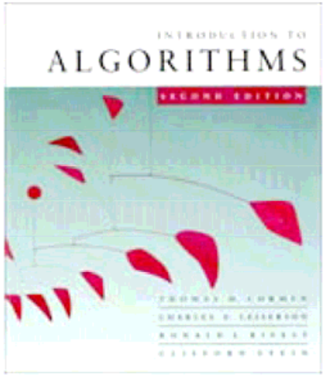
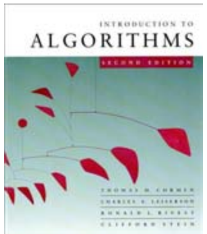


Dynamic Programming

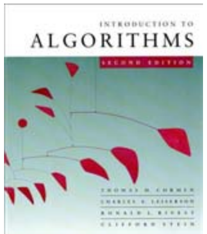


Dynamic programming

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.



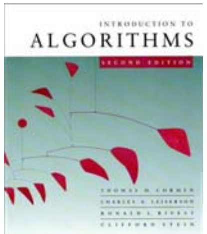
Dynamic programming

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.

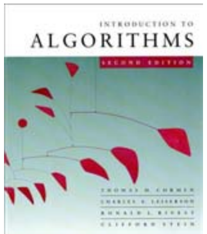
Example: ***Longest Common Subsequence (LCS)***

- 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:

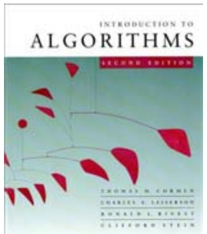
1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

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 $|s|$.

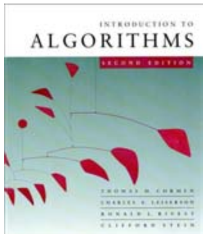
Strategy: Consider ***prefixes*** of x and y.

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

Recursive formulation

Theorem.

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

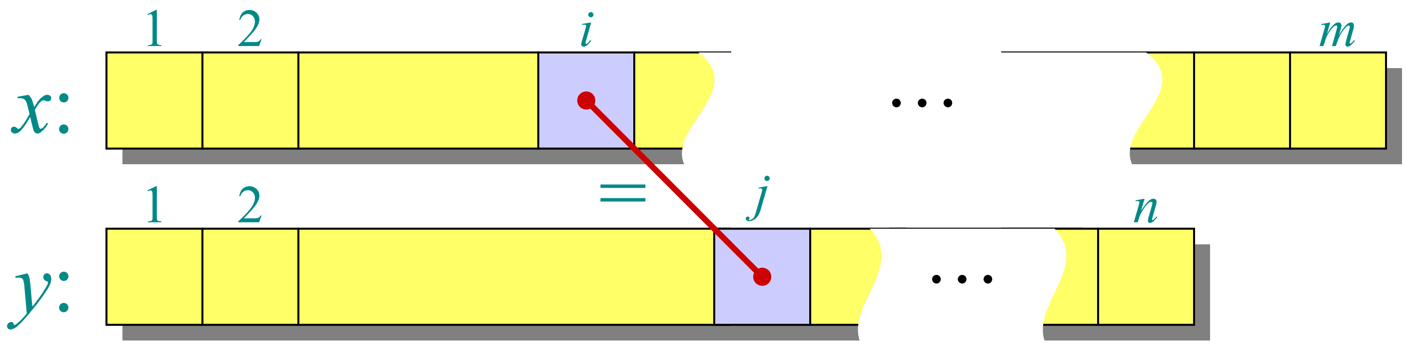


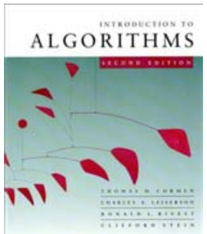
Recursive formulation

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



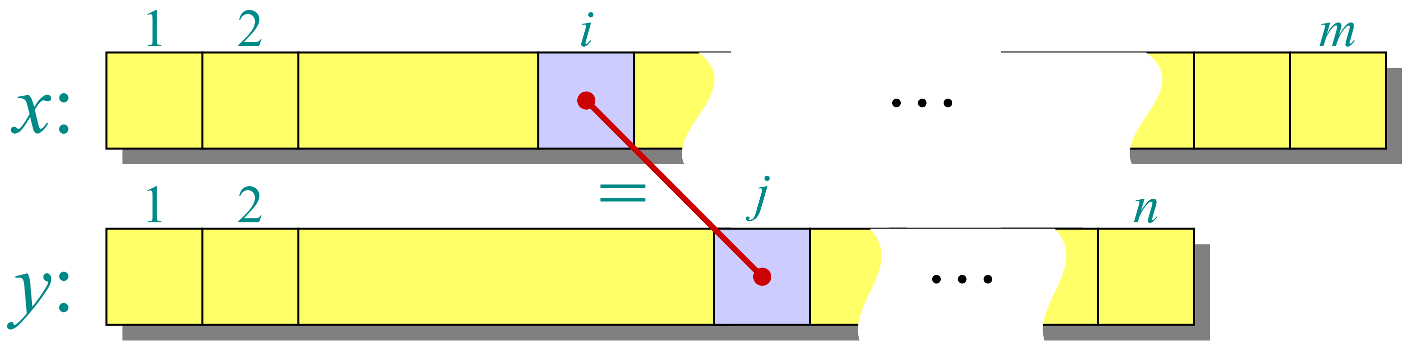


Recursive formulation

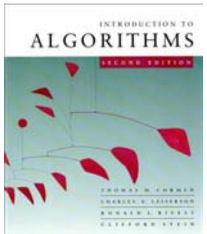
Theorem.

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

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



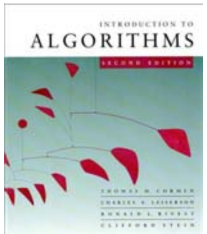
Let $z[1 \,.\,.\, k] = \mathrm{LCS}(x[1 \,.\,.\, i], y[1 \,.\,.\, j])$, where $c[i, j] = k$. Then, $z[k] = x[i]$, or else z could be extended. Thus, $z[1 \,.\,.\, k-1]$ is CS of $x[1 \,.\,.\, i-1]$ and $y[1 \,.\,.\, j-1]$.



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 $\big|w \,||\, z[k]\big| > k$. Contradiction, proving the claim.



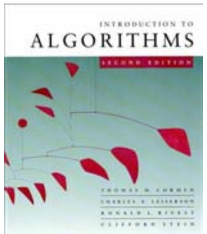
Proof (continued)

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

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 $|w \,||\, z[k]| > k$. 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. $\square$



Dynamic-programming hallmark #1

Optimal substructure

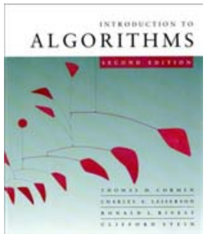
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Dynamic-programming hallmark #1

Optimal substructure

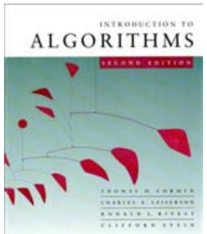
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If $z = \mathrm{LCS}(x, y)$, 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)
    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$(x, y, i, j–1)\}$



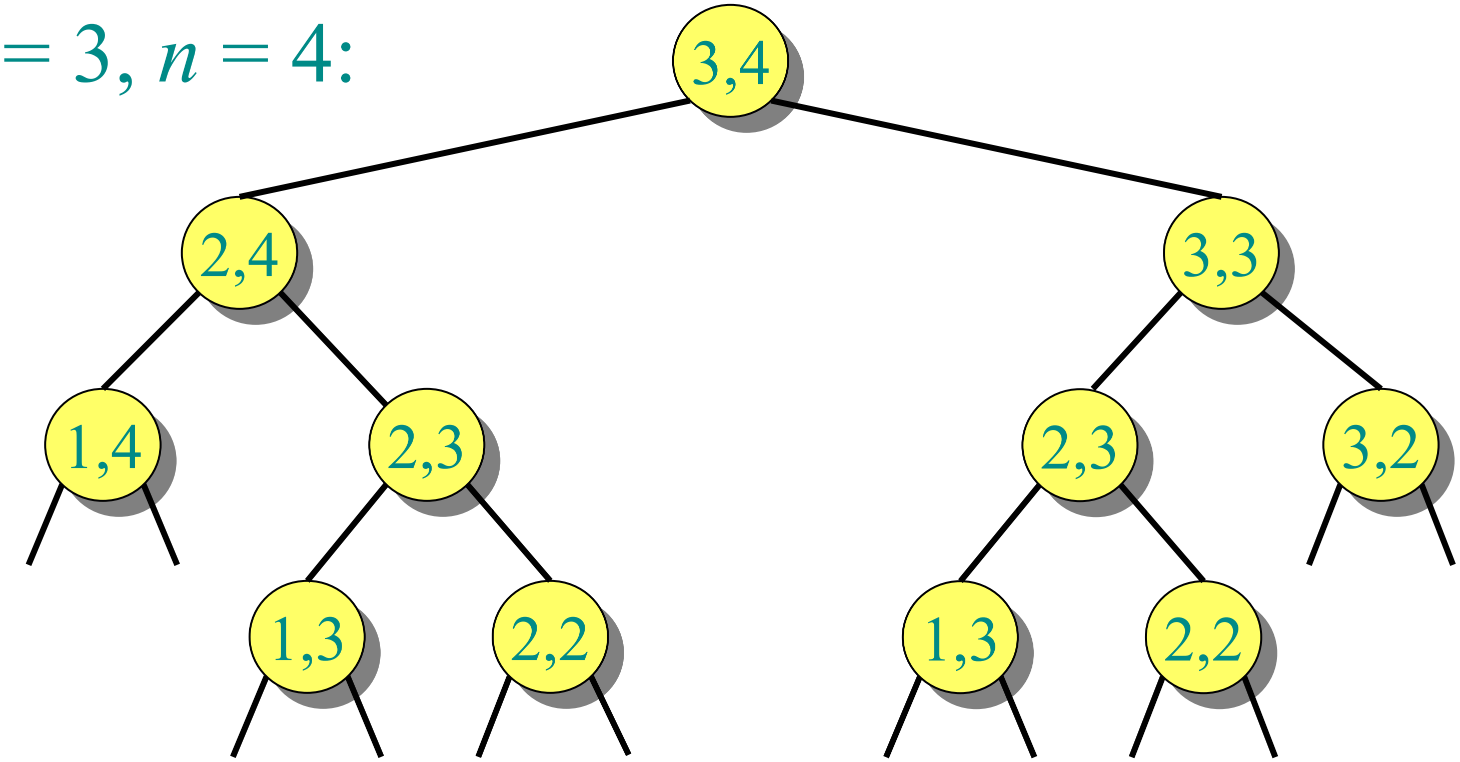
Recursive algorithm for LCS

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

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

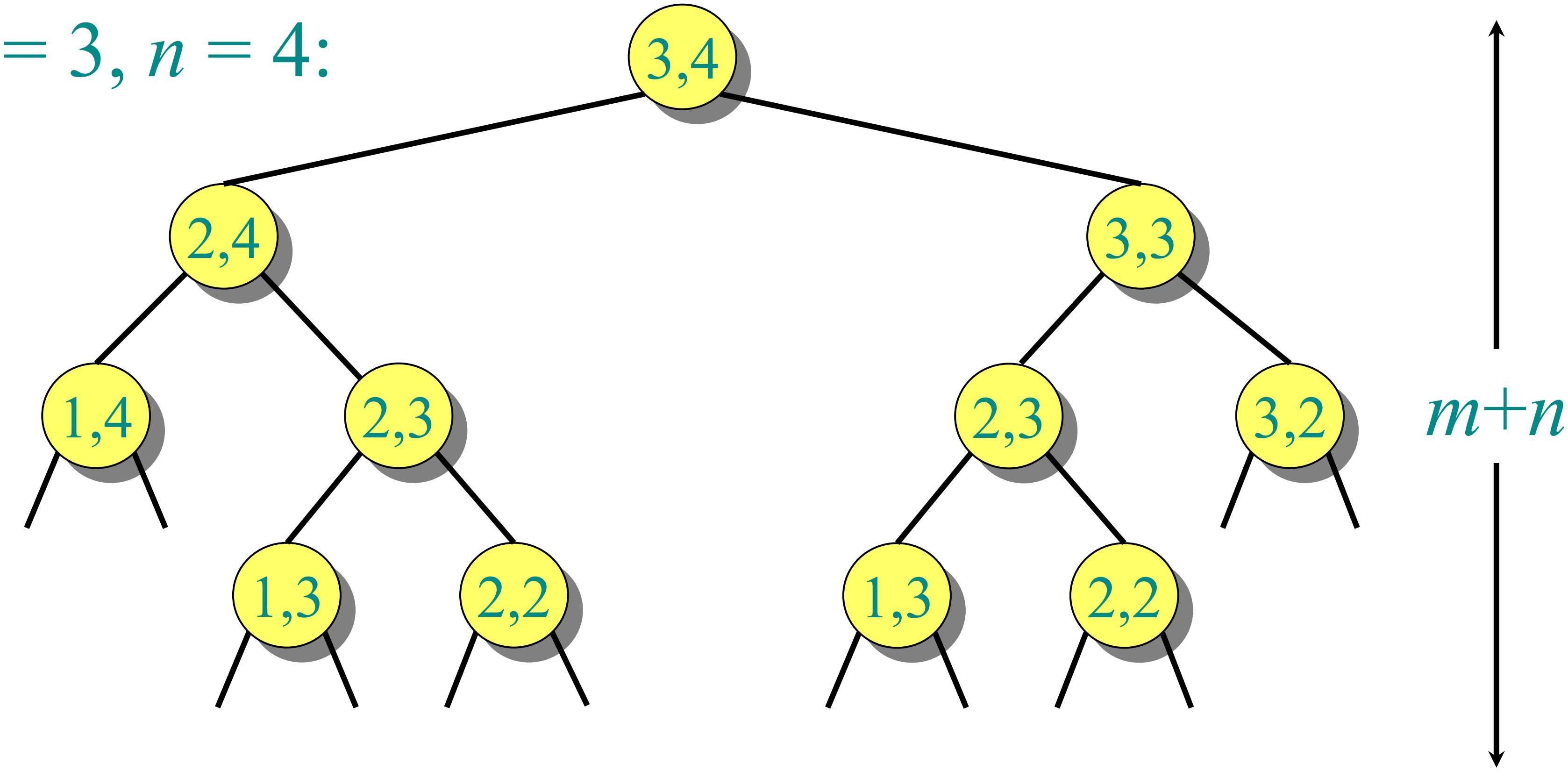
Recursion tree

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

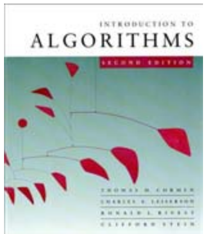


Recursion tree

$m = 3, n = 4$:

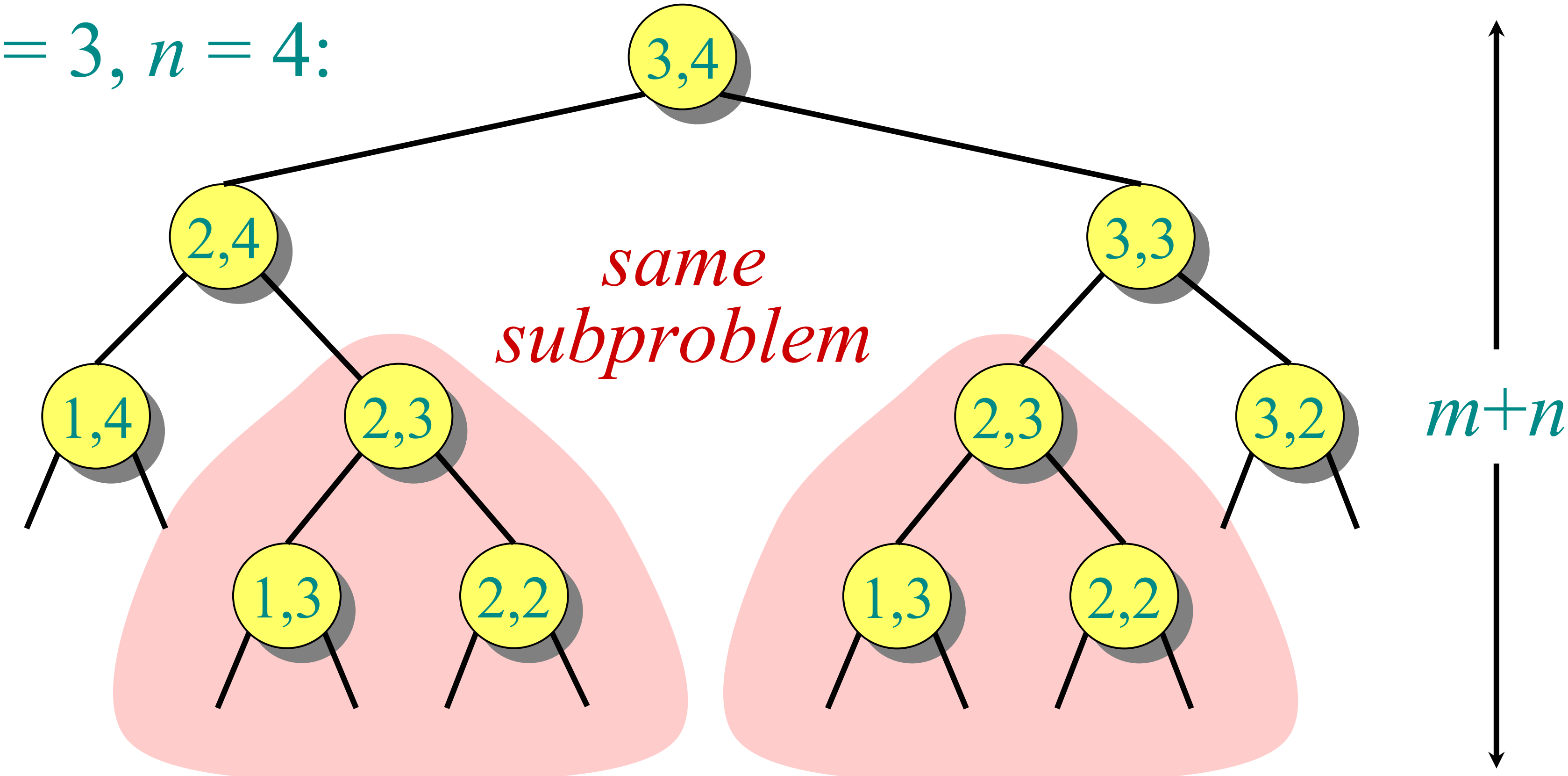


Height $= m + n \Rightarrow$ work potentially exponential.

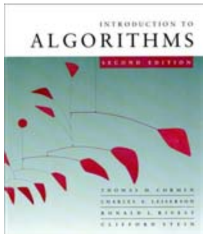


Recursion tree

$m = 3,\ n = 4$:



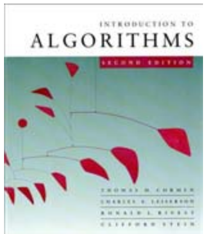
Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!



Dynamic-programming hallmark #2

Overlapping subproblems

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

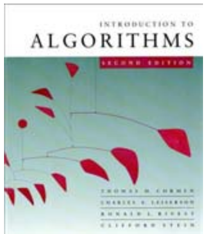


Dynamic-programming hallmark #2

Overlapping subproblems

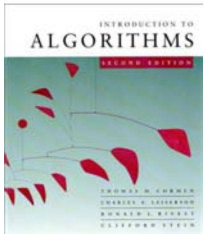
A recursive solution contains a "small" number of distinct subproblems repeated many times.

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)

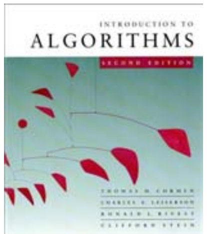
    if $c[i, j] = \text{NIL}$

        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



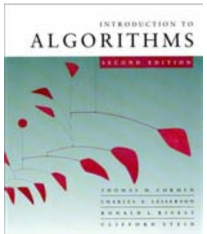
Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the 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 $= \Theta(mn)$.

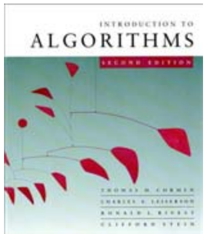


Dynamic-programming algorithm

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



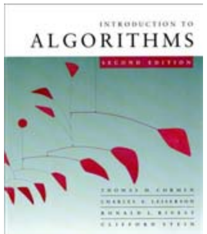
Dynamic-programming algorithm

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

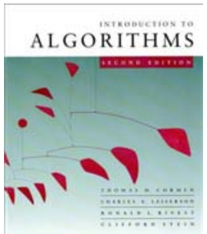
IDEA:

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.

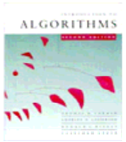
Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

Space = $\Theta(mn)$.

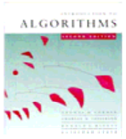
Exercise:
$O(\min\{m, n\})$.

		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



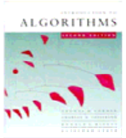
Computing the length of an LCS

```
LCS-LENGTH(X,Y)
m ← length[X]
n ← length[y]
for i ← 1to m
     do c[i,0] ← 0
for j ←0 to n
     do c[0,j] ← 0
for ← 1 to m
     do for j ← 1 to n
               do if x[i] = y[j]
                   then c[i, j] ← c[i-1, j-1] + 1
                          b[i, j] ← “↖”
                   else if c[i-1, j]≥c[i, j-1]
                          then 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
```



Constructing an LCS

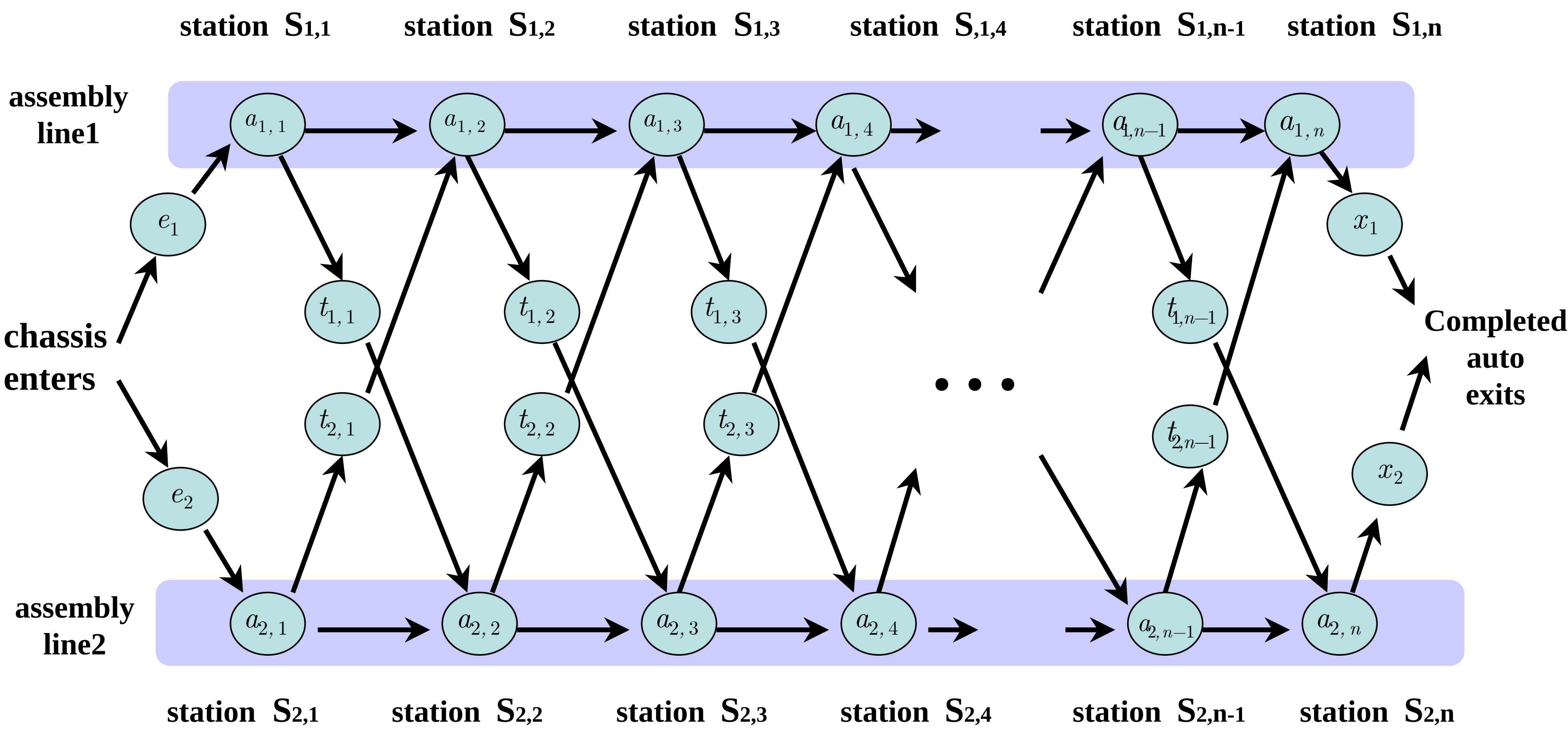
PRINT-LCS(b, X, i, j)
if i = 0 **or** j = 0
    then return
if b[i, j] **=** "↖"
    then PRINT-LCS(b, X, i-1, j-1)
            print x[i]
elseif b[i, j] = "↑"
    then PRINT-LCS(b, X, i-1, j)
else PRINT-LCS(b, X, i, j-1)



Assembly-line scheduling

Manufacturing problem to find the fastest way through a factory.

- Produces automobiles in a factory that has two assembly lines.
- An automobile chassis enters each assembly line and parts are added to it at the stations.
- Each assembly line: n stations, numbered $j = 1,2, \ldots, n$.
- Denote jth station on line i ($i = 1, 2$) by $S_{i,j}$.
- The jth station on line 1 ($S_{1,j}$) performs the same function as the jth station on line 2 ($S_{2, j}$).
- The time required at each station varies, even between stations at the same position on two different lines.
  - denote time required at $S_{i,j}$ by $a_{i,j}$
  - entry time e_i
  - exit time x_i
  - transfer time at $S_{i,j}$ is $t_{i,j}$.

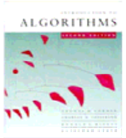


station $S_{1,1}$
station $S_{1,2}$
station $S_{1,3}$
station $S_{,1,4}$
station $S_{1,n-1}$
station $S_{1,n}$
assembly line1
chassis enters
assembly line2
$a_{1,1}$
$a_{1,2}$
$a_{1,3}$
$a_{1,4}$
$a_{1,n-1}$
$a_{1,n}$
e_1
e_2
$t_{1,1}$
$t_{2,1}$
$t_{1,2}$
$t_{2,2}$
$t_{1,3}$
$t_{2,3}$
$t_{1,n-1}$
$t_{2,n-1}$
x_1
x_2
Completed auto exits
$a_{2,1}$
$a_{2,2}$
$a_{2,3}$
$a_{2,4}$
$a_{2,n-1}$
$a_{2,n}$
station $S_{2,1}$
station $S_{2,2}$
station $S_{2,3}$
station $S_{2,4}$
station $S_{2,n-1}$
station $S_{2,n}$

Assembly-line scheduling

Brute force way of minimizing the time through the factory.

- There are 2^n possible ways to choose stations.
- View the set of stations used in line 1 as a subset of $\{1, 2, \ldots, n\}$.
  - Number of all the subsets (power set) is 2^n.
  - Which is infeasible for large n.



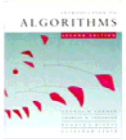
Assembly-line scheduling

Optimal substructure

of the fastest way through the factory.

- *An optimal solution to a problem contains within it an optimal solution to subproblems*.

That is, the fastest way through station $S_{i,j}$ contains within it the fastest way through either $S_{1,j-1}$ or $S_{2,j-1}$).



Assembly-line scheduling

Recursive solution of the fastest way through the factory.

- Define the value of an optimal solution recursively in terms of the optimal solutions to subproblems.

$$f^* = \min(f_1[n] + x_1,\ f_2[n] + x_2)$$

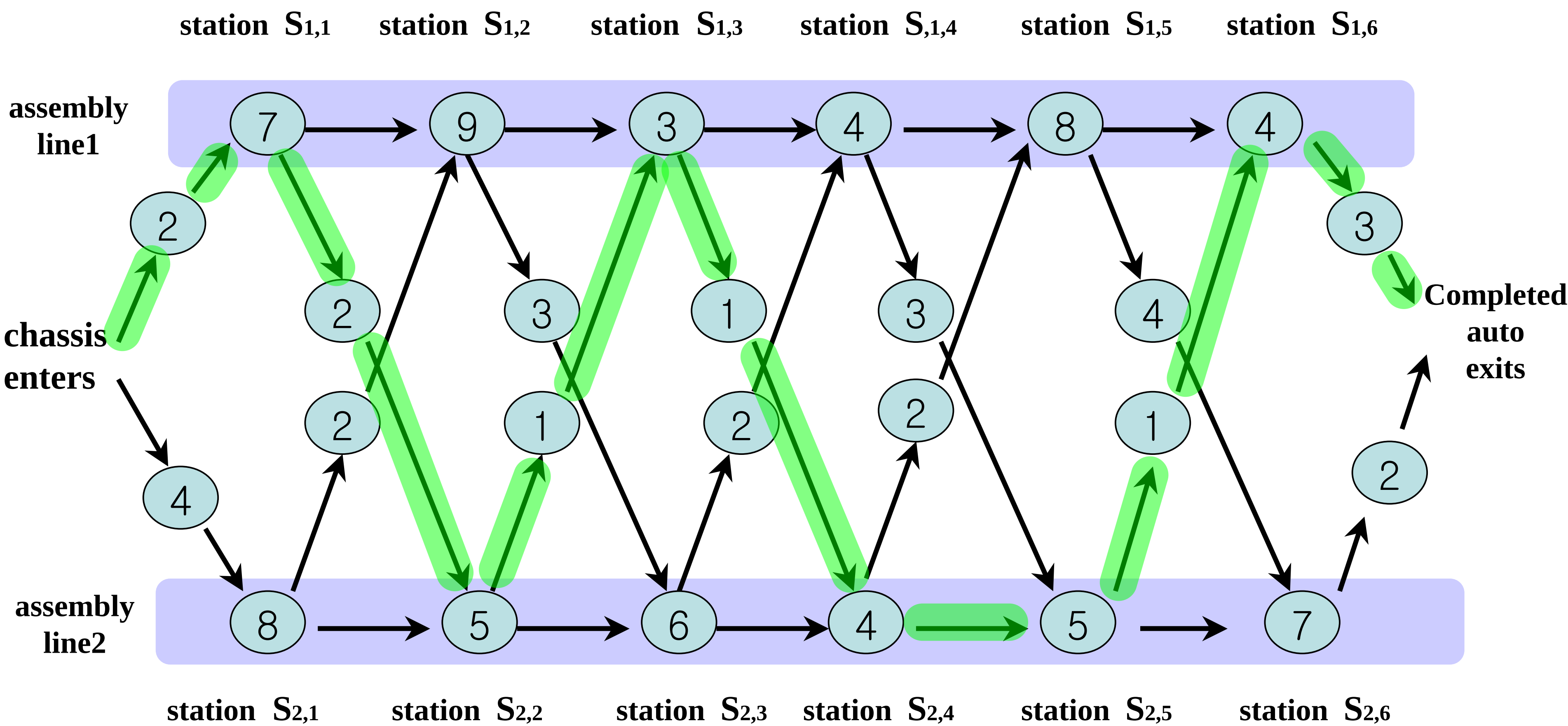
Fast time to get a chassis all the way through the factory.

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1, \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

$l_i[j]$: line number, 1 or 2, whose station $S_{i,\,j-1}$ is used in a fastest way through $S_{i,\,j}$.

l^*: line number whose station n is used .

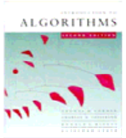


j	1	2	3	4	5	6
$f_1[j]$	**9**	**18**	**20**	**24**	**32**	**35**
$f_2[j]$	**12**	**16**	**22**	**25**	**30**	**37**

$f^* = 38$

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

$l^* = 1$



Assembly-line scheduling

Overlapping subproblems

of recursive solution of the fastest way through the factory.

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

- The recursive algorithm: running time is proportional to 2^n.
  - let $r_i(j)$: the number of references made to $f_i[j]$ in recursive algorithm.

    $r_1(n) = r_2(n) = 1$

    $r_1(j) = r_2(j) = r_1(j + 1) + r_2(j + 1)$ for $j = 1, 2, \ldots, n-1$

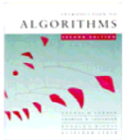
    $r_i(j) = 2^{n-j}$
  - Thus, $f_1[1]$ alone is referenced 2^{n-1} times.
  - The total number of references to all $f_i[j]$ values is $\Theta(2^n)$.

Assembly-line scheduling

- *Computing the fastest times.*

- We can do much better if we compute the $f_i[j]$ values in a different order from the recursive way.

  - $f_1[j]$ depends only on $f_1[j-1]$ and $f_2[j-1]$.
  - compute $f_i[j]$ in order of increasing station numbers j.
  - The time it takes is $\Theta(n)$.



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 $f_2[j] \leftarrow f_1[j-1] + + t_{1,j-1} + a_{2,j}$

      $l_2[j] \leftarrow 1$

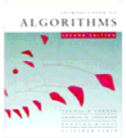
If $f_1[n] + x_1 \leq f_2[n] + x_2$

 then $f^* \leftarrow f_1[n] + x_1$

   $l^* \leftarrow 1$

 else $f^* \leftarrow f_2[n] + x_2$

   $l^* \leftarrow 2$

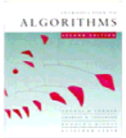


Assembly-line scheduling

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

PRINT-STATIONS procedure produce the output

line 1, station 6
line 2, station 5
line 2, station 4
line 1, station 3
line 2, station 2
line 1, station 1



Dynamic Programming

- *Divide-and-conquer* algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.

- Dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems.

- Applicable condition of dynamic programming
  - optimal substructure
  - overlapping subproblems