CMPS 2200 – Fall 2017

Dynamic Programming II

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**

Longest Common Subsequence

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 $y[1 \,..\, n]$.

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

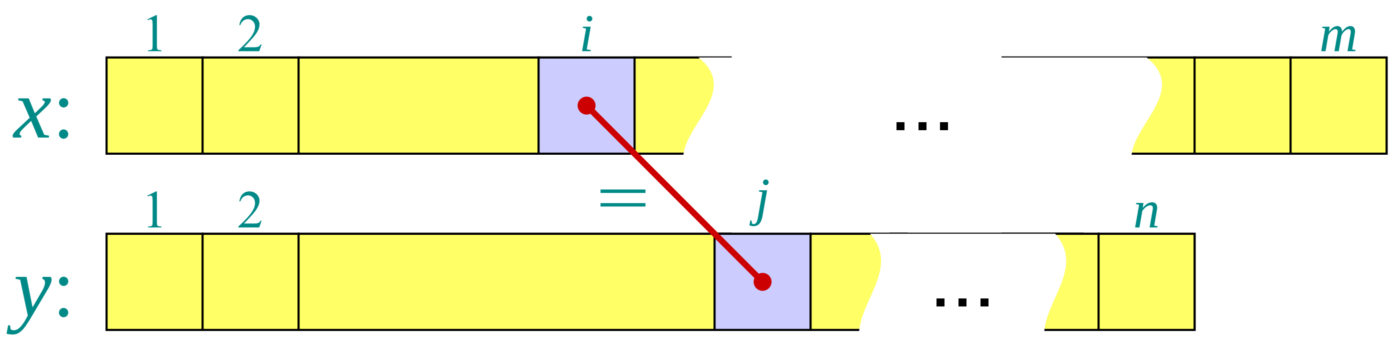
Recursive formulation

Theorem.

Longest common subsequence

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

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 $\left|w \,||\, z[k]\right| > k$. 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.

→ ***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.

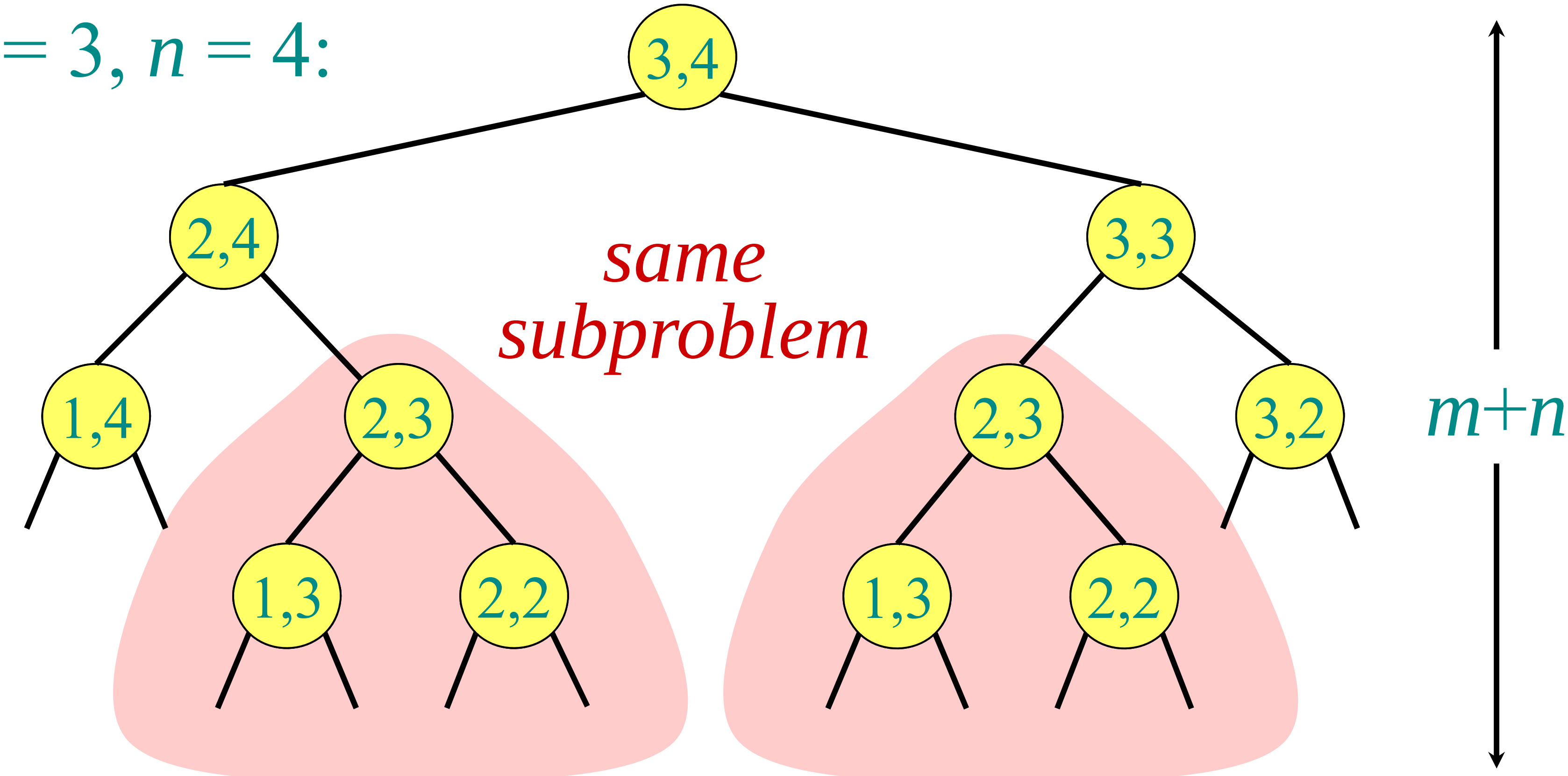
Recursive algorithm for LCS

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

$m = 3, n = 4$:



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_mem(x, y, i, j)
    if $c[i, j] = \text{null}$
        if $(i=0 \text{ or } j=0)$
            $c[i, j] \leftarrow 0$
        else if $x[i] = y[j]$
            $c[i, j] \leftarrow$ LCS_mem$(x, y, i–1, j–1) + 1$
        else $c[i, j] \leftarrow \max\{$LCS_mem $(x, y, i–1, j)$,
                            LCS_mem $(x, y, i, j–1)\}$
    return $c[i, j]$

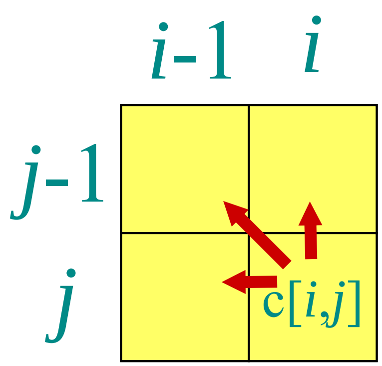
same as before

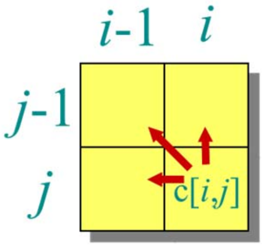
Space = time = $\Theta(mn)$; constant work per table entry.

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.

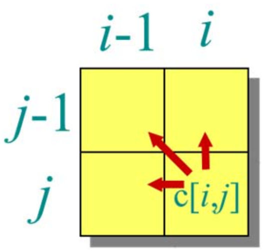
Time = $\Theta(mn)$.

$y \downarrow$ \ $x \rightarrow$		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	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

Bottom-up DP

```
LCS_bottomUp(x[1..m], y[1..n])
    for (i=0; i≤m; i++) c[i,0]=0;
    for (j=0; j≤n; j++) c[0,j]=0;
    for (j=1; j≤n; j++)
        for (i=1; i≤m; i++)
            if x[i] = y[ j] {
                c[i, j] ← c[i-1, j-1]+1
                arrow[i,j]="diagonal";
            } else { // compute max
                if (c[i-1, j]≥ c[i, j-1]){
                    c[i, j] ← c[i-1, j]
                    arrow[i,j]="left";
                } else{
                    c[i, j] ← c[i, j-1]
                    arrow[i,j]="right";
                }
            }
    return c and arrow
```

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



Bottom-up dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by back-tracking.

Space = $\Theta(mn)$.

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

$y \downarrow$ \ $x \rightarrow$		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