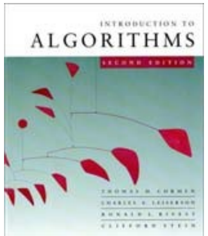
CS60020: Foundations of Algorithm Design and Machine Learning

Sourangshu Bhattacharya



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.

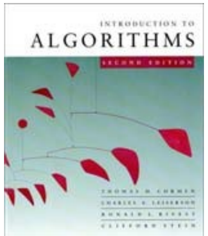
Dynamic programming

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.

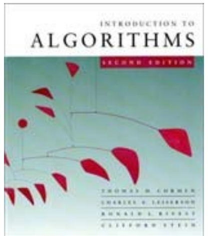
Example: ***Longest Common Subsequence (LCS)***

- 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.

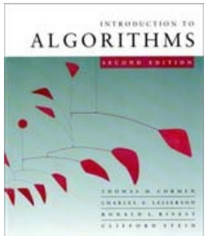
"a" *not* "the"

x: A B C B D A B

y: B D C A B A

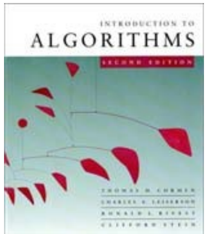
BCBA = LCS(x, y)

functional notation, but not a function



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

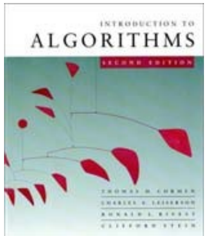
Check every subsequence of $x[1 \,.\,.\, m]$ to see if it is also a subsequence of $y[1 \,.\,.\, n]$.

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 $= O(n2^m)$

$=$ exponential time.



Towards a better algorithm

Simplification:

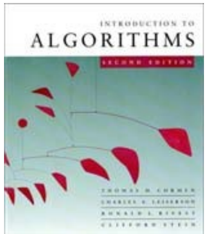
1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Towards a better algorithm

Simplification:

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 $|s|$.

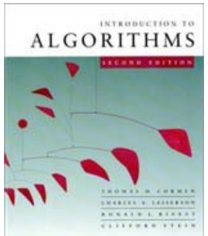
Strategy: Consider ***prefixes*** of x and y.

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

Recursive formulation

Theorem.

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

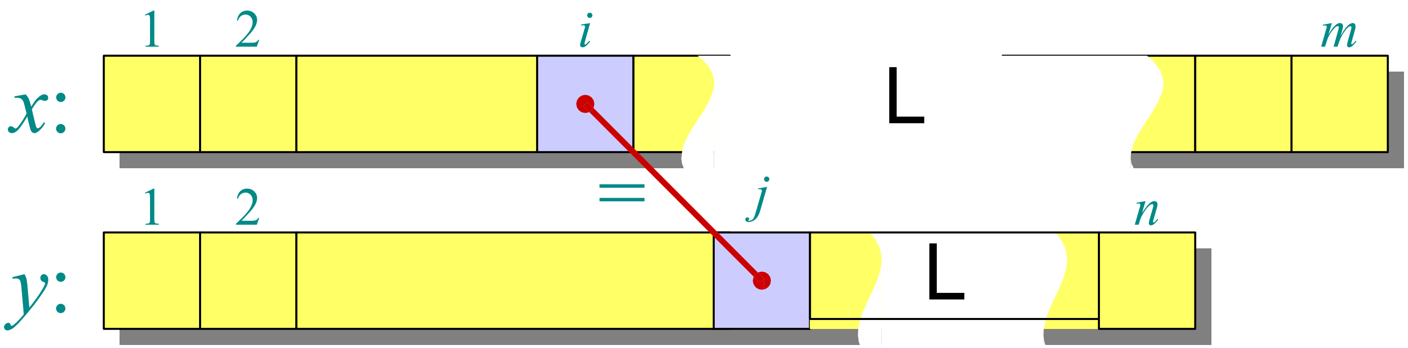


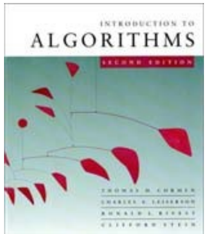
Recursive formulation

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



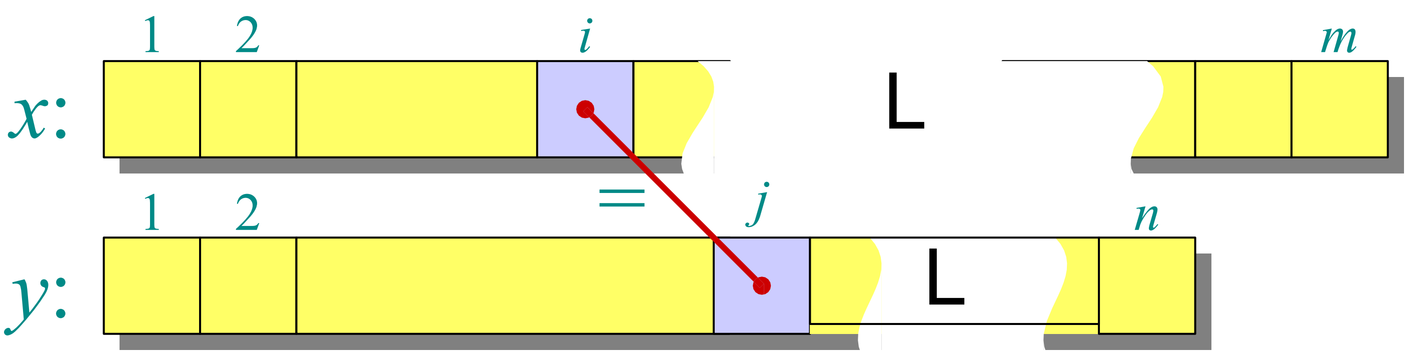


Recursive formulation

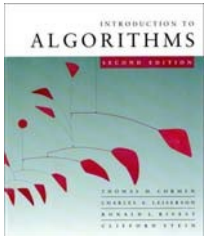
Theorem.

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

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



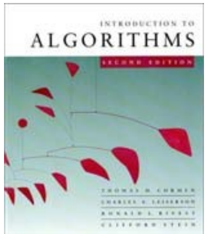
Let $z[1 \,.\,.\, k] = \text{LCS}(x[1 \,.\,.\, i], y[1 \,.\,.\, j])$, where $c[i, j] = k$. Then, $z[k] = x[i]$, or else z could be extended. Thus, $z[1 \,.\,.\, k-1]$ is CS of $x[1 \,.\,.\, i-1]$ and $y[1 \,.\,.\, j-1]$.



Proof (continued)

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

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



Proof (continued)

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

Suppose w is a longer CS of $x[1\,.\,.\,i–1]$ and $y[1\,.\,.\,j–1]$, that is, $|w| > k–1$. Then, ***cut and paste***: $w \,\|\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1\,.\,.\,i]$ and $y[1\,.\,.\,j]$ with $\big|w \,\|\, z[k]\big| > k$. 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. $\square$

Dynamic-programming hallmark #1

Optimal substructure

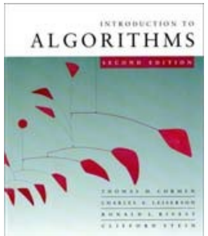
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Dynamic-programming hallmark #1

Optimal substructure

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

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

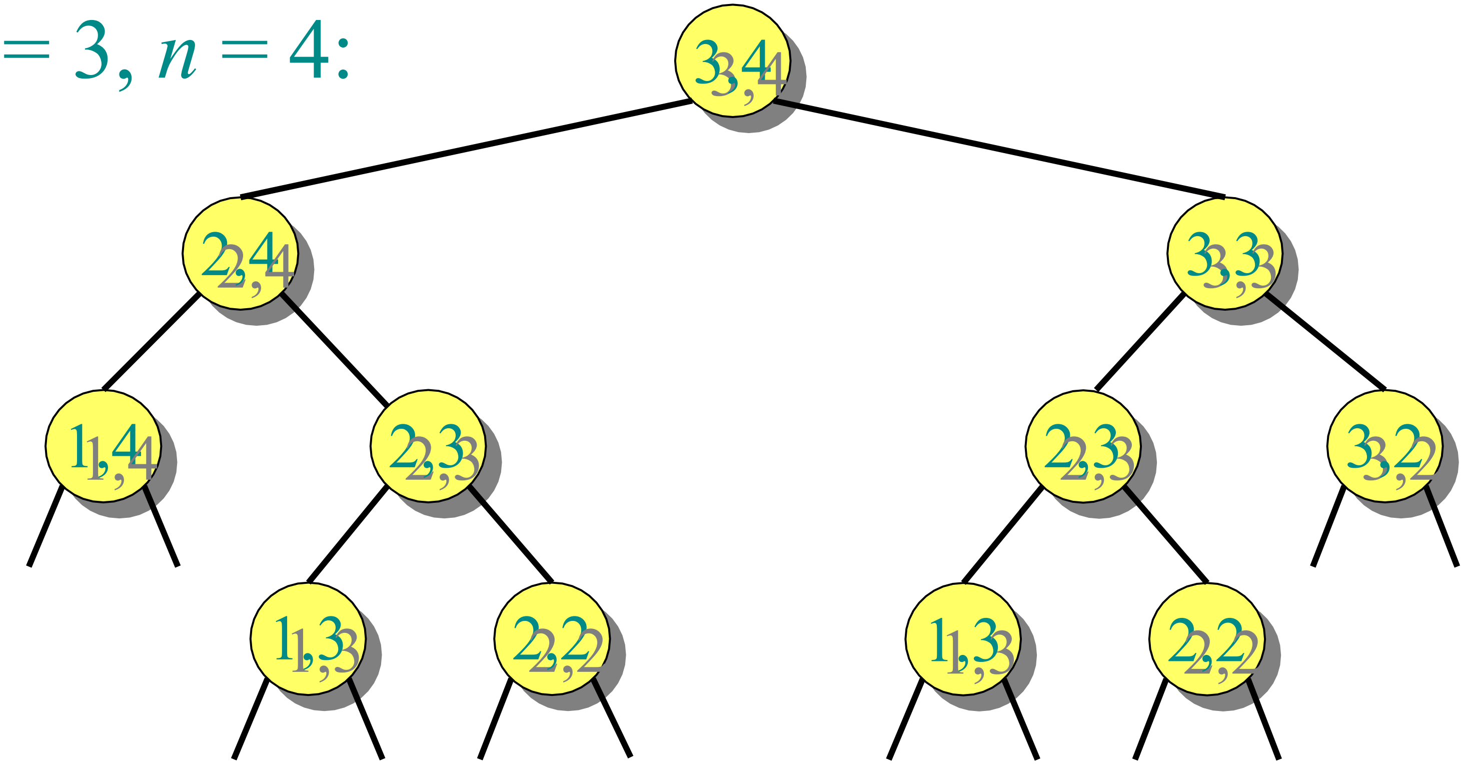
Recursive algorithm for LCS

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.

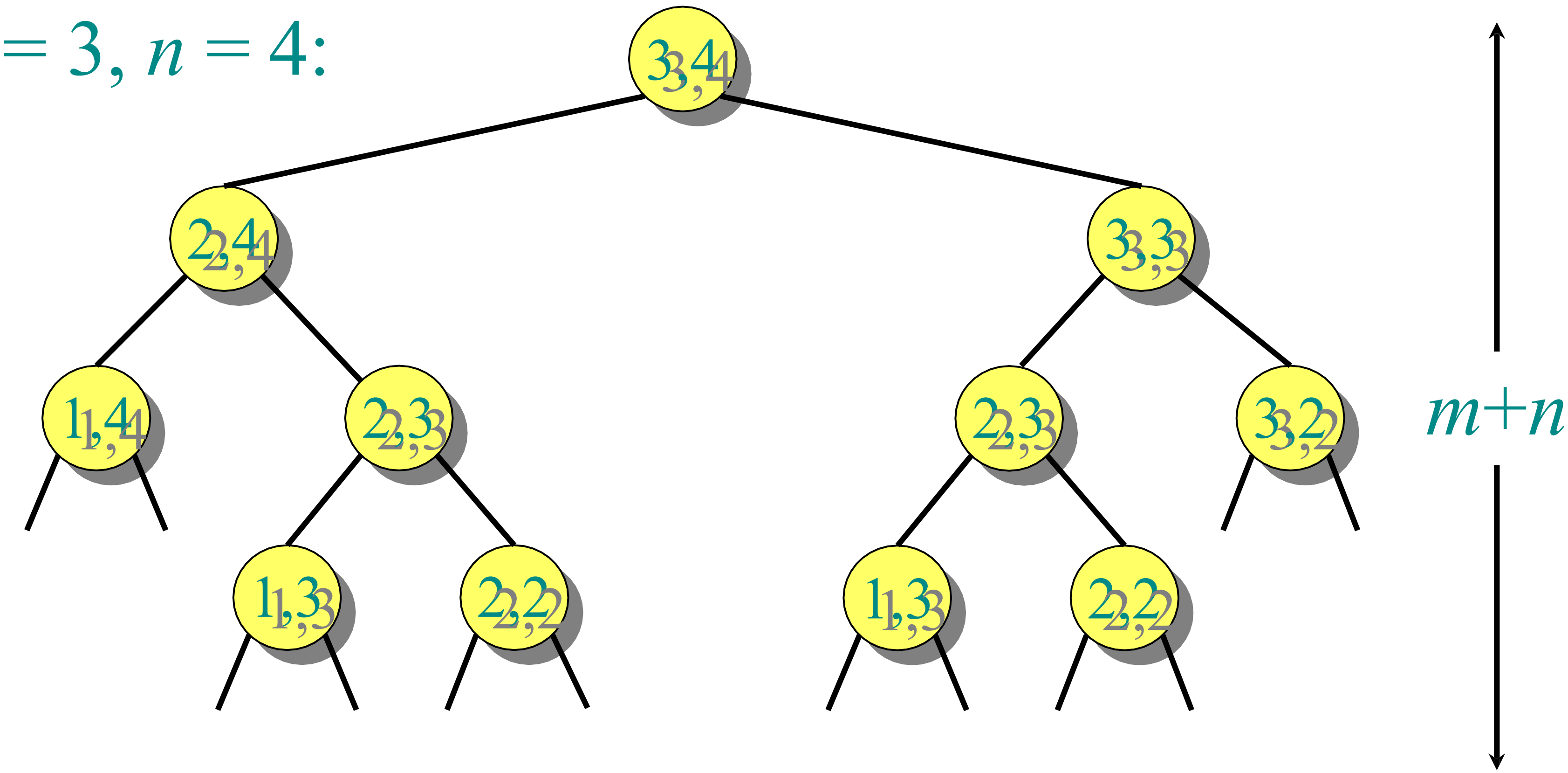
Recursion tree

$m = 3, n = 4$:



Recursion tree

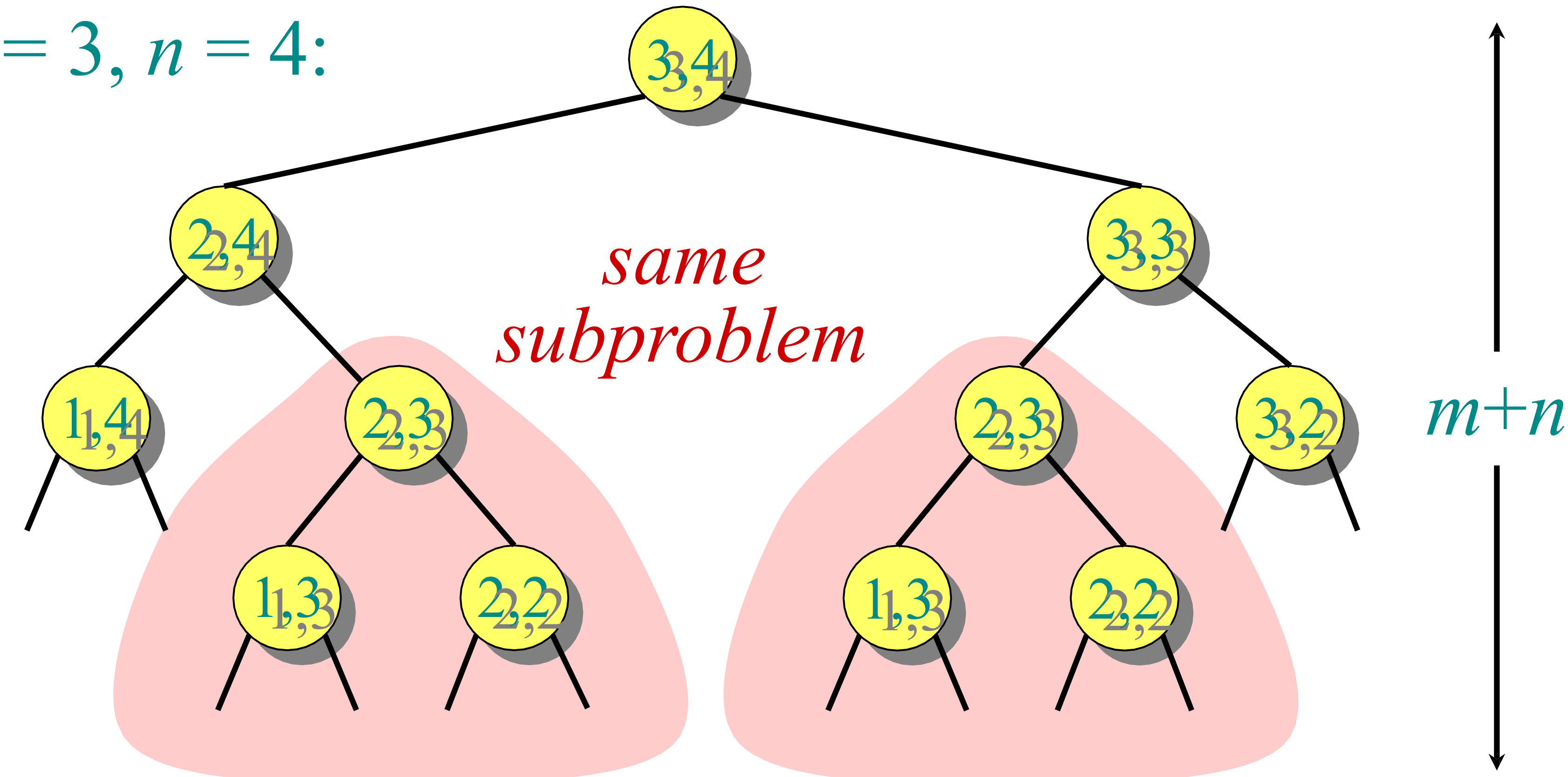
$m = 3, n = 4$:



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

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

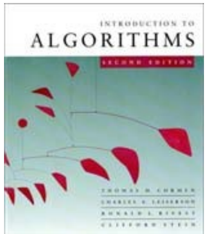
A recursive solution contains a "small" number of distinct subproblems repeated many times.

Dynamic-programming hallmark #2

Overlapping subproblems

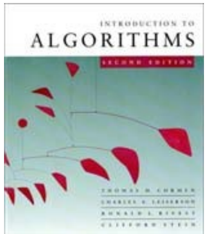
A recursive solution contains a "small" number of distinct subproblems repeated many times.

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

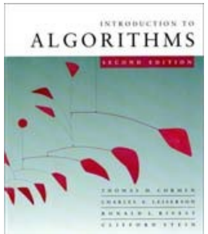
  then if $x[i] = y[j]$

   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



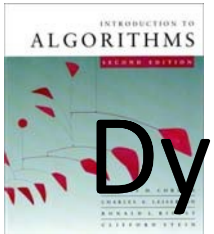
Memoization algorithm

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 $= \Theta(mn)$.



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

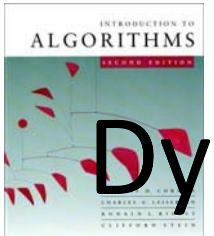
Dynamic-programming algorithm

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



Dynamic-programming algorithm

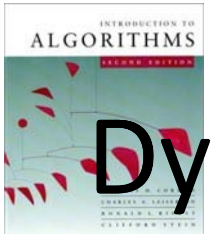
IDEA:

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