Lecture 10
Dynamic Programming

View in slide-show mode
Introduction

• An algorithm design paradigm like divide-and-conquer
• “Programming”: A tabular method (not writing computer code)
  Older sense of planning or scheduling, typically by filling in a table
• 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
Example: Fibonacci Numbers (Recursive Solution)

Reminder:
F(0) = 0 and F(1) = 1
F(n) = F(n-1) + F(n-2)

REC-FIBO(n)
if n < 2
  return n
else
  return REC-FIBO(n-1) + REC-FIBO(n-2)

Overlapping subproblems in different recursive calls. Repeated work!
Example: Fibonacci Numbers (Recursive Solution)

Recurrence:

\[ T(n) = T(n-1) + T(n-2) + 1 \]

\[ \rightarrow \text{exponential runtime} \]

Recursive algorithm inefficient because it recomputes the same \( F(i) \) repeatedly in different branches of the recursion tree.
Example: Fibonacci Numbers (Bottom-up Computation)

Reminder:
F(0) = 0 and F(1) = 1
F(n) = F(n-1) + F(n-2)

ITER-FIBO(n)
F[0] = 0
F[1] = 1
for i = 2 to n do
  F[i] = F[i-1] + F[i-2]
return F[n]

Runtime: \( \Theta(n) \)
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.

\[
\begin{align*}
(A_i(A_{i+1}A_{i+2} \ldots A_j)) \\
((A_iA_{i+1}A_{i+2} \ldots A_{j-1})A_j) \\
((A_iA_{i+1}A_{i+2} \ldots A_k)(A_{k+1}A_{k+2} \ldots A_j)) \quad \text{for } i \leq k < j
\end{align*}
\]
- 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
  
  \[
  \begin{align*}
  & (A_1(A_2(A_3A_4))) \\
  & (A_1((A_2A_3)A_4)) \\
  & ((A_1A_2)(A_3A_4)) \\
  & ((A_1(A_2A_3))A_4) \\
  & (((A_1A_2)A_3)A_4)
  \end{align*}
  \]
- The way we parenthesize a chain of matrices can have a dramatic effect on the cost of computing the product
Reminder: Matrix Multiplication

**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

A \times B = C

\begin{align*}
\text{rows}(A) &= p & \text{rows}(B) &= q \\
\text{cols}(A) &= q & \text{cols}(B) &= r \\
\text{rows}(C) &= p & \text{cols}(C) &= r
\end{align*}
**Reminder: Matrix Multiplication**

**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

A: p x q
B: q x r
C: p x r

# of mult-add ops = rows[A] x cols[B] x cols[A]

# of mult-add ops = p x q x r
Matrix Chain Multiplication: Example

A₁: 10x100 \hspace{1cm} A₂: 100x5 \hspace{1cm} A₃: 5x50

Which paranthesization is better? \((A₁A₂)A₃\) or \(A₁(A₂A₃)\)?

\[
\begin{align*}
10 \begin{bmatrix} 100 \\ A₁ \end{bmatrix} & \times 100 \begin{bmatrix} 5 \\ A₂ \end{bmatrix} = 10 \begin{bmatrix} 5 \\ A₁A₂ \end{bmatrix} & \text{# of ops: } 10 \cdot 100 \cdot 5 \\
10 \begin{bmatrix} 5 \\ A₁A₂ \end{bmatrix} & \times 5 \begin{bmatrix} 50 \\ A₃ \end{bmatrix} = 10 \begin{bmatrix} 50 \\ A₁A₂A₃ \end{bmatrix} & \text{# of ops: } 10 \cdot 5 \cdot 50 \\
\end{align*}
\]

Total # of ops: 7500
Matrix Chain Multiplication: Example

A₁: 10x100  A₂: 100x5  A₃: 5x50

Which paranthesization is better? (A₁A₂)A₃ or A₁(A₂A₃)?

\[
\begin{align*}
A_1 &: \begin{bmatrix} 100 \\ \end{bmatrix} \\
A_2 &: \begin{bmatrix} 5 \\ \end{bmatrix} \\
A_3 &: \begin{bmatrix} 50 \\ \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
A_2 \times A_3 &= \begin{bmatrix} 100 \\ \end{bmatrix} \\
\end{align*}
\]

\[
\begin{align*}
A_1 \times (A_2 \times A_3) &= \begin{bmatrix} 5 \\ \end{bmatrix} \\
&= \begin{bmatrix} 50 \\ \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
# \text{ of ops: } 100 \times 5 \times 50 &= 25000
\end{align*}
\]

\[
\begin{align*}
A_1 \times (A_2 \times A_3) &= \begin{bmatrix} 100 \\ \end{bmatrix} \\
&= \begin{bmatrix} 10 \\ \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
# \text{ of ops: } 10 \times 100 \times 50 &= 50000
\end{align*}
\]

Total # of ops: 75000
Matrix Chain Multiplication: Example

\[ A_1: 10 \times 100 \quad A_2: 100 \times 5 \quad A_3: 5 \times 50 \]

Which paranthesization is better? \((A_1A_2)A_3\) or \(A_1(A_2A_3)\)?

\[ \text{In summary:} \quad (A_1A_2)A_3 \Rightarrow \# \text{ of multiply-add ops: 7500} \]
\[ A_1(A_2A_3) \Rightarrow \# \text{ of multiple-add ops: 75000} \]

\[ \Rightarrow \text{First parenthesization yields 10x faster computation} \]
Matrix-chain Multiplication Problem

**Input:** A chain \(\langle A_1, A_2, \ldots, A_n \rangle\) of \(n\) matrices, where \(A_i\) is a \(p_{i-1} \times p_i\) matrix.

**Objective:** Fully parenthesize the product

\[
A_1 \cdot A_2 \cdot \ldots \cdot A_n
\]

such that the number of scalar mult-adds is minimized.
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 $k^{th}$ and $(k+1)^{st}$ matrices for any $k=1, 2, \ldots, n-1$, 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 Optimal Parenthesization

**Notation**: \( A_{i..j} \): The matrix that results from evaluation of the product: \( A_i \ A_{i+1} \ A_{i+2} \ldots \ A_j \)

**Observation**: Consider the last multiplication operation in any parenthesization: \( (A_1 \ A_2 \ldots \ A_k) \cdot (A_{k+1} \ A_{k+2} \ldots \ A_n) \)

There is a \( k \) value \( (1 \leq k < n) \) such that:

First, the product \( A_{1..k} \) is computed
Then, the product \( A_{k+1..n} \) is computed
Finally, the matrices \( A_{1..k} \) and \( A_{k+1..n} \) are multiplied
Step 1: Characterize the structure of an optimal solution

- An optimal parenthesization of product $A_1A_2...A_n$ will be: $(A_1 A_2 ... A_k) . (A_{k+1} A_{k+2} ... A_n)$ for some $k$ value.

- The cost of this optimal parenthesization will be:
  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.
Step 2: A Recursive Solution

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

Assume we are trying to determine the min cost of computing $A_{i..j}$

$m_{i,j}$: min # of scalar multiply-add opns needed to compute $A_{i..j}$

*Note: The optimal cost of the original problem: $m_{1,n}$*

How to compute $m_{i,j}$ recursively?
Step 2: A recursive Solution

Base case: $m_{i,i} = 0$ (single matrix, no multiplication)

Let the size of matrix $A_i$ be $(p_{i-1} \times p_i)$

Consider an optimal parenthesization of chain $A_i \ldots A_j$: $(A_i \ldots A_k) \cdot (A_{k+1} \ldots A_j)$

The optimal cost:

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

where:

- $m_{i,k}$: Optimal cost of computing $A_{i..k}$
- $m_{k+1,j}$: Optimal cost of computing $A_{k+1..j}$
- $p_{i-1} \times p_k \times p_j$: Cost of multiplying $A_{i..k}$ and $A_{k+1..j}$
Step 2: A Recursive Solution

In an optimal parenthesization:

\[ k \text{ must be chosen to minimize } m_{ij} \]

The recursive formulation for \( m_{ij} \):

\[
m_{ij} = \begin{cases} 
0 & \text{if } i=j \\
\min_{i \leq k < j} \{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \} & \text{if } i < j
\end{cases}
\]
Step 2: A Recursive Solution

- 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
  $$m_{ij} = m_{ik} + m_{k+1, j} + p_{i-1} p_k p_j$$ holds.
Direct Recursion: Inefficient!

Recursive matrix-chain order

\[
\text{RMC}(p, i, j)
\]

- if \( i = j \) then
  - return 0
- \( m[i, j] \leftarrow \infty \)
- for \( k \leftarrow i \) to \( j - 1 \) do
  - \( q \leftarrow \text{RMC}(p, i, k) + \text{RMC}(p, k+1, j) + p_{i-1} p_k p_j \)
  - if \( q < m[i, j] \) then
    - \( m[i, j] \leftarrow q \)
- return \( m[i, j] \)
Direct Recursion: Inefficient!

Recursion tree for $\text{RMC}(p,1,4)$

Nodes are labeled with $i$ and $j$ values

Redundant calls are filled
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 $1 \leq i \leq j \leq n$
  - 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 $\text{length}[p] = n + 1$

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 $m[i, j]$
Bottom-up computation

\[ m_{ij} = \min_{i \leq k < j} \left\{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \right\} \]

How to choose the order in which we process \( m_{ij} \) values?

Before computing \( m_{ij} \), we have to make sure that the values for \( m_{ik} \) and \( m_{k+1,j} \) have been computed for all \( k \).
\[ m_{ij} = \min_{i \leq k < j} \left\{ m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j \right\} \]

Reminder: \( m_{ij} \) must be processed after \( m_{ik} \) and \( m_{j,k+1} \)

Reminder: \( m_{ij} \) computed only for \( j > i \)
\[ m_{ij} = \min_{i \leq k < j} \{ m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j \} \]

How to set up the iterations over \(i\) and \(j\) to compute \(m_{ij}\)?
If the entries $m_{ij}$ are computed in the shown order, then $m_{ik}$ and $m_{k+1,j}$ values are guaranteed to be computed before $m_{ij}$. 

$$m_{ij} = \min_{i \leq k < j} \left\{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \right\}$$
\[ m_{ij} = \min_{i \leq k < j} \left\{ m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j \right\} \]
\[ m_{ij} = \min_{i \leq k < j} \{ m_{ik} + m_{k+1,j} + p_{i-\ell} p_k p_j \} \]

for \( \ell = 2 \) to \( n \)
for \( i = 1 \) to \( n-\ell+1 \)

\[ j = i + \ell - 1 \]

\[ \ldots \]

\[ m_{ij} = \ldots \]
Algorithm for Computing the Optimal Costs

**MATRIX-CHAIN-ORDER**\((p)\)

\[
n \leftarrow \text{length}[p] - 1
\]
for \(i \leftarrow 1\) to \(n\) do

\[
m[i, i] \leftarrow 0
\]
for \(\ell \leftarrow 2\) to \(n\) do

for \(i \leftarrow 1\) to \(n - \ell + 1\) do

\[
j \leftarrow i + \ell - 1
\]

\[
m[i, j] \leftarrow \infty
\]
for \(k \leftarrow i\) to \(j - 1\) do

\[
q \leftarrow m[i, k] + m[k+1, j] + p_{i-1}p_kp_j
\]
if \(q < m[i, j]\) then

\[
m[i, j] \leftarrow q
\]

\[
s[i, j] \leftarrow k
\]

return \(m\) and \(s\)
Algorithm for Computing the Optimal Costs

- The algorithm first computes
  \[ m[i, i] \leftarrow 0 \text{ for } i = 1, 2, \ldots, n \]
  min costs for all chains of length 1

- Then, for \( \ell = 2, 3, \ldots, n \) computes
  \[ m[i, i+\ell-1] \text{ for } i = 1, \ldots, n-\ell+1 \]
  min costs for all chains of length \( \ell \)

- For each value of \( \ell = 2, 3, \ldots, n \),
  \[ m[i, i+\ell-1] \text{ depends only on table entries } m[i, k] \& m[k+1, i+\ell-1] \]
  for \( i \leq k < i+\ell-1 \), which are already computed
Algorithm for Computing the Optimal Costs

\[ \ell = 2 \]
for \( i = 1 \) to \( n - 1 \)
\[
m[i, i+1] = \infty
\]
for \( k = i \) to \( i \) do
\[
\cdot
\]
\[ \ell = 3 \]
for \( i = 1 \) to \( n - 2 \)
\[
m[i, i+2] = \infty
\]
for \( k = i \) to \( i+1 \) do
\[
\cdot
\]
\[ \ell = 4 \]
for \( i = 1 \) to \( n - 3 \)
\[
m[i, i+3] = \infty
\]
for \( k = i \) to \( i+2 \) do
\[
\cdot
\]
Table access pattern in computing $m[i, j]s$ for $\ell = j - i + 1$

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

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

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

$$((A_i) (A_{i+1} A_{i+2} \ldots A_j))$$

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 $\ell=j-i+1$

\[
(((A_i A_{i+1}) (A_{i+2} \ldots A_j))
\]

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 $\ell=j−i+1$

$((A_iA_{i+1}A_{i+2})(A_{i+3}...A_j))$

for $k \leftarrow i$ to $j−1$ do
$q \leftarrow m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$

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

$((A_i A_{i+1} \ldots A_{j-1}) (A_j))$

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
Example

\[ m_{ij} = \min_{i \leq k < j} \{ m_{ik} + m_{k+1,j} + p_{i-1}p_k p_j \} \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁: (30x35)</td>
<td>0</td>
<td>15750</td>
<td>7875</td>
<td>9375</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>A₂: (35x15)</td>
<td>0</td>
<td>2625</td>
<td>4375</td>
<td>???</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>A₃: (15x5)</td>
<td>0</td>
<td>750</td>
<td>2500</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>A₄: (5x10)</td>
<td>0</td>
<td>1000</td>
<td>3500</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A₅: (10x20)</td>
<td>0</td>
<td>5000</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A₆: (20x25)</td>
<td>0</td>
<td>6</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compute \( m_{25} \)

\( k = 2 \)

\[ (A₂)(A₃A₄A₅) \]

\[
\text{cost} = m_{22} + m_{35} + p₁p₂p₅ \\
= 0 + 2500 + 35\times15\times20 \\
= 13000
\]

Choose the \( k \) value that leads to min cost
Example

\[ m_{ij} = \min_{i \leq k < j} \left\{ m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j \right\} \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_1): (30x35)</td>
<td>0</td>
<td>15750</td>
<td>7875</td>
<td>9375</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>(A_2): (35x15)</td>
<td></td>
<td>2625</td>
<td>4375</td>
<td></td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>(A_3): (15x5)</td>
<td></td>
<td>0</td>
<td>750</td>
<td>2500</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>(A_4): (5x10)</td>
<td></td>
<td></td>
<td>0</td>
<td>1000</td>
<td>3500</td>
<td>4</td>
</tr>
<tr>
<td>(A_5): (10x20)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>5000</td>
<td>5</td>
</tr>
<tr>
<td>(A_6): (20x25)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

\((A_2A_3) (A_4A_5)\)

\[
\text{cost} = m_{23} + m_{45} + p_1p_3p_5 \\
= 2625 + 1000 + 35 \times 5 \times 20 \\
= 7125
\]

Choose the \(k\) value that leads to min cost.
## Example

### $m_{ij} = \min_{i \leq k < j} \left\{ m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j \right\}$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$: (30x35)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>$A_2$: (35x15)</td>
<td></td>
<td></td>
<td></td>
<td>15750</td>
<td>7875</td>
<td>9375</td>
</tr>
<tr>
<td>$A_3$: (15x5)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>2625</td>
<td>4375</td>
</tr>
<tr>
<td>$A_4$: (5x10)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>750</td>
<td>2500</td>
</tr>
<tr>
<td>$A_5$: (10x20)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1000</td>
<td>3500</td>
</tr>
<tr>
<td>$A_6$: (20x25)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>5000</td>
<td></td>
</tr>
</tbody>
</table>

**Compute $m_{25}$**

$k = 4$

$$(A_2A_3A_4)(A_5)$$

**cost** = $m_{24} + m_{55} + p_1p_4p_5$

\[= 4375 + 0 + 35 \times 10 \times 20\]

\[= 11375\]

---

*Choose the $k$ value that leads to min cost*
**Example**

\[ m_{ij} = \min_{i \leq k < j} \{ m_{ik} + m_{k+1,j} + p_{i-1} p_k p_j \} \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1: (30x35)</td>
<td>0</td>
<td>15750</td>
<td>7875</td>
<td>9375</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>A2: (35x15)</td>
<td>0</td>
<td>2625</td>
<td>4375</td>
<td>7125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3: (15x5)</td>
<td>0</td>
<td>750</td>
<td>2500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4: (5x10)</td>
<td>0</td>
<td>1000</td>
<td>3500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5: (10x20)</td>
<td>0</td>
<td>5000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6: (20x25)</td>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compute \( m_{25} \)

\( k = 3 \)

\( (A_2A_3)(A_4A_5) \)

\( m_{25} = 7125 \)

\( s_{25} = 3 \)

Choose \( k = 3 \)
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 $A_{1\ldots s[1,n]} \times A_{s[1,n]+1,n}$
Example: Constructing an Optimal Solution

**Reminder**: \( s_{ij} \) is the optimal top-level split of \( A_i \ldots A_j \)

What is the optimal top-level split for:

\[ A_1A_2A_3A_4A_5A_6 \]

\[ s_{16} = 3 \]
Reminder: $s_{ij}$ is the optimal top-level split of $A_i \ldots A_j$

$k = 3$

$(A_1A_2A_3)(A_4A_5A_6)$

What is the optimal split for $A_1 \ldots A_3$? $s_{13} = 1$

What is the optimal split for $A_4 \ldots A_6$? $s_{46} = 5$
Example: Constructing an Optimal Solution

Reminder: $s_{ij}$ is the optimal top-level split of $A_i \ldots A_j$

$$k=1 \quad k=5$$

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

What is the optimal split for $A_1 \ldots A_3$? $s_{13} = 1$

What is the optimal split for $A_4 \ldots A_6$? $s_{46} = 5$
Example: Constructing an Optimal Solution

**Reminder**: $s_{ij}$ is the optimal top-level split of $A_i \ldots A_j$

What is the optimal split for $A_2A_3$? $s_{23} = 2$

What is the optimal split for $A_4A_5$? $s_{45} = 4$
Example: Constructing an Optimal Solution

Reminder: $s_{ij}$ is the optimal top-level split of $A_i...A_j$

What is the optimal split for $A_2A_3$? $s_{23} = 2$

What is the optimal split for $A_4A_5$? $s_{45} = 4$
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 \textsc{Matrix-Chain-Order}

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

\textbf{MATRIX-CHAIN-MULTIPLY}(A, s, i, j)

\begin{verbatim}
if \( j > i \) then
    X ← \textsc{Matrix-Chain-Multiply}(A, s, i, s[i,j])
    Y ← \textsc{Matrix-Chain-Multiply}(A, s, s[i,j]+1, j)
    return \textsc{Matrix-Multiply}(X, Y)
else
    return \( A_i \)
\end{verbatim}

Invocation: \textsc{Matrix-Chain-Multiply}(A, s, 1, n)
Example: Recursive Construction of an Optimal Solution

\[
\begin{align*}
MCM(1,6) &\quad X &\leftarrow MCM(1,3) = (A_1A_2A_3) \\
Y &\leftarrow MCM(4,6) = (A_4A_5A_6) \\
\text{return (\?)} \\
MCM(1,3) &\quad X &\leftarrow MCM(1,1) = A_1 \\
Y &\leftarrow MCM(2,3) = (A_2A_3) \\
\text{return (\?)} \\
\end{align*}
\]
Example: Recursive Construction of an Optimal Solution

MCM(1,6)
X ← MCM(1,3) = (A_1(A_2A_3))
Y ← MCM(4,6) = (A_4A_5A_6)
return (?)
Example: Recursive Construction of an Optimal Solution

\[ s[1\ldots6, 1\ldots6] = \begin{array}{cccccc}
1 & 1 & 1 & 3 & 3 & 3 \\
2 & 2 & 3 & 4 & 3 & \\
3 & 3 & 3 & 3 & & \\
\end{array} \]

- \( MCM(1,6) \)
  - \( X \leftarrow MCM(1,3) \)
  - \( Y \leftarrow MCM(4,6) \)
  - \( \text{return } (A_1(A_2A_3))((A_4A_5)A_6) \)

- \( MCM(1,3) \)
  - \( X \leftarrow MCM(1,1) = A_1 \)
  - \( Y \leftarrow MCM(2,3) = (A_2A_3) \)
  - \( \text{return } (A_1(A_2A_3)) \)

- \( MCM(4,6) \)
  - \( X \leftarrow MCM(4,5) = (A_4A_5) \)
  - \( Y \leftarrow MCM(6,6) = A_6 \)
  - \( \text{return } ((A_4A_5)A_6) \)

- \( MCM(4,5) \)
  - \( X \leftarrow MCM(4,4) = A_4 \)
  - \( Y \leftarrow MCM(5,5) = A_5 \)
  - \( \text{return } (A_4A_5) \)

- \( MCM(2,3) \)
  - \( X \leftarrow MCM(2,2) = A_2 \)
  - \( Y \leftarrow MCM(3,3) = A_3 \)
  - \( \text{return } (A_2A_3) \)

- \( MCM(1,1) \)
  - \( X \leftarrow MCM(2,1) = A_1 \)
  - \( Y \leftarrow MCM(3,1) = A_3 \)
  - \( \text{return } A_1 \)

- \( MCM(2,2) \)
  - \( X \leftarrow MCM(3,2) = (A_2A_3) \)
  - \( Y \leftarrow MCM(4,2) = A_4 \)
  - \( \text{return } A_2 \)

- \( MCM(3,3) \)
  - \( X \leftarrow MCM(4,3) = (A_4A_5) \)
  - \( Y \leftarrow MCM(5,3) = A_5 \)
  - \( \text{return } A_3 \)

- \( MCM(4,4) \)
  - \( X \leftarrow MCM(5,4) = (A_4A_5) \)
  - \( Y \leftarrow MCM(6,4) = A_6 \)
  - \( \text{return } (A_4A_5) \)

- \( MCM(5,5) \)
  - \( X \leftarrow MCM(6,5) = A_6 \)
  - \( Y \leftarrow MCM(1,5) = A_1 \)
  - \( \text{return } A_5 \)

- \( MCM(6,6) \)
  - \( X \leftarrow MCM(1,6) = (A_1(A_2A_3)) \)
  - \( Y \leftarrow MCM(4,6) = ((A_4A_5)A_6) \)
  - \( \text{return } (A_1(A_2A_3))((A_4A_5)A_6) \)
Table reference pattern for $m[i,j]$ ($1 \leq i \leq j \leq n$)

$m[i,j]$ is referenced for the computation of
- $m[i,r]$ for $j < r \leq n$ \((n - j)\) times
- $m[r,j]$ for $1 \leq r < i$ \((i - 1)\) times
Table reference pattern for $m[i, j]$ ($1 \leq i \leq j \leq n$)

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

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

$= (n-1) - (j-i)$

The total # of references for the entire table is

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

1. Identification of the optimal substructure property

2. Recursive formulation to compute the cost of the optimal solution

3. Bottom-up computation of the table entries

4. Constructing the optimal solution by backtracking the table entries
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
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 $A_1A_2\ldots A_n$ that splits the product between $A_k$ and $A_{k+1}$, contains within it optimal soln’s to the problems of parenthesizing $A_1A_2\ldots A_k$ and $A_{k+1}A_{k+2}\ldots A_n$
Optimal Substructure

Finding a suitable space of subproblems

• Iterate on subproblem instances

• Example: matrix-chain-multiplication
  – Iterate and look at the structure of optimal solution’s to subproblems, sub-subproblems, and so forth
  – Discover that all subproblems consist of subchains of \( \langle A_1, A_2, \ldots, A_n \rangle \)
  – Thus, the set of chains of the form
    \[ \langle A_i, A_{i+1}, \ldots, A_j \rangle \text{ for } 1 \leq i \leq j \leq n \]
  – Makes a natural and reasonable space of subproblems
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

\[ \text{RMC}(p, i, j) \]

\[
\begin{align*}
\text{if } i = j & \text{ then} \\
& \text{return } 0 \\
\text{ } m[i, j] & \leftarrow \infty \\
\text{for } k \leftarrow i \text{ to } j - 1 & \text{ do} \\
& q \leftarrow \text{RMC}(p, i, k) + \text{RMC}(p, k+1, j) + p_{i-1} p_k p_j \\
\text{if } q < m[i, j] & \text{ then} \\
& m[i, j] \leftarrow q \\
\text{return } m[i, j]
\end{align*}
\]
Recursive Matrix-chain Order

Recursion tree for $\text{RMC}(p, 1, 4)$

Nodes are labeled with $i$ and $j$ values

Redundant calls are filled
Running Time of RMC

\[ T(1) \geq 1 \]
\[ T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \text{ for } n > 1 \]

- 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 $n \geq 2$

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

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

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

$\Rightarrow T(n) \geq 2^{n-1}$  \textbf{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 solution 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 Matrix-chain Order**

**LookupC(p, i, j)**

- If \( m[i, j] = \infty \) then
  - If \( i = j \) then \( m[i, j] \leftarrow 0 \)
  - Else
    - For \( k \leftarrow i \) to \( j - 1 \) do
      - \( q \leftarrow \text{LookupC}(p, i, k) + \text{LookupC}(p, k+1, j) + p_{i-1}p_kp_j \)
      - If \( q < m[i, j] \) then \( m[i, j] \leftarrow q \)

- Return \( m[i, j] \)

---

**MemoizedMatrixChain(p)**

- \( n \leftarrow \text{length}[p] - 1 \)
- For \( i \leftarrow 1 \) to \( n \) do
  - For \( j \leftarrow 1 \) to \( n \) do
    - \( m[i, j] \leftarrow \infty \)
- Return \( \text{LookupC}(p, 1, n) \)

▷ Shaded subtrees are looked-up rather than recomputing
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

Memoization-based solutions will NOT BE ACCEPTED in the exams!
Dynamic Programming vs Memoization

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
Dynamic Programming vs Memoization 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
Problem 2: Longest Common Subsequence
Definitions

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

- Example:

  \[
  X = < A, B, C, B, D, A, B >
  \]
  \[
  Z = < B, C, D, B >
  \]

  \( \Rightarrow \) Z is a subsequence of X
Definitions

**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 \( x_{i_j} = z_j \) for all \( j = 1, 2, \ldots, k \), where \( 1 \leq k \leq m \).

**Example:** \( Z = \langle B, C, D, B \rangle \) is a subsequence of \( X = \langle A, B, C, B, D, A, B \rangle \) with the index sequence \( \langle i_1, i_2, i_3, i_4 \rangle = \langle 2, 3, 5, 7 \rangle \).
Definitions

If $Z$ is a subsequence of both $X$ and $Y$, we denote $Z$ as a \textit{common subsequence} of $X$ and $Y$.

\textit{Example}: $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$

Sequence $Z = \langle B, C, A \rangle$ is a \textit{common subsequence} of $X$ and $Y$.

What is a \textit{longest common subsequence (LCS)} of $X$ and $Y$?

$\langle B, C, B, A \rangle$
Longest Common Subsequence (LCS) Problem

- **LCS problem**: Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, find the LCS of $X$ & $Y$.

- **Brute force approach**:
  - Enumerate all subsequences of $X$
  - Check if each subsequence is also a subsequence of $Y$
  - Keep track of the LCS
  - What is the complexity?
    - There are $2^m$ subsequences of $X$
    - Exponential runtime
Notation

**Notation**: Let $X_i$ denote the $i^{th}$ prefix of $X$

i.e. $X_i = \langle x_1, x_2, \ldots, x_i \rangle$

**Example**: $X = \langle A, B, C, B, D, A, B \rangle$

$X_4 = \langle A, B, C, B \rangle$, $X_0 = \langle \rangle$
Optimal Substructure of an LCS

Let $X = <x_1, x_2, \ldots, x_m>$ and $Y = <y_1, y_2, \ldots, y_n>$ are given
Let $Z = <z_1, z_2, \ldots, z_k>$ be an LCS of $X$ and $Y$

**Question 1**: If $x_m = y_n$, how to define the optimal substructure?

We must have $z_k = x_m = y_n$ and $Z_{k-1} = \text{LCS}(X_{m-1}, Y_{n-1})$
Optimal Substructure of an LCS

Let \( X = <x_1, x_2, \ldots, x_m> \) and \( Y = <y_1, y_2, \ldots, y_n> \) are given

Let \( Z = <z_1, z_2, \ldots, z_k> \) be an LCS of \( X \) and \( Y \)

**Question 2**: If \( x_m \neq y_n \) and \( z_k \neq x_m \), how to define the optimal substructure?

We must have \( Z = \text{LCS}(X_{m-1}, Y) \)
Optimal Substructure of an LCS

Let $X = <x_1, x_2, \ldots, x_m>$ and $Y = <y_1, y_2, \ldots, y_n>$ be given.

Let $Z = <z_1, z_2, \ldots, z_k>$ be an LCS of $X$ and $Y$.

**Question 3**: If $x_m \neq y_n$ and $z_k \neq y_n$, how to define the optimal substructure?

We must have $Z = \text{LCS}(X, Y_{n-1})$
Theorem: Optimal Substructure of an LCS

Let \( X = <x_1, x_2, \ldots, x_m> \) and \( Y = <y_1, y_2, \ldots, y_n> \) are given
Let \( Z = <z_1, z_2, \ldots, z_k> \) be an LCS of \( X \) and \( Y \)

**Theorem**: Optimal substructure of an LCS:

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 $Y_{n-1}$

\[
X = \begin{array}{cc}
1 & 2 \\
\end{array}, \quad m \\
\]

\[
Y = \begin{array}{cc}
1 & 2 \\
\end{array}, \quad n \\
\]

\[
Z = \begin{array}{cc}
1 & 2 \\
\end{array}, \quad k \\
\]

\[
Z = \begin{array}{cc}
1 & 2 \\
\end{array}, \quad m \\
\]

\[
Y_{n-1} \\
\]

LCS
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)
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
Recursive Algorithm (Inefficient!!)

LCS\((X, Y)\)

\[
m \leftarrow \text{length}[X]
\]

\[
n \leftarrow \text{length}[Y]
\]

if \(x_m = y_n\) then

\[
Z \leftarrow \text{LCS}(X_{m-1}, Y_{n-1}) \quad \triangleright \text{solve one subproblem}
\]

return \(<Z, x_m = y_n>\) \quad \triangleright \text{append } x_m = y_n \text{ to } Z

else

\[
Z' \leftarrow \text{LCS}(X_{m-1}, Y) \quad \triangleright \text{solve two subproblems}
\]
\[
Z'' \leftarrow \text{LCS}(X, Y_{n-1})
\]

return longer of \(Z'\) and \(Z''\)
A Recursive Solution

\[ c[i, j] : \text{length of an LCS of } X_i \text{ and } Y_j \]

\[
c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
 c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\
 \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j 
\end{cases}
\]
Computing the Length of an LCS

- We can easily write an **exponential-time recursive algorithm** based on the given recurrence. ➔ **Inefficient!**

- How many distinct subproblems to solve?
  \[ \Theta(mn) \]

- **Overlapping subproblems property**: Many subproblems share the same sub-subproblems.
  e.g. Finding an LCS to \( X_{m-1} \) & \( Y \) and an LCS to \( X \) & \( Y_{n-1} \) has the sub-subproblem of finding an LCS to \( X_{m-1} \) & \( Y_{n-1} \)

- Therefore, we can use **dynamic programming**.
Data Structures

Let:

\( c[i, j] \): length of an LCS of \( X_i \) and \( Y_j \)

\( b[i, j] \): direction towards the table entry corresponding to the optimal subproblem solution chosen when computing \( c[i, j] \). Used to simplify the construction of an optimal solution at the end.

Maintain the following tables:

\( c[0\ldots m, 0\ldots n] \)

\( b[1\ldots m, 1\ldots n] \)
Bottom-up Computation

Reminder:

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

How to choose the order in which we process \(c[i, j]\) values?

The values for \(c[i-1, j-1]\), \(c[i, j-1]\), and \(c[i-1, j]\) must be computed before computing \(c[i, j]\).
\[ c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
 c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\
\max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j 
\end{cases} \]

Need to process:
\(c[i, j]\)

after computing:
\(c[i-1, j-1], c[i, j-1], c[i-1, j]\)
\[ c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
\max \{c[i, j - 1], c[i - 1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \\
c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j 
\end{cases} \]
Computing the Length of an LCS

LCS-LENGTH($X,Y$)

$m ← \text{length}[X]; n ← \text{length}[Y]$

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

$c[i,j] ← c[i-1,j-1]+1$

$b[i,j] ← “\leftarrow”$

else if $c[i-1,j] ≥ c[i,j-1]$

$c[i,j] ← c[i-1,j]$

$b[i,j] ← “↑”$

else

$c[i,j] ← c[i,j-1]$

$b[i,j] ← “\leftarrow”$

Total runtime = $\Theta(mn)$

Total space = $\Theta(mn)$
Computing the Length of an LCS

Operation of **LCS-LENGTH** on the sequences

\[ X = \langle A, B, C, B, D, A, B \rangle \]
\[ Y = \langle B, D, C, A, B, A \rangle \]
Computing the Length of an LCS

Operation of **LCS-LENGTH** on the sequences

\( X = \langle A, B, C, B, D, A, B \rangle \)
\( Y = \langle B, D, C, A, B, A \rangle \)
### Computing the Length of an LCS

**Operation of LCS-LENGTH on the sequences**

\[ X = \langle A, B, C, B, D, A, B \rangle \]

\[ Y = \langle B, D, C, A, B, A \rangle \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 A</td>
<td>0</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>⇣</td>
<td>1</td>
<td>⇣</td>
</tr>
<tr>
<td>2 B</td>
<td>0</td>
<td>⇣</td>
<td>1</td>
<td>⇣</td>
<td>1</td>
<td>↑</td>
<td>⇣</td>
</tr>
<tr>
<td>3 C</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 D</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Computing the Length of an LCS

Operation of **LCS-LENGTH** on the sequences

\[ X = <A, B, C, B, D, A, B> \]
\[ Y = <B, D, C, A, B, A> \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Computing the Length of an LCS

Operation of **LCS-LENGTH** on the sequences

\[ X = \langle A, B, C, B, D, A, B \rangle \]

\[ Y = \langle B, D, C, A, B, A \rangle \]
Computing the Length of an LCS

Operation of \textbf{LCS-LENGTH} on the sequences

\begin{align*}
X &= \langle A, B, C, B, D, A, B \rangle \\
Y &= \langle B, D, C, A, B, A \rangle
\end{align*}
Computing the Length of an LCS

**Operation of LCS-LENGTH on the sequences**

\[ X = \langle A, B, C, B, D, A, B \rangle \]
\[ Y = \langle B, D, C, A, B, A \rangle \]

<table>
<thead>
<tr>
<th>( i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_i )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 A</td>
<td>0</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>⇐</td>
<td>1</td>
<td>⇐</td>
<td>1</td>
</tr>
<tr>
<td>2 B</td>
<td>0</td>
<td>⇐</td>
<td>⇐</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3 C</td>
<td>0</td>
<td>↑</td>
<td>↑</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4 B</td>
<td>0</td>
<td>⇐</td>
<td>↑</td>
<td>↑</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5 D</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6 A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7 B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

\[ X = \langle A, B, C, B, D, A, B \rangle \]

\[ Y = \langle B, D, C, A, B, A \rangle \]
Computing the Length of an LCS

Operation of **LCS-LENGTH** on the sequences

\[ X = \langle A, B, C, B, D, A, B \rangle \]

\[ Y = \langle B, D, C, A, B, A \rangle \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1 B</th>
<th>2 D</th>
<th>3 C</th>
<th>4 A</th>
<th>5 B</th>
<th>6 A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 x_i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 A</td>
<td>0</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>⇣</td>
<td>⇣</td>
<td>⇣</td>
</tr>
<tr>
<td>2 B</td>
<td>0</td>
<td>⇣</td>
<td>⇣</td>
<td>⇣</td>
<td>↑</td>
<td>⇣</td>
<td>⇣</td>
</tr>
<tr>
<td>3 C</td>
<td>0</td>
<td>↑</td>
<td>↑</td>
<td>⇣</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4 B</td>
<td>0</td>
<td>⇣</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>2</td>
</tr>
<tr>
<td>5 D</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Computing the Length of an LCS

### Operation of **LCS-LENGTH** on the sequences

\( X = \langle A, B, C, B, D, A, B \rangle \)  
\( Y = \langle B, D, C, A, B, A \rangle \)

The table below illustrates the process of computing the length of the LCS using the LCS-LENGTH algorithm.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( j )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_i )</td>
<td>( y_j )</td>
<td>( A )</td>
<td>( B )</td>
<td>( D )</td>
<td>( C )</td>
<td>( A )</td>
<td>( B )</td>
<td>( A )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>( \uparrow )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( \leftarrow )</td>
<td>( \leftarrow )</td>
<td>( \leftarrow )</td>
<td>( \leftarrow )</td>
</tr>
<tr>
<td>2</td>
<td>( \leftarrow )</td>
<td>1</td>
<td>( \leftarrow )</td>
<td>( \leftarrow )</td>
<td>( \uparrow )</td>
<td>( \leftarrow )</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>( \uparrow )</td>
<td>1</td>
<td>1</td>
<td>( \leftarrow )</td>
<td>( \leftarrow )</td>
<td>( \uparrow )</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>( \leftarrow )</td>
<td>1</td>
<td>1</td>
<td>( \uparrow )</td>
<td>( \uparrow )</td>
<td>( \uparrow )</td>
<td>3</td>
<td>( \leftarrow )</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The algorithm iterates through each element of the sequences, comparing them and updating the table accordingly. The length of the LCS is determined by the value in the bottom-right cell of the table, which is 3 in this case.
Computing the Length of an LCS

Operation of **LCS-LENGTH** on the sequences

\[ X = \langle A, B, C, B, D, A, B \rangle \]
\[ Y = \langle B, D, C, A, B, A \rangle \]
Computing the Length of an LCS

Operation of **LCS-LENGTH** on the sequences

\[ X = \langle A, B, C, B, D, A, B \rangle \]
\[ Y = \langle B, D, C, A, B, A \rangle \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0</td>
<td>B</td>
<td>D</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>x_i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>\uparrow</td>
<td>\uparrow</td>
<td>\uparrow</td>
<td>\leftarrow</td>
<td>\leftarrow</td>
<td>\leftarrow</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>\leftrightarrow</td>
<td>\leftarrow</td>
<td>\leftarrow</td>
<td>\uparrow</td>
<td>\leftarrow</td>
<td>\leftarrow</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>\uparrow</td>
<td>\uparrow</td>
<td>\leftarrow</td>
<td>\leftarrow</td>
<td>\uparrow</td>
<td>\uparrow</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>\leftrightarrow</td>
<td>\uparrow</td>
<td>\leftarrow</td>
<td>\uparrow</td>
<td>\leftarrow</td>
<td>\leftarrow</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>\uparrow</td>
<td>\leftarrow</td>
<td>\uparrow</td>
<td>\uparrow</td>
<td>\uparrow</td>
<td>\uparrow</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>\uparrow</td>
<td>\uparrow</td>
<td>\uparrow</td>
<td>\leftarrow</td>
<td>\uparrow</td>
<td>\leftarrow</td>
</tr>
<tr>
<td>7</td>
<td>B</td>
<td>\uparrow</td>
<td>\uparrow</td>
<td>\uparrow</td>
<td>\leftarrow</td>
<td>\uparrow</td>
<td>\leftarrow</td>
</tr>
</tbody>
</table>

\[ j \]

\[ y_j \]

\[ i \]

\[ x_i \]
Computing the Length of an LCS

Operation of LCS-LENGTH on the sequences

\[ X = \langle A, B, C, B, D, A, B \rangle \]

\[ Y = \langle B, D, C, A, B, A \rangle \]

Running-time = \( O(mn) \) since each table entry takes \( O(1) \) time to compute
Computing the Length of an LCS

Operation of **LCS-LENGTH** on the sequences

\[ X = \langle A, B, C, B, D, A, B \rangle \]

\[ Y = \langle B, D, C, A, B, A \rangle \]

Running-time = \( O(mn) \)
since each table entry takes \( O(1) \) time to compute

LCS of \( X \) & \( Y \) = \( \langle B, C, B, A \rangle \)
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 \(b[i, j] = \text{"\textless\textgreater}\) then
  PRINT-LCS\((b, X, i-1, j-1)\)
  print \(x_i\)

else if \(b[i, j] = \text{"\textuparrow\textdownarrow}\) then
  PRINT-LCS\((b, X, i-1, j)\)

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

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 decremented in each stage of the recursion
Do we really need the b table (back-pointers)?

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>D</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>ø</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Question:** From which neighbor did we expand to the highlighted cell?

**Answer:** Upper-left neighbor, because $X[i] = Y[j]$. 
Do we really need the b table (back-pointers)?

<table>
<thead>
<tr>
<th></th>
<th>∅</th>
<th>B</th>
<th>D</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Question:** From which neighbor did we expand to the highlighted cell?

**Answer:** Left neighbor, because $X[i] \neq Y[j]$ and $LCS[i, j-1] > LCS[i-1, j]$. 
Do we really need the b table (back-pointers)?

<table>
<thead>
<tr>
<th></th>
<th>∅</th>
<th>B</th>
<th>D</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Question:** From which neighbor did we expand to the highlighted cell?

**Answer:** Upper neighbor, because X[i] ≠ Y[j] and LCS[i, j-1] = LCS[i-1, j].

*(See pseudo-code to see how ties are handled.)*
Improving the Space Requirements

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
What if we store the last 2 rows only?

<table>
<thead>
<tr>
<th></th>
<th>∅</th>
<th>B</th>
<th>D</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To compute $c[i, j]$, we only need $c[i-1, j-1]$, $c[i-1, j]$, and $c[i-1, j-1]$. So, we can store only the last two rows.
What if we store the last 2 rows only?

To compute $c[i, j]$, we only need $c[i-1, j-1]$, $c[i-1, j]$, and $c[i-1, j-1]$. So, we can store only the last two rows.
What if we store the last 2 rows only?

To compute $c[i, j]$, we only need $c[i-1, j-1]$, $c[i-1, j]$, and $c[i-1, j-1]$. So, we can store only the last two rows.

This reduces space complexity from $\Theta(mn)$ to $\Theta(n)$.

Is there a problem with this approach?
What if we store the last 2 rows only?

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>D</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ø</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is there a problem with this approach?

We cannot construct the optimal solution because we cannot backtrace anymore.

This approach works if we only need the length of an LCS, not the actual LCS.