CS473-Algorithms I

Lecture 10

Dynamic Programming

Introduction

- An algorithm design paradigm like divide-and-conquer
- "Programming": A tabular method (not writing computer code)
- Divide-and-Conquer (DAC): subproblems are independent
- Dynamic Programming (DP): subproblems are not independent
- Overlapping subproblems: subproblems share sub-subproblems
  - In solving problems with overlapping subproblems
    - A DAC algorithm does redundant work
      - Repeatedly solves common subproblems
    - A DP algorithm solves each problem just once
      - Saves its result in a table

Optimization Problems

- DP typically applied to optimization problems
- In an optimization problem
  - There are many possible solutions (feasible solutions)
  - Each solution has a value
  - Want to find an optimal solution to the problem
    - A solution with the optimal value (min or max value)
  - Wrong to say "the" optimal solution to the problem
    - There may be several solutions with the same optimal value

Development of a DP 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 the information computed in Step 3

Example: Matrix-chain Multiplication

- Input: a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices
- Aim: compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$
- A product of matrices is fully parenthesized if
  - It is either a single matrix
  - Or, the product of two fully parenthesized matrix products surrounded by a pair of parentheses.
    - $(A_i(A_{i+1}A_{i+2} \ldots A_j))$
    - $((A_iA_{i+1}A_{i+2} \ldots A_{j-1})A_j)$
    - $((A_iA_{i+1}A_{i+2} \ldots A_k)(A_{k+1}A_{k+2} \ldots A_j))$ for $i \leq k < j$
  - All parenthesizations yield the same product; matrix product is associative

Matrix-chain Multiplication: An Example Parenthesization

- Input: $\langle A_1, A_2, A_3, A_4 \rangle$
- 5 distinct ways of full parenthesization

$(A_1(A_2(A_3A_4)))$

$(A_1((A_2A_3)A_4))$

$((A_1A_2)(A_3A_4))$

$((A_1(A_2A_3))A_4)$

$(((A_1A_2)A_3)A_4)$

- The way we parenthesize a chain of matrices can have a dramatic effect on the cost of computing the product

Cost of Multiplying two Matrices

Matrix has two attributes

- rows[A]: # of rows
- cols[A]: # of columns

# of scalar mult-adds in C ← AB is rows[A] × cols[B] × cols[A]

A: (p × q)

B: (q × r)

C=A·B is p × r.

# of mult-adds is p × r × q

MATRIX-MULTIPLY(A, B)

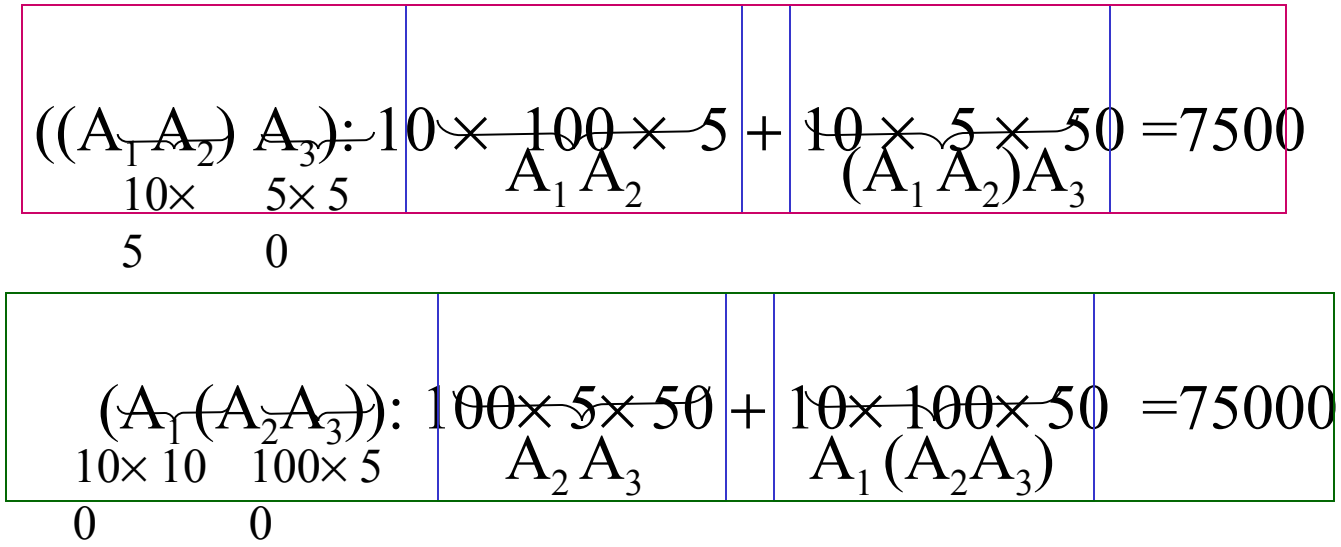
```
if cols[A] ≠ rows[B] then
    error("incompatible dimensions")
for i ← 1 to rows[A] do
    for j ← 1 to cols[B] do
        C[i,j] ← 0
        for k ← 1 to cols[A] do
            C[i,j] ← C[i,j] + A[i,k] · B[k,j]
return C
```

Matrix-chain Multiplication Problem

Input: a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, A_i is a $p_{i-1} \times p_i$ matrix

Aim: fully parenthesize the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ such that the number of scalar mult-adds are minimized.

- Ex.: $\langle A_1, A_2, A_3 \rangle$ where A_1: 10×100; A_2: 100×5; A_3: 5×50



$\Rightarrow$ First parenthesization yields 10 times faster computation.

Counting the Number of Parenthesizations

- Brute force approach: exhaustively check all parenthesizations
- P(n): # of parenthesizations of a sequence of n matrices
- We can split sequence between kth and (k+1)st matrices for any k=1, 2, … , n–1, then parenthesize the two resulting sequences independently, i.e.,

$$(A_1 A_2 A_3 \ldots A_k)(A_{k+1} A_{k+2} \ldots A_n)$$

- We obtain the recurrence

$$P(1) = 1 \text{ and } P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$$

Number of Parenthesizations: $\sum_{k=1}^{n-1} P(k)P(n-k)$

- The recurrence generates the sequence of Catalan Numbers
- Solution is $P(n) = C(n-1)$ where

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

- The number of solutions is exponential in n
- Therefore, brute force approach is a poor strategy

The Structure of an Optimal Parenthesization

Step 1: Characterize the structure of an optimal solution

- $A_{i..j}$: matrix that results from evaluating the product $A_i A_{i+1} A_{i+2} \ldots A_j$
- An optimal parenthesization of the product $A_1 A_2 \ldots A_n$
  - Splits the product between A_k and A_{k+1}, for some $1 \leq k < n$
    $(A_1 A_2 A_3 \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_n)$
  - i.e., first compute $A_{1..k}$ and $A_{k+1..n}$ and then multiply these two
- The cost of this optimal parenthesization
  Cost of computing $A_{1..k}$
  + Cost of computing $A_{k+1..n}$
  + Cost of multiplying $A_{1..k} \cdot A_{k+1..n}$

Step 1: Characterize the Structure of an Optimal Solution

- Key observation: given optimal parenthesization

  $(A_1A_2A_3 \ldots A_k) \cdot (A_{k+1}A_{k+2} \ldots A_n)$

  – Parenthesization of the subchain $A_1A_2A_3 \ldots A_k$

  – Parenthesization of the subchain $A_{k+1}A_{k+2} \ldots A_n$

  should both be optimal

  – Thus, optimal solution to an instance of the problem contains optimal solutions to subproblem instances

  – i.e., optimal substructure within an optimal solution exists.

The Structure of an Optimal Parenthesization

Step 2: Define the value of an optimal solution recursively in terms of optimal solutions to the subproblems

- Subproblem: The problem of determining the minimum cost of computing $A_{i..j}$, i.e., parenthesization of $A_i A_{i+1} A_{i+2} \ldots A_j$
- m_{ij}: min # of scalar mult-adds needed to compute subchain $A_{i..j}$
  - the value of an optimal solution is m_{1n}
  - $m_{ii} = 0$, since subchain $A_{i..i}$ contains just one matrix; no multiplication at all
  - $m_{ij} = ?$

Step 2: Define Value of an Optimal Soln Recursively(m_{ij}=?)

- For $i < j$, optimal parenthesization splits subchain $A_{i..j}$ as $A_{i..k}$ and $A_{k+1..j}$ where $i \leq k < j$

  optimal cost of computing $A_{i..k}$: m_{ik}

  + optimal cost of computing $A_{k+1..j}$: $m_{k+1,j}$

  + cost of multiplying $A_{i..k} A_{k+1..j}$: $p_{i-1} \times p_k \times p_j$

  ($A_{i..k}$ is a $p_{i-1} \times p_k$ matrix and $A_{k+1..j}$ is a $p_k \times p_j$ matrix)

  $$\Rightarrow m_{ij} = m_{ik} + m_{k+1,j} + p_{i-1} \times p_k \times p_j$$

- The equation assumes we know the value of k, but we do not

Step 2: Recursive Equation for m_{ij}

- $m_{ij} = m_{ik} + m_{k+1,j} + p_{i-1} \times p_k \times p_j$
  - We do not know k, but there are $j-i$ possible values for k; $k = i, i+1, i+2, \ldots, j-1$
  - Since optimal parenthesization must be one of these k values we need to check them all to find the best

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

Step 2: $m_{ij} = \text{MIN}\{m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j\}$

- The m_{ij} values give the costs of optimal solutions to subproblems
- In order to keep track of how to construct an optimal solution
  - Define S_{ij} to be the value of k which yields the optimal split of the subchain $A_{i..j}$

    That is, $S_{ij} = k$ such that

    $$m_{ij} = m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \quad \text{holds}$$

Computing the Optimal Cost (Matrix-Chain Multiplication)

An important observation:

- We have relatively few subproblems
  - one problem for each choice of i and j satisfying $1 \le i \le j \le n$
  - total $n + (n-1) + \ldots + 2 + 1 = \frac{1}{2} n(n+1) = \Theta(n^2)$ subproblems
- We can write a recursive algorithm based on recurrence.
- However, a recursive algorithm may encounter each subproblem many times in different branches of the recursion tree
- This property, overlapping subproblems, is the second important feature for applicability of dynamic programming

Computing the Optimal Cost (Matrix-Chain Multiplication)

Compute the value of an optimal solution in a bottom-up fashion

- matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$
- the input is a sequence $\langle p_0, p_1, \ldots, p_n \rangle$ where length$[p] = n + 1$

Procedure uses the following auxiliary tables:

- $m[1 \ldots n, 1 \ldots n]$: for storing the $m[i, j]$ costs
- $s[1 \ldots n, 1 \ldots n]$: records which index of k achieved the optimal cost in computing $m[i, j]$

Algorithm for Computing the 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
        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
```

Algorithm for Computing the Optimal Costs

- The algorithm first computes

  $m[i, i] \leftarrow 0$ for $i = 1, 2, \ldots, n$ min costs for all chains of length 1

- Then, for $\ell = 2, 3, \ldots, n$ computes

  $m[i, i+\ell-1]$ for $i = 1, \ldots, n-\ell+1$ min costs for all chains of length ℓ

- For each value of $\ell = 2, 3, \ldots, n$,

  $m[i, i+\ell-1]$ depends only on table entries $m[i, k]$ & $m[k+1, i+\ell-1]$

  ~~for $i \le k < i+\ell-1$, which are already computed~~

Algorithm for Computing the Optimal Costs

```
ℓ = 2
for i = 1 to n – 1
      m[i, i+1] = ∞
      for k = i to i do
            .
            .
```

compute $m[i, i+1]$
$\{m[1, 2], m[2, 3], \ldots, m[n-1, n]\}$

$(n-1)$ values

```
ℓ = 3
for i = 1 to n – 2
      m[i, i+2] = ∞
      for k = i to i+1 do
            .
            .
```

compute $m[i, i+2]$
$\{m[1, 3], m[2, 4], \ldots, m[n-2, n]\}$

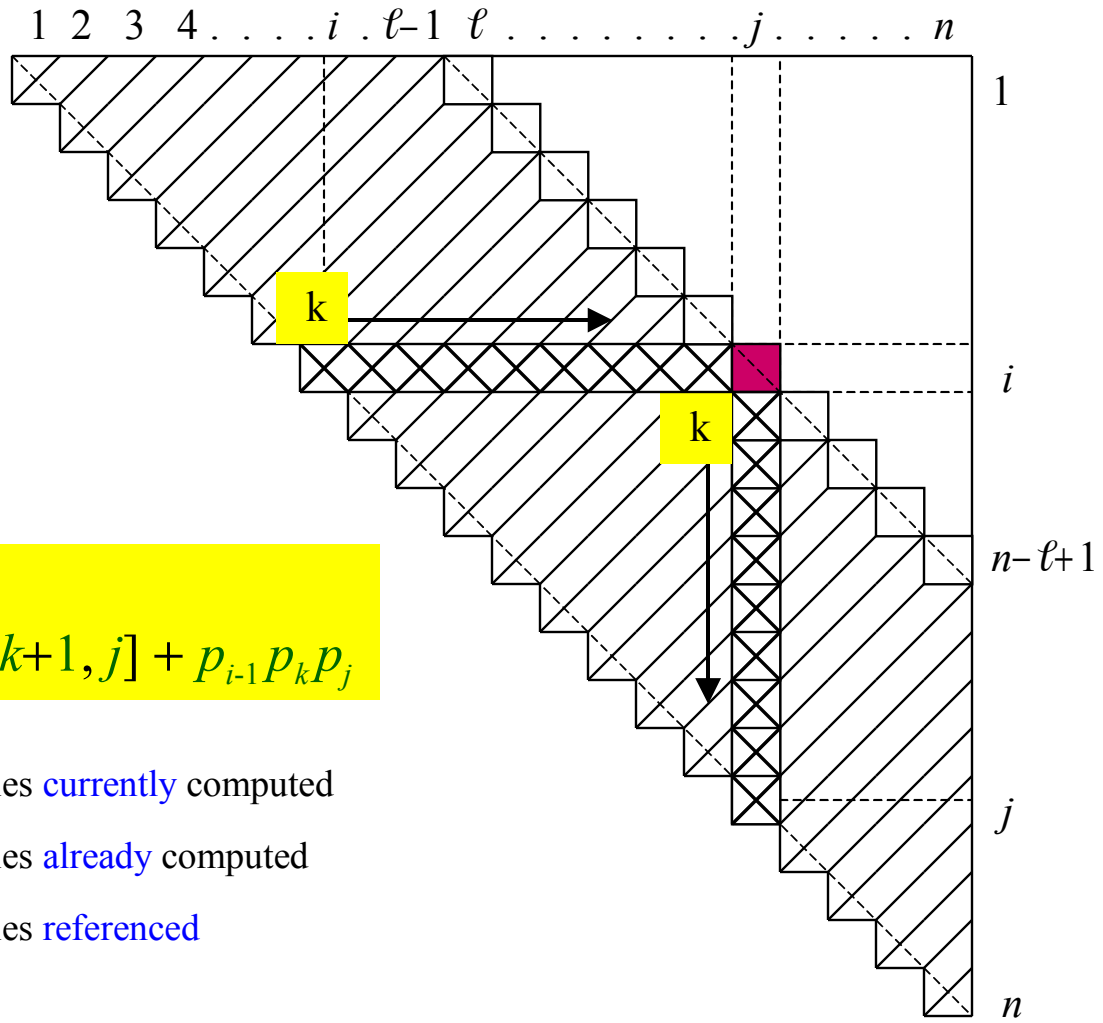
$(n-2)$ values

```
ℓ = 4
for i = 1 to n – 3
      m[i, i+3] = ∞
      for k = i to i+2 do
            .
            .
```

compute $m[i, i+3]$
$\{m[1, 4], m[2, 5], \ldots, m[n-3, n]\}$

$(n-3)$ values

Table access pattern in computing $m[i, j]$s for $\ell = j - i + 1$



for $k \leftarrow i$ to $j-1$ do

$q \leftarrow m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$

Table entries currently computed

Table entries already computed

Table entries referenced

Table access pattern in computing $m[i, j]$s for $\ell = j - i + 1$

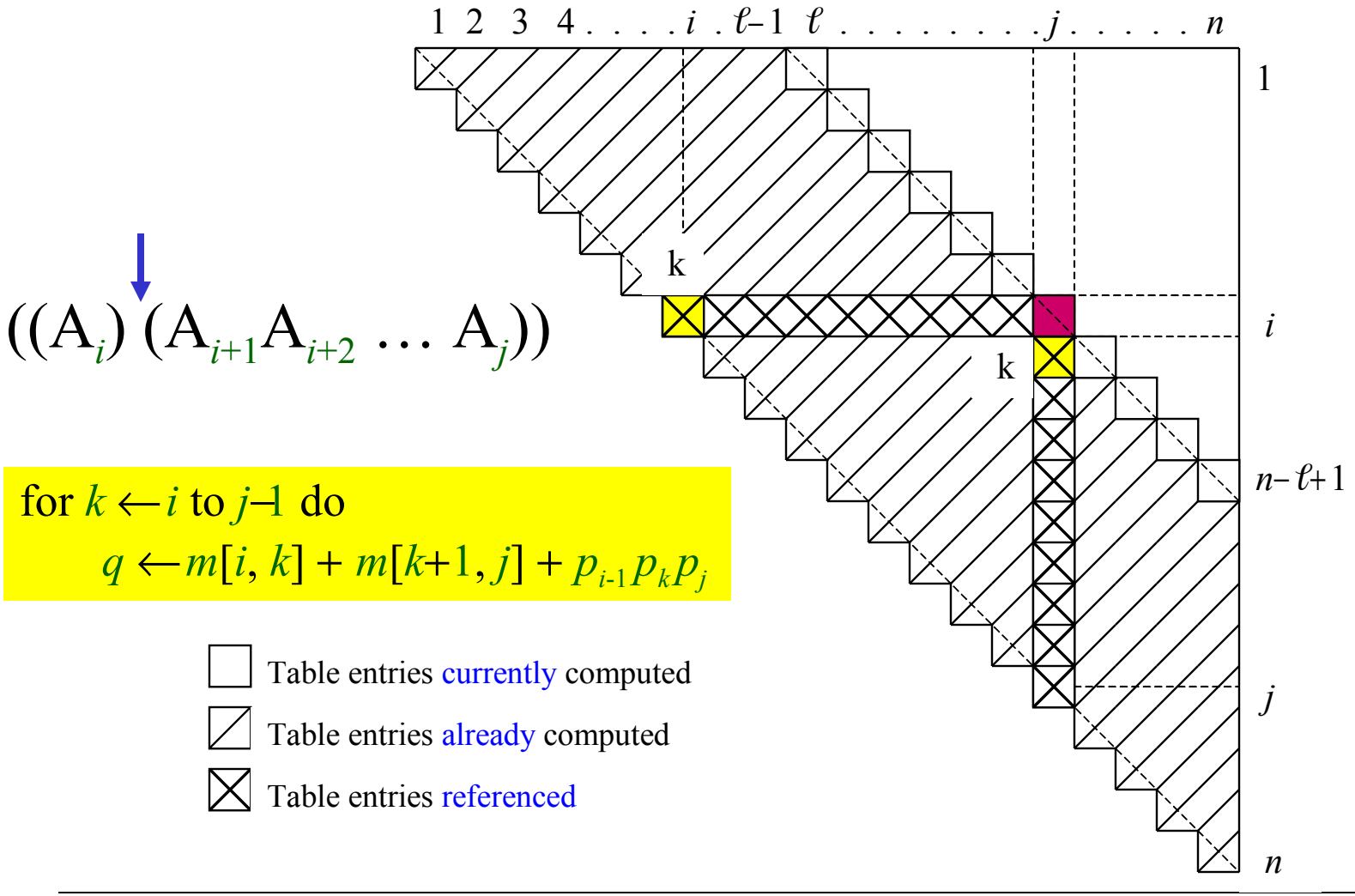


Table access pattern in computing $m[i, j]$s for $\ell = j - i + 1$

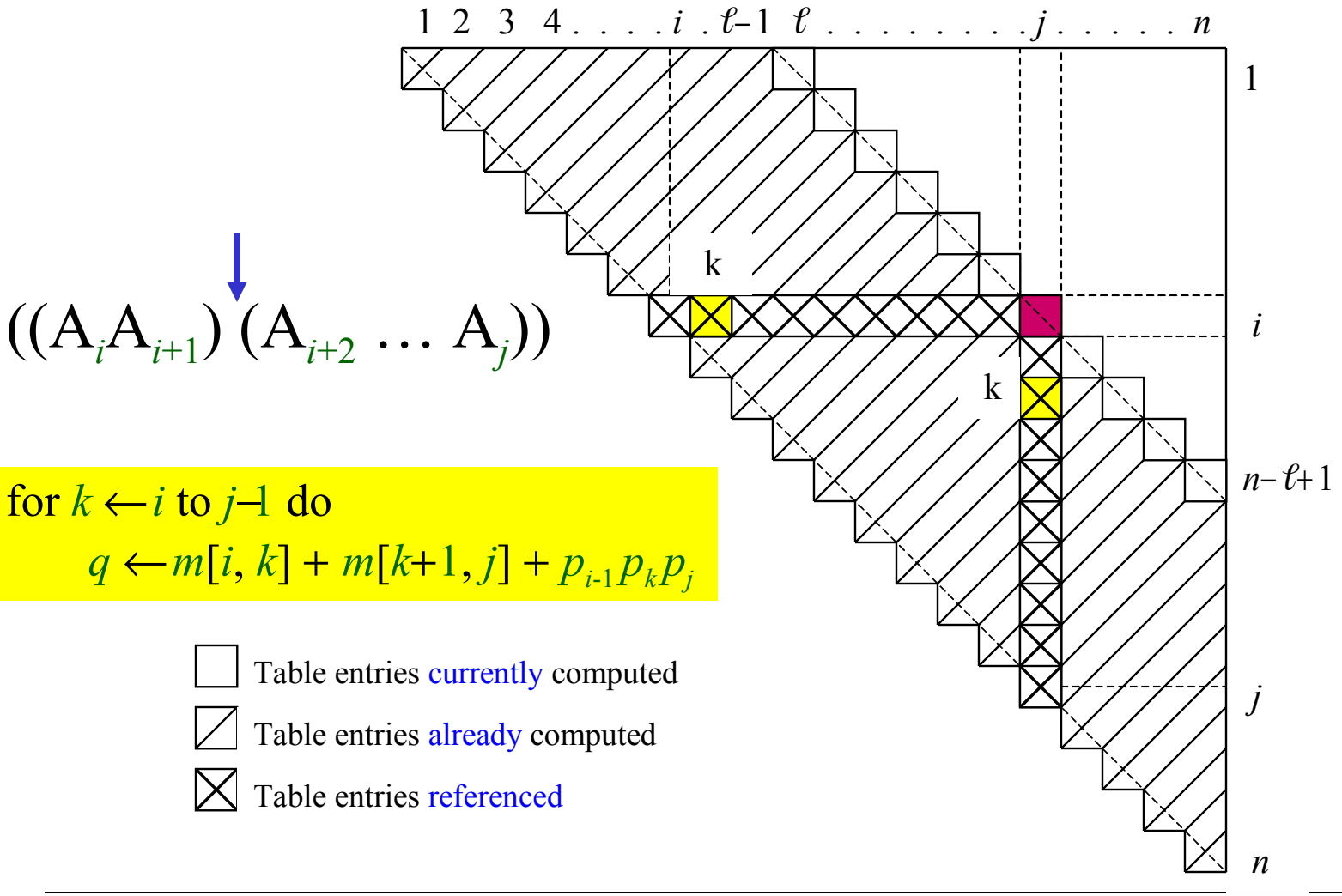


Table access pattern in computing $m[i, j]$s for $\ell = j - i + 1$

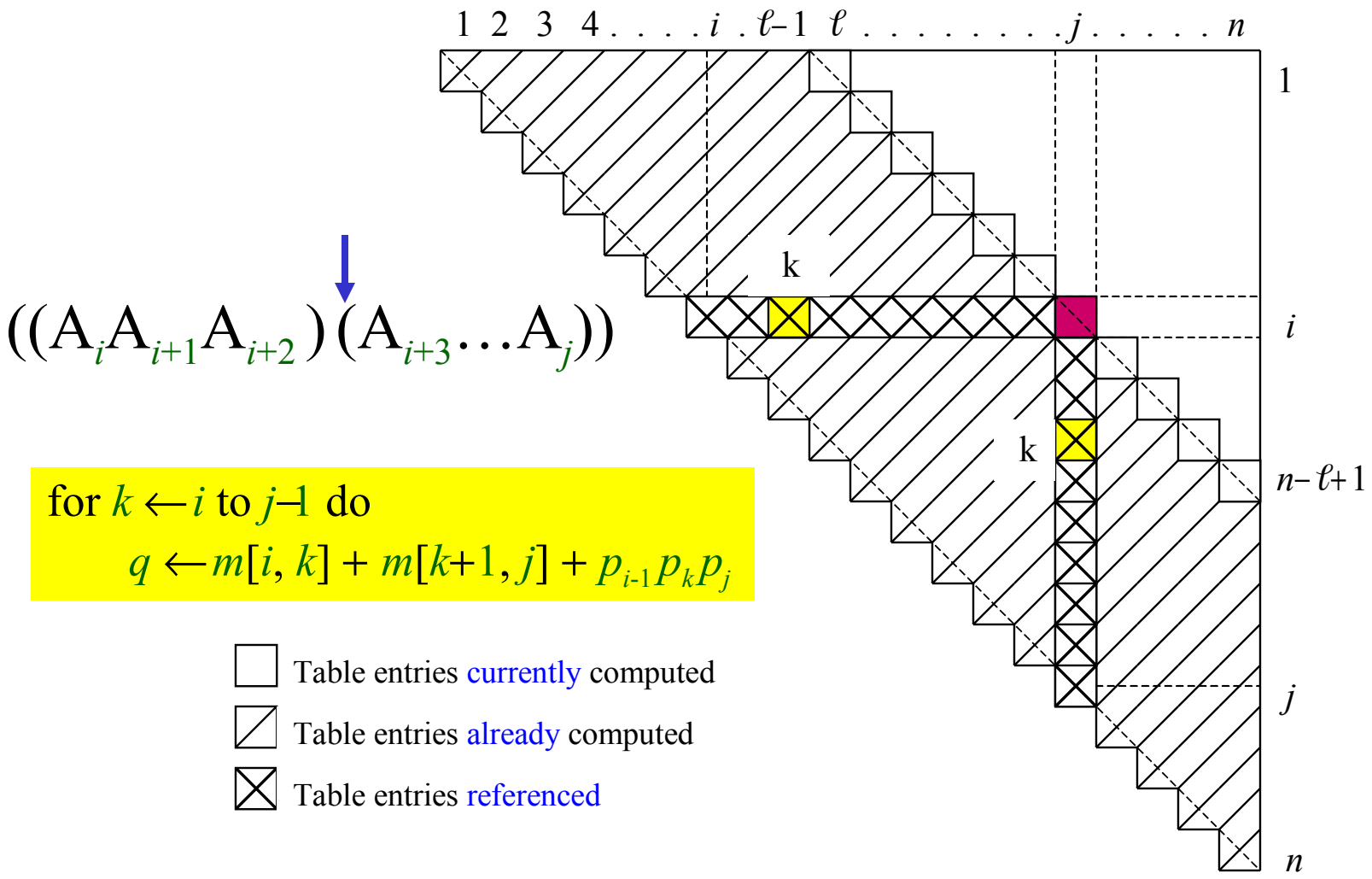


Table access pattern in computing $m[i, j]$s for $\ell = j - i + 1$

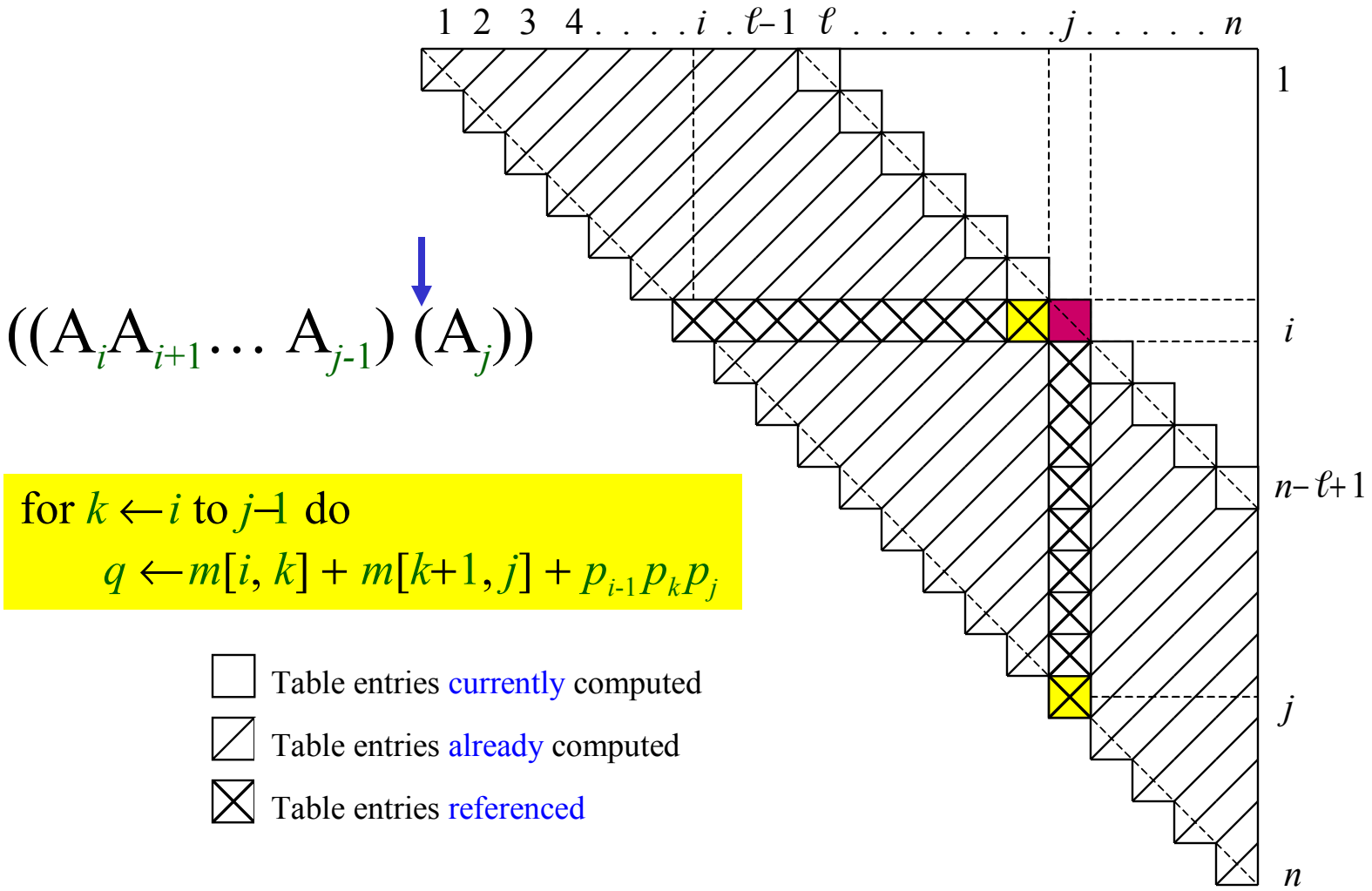
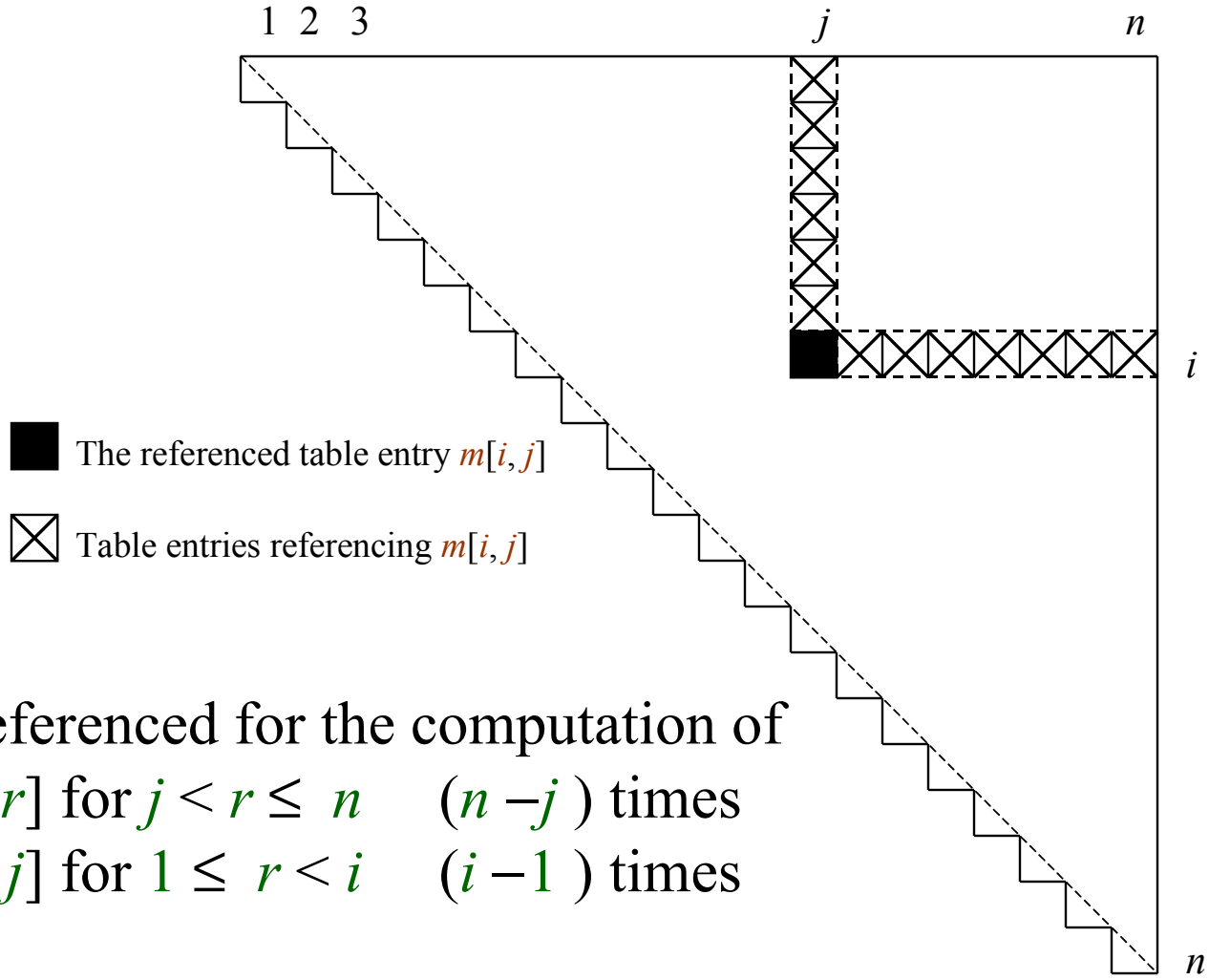


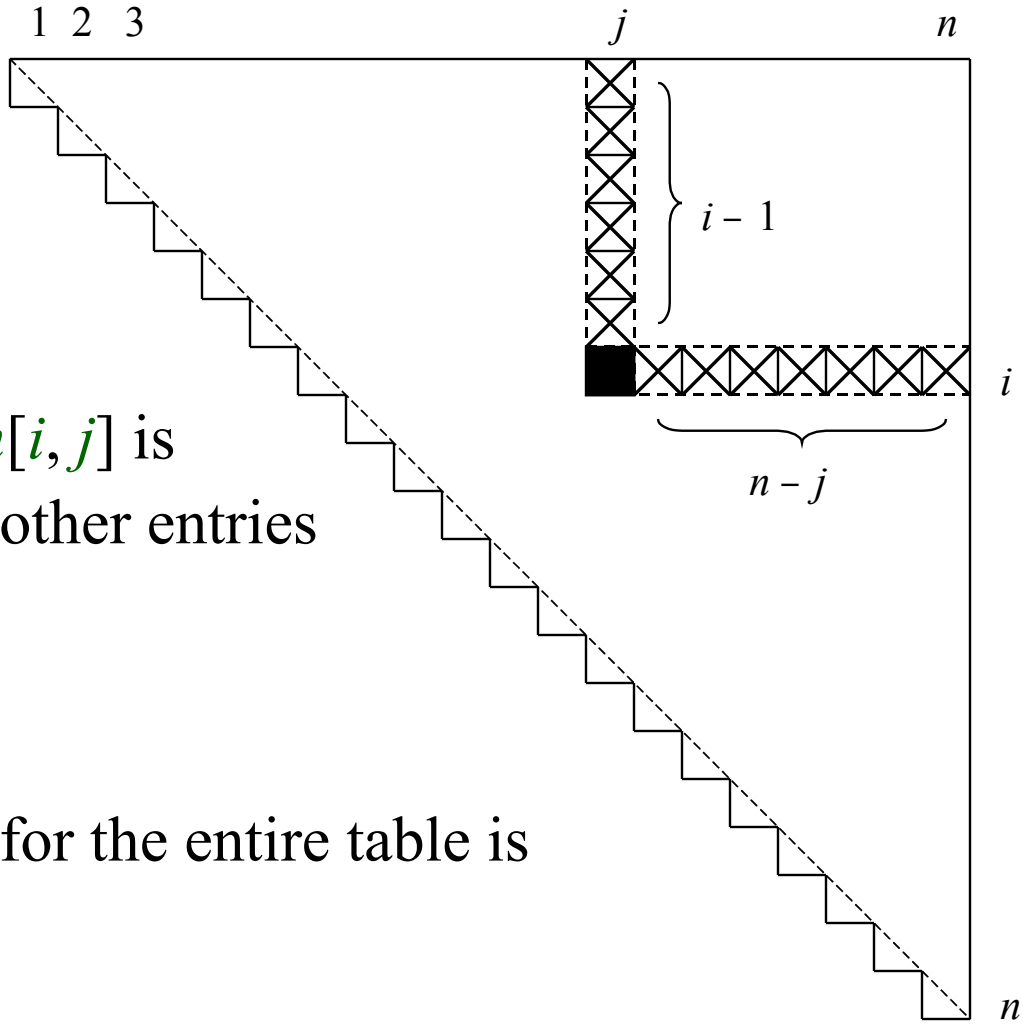
Table reference pattern for $m[i, j]$ $(1 \le i \le j \le n)$



$m[i, j]$ is referenced for the computation of

- $m[i, r]$ for $j < r \le n$ $(n - j)$ times
- $m[r, j]$ for $1 \le r < i$ $(i - 1)$ times

Table reference pattern for $m[i, j]$ $(1 \le i \le j \le n)$



$R(i, j)$ = # of times that $m[i, j]$ is referenced in computing other entries

$$R(i, j) = (n - j) + (i - 1)$$
$$= (n - 1) - (j - i)$$

The total # of references for the entire table is

$$\sum_{i=1}^{n} \sum_{j=i}^{n} R(i, j) \frac{n^3 - n}{3}$$

Constructing an Optimal Solution

- MATRIX-CHAIN-ORDER determines the optimal # of scalar mults/adds
  - needed to compute a matrix-chain product
  - it does not directly show how to multiply the matrices

- That is,
  - it determines the cost of the optimal solution(s)
  - it does not show how to obtain an optimal solution

- Each entry $s[i, j]$ records the value of k such that optimal parenthesization of $A_i \ldots A_j$ splits the product between A_k & A_{k+1}

- We know that the final matrix multiplication in computing $A_{1\ldots n}$ optimally is $A_{1\ldots s[1,n]} \times A_{s[1,n]+1,n}$

Constructing an Optimal Solution

Earlier optimal matrix multiplications can be computed recursively

Given:

- the chain of matrices $A = \langle A_1, A_2, \ldots A_n \rangle$
- the s table computed by MATRIX-CHAIN-ORDER

The following recursive procedure computes the matrix-chain product $A_{i \ldots j}$

```
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-MUTIPLY(X, Y)
    else
        return A_i
```

Invocation: MATRIX-CHAIN-MULTIPLY(A, s, 1, n)

Example: Recursive Construction of an Optimal Solution

$s[1\ldots6, 1\ldots6]$

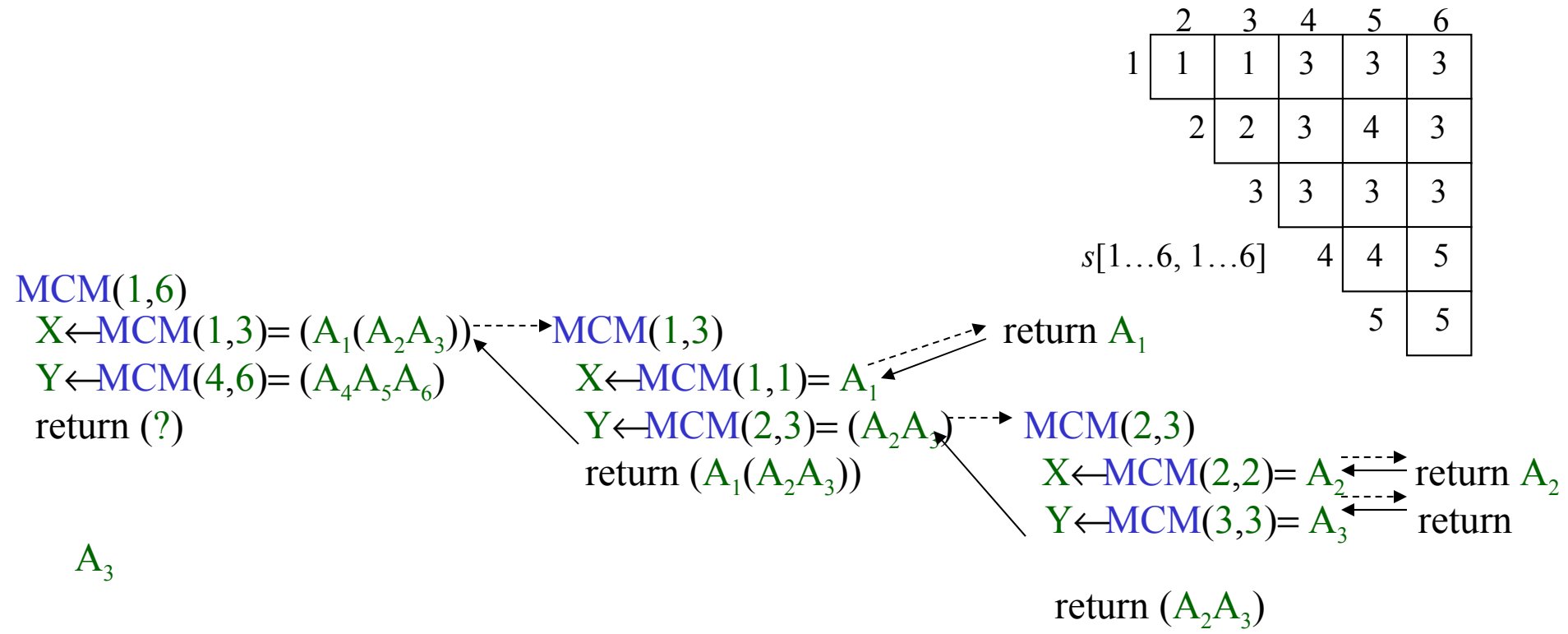
	2	3	4	5	6
1	1	1	3	3	3
2		2	3	4	3
3			3	3	3
4				4	5
5					5

MCM(1,6)
X←MCM(1,3)= ($A_1A_2A_3$)
Y←MCM(4,6)= ($A_4A_5A_6$)
return (?)

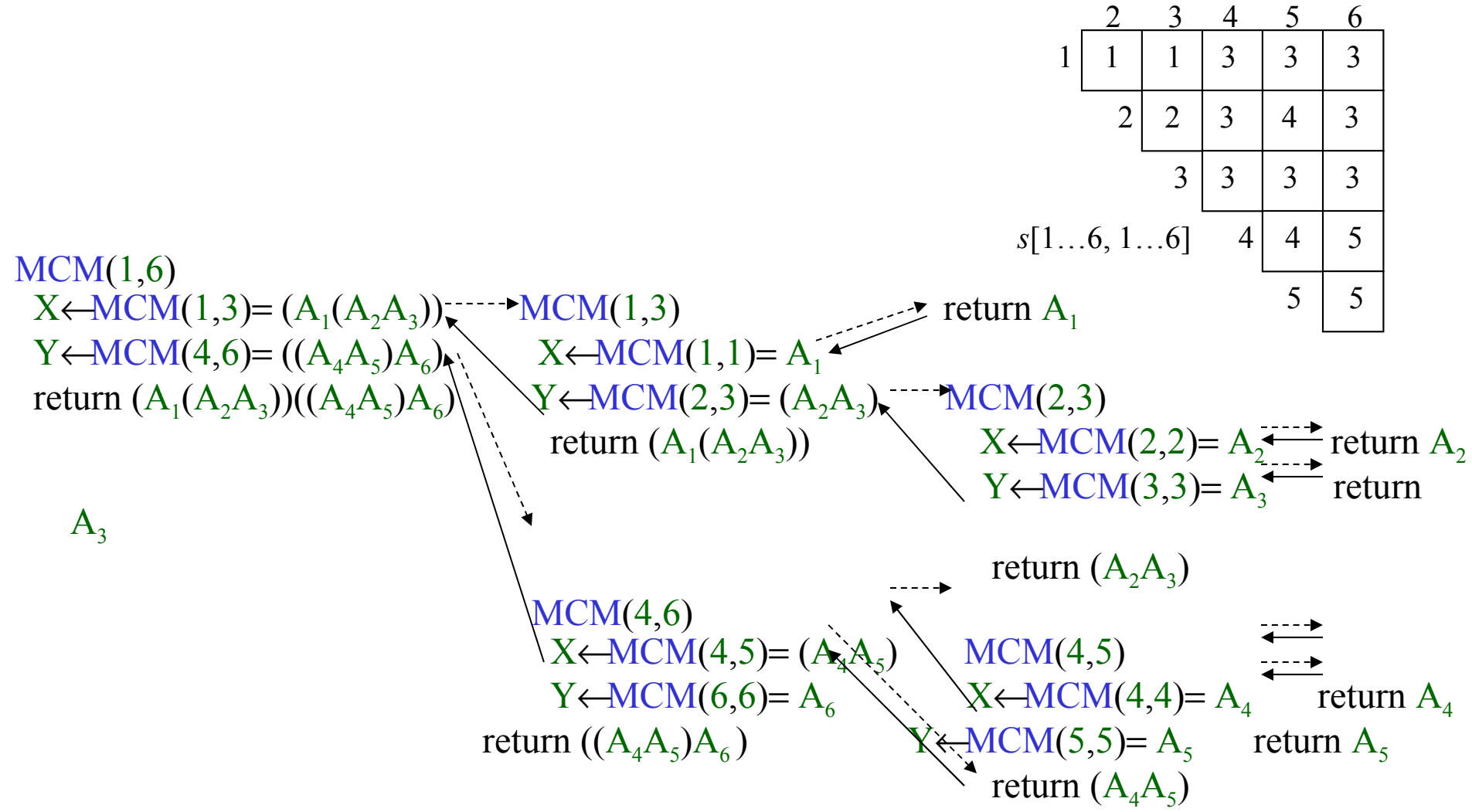
MCM(1,3)
X←MCM(1,1)= A_1
Y←MCM(2,3)= (A_2A_3)
return (?)

return A_1

Example: Recursive Construction of an Optimal Solution



Example: Recursive Construction of an Optimal Solution



Elements of Dynamic Programming

- When should we look for a DP solution to an optimization problem?
- Two key ingredients for the problem
  - Optimal substructure
  - Overlapping subproblems

DP Hallmark #1

Optimal Substructure

- A problem exhibits optimal substructure
  - if an optimal solution to a problem contains within it optimal solutions to subproblems
- Example: matrix-chain-multiplication

  Optimal parenthesization of $A_1 A_2 \ldots A_n$ that splits the product between A_k and A_{k+1},

  contains within it optimal soln's to the problems of parenthesizing $A_1 A_2 \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_n$

Optimal Substructure

- The optimal substructure of a problem often suggests a suitable space of subproblems to which DP can be applied
- Typically, there may be several classes of subproblems that might be considered natural
- Example: matrix-chain-multiplication
  - All subchains of the input chain

    We can choose an arbitrary sequence of matrices from the input chain
  - However, DP based on this space solves many more subproblems

Optimal Substructure

Finding a suitable space of subproblems

- Iterate on subproblem instances
- Example: matrix-chain-multiplication
  - Iterate and look at the structure of optimal soln's to subproblems, sub-subproblems, and so forth
  - Discover that all subproblems consists of subchains of $\langle A_1, A_2, \ldots, A_n \rangle$
  - Thus, the set of chains of the form

$$\langle A_i, A_{i+1}, \ldots, A_j \rangle \text{ for } 1 \le i \le j \le n$$

  - Makes a natural and reasonable space of subproblems

DP Hallmark #2

Overlapping Subproblems

- Total number of distinct subproblems should be polynomial in the input size
- When a recursive algorithm revisits the same problem over and over again

  we say that the optimization problem has overlapping subproblems

Overlapping Subproblems

- DP algorithms typically take advantage of overlapping subproblems
  - by solving each problem once
  - then storing the solutions in a table where it can be looked up when needed
  - using constant time per lookup

Overlapping Subproblems

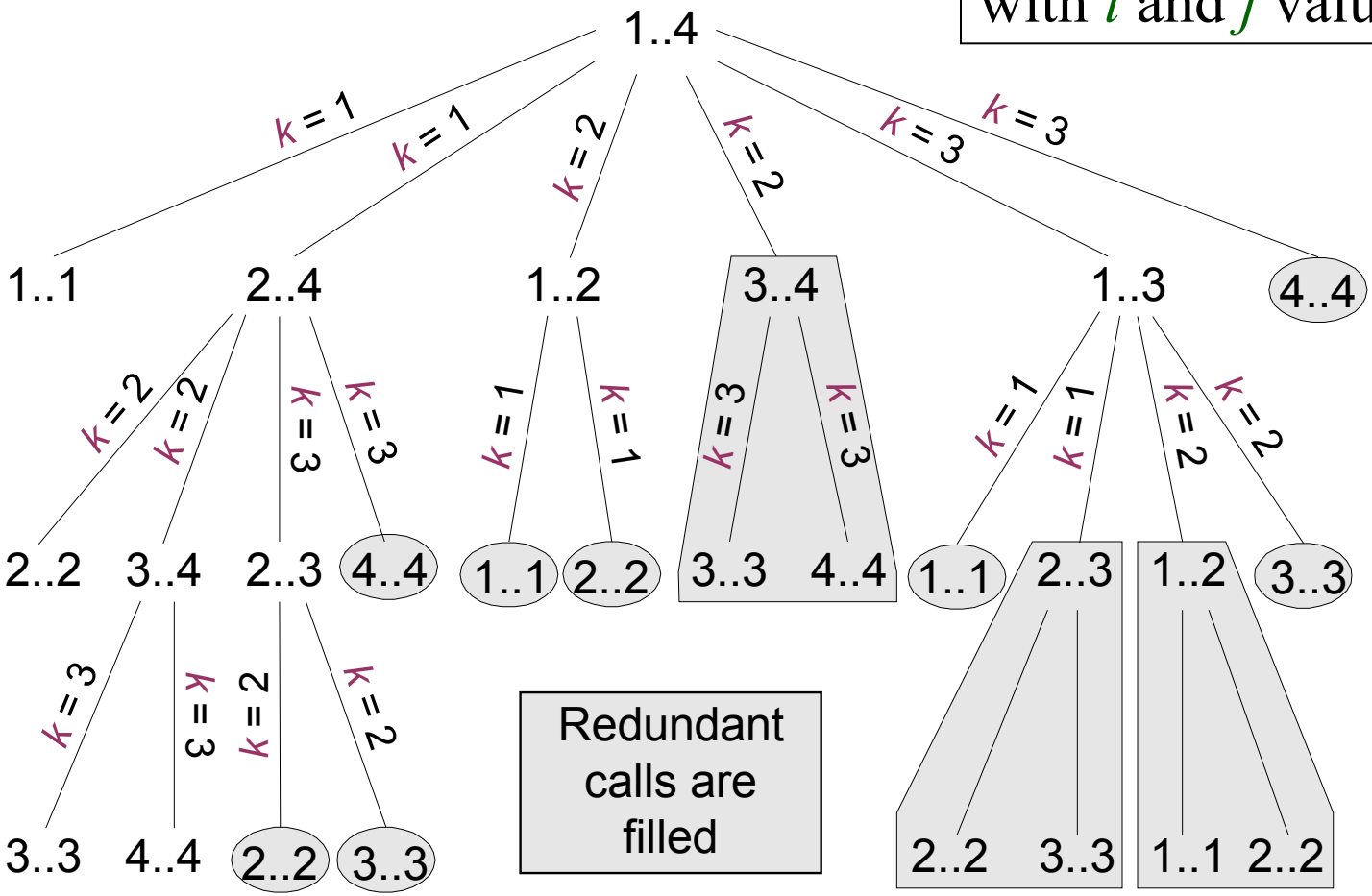
Recursive matrix-chain order

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

Recursive Matrix-chain Order

Recursion tree for RMC(p,1,4)

Nodes are labeled with i and j values



Running Time of RMC

$$T(1) \geq 1$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} \left(T(k) + T(n-k) + 1 \right) \text{ for } n > 1$$

- For $i = 1, 2, \ldots, n$ each term $T(i)$ appears twice
  - Once as $T(k)$, and once as $T(n-k)$
- Collect $n-1$ 1's in the summation together with the front 1

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

- Prove that $T(n) = \Omega(2^n)$ using the substitution method

Running Time of RMC: Prove that $T(n) = \Omega(2^n)$

- Try to show that $T(n) \geq 2^{n-1}$ (by substitution)

Base case: $T(1) \geq 1 = 2^0 = 2^{1-1}$ for $n = 1$

IH: $T(i) \geq 2^{i-1}$ for all $i = 1, 2, \ldots, n-1$ and $n \geq 2$

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

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

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

$$\Rightarrow T(n) \geq 2^{n-1}$$ Q.E.D.

Running Time of RMC: $T(n) \geq 2^{n-1}$

Whenever

- a recursion tree for the natural recursive solution to a problem contains the same subproblem repeatedly
- the total number of different subproblems is small

it is a good idea to see if DP can be applied

Memoization

- Offers the efficiency of the usual DP approach while maintaining top-down strategy
- Idea is to memoize the natural, but inefficient, recursive algorithm

Memoized Recursive Algorithm

- Maintains an entry in a table for the soln to each subproblem
- Each table entry contains a special value to indicate that the entry has yet to be filled in
- When the subproblem is first encountered its solution is computed and then stored in the table
- Each subsequent time that the subproblem encountered the value stored in the table is simply looked up and returned

Memoized Recursive Algorithm

- The approach assumes that
  - The set of all possible subproblem parameters are known
  - The relation between the table positions and subproblems is established
- Another approach is to memoize
  - by using hashing with subproblem parameters as *key*

Memoized Recursive Matrix-chain Order

```
MemoizedMatrixChain(p)
    n ← length[p] – 1
    for i ← 1 to n do
        for j ← 1 to n do
            m[i, j] ← ∞
    return LookupC(p, 1, n)
```

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

▷ Shaded subtrees are looked-up rather than recomputing

Elements of Dynamic Programming: Summary

- Matrix-chain multiplication can be solved in $O(n^3)$ time
  - by either a top-down memoized recursive algorithm
  - or a bottom-up dynamic programming algorithm
- Both methods exploit the overlapping subproblems property
  - There are only $\Theta(n^2)$ different subproblems in total
  - Both methods compute the soln to each problem once
- Without memoization the natural recursive algorithm runs in exponential time since subproblems are solved repeatedly

Elements of Dynamic Programming: Summary

In general practice

- If all subproblems must be solved at once
  - a bottom-up DP algorithm always outperforms a top-down memoized algorithm by a constant factor

  because, bottom-up DP algorithm
    - Has no overhead for recursion
    - Less overhead for maintaining the table
- DP: Regular pattern of table accesses can be exploited to reduce the time and/or space requirements even further
- Memoized: If some problems need not be solved at all, it has the advantage of avoiding solutions to those subproblems

Longest Common Subsequence

A subsequence of a given sequence is just the given sequence with some elements (possibly none) left out

Formal definition: Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a subsequence of X if $\exists$ a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that $x_i = z_j$ for all $j = 1, 2, \ldots, k$, where $1 \le k \le m$

Example: $Z = \langle B,C,D,B \rangle$ is a subsequence of $X = \langle A,B,C,B,D,A,B \rangle$ (indices 1 2 3 4 5 6 7) with the index sequence $\langle i_1, i_2, i_3, i_4 \rangle = \langle 2, 3, 5, 7 \rangle$

Longest Common Subsequence (LCS)

Given two sequences X & Y, Z is a common subsequence of X & Y

Example: $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$
Sequence $\langle B, C, A \rangle$ is a common subsequence of X and Y.
However, $\langle B, C, A \rangle$ is not a longest common subsequence (LCS) of X and Y.
$\langle B, C, B, A \rangle$ is an LCS of X and Y.

Longest common subsequence (LCS):

Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$

We wish to find the LCS of X & Y

Characterizing a Longest Common Subsequence

A brute force approach

- Enumerate all subsequences of X
- Check each subsequence to see if it is also a subsequence of Y meanwhile keeping track of the LCS found
- Each subsequence of X corresponds to a subset of the index set $\{1, 2, \ldots, m\}$ of X

- So, there are 2^m subsequences of X
- Hence, this approach requires exponential time

Characterizing a Longest Common Subsequence

Definition: The i-th prefix X_i of X for $i = 0,1, \ldots, m$ is

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

Example: Given $X = \overset{1}{\text{A}}\ \overset{2}{\text{B}}\ \overset{3}{\text{C}}\ \overset{4}{\text{B}}\ \overset{5}{\text{D}}\ \overset{6}{\text{A}}\ \overset{7}{\text{B}}$

$X_4 = \langle A, B, C, B \rangle$ and $X_\varnothing =$ empty sequence

Theorem: (Optimal substructure of an LCS)

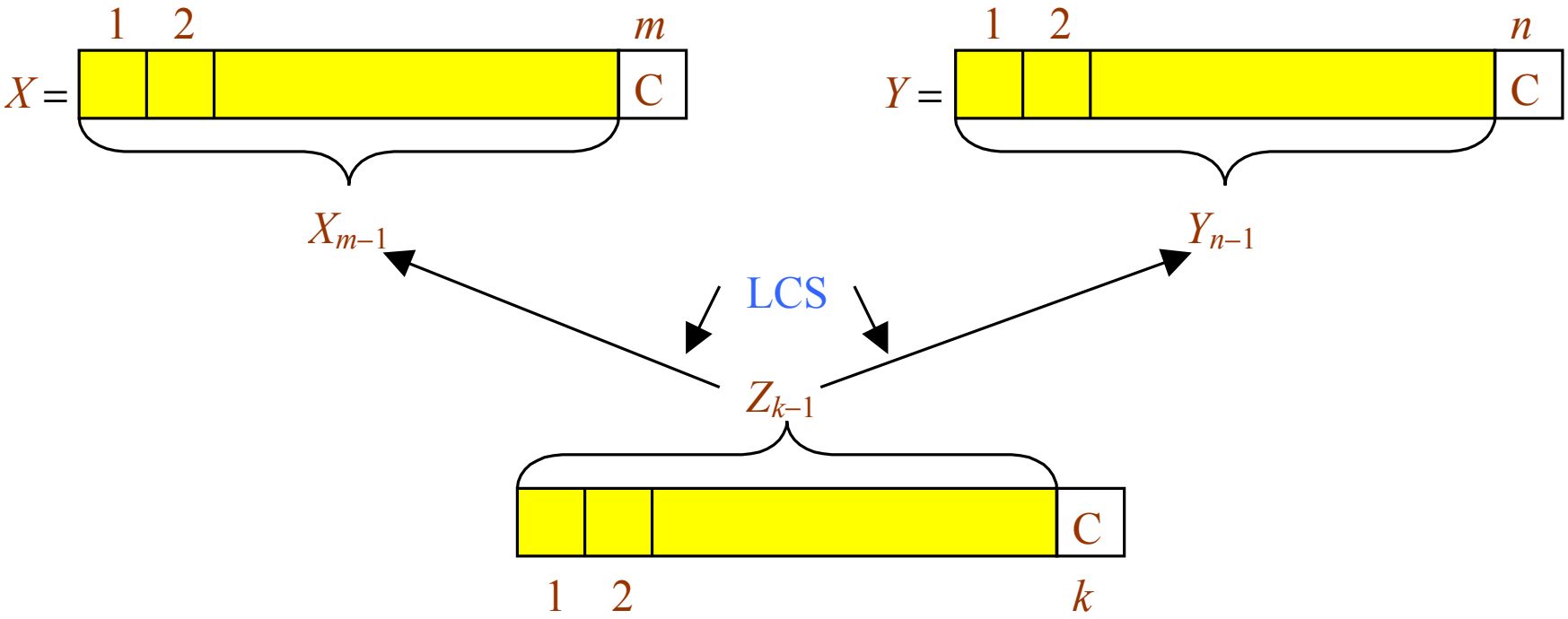
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ are given

Let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y

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

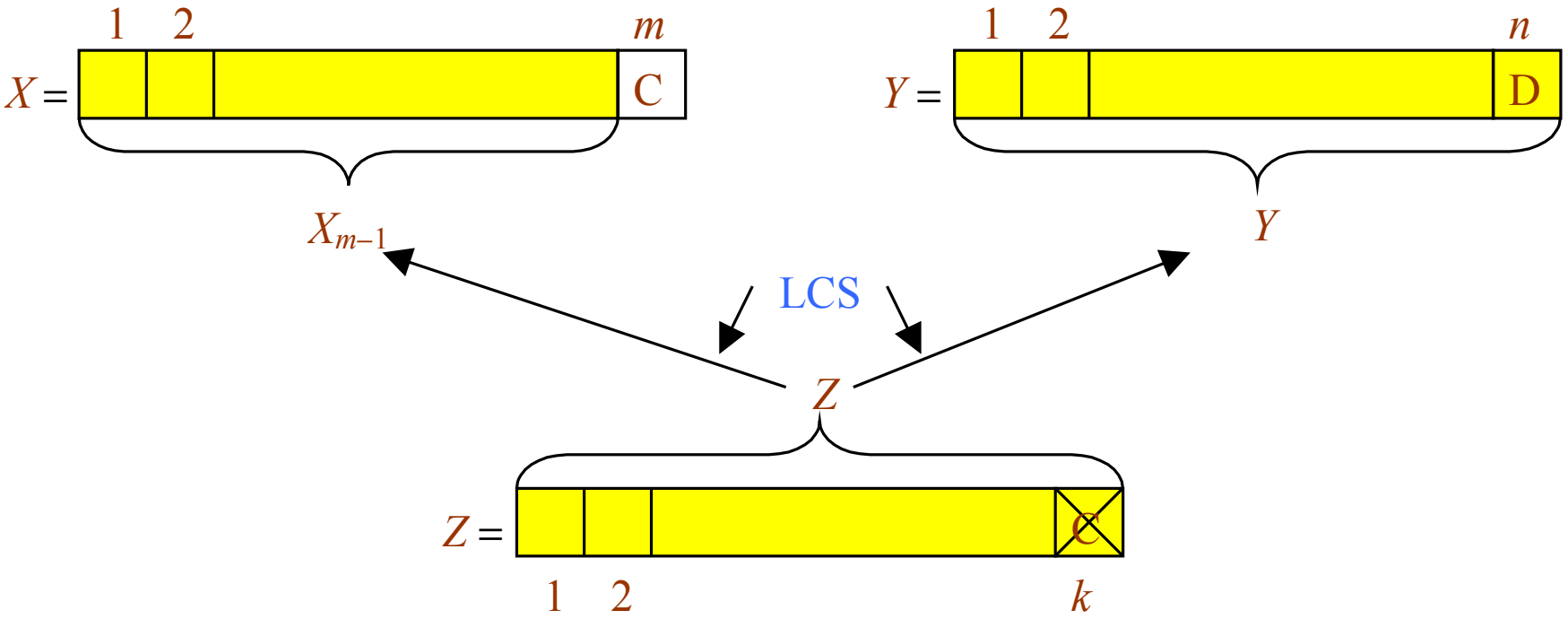
Optimal Substructure Theorem (case 1)

If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}



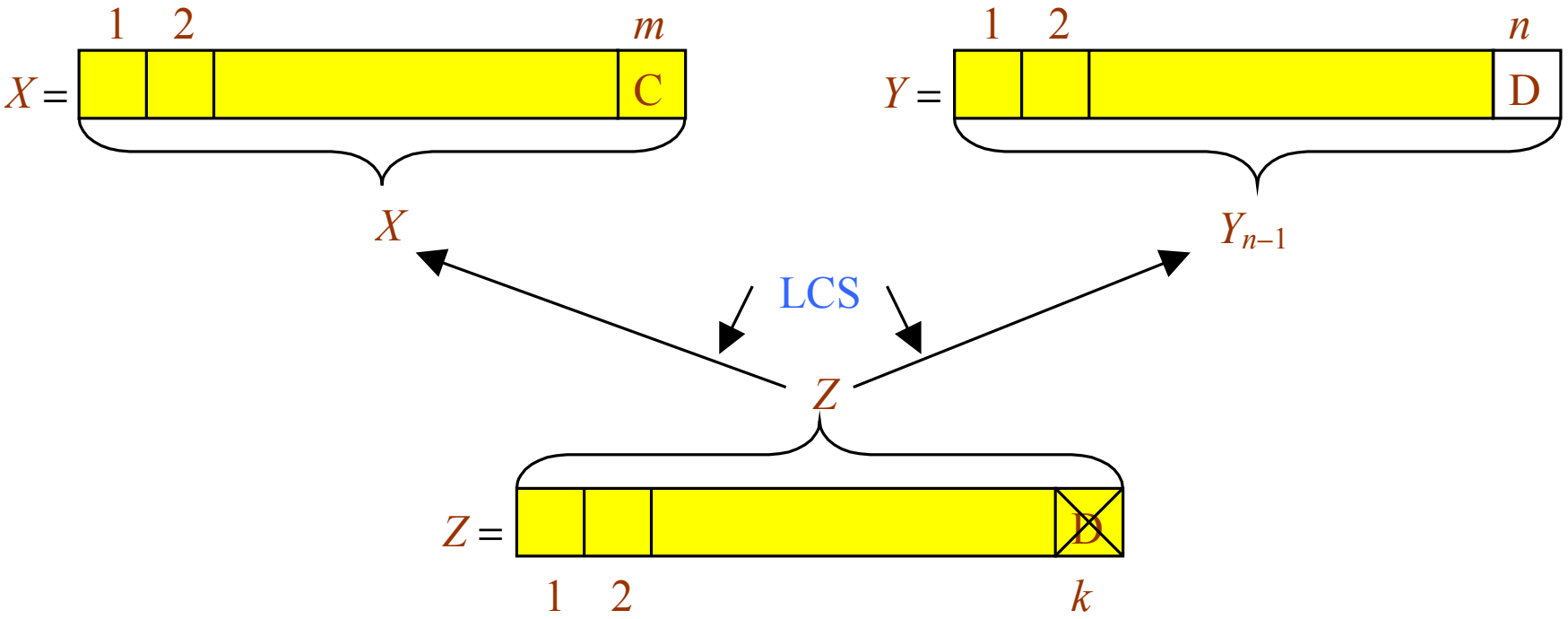
Optimal Substructure Theorem (case 2)

If $x_m \neq y_n$ and $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y



Optimal Substructure Theorem (case 3)

If $x_m \neq y_n$ and $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}



Proof of Optimal Substructure Theorem (case 1)

If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}

Proof: If $z_k \neq x_m = y_n$ then

we can append $x_m = y_n$ to Z to obtain a common subsequence of length $k+1$ $\Rightarrow$ contradiction

Thus, we must have $z_k = x_m = y_n$

Hence, the prefix Z_{k-1} is a length-$(k-1)$ CS of X_{m-1} and Y_{n-1}

We have to show that Z_{k-1} is in fact an LCS of X_{m-1} and Y_{n-1}

Proof by contradiction:

Assume that $\exists$ a CS W of X_{m-1} and Y_{n-1} with $|W| = k$

Then appending $x_m = y_n$ to W produces a CS of length $k+1$

Proof of Optimal Substructure Theorem (case 2)

If $x_m \neq y_n$ and $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y

Proof : If $z_k \neq x_m$ then Z is a CS of X_{m-1} and Y_n

We have to show that Z is in fact an LCS of X_{m-1} and Y_n

(Proof by contradiction)

Assume that $\exists$ a CS W of X_{m-1} and Y_n with $|W| > k$

Then W would also be a CS of X and Y

Contradiction to the assumption that

Z is an LCS of X and Y with $|Z| = k$

Case 3: Dual of the proof for (case 2)

Longest Common Subsequence Algorithm

```
LCS(X, Y)
    m ← length[X]
    n ← length[Y]
    if x_m = y_n then
        Z ← LCS(X_{m-1}, Y_{n-1})     ▷ solve one subproblem
        return <Z, x_m = y_n>          ▷ append x_m = y_n to Z
    else
        Z' ← LCS(X_{m-1}, Y)           } ▷ solve two subproblems
        Z'' ← LCS(X, Y_{n-1})          }
        return longer of Z' and Z''
```

A Recursive Solution to Subproblems

Theorem implies that there are one or two subproblems to examine

if $x_m = y_n$ then

we must solve the subproblem of finding an LCS of X_{m-1} & Y_{n-1}

appending $x_m = y_n$ to this LCS yields an LCS of X & Y

else

we must solve two subproblems

– finding an LCS of X_{m-1} & Y

– finding an LCS of X & Y_{n-1}

longer of these two LCSs is an LCS of X & Y

endif

A Recursive Solution to Subproblems

Overlapping-subproblems property

- finding an LCS to X_{m-1} & Y and an LCS to X & Y_{n-1} has the subsubproblem of finding an LCS to X_{m-1} & Y_{n-1}
- many other subproblems share subsubproblems

A recurrence for the cost of an optimal solution

$c[i, j]$: length of an LCS of the prefix subsequences X_i & Y_j

If either $i = 0$ or $j = 0$, one of the prefix sequences has length 0, so the LCS has length 0

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

Computing the Length of an LCS

We can easily write an exponential-time recursive algorithm based on the given recurrence

However, there are only $\Theta(mn)$ distinct subproblems

Therefore, we can use dynamic programming

Data structures:

Table $c[0\ldots m, 0\ldots n]$ is used to store $c[i, j]$ values

Entries of this table are computed in row-major order

Table $b[1\ldots m, 1\ldots n]$ is maintained to simplify the construction of an optimal solution

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

Computing the Length of an LCS

```
LCS-LENGTH(X,Y)
    m ← length[X]; n ← length[Y]
    for i ← 0 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]
                c[i, j] ← c[i–1, j]
                b[i, j] ← "□"
            else
                c[i, j] ← c[i, j–1]
                b[i, j] ← "□"
```

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

$$X = \langle A, B, C, B, D, A, B \rangle \quad (1\ 2\ 3\ 4\ 5\ 6\ 7)$$

$$Y = \langle B, D, C, A, B, A \rangle \quad (1\ 2\ 3\ 4\ 5\ 6)$$

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						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

X = <A, B, C, B, D, A, B> (1 2 3 4 5 6 7)

Y = <B, D, C, A, B, A> (1 2 3 4 5 6)

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						
3 C	0						
4 B	0						
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

X = <A, B, C, B, D, A, B> (1 2 3 4 5 6 7)

Y = <B, D, C, A, B, A> (1 2 3 4 5 6)

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						
4 B	0						
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

X = <A, B, C, B, D, A, B> (1 2 3 4 5 6 7)

Y = <B, D, C, A, B, A> (1 2 3 4 5 6)

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						
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

X = <A, B, C, B, D, A, B> (1 2 3 4 5 6 7)

Y = <B, D, C, A, B, A> (1 2 3 4 5 6)

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					
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

X = <A, B, C, B, D, A, B> (1 2 3 4 5 6 7)

Y = <B, D, C, A, B, A> (1 2 3 4 5 6)

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				
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

X = <A, B, C, B, D, A, B> (1 2 3 4 5 6 7)

Y = <B, D, C, A, B, A> (1 2 3 4 5 6)

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			
5	D	0						
6	A	0						
7	B	0						

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

$$\begin{array}{lccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ X = & \langle A, & B, & C, & B, & D, & A, & B \rangle \end{array}$$

$$\begin{array}{lcccccc} Y = & \langle B, & D, & C, & A, & B, & A \rangle \\ & 1 & 2 & 3 & 4 & 5 & 6 \end{array}$$

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		
5 D	0						
6 A	0						
7 B	0						

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

$$X = \langle A, B, C, B, D, A, B \rangle \quad (1\ 2\ 3\ 4\ 5\ 6\ 7)$$

$$Y = \langle B, D, C, A, B, A \rangle \quad (1\ 2\ 3\ 4\ 5\ 6)$$

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	
5	D	0						
6	A	0						
7	B	0						

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

X = <A, B, C, B, D, A, B> (1 2 3 4 5 6 7)

Y = <B, D, C, A, B, A> (1 2 3 4 5 6)

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						
6	A	0						
7	B	0						

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

$$\begin{matrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ X = & \langle A, & B, & C, & B, & D, & A, & B \rangle \end{matrix}$$

$$\begin{matrix} Y = & \langle B, & D, & C, & A, & B, & A \rangle \\ & 1 & 2 & 3 & 4 & 5 & 6 \end{matrix}$$

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						
7	B	0						

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

X = <A, B, C, B, D, A, B> (1 2 3 4 5 6 7)

Y = <B, D, C, A, B, A> (1 2 3 4 5 6)

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						

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

X = <A, B, C, B, D, A, B> (1 2 3 4 5 6 7)

Y = <B, D, C, A, B, A> (1 2 3 4 5 6)

Running-time = O(mn) since each table entry takes

O(1) time to compute

LCS of X & Y = <B, C, B, A>

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	↑ 3	↑ 3	↖ 4	↑ 4

Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

X = <A, B, C, B, D, A, B> (1 2 3 4 5 6 7)

Y = <B, D, C, A, B, A> (1 2 3 4 5 6)

Running-time = O(mn) since each table entry takes

O(1) time to compute

LCS of X & Y = <B, C, B, A>

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	↑ 3	↑ 3	↖ 4	↑ 4

Constructing an LCS

The b table returned by LCS-LENGTH can be used to quickly construct an LCS of X & Y

Begin at $b[m, n]$ and trace through the table following arrows

Whenever you encounter a "□" in entry $b[i, j]$
it implies that $x_i = y_j$ is an element of LCS

The elements of LCS are encountered in reverse order

Constructing an 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)
```

The initial invocation:
PRINT-LCS(b, X, length[X], length[Y])

The recursive procedure PRINT-LCS prints out LCS in proper order

This procedure takes O(m+n) time
since at least one of i and j is determined in each stage of the recursion

Longest Common Subsequence

Improving the code:

- we can eliminate the b table altogether
- each $c[i, j]$ entry depends only on 3 other c table entries $c[i–1, j–1]$, $c[i–1, j]$ and $c[i, j–1]$

Given the value of $c[i, j]$

- we can determine in O(1) time which of these 3 values was used to compute $c[i, j]$ without inspecting table b
- we save $\Theta(mn)$ space by this method
- however, space requirement is still $\Theta(mn)$ since we need $\Theta(mn)$ space for the c table anyway

We can reduce the asymptotic space requirement for LCS-LENGTH

- since it needs only two rows of table c at a time
- the row being computed and the previous row

~~This improvement works if we only need the length of an LCS~~