Lecture 18.
Dynamic Programming

Introduction to Algorithms

Da Nang University of Science and Technology

Dang Thien Binh

dtbinh@dut.udn.vn

Dynamic Programming

- Dynamic programming solves *optimization problems* by combining solutions to subproblems
- "Programming" refers to a tabular method with a series of choices, not "coding"
- Recall the divide-and-conquer approach
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine solutions of subproblems
- Dynamic programming is applicable when *subproblems are not independent*, i.e., subproblems share subsubproblems
  - Solve every subsubproblem only once and store the answer for use when it reappears
  - A divide-and-conquer approach will do more work than necessary

Dynamic Programming Solution

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

Matrix Multiplication (1/3)

- Matrix multiplication
  - Two matrices A and B can be multiplied
  - The number of columns of A must equal the number of rows of B
    - $A(p\times q)$ x $B(q\times r)$ → $C(p\times r)$
    - The number of scalar multiplications is $p\times q\times r$
  - For a $p\times q$ matrix A and a $q\times r$ matrix B, the product AB is a $p\times r$ matrix C

$$AB = C = [c_{i,j}]_{p\times r} \equiv \left[\sum_{k=1}^{q} a_{i,k} b_{k,j}\right]_{p\times r}$$

Matrix Multiplication (2/3)

$$\begin{bmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

Matrix Multiplication (3/3)

- Matrix multiplication

```
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] ← C[i, j] + A[i, k] · B[k, j]
         return C
```

Matrix-Chain Multiplication (1/2)

- Matrix multiplication
  - $A(p\times q)$ x $B(q\times r)$ → $C(p\times r)$
  - The number of scalar multiplications is $p\times q\times r$
- e.g. < $A_1(10\times100)$, $A_2(100\times5)$, $A_3(5\times50)$ >
  - $((A_1A_2)A_3)$ → 10x100x5 + 10x5x50 = 5000 + 2500 = 7,500
  - $(A_1(A_2A_3))$ → 100x5x50 + 10x100x50 = 25000 + 50000 = 75,000
  - 10 times faster
- Given a sequence (chain) <A_1, A_2,..., A_n> of n matrices to be multiplied, where $i = 1,2,\ldots,n$, matrix A_i has dimension $p_{i-1}\times p_i$, fully parenthesize the product $A_1A_2 \ldots An$ in a way that minimizes the number of scalar multiplications
  - Determine an order for multiplying matrices that has the lowest cost

Matrix-Chain Multiplication (2/2)

- Determine an order for multiplying matrices that has the lowest cost
- Counting the number of parenthesizations
  - $A_1 A_2 \ldots A_k \,|\, A_{k+1} \ldots A_{n-1} A_n$
  - $P(n) = \begin{cases} 1 & if \quad n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & if \quad n \geq 2 \end{cases}$ $\quad \Omega(2^n)$
  - Exercise 15.2-3 on page 338
  - Impractical to check all possible parenthesizations

Step 1 (1/3)

- The structure of an optimal parenthesization
  - Notation: $A_{i..j}$
    - Result from evaluating $A_i A_{i+1} \dots A_j$ $(i < j)$
  - Any parenthesization of $A_i A_{i+1} \dots A_j$ must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$
  - The cost of this parenthesization
    - cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ together

Step 1 (2/3)

- Suppose that an optimal parenthesization of $A_i A_{i+1} \dots A_j$ splits the product between A_k and A_{k+1}
  - The parenthesization of the prefix sub-chain $A_i A_{i+1} \dots A_k$ must be an optimal parenthesization of $A_i A_{i+1} \dots A_j$
  - The parenthesization of the postfix sub-chain $A_{k+1} A_{i+1} \dots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \dots A_j$
- That is, the optimal solution to the problem contains within it the optimal solution to subproblems

Step 1 (3/3)

Minimal
$Cost\_A_{1..6}$ + $Cost\_A_{7..9}$+$p_0p_6p_9$

$A_1A_2A_3A_4A_5A_6A_7A_8A_9$

Suppose $((A_1A_2)(A_3((A_4A_5)A_6)))$ $((A_7A_8)A_9)$ is optimal

Then (A_1A_2) $(A_3((A_4A_5)A_6))$ must be optimal for $A_1A_2A_3A_4A_5A_6$

Otherwise, if $(A_1(A_2A_3))$ $((A_4A_5)A_6)$ is optimal for $A_1A_2A_3A_4A_5A_6$

Then $((A_1(A_2A_3))((A_4A_5)A_6))$ $((A_7A_8)A_9)$ will be better than

$((A_1A_2)(A_3((A_4A_5)A_6)))$ $((A_7A_8)A_9)$

Contradiction!

Step 2

- A Recursive Solution
  - Subproblem
    - Determine the minimum cost of a parenthesization of $A_i A_{i+1} \dots A_j$ $(1 \le i \le j \le n)$
  - $m[i,j]$: the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$
  - $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
  - However, we do not know the value of k $(s[i,j])$, so we have to try all j-i possibilities

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

  - A recursive solution takes exponential time

Step 3 (1/3)

- Computing the optimal costs
  - How much subproblems in total?
    - One for each choice of i and j satisfying $1 \leq i \leq j \leq n$
    - $\Theta(n^2)$
  - MATRIX-CHAIN-ORDER(p)
    - Input: a sequence $p = < p_0, p_1, p_2, \dots, p_n >$ (length[p] $= n + 1$)
    - Try to fill in the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length
    - Lines 4-12: compute $m[i, i+1]$, $m[i, i+2]$, … each time

Step 3 (2/3)

- e.g. < $A_1(8\times3)$, $A_2(3\times5)$, $A_3(5\times10)$ >
  - m_{12} = 8x3x5 = 120
  - m_{23} = 3x5x10 = 150
  - m_{13} = 390
    - $m_{11} + m_{23} + p_0p_1p_3$ = 0 + 150 + 8x3x10 = 390
    - $m_{12} + m_{33} + p_0p_2p_3$ = 120 + 0 + 8x5x10 = 520

m (j, i)

1	2	3	
0	120	390	1
	0	150	2
		0	3

s (j, i)

2	3	
1	1	1
	2	2

Step 3 (3/3)

$O(n^3)$, $\Omega(n^3) \Rightarrow \Theta(n^3)$ running time

$\Theta(n^2)$ space

```
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
```

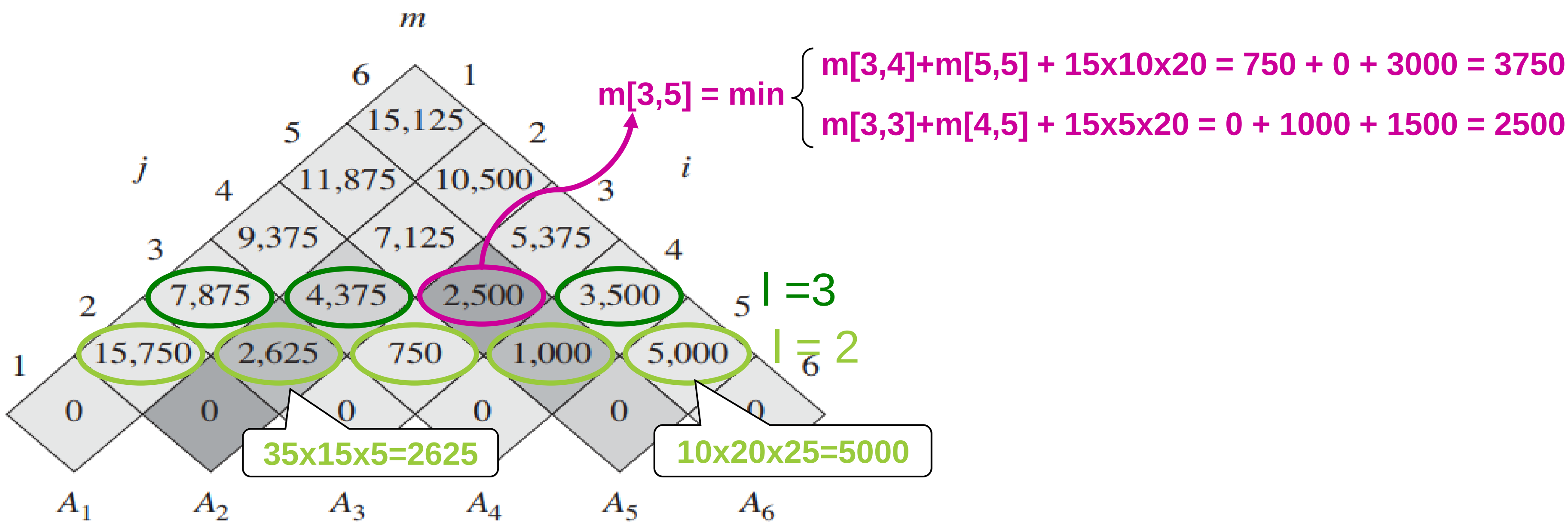


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. 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 10 when computing

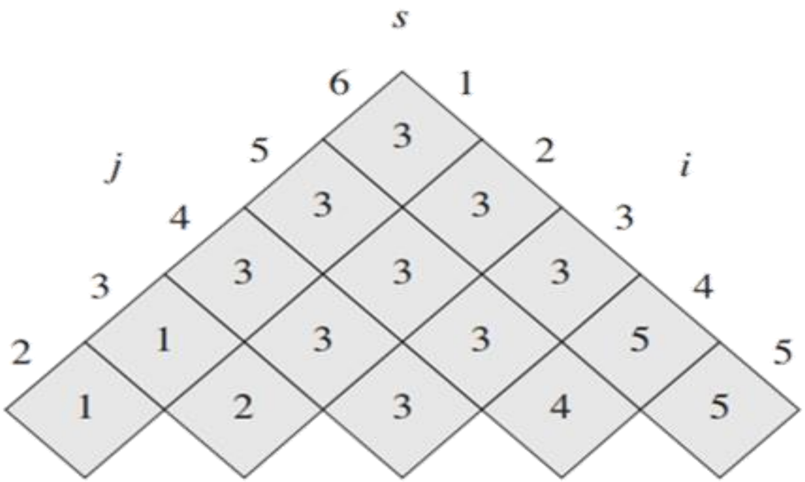
$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
$$= 7125.$$

Step 4 (1/2)

- Constructing an optimal solution
  - Each entry $s[i,j]$ records the value of k such that the optimal parenthesization of $A_i A_{i+1} \dots A_j$ splits the product between A_k and A_{k+1}
  - $A_{1..n} \Rightarrow A_{1..s[1..n]} A_{s[1..n]+1..n}$
  - $A_{1..s[1..n]} \Rightarrow A_{1..s[1,\, s[1..n]]} A_{s[1,\, s[1..n]]+1..s[1..n]}$
  - Recursive…
    - PRINT-OPTIMAL-PARENS(s, i, j)

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

Step 4 (2/2)

- Constructing an optimal solution
  - 

  - $((A_1\ (A_2\ \ A_3))((A_4\ \ A_5)\ A_6))$

Elements of Dynamic Programming (1/2)

- Optimal substructure
  - If an optimal solution contains within it optimal solutions to subproblems
  - Build an optimal solution from optimal solutions to subproblems
- Example
  - Matrix-chain multiplication
    - An optimal parenthesization of $A_i A_{i+1} \dots A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problem of parenthesizing $A_i A_{i+1} \dots A_k$ and $A_{k+1} A_{k+2} \dots A_j$

Elements of Dynamic Programming (2/2)

- 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
  - Typically, the total number of distinct subproblems is a polynomial in the input size
- Divide-and-Conquer is suitable usually generate brand-new problems at each step of the recursion

Characteristics of Optimal Substructure

- How many subproblems are used in an optimal solution to the original problem?
  - Matrix-Chain scheduling
    - 2 $(A_1A_2 \dots A_k$ and $A_{k+1}A_{k+2} \dots A_j)$
- How many choices we have in determining which subproblems to use in an optimal solution?
  - Matrix-chain scheduling: $j - i$ (choice for k)
- Informally, the running time of a dynamic-programming algorithm s on: the number of subproblems overall and how many choices we look at for each subproblem
  - Matrix-Chain scheduling: $\Theta(n^2) * O(n) = O(n^3)$

Overlapping Subproblems (1/2)

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j - 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
        + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
        + p_{i-1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

Overlapping Subproblems (2/2)

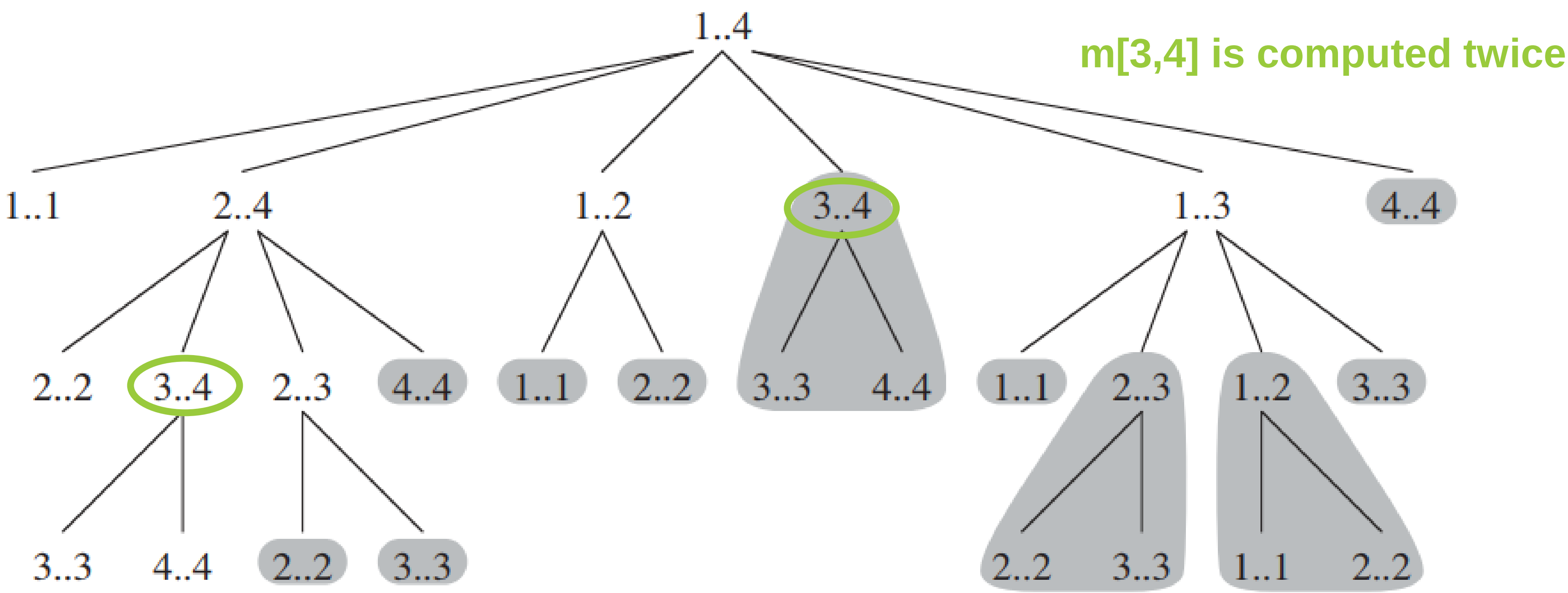


Figure 15.7 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node contains the parameters i and j. The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN.

Recursive Procedure for Matrix-Chain Multiplication

- The time to compute $m[1, n]$ is at least exponential in n
  - $T(1) \geq 1$

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

    $$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n$$
  - Prove $T(n) = \Omega(2^n)$ using the substitution method
    - Show that $T(n) \geq 2^{n-1}$
    - $T(n) \geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n = 2 \sum_{i=0}^{n-2} 2^i + n$

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

Memoization

- A variation of dynamic programming that often offers the efficiency of the usual dynamic programming approach while maintaining a top-down strategy
  - Memoize the natural, but inefficient, recursive algorithm
  - Maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm
- Memoization for matrix-chain multiplication
  - Calls in which $m[i,j] = \infty \Rightarrow \Theta(n^2)$ calls
  - Calls in which $m[i,j] < \infty \Rightarrow O(n^3)$ calls
  - Turns an $\Omega(2^n)$-time into an $O(n^3)$-time algorithm

Memoization

MEMOIZED-MATRIX-CHAIN(p)

```
n ← length[p] − 1
for i ← 1 to n
     do for j ← i to n
          do m[i, j] ← ∞
return LOOKUP-CHAIN(p, 1, n)
```

Lookup-Chain(p, i, j)

```
LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j − 1
        q = LOOKUP-CHAIN(m, p, i, k)
            + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

Lookup-Chain(p, i, j)

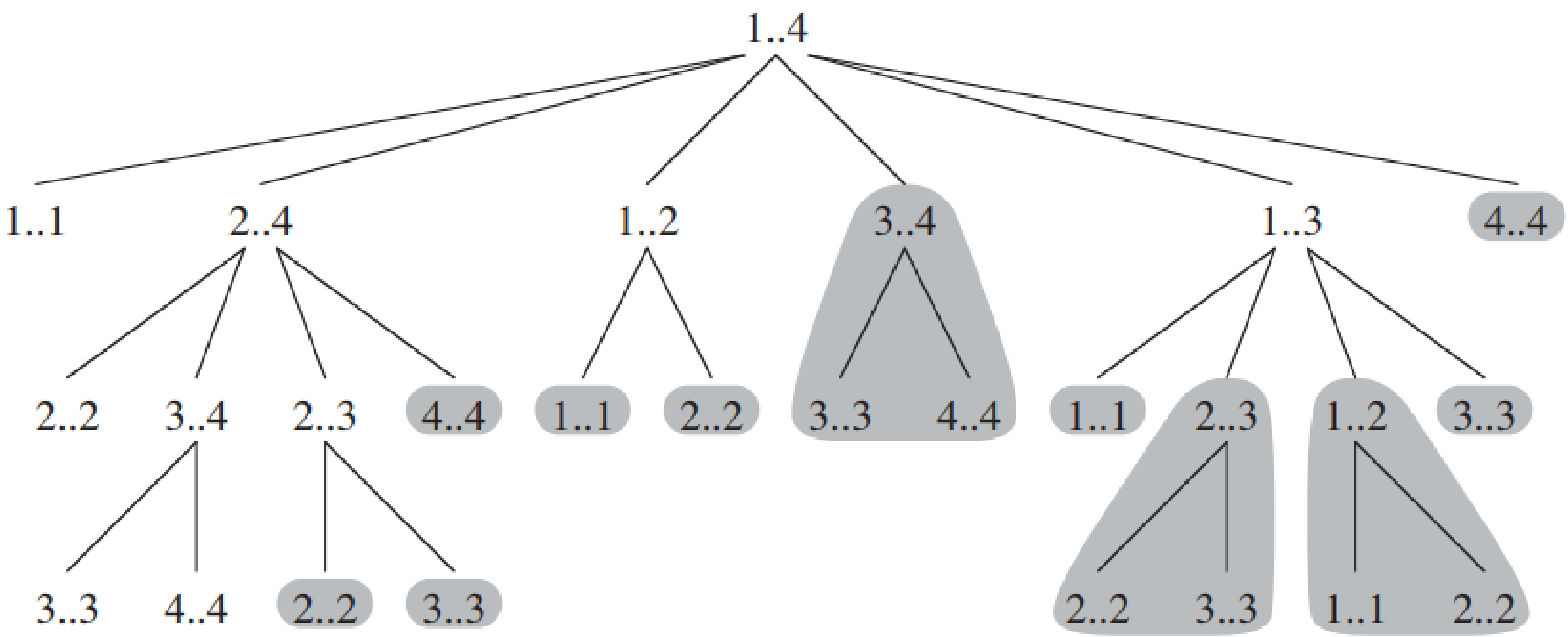


Figure 15.7 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN($p, 1, 4$). Each node contains the parameters i and j. The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN.

Dynamic Programming vs. Memoization

- If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor
  - No overhead for recursion and less overhead for maintaining table
  - There are some problems for which the regular pattern of table accesses in the dynamic programming algorithm can be exploited to reduce the time or space requirements even further

Self-Study

- Two more dynamic-programming problems
  - Section 15.4 Longest Common Subsequence
  - Section 15.5 Optimal Binary Search Trees

Longest Common Subsequence (LCS)

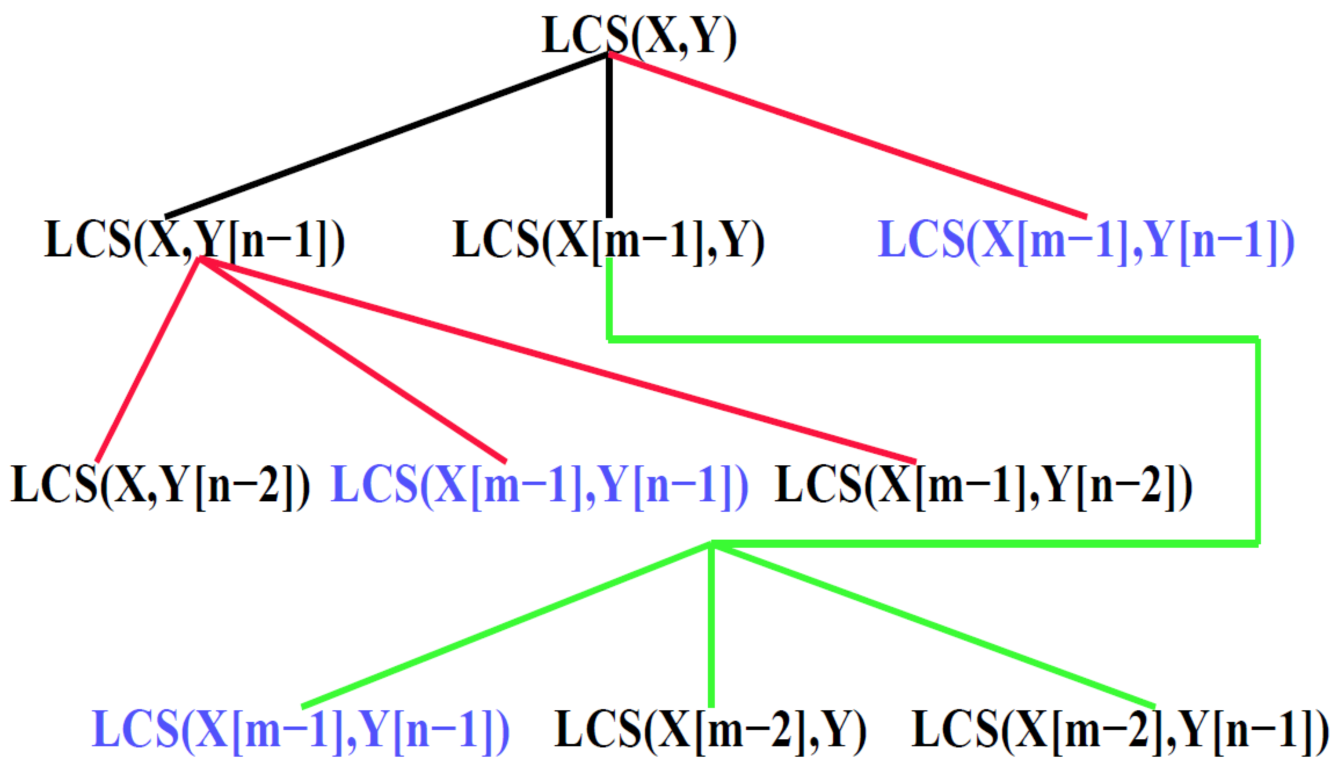
- Problem: Given two sequences X=<$x_1, \dots, x_m$> and Y=<$y_1, \dots, y_n$>, find the longest subsequence Z=<$z_1, \dots, z_k$> that is common to X and Y
  - A subsequence is a subset of elements from the sequence with **strictly increasing** order (not necessarily contiguous)
  - There are 2^m subsequences of X → checking all subsequences is impractical for long sequences
- Example: X=<A, B, C, B, D, A, B> and Y=<B, D, C, A, B, A>
  - Common subsequences: <A>; <B>; <C>; <D>; <A, A>; <B, B>; <B, C, A>; <B, C, B>; <C, B, A>; etc.
  - The longest common subsequences: <B, C, B, A>; <B, D, A, B>

Step 1: Optimal Structure of an LCS

- Let X=<$x_1, \ldots, x_m$> and Y=<$y_1, \ldots, y_n$> be sequences, and let Z=<$z_1, \ldots, z_k$> be any LCS of X and Y
  - If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
  - If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y
  - If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

Step 2: Recursive Solution (1/2)

- Overlapping-subproblems



Step 2: Recursive Solution (2/2)

- Define $c[i, j]$ = length of LCS for X_i and Y_j

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

Step 3: Computing the length of an LCS

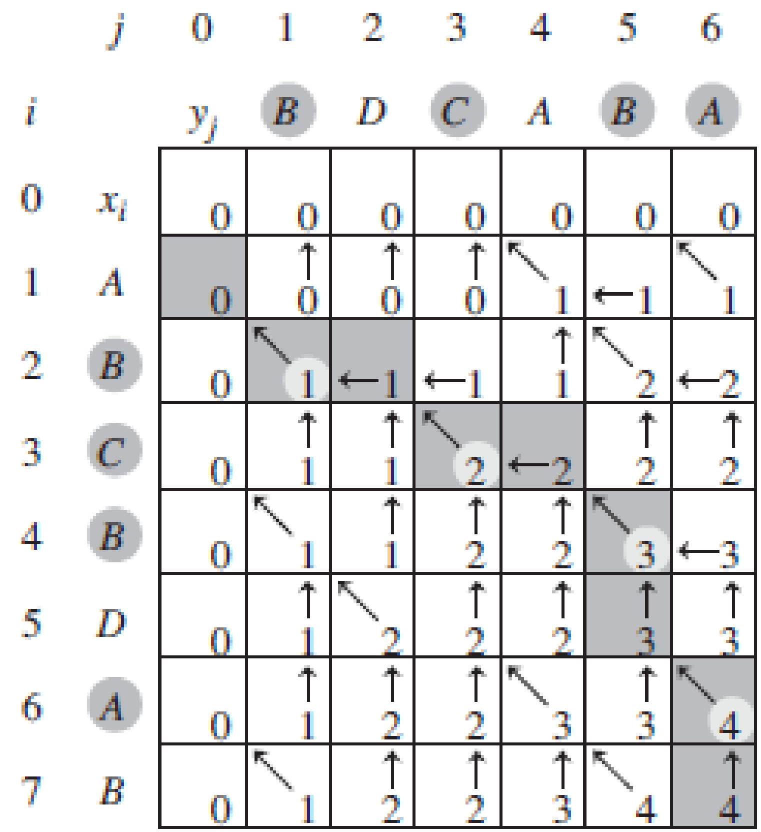
```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            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
```

$b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$

LCS-LENGTH(X, Y) is $\Theta(mn)$

Step 4: Constructing an LCS

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



PRINT-LCS(b, X, i, j) is $O(m + n)$

Thanks to contributors

Mr. Phuoc-Nguyen Bui (2022)

Dr. Thien-Binh Dang (2017 – 2022)

Prof. Hyunseung Choo (2001 – 2022)