Dynamic Programming

Design technique (like D&C)

Example Problem: Longest common subsequence (LCS)

Given: 2 sequences $x[1..m]$, $y[1..n]$.

Find: sequence $z[1..k]$ s.t.

- z is subsequence of x and y
- z has maximal length

Note: several z may exist

X: A B C B D A B

Y: B D C A B A

LCSs: B C B A
B C A B
B D A B

Brute force algorithm

for every subsequence z of $x[1..m]$
check if z is a subsequence of $y[1..n]$

Analysis

- check: match each character of z to next character of y

  $\Rightarrow O(n)$

- Number of subsequences of x
  = number of subsets of $\{1, \ldots, m\}$
  $= 2^m$

$\Rightarrow$ Total running time $= O(n \cdot 2^m)$ 

Simplification

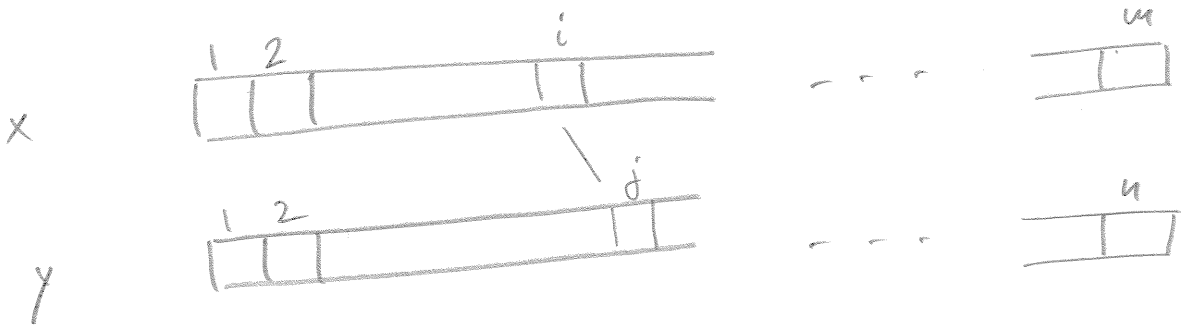
1. Find length of LCS(x,y)
2. Extend alg 1) to find LCS(x,y)

$|s|$:= length of string s

Strategy: considers prefixes of x and y

$$c[i,j] := |LCS(x[1..i], y[1..j])|$$

$$\Rightarrow c[m,n] = |LCS(x,y)|$$



What about $c[i,j]$?

- $x[i] \neq y[j]$

  Let $z[1..k]$ be a LCS $(x[1..i], y[1..j])$.

  Then $x[i] \neq z[k]$ or $y[j] \neq z[k]$

  Thus:

  $$c[i,j] = \max\{c[i,j-1], c[i-1,j]\}$$
  $$\text{if } x[i] \neq x[j]$$

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

  $\Rightarrow c[i,j] = c[i-1, j-1] + 1$

  Proof: Let $z[1..k]$ be as before.

  Then, $z[k] = x[i] = y[j]$.

  If not, z could be extended by $x[i]$.

  $\Rightarrow$ $z[1..k-1]$ is CS of $x[1..i-1], y[1..j-1]$

Claim: $z[1..k-1]$ is LCS $(x[1..i-1], y[1..j-1])$

Suppose w is longer CS, that is, $|w| > k-1$.

$\Rightarrow$ $w \cdot z[k]$ is a CS of $x[1..i]$ and $y[1..j]$ with length $> k$

($\cdot$: concatenation)

↯

We have shown

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

Dynamic Programming Hallmark #1:

Optimal Substructure: An optimal solution to a problem (instance) contains optimal solutions to subproblems

Example: $z = LCS(x, y)$

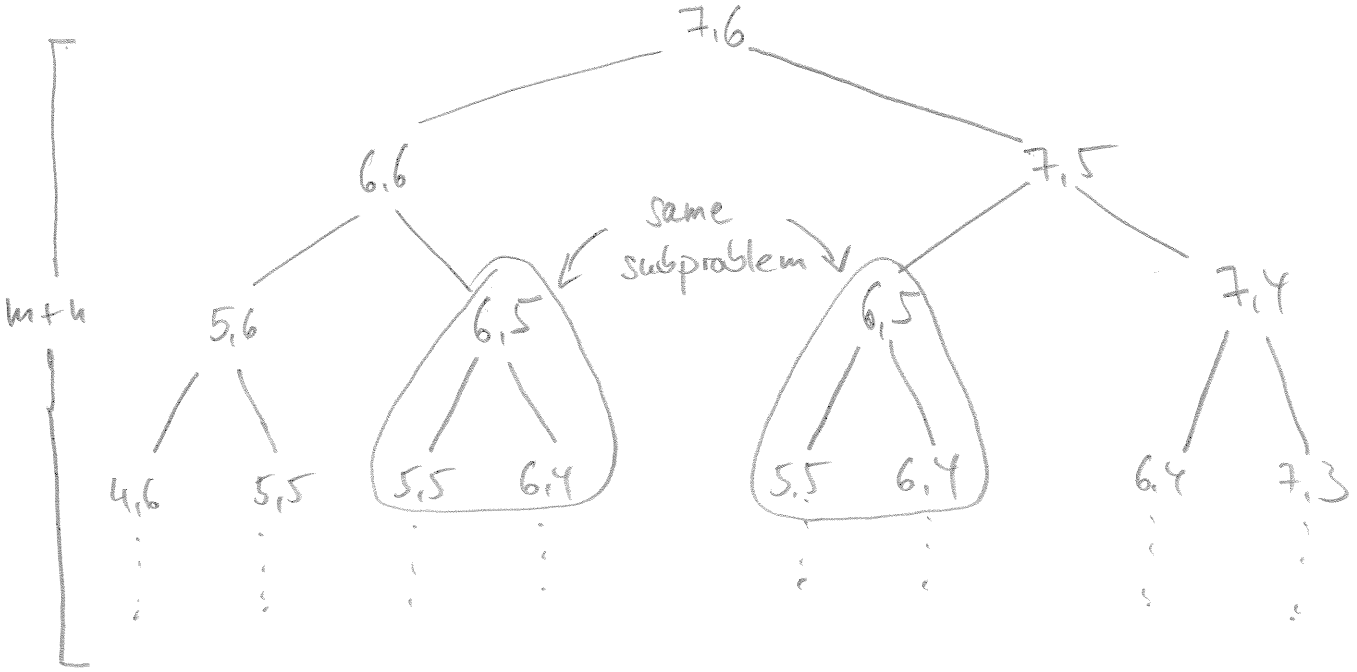
$\Rightarrow$ if z' is prefix of z
then there are prefixes x' of x, y' of y
s.th $z' = LCS(x', y')$

Recursive algorithm

```
LLCS (x, y, i, j)
  // length of longest common subsequence

  if  i = 0  or  j = 0
    then  return 0

  else if  x[i] = y[j]
    then  return LLCS(x, y, i-1, j-1) + 1

  else return max { LLCS(x, y, i-1, j),
                    LLCS(x, y, i, j-1) }
```

Recursion tree: Worst case, $x[i] \neq y[j]$ f.a. i, j,

$m = 7,\ n = 6$



Height = $m+n$ $\Rightarrow$ exponential size

$\Rightarrow$ repeated work

Dynamic Programming Hallmark #2

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

LLCS's (and LCS's) subproblem space contains $m \cdot n$ distinct subproblems

Memoization algorithm ← make a "memo"

Array of int c[0..m, 0..n] // the memos

initialize:
for i = 0 to m do
  c[i,0] := 0
for j = 1 to n do
  c[0,j] := 0

```
LLCS(x, y, i, j)
  if c[i,j] = nil
    then if X[i] = Y[j]
           then c[i,j] := LLCS(x, y, i-1, j-1) + 1
           else c[i,j] := max{LLCS(x, y, i-1, j),
                              LLCS(x, y, i, j-1)}
  return c[i,j]
```

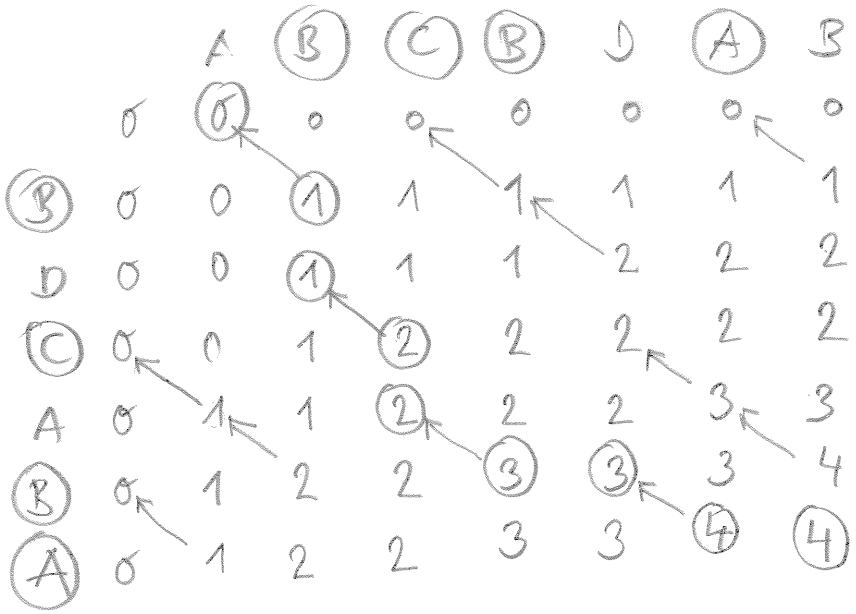
Space = $\Theta(m \cdot n)$

Time: up to two lookups per new entry

$\Rightarrow \Theta(mn)$

Dynamic programming algorithm

Idea: compute the table bottom up




```
LLCS(X,Y)    // without initialization

for i := 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) // diagonal step
          print x[i]
  else if b[i,j] = "↑"
     then Print-LCS(b, x, i-1, j) // upward step
     else Print-LCS(b, x, i, j-1) // horizontal
                                  step
```

Code improvements

- Eliminate b-table
- Use only one full row to compute LCS(x, y)