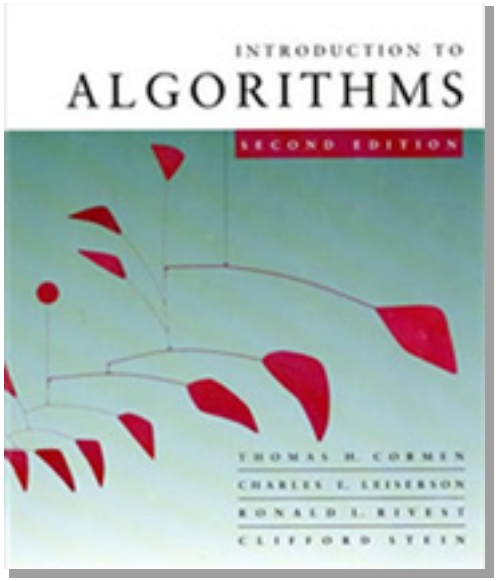
Introduction to Algorithms

6.046J/18.401J



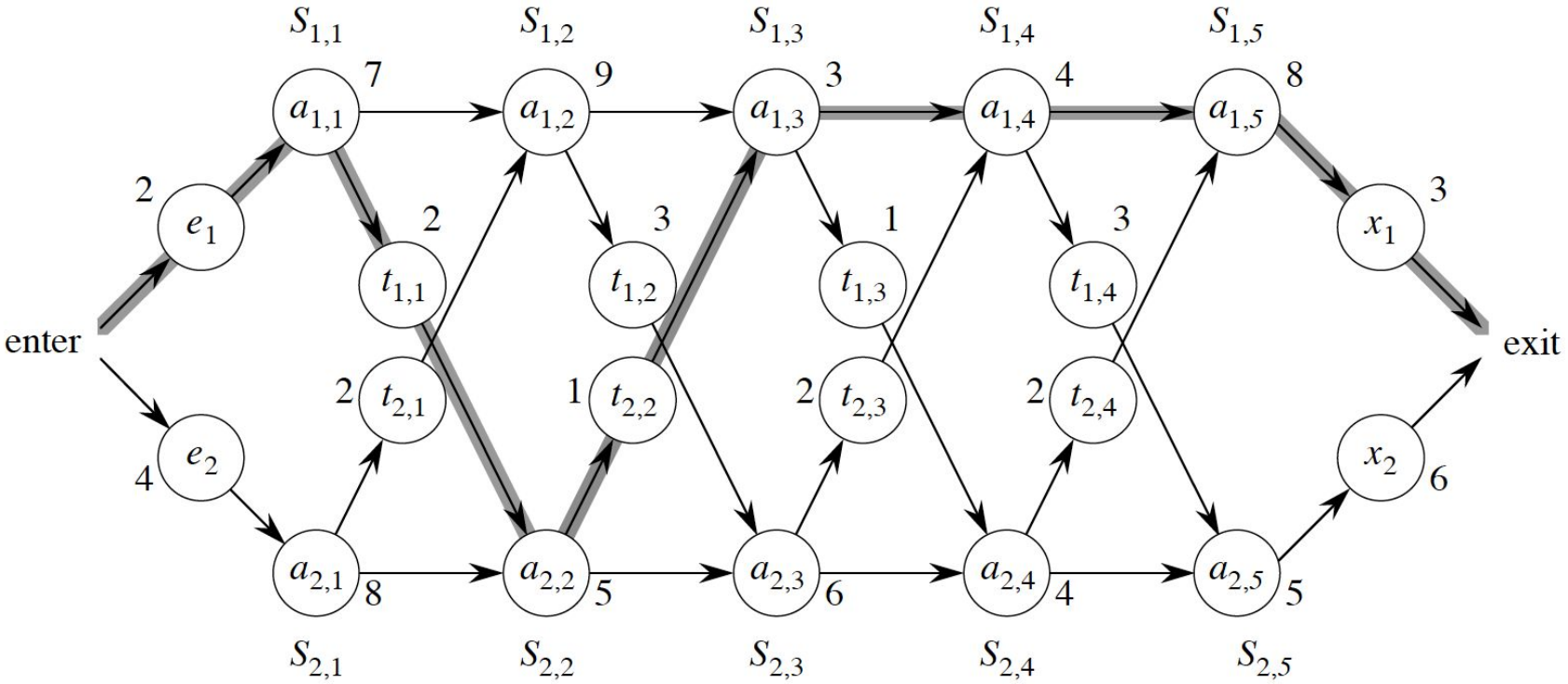
LECTURE 15

Dynamic Programming

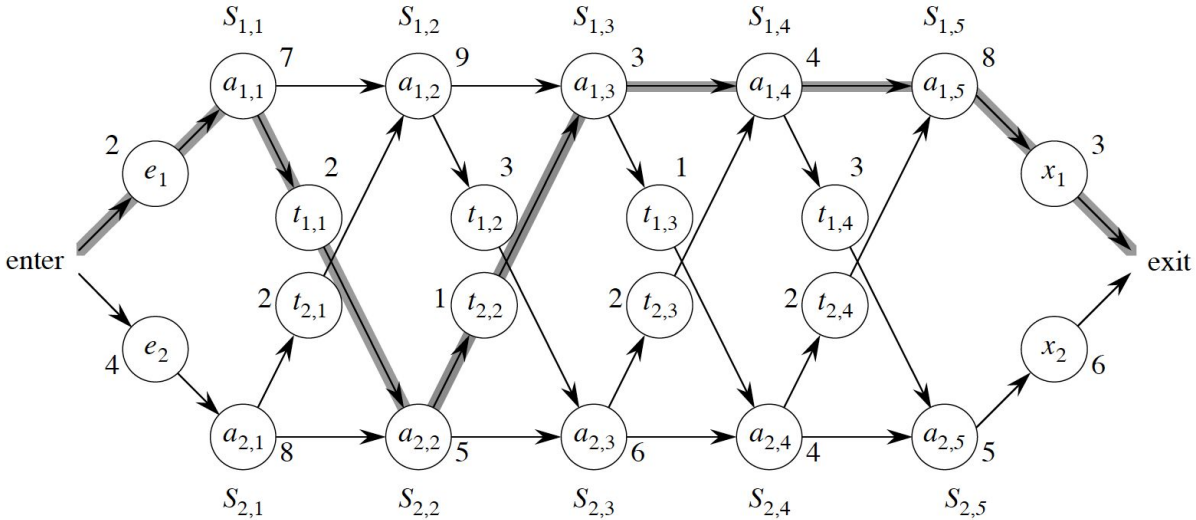
- Longest common subsequence
- Optimal substructure
- Overlapping subproblems

Prof. Charles E. Leiserson

Assembly-line scheduling



Assembly-line scheduling - recursive solution



j	1	2	3	4	5
$f_1[j]$	9	18	20	24	32
$f_2[j]$	12	16	22	25	30

$$f^* = 35$$

j	2	3	4	5
$l_1[j]$	1	2	1	1
$l_2[j]$	1	2	1	2

$$l^* = 1$$

Assembly-line scheduling -

FASTEST-WAY(a, t, e, x, n)

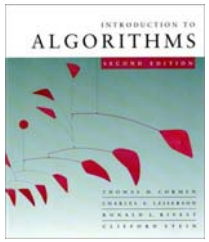
$f_1[1] \leftarrow e_1 + a_{1,1}$
$f_2[1] \leftarrow e_2 + a_{2,1}$
for $j \leftarrow 2$ **to** n
    do if $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$
        then $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$
            $l_1[j] \leftarrow 1$
        else $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$
            $l_1[j] \leftarrow 2$
      if $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$
        then $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$
            $l_2[j] \leftarrow 2$
        else $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$
            $l_2[j] \leftarrow 1$
if $f_1[n] + x_1 \leq f_2[n] + x_2$
  then $f^* = f_1[n] + x_1$
      $l^* = 1$
  else $f^* = f_2[n] + x_2$
      $l^* = 2$

PRINT-STATIONS(l, n)

$i \leftarrow l^*$
print "line " i ", station " n
for $j \leftarrow n$ **downto** 2
    do $i \leftarrow l_i[j]$
        print "line " i ", station " $j - 1$

Go through example.

Time $= \Theta(n)$

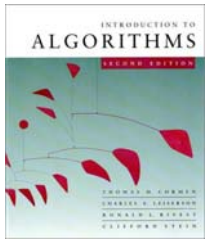


Dynamic programming

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.



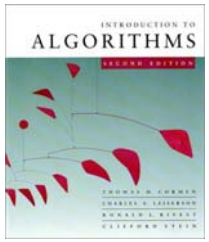
Dynamic programming

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.

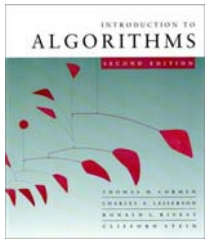
Example: *Longest Common Subsequence (LCS)*

- 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



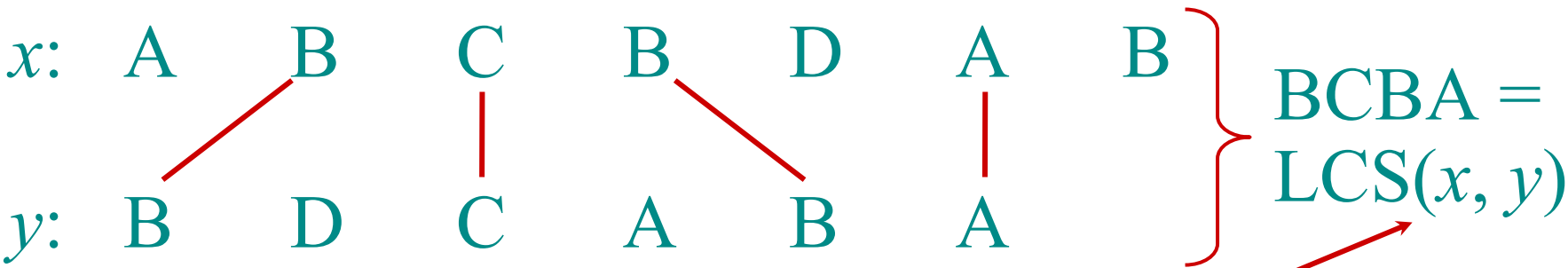
Dynamic programming

Design technique, like divide-and-conquer.

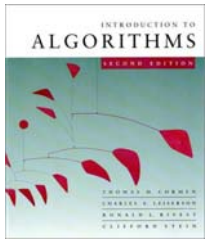
Example: *Longest Common Subsequence (LCS)*

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

"a" *not* "the"

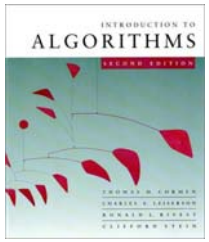


functional notation,
but not a function



Brute-force LCS algorithm

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



Brute-force LCS algorithm

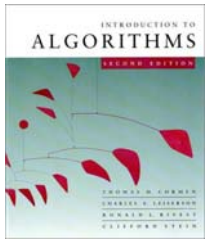
Check every subsequence of $x[1 \,..\, m]$ to see if it is also a subsequence of $y[1 \,..\, n]$.

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 $= O(n2^m)$

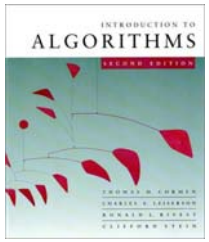
$=$ exponential time.



Towards a better algorithm

Simplification:

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

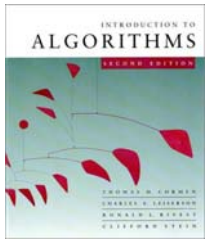


Towards a better algorithm

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

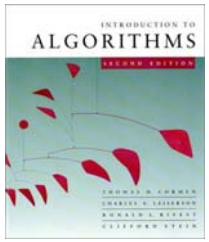
Simplification:

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.

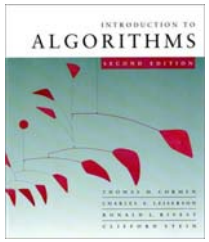
- Define $c[i, j] = |\text{LCS}(x[1 \,..\, i], y[1 \,..\, j])|$.
- Then, $c[m, n] = |\text{LCS}(x, y)|$.



Recursive formulation

Theorem.

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

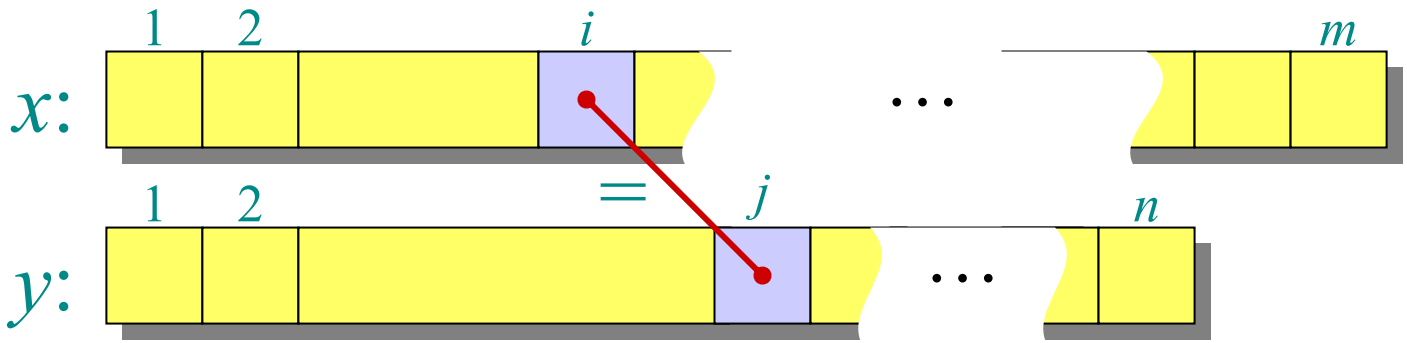


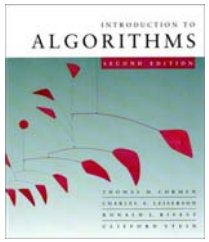
Recursive formulation

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



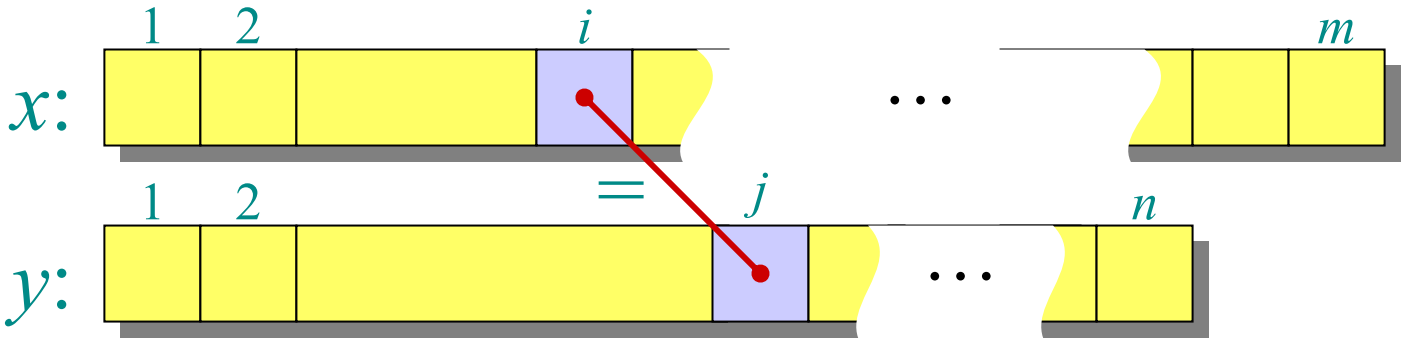


Recursive formulation

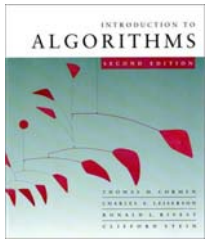
Theorem.

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

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



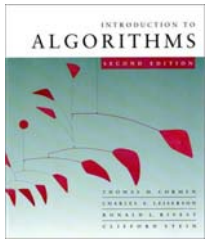
Let $z[1\,..\,k] = \text{LCS}(x[1\,..\,i], y[1\,..\,j])$, where $c[i,j] = k$. Then, $z[k] = x[i]$, or else z could be extended. Thus, $z[1\,..\,k-1]$ is CS of $x[1\,..\,i-1]$ and $y[1\,..\,j-1]$.



Proof (continued)

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

Suppose w is a longer CS of $x[1\,..\,i\text{–}1]$ and $y[1\,..\,j\text{–}1]$, that is, $|w| > k\text{–}1$. 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 $|w \,\|\, z[k]| > k$. Contradiction, proving the claim.



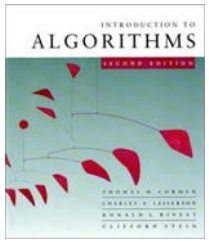
Proof (continued)

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

Suppose w is a longer CS of $x[1\,..\,i\text{–}1]$ and $y[1\,..\,j\text{–}1]$, that is, $|w| > k\text{–}1$. 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 $|w \,\|\, z[k]| > k$. Contradiction, proving the claim.

Thus, $c[i\text{–}1, j\text{–}1] = k\text{–}1$, which implies that $c[i, j] = c[i\text{–}1, j\text{–}1] + 1$.

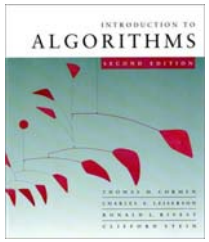
Other cases are similar. □



Dynamic-programming hallmark #1

Optimal substructure

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

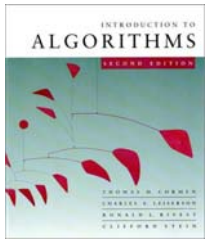


Dynamic-programming hallmark #1

Optimal substructure

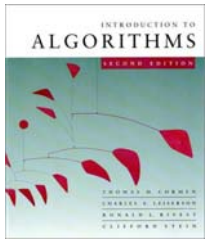
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

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

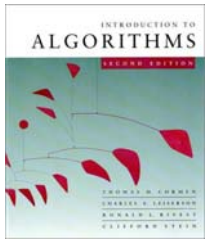
LCS(x, y, i, j)
  if $x[i] = y[j]$
    then $c[i, j] \leftarrow$ LCS$(x, y, i–1, j–1) + 1$
    else $c[i, j] \leftarrow \max\{LCS(x, y, i–1, j)$,
                LCS$(x, y, i, j–1)\}$



Recursive algorithm for LCS

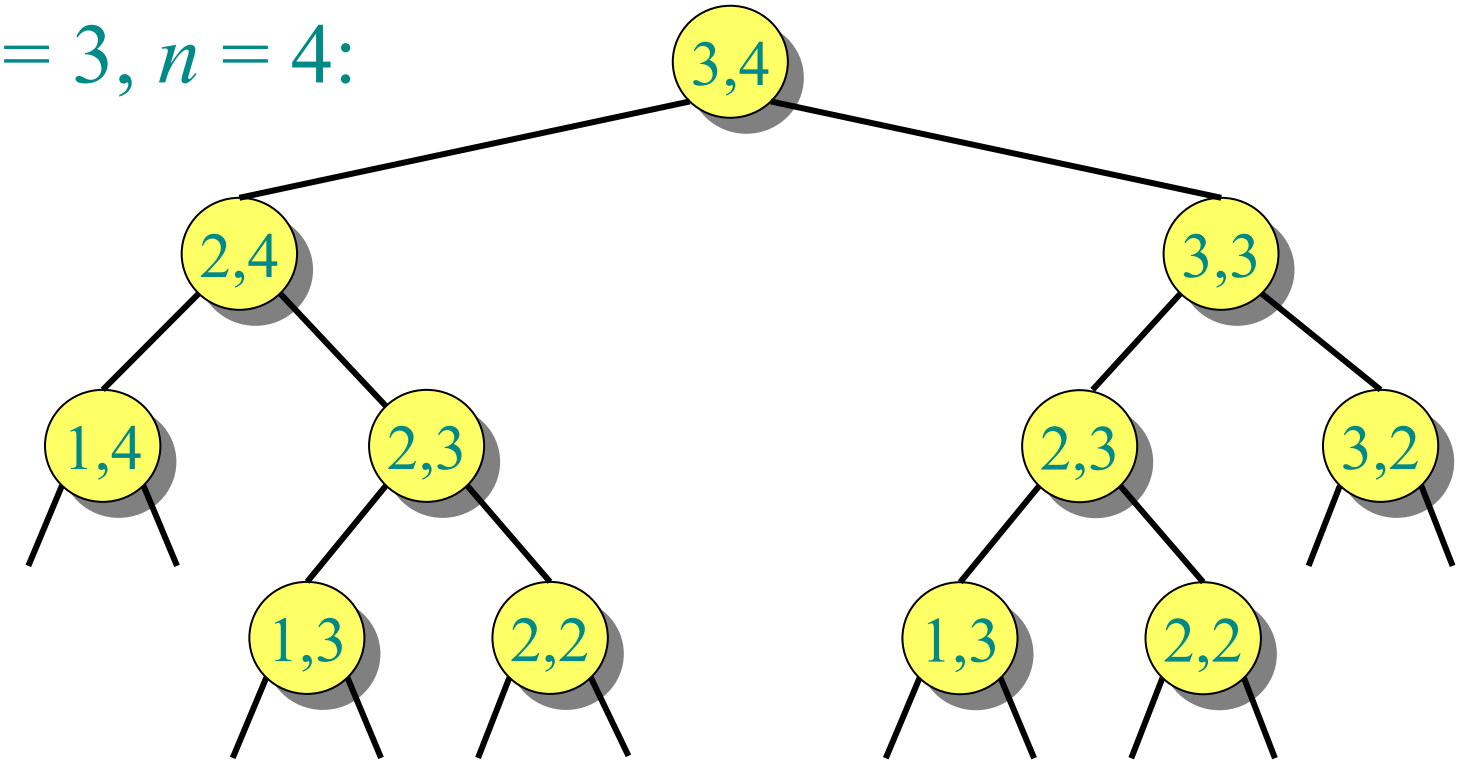
$$\begin{aligned}
&\text{LCS}(x, y, i, j) \\
&\quad \textbf{if } x[i] = y[j] \\
&\quad\quad \textbf{then } c[i, j] \leftarrow \text{LCS}(x, y, i{-}1, j{-}1) + 1 \\
&\quad\quad \textbf{else } c[i, j] \leftarrow \max\{\text{LCS}(x, y, i{-}1, j), \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{LCS}(x,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

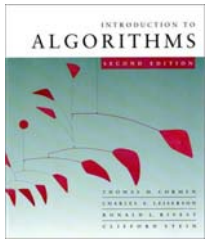
Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.



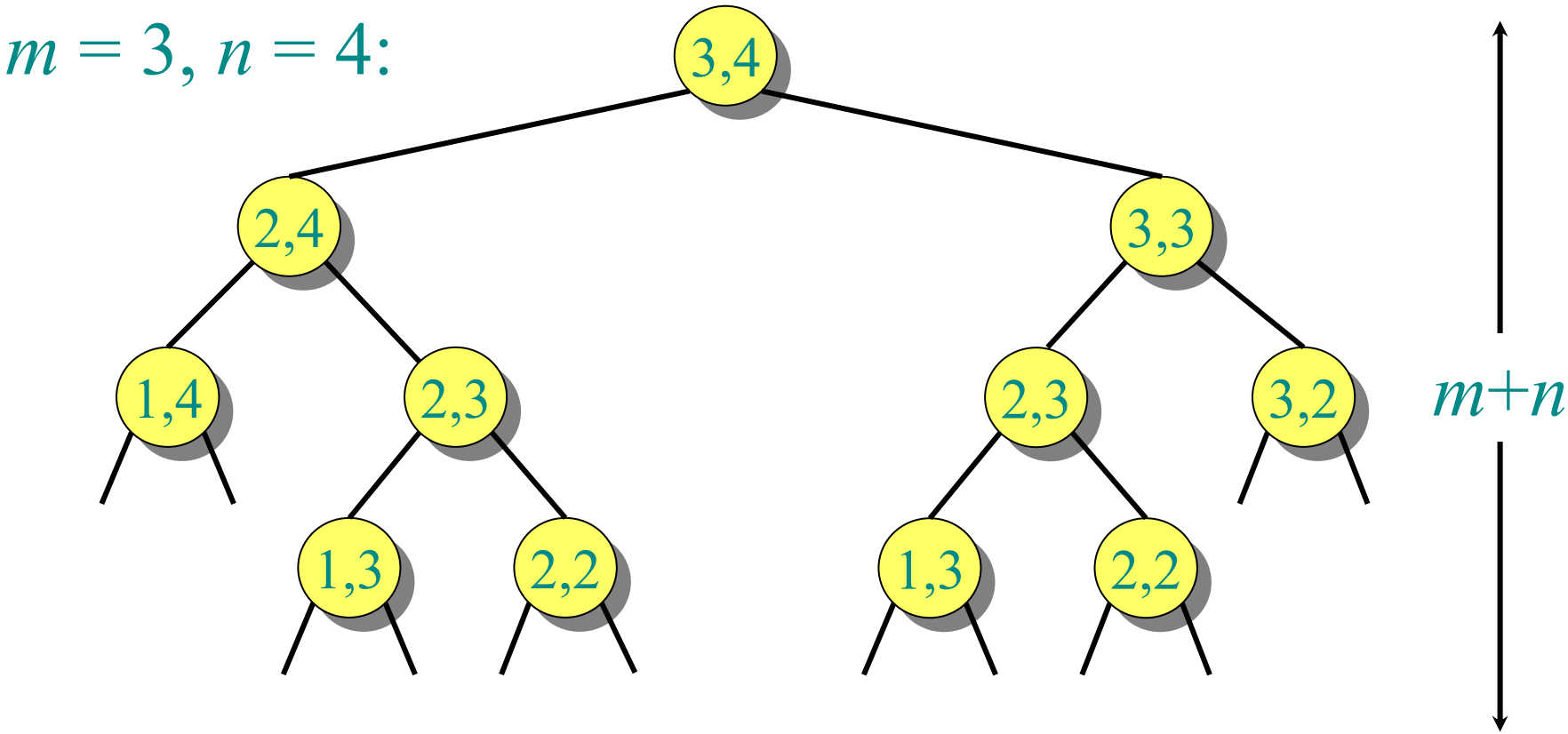
Recursion tree

$m = 3$, $n = 4$:

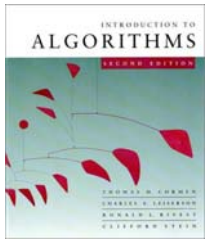




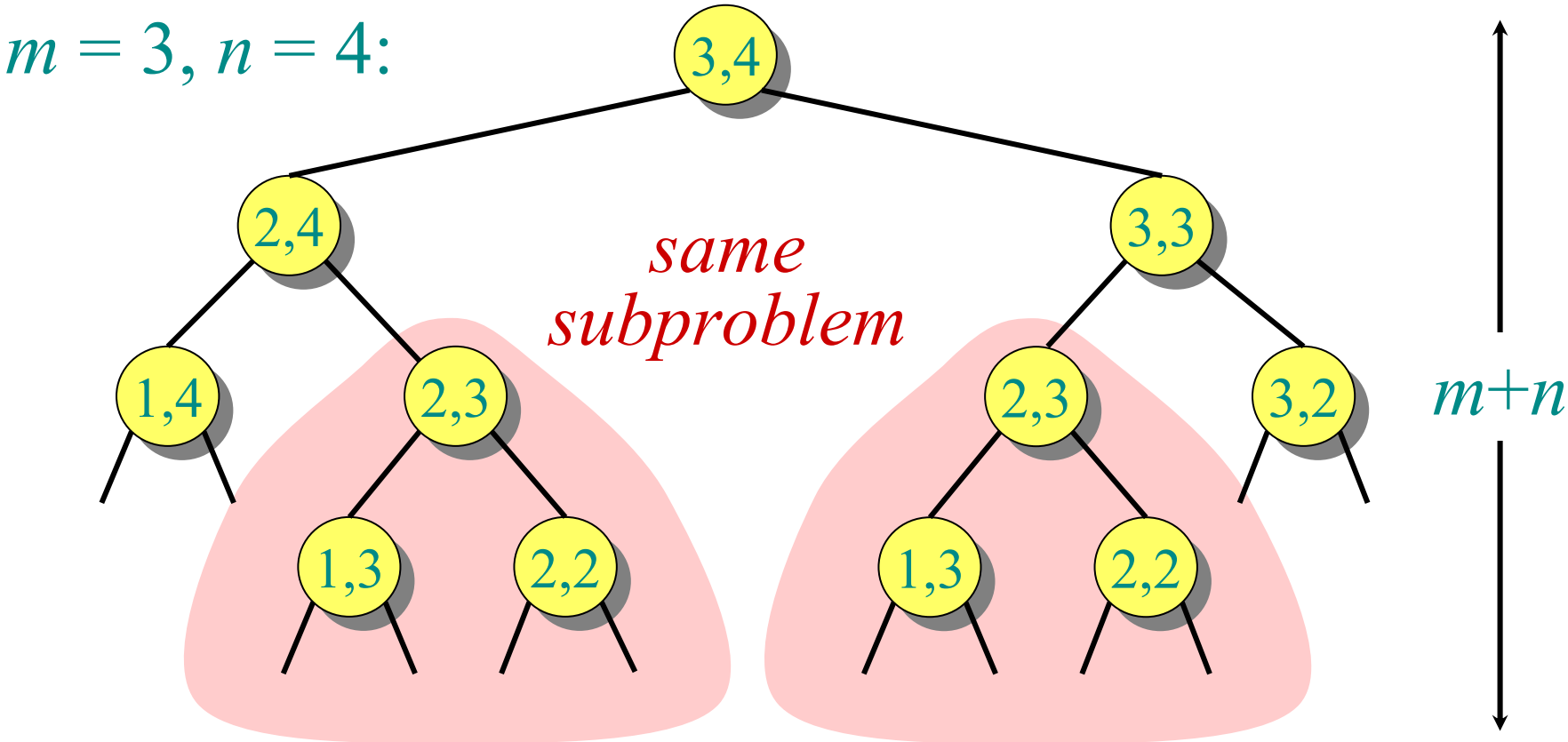
Recursion tree



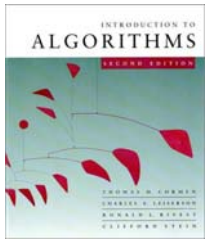
Height = $m + n$ ⇒ work potentially exponential.



Recursion tree



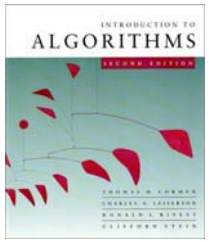
Height = $m + n$ ⇒ work potentially exponential, but we're solving subproblems already solved!



Dynamic-programming hallmark #2

Overlapping subproblems

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

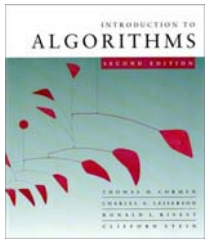


Dynamic-programming hallmark #2

Overlapping subproblems

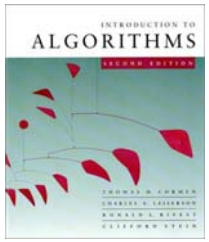
A recursive solution contains a "small" number of distinct subproblems repeated many times.

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.

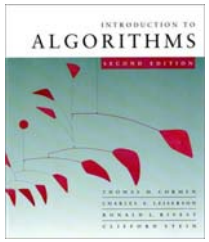


Memoization algorithm

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] = NIL
        then if x[i] = y[j]
                then c[i, j] ← LCS(x, y, i–1, j–1) + 1
                else c[i, j] ← max { LCS(x, y, i–1, j),
                                     LCS(x, y, i, j–1) }
```

same as before



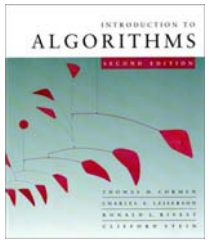
Memoization algorithm

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] = NIL
        then if x[i] = y[j]
                then c[i, j] ← LCS(x, y, i–1, j–1) + 1
                else c[i, j] ← 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 = $\Theta(mn)$.

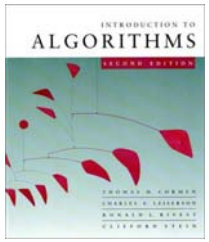


Dynamic-programming algorithm

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



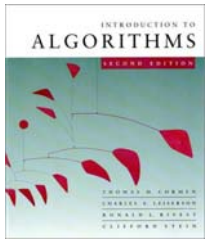
Dynamic-programming algorithm

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

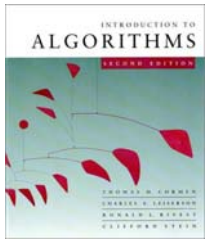
IDEA:

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