Dynamic Programming

Longest Common Subsequence

Spring 2022

Introduction

- Biological applications often need to compare the DNA of two (or more) different organisms.
- A strand of DNA consists of a string of molecules called bases
- Where the possible bases are **adenine, guanine, cytosine, and thymine**.
- Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the finite set **{A; C; G; T}**.
- For example, the DNA of one organism may be **S1=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, and the DNA of another organism may be S2= GTCGTTCGGAATGCCGTTGCTCTGTAAA.**

Introduction

- One reason to compare two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are.
- We can, and do, define similarity in many different ways.
- For example, we can say that two DNA strands are similar if one is a substring of the other.

Longest-Common-Subsequence Problem

- Another way to measure the similarity of strands S1 and S2 is by finding a third strand S3 in which the bases in S3 appear in each of S1 and S2;
- These bases must appear in the **same order, but not necessarily consecutively.**
- **The longer the strand S3 we can find, the more similar S1 and S2 are**.
- In our example, the longest strand S3 is GTCGTCGGAAGCCGGCCGAA.
- **Longest-Common-Subsequence Problem**

Dynamic programming

Design technique, like divide-and-conquer.

Example: ***Longest Common Subsequence (LCS)***

- Given two sequences $x[1 \,.\,.\, m]$ and $y[1 \,.\,.\, n]$, find a longest subsequence common to them both.

Dynamic programming

Design technique, like divide-and-conquer.

Example: ***Longest Common Subsequence (LCS)***

- Given two sequences $x[1\,.\,.\,m]$ and $y[1\,.\,.\,n]$, find a longest subsequence common to them both.

"a" *not* "the"

DYNAMIC PROGRAMMING

Design technique, like divide-and-conquer.

Example: ***Longest Common Subsequence (LCS)***

- Given two sequences $x[1 \,.\,.\, m]$ and $y[1 \,.\,.\, n]$, find a longest subsequence common to them both.

"a" *not* "the"

x: A B C B D A B

y: B D C A B A

Dynamic programming

Design technique, like divide-and-conquer.

Example: ***Longest Common Subsequence (LCS)***

- Given two sequences $x[1 \,.\,.\, m]$ and $y[1 \,.\,.\, n]$, find a longest subsequence common to them both.

"a" *not* "the"

$$
\begin{array}{lccccccc}
x: & A & B & C & B & D & A & B \\
y: & B & D & C & A & B & A &
\end{array}
\Bigg\} \quad \text{BCBA} = \text{LCS}(x, y)
$$

functional notation, but not a function

Example

X : ABCDEFGHIJ

Y : CDGI

Z :CDGI

Example

X : ABCDEFGHIJ

Y : FCDHJ

Z :FHJ

X : ABCDEFGHIJ

Y : FCDHJ

Z :CDHJ

LCS

Example

X : ABDACE

Y : BABCE

Z :BACE

X : ABDACE

Y : BABCE

Z :ABCE

LCS

Brute-force LCS algorithm

- In a brute-force approach to solving the LCS problem, we would enumerate **all subsequences of X** and check each subsequence to see whether it is also a **subsequence of Y** ,
- Keeping track of the longest subsequence we find. Each subsequence of X corresponds to a subset of the indices {1, 2 , . . ., m} of X.
- Because X has 2^m subsequences,
- This approach requires exponential time, making it impractical for long sequences.

Brute-force LCS algorithm

Check every subsequence of $x[1 \,..\, m]$ to see if it is also a subsequence of $y[1 \,..\, n]$.

Brute-force LCS algorithm

Check every subsequence of $x[1 \,.\,.\, m]$ to see if it is also a subsequence of $y[1 \,.\,.\, n]$.

Analysis

- Checking $= O(n)$ time per subsequence.
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time $= O(n2^m)$

$= \text{exponential time.}$

Towards a better algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

TOWARDS A BETTER ALGORITHM

Simplification:

- The LCS problem has an optimal-substructure property.
- The natural classes of subproblems correspond to pairs of “prefixes” of the two input sequences
- Given a sequence $X = <x_1, x_2, . . . , x_m>$, we define the i^{th} ***prefix*** of X,
- for i = 0, 1, . .., m as $X_i = <x_1, x_2, . . . , x_m>$
- For example, if **$X = <A, B, C, B, D, A, B>$,** then **$X_4 = <A,B,C,B>$** and X_0 is the empty sequence.

Optimal substructure -LCS

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

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

Recursive Solution

- We should examine either one or two subproblems when finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$, and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, .

If $x_m = y_n$, we must find an LCS of X_{m-1} and Y_{n-1}

If $\mathbf{x_m \neq y_n}$, then we must solve two subproblems:

Finding an LCS of $\mathbf{X_{m-1}}$ and $\mathbf{Y}$
and finding an LCS of $\mathbf{X}$ and $\mathbf{Y_{n-1}}$

Whichever of these two LCSs is longer is an LCS of X and Y .

- these cases exhaust all possibilities,
- we know that one of the optimal subproblem solutions must appear within an LCS of X and Y .

Longest Common Subsequence

- Look at example

C C G C T T

A C G G A T

Longest Common Subsequence

- Look at example

C C G C T T

A C G G A T

Last letter of both strings identical
What to do??

Longest Common Subsequence

- Look at example

C C G C T T

A C G G A T

Last letter of both strings identical:
Recurse on LCS(5,5)

Solution here?

Longest Common Subsequence

- Look at example

C C G C T T

A C G G A T

Last letter of both strings identical:
Recurse on LCS(5,5)

Solution here?
LCS(6,6) = LCS(5,5) + 1 = … 3
CCGCT T
ACGGA T

Longest Common Subsequence

- Look at example

C C G C T C

A C G G A T

Last letter of both strings different:
What to do??

Longest Common Subsequence

- Look at example

C C G C T C   C C G C T C

A C G G A T   A C G G A T

Last letter of both strings different:

LCS[6,6] = max(LCS[5,6], LCS(6,5]) = … 3

CCGCT	CCGCTC
ACGGAT	ACGGA

Longest Common Subsequence

- Look at example

$\boxed{\text{C C G C T}}$ C $\boxed{\text{C C G C T C}}$

$\boxed{\text{A C G G A T}}$ $\boxed{\text{A C G G A}}$ T

Last letter of both strings different:

LCS[6,6] = max(LCS[5,6], LCS(6,5]) = … 3

CCGCT	CCGCTC
ACGGAT	ACGGA
= 3 CGT	= 2 CG

Overlapping-Subproblem - LCS

- We can readily see the overlapping-subproblems property in the LCS problem.
- To find an LCS of X and Y , we may need to find the LCSs of $\mathbf{X}$ and $\mathbf{Y_{n-1}}$ and of $\mathbf{X_{m-1}}$ and $\mathbf{Y}$.
- But each of these subproblems has the subsubproblem of finding an LCS of $\mathbf{X_{m-1}}$ and $\mathbf{Y_{n-1}}$
- Many other subproblems share subsubproblems.

Towards a better algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|s|$.

Towards a better algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|\, s\,|$.

the length of an LCS of the sequences X_i and Y_j.

Strategy: Consider ***prefixes*** of x and y.

- Define $c[i, j] = |\,\mathrm{LCS}(x[1\,.\,.\,i], y[1\,.\,.\,j])\,|$.
- Then, $c[m, n] = |\,\mathrm{LCS}(x, y)\,|$.

Recursive formulation

Theorem.

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

The optimal substructure of the LCS problem gives the recursive formula

Recursive formulation

Theorem.

$$c[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

When $x_i = y_j$, we can and should consider the subproblem of finding an LCS of X_{i-1} and Y_{j-1}.

Otherwise, we instead consider the two subproblems of finding an LCS of X_i and Y_{j-1} and of X_{i-1} and Y_j.

DYNAMIC-PROGRAMMING HALLMARK #1

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

DYNAMIC-PROGRAMMING HALLMARK #1

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If $z = \mathrm{LCS}(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

Recursive algorithm for LCS

LCS(x, y, i, j) // ignoring base cases

$$
\begin{array}{l}
\textbf{if } x[i] = y[j] \\
\quad \textbf{then } c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1 \\
\quad \textbf{else } c[i, j] \leftarrow \max\{\text{LCS}(x, y, i-1, j), \\
\qquad\qquad\qquad\qquad\quad \text{LCS}(x, y, i, j-1)\} \\
\textbf{return } c[i, j]
\end{array}
$$

33

Recursive algorithm for LCS

LCS(x, y, i, j) // ignoring base cases

$\quad$ **if** $x[i] = y[j]$

$\quad\quad$ **then** $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$

$\quad\quad$ **else** $c[i, j] \leftarrow \max\{LCS(x, y, i-1, j)$,

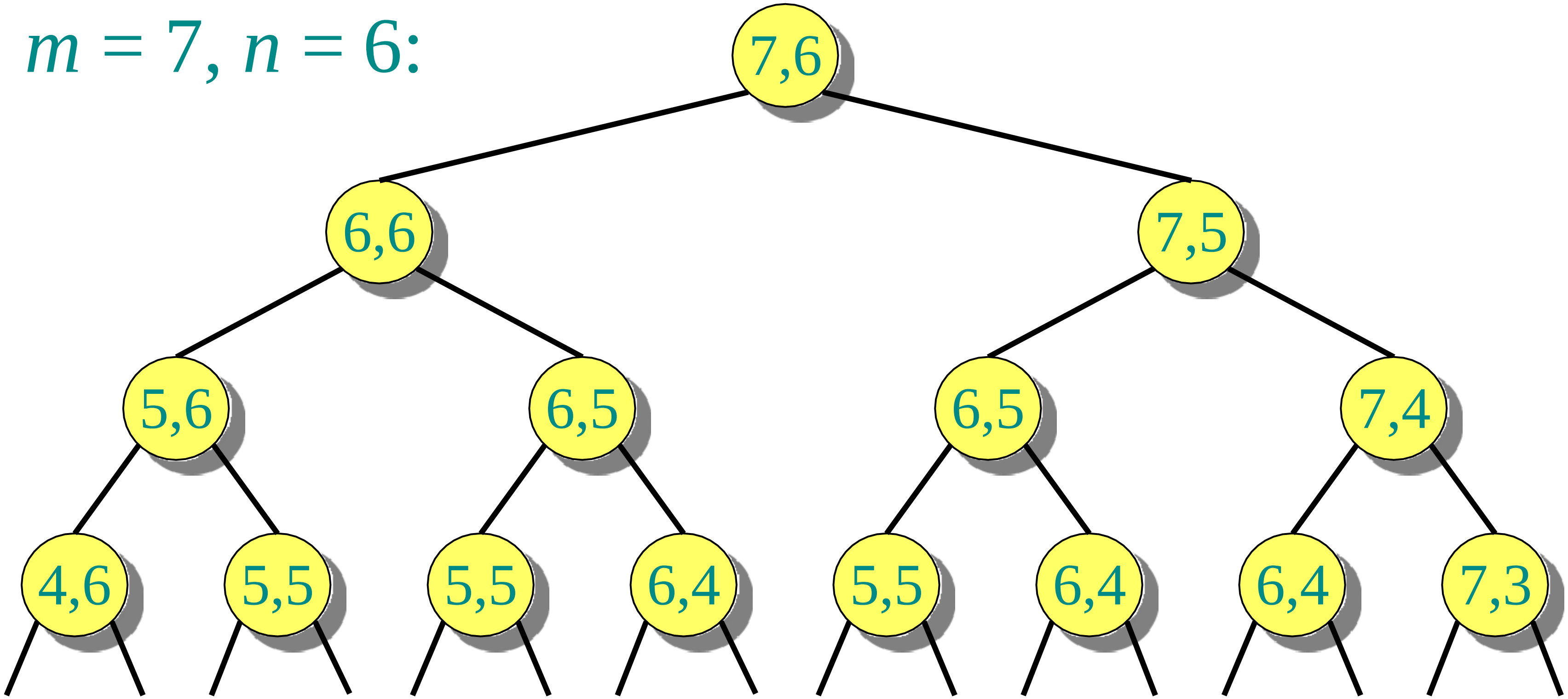
$\quad\quad\quad\quad$ LCS$(x, y, i, j-1)\}$

$\quad$ **return** $c[i, j]$

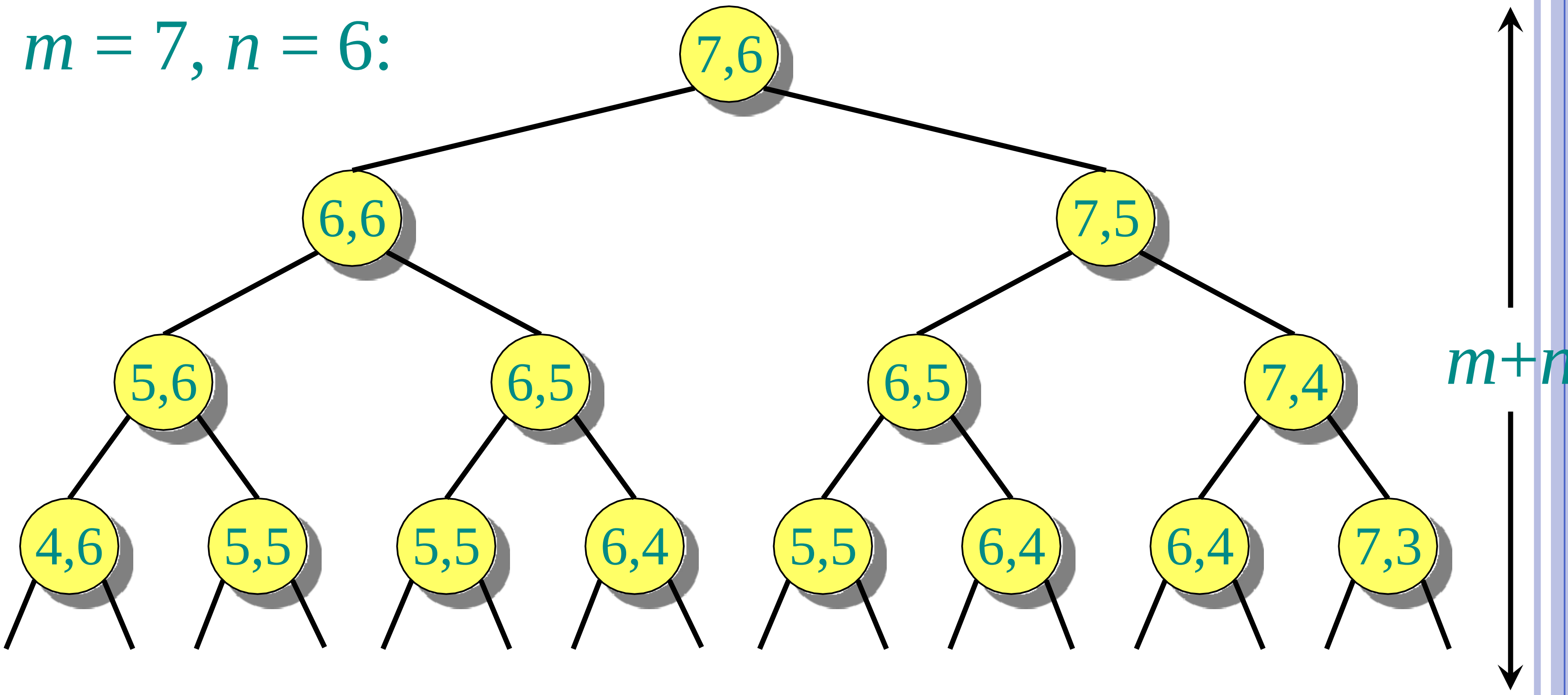
Worse case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion tree

$m = 7, n = 6$:

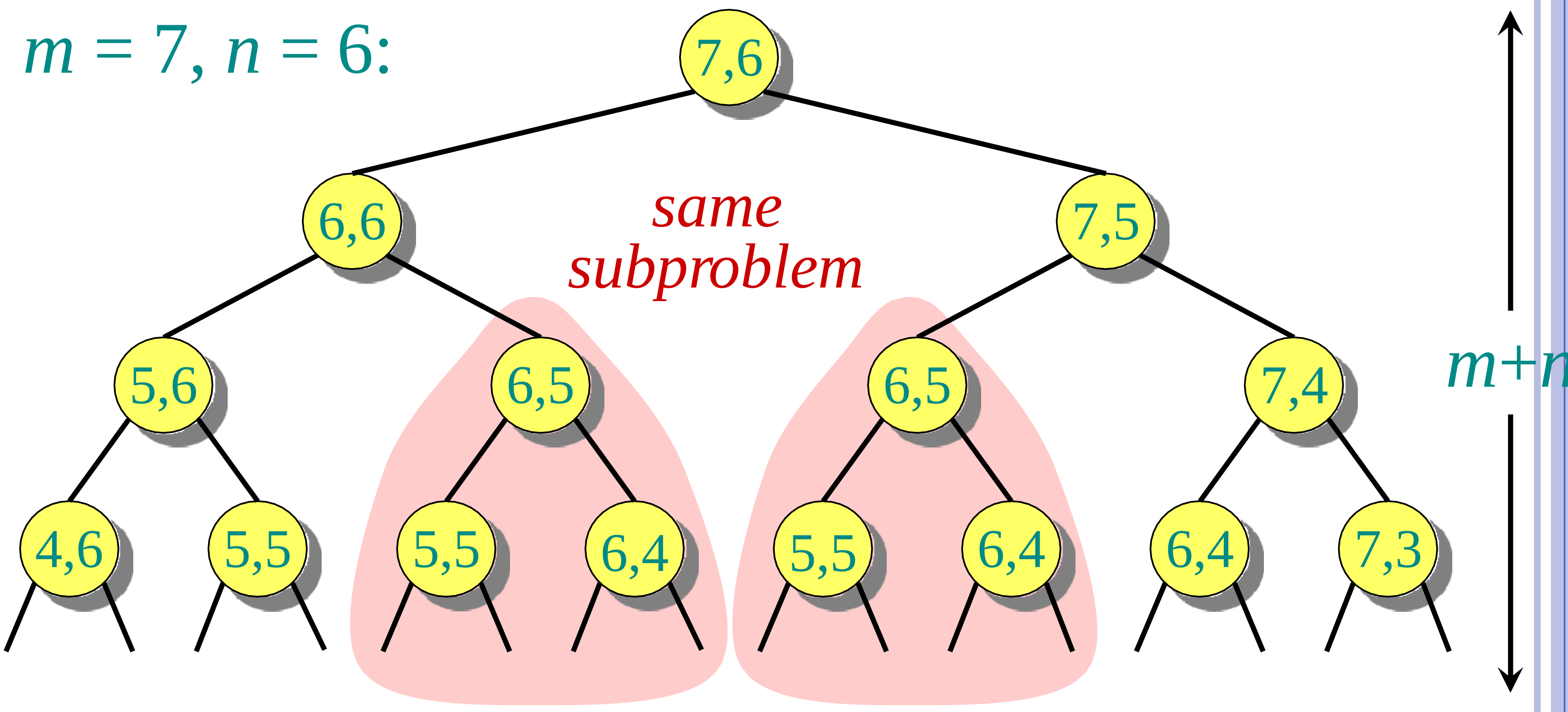


Recursion tree



Height $= m + n \Rightarrow$ work potentially exponential.

RECURSION TREE



Height $= m + n \Rightarrow$ work potentially exponential.,
but we're solving subproblems already solved!

DYNAMIC-PROGRAMMING HALLMARK #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

DYNAMIC-PROGRAMMING HALLMARK #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths m and n is only mn.

Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

LCS(x, y, i, j)
    if $c[i, j] = \mathrm{NIL}$
        then if $x[i] = y[j]$
            then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$
            else $c[i, j] \leftarrow \max\{LCS(x, y, i-1, j)$, LCS$(x, y, i, j-1)\}$

same as before

Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

LCS(x, y, i, j)
    if $c[i, j] = \text{NIL}$
        then if $x[i] = y[j]$
            then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$
            else $c[i, j] \leftarrow \max\{LCS(x, y, i-1, j)$, LCS$(x, y, i, j-1)\}$

same as before

Time $= \Theta(mn) =$ constant work per table entry.
Space $= \Theta(mn)$.

Recursive formulation

Theorem.

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

When $x_i = y_j$, we can and should consider the subproblem of finding an LCS of X_{i-1} and Y_{j-1}.

Otherwise, we instead consider the two subproblems of finding an LCS of X_i and Y_{j-1} and of X_{i-1} and Y_j.

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

max

$\max\{c[i-1, j],\ c[i, j-1]\}$

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0						
D	0							
C	0							
A	0							
B	0							
A	0							

Dynamic-programming algorithm

$c[i, j] \leftarrow \mathbf{LCS}(x, y, i-1, j-1) + 1$

Idea:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1					
D	0							
C	0							
A	0							
B	0							
A	0							

45

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

max

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1				
D	0							
C	0							
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	**B**	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1			
D	0							
C	0							
A	0							
B	0							
A	0							

$c[i, j] \leftarrow \mathrm{LCS}(x, y, i-1, j-1) + 1$

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

max

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1		
D	0							
C	0							
A	0							
B	0							
A	0							

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

max

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	
D	0							
C	0							
A	0							
B	0							
A	0							

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0							
C	0							
A	0							
B	0							
A	0							

$c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0							
C	0							
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0						
C	0							
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1					
C	0							
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1				
C	0							
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1			
C	0							
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2		
C	0							
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	
C	0							
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0							
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0							
A	0							
B	0							
A	0							

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0						
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1					
A	0							
B	0							
A	0							

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2				
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2			
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2		
A	0							
B	0							
A	0							

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0							
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0							
B	0							
A	0							

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1						
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1					
B	0							
A	0							

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2				
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2			
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2		
B	0							
A	0							

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0							
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1						
A	0							

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2					
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2				
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3			
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3		
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0							

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1						

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2					

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2				

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3			

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3		

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

Time $= \Theta(mn)$.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

LCS-LENGTH(X, Y)

```
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

The procedure takes time O(m +n),
since it decrements at least one of i and j in each recursive call.

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

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

Time $= \Theta(mn)$.

Reconstruct LCS by tracing backwards.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

DYNAMIC-PROGRAMMING ALGORITHM

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Dynamic-programming algorithm

Idea:

Compute the table bottom-up.

Time $= \Theta(mn)$.

Reconstruct LCS by tracing backwards.

Space $= \Theta(mn)$.

Exercise: $O(\min\{m, n\})$.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Reference

- **Introduction to Algorithms**
  - Thomas H. Cormen
  - Chapter # 15

- http://lcs-demo.sourceforge.net/