

# CSC 611: Analysis of Algorithms

---

## Lecture 10

### Dynamic Programming

## Dynamic Programming

---

- An algorithm design technique for **optimization problems** (similar to 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

---

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

CSC611/ Lecture10

## Principal of Optimality

---

An optimal sequence of decisions has the property that whatever the initial state and decisions are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision

CSC611/ Lecture10

# Principal of Optimality

---

- Dynamic programming uses the principle of optimality in order to drastically reduce the amount of enumeration by
  - Avoiding the enumeration of some decision sequences that cannot possibly be optimal
- It is a bottom-up approach with the sub-solution stored in an array

CSC611/ Lecture10

## Dynamic Programming

---

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

*E.g.:* Fibonacci numbers:

- Recurrence:  $F(n) = F(n-1) + F(n-2)$
- Boundary conditions:  $F(1) = 0, F(2) = 1$
- Compute:  $F(5) = 3, F(3) = 1, F(4) = 2$
- 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

CSC611/ Lecture10

# Tackling Dynamic Programming ...

---

- Try to save results to avoid repeated computations
- Figure out the order in which the table entries must be computed
- Determine how you can find the solution to the problem from the data in the table

## 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

# Elements of Dynamic Programming

---

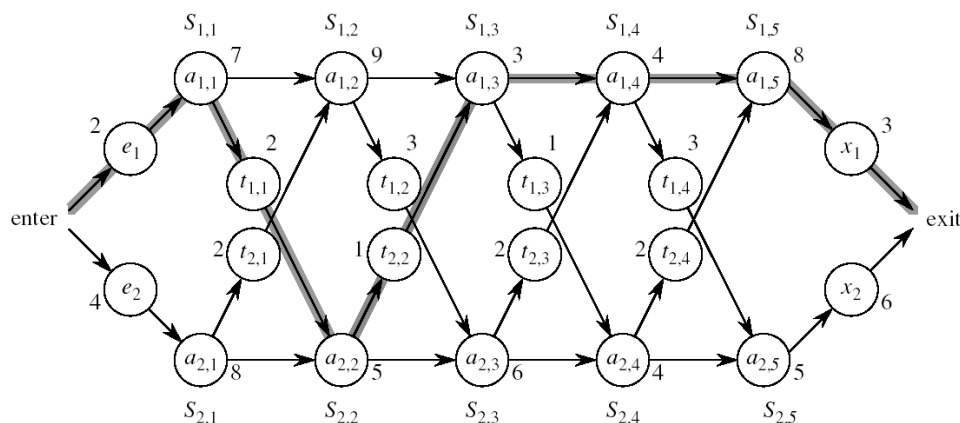
- Optimal Substructure
  - An optimal solution to a problem contains within it an optimal solution to subproblems
  - Optimal solution to the entire problem is built in a bottom-up manner from optimal solutions to subproblems
- Overlapping Subproblems
  - If a recursive algorithm revisits the same subproblems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

CSC611/ Lecture10

## Assembly Line Scheduling

---

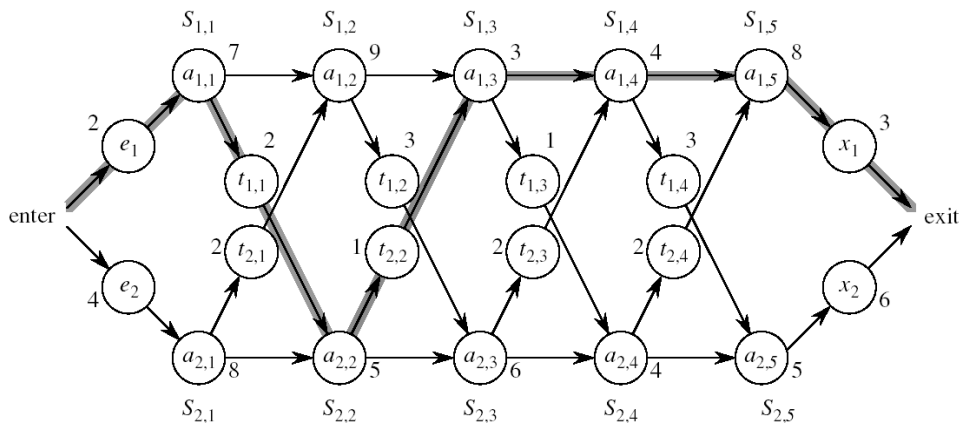
- Automobile factory with two assembly lines
  - Each line has  $n$  stations:  $S_{1,1}, \dots, S_{1,n}$  and  $S_{2,1}, \dots, S_{2,n}$
  - Corresponding stations  $S_{1,j}$  and  $S_{2,j}$  perform the same function but can take different amounts of time  $a_{1,j}$  and  $a_{2,j}$
  - Times to enter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



CSC611/ Lecture10

# Assembly Line

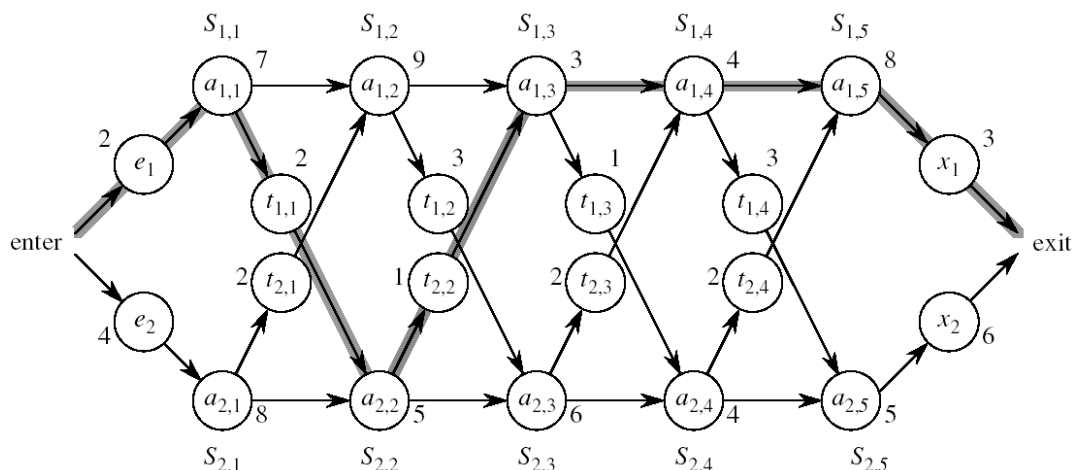
- After going through a station, the car can either:
  - stay on same line at no cost, or
  - transfer to other line: cost after  $S_{i,j}$  is  $t_{i,j}$ ,  $i = 1, 2, j = 1, \dots, n-1$



CSC611/ Lecture10

## Assembly Line Scheduling

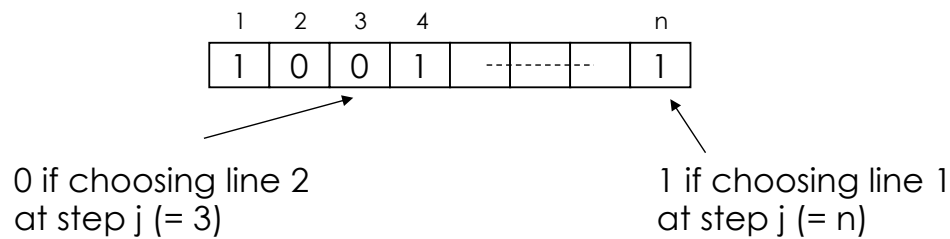
- Problem:  
What stations should be chosen from line 1 and what from line 2 in order to **minimize the total time through the factory for one car?**



CSC611/ Lecture10

# One Solution

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

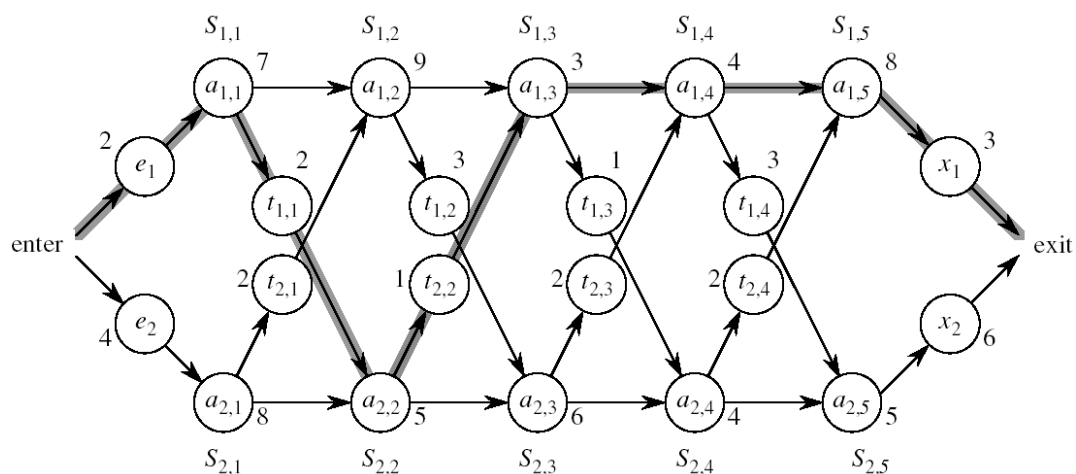


- There are  $2^n$  possible ways to choose stations
- Infeasible when  $n$  is large

CSC611/ Lecture10

## 1. Structure of the Optimal Solution

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

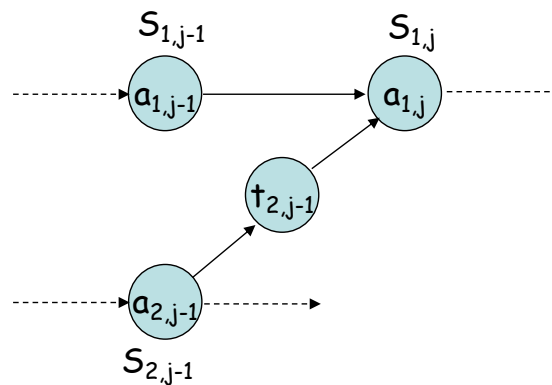


CSC611/ Lecture10

# 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  $S_{1,j}$

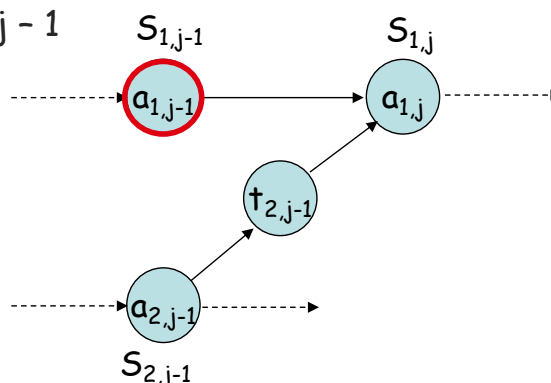


CSC611/ Lecture10

# 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 the 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  $S_{2,j-1}$



CSC611/ Lecture10



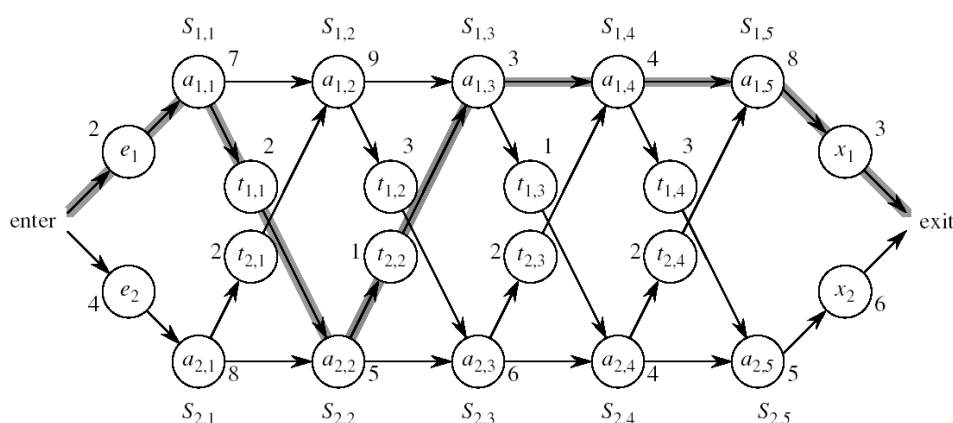
# Optimal Substructure

- **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

CSC611/ Lecture10

## 2. A Recursive Solution

- Define the value of an optimal solution in terms of the optimal solution to subproblems
- Assembly line subproblems
  - Finding the fastest way through each station  $j$  on each line  $i$  ( $i = 1, 2, j = 1, 2, \dots, n$ )

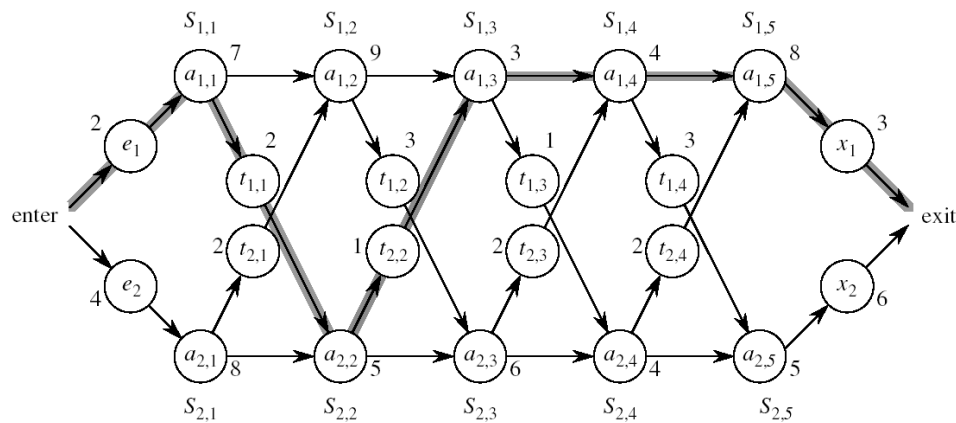


CSC611/ Lecture10

## 2. A Recursive Solution

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

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



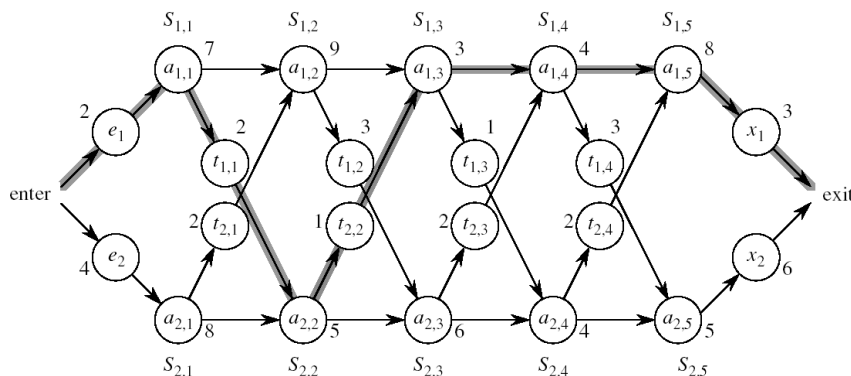
CSC611/ Lecture10

## 2. A Recursive Solution

- $f_i[j]$  = the fastest time to get from the starting point through station  $S_{i,j}$
- $j = 1$  (getting through station 1)

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

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



CSC611/ Lecture10

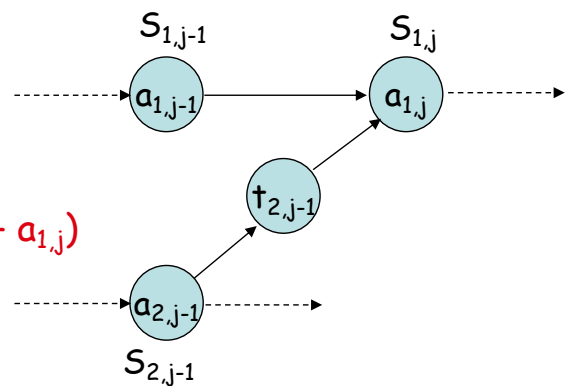
## 2. A Recursive Solution

- Compute  $f_i[j]$  for  $j = 2, 3, \dots, n$ , 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  $S_{1,j}$   

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

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



CSC611/ Lecture10

## 2. A Recursive Solution

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

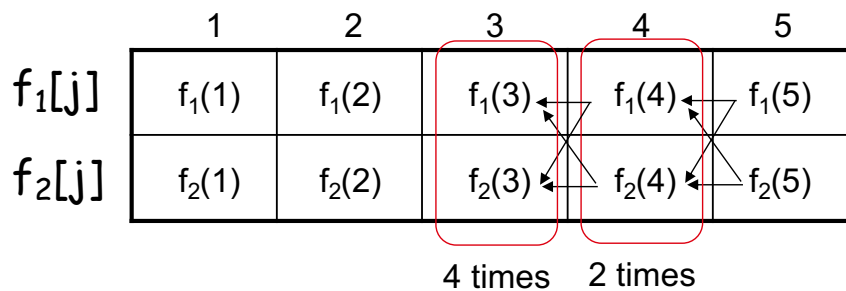
$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

CSC611/ Lecture10

### 3. Computing the Optimal Value

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

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

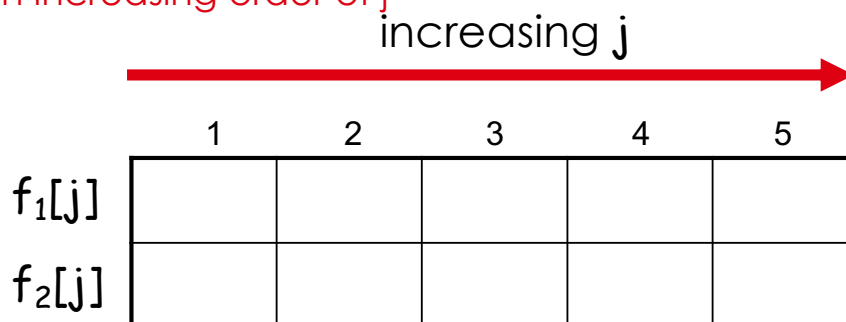


- Solving top-down would result in exponential running time

CSC611/ Lecture10

### 3. Computing the Optimal Value

- For  $j \geq 2$ , each value  $f_i[j]$  depends only on the values of  $f_1[j-1]$  and  $f_2[j-1]$
- Compute the values of  $f_i[j]$ 
  - in increasing order of  $j$



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

CSC611/ Lecture10

## 4. Construct the Optimal Solution

- We need the information about which line has been used at each station:
  - $l_i[j]$  – the line number (1, 2) whose station  $(j - 1)$  has been used to get in fastest time through  $S_{i,j}$ ,  $j = 2, 3, \dots, n$
  - $l^*$  – the line number (1, 2) whose station  $n$  has been used to get in fastest time through the exit point



CSC611/ Lecture10

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

1.  $f_1[1] \leftarrow e_1 + a_{1,1}$
  2.  $f_2[1] \leftarrow e_2 + a_{2,1}$
  3. **for**  $j \leftarrow 2$  **to**  $n$
  4.     **do if**  $f_1[j - 1] + a_{1,j} \leq f_2[j - 1] + t_{2,j-1} + a_{1,j}$
  5.         **then**  $f_1[j] \leftarrow f_1[j - 1] + a_{1,j}$
  6.          $l_1[j] \leftarrow 1$
  7.         **else**  $f_1[j] \leftarrow f_2[j - 1] + t_{2,j-1} + a_{1,j}$
  8.          $l_1[j] \leftarrow 2$
  9.     **if**  $f_2[j - 1] + a_{2,j} \leq f_1[j - 1] + t_{1,j-1} + a_{2,j}$
  10.         **then**  $f_2[j] \leftarrow f_2[j - 1] + a_{2,j}$
  11.          $l_2[j] \leftarrow 2$
  12.         **else**  $f_2[j] \leftarrow f_1[j - 1] + t_{1,j-1} + a_{2,j}$
  13.          $l_2[j] \leftarrow 1$
- Compute initial values of  $f_1$  and  $f_2$   
 Compute the values of  $f_1[j]$  and  $l_1[j]$   
 Compute the values of  $f_2[j]$  and  $l_2[j]$

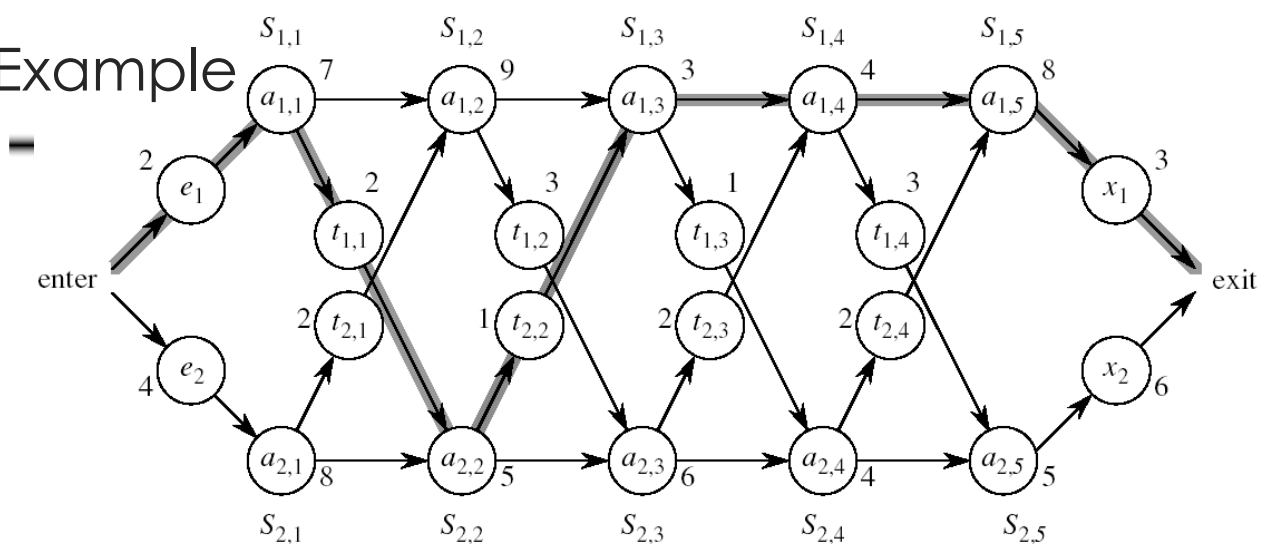
CSC611/ Lecture10

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

- |   |   |  |
|---|---|--|
| <p>14. if <math>f_1[n] + x_1 \leq f_2[n] + x_2</math></p> <p>15.    <b>then</b> <math>f^* = f_1[n] + x_1</math></p> <p>16.       <math>l^* = 1</math></p> <p>17.    <b>else</b> <math>f^* = f_2[n] + x_2</math></p> <p>18.       <math>l^* = 2</math></p> | } | <p>Compute the values of<br/>the fastest time through the<br/>entire factory</p> |
|---|---|--|

CSC611/ Lecture10

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 <sup>[1]</sup>	20 <sup>[2]</sup>	24 <sup>[1]</sup>	32 <sup>[1]</sup>
$f_2[j]$	12	16 <sup>[1]</sup>	22 <sup>[2]</sup>	25 <sup>[1]</sup>	30 <sup>[2]</sup>

$f^* = 35^{[1]}$

## 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 "  $j - 1$

line 1, station 5

line 1, station 4

line 1, station 3

line 2, station 2

line 1, station 1

	1	2	3	4	5
$f_1[j]$ $l^1[j]$	9	18 <sup>[1]</sup>	20 <sup>[2]</sup>	24 <sup>[1]</sup>	32 <sup>[1]</sup>
$f_2[j]$ $l^2[j]$	12	16 <sup>[1]</sup>	22 <sup>[2]</sup>	25 <sup>[1]</sup>	30 <sup>[2]</sup>

$l^* = 1$

CSC611/ Lecture10

## Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution
  - Fastest time through a station depends on the fastest time on previous stations
2. Recursively define the value of an optimal solution
  - $f_1[j] = \min(f_1[j - 1] + a_{1,j}, f_2[j - 1] + t_{2,j-1} + a_{1,j})$
3. Compute the value of an optimal solution in a bottom-up fashion
  - Fill in the fastest time table in increasing order of  $j$  (station #)
4. Construct an optimal solution from computed information
  - Use an additional table to help reconstruct the optimal solution

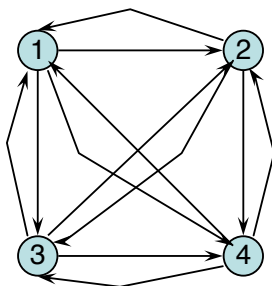
CSC611/ Lecture10

# Traveling Salesman Problem

- A tour maybe considered to be a simple path from 1 to 1
  - Then, a tour will consist of an edge (1, k) and a path from k to 1
  - Path goes through each vertex in  $V - \{1, k\}$  only once
- If tour is optimal
  - Path from k to 1 must be shortest k to 1 path going through all vertices  $V - \{1, k\}$
  - Thus, principal of optimality holds
- $g(i, s)$ 
  - Length of a shortest path from i to 1 through all vertices in S
- Principal of Optimality  $\Rightarrow$ 
  - $g(i, V - \{1\}) = \min \{c_{1k}, g(k, V - \{1, k\})\}$  or generalizing  
 $g(i, s) = \min \{c_{ij}, g(j, s - \{j\})\}$

CSC611/ Lecture10

## TSP: Example



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$$\begin{aligned}
 g(1, \{2, 3, 4\}) &= \min \{c_{12} + g(2, \{3, 4\}), \\
 &\quad c_{13} + g(3, \{2, 4\}), \\
 &\quad c_{14} + g(4, \{2, 3\})\} \\
 &= \min \{35, 40, 43\} \\
 &= 35
 \end{aligned}$$

$$\begin{aligned}
 g(2, \{3, 4\}) &= \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} = 25 \\
 g(3, \{2, 4\}) &= \min \{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\} = 25 \\
 g(4, \{2, 3\}) &= \min \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\} = 23
 \end{aligned}$$

$$\begin{aligned}
 g(2, \Phi) &= c_{21} = 5 \\
 g(3, \Phi) &= c_{31} = 6 \\
 g(4, \Phi) &= c_{41} = 8
 \end{aligned}$$

$$\begin{aligned}
 g(2, \{3\}) &= c_{23} + g(3, \Phi) = 15 \\
 g(2, \{4\}) &= c_{24} + g(4, \Phi) = 10 + 8 = 18 \\
 g(3, \{2\}) &= c_{32} + g(2, \Phi) = 13 + 6 = 18 \\
 g(3, \{4\}) &= c_{34} + g(4, \Phi) = 15 \\
 g(4, \{2\}) &= c_{42} + g(2, \Phi) = 13
 \end{aligned}$$

**Optimal Tour is 1, 2, 4, 3, 1**

CSC611/ Lecture10



# Matrix-Chain Multiplication

---

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

$$A_1 \cdot A_2 \cdots A_n$$

CSC611/ Lecture10

## Review of Matrix Multiplication

---

- The product  $C = AB$  of a  $p \times q$  matrix  $A$  and a  $q \times r$  matrix  $B$  is a  $p \times r$  matrix  $C$  given by

$$c[i, j] = \sum_{k=1}^q a[i, k]b[k, j], \quad \text{for } 1 \leq i \leq p \text{ and } 1 \leq j \leq r$$

- Complexity of Matrix multiplication:**  $C$  has  $pr$  entries and each entry takes  $\theta(q)$  time to compute so the total procedure takes  $\theta(pqr)$  time.

$$A = \begin{bmatrix} 1 & 8 & 9 \\ 7 & 6 & -1 \\ 5 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 8 \\ 7 & 6 \\ 5 & 5 \end{bmatrix}, \quad C = AB = \begin{bmatrix} 102 & 101 \\ 44 & 87 \\ 70 & 100 \end{bmatrix}$$

CSC611/ Lecture10

# Matrix-Chain Multiplication

- Matrix compatibility:

$$C = A \cdot B$$

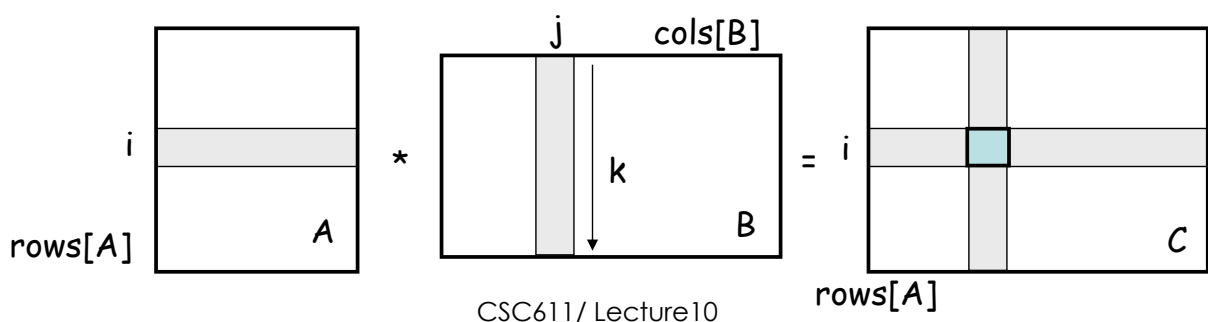
$$\text{col}_A = \text{row}_B$$

$$\text{row}_C = \text{row}_A$$

$$\text{col}_C = \text{col}_B$$

$$A_1 \cdot A_2 \cdot A_i \cdot A_{i+1} \cdots A_n$$

$$\text{col}_i = \text{row}_{i+1}$$



# Matrix-Chain Multiplication

- In what order should we multiply the matrices?

$$A_1 \cdot A_2 \cdots A_n$$

- Matrix multiplication is associative:

• *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 overall cost of executing the entire chain of multiplications

# Matrix Multiplication of ABC

---

- Given  $p \times q$  matrix A,  $q \times r$  matrix B and  $r \times s$  matrix C, ABC can be computed in two ways:  $(AB)C$  and  $A(BC)$
- The number of multiplications needed are:
  - $\text{mult}[(AB)C] = pqr + prs$ ;
  - $\text{mult}[A(BC)] = qrs + pqs$ ;
- Example
  - If  $p = 5$ ,  $q = 4$ ,  $r = 6$  and  $s = 2$ , then:
    - $\text{mult}[(AB)C] = 180$ ;
    - $\text{mult}[A(BC)] = 88$ ;
  - A big difference!

CSC611/ Lecture10

## Example

---

$$A_1 \cdot A_2 \cdot A_3$$

- $A_1$ :  $10 \times 100$
- $A_2$ :  $100 \times 5$
- $A_3$ :  $5 \times 50$

1.  $((A_1 \cdot A_2) \cdot A_3)$ :
  - $A_1 \cdot A_2$  takes  $10 \times 100 \times 5 = 5,000$   
(its size is  $10 \times 5$ )
  - $((A_1 \cdot A_2) \cdot A_3)$  takes  $10 \times 5 \times 50 = 2,500$
  - Total: 7,500 scalar multiplications
2.  $(A_1 \cdot (A_2 \cdot A_3))$ :
  - $A_2 \cdot A_3$  takes  $100 \times 5 \times 50 = 25,000$   
(its size is  $100 \times 50$ )
  - $(A_1 \cdot (A_2 \cdot A_3))$  takes  $10 \times 100 \times 50 = 50,000$
  - Total: 75,000 scalar multiplications

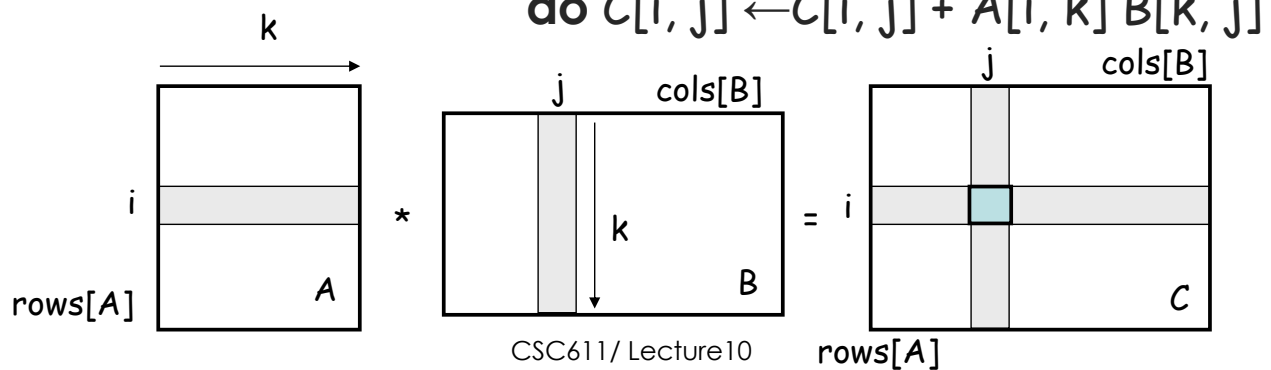
one order of magnitude difference!!

CSC611/ Lecture10

# MATRIX-MULTIPLY(A, B)

```

if columns[A]  $\neq$  rows[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] = 0
    for k  $\leftarrow$  1 to columns[A]
      do C[i, j]  $\leftarrow$  C[i, j] + A[i, k] B[k, j]
  
```



## Matrix-Chain Multiplication

- Given a chain of matrices  $\langle A_1, A_2, \dots, A_n \rangle$ , where for  $i = 1, 2, \dots, n$  matrix  $A_i$  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.

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

# Matrix-Chain Multiplication

---

- Recall that if each matrix  $A_i$  is  $p_{i-1} \times p_i$ , computing the matrix product  $A_{i..k} A_{k+1..j}$  takes  $p_{i-1} \times p_k \times p_j$  scalar multiplications

CSC611/ Lecture10

## 1. The Structure of an Optimal Parenthesization

---

- Notation:

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

- For  $i < j$ :

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

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

CSC611/ Lecture10

# Optimal Substructure

---

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

- Such a parenthesization of the “prefix”  $A_{i\dots k}$  must be an optimal parenthesization
- If there were a less costly way to parenthesize  $A_{i\dots k}$ , we could substitute that one in the parenthesization of  $A_{i\dots j}$  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**

CSC611/ Lecture10

## 2. A Recursive Solution

---

- Subproblem:

determine the minimum cost of parenthesizing

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

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

CSC611/ Lecture10

## 2. A Recursive Solution

---

Consider the subproblem of parenthesizing

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

$$= \underbrace{A_{i\dots k}}_{m[i, k]} \underbrace{A_{k+1\dots j}}_{m[k+1, j]} \quad \text{for } i \leq k < j$$

$p_{i-1}p_kp_j$

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

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

min # of multiplications  
to compute  $A_{i\dots k}$

min # of multiplications  
to compute  $A_{k+1\dots j}$

CSC611/ Lecture10

# of multiplications  
to compute  
 $A_{i\dots k}A_{k\dots j}$

## 2. A Recursive Solution

---

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

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

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

### 3. Computing the Optimal Costs

---

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

- How many subproblems do we have?

- Parenthesize  $A_{i..j}$   
for  $1 \leq i \leq j \leq n \Rightarrow \Theta(n^2)$
- One subproblem for each choice of  $i$  and  $j$

	1	2	3			n
n						
3						
2						
1						

i

j

CSC611/ Lecture10

### 3. Computing the Optimal Costs

---

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

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

$$A_{i..j} = A_{i..k} A_{k+1..j}$$

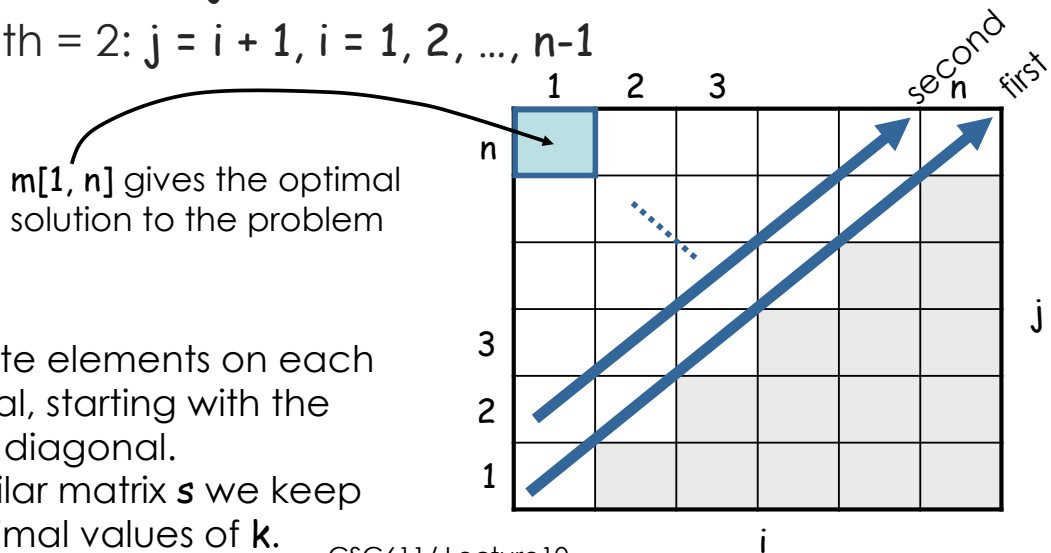
- Fill in  $m$  such that it corresponds to solving problems of increasing length



### 3. Computing the Optimal Costs

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

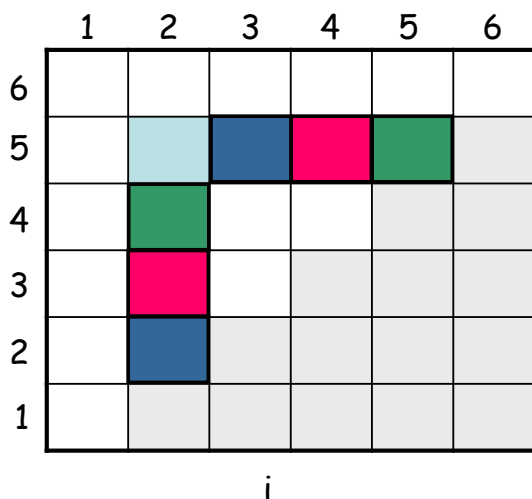
- Length = 1:  $i = j, i = 1, 2, \dots, n$
- Length = 2:  $j = i + 1, i = 1, 2, \dots, n-1$



CSC611/ Lecture10

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

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



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

Compute  $A_1 \cdot A_2 \cdot A_3$

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

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

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

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

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

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

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

CSC611/ Lecture10

	1	2	3
3	<sup>2</sup> 7500	<sup>2</sup> 25000	0
2	<sup>1</sup> 5000	0	
1	0		

## 4. Construct the Optimal Solution

- Store the optimal choice made at each subproblem
- $s[i, j] =$  a value of  $k$  such that an optimal parenthesization of  $A_{i..j}$  splits the product between  $A_k$  and  $A_{k+1}$

	1	2	3		n
n					
			k		
3					
2					
1					

i

j

## 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  $k = s[1, n]$

$$A_{1..n} = A_{1..k} \cdot A_{k+1..n}$$

$$A_{1..n} = A_{1..s[1, n]} \cdot A_{s[1, n]+1..n}$$

- For each subproduct recursively find the corresponding value of  $k$  that results in an optimal parenthesization

	1	2	3		n
n					
3					
2					
1					

i

j

CSC611/ Lecture10

## 4. Construct the Optimal Solution

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

	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	-					

i

j

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

CSC611/ Lecture10

# 4. Construct the Optimal Solution

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

```

if i = j
  then print "A";
  else
    print "("
    PRINT-OPT-PARENS(s, i, s[i, j])
    PRINT-OPT-PARENS(s, s[i, j] + 1, j)
    print ")"
  
```

	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	-					

i

j

CSC611/ Lecture10

Example:  $A_1 \cdots A_6$   $((A_1 (A_2 A_3)) ((A_4 A_5) A_6))$

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

```

if i = j
  then print "A";
  else print "("
    PRINT-OPT-PARENS(s, i, s[i, j])
    PRINT-OPT-PARENS(s, s[i, j] + 1, j)
    print ")"
  
```

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)  $\Rightarrow "A_3"$

)"

)" ...

$s[1..6, 1..6]$

	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	-					

i

j

# Memoization

---

- Top-down approach with the efficiency of typical bottom-up dynamic programming approach
- Maintains an entry in a table for the solution to each subproblem
  - **memoize** the inefficient recursive top-down 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

CSC611/ Lecture10

## Memoized Matrix-Chain

---

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

1.  $n \leftarrow \text{length}[p]$
  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( $p, 1, n$ )  $\leftarrow$  Top-down approach
- } Initialize the **m** table with large values that indicate whether the values of **m[i, j]** have been computed

CSC611/ Lecture10

# 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**  $m[i, j]$

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

CSC611/ Lecture10

## Dynamic Programming vs. Memoization

---

- Advantages of dynamic programming vs. memoized algorithms
  - No overhead for recursion
  - The regular pattern of table accesses may be used to reduce time or space requirements
- Advantages of memoized algorithms vs. dynamic programming
  - More intuitive

# Optimal Substructure - Examples

---

- Assembly line
  - Fastest way of going through a station  $j$  contains: the fastest way of going through station  $j-1$  on either line
- Matrix multiplication
  - Optimal parenthesization of  $A_i \cdot A_{i+1} \cdots A_j$  that splits the product between  $A_k$  and  $A_{k+1}$  contains:
    - an optimal solution to the problem of parenthesizing  $A_{i..k}$
    - an optimal solution to the problem of parenthesizing  $A_{k+1..j}$

CSC611/ Lecture10

## 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)

CSC611/ Lecture10

# 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 examine for each subproblem
- Assembly line
  - $\Theta(n)$  subproblems ( $n$  stations)  $\Theta(n)$  overall
  - 2 choices for each subproblem
- Matrix multiplication:
  - $\Theta(n^2)$  subproblems ( $1 \leq i \leq j \leq n$ )  $\Theta(n^3)$  overall
  - At most  $n-1$  choices

CSC611/ Lecture10

## The Knapsack Problem

---

- The 0-1 knapsack problem
  - A thief robbing a store finds  $n$  items: the  $i$ -th item is worth  $v_i$  dollars and weights  $w_i$  pounds ( $v_i, w_i$  integers)
  - The thief can only carry  $W$  pounds in his knapsack
  - Items must be taken entirely or left behind
  - Which items should the thief take to maximize the value of his load?
- The fractional knapsack problem
  - Similar to the above problem
  - The thief can take fractions of items

CSC611/ Lecture10



# The 0-1 Knapsack Problem

---

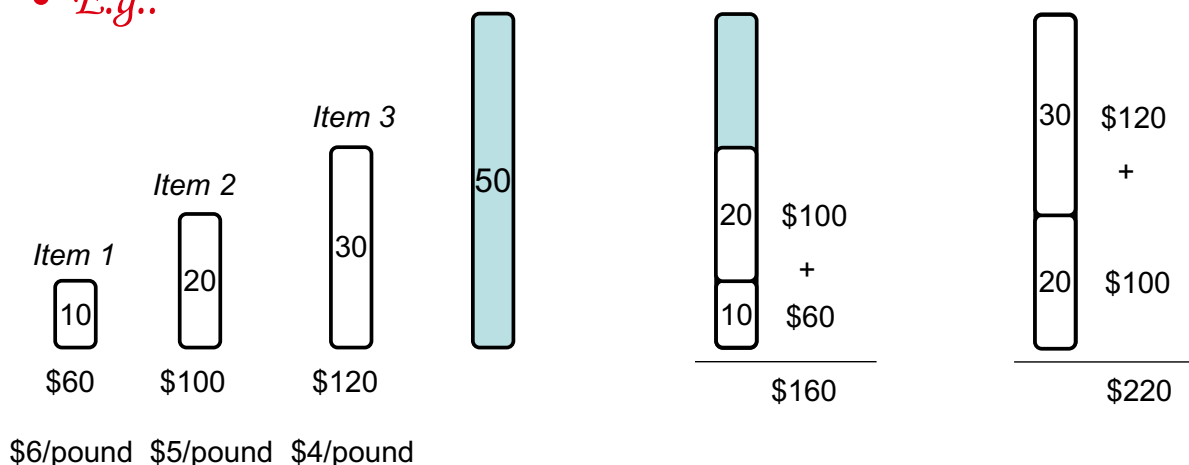
- Thief has a knapsack of capacity  $W$
- There are  $n$  items: for  $i$ -th item value  $v_i$  and weight  $w_i$
- Goal:
  - Find coefficients  $x_i$  so that for all  $x_i = \{0, 1\}$ ,  $i = 1, 2, \dots, n$   
 $\sum w_i x_i \leq W$  and  
 $\sum x_i v_i$  is maximum

CSC611/ Lecture10

## 0-1 Knapsack - Greedy Strategy

---

- *E.g.:*



- Cannot solve based on a greedy approach!

CSC611/ Lecture10

# 0-1 Knapsack - Dynamic Programming

- $P(i, w)$  – the maximum profit that can be obtained from items 1 to  $i$ , if the knapsack has size  $w$

- Case 1: thief takes item  $i$

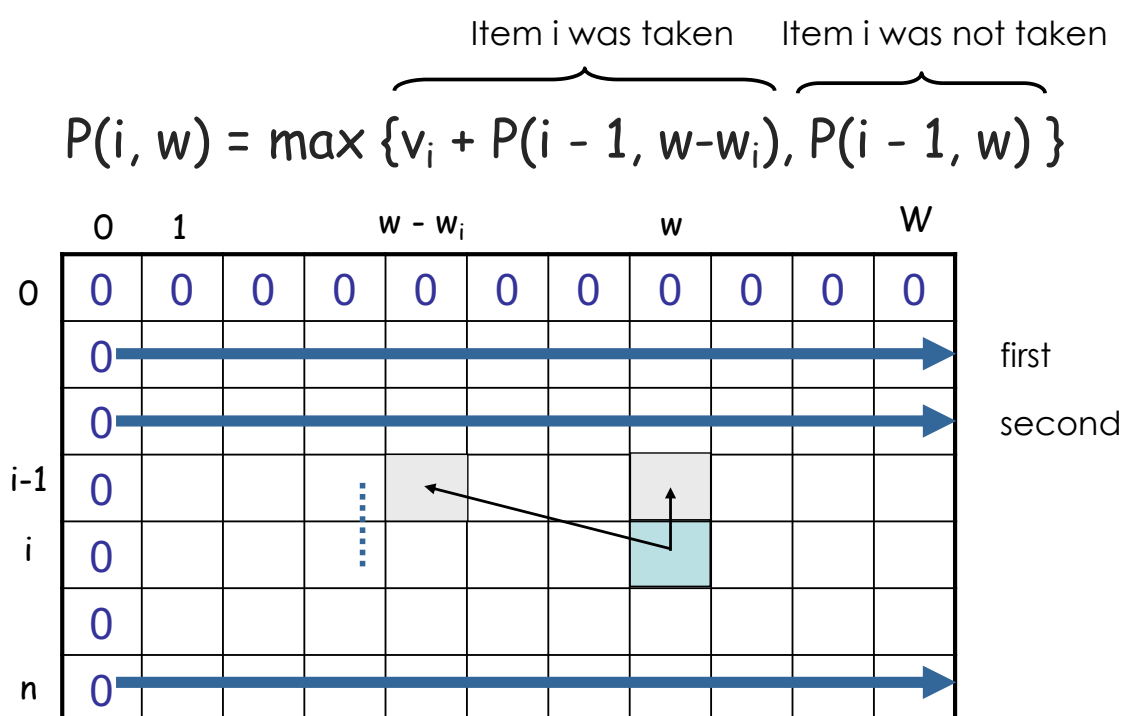
$$P(i, w) = v_i + P(i - 1, w - w_i)$$

- Case 2: thief does not take item  $i$

$$P(i, w) = P(i - 1, w)$$

CSC611/ Lecture10

# 0-1 Knapsack - Dynamic Programming



Example:

W = 5

$$P(i, w) = \max \{v_i + P(i - 1, w-w_i), P(i - 1, w) \}$$

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

$P(1, 1) = P(0, 1) = 0$   
 $P(1, 2) = \max\{12+0, 0\} = 12$   
 $P(1, 3) = \max\{12+0, 0\} = 12$   
 $P(1, 4) = \max\{12+0, 0\} = 12$   
 $P(1, 5) = \max\{12+0, 0\} = 12$

$$P(2, 1) = \max\{10+0, 0\} = 10$$

$$P(3, 1) = P(2, 1) = 10$$

$$P(4, 1) = P(3, 1) = 10$$

$$P(2, 2) = \max\{10+0, 12\} = 12$$

$$P(3, 2) = P(2, 2) = 12$$

$$P(4, 2) = \max\{15+0, 12\} = 15$$

$$P(2, 3) = \max\{10+12, 12\} = 22$$

$$P(3, 3) = \max\{20+0, 22\} = 22$$

$$P(4, 3) = \max\{15+10, 22\} = 25$$

$$P(2, 4) = \max\{10+12, 12\} = 22$$

$$P(3, 4) = \max\{20+10, 22\} = 30$$

$$P(4, 4) = \max\{15+12, 30\} = 30$$

$$P(2, 5) = \max\{10+12, 12\} = 22$$

$$P(3, 5) = \max\{20+12, 22\} = 32$$

$$P(4, 5) = \max\{15+22, 32\} = 37$$

CSC611/ Lecture10

## Reconstructing the Optimal Solution

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

- Item 4
- Item 2
- Item 1

- Start at  $P(n, W)$
- When you go left-up  $\Rightarrow$  item  $i$  has been taken
- When you go straight up  $\Rightarrow$  item  $i$  has not been taken

CSC611/ Lecture10

# Optimal Substructure

- Consider the most valuable load that weighs at most  $W$  pounds
- If we remove item  $j$  from this load  
 $\Rightarrow$  The remaining load must be the most valuable load weighing at most  $W - w_j$  that can be taken from the remaining  $n - 1$  items

CSC611/ Lecture10

# Overlapping Subproblems

$$P(i, w) = \max \{v_i + P(i - 1, w - w_i), P(i - 1, w)\}$$

	0	1				w					W
0	0	0	0	0	0	0	0	0	0	0	0
	0										
	0										
i-1	0										
i	0										
	0										
n	0										

*E.g.*: all the subproblems shown in grey may depend on  $P(i-1, w)$

CSC611/ Lecture10

# Longest Common Subsequence

---

- Given two sequences

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

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

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

- E.g.:*

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

- Subsequence of X:

- A subset of elements in the sequence taken in order (but not necessarily consecutive)

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

CSC611/ Lecture10

## Example

---

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

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


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

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


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

CSC611/ Lecture10

# 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:  $\Theta(n2^m)$

CSC611/ Lecture10

## 1. 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

$X = \langle A, B, D, \cancel{E} \rangle$

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

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

$Y = \langle Z, B, \cancel{E} \rangle$

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

CSC611/ Lecture10

# Notations

---

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

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

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

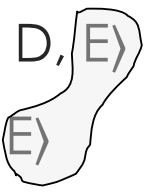
CSC611/ Lecture10

## 2. 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

CSC611/ Lecture10

## 2. 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  $Y_{j-1} = \langle Z, B \rangle$
- Optimal solution to a problem includes optimal solutions to subproblems

CSC611/ Lecture10

## 3. Computing the Length of the LCS

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

		0	1	2		n
		$y_j$	$y_1$	$y_2$		$y_n$
0	$x_i$	0	0	0	0	0
1	$x_1$	0	→			
2	$x_2$	0	→			
		0			⋮	
		0				
m	$x_m$	0	→			
					j	

first  
second  
i

CSC611/ Lecture10



## 4. 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
1 A	0				
2 B	0			$c[i-1, j]$	
3 C	0		$c[i, j-1]$	$\uparrow$	
	0				
m D	0				

j

i

A matrix  $b[i, j]$ :

- For a subproblem  $[i, j]$  it tells us what choice was made to obtain the optimal value
- If  $x_i = y_j$   
 $b[i, j] = "\nwarrow"$
- Else, if  
 $c[i-1, j] \geq c[i, j-1]$   
 $b[i, j] = "\uparrow"$
- else  
 $b[i, j] = "\leftarrow"$

CSC611/ Lecture10

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

```

1. for i ← 1 to m
2.   do c[i, 0] ← 0
3. for j ← 0 to n
4.   do c[0, j] ← 0
5. for i ← 1 to m
6.   do for j ← 1 to n
7.     do if  $x_i = y_j$ 
8.       then  $c[i, j] \leftarrow c[i-1, j-1] + 1$ 
9.          $b[i, j] \leftarrow "\nwarrow"$ 
10.    else if  $c[i-1, j] \geq c[i, j-1]$ 
11.      then  $c[i, j] \leftarrow c[i-1, j]$ 
12.         $b[i, j] \leftarrow "\uparrow"$ 
13.    else  $c[i, j] \leftarrow c[i, j-1]$ 
14.         $b[i, j] \leftarrow "\leftarrow"$ 
15. return c and b
  
```

} The length of the LCS is zero if one of the sequences is empty  
 } Case 1:  $x_i = y_j$   
 } Case 2:  $x_i \neq y_j$

Running time:  $\Theta(mn)$

CSC611/ Lecture10

# Example

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

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

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

If  $x_i = y_j$

$b[i, j] = \nwarrow$

Else if

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

$b[i, j] = \uparrow$

else

$b[i, j] = \leftarrow$

		Y <sub>j</sub>	B	D	C	A	B	A
0	x <sub>i</sub>	0	0	0	0	0	0	0
1	A	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

CSC611/ Lecture10

## 4. Constructing a LCS

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

		0	1	2	3	4	5	6
	Y <sub>j</sub>	B	D	C	A	B	A	
0	x <sub>i</sub>	0	0	0	0	0	0	0
1	A	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

CSC611/ Lecture10

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

---

1. **if**  $i = 0$  or  $j = 0$  Running time:  $\Theta(m + n)$
2.     **then return**
3. **if**  $b[i, j] = "\nwarrow"$
4.     **then** PRINT-LCS( $b, X, i - 1, j - 1$ )
5.         print  $x_i$
6. **else if**  $b[i, j] = "\uparrow"$
7.     **then** PRINT-LCS( $b, X, i - 1, j$ )
8.     **else** PRINT-LCS( $b, X, i, j - 1$ )

Initial call: PRINT-LCS( $b, X, \text{length}[X], \text{length}[Y]$ )

CSC611/ Lecture10

## 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

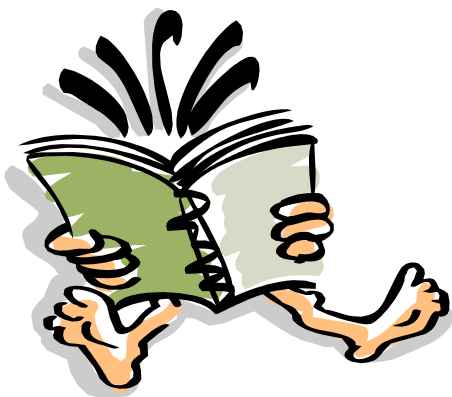
---

- 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

CSC611/ Lecture10

## Readings

---



- Chapters 14, 15

CSC611/ Lecture10