Chapter 15

Dynamic Programming

Chapter 15

The slides for this course are based on the course textbook: Cormen, Leiserson, Rivest, and Stein, Introduction to Algorithms, 3rd edition, The MIT Press, McGraw-Hill, 2010.

Many of the slides were provided by the publisher for use with the textbook. They are copyrighted, 2010.

These slides are for classroom use only, and may be used only by students in this specific course this semester. They are NOT a substitute for reading the textbook!

Chapter 15 Topics

- Assembly-line scheduling
- Matrix-chain multiplication
- Elements of dynamic programming
- Longest common subsequence
- Optimal binary search trees

Dynamic Programming

- General approach – combine solutions to subproblems to get the solution to an problem
- Unlike Divide and Conquer in that
  - subproblems are dependent rather than independent
  - bottom-up approach
  - save the values of subproblems in a table and use them more than one time

Usual Approach

- Start with the smallest, simplest subproblems
- Combine “appropriate” subproblem solutions to get a solution to the bigger problem
- If you have a Divide and Conquer algorithm that does a lot of duplicate computation, you can often find a Dynamic Programming solution that is more efficient

A Simple Example

- Calculating binomial coefficients

$$\binom{n}{k}$$

- *n choose k* is the number of different combinations of *n* things taken *k* at a time
- These are also the coefficients of the binomial expansion $(x+y)^n$

Two Definitions

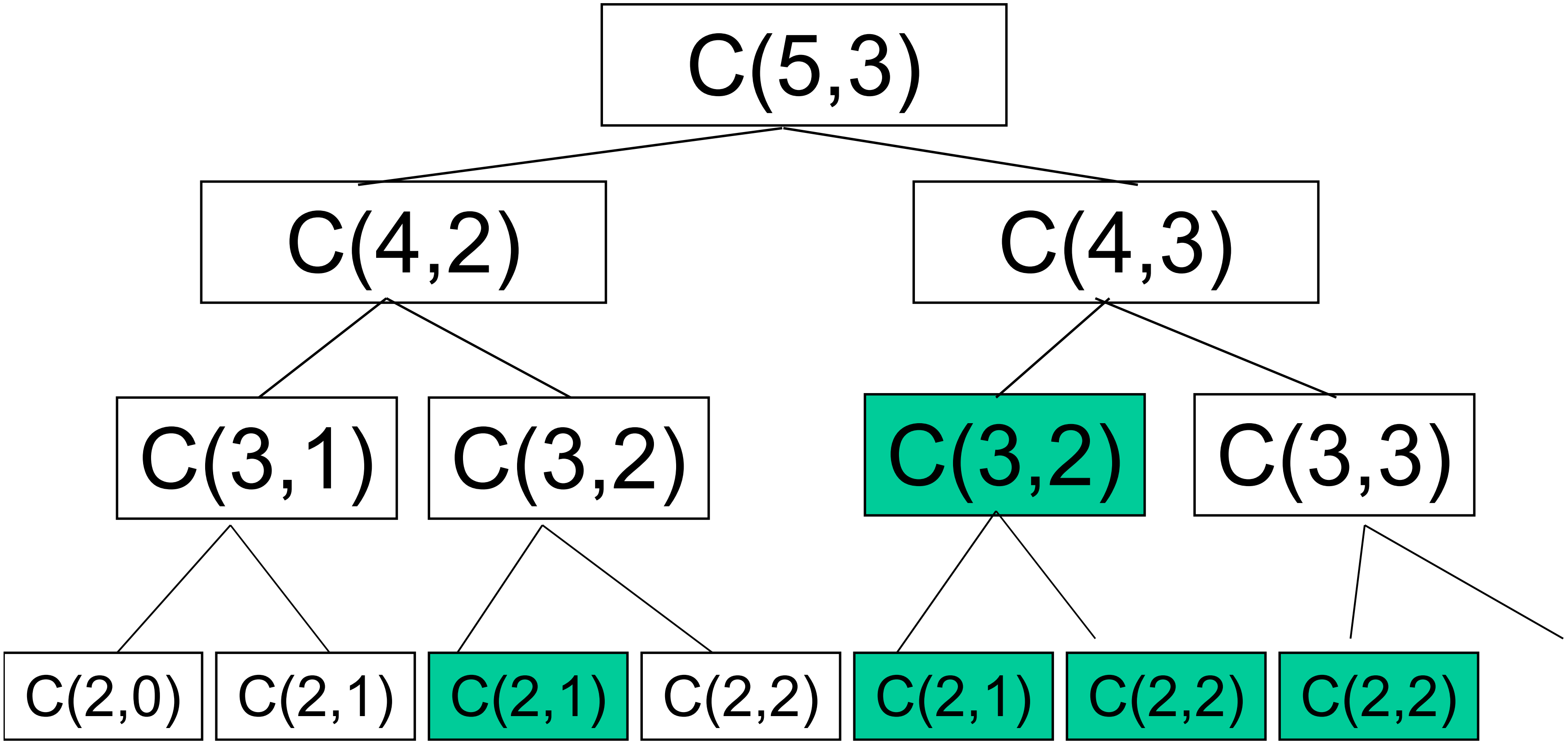
$$\text{Defintion} \quad 1: \binom{n}{k} = \frac{n!}{(n-k)!\,k!}$$

$$\text{Defintion} \quad 2:$$

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

Algorithm for Recursive Definition

```
function C(n,k)
  if k = 0 or k = n then
    return 1
  else
    return C(n-1, k-1) + C(n-1, k)
```



C(5,3)

C(4,2) C(4,3)

C(3,1) C(3,2) C(3,2) C(3,3)

C(2,0) C(2,1) C(2,1) C(2,2) C(2,1) C(2,2) C(2,2)

etc.

Complexity of Divide and Conquer Algorithm

- Time complexity is Ω(2^n)
- But we did a lot of duplicate computation
- Dynamic programming approach
  - Store solutions to subproblems in a table
  - Bottom-up approach

```
                 2  1
function C(n,k)
  if k = 0 or k = n then
    return 1
  else             1    0          1     1
    return C(n-1, k-1) + C(n-1, k)
```

k

n

	0	*1*	*2*	*3*
1	1	1		
2	1	2	1	
3	1	3?	3 ?	1
4	1		6	4
5	1			10

Analysis of Dynamic Programming Version

- Time complexity

  O(n k)

- Storage requirements
  - full table O(n k)
  - less space often possible

Typical Problem

- Dynamic Programming is often used for optimization problems that satisfy the principle of optimality
- Principle of optimality
  - In an optimal sequence of decisions or choices, each subsequence must be optimal

Optimization Problems

- Problem has many possible solutions
- Each solution has a value
- Goal is to find the solution with the optimal value

Steps in Dynamic Programming Solutions to Optimization Problems

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 manner
4. Construct an optimal solution from computed information

Assembly-Line Scheduling

Assembly-Line Scheduling

- Problem: 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.
- Brute force approach: 2^n possibilities!

Assembly-Line Scheduling

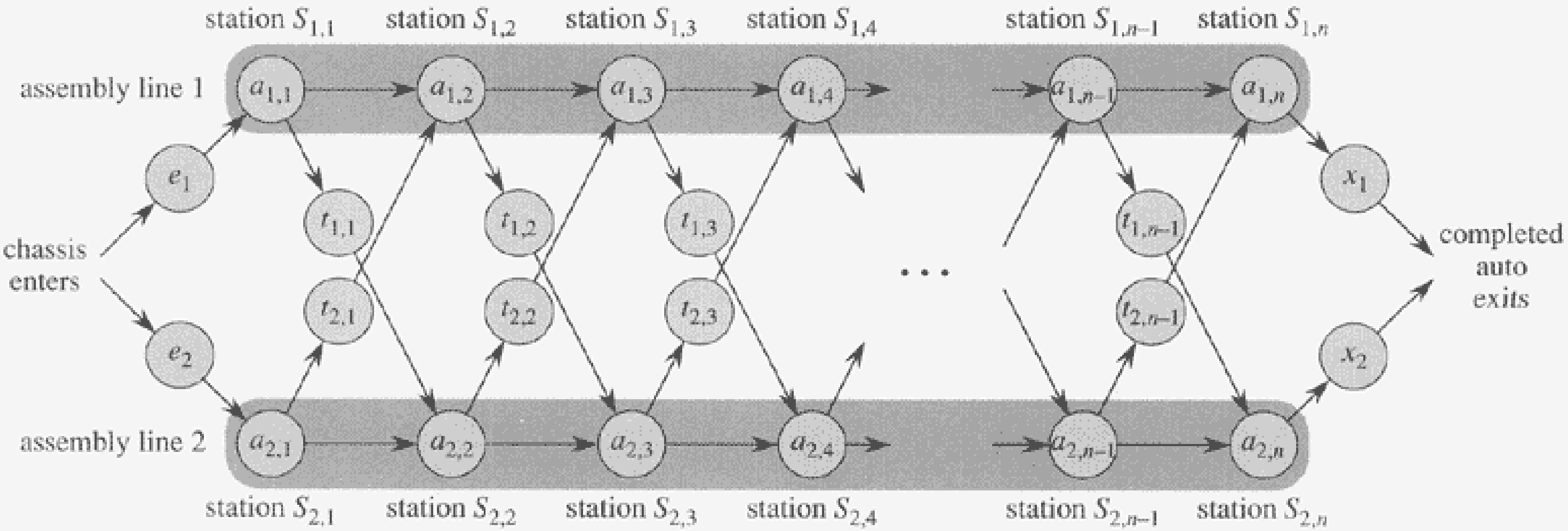


Figure 15.1 A manufacturing problem to find the fastest way through a factory. There are two assembly lines, each with n stations; the jth station on line i is denoted $S_{i,j}$ and the assembly time at that station is $a_{i,j}$. An automobile chassis enters the factory, and goes onto line i (where $i = 1$ or 2), taking e_i time. After going through the jth station on a line, the chassis goes on to the $(j+1)$st station on either line. There is no transfer cost if it stays on the same line, but it takes time $t_{i,j}$ to transfer to the other line after station $S_{i,j}$. After exiting the nth station on a line, it takes x_i time for the completed auto to exit the factory. 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.

Step 1: Structure of Optimal Solution

- Fastest way through station $S_{1,j}$ is either:
  - the fastest way through $S_{1,j-1}$ and then directly through $S_{1,j}$, or
  - the fastest way through $S_{2,j-1}$, a transfer from line 2 to line 1, and then through $S_{1,j}$.
- There is a symmetric argument for fastest way through $S_{2,j}$.

Step 2: Recursive Solution

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

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

Step 3: Computing Fastest Times

- Recursive algorithm based on equations 15.1, 15.6, and 15.7 will have running time of $\Theta(2^n)$.
- Better way: Compute the $f_i[j]$ values in order of increasing station numbers; time will be $\Theta(n)$.

FASTEST-WAY

Think about it this way:

For both assembly lines, simultaneously, work your way from the entrance to the exit. As you encounter each station:

1. Stand on the station and ask yourself, “What was the fastest way for me to get to and through this station?”

   A. For the first station on either assembly line, just add the cost of the station you are standing on to the entrance cost to the line you are in.

FASTEST-WAY

B. For station 2 through *n*, the cost is computed this way:

1) Add the cost of the station you are standing on to the cost of the previous station in your line.

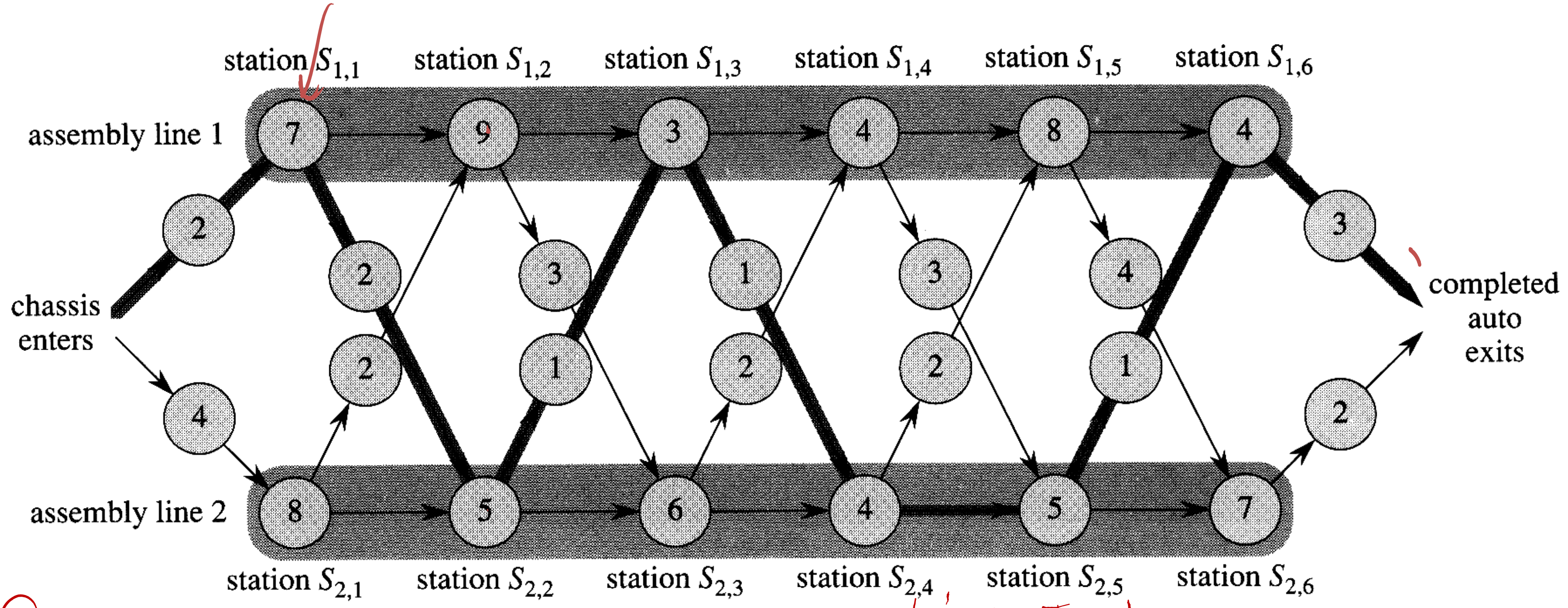
2) Add the cost of the station you are standing on to the cost of the previous station in the *other* assembly line, plus the transit cost of moving from one line to another.

3) The min of these 2 values is your cost.

FASTEST-WAY

2. Record the cost of the station in Table 1. In Table 2, keep track of the previous station you passed through on your fastest way to this station.
3. If you are on one of the two end stations, add the cost of the station to the exit cost for that station. The station with the *min* total value is the station from which you want to exit the assembly process.

FASTEST-WAY



f	1	2	3	4	5	6
1	9	18	20	24	32	35
2	12	16	22	25	30	37

3

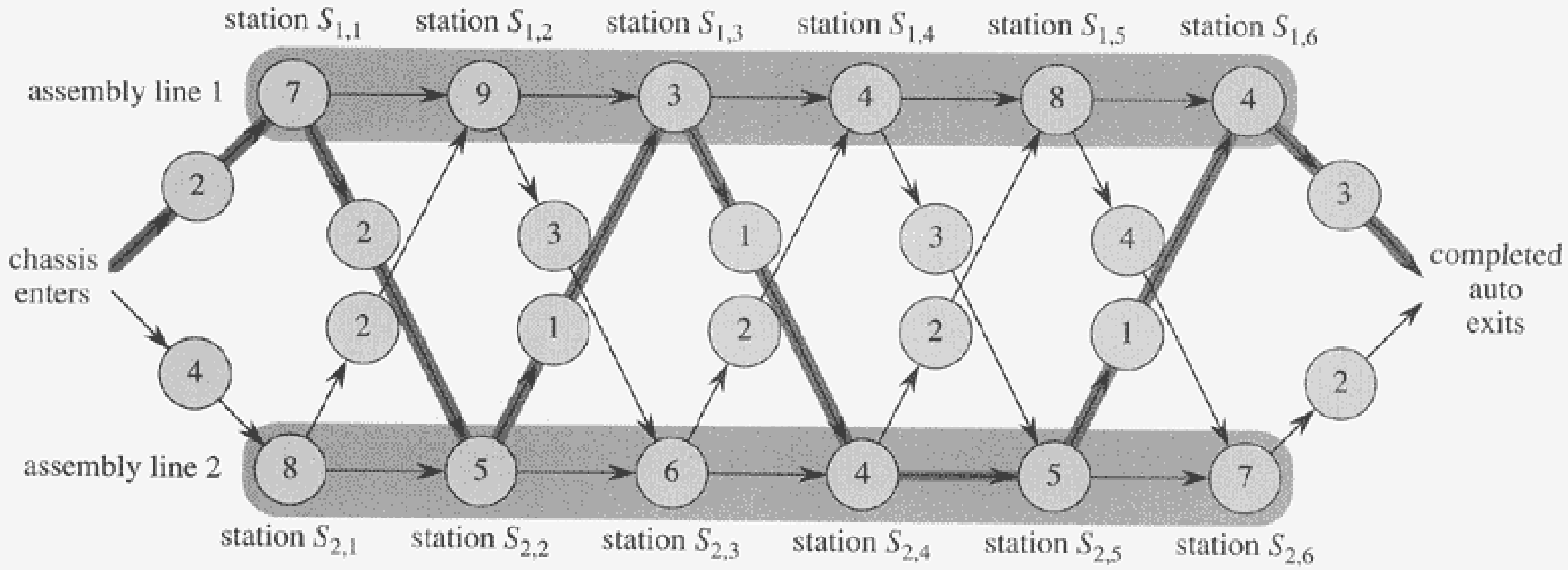
38

Table 1

optimal stations

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

Table 2



(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)

Figure 15.2 **(a)** An instance of the assembly-line problem with costs e_i, $a_{i,j}$, $t_{i,j}$, and x_i indicated. The heavily shaded path indicates the fastest way through the factory. **(b)** The values of $f_i[j]$, f^*, $l_i[j]$, and l^* for the instance in part (a).

FASTEST-WAY

```
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 $f_1[j-1] + a_{1,j} \le f_2[j-1] + t_{2,j-1} + a_{1,j}$
            then $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$
                 $l_1[j] \leftarrow 1$
            else $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$
                 $l_1[j] \leftarrow 2$
        if $f_2[j-1] + a_{2,j} \le f_1[j-1] + t_{1,j-1} + a_{2,j}$
            then $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$
                 $l_2[j] \leftarrow 2$
            else $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$
                 $l_2[j] \leftarrow 1$
    if $f_1[n] + x_1 \le f_2[n] + x_2$
        then $f^* = f_1[n] + x_1$
             $l^* = 1$
        else $f^* = f_2[n] + x_2$
             $l^* = 2$
```

FASTEST-WAY

Why is this the fastest way?

1. Because we are not re-doing any of our steps. We start at the beginning and work our way to the end, saving the work that we have done so far at each stage. The recursive solution makes us recalculate solutions we already calculated previously.
2. Because we skip some of the possible paths. The fastest path to $a_{1,4}$ must include the fastest path to $a_{1,3}$ or $a_{2,3}$. This algorithm ignores all other possible paths to $a_{1,4}$.

Step 4: Construct an Optimal Solution

```
PRINT-STATIONS
i ← $l^*$
print “line ” i “, station ” n
for j ← n downto 2 do
    i ← $l_i[j]$
    print “line” i “,station” j-1
```

Matrix-Chain Multiplication

Matrix-Chain Multiplication

Matrix multiplication is another example of a problem for which the dynamic programming approach can save us a lot of work.

We can multiply two matrices only if they are compatible; the number of rows of one must equal the number of columns of the other.

Matrix-Chain Multiplication

We know how to multiply 2 matrices:

```
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] $\leftarrow$ 0
               for k $\leftarrow$ 1 to columns[A] do
                  C[i, j] $\leftarrow$ C[i, j] +
                   A[i, k] * B[k, j]
return C
```

Matrix-Chain Multiplication

But what if we want to multiply a sequence of matrices, such as:

$A_1 = 20 \text{ x } 100$, $A_2 = 100 \text{ x } 40$, $A_3 = 40 \text{ x } 20$

and return the product of this “chain” of matrix multiplications?

We could multiply $A_1 \text{ x } A_2$ to get $B_1 = 20 \text{ x } 40$, and then multiply $B_1 \text{ x } A_3$ to get $C = 20 \text{ x } 20$, or

We could multiply $A_2 \text{ x } A_3$ to get $B_2 = 100 \text{ x } 20$, and then multiply $A_1 \text{ x } B_2$ to get $C = 20 \text{ x } 20$

Matrix-Chain Multiplication

We can enclose the components of the sequence in parentheses to show in what order we would carry out the matrix multiplications:

cost:

$(A_1 \text{ x } A_2) \text{ x } A_3$ or $20 \times 100 \times 40 + 20 \times 40 \times 20$

$A_1 \text{ x } (A_2 \text{ x } A_3) \rightarrow 100 \times 40 \times 20 + 20 \times 100 \times 20$

Matrix Chain Multiplication

$$A_1 \times A_2 \times A_3 \times .... \times A_n$$

- Matrix multiplication is associative
- We can determine all ways that a sequence can be parenthesized give the same answer
- But some are much less expensive to compute

Matrix-Chain Multiplication

With a sequence of several matrices, the order in which we multiply them may make a difference in the number of multiplication operations required.

The matrix chain multiplication problem asks:

Given a sequence $< A_1, A_2, \dots A_n>$ of matrices to be multiplied, how do we parenthesize the sequence so to require the fewest multiplication operations?

Matrix Chain Multiplication Problem

- Given a chain $< A_1,A_2, . . .,A_n>$ of n matrices, where i = 1, 2, . . ., n and matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 \times A_2 \times .... \times A_n$ in a way that minimizes the number of scalar multiplications

Example

Matrix	Dimensions
A	13 x 5
B	5 x 89
C	89 x 3
D	3 x 34
M = A B C D	13 x 34

Matrix	*Dimensions*
A	13 x 5
B	5 x 89
C	89 x 3
D	3 x 34

13x3

5x3

13x89

89x34

13x34

Multiply:	*# of operations*	*Resulting matrix*
(A B)	13 x 5 x 89	13 x 89
((AB)C)	13 x 89 x 3	13 x 3
(((AB)C)D)	13 x 3 x 34	13 x 34

Comparison of results

Parenthesization	Scalar multiplications
1 ((A B) C) D	10,582
2 (A B) (C D)	54,201
3 (A (B C)) D	2,856
4 A ((B C) D)	4,055
5 A (B (C D))	26,418

Number of Parenthesizations

$T(n) = \text{number of different ways to parenthesize}$

Make first cut between position i and $(i+1)$ where

$(1 \leq \mathrm{i} < \mathrm{n})$

$M = M_1 M_{2\ldots} M_i M_{i+1} M_{i+2} \ldots M_n$

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

T(n) ways to parenthesize

n	1	2	3	4	5	10	15
T(n)	1	1	2	5	14	4,862	2,674,440

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

Catalan numbers

Very fast growing exponential

$$T(n) = \Omega\left(\frac{4^n}{n}\right)$$

Steps in DP Solutions to Optimization Problems

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 manner

4 Construct an optimal solution from computed information

Step 1

- Show that the principle of optimality applies
  - An optimal solution to the problem contains within it optimal solutions to subproblems
  - Let $A_{i..j}$ be the optimal way to parenthesize $A_iA_{i+1}...A_j$
  - Suppose the optimal solution has the first split at position k

    $A_{1..k}$ $A_{k+1..n}$
  - Each of these subproblems must be optimally parenthesized

Step 1 continued

- Example: $A_1A_2A_3A_4A_5A_6$

- Suppose this is the optimal solution $(A_1A_2)(A_3A_4A_5A_6)$, where

  $i = 1, k = 2, j = 6$

- The subproblem $(A_3A_4A_5A_6)$ must be optimally parenthesized

Step 2: Recursive Solution

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

The running time of this recurrence is:
$\Omega(2^n)$

Step 2: Example

A: 20 x 2

B: 2 x 30

C: 30 x 12

D: 12 x 8

$p_0 = 20$

$p_1 = 2$

$p_2 = 30$

$p_3 = 12$

$p_4 = 8$

Step 2: Example

A(B(CD)): 30x12x8 + 2x30x8 + 20x2x8 = 3,680
(AB)(CD): 20x2x30 + 30x12x8 + 20x30x8 = 8,880
A((BC)D): 2x30x12 + 2x12x8 + 20x2x8 = 1,232
(A(BC))D: 2x30x12 + 20x2x12 + 20x12x8 = 3,120
((AB)C)D: 20x2x30 + 20x30x12 + 20x12x8 = 10,320

We want the MIN of these matrix multiplication operations. That is, we want to know the optimal order for multiplying *n* matrices.

Step 3: Computing Optimal Costs

```
MATRIX-CHAIN-ORDER (p)
n ← length[p] - 1
for i ← 1 to n do
    m[i, i] ← 0
for L ← 2 to n do   ▷ L is the chain length
    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] + $p_{i-1}p_kp_j$
            if q < m[i, j]
                then m[i, j] ← q
                     s[i, j] ← k
return m and s
```

Step 4: Constructing Optimal Solution

- Consider two procedures:
  - one that prints optimal parenthesization for $A_{i..j}$
  - one that computes $A_{i..j}$ using optimal parenthesization

Printing Optimal Parenthesization

```
PRINT-OPTIMAL-PARENS (s, i, j)
  if i = j
    then print “A”$_i$
    else print “(”
           PRINT-OPTIMAL-PARENS (s, i, s[i, j])
           PRINT-OPTIMAL-PARENS (s, s[i, j] + 1, j)
           print “)”
```

Multiplying Using Optimal Parenthesization

```
MATRIX-CHAIN-MULTIPLY (A, s, i, j)
   if j > i
      then X ← MATRIX-CHAIN-MULTIPLY (A, s, i, s[i, j])
           Y ← MATRIX-CHAIN-MULTIPLY (A, s, s[i, j] + 1, j)
           return MATRIX-MULTIPLY (X, Y)
      else return $A_i$
```

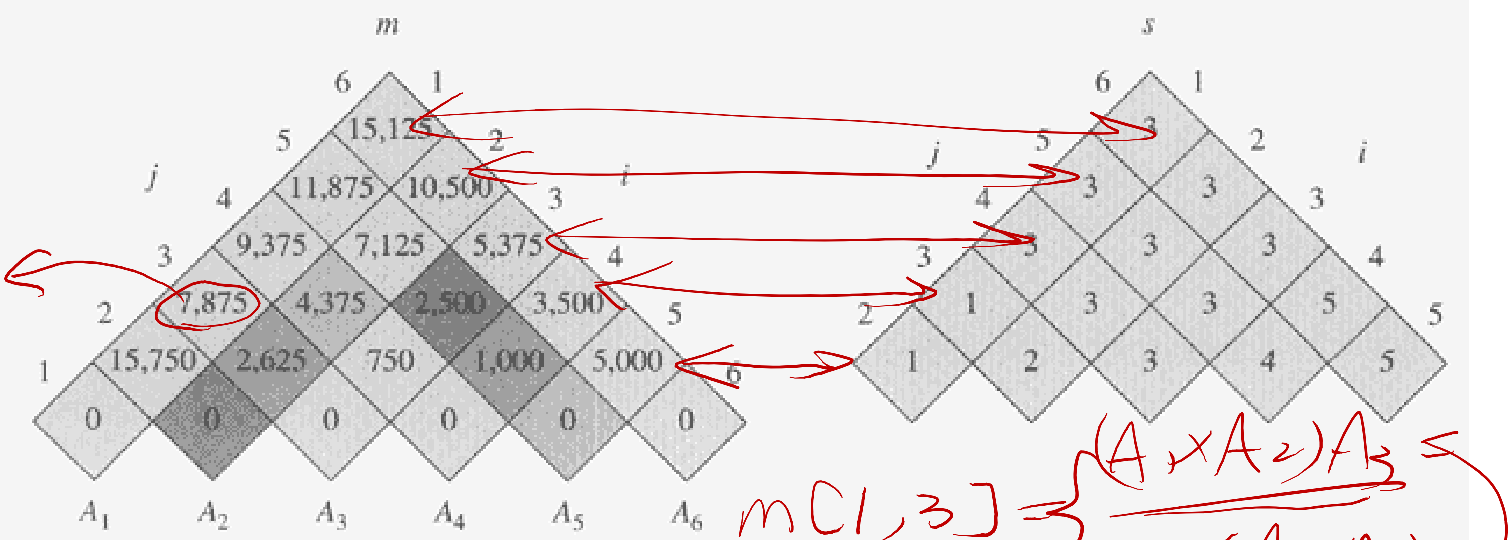
Multiplying Using Optimal Parenthesization

```
MATRIX-CHAIN-MULTIPLY (A, s, i, j)
    if j > i
      then X ← MATRIX-CHAIN-MULTIPLY (A, s, i, s[i, j])
           Y ← MATRIX-CHAIN-MULTIPLY (A, s, s[i, j] + 1, j)
           return MATRIX-MULTIPLY (X, Y)
      else return $A_i$
```

Initial call:

MATRIX-CHAIN-MULTIPLY(A,s,1,n) where **A = < A1,A2, . . .,An>**

(From Exercise 15.2-2)



$m[1,3] = \begin{cases} (A_1 \times A_2)A_3 \\ A_1 \times (A_2 \times A_3) \end{cases}$

Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

$p_0 = 30$ $p_4 = 10$
$p_1 = 35$ $p_5 = 20$
$p_2 = 15$ $p_6 = 25$
$p_3 = 5$

$= \min \begin{cases} m[1,2] + m[3,3] + p_0 \times p_2 \times p_3 \\ m[1,1] + m[2,3] + p_0 \times p_1 \times p_3 \end{cases}$

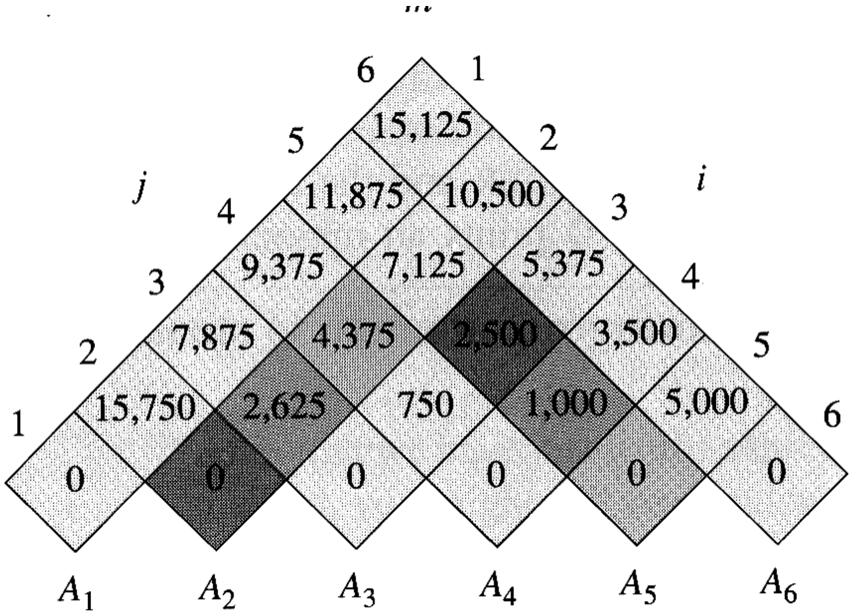
The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1,6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

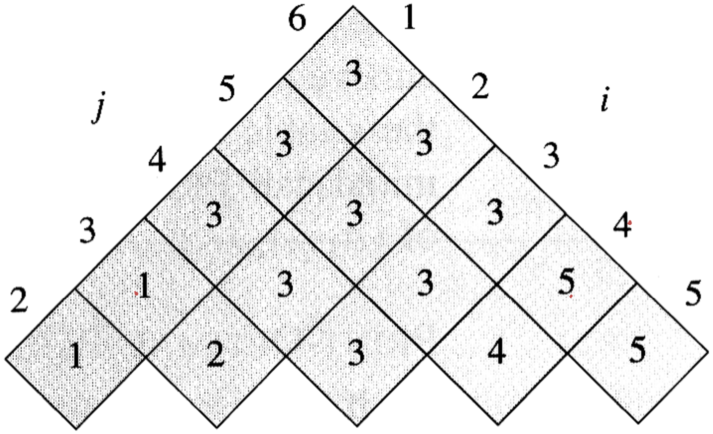
$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125\,.$$

$= \min \begin{cases} 15750 + 0 + 30 \times 15 \times 5 \\ 0 + 2625 + 30 \times 5 \times 35 \end{cases}$

$= 7875$

Building the Tables





Compute k, where k = the position to at which to split the sequence of matrix multiplications in order to minimize the cost to multiply a specific pair of tables. For example,

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p1p2p5 = 0 + 2500 + 35 * 15 * 20 = 13000 \\ m[2, 3] + m[4, 5] + p1p3p5 = 2625 + 1000 + 35 * 5 * 20 = 7125 \\ m[2, 4] + m[5,5] + p1p4p5 = 4375 + 0 + 35 * 10 * 20 = 11375 \end{cases}$$

$$= 7125$$

Elements of Dynamic Programming

- Optimal Substructure
- Overlapping subproblems

Optimal Substructure

- An optimal solution to the problem contains within it optimal solutions to subproblems
- Steps in discovering optimal substructure:
    - Show that a solution to the problem involves making a choice which leaves one or more subproblems to be solved
    - You assume you are given a choice that leads to an optimal solution
    - Given this choice, which subproblems ensue and how do you characterize the resulting space of subproblems?
    - Show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal

Optimal Substructure (Subtleties)

- Don’t assume that optimal substructure applies where it doesn’t
- Examples:
  - Unweighted shortest path (i.e., finding a simple path in a graph from node *u* to node *v* consisting of the fewest edges) does exhibit optimal substructure
  - Unweighted longest path *does not*

Overlapping subproblems

- The space of subproblems must be “small” in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems.
- DP algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table (for constant time lookup)

Comments on Dynamic Programming

- Useful when the problem can be reduced to several “overlapping” subproblems.
- All possible subproblems are computed
- Computation is done by maintaining a large matrix
- Usually has large space requirement
- Running times are usually at least quadratic

Longest Common Subsequence

Longest Common Subsequence

- Definition 1: Subsequence

Given a sequence

$X = < x_1, x_2, . . . , x_m>$

then another sequence

$Z = < z_1, z_2, . . . , z_k>$

is a subsequence of X if there exists a strictly increasing sequence $<i_1, i_2, . . . , i_k>$ of indices of x such that for all j = 1,2,...k we have $x_{i_j} = z_j$

Example

The items in the sequence do not have to be adjacent. For example:

X = <A,B,D,F,M,Q>

Z = <B, F, M>

Z is a subsequence of X with index sequence <2,4,5>

More Definitions

- Definition 2: Common subsequence
  - Given 2 sequences X and Y, we say Z is a common subsequence of X and Y if Z is a subsequence of X and a subsequence of Y
- Defintion 3: Longest common subsequence problem
  - Given X = < x_1, x_2, . . . , x_m> and Y = < y_1, y_2, . . . , y_n> find a maximum length common subsequence of X and Y

Example

X = <A,B,C,B,D,A,B>

Y = <B,D,C,A,B,A>

What is the *longest common subsequence*?

Is it <B, C, A, B> ?

Or maybe <B, D, A, B>?

Or <B, C, B, A>?

Is there a subsequence longer than 4?

Yet More Definitions

- Definition 4: Prefix of a subsequence

  If $X = < x_1, x_2, . . . , x_m>$, the ith prefix of X for i = 0,1,...,m is $X_i = < x_1, x_2, . . . , x_i>$

- Example
  - if X = <A,B,C,D,E,F,H,I,J,L> then:

    X_4 = <A,B,C,D>

    X_0 = <>

Optimal Substructure

- Theorem 15.1 Optimal Substructure of LCS

  Let X = < x_1, x_2, . . . , x_m> and Y = < y_1, y_2, . . . , y_n> be sequences and let Z = < z_1, z_2, . . . ,z_k> be any LCS of X and Y

  – Case 1
    - if $x_m = y_n$ then there is one subproblem to solve:
      find a LCS of X_{m-1} and Y_{n-1} and append x_m

  – Case 2
    - if $x_m \neq y_n$ then there are two subproblems:
      - find an LCS of $X_m$and Y_{n-1}
      - find an LCS of X_{m-1} and Y_n
      - pick the longer of the two

Cost of Optimal Solution

- Cost is the length of the common subsequence
- We want to pick the longest one
- Let c[i,j] be the length of an LCS of the sequences X_i and Y_j
- Base case is an empty subsequence:

  in that case, c[i,j] = 0 because there is no LCS

Recurrence for LCS

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

Recurrence for LCS

There are 2^m subsequences, so the running time of this brute force method would be $O(2^m)$.

Using a Dynamic Programming approach, we can compute all O(nm) distinct subproblems in O(nm) time, and use them to construct a LCS in O(m + n) time.

Dynamic Programming Solution

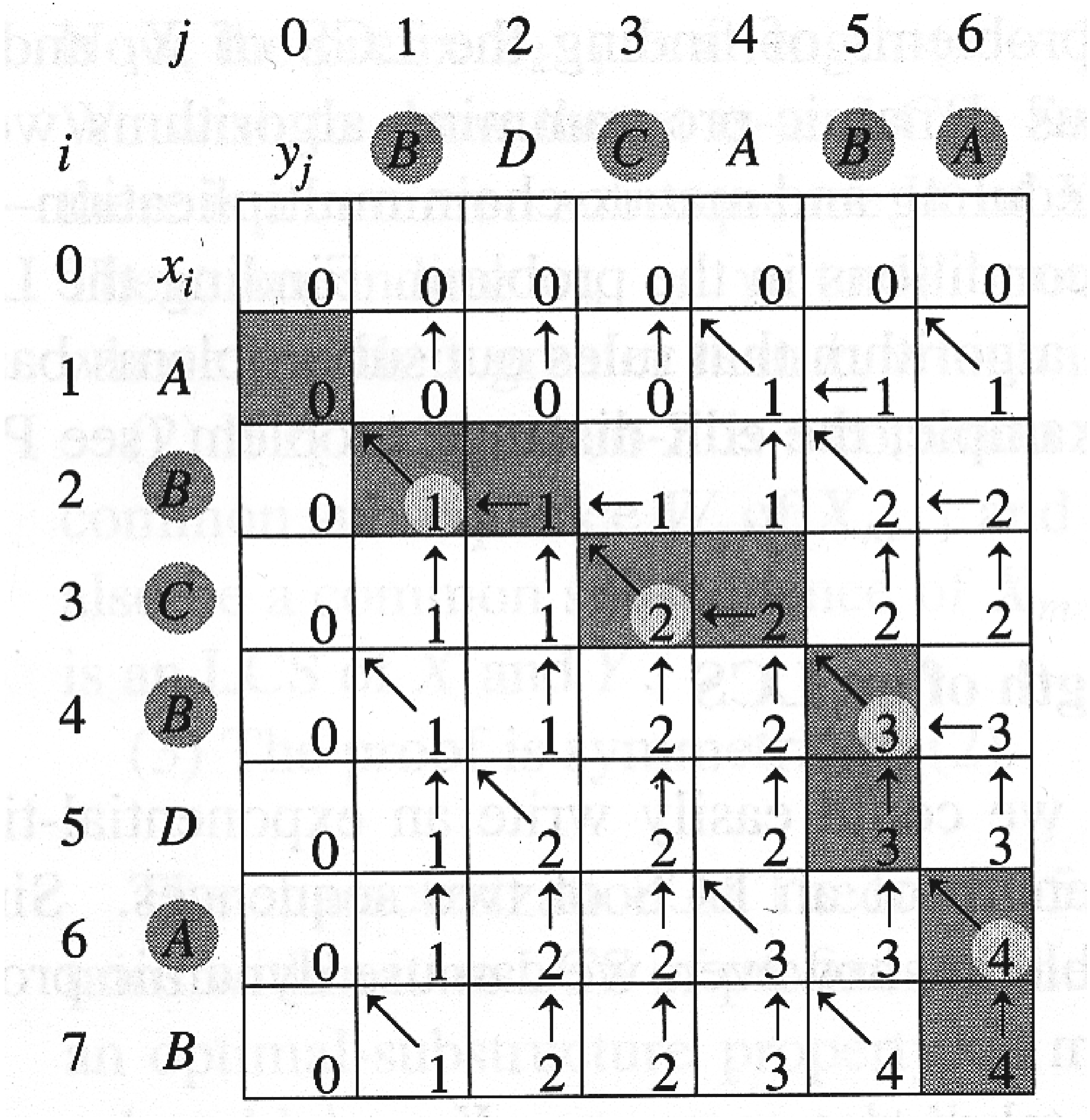
- Table c[0..m, 0..n] stores the length of an LCS of X_i and Y_j c[i,j]
- Table b[1..m, 1..n] stores pointers to optimal subproblem solutions

X

Y

			C	B	R	F	T	S	Q
		0	1	2	3	4	5	6	7
	0								
A	1								
B	2								
D	3								
F	4								
M	5								
Q	6								

Tables 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

```
LCS-LENGTH(X,Y)
m ← length[X]
n ← length[Y]
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
```

PRINT-LCS

```
PRINT-LCS (b, X, i, 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)
              else PRINT-LCS(b,X,i,j-1)
```

Code Complexity

- Time complexity?
  - O(mn) to compute tables b and c
  - O(m + n) to print out the LCS
- Space complexity
  - O(mn) for tables b and c
- Improved space complexity?
  - We can eliminate table b, but still need table c
  - Print-LCS can be improved to use only two rows of table C at a time to compute the *length* of an LCS
  - But not if we need to reconstruct the *sequence*

LCS & Matrix-chain multiplication

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

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

The recurrences for matrix-chain-order parenthesization and longest common subsequence are very similar; no wonder they can be solved in very similar ways.

Conclusion

Dynamic Programming :

- General approach – combine solutions to subproblems to get the solution to a problem
- Unlike Divide and Conquer in that
  - subproblems are dependent rather than independent
  - bottom-up approach
  - save the values of subproblems in a table and use them more than one time

Conclusion

- Start with the smallest, simplest subproblems
- Combine “appropriate” subproblem solutions to get a solution to the bigger problem
- If you have a Divide and Conquer algorithm that does a lot of duplicate computation, you can often find a Dynamic Programming solution that is more efficient.
- Many possible applications; we discussed only a few.