

CMPS 6610/4610 – Fall 2016

Dynamic Programming

Carola Wenk

Slides courtesy of Charles Leiserson with changes and additions 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

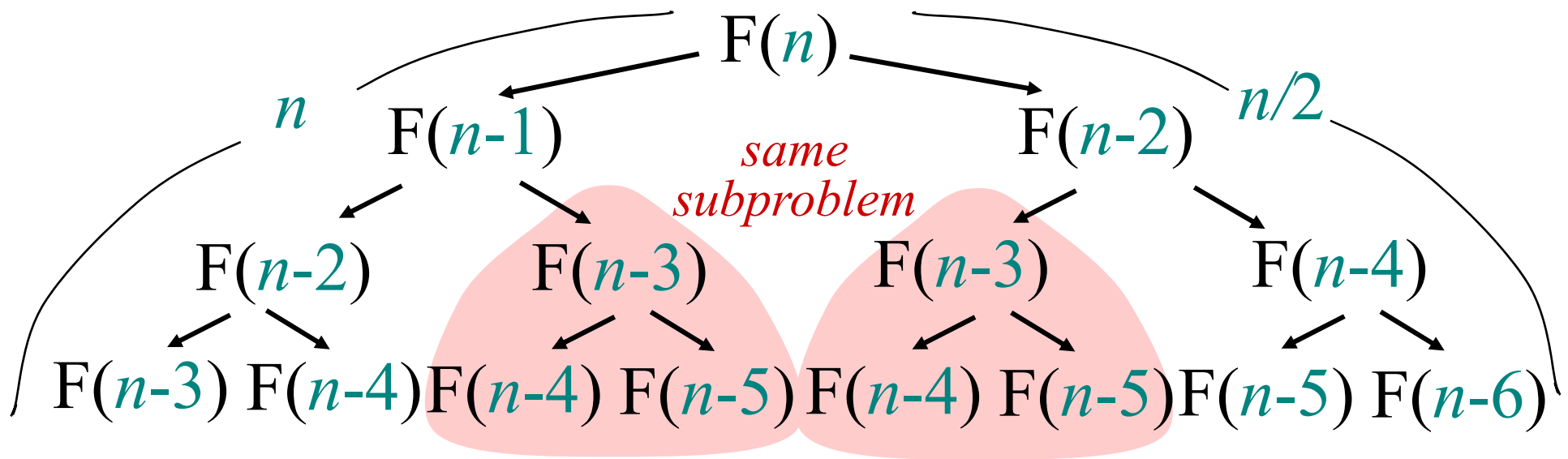
Optimal substructure

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

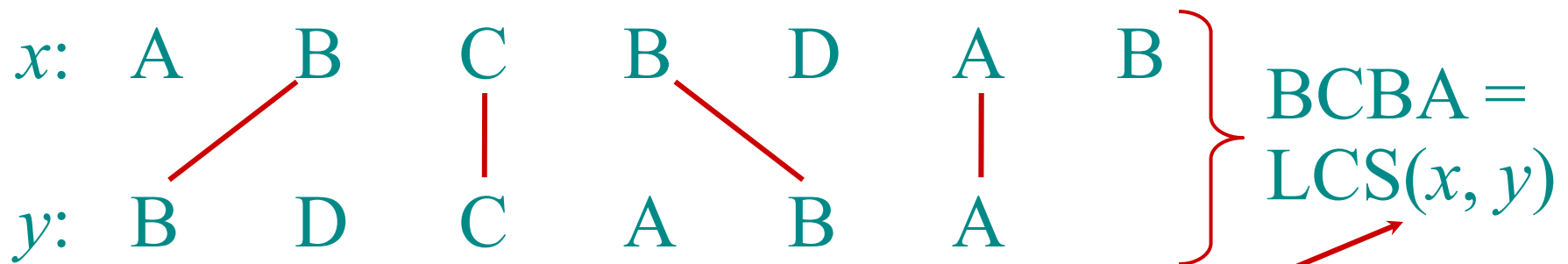
- Time = $\Theta(n)$, space = $\Theta(n)$

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”



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 .

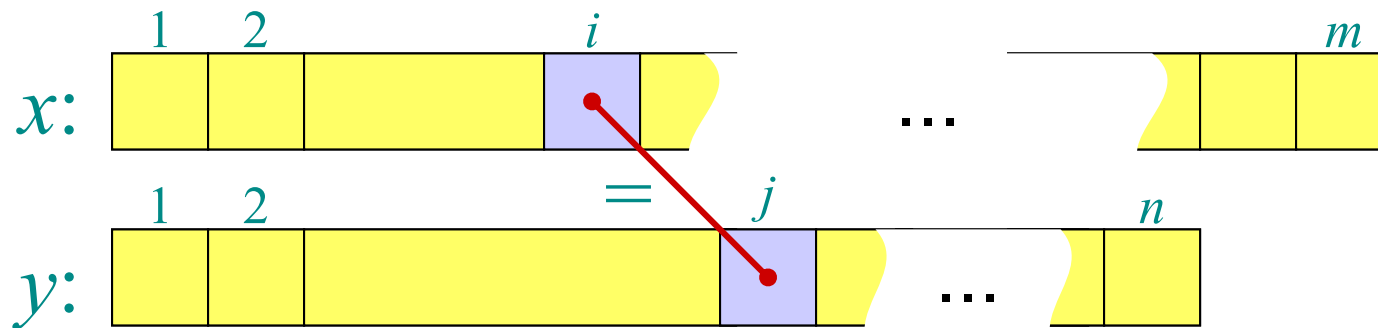
- Define $c[i, j] = |\text{LCS}(x[1 \dots i], y[1 \dots j])|$.
- Then, $c[m, n] = |\text{LCS}(x, y)|$.

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.*

 *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 .

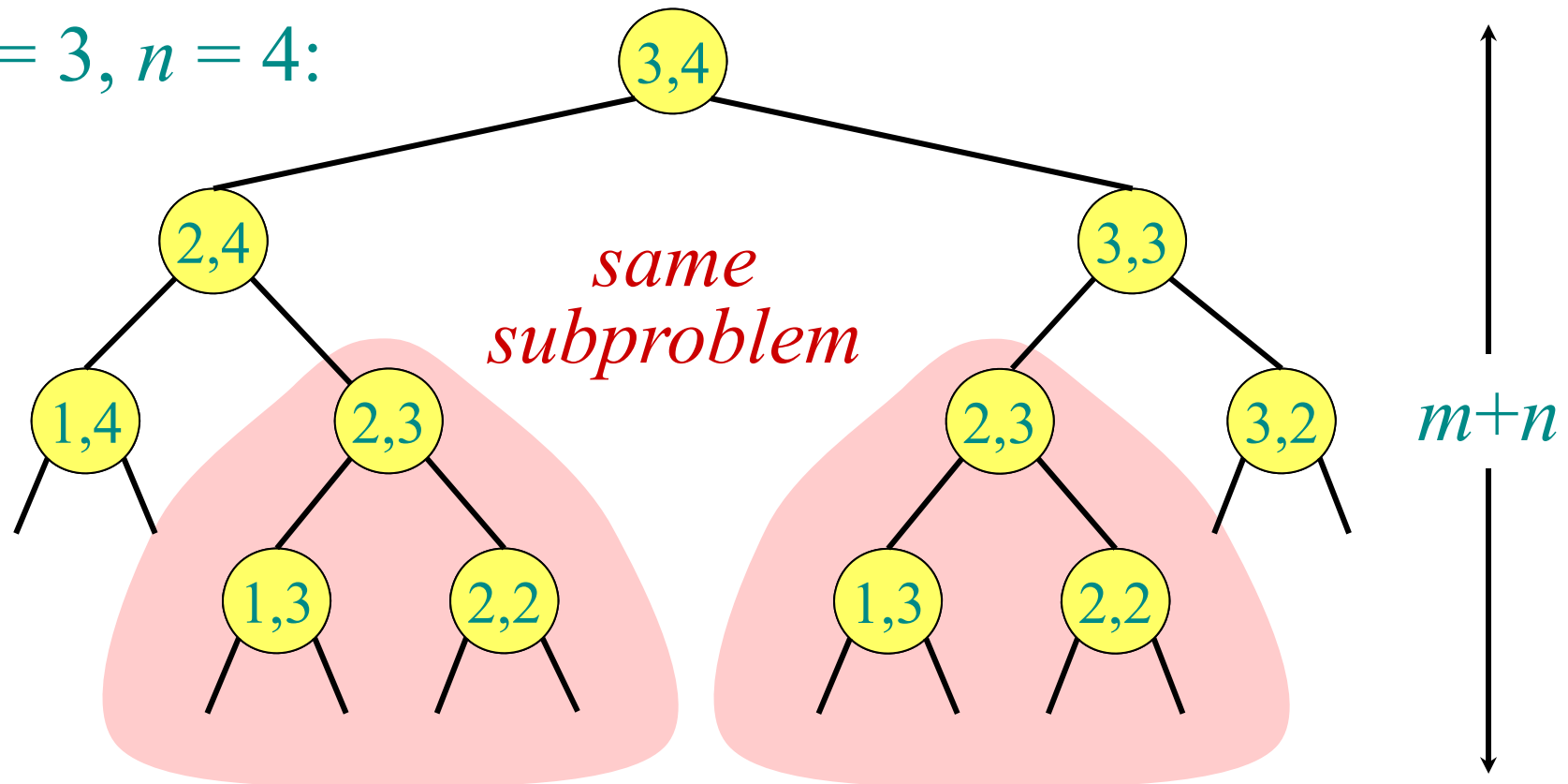
Recursive algorithm for LCS

```
LCS( $x, y, i, j$ )  
  if ( $i=0$  or  $j=0$ )  
     $c[i, j] \leftarrow 0$   
  else if  $x[i] = y[j]$   
     $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$   
  else  $c[i, j] \leftarrow \max\{\text{LCS}(x, y, i-1, j),$   
     $\text{LCS}(x, y, i, j-1)\}$   
  return  $c[i, j]$ 
```

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 distinct LCS subproblems are all the pairs (i,j) . The number of such pairs 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.

LCS(x, y, i, j)

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

if ($i=0$ or $j=0$)

$c[i, j] \leftarrow 0$

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

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

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

return $c[i, j]$

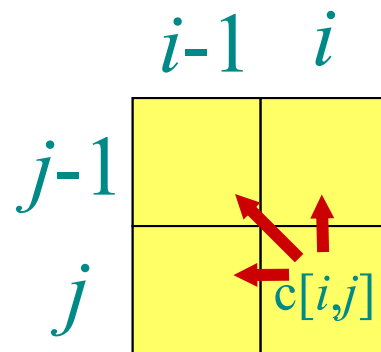
*same
as
before*

Space = time = $\Theta(mn)$; constant work per table entry.

Recursive formulation

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

c :



Bottom-up dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

$y \downarrow x \rightarrow$	A	B	C	B	D	A	B
	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

Bottom-up dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by back-tracking.

Space = $\Theta(mn)$.

Exercise:

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

$x \rightarrow$	A	B	C	B	D	A	B
$y \downarrow$	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