j-1} + a_{1,j}$




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

# 2. A Recursive Solution (cont.)

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

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

# 3. Computing the Optimal Solution

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

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

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

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $f_1[j]$ | $f_1(1)$ | $f_1(2)$ | $f_1(3)$ | $f_1(4)$ | $f_1(5)$ |
| $f_2[j]$ | $f_2(1)$ | $f_2(2)$ | $f_2(3)$ | $f_2(4)$ | $f_2(5)$ |
| | | | 4 times | 2 times | |

- Solving top-down would result in exponential running time

# 3. Computing the Optimal Solution

- For $j \geq 2$, each value $f_i[j]$ depends only on the values of $f_1[j-1]$ and $f_2[j-1]$
- Idea: compute the values of $f_i[j]$ as follows:

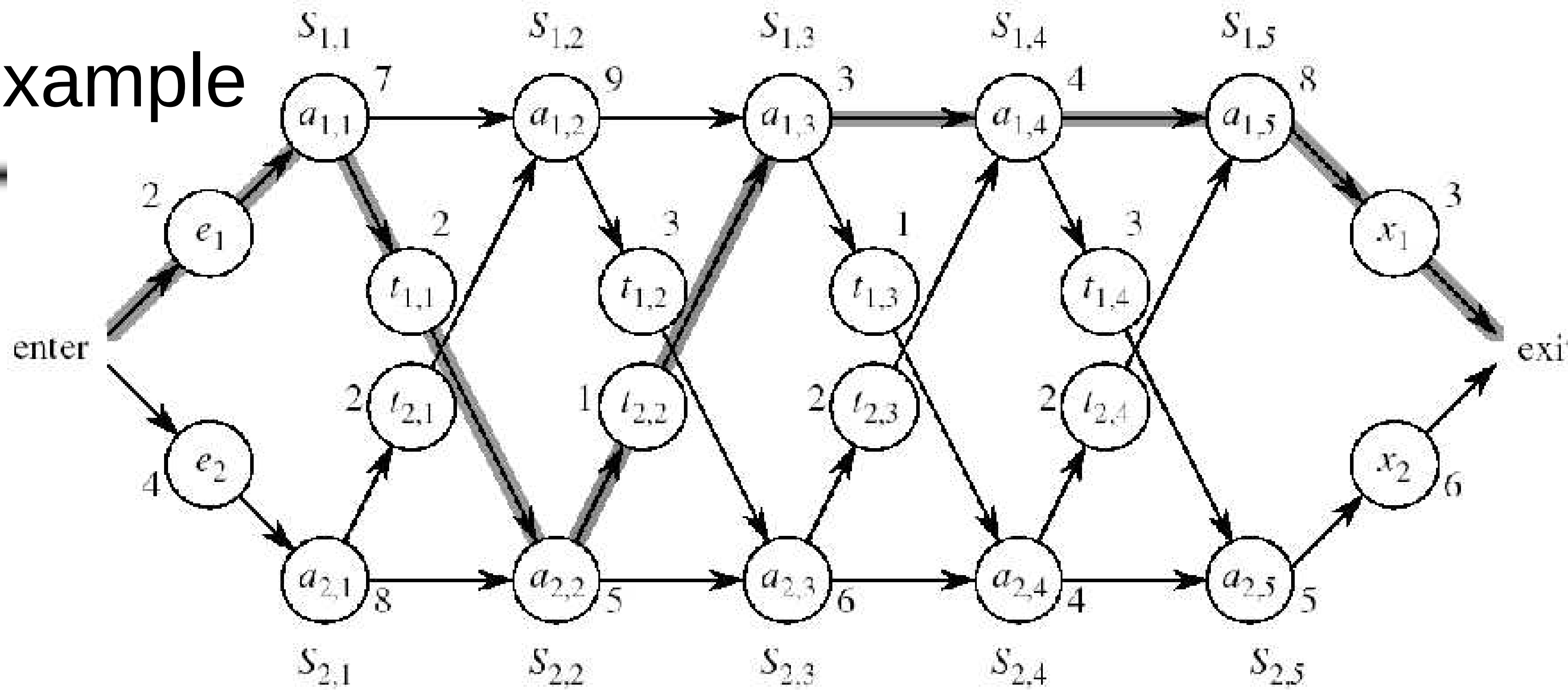
in increasing order of $j$

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $f_1[j]$ | | | | | |
| $f_2[j]$ | | | | | |

- Bottom-up approach
  - First find optimal solutions to subproblems
  - Find an optimal solution to the problem from the subproblems

# Example

-




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

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

f* = 35[1]

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

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

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

Compute initial values of $f_1$ and $f_2$

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

**O(N)**

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

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

            $l_1[j] \leftarrow 1$

        **else** $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$

            $l_1[j] \leftarrow 2$

Compute the values of $f_1[j]$ and $l_1[j]$

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

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

            $l_2[j] \leftarrow 2$

        **else** $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$

            $l_2[j] \leftarrow 1$

Compute the values of $f_2[j]$ and $l_2[j]$

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

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

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

    $l^* = 1$

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

    $l^* = 2$

Compute the values of the fastest time through the entire factory

# 4. Construct an Optimal Solution

*Alg.:* PRINT-STATIONS(l, n)

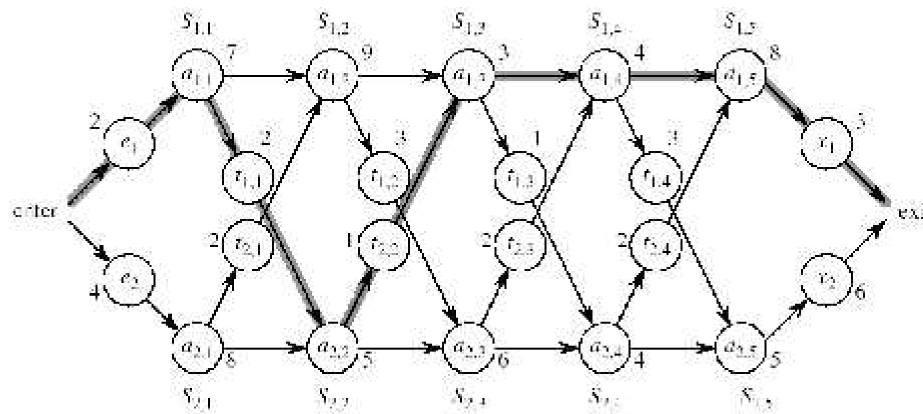
$i \leftarrow l^*$

print "line " i ", station " n

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

  **do** $i \leftarrow l_i[j]$

  print "line " i ", station " j - 1




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $f_1[j]/l_1[j]$ | 9 | 18[1] | 20[2] | 24[1] | 32[1] |
| $f_2[j]/l_2[j]$ | 12 | 16[1] | 22[2] | 25[1] | 30[2] |

$l^* = 1$

# Matrix-Chain Multiplication

**Problem**: given a sequence $\langle A_1, A_2, \ldots, A_n \rangle$, compute the product:

$$A_1 \cdot A_2 \cdots A_n$$

- Matrix compatibility:

| | |
|---|---|
| $C = A \cdot B$ | $C=A_1 \cdot A_2 \cdots A_i \cdot A_{i+1} \cdots A_n$ |
| $col_A = row_B$ | $col_i = row_{i+1}$ |
| $row_C = row_A$ | $row_C = row_{A1}$ |
| $col_C = col_B$ | $col_C = col_{An}$ |

# MATRIX-MULTIPLY(A, B)

**if** columns[A] ≠ rows[B]

  **then error** "incompatible dimensions"

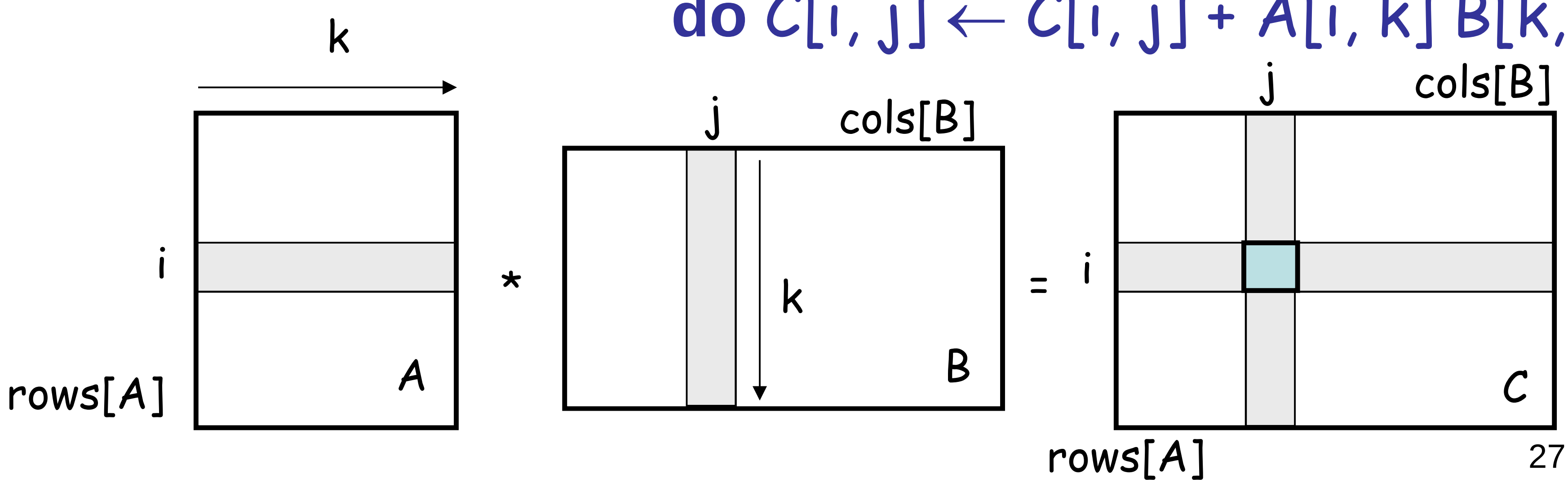
  **else for** i ← 1 to rows[A]

    **do for** j ← 1 to columns[B]

      **do** C[i, j] = 0

        **for** k ← 1 to columns[A]

          **do** $C[i, j] \leftarrow C[i, j] + \overbrace{A[i, k]\, B[k, j]}^{\text{rows}[A] \cdot \text{cols}[A] \cdot \text{cols}[B] \text{ multiplications}}$



# Matrix-Chain Multiplication

- In what order should we multiply the matrices?

$$A_1 \cdot A_2 \cdots A_n$$

- Parenthesize the product to get the order in which matrices are multiplied
- *E.g.:* $A_1 \cdot A_2 \cdot A_3 = ((A_1 \cdot A_2) \cdot A_3)$

$$= (A_1 \cdot (A_2 \cdot A_3))$$

- Which one of these orderings should we choose?
  - The order in which we multiply the matrices has a significant impact on the cost of evaluating the product

# Example

$$A_1 \cdot A_2 \cdot A_3$$

- $A_1$: 10 x 100
- $A_2$: 100 x 5
- $A_3$: 5 x 50

1. $((A_1 \cdot A_2) \cdot A_3)$: $A_1 \cdot A_2$ = 10 x 100 x 5 = 5,000 (10 x 5)

   $((A_1 \cdot A_2) \cdot A_3)$ = 10 x 5 x 50 = 2,500

   Total: 7,500 scalar multiplications

2. $(A_1 \cdot (A_2 \cdot A_3))$: $A_2 \cdot A_3$ = 100 x 5 x 50 = 25,000 (100 x 50)

   $(A_1 \cdot (A_2 \cdot A_3))$ = 10 x 100 x 50 = 50,000

   Total: 75,000 scalar multiplications

one order of magnitude difference!!

# Matrix-Chain Multiplication: Problem Statement

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

$$\begin{array}{ccccccccc} A_1 & \cdot & A_2 & \cdots & A_i & \cdot & A_{i+1} & \cdots & A_n \\ p_0 \times p_1 & & p_1 \times p_2 & & p_{i-1} \times p_i & & p_i \times p_{i+1} & & p_{n-1} \times p_n \end{array}$$

# What is the number of possible parenthesizations?

- Exhaustively checking all possible parenthesizations is not efficient!
- It can be shown that the number of parenthesizations grows as $\Omega(4^n/n^{3/2})$

  (see page 333 in your textbook)

# 1. The Structure of an Optimal Parenthesization

- Notation:

  $$A_{i\dots j} = A_i\ A_{i+1} \cdots A_j,\ i \le j$$

- Suppose that an optimal parenthesization of $A_{i\dots j}$ splits the product between $A_k$ and $A_{k+1}$, where $i \le k < j$

$$\begin{aligned} A_{i\dots j} &= A_i\ A_{i+1} \cdots A_j \\ &= A_i\ A_{i+1} \cdots A_k\ A_{k+1} \cdots A_j \\ &= A_{i\dots k}\ A_{k+1\dots j} \end{aligned}$$

# Optimal Substructure

$$A_{i\ldots j} = A_{i\ldots k}\, A_{k+1\ldots j}$$

- The parenthesization of the “prefix” $A_{i\ldots k}$ must be an optimal parentesization
- If there were a less costly way to parenthesize $A_{i\ldots k}$, we could substitute that one in the parenthesization of $A_{i\ldots j}$ and produce a parenthesization with a lower cost than the optimum $\Rightarrow$ contradiction!
- *An optimal solution to an instance of the matrix-chain multiplication contains within it optimal solutions to subproblems*

# 2. A Recursive Solution

- Subproblem:

  determine the minimum cost of parenthesizing

  $A_{i...j} = A_i A_{i+1} \cdots A_j$ for $1 \leq i \leq j \leq n$

- Let $m[i, j]$ = the minimum number of multiplications needed to compute $A_{i...j}$
  - full problem ($A_{1..n}$): $m[1, n]$
  - $i = j$: $A_{i...i} = A_i \Rightarrow m[i, i] =$ 0, for i = 1, 2, …, n

# 2. A Recursive Solution

- Consider the subproblem of parenthesizing

$$A_{i\ldots j} = A_i\, A_{i+1} \cdots A_j \qquad \text{for } 1 \le i \le j \le n$$

$$= \underbrace{A_{i\ldots k}}_{m[i,\,k]}\ \underbrace{A_{k+1\ldots j}}_{m[k+1,j]} \quad p_{i-1}p_kp_j \qquad \text{for } i \le k < j$$

- Assume that the optimal parenthesization splits the product $A_i\, A_{i+1} \cdots A_j$ at k ($i \le k < j$)

$$m[i, j] = \underbrace{m[i, k]}_{\text{min \# of multiplications to compute } A_{i\ldots k}} + \underbrace{m[k+1, j]}_{\text{min \# of multiplications to compute } A_{k+1\ldots j}} + \underbrace{p_{i-1}p_kp_j}_{\text{\# of multiplications to compute } A_{i\ldots k}A_{k\ldots j}}$$

# 2. A Recursive Solution (cont.)

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$$

- We do not know the value of k
  - There are j – i possible values for k: k = i, i+1, …, j-1
- Minimizing the cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

# 3. Computing the Optimal Costs

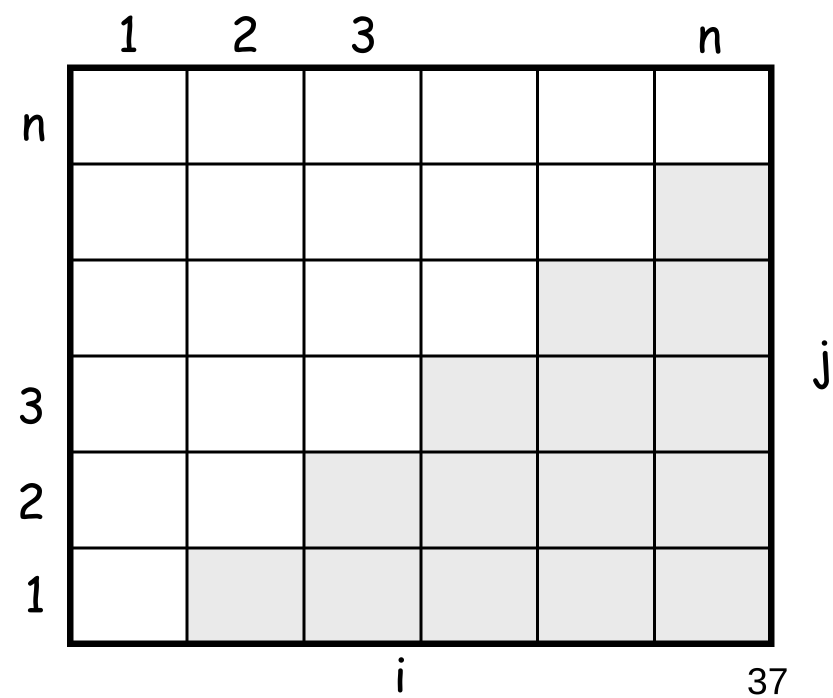
$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

- Computing the optimal solution recursively takes exponential time!
- How many subproblems?

  $\Rightarrow \Theta(n^2)$

  – Parenthesize $A_{i\ldots j}$ for $1 \le i \le j \le n$

  – One problem for each choice of i and j



# 3. Computing the Optimal Costs (cont.)

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

- How do we fill in the tables m[1..n, 1..n]?
  - Determine which entries of the table are used in computing $m[i, j]$

$$A_{i\ldots j} = A_{i\ldots k} A_{k+1\ldots j}$$

  - Subproblems' size is one less than the original size
  - **Idea:** fill in $m$ such that it corresponds to solving problems of increasing length

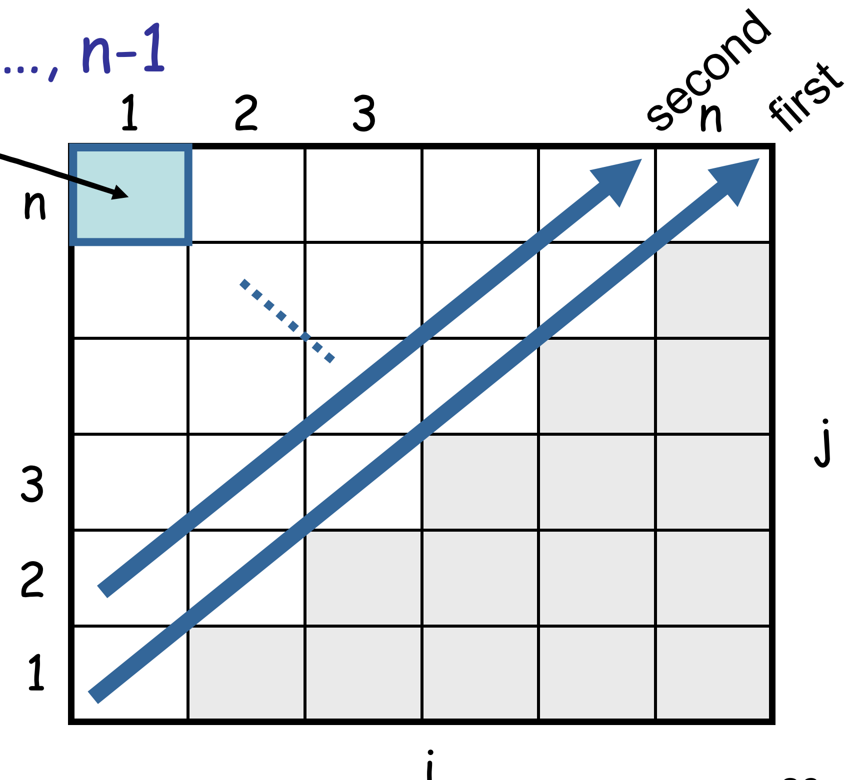
# 3. Computing the Optimal Costs (cont.)

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

- Length = 1: $i = j$, $i = 1, 2, \ldots, n$
- Length = 2: $j = i + 1$, $i = 1, 2, \ldots, n-1$

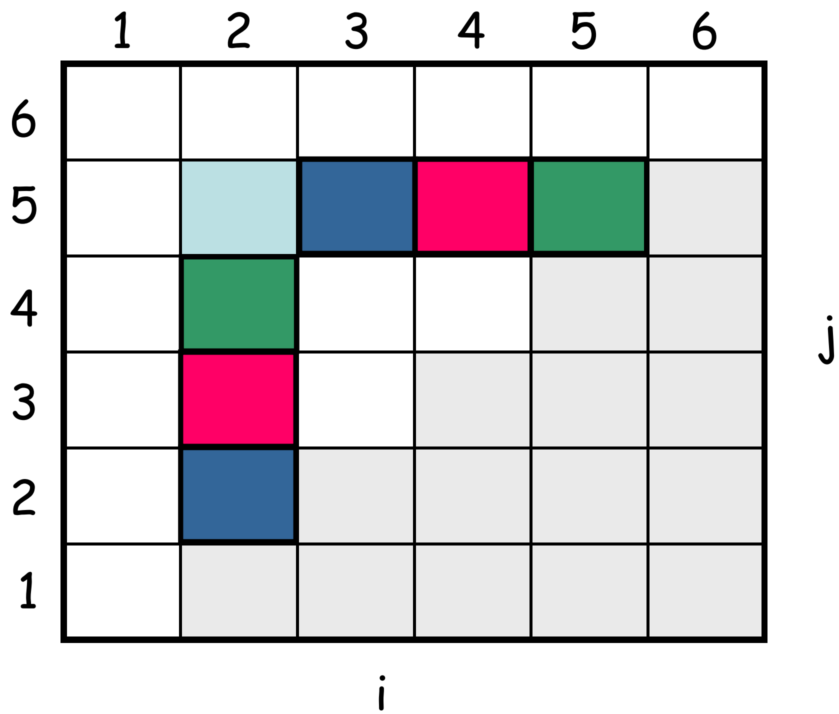
$m[1, n]$ gives the optimal solution to the problem

Compute rows from bottom to top and from left to right



# Example: $\min \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\}$

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1p_2p_5 & k = 2 \\ m[2, 3] + m[4, 5] + p_1p_3p_5 & k = 3 \\ m[2, 4] + m[5, 5] + p_1p_4p_5 & k = 4 \end{cases}$$




- Values m[i, j] depend only on values that have been previously computed

# Example $\min\{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\}$

Compute $A_1 \cdot A_2 \cdot A_3$

- $A_1$: 10 x 100 ($p_0 \times p_1$)
- $A_2$: 100 x 5 ($p_1 \times p_2$)
- $A_3$: 5 x 50 ($p_2 \times p_3$)

| | 1 | 2 | 3 |
|---|---|---|---|
| 3 | 2<br>7500 | 2<br>25000 | 0 |
| 2 | 1<br>5000 | 0 | |
| 1 | 0 | | |

$m[i, i] = 0$ for $i = 1, 2, 3$

$m[1, 2] = m[1, 1] + m[2, 2] + p_0p_1p_2$ $(A_1A_2)$

$= 0 + 0 + 10 * 100 * 5 = 5{,}000$

$m[2, 3] = m[2, 2] + m[3, 3] + p_1p_2p_3$ $(A_2A_3)$

$= 0 + 0 + 100 * 5 * 50 = 25{,}000$

$$m[1, 3] = \min \begin{cases} m[1, 1] + m[2, 3] + p_0p_1p_3 = 75{,}000 & (A_1(A_2A_3)) \\ m[1, 2] + m[3, 3] + p_0p_2p_3 = 7{,}500 & ((A_1A_2)A_3) \end{cases}$$

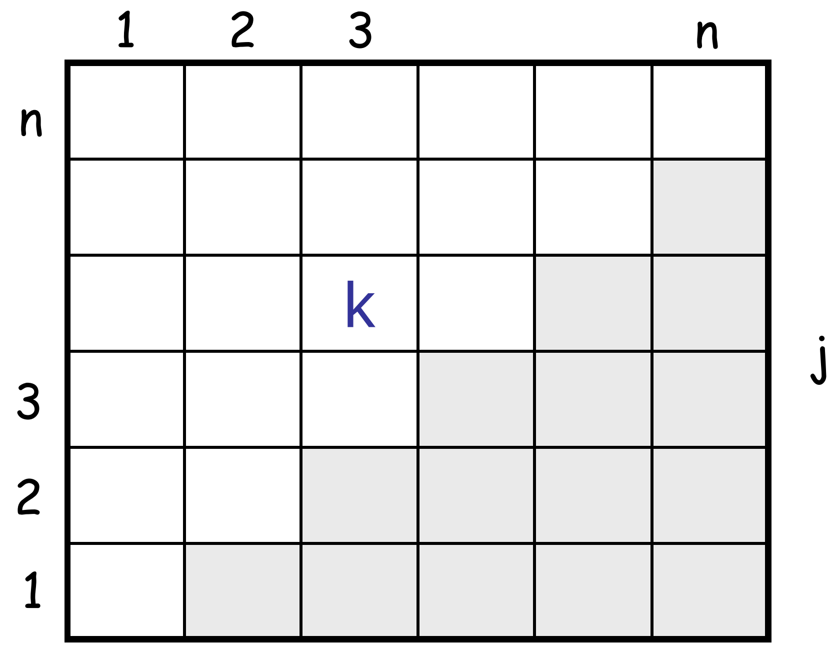
# Matrix-Chain-Order(p)

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
     do m[i, i] ← 0
for l ← 2 to n          ▷ l is the chain length.
     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
```

$O(N^3)$

# 4. Construct the Optimal Solution

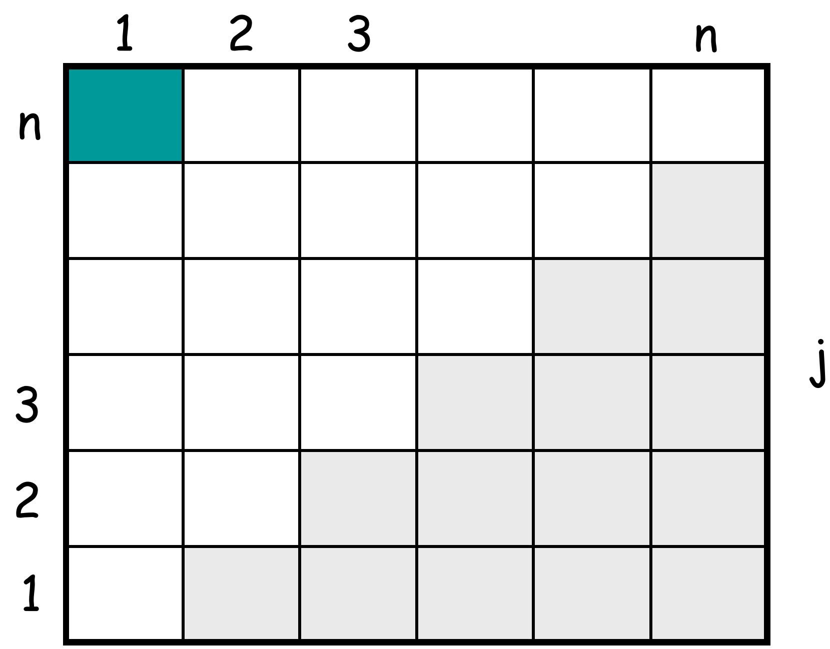
- In a similar matrix s we keep the optimal values of k
- $s[i, j]$ = a value of $k$ such that an optimal parenthesization of $A_{i..j}$ splits the product between $A_k$ and $A_{k+1}$



# 4. Construct the Optimal Solution

- $s[1, n]$ is associated with the entire product $A_{1..n}$
  - The final matrix multiplication will be split at k = $s[1, n]$

    $A_{1..n} = A_{1..s[1, n]} \cdot A_{s[1, n]+1..n}$
  - For each subproduct recursively find the corresponding value of k that results in an optimal parenthesization



# 4. Construct the Optimal Solution

- $s[i, j]$ = value of $k$ such that the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between $A_k$ and $A_{k+1}$

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 6 | 3 | 3 | 3 | 5 | 5 | - |
| 5 | 3 | 3 | 3 | 4 | - | |
| 4 | 3 | 3 | 3 | - | | |
| 3 | 1 | 2 | - | | | |
| 2 | 1 | - | | | | |
| 1 | - | | | | | |

j

i

- $s[1, n] = 3 \Rightarrow A_{1..6} = A_{1..3}\, A_{4..6}$
- $s[1, 3] = 1 \Rightarrow A_{1..3} = A_{1..1}\, A_{2..3}$
- $s[4, 6] = 5 \Rightarrow A_{4..6} = A_{4..5}\, A_{6..6}$

# 4. Construct the Optimal Solution (cont.)

PRINT-OPT-PARENS($s$, $i$, $j$)

**if** $i = j$

  **then** print "A"$_i$

  **else** print "("

    PRINT-OPT-PARENS($s$, $i$, $s[i, j]$)

    PRINT-OPT-PARENS($s$, $s[i, j] + 1$, $j$)

    print ")"

| j \ i | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 6 | 3 | 3 | 3 | 5 | 5 | - |
| 5 | 3 | 3 | 3 | 4 | - | |
| 4 | 3 | 3 | 3 | - | | |
| 3 | 1 | 2 | - | | | |
| 2 | 1 | - | | | | |
| 1 | - | | | | | |

# Example: $A_1 \cdot \cdot \cdot A_6$ $((A_1(A_2A_3))((A_4A_5)A_6))$

PRINT-OPT-PARENS(*s*, i, j)

**if** i = j

**then** print "A"$_i$

**else** print "("

PRINT-OPT-PARENS(*s*, i, *s*[i, j])

PRINT-OPT-PARENS(*s*, *s*[i, j] + 1, j)

print ")"

**s[1..6, 1..6]**

| j \ i | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 6 | 3 | 3 | 3 | 5 | 5 | - |
| 5 | 3 | 3 | 3 | 4 | - | |
| 4 | 3 | 3 | 3 | - | | |
| 3 | 1 | 2 | - | | | |
| 2 | 1 | - | | | | |
| 1 | - | | | | | |

P-O-P(s, 1, 6) s[1, 6] = 3

i = 1, j = 6 "(" P-O-P (s, 1, 3) s[1, 3] = 1

i = 1, j = 3 "(" P-O-P(s, 1, 1) $\Rightarrow$ "$A_1$"

P-O-P(s, 2, 3) s[2, 3] = 2

i = 2, j = 3 "(" P-O-P (s, 2, 2) $\Rightarrow$ "$A_2$"

P-O-P (s, 3, 3) $\Rightarrow$ "$A_3$"

")"

")" ...

# Memoization

- Top-down approach with the efficiency of typical dynamic programming approach
- Maintaining an entry in a table for the solution to each subproblem
  - **memoize** the inefficient recursive algorithm
- When a subproblem is first encountered its solution is computed and stored in that table
- Subsequent “calls” to the subproblem simply look up that value

# Memoized Matrix-Chain

*Alg.:* MEMOIZED-MATRIX-CHAIN(p)

1. $n \leftarrow length[p] - 1$
2. **for** $i \leftarrow 1$ **to** $n$
3. **do for** $j \leftarrow i$ **to** $n$
4. **do** $m[i, j] \leftarrow \infty$
5. **return** LOOKUP-CHAIN(p, 1, n)

Initialize the $m$ table with large values that indicate whether the values of $m[i, j]$ have been computed

Top-down approach

# Memoized Matrix-Chain

*Alg.:* LOOKUP-CHAIN(p, i, j)

Running time is $O(n^3)$

1. **if** $m[i, j] < \infty$
2. **then return** $m[i, j]$
3. **if** $i = j$
4. **then** $m[i, j] \leftarrow 0$
5. **else for** $k \leftarrow i$ **to** $j - 1$
6. **do** $q \leftarrow$ LOOKUP-CHAIN(p, i, k) + LOOKUP-CHAIN(p, k+1, j) + $p_{i-1}p_kp_j$
7. **if** $q < m[i, j]$
8. **then** $m[i, j] \leftarrow q$
9. **return** $m[i, j]$

# Dynamic Progamming vs. Memoization

- Advantages of dynamic programming vs. memoized algorithms
  - No overhead for recursion, less overhead for maintaining the table
  - The regular pattern of table accesses may be used to reduce time or space requirements
- Advantages of memoized algorithms vs. dynamic programming
  - Some subproblems do not need to be solved

# Matrix-Chain Multiplication - Summary

- Both the dynamic programming approach and the memoized algorithm can solve the matrix-chain multiplication problem in $O(n^3)$
- Both methods take advantage of the overlapping subproblems property
- There are only $\Theta(n^2)$ different subproblems
  - Solutions to these problems are computed only once
- Without memoization the natural recursive algorithm runs in exponential time

# Elements of Dynamic Programming

- Optimal Substructure
  - An optimal solution to a problem contains within it an optimal solution to subproblems
  - Optimal solution to the entire problem is build in a bottom-up manner from optimal solutions to subproblems
- Overlapping Subproblems
  - If a recursive algorithm revisits the same subproblems over and over $\Rightarrow$ the problem has overlapping subproblems

# Parameters of Optimal Substructure

- How many subproblems are used in an optimal solution for the original problem
  - Assembly line: One subproblem (the line that gives best time)
  - Matrix multiplication: Two subproblems (subproducts $A_{i..k}$, $A_{k+1..j}$)
- How many choices we have in determining which subproblems to use in an optimal solution
  - Assembly line: Two choices (line 1 or line 2)
  - Matrix multiplication: $j - i$ choices for $k$ (splitting the product)

# Parameters of Optimal Substructure

- Intuitively, the running time of a dynamic programming algorithm depends on two factors:
  - Number of subproblems overall
  - How many choices we look at for each subproblem
- Assembly line
  - $\Theta(n)$ subproblems (n stations)
  - 2 choices for each subproblem

  $\Theta(n)$ overall
- Matrix multiplication:
  - $\Theta(n^2)$ subproblems ($1 \le i \le j \le n$)
  - At most $n-1$ choices

  $\Theta(n^3)$ overall

# Longest Common Subsequence

- Given two sequences

  $X = \langle x_1, x_2, \ldots, x_m \rangle$

  $Y = \langle y_1, y_2, \ldots, y_n \rangle$

  find a maximum length common subsequence (LCS) of X and Y

- *E.g.:*

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

- Subsequences of X:
  - A subset of elements in the sequence taken in order

  $\langle A, B, D \rangle$, $\langle B, C, D, B \rangle$, etc.

# Example

X = ⟨A, B, C, B, D, A, B⟩

Y = ⟨B, D, C, A, B, A⟩

X = ⟨A, B, C, B, D, A, B⟩

Y = ⟨B, D, C, A, B, A⟩

- ⟨B, C, B, A⟩ and ⟨B, D, A, B⟩ are longest common subsequences of X and Y (length = 4)
- ⟨B, C, A⟩, however is not a LCS of X and Y

# Brute-Force Solution

- For every subsequence of X, check whether it's a subsequence of Y
- There are $2^m$ subsequences of X to check
- Each subsequence takes $\Theta(n)$ time to check
  - scan Y for first letter, from there scan for second, and so on
- Running time: $\Theta(n2^m)$

# Making the choice

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

$Y = \langle Z, B, E \rangle$

- Choice: include one element into the common sequence (E) and solve the resulting subproblem

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

$Y = \langle Z, B, D \rangle$

- Choice: exclude an element from a string and solve the resulting subproblem

# Notations

- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ we define the i-th prefix of X, for i = 0, 1, 2, …, m

  $$X_i = \langle x_1, x_2, \ldots, x_i \rangle$$

- $c[i, j]$ = the length of a LCS of the sequences $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$

# A Recursive Solution

Case 1: $x_i = y_j$

*e.g.:* $X_i = \langle A, B, D, E \rangle$

$Y_j = \langle Z, B, E \rangle$

$$c[i, j] = c[i - 1, j - 1] + 1$$

– Append $x_i = y_j$ to the LCS of $X_{i-1}$ and $Y_{j-1}$

– Must find a LCS of $X_{i-1}$ and $Y_{j-1}$ $\Rightarrow$ optimal solution to a problem includes optimal solutions to subproblems

# A Recursive Solution

Case 2: $x_i \neq y_j$

*e.g.:* $X_i = \langle A, B, D, G \rangle$

$Y_j = \langle Z, B, D \rangle$

$$c[i, j] = \max \{ c[i - 1, j], c[i, j-1] \}$$

– Must solve two problems

- find a LCS of $X_{i-1}$ and $Y_j$: $X_{i-1} = \langle A, B, D \rangle$ and $Y_j = \langle Z, B, D \rangle$
- find a LCS of $X_i$ and $Y_{j-1}$: $X_i = \langle A, B, D, G \rangle$ and $Y_j = \langle Z, B \rangle$

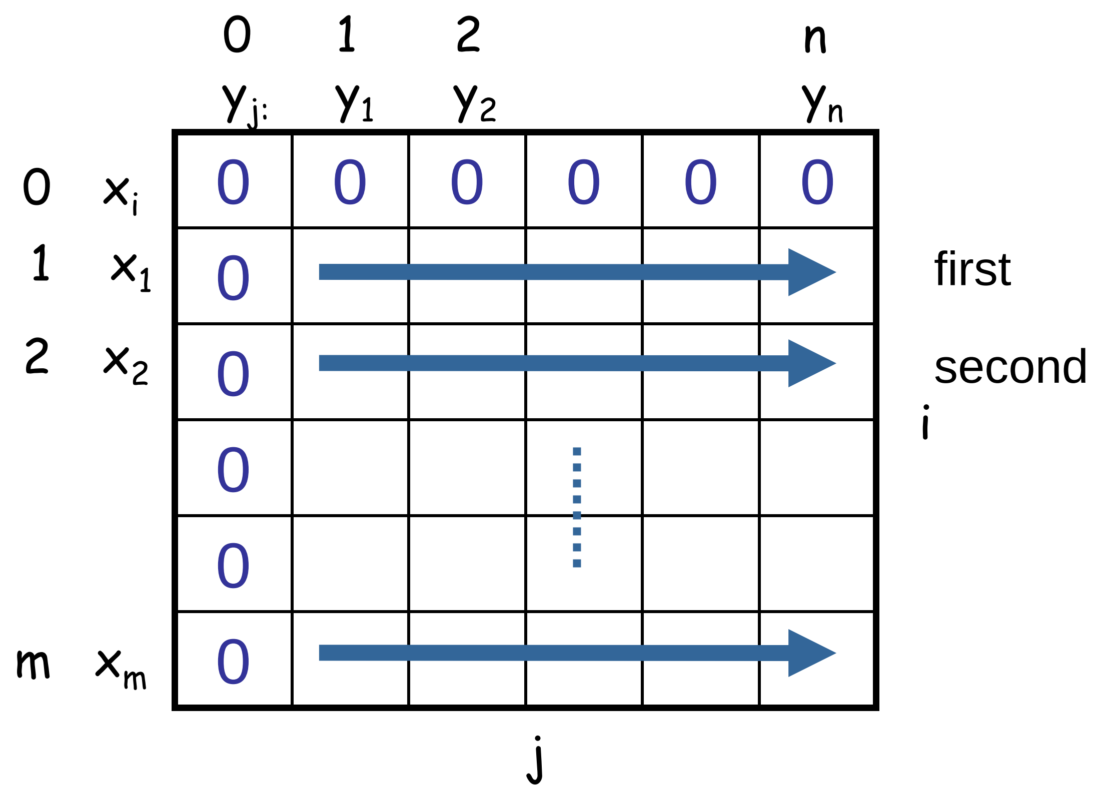
- Optimal solution to a problem includes optimal solutions to subproblems

# Overlapping Subproblems

- To find a LCS of X and Y
  - we may need to find the LCS between X and $Y_{n-1}$ and that of $X_{m-1}$ and Y
  - Both the above subproblems has the subproblem of finding the LCS of $X_{m-1}$ and $Y_{n-1}$
- Subproblems share subsubproblems

# 3. Computing the Length of the LCS

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



# Additional Information

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

b & c:

| | | 0 | 1 | 2 | 3 | | n |
|---|---|---|---|---|---|---|---|
| | | $y_j$: | A | C | D | | F |
| 0 | $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | | | | | |
| 2 | B | 0 | | | c[i-1,j] | | |
| 3 | C | 0 | | ↖ c[i,j-1] | ↑ | | |
| | | 0 | | | | | |
| m | D | 0 | | | | | |

i

j

A matrix b[i, j]:

- For a subproblem [i, j] it tells us what choice was made to obtain the optimal value
- If $x_i = y_j$
  
  b[i, j] = "↖"
- Else, if
  
  c[i - 1, j] ≥ c[i, j-1]
  
  b[i, j] = " ↑ "
  
  else
  
  b[i, j] = " ← "

# LCS-LENGTH(X, Y, m, n)

```
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i - 1, j - 1] + 1
                 b[i, j ] ← "↖"
            else if c[i - 1, j] ≥ c[i, j - 1]
                then c[i, j] ← c[i - 1, j]
                     b[i, j] ← "↑"
                else c[i, j] ← c[i, j - 1]
                     b[i, j] ← "←"
return c and b
```

The length of the LCS if one of the sequences is empty is zero

Case 1: $x_i = y_j$

Case 2: $x_i \neq y_j$

Running time: $\Theta(mn)$

# Example

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

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

If $x_i = y_j$

$b[i, j] =$ " ↖ "

Else if

$c[i - 1, j] \geq c[i, j-1]$

$b[i, j] =$ " ↑ "

else

$b[i, j] =$ " ← "

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| | | $y_j$ | 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 |

# 4. Constructing a LCS

- Start at b[m, n] and follow the arrows
- When we encounter a "↖" in b[i, j] ⇒ $x_i = y_j$ is an element of the LCS

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| | | $y_j$ | 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 |

# PRINT-LCS(b, X, i, j)

Running time: $\Theta(m + n)$

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

Initial call: PRINT-LCS(b, X, length[X], length[Y])

# Improving the Code

- What can we say about how each entry $c[i, j]$ is computed?
  - It depends only on $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$
  - Eliminate table $b$ and compute in $O(1)$ which of the three values was used to compute $c[i, j]$
  - We save $\Theta(mn)$ space from table $b$
  - However, we do not asymptotically decrease the auxiliary space requirements: still need table $c$

# Improving the Code

- If we only need the length of the LCS
  - LCS-LENGTH works only on two rows of c at a time
    - The row being computed and the previous row
  - We can reduce the asymptotic space requirements by storing only these two rows