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.

\[
\bigg( A_i (A_{i+1} A_{i+2} \ldots A_j) \bigg) \\
\bigg( (A_i A_{i+1} A_{i+2} \ldots A_{j-1}) A_j \bigg) \\
\bigg( (A_i A_{i+1} A_{i+2} \ldots A_k)(A_{k+1} A_{k+2} \ldots A_j) \bigg) \quad \text{for } i \leq k < j
\]

- 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
  
  \[
  \begin{align*}
  & (A_1(A_2(A_3A_4))) \\
  & (A_1(((A_2A_3)A_4)) \\
  & ((A_1A_2)(A_3A_4)) \\
  & ((A_1(A_2A_3))A_4) \\
  & (((A_1A_2)A_3)A_4)
  \end{align*}
  \]

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

Matrix has two attributes

- \textbf{rows}[A]: \# of rows
- \textbf{cols}[A]: \# of columns

\# of scalar mult-adds in

\(\leftarrow\) \(AB\) is \(\text{rows}[A] \times \text{cols}[B] \times \text{cols}[A]\)

\(A: (p \times q)\) \(C = A \cdot B\) is \(p \times r\).

\(B: (q \times r)\)

\# of mult-adds is \(p \times r \times q\)

\begin{algorithm}
\begin{algorithmic}
\Procedure{MATRIX-MULTIPLY}{A, B}
\If {\text{cols}[A] \neq \text{rows}[B]}
\State \textbf{error} ("incompatible dimensions")
\EndIf
\For {\(i \leftarrow 1\) \text{ to } \text{rows}[A]}
\For {\(j \leftarrow 1\) \text{ to } \text{cols}[B]}
\State \(C[i,j] \leftarrow 0\)
\EndFor
\EndFor
\For {\(k \leftarrow 1\) \text{ to } \text{cols}[A]}
\State \(C[i,j] \leftarrow C[i,j] + A[i,k] \cdot B[k,j]\)
\EndFor
\State \Return \(C\)
\EndProcedure
\end{algorithmic}
\end{algorithm}
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 \( A_1 \cdot A_2 \cdots \cdot A_n \) such that the number of scalar mult-adds are minimized.

- **Ex.:** \( \langle A_1, A_2, A_3 \rangle \) where \( A_1: 10 \times 100; A_2: 100 \times 5; A_3: 5 \times 50 \)

\[
\begin{array}{c|c|c}
& 10 \times 100 \times 5 & 10 \times 5 \times 50 \\
\hline
\begin{array}{c}
(\langle A_1 A_2 \rangle A_3) \\
10 \times 10 \\
A_1 A_2
\end{array}
& \begin{array}{c}
10 \times 5 \times 50 \\
A_1 A_2
\end{array}
& \begin{array}{c}
0 \\
0
\end{array}
\end{array}
\]

\[
\begin{array}{c|c|c}
& 100 \times 5 \times 50 & 10 \times 100 \times 50 \\
\hline
\begin{array}{c}
(A_1 (A_2 A_3)) \\
10 \times 10 \\
A_2 A_3
\end{array}
& \begin{array}{c}
100 \times 5 \times 50 \\
A_2 A_3
\end{array}
& \begin{array}{c}
0 \\
0
\end{array}
\end{array}
\]

\( \Rightarrow \) 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 $k$th and $(k+1)$st matrices for any $k=1, 2, \ldots, n-1$, then parenthesize the two resulting sequences independently, i.e.,
  $$(A_1A_2A_3 \ldots A_k)(A_{k+1}A_{k+2} \ldots A_n)$$

- 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 \textbf{Catalan Numbers}
- Solution is \( P(n) = C(n-1) \) where

\[
C(n) = \frac{1}{n+1} \binom{2n}{n} = \Omega\left(\frac{4^n}{n^{3/2}}\right)
\]

- 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 parentheses split 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)

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

  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

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

- 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
  \[ m_{ij} = m_{ik} + m_{k+1,j} + p_{i \rightarrow p_k p_j} \]
  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 \( 1 \leq i \leq j \leq n \)
  - 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 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

\textbf{MATRIX-CHAIN-ORDER}(p)

\[ n \leftarrow \text{length}[p] - 1 \]

\text{for } i \leftarrow 1 \text{ to } n \text{ do}

\[ m[i, i] \leftarrow 0 \]

\text{for } \ell \leftarrow 2 \text{ to } n \text{ do}

\text{for } i \leftarrow 1 \text{ to } n - \ell + 1 \text{ do}

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

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

\text{for } k \leftarrow i \text{ to } j-1 \text{ do}

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

\text{if } q < m[i, j] \text{ then}

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

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

\text{return } m \text{ 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 \( \ell \)

- 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 \( i \leq k < i+\ell-1 \), 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
  
  \( \cdots \)

\( \ell = 3 \)
for \( i = 1 \) to \( n - 2 \)
\[ m[i, i+2] = \infty \]
for \( k = i \) to \( i+1 \) do
  
  \( \cdots \)

\( \ell = 4 \)
for \( i = 1 \) to \( n - 3 \)
\[ m[i, i+3] = \infty \]
for \( k = i \) to \( i+2 \) do
  
  \( \cdots \)
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 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

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

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

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 $\ell = j-i+1$

\[
\begin{align*}
(A_i A_{i+1} A_{i+2}) (A_{i+3} \ldots A_j))
\end{align*}
\]

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
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

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

```plaintext
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
    if $j > i$ then
        $X \leftarrow \text{MATRIX-CHAIN-MULTIPLY}(A, s, i, s[i,j])$
        $Y \leftarrow \text{MATRIX-CHAIN-MULTIPLY}(A, s, s[i,j]+1, j)$
        return $\text{MATRIX-MULTIPLY}(X, Y)$
    else
        return $A_i$
```

Invocation: `MATRIX-CHAIN-MULTIPLY(A, s, 1, n)`
Example: Recursive Construction of an Optimal Solution

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

\[
\begin{array}{ccccccc}
1 & 1 & 1 & 3 & 3 & 3 \\
2 & 2 & 3 & 4 & 3 & 3 \\
3 & 3 & 3 & 3 & 3 & 3 \\
\end{array}
\]

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

\[
\begin{array}{ccccccc}
2 & 3 & 4 & 5 & 6 \\
1 & 1 & 3 & 3 & 3 \\
2 & 2 & 3 & 4 & 3 \\
3 & 3 & 3 & 3 & 3 \\
\end{array}
\]

MCM(1,6)
\[
X \leftarrow \text{MCM}(1,3) = (A_1A_2A_3)
\]
\[
Y \leftarrow \text{MCM}(4,6) = (A_4A_5A_6)
\]

return (?)

MCM(1,6)
\[
X \leftarrow \text{MCM}(1,3) = (A_1A_2A_3)
\]
\[
Y \leftarrow \text{MCM}(2,3) = (A_2A_3)
\]

return (?)

return A_1
Example: Recursive Construction of an Optimal Solution

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

**MCM(1,6)**
- \(X \leftarrow \text{MCM}(1,3) = (A_1(A_2A_3))\)
- \(Y \leftarrow \text{MCM}(4,6) = (A_4A_5A_6)\)
- Return (?)

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

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

**MCM(2,2)**
- \(X \leftarrow \text{MCM}(2,2) = A_2\)
- Return \(A_2\)

**MCM(3,3)**
- \(X \leftarrow \text{MCM}(3,3) = A_3\)
- Return \(A_3\)

**MCM(2,3)**
- \(X \leftarrow \text{MCM}(2,2) = A_2\)
- \(Y \leftarrow \text{MCM}(3,3) = A_3\)
- Return \((A_2A_3)\)
Example: Recursive Construction of an Optimal Solution

\[
\begin{array}{cccccc}
1 & 1 & 1 & 3 & 3 & 3 \\
2 & 2 & 3 & 4 & 3 & 3 \\
3 & 3 & 3 & 3 & 3 & 3 \\
\end{array}
\]

\[s[1...6, 1...6] = \begin{bmatrix} 4 & 4 & 5 \\ 5 & 5 & \end{bmatrix} \]

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

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

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

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

- \[MCM(4,5)\]
  \[X \leftarrow MCM(4,4) = A_4\]
  \[\text{return } A_4\]

- \[MCM(5,5)\]
  \[X \leftarrow MCM(5,5) = A_5\]
  \[\text{return } A_5\]

- \[MCM(6,6)\]
  \[Y \leftarrow MCM(5,5) = A_5\]
  \[\text{return } (A_4A_5)\]

- \[Y \leftarrow MCM(6,6) = A_6\]
  \[\text{return } A_6\]
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 $A_1 A_2 \ldots A_n$ 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
    \[\langle A_i, A_{i+1}, \ldots , A_j \rangle\text{ for } 1 \leq i \leq j \leq n\]

  – 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

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

\[
\begin{align*}
\text{if } i &= j \text{ then} \\
&\quad \text{return 0} \\
\end{align*}
\]

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

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

\[ = 1 \]

• 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

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

• 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 $n \geq 2$

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

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

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

$\Rightarrow T(n) \geq 2^{n-1}$ \hspace{1cm} 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**

\[
\text{lookupC}(p, i, j) \\
\text{if } m[i, j] = \infty \text{ then} \\
\quad \text{if } i = j \text{ then} \\
\quad \quad m[i, j] \leftarrow 0 \\
\quad \text{else} \\
\quad \quad \text{for } k \leftarrow i \text{ to } j - 1 \text{ do} \\
\quad \quad \quad q \leftarrow \text{lookupC}(p, i, k) + \text{lookupC}(p, k+1, j) + p_{i-1} p_k p_j \\
\quad \quad \quad \text{if } q < m[i, j] \text{ then} \\
\quad \quad \quad \quad m[i, j] \leftarrow q \\
\quad \quad \quad \text{return } m[i, j]
\]

\[
\text{memoizedMatrixChain}(p) \\
\quad n \leftarrow \text{length}[p] - 1 \\
\quad \text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
\quad \quad \quad \text{for } j \leftarrow 1 \text{ to } n \text{ do} \\
\quad \quad \quad \quad m[i, j] \leftarrow \infty \\
\quad \quad \quad \text{return } \text{lookupC}(p, 1, n)
\]

\[\text{Shaded subtrees are looked-up rather than recomputing}\]
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 solution 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 there 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 $\langle i_1, i_2, i_3, i_4 \rangle = \langle 2, 3, 5, 7 \rangle$. 
Longest Common Subsequence (LCS)

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

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

Longest common subsequence (LCS):
Given two sequences $X = <x_1, x_2, \ldots, x_m>$ and $Y = <y_1, y_2, \ldots, y_n>$
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

$$X_i = <x_1, x_2, \ldots, x_i>$$

Example: Given $X = <A, B, C, B, D, A, B>$

$X_4 = <A, B, C, B>$ and $X_{\emptyset} =$ empty sequence

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 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 \( Y_{n-1} \)

\[
LCS
\]

\[
Z_{k-1}
\]

\[
X_{m-1}
\]

\[
Y_{n-1}
\]
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}{c}
1 & 2 \\
\end{array}
\quad C
\quad \begin{array}{c}
m \\
\end{array}

Y = \begin{array}{c}
1 & 2 \\
\end{array}
\quad D
\quad \begin{array}{c}
n \\
\end{array}

Z = \begin{array}{c}
1 & 2 \\
\end{array}
\quad \begin{array}{c}
k \\
\end{array}

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)
Longest Common Subsequence Algorithm

LCS($X$, $Y$)

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

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

if $x_m = y_n$ then

$Z \leftarrow \text{LCS}(X_{m-1}, Y_{n-1})$ ▷ solve one subproblem

return $<Z, x_m = y_n>$ ▷ append $x_m = y_n$ to $Z$

else

$Z' \leftarrow \text{LCS}(X_{m-1}, Y)$ ▷ solve two subproblems

$Z'' \leftarrow \text{LCS}(X, Y_{n-1})$

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

\textbf{LCS-LENGTH}(X,Y) \\
m \leftarrow \text{length}[X]; \ n \leftarrow \text{length}[Y] \\
\text{for } i \leftarrow 0 \text{ to } m \text{ do } c[i, 0] \leftarrow 0 \\
\text{for } j \leftarrow 0 \text{ to } n \text{ do } c[0, j] \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ to } m \text{ do} \\
\text{\quad for } j \leftarrow 1 \text{ to } n \text{ do} \\
\text{\quad\quad if } x_i = y_j \text{ then} \\
\text{\quad\quad\quad } c[i, j] \leftarrow c[i-1, j-1]+1 \\
\text{\quad\quad\quad } b[i, j] \leftarrow “\Box” \\
\text{\quad\quad else if } c[i-1, j] \geq c[i, j-1] \\
\text{\quad\quad\quad } c[i, j] \leftarrow c[i-1, j] \\
\text{\quad\quad\quad } b[i, j] \leftarrow “\Box” \\
\text{\quad\quad else} \\
\text{\quad\quad\quad } c[i, j] \leftarrow c[i, j-1] \\
\text{\quad\quad\quad } b[i, j] \leftarrow “\Box”
Computing the Length of an LCS

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

<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>
</tr>
</thead>
<tbody>
<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>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

1 2 3 4 5 6 7
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

\[ X = <A, B, C, B, D, A, B> \]
\[ Y = <B, D, C, A, B, A> \]
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></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>
</tr>
<tr>
<td>1 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>2 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>3 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>4 B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5 D</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 A</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>
</tr>
</tbody>
</table>
Computing the Length of an LCS

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

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

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

<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>
</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>( \uparrow )</td>
<td>( \uparrow )</td>
<td>( \uparrow )</td>
<td>( \leftarrow )</td>
<td>1</td>
<td>( \leftarrow )</td>
<td>1</td>
</tr>
<tr>
<td>2 B</td>
<td>( \leftarrow )</td>
<td>( \leftarrow )</td>
<td>( \uparrow )</td>
<td>( \uparrow )</td>
<td>1</td>
<td>2</td>
<td>( \leftarrow )</td>
</tr>
<tr>
<td>3 C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>( \leftarrow )</td>
<td>2</td>
<td>( \uparrow )</td>
</tr>
<tr>
<td>4 B</td>
<td>( \leftarrow )</td>
<td>( \uparrow )</td>
<td>( \uparrow )</td>
<td>( \uparrow )</td>
<td>2</td>
<td>( \uparrow )</td>
<td>( \uparrow )</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

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

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

| \( x_i \) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 A| 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 2 B| 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| 3 C| 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| 4 B| 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| 5 D| 0 | 0 | 0 | 0 |
| 6 A| 0 | 0 | 0 | 0 |
| 7 B| 0 | 0 | 0 | 0 |
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>i</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>2</td>
<td>2</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>3</td>
<td>3</td>
</tr>
<tr>
<td>4</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>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>

```
X = <A, B, C, B, D, A, B>  
Y = <B, D, C, A, B, A>
```
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>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>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>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</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>2 B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3 C</td>
<td>0</td>
<td>1</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>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4 B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</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>
<td>0</td>
</tr>
<tr>
<td>5 D</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>6</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>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</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>

\[ X = \langle A, B, C, B, D, A, B \rangle \]
\[ Y = \langle B, D, C, A, B, A \rangle \]
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>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 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>3 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>4 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>5 D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</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 \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*}

\begin{table}[h]
\begin{tabular}{c|cccccc}
\hline
\textit{j} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
\textit{i} & & & & & & & \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & A & & & & & & \\
2 & B & & & & & & \\
3 & C & & & & & & \\
4 & B & & & & & & \\
5 & D & & & & & & \\
6 & A & & & & & & \\
7 & B & & & & & & \\
\hline
\end{tabular}
\end{table}
### Computing the Length of an LCS

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

\[
\begin{align*}
X &= \langle A, B, C, B, D, A, B \rangle \\
Y &= \langle B, D, C, A, B, A \rangle
\end{align*}
\]

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

**LCS of** \(X \& Y = \langle B, C, B, A \rangle \)

<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>
</tr>
</thead>
<tbody>
<tr>
<td>(x_i)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>(y_j)</td>
<td>B</td>
<td>D</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

<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>
</tr>
</thead>
<tbody>
<tr>
<td>(j)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>(y_j)</td>
<td>B</td>
<td>D</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

Table entries:

- \(\uparrow\) indicates a match
- \(\leftarrow\) indicates moving left
- \(\uparrow\) indicates moving up
- \(\uparrow\) indicates moving diagonally

LCS of \(X \& Y = \langle B, C, B, A \rangle\)
### 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>

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

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

The $b$ table returned by \texttt{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 “$\Box$” 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\((b, X, i, j)\)

\[
\text{if } i = 0 \text{ or } j = 0 \text{ then return }
\]
\[
\text{if } b[i, j] = "\boxempty" \text{ then }
\]
\[
\text{PRINT-LCS}(b, X, i-1, j-1)
\]
\[
\text{print } x_i
\]
\[
\text{else if } b[i, j] = "\boxempty" \text{ then }
\]
\[
\text{PRINT-LCS}(b, X, i-1, j)
\]
\[
\text{else }
\]
\[
\text{PRINT-LCS}(b, X, i, j-1)
\]

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 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 \( \text{LCS-LENGTH} \)
- 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.