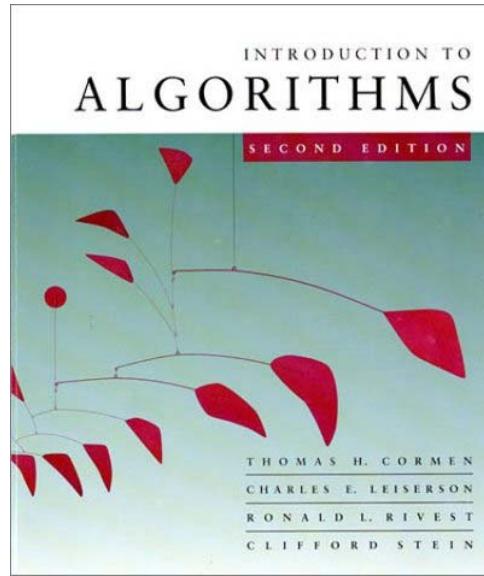
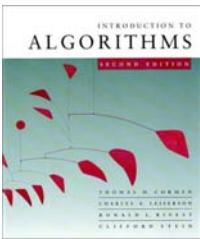


CS 3343 – Fall 2007



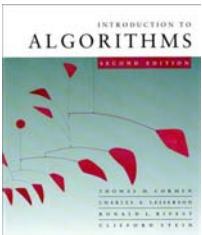
Dynamic Programming Carola Wenk

Slides courtesy of Charles Leiserson with changes
and additions by Carola Wenk



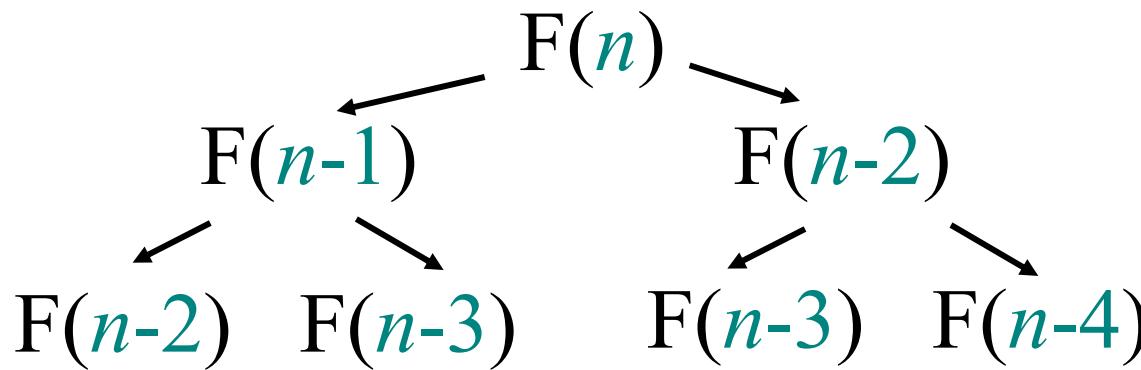
Dynamic programming

- Algorithm design technique (like divide and conquer)
- Is a technique for solving problems that have
 - overlapping subproblems
 - and, when used for optimization, have an optimal substructure property
- **Idea:** Do not repeatedly solve the same subproblems, but solve them only once and store the solutions in a **dynamic programming table**



Example: Fibonacci numbers

- $F(0)=0$; $F(1)=1$; $F(n)=F(n-1)+F(n-2)$ for $n \geq 2$
- Implement this recursion naively:



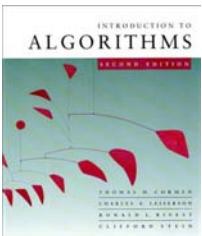
Solve same
subproblems
many times !

Runtime is
exponential in n .

- Store 1D DP-table and fill bottom-up in $O(n)$ time:

F:

0	1	1	2	3	5	8			
---	---	---	---	---	---	---	--	--	--

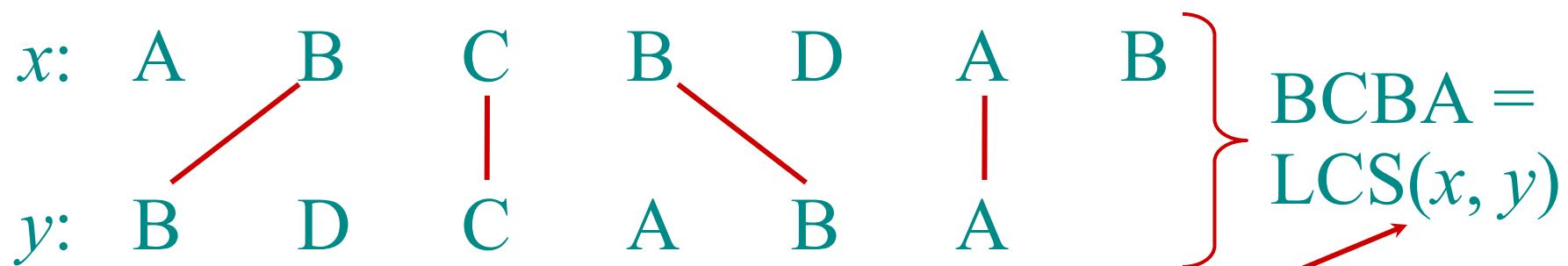


Longest Common Subsequence

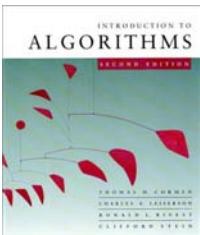
Example: *Longest Common Subsequence (LCS)*

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

“a” not “the”



functional notation,
but not a function

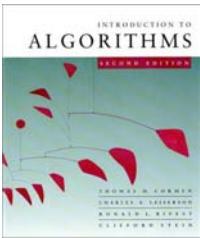


Brute-force LCS algorithm

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

Analysis

- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).
- Hence, the runtime would be exponential !



Towards a better algorithm

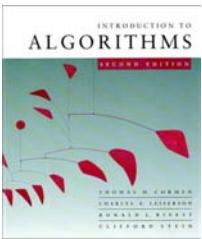
Two-Step Approach:

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 $|s|$.

Strategy: Consider *prefixes* of x and y .

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

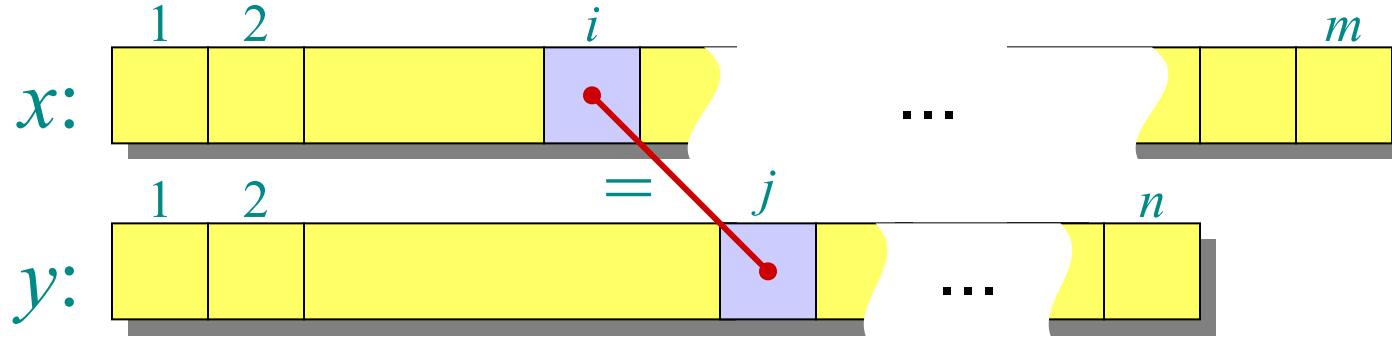


Recursive formulation

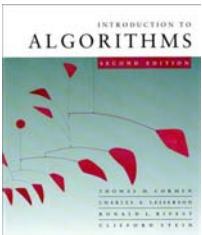
Theorem.

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

Proof. Case $x[i] = y[j]$:



Let $z[1 \dots k] = \text{LCS}(x[1 \dots i], y[1 \dots j])$, where $c[i, j] = k$. Then, $z[k] = x[i]$, or else z could be extended. Thus, $z[1 \dots k-1]$ is CS of $x[1 \dots i-1]$ and $y[1 \dots j-1]$.



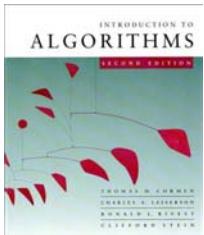
Proof (continued)

Claim: $z[1 \dots k-1] = \text{LCS}(x[1 \dots i-1], y[1 \dots j-1])$.

Suppose w is a longer CS of $x[1 \dots i-1]$ and $y[1 \dots j-1]$, that is, $|w| > k-1$. Then, ***cut and paste***: $w \parallel z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1 \dots i]$ and $y[1 \dots j]$ with $|w \parallel z[k]| > k$. Contradiction, proving the claim.

Thus, $c[i-1, j-1] = k-1$, which implies that $c[i, j] = c[i-1, j-1] + 1$.

Other cases are similar. 



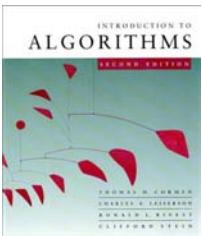
Dynamic-programming hallmark #1

Optimal substructure

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

→ *Recurrence*

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



Recursive algorithm for LCS

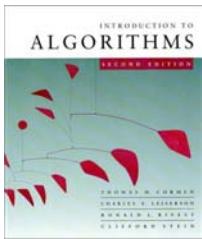
$\text{LCS}(x, y, i, j)$

if $x[i] = y[j]$

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

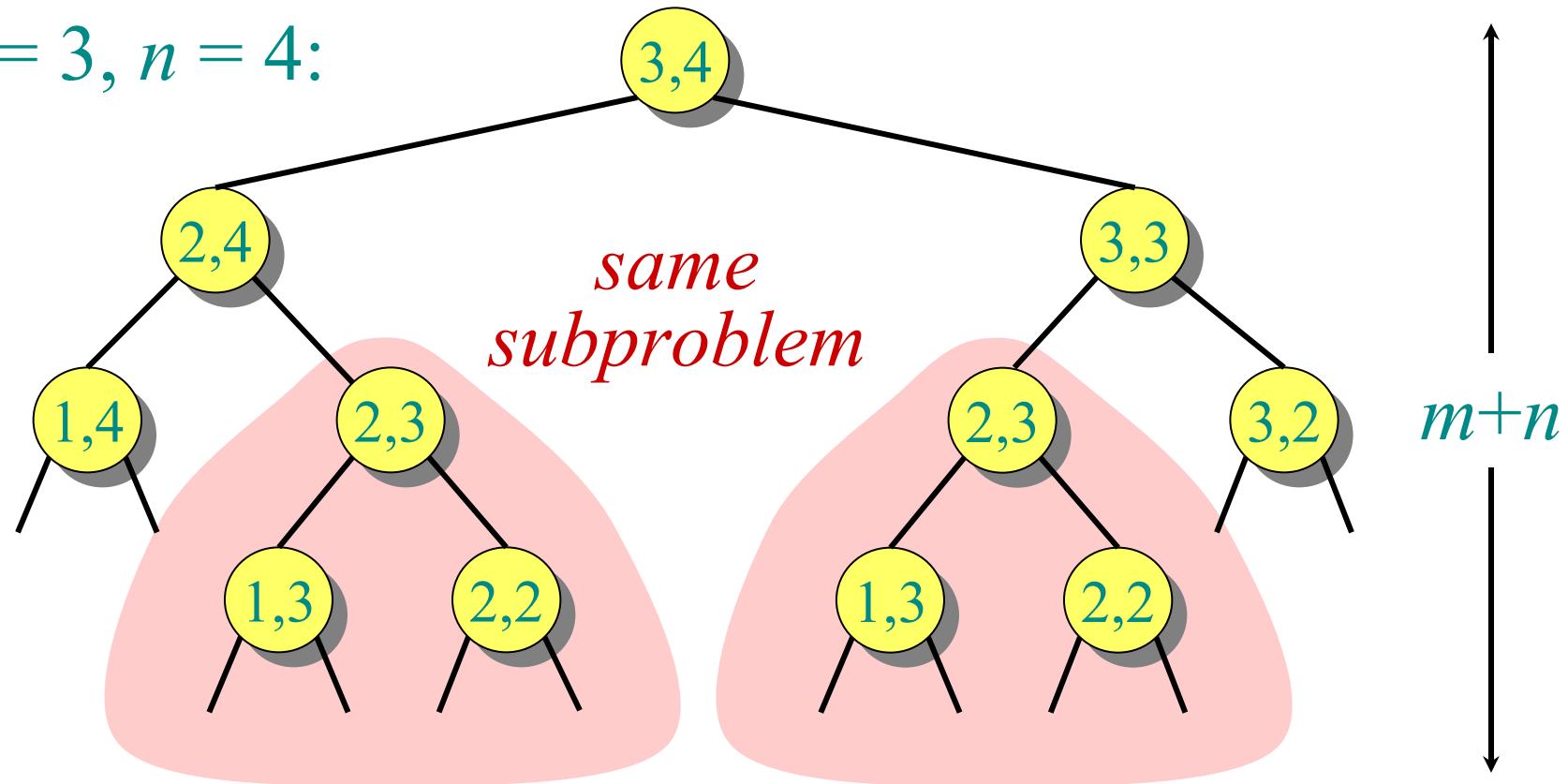
else $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j),$
 $\text{LCS}(x, y, i, j-1) \}$

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

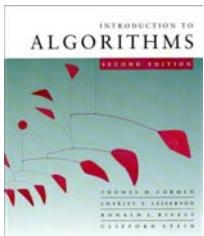


Recursion tree

$m = 3, n = 4$:



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

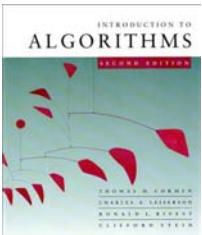


Dynamic-programming hallmark #2

Overlapping subproblems

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

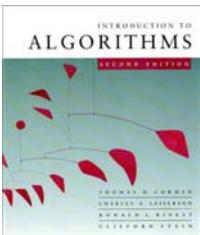
The number of distinct LCS subproblems for two strings of lengths m and n is only mn .



Dynamic-programming

There are two variants of dynamic programming:

1. Memoization
2. Bottom-up dynamic programming
(often referred to as “dynamic programming”)



Memoization algorithm

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

for all i, j : $c[i, 0] = 0$ and $c[0, j] = 0$

$\text{LCS}(x, y, i, j)$

if $c[i, j] = \text{NIL}$

then if $x[i] = y[j]$

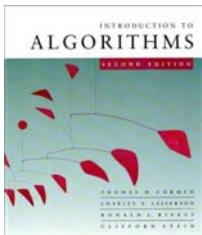
then $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$

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

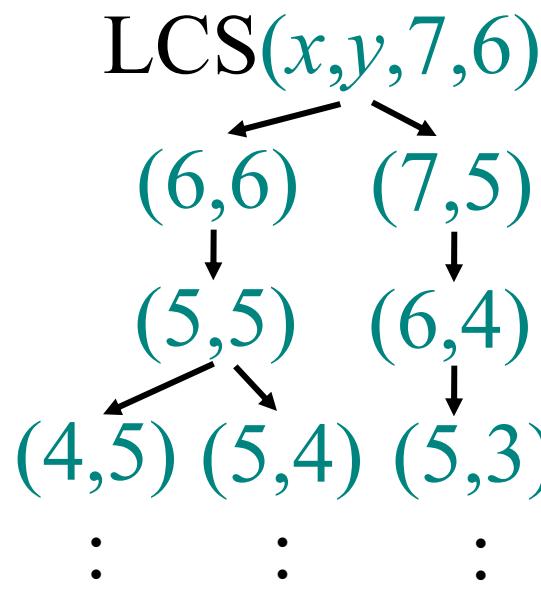
*same
as
before*

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

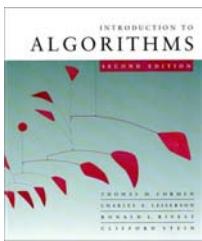
Space = $\Theta(mn)$.



Memoization



	1 x: A	2 B	3 C	4 B	5 D	6 A	7 B
y:	0	0	0	0	0	0	0
1 B	0	0	1	nil	nil	nil	nil
2 D	0	0	1	nil	nil	nil	nil
3 C	0	0	2	nil	nil	nil	nil
4 A	0	1	nil	nil	nil	nil	nil
5 B	0	nil	nil	nil	nil	nil	nil
6 A	0	nil	nil	nil	nil	nil	nil



Bottom-up dynamic-programming algorithm

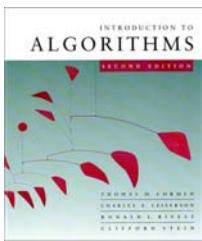
IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

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

The diagram shows a 7x8 grid of numbers representing a dynamic programming table. The columns are labeled A, B, C, B, D, A, B from left to right, and the rows are labeled A, B, D, C, A, B, A from top to bottom. Red arrows indicate the bottom-up computation path: starting from the bottom-right cell (4,4) and moving towards the top-left cell (0,0). Each cell contains its value, which is the sum of the values of the cell directly above it and the cell to its left. For example, the value 4 at (4,4) is the sum of 3 (from (3,4)) and 1 (from (4,3)). The first few cells are initialized to 0.



Bottom-up dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by back-tracing.

Space = $\Theta(mn)$.

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

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

The diagram illustrates a bottom-up dynamic programming algorithm for finding the Longest Common Subsequence (LCS) between two sequences, A and B. The sequences are represented as rows above the table. Sequence A consists of characters A, B, C, B, D, A, B. Sequence B consists of characters B, D, C, A, B, A. The table itself is an 8x8 grid where each cell contains a value representing the length of the LCS of the substrings ending at the corresponding positions in A and B. Red arrows indicate the path of back-tracing from the bottom-right cell (labeled 4) towards the top-left cell (labeled 0). The path starts at (B, A) with value 4, moves up to (B, B) with value 3, then right to (A, B) with value 3, up to (A, A) with value 2, right to (B, A) with value 2, up to (B, D) with value 1, right to (C, B) with value 1, up to (C, D) with value 1, right to (D, B) with value 1, up to (D, D) with value 0, and finally right to (A, B) with value 0. The cells are colored in a checkerboard pattern of light blue and yellow, with the bottom-right corner being yellow.