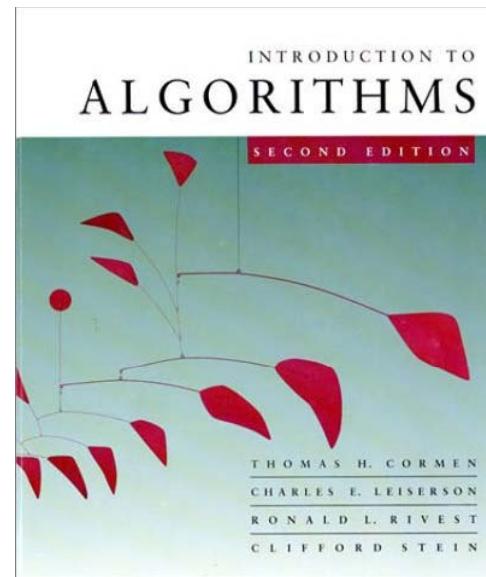
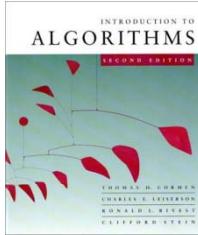


CS 3343 – Fall 2011



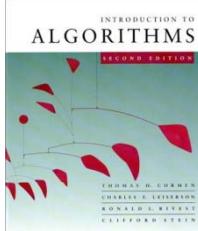
Dynamic Programming Carola Wenk

Slides courtesy of Charles Leiserson with small changes by Carola Wenk



Dynamic programming

- Algorithm design technique
- A technique for solving problems that have
 1. an optimal substructure property (recursion)
 2. overlapping subproblems
- **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 $n \geq 2$

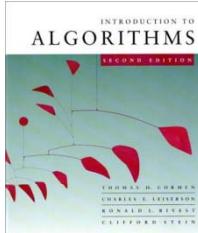
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Dynamic-programming hallmark #1

Optimal substructure

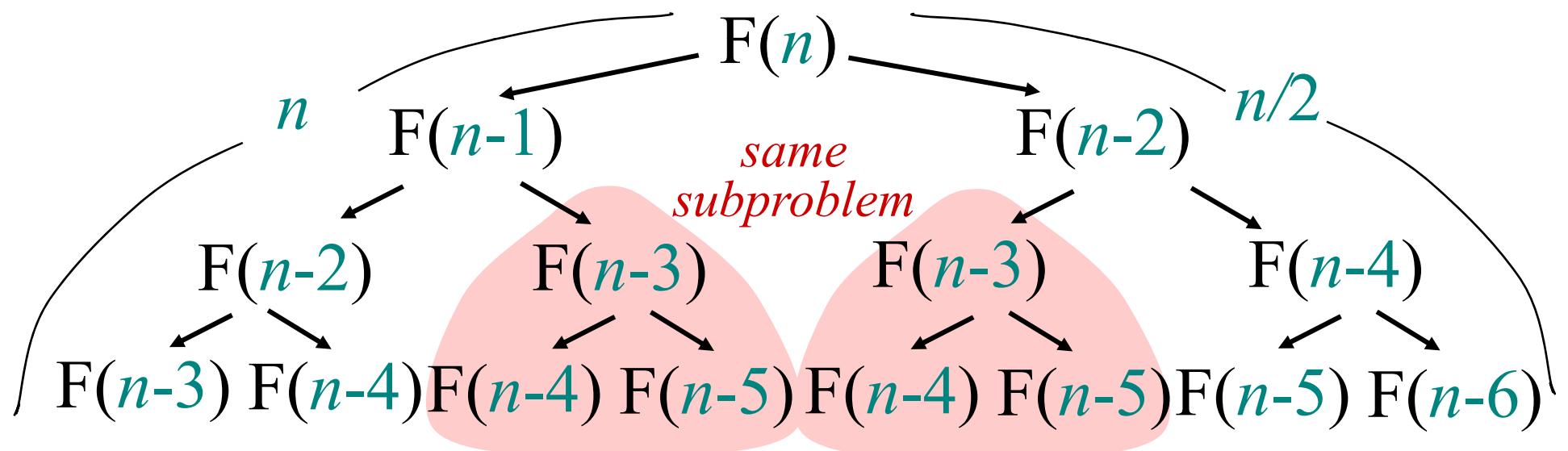
*An optimal solution to a problem
(instance) contains optimal
solutions to subproblems.*

→ *Recursion*

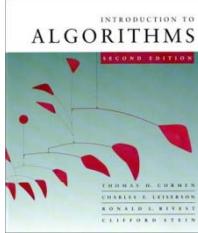


Example: Fibonacci numbers

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



- Runtime is exponential: $2^{n/2} \leq T(n) \leq 2^n$
- But we are repeatedly solving the same subproblems

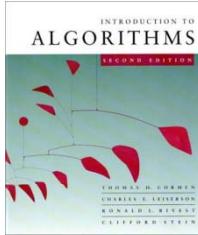


Dynamic-programming hallmark #2

Overlapping subproblems

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

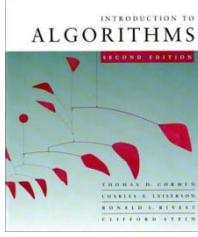
The number of distinct Fibonacci subproblems is only n .



Dynamic-programming

There are two variants of dynamic programming:

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



Bottom-up dynamic-programming algorithm

- Store 1D DP-table and fill bottom-up:

F:	0	1	1	2	3	5	8				
----	---	---	---	---	---	---	---	--	--	--	--

`fibBottomUpDP(n)`

$F[0] \leftarrow 0$

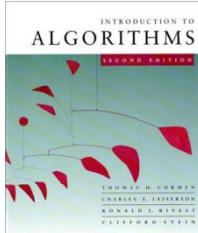
$F[1] \leftarrow 1$

for ($i \leftarrow 2, i \leq n, i++$)

$F[i] \leftarrow F[i-1] + F[i-2]$

return $F[n]$

- Time = $\Theta(n)$, space = $\Theta(n)$



Memoization algorithm

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

fibMemoization(n)

for all i : $F[i] = \text{null}$

fibMemoizationRec(n, F)

return $F[n]$

fibMemoizationRec(n, F)

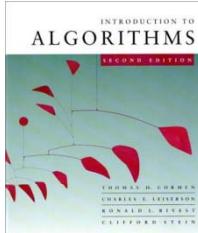
if ($F[n] = \text{null}$)

if ($n=0$) $F[n] \leftarrow 0$

if ($n=1$) $F[n] \leftarrow 1$

$F[n] \leftarrow \text{fibMemoizationRec}(n-1, F)$
 + $\text{fibMemoizationRec}(n-2, F)$

- Time = $\Theta(n)$, space = $\Theta(n)$

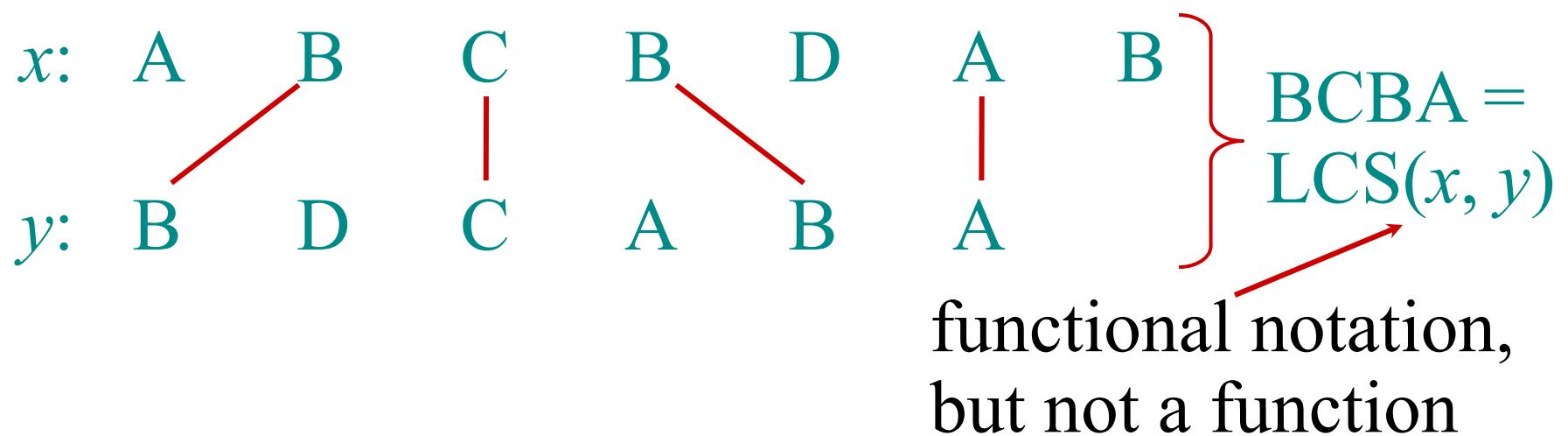


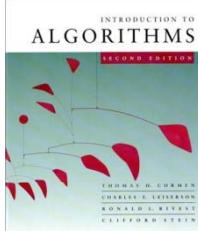
Longest Common Subsequence

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”



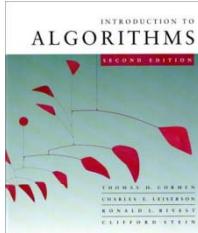


Brute-force LCS algorithm

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

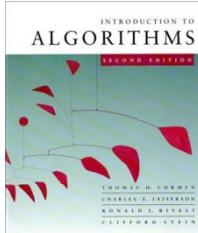
Two-Step Approach:

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, $c[m, n] = |\text{LCS}(x, y)|$.

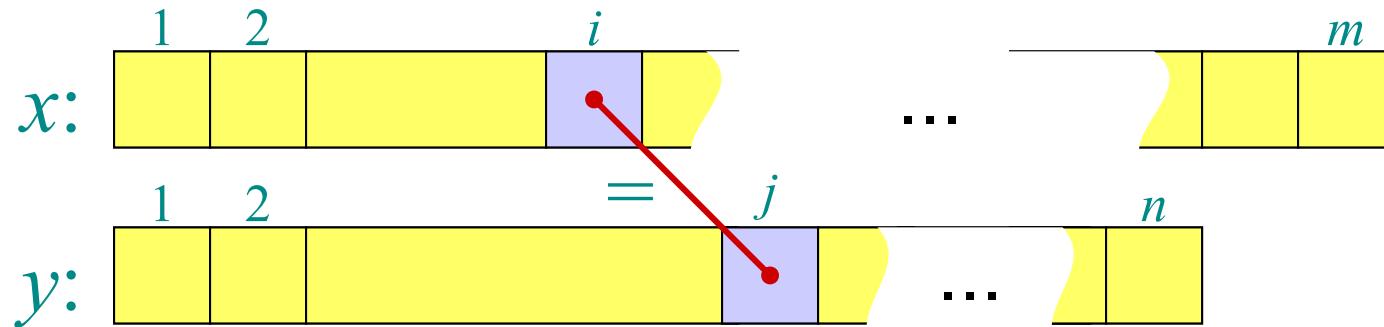


Recursive formulation

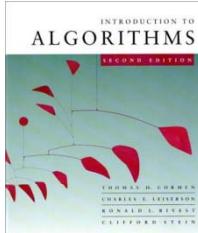
Theorem.

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

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 $y[1 \dots j-1]$.



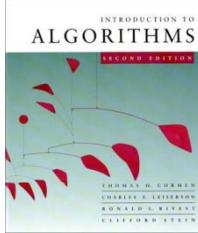
Proof (continued)

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. □



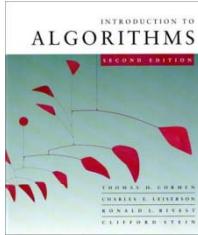
Dynamic-programming hallmark #1

Optimal substructure

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

→ *Recursion*

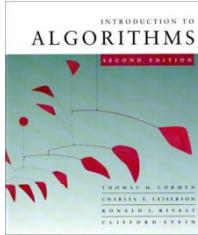
If $z = \text{LCS}(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y .



Recursive algorithm for LCS

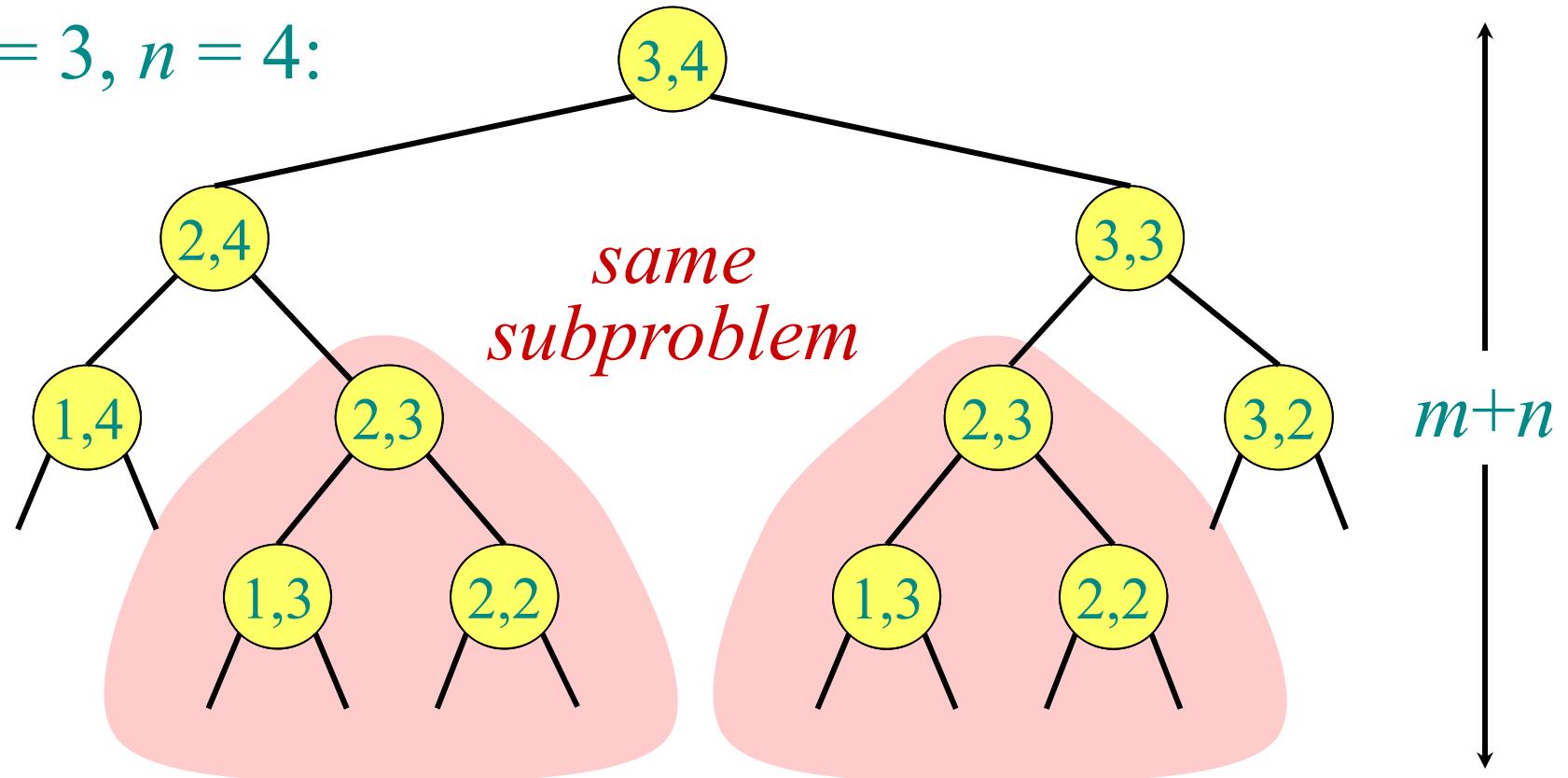
```
LCS( $x, y, i, j$ )
  if  $x[i] = y[j]$ 
    then  $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$ 
  else  $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j),$ 
         $\text{LCS}(x, y, i, j-1) \}$ 
```

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

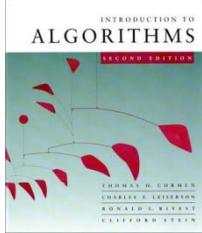


Recursion tree

$m = 3, n = 4$:



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

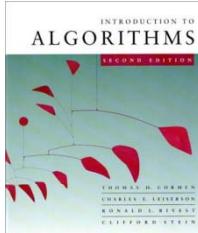


Dynamic-programming hallmark #2

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 .



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 $x[i] = y[j]$

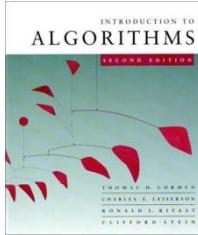
then $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$

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 = $\Theta(mn)$.

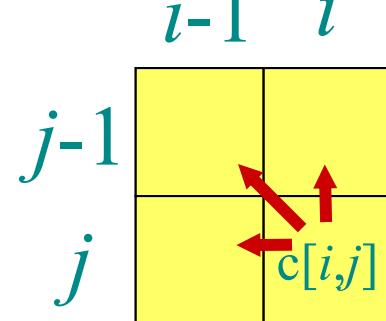


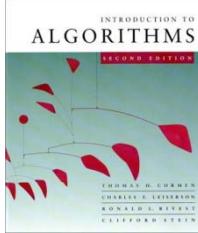
Recursive formulation

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

Annotations: A red arrow points from the term $c[i-1, j-1]$ to the first term in the max expression. Another red arrow points from the term $c[i, j-1]$ to the second term in the max expression.

c :





Bottom-up dynamic-programming algorithm

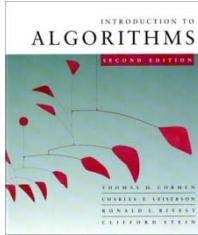
IDEA:

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	2	3
B	0	1	2	2	3	3	4
A	0	1	2	2	3	3	4

The diagram illustrates a bottom-up dynamic programming algorithm for a sequence matching problem. The table has rows and columns labeled A, B, C, B, D, A, B. The first row and column are initialized to 0. Subsequent cells contain values 1, 2, or 3, indicating matches or edits. Red arrows show the path from each cell to its predecessor, illustrating the recurrence relation used in the computation. The final value in the bottom-right cell is 4.



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	3	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 listed above the table: A = [A, B, C, B, D, A, B] and B = [B, D, C, A, B, A]. The table itself contains the lengths of the LCS up to that point. Red arrows show the back-tracing path from the bottom-right cell (labeled 4) to the top-left cell (labeled 0). The path starts at (4,4), moves up to (3,4), then right to (4,3), up to (3,3), right to (4,2), up to (3,2), right to (4,1), up to (3,1), right to (4,0), and finally up to (3,0).