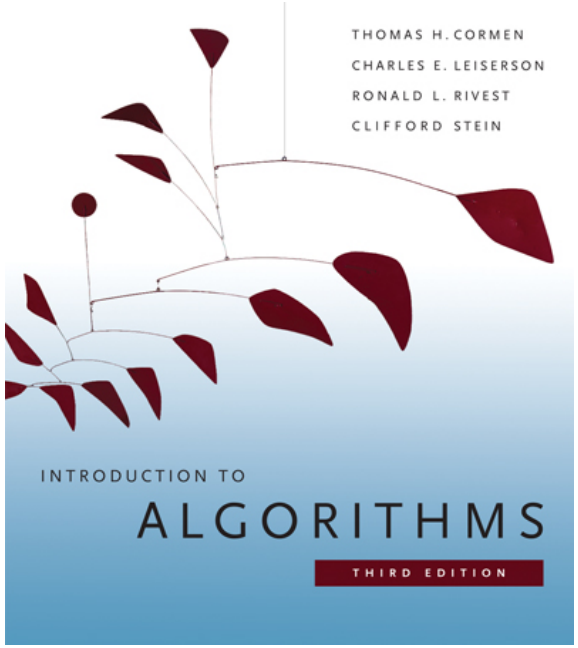
6.006- *Introduction to Algorithms*



Lecture 20

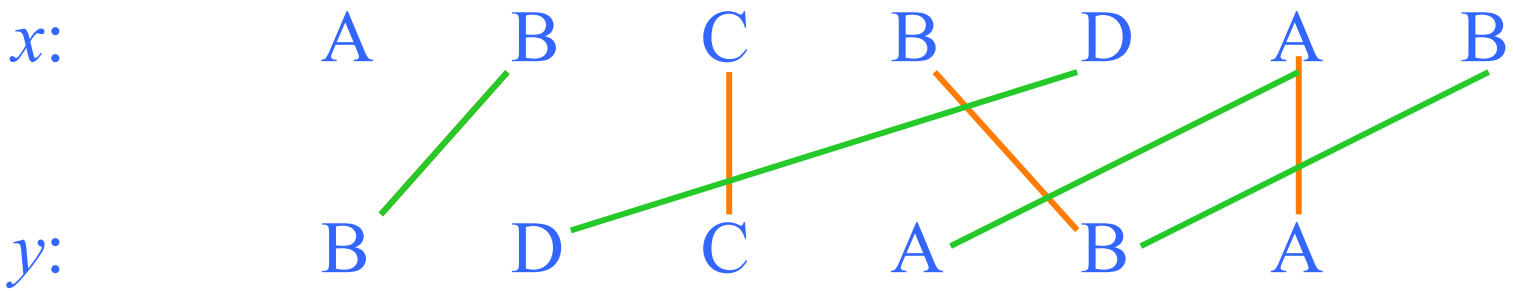
Prof. Constantinos Daskalakis

Lecture overview

- longest common subsequence:
  - the bottom-up approach
  - reconstructing the LCS: back-pointers
- knapsack

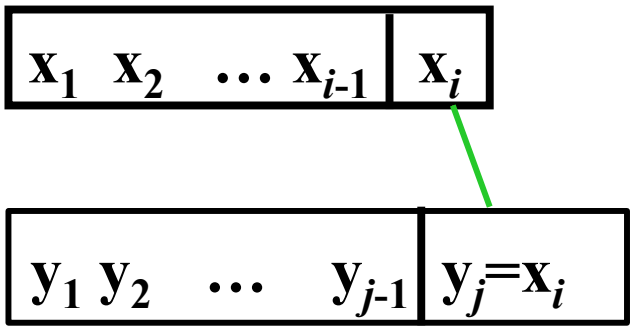
Longest Common Subsequence

- given two sequences x[1..*m*] and y[1..*n*], find a longest subsequence LCS(x,y) common to both:



- find the length of a longest common subsequence
- DP subproblem:
- c(*i*, *j*)= length of longest common subsequence between strings x[1..*i*] and y[1..*j*]

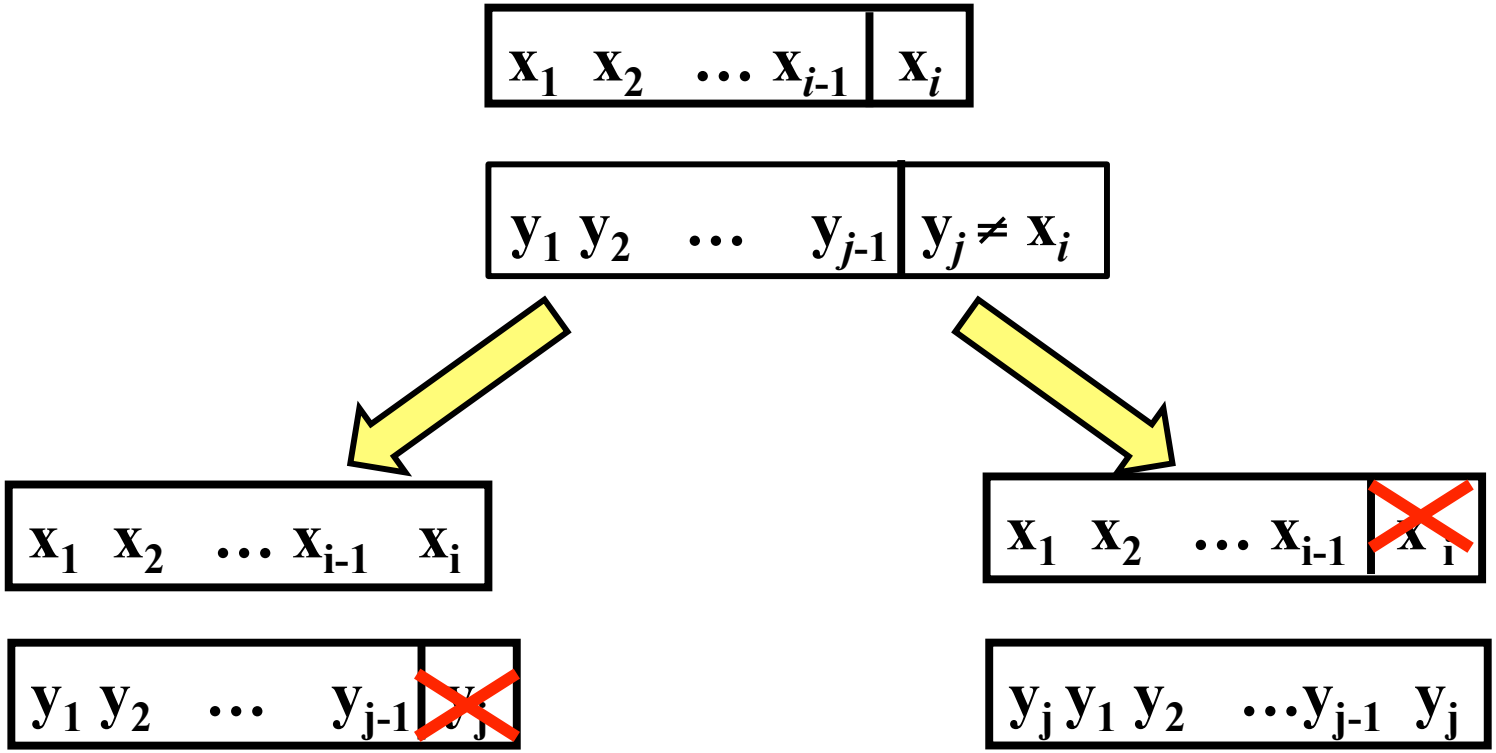
1) x[1..*i*] and y[1..*j*] end with $x_i = y_j$



I might as well match x_i and y_j and look for LCS of x[1..*i*-1] and y[1..*j*-1].
So

$$c(i, j) = c(i\text{-}1, j\text{-}1)+1, \text{ if } x_i = y_j$$

2) x[1..*i*] and y[1..*j*] end with $x_i \neq y_j$



LCS of x[1..*i*] and y[1..*j*-1]

LCS of x[1..*i*-1] and y[1..*j*]

$c(i, j)=\max\{c(i, j\text{-}1), c(i\text{-}1, j)\}$, if $x_i \neq y_j$

Solving LCS with Recursion+Memoization

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

```
memo = { }
c(i, j):
    if (i, j) in memo: return memo[i, j]
    else if i=0 OR  j=0: return 0
    else if x_i=y_j: f = c(i-1, j-1)+1
    else f = max{c(i, j-1), c(i-1, j)}
    memo[i, j]=f
    return f
return c(n,m)
```

The Bottom-Up Approach

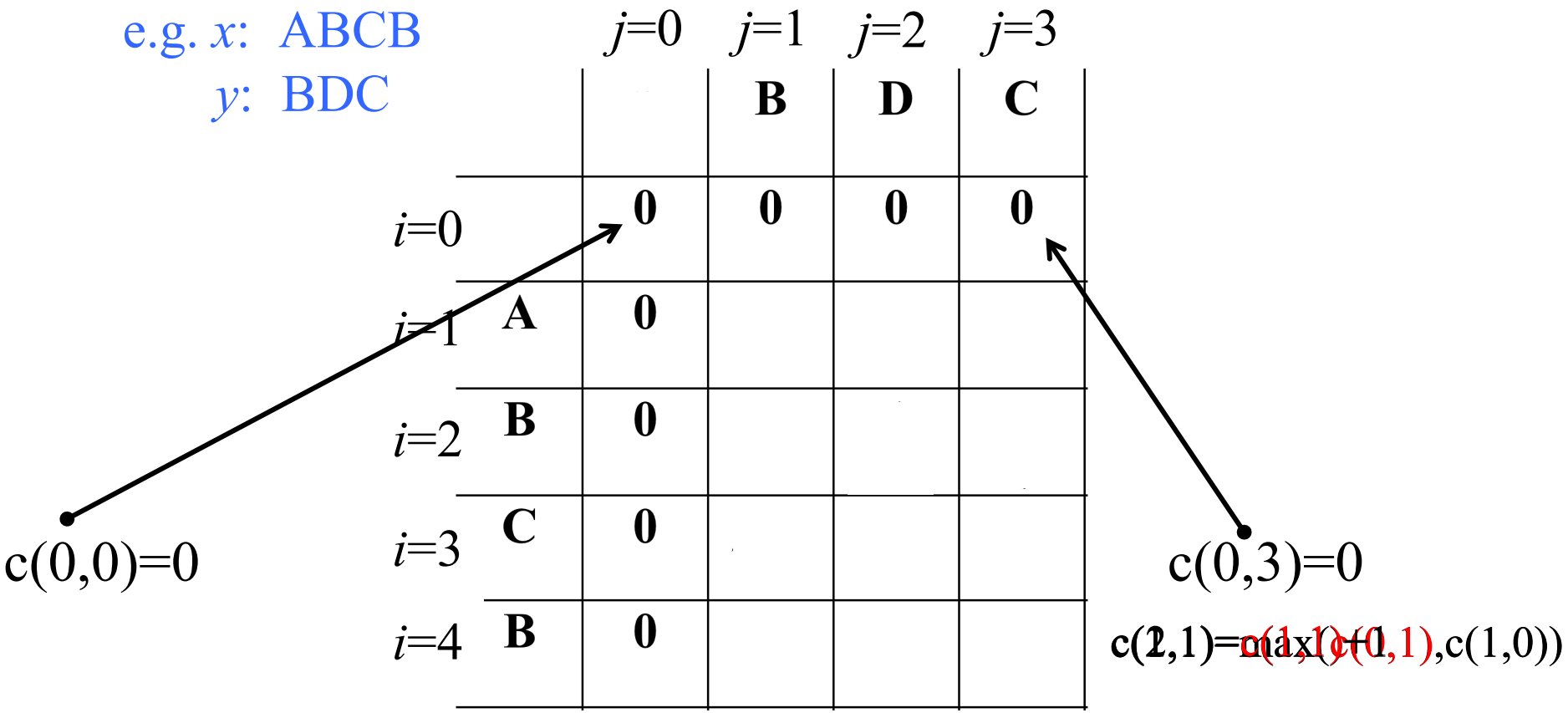
$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

"bottom-up approach": solve sub-problems in an order that allows you to never recurse

The Bottom-Up Approach

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

e.g. x: ABCB
y: BDC



The Bottom-Up Approach

Length of Longest Common Subsequence(x,y)

$m \leftarrow length[x]$
$n \leftarrow length[y]$
for $i \leftarrow 1$ **to** m
  do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ **to** n
  do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
  do for $j \leftarrow 1$ **to** n
    do if $x_i = y_j$
      then $c[i, j] \leftarrow c[i\text{-}1, j\text{-}1] + 1$
        $p[i, j] \leftarrow$ "↖"
      else if $c[i\text{-}1, j] \geq c[i, j\text{-}1]$
        then $c[i, j] \leftarrow c[i\text{-}1, j]$
          $p[i, j] \leftarrow$ "↑"
        else $c[i, j] \leftarrow c[i, j\text{-}1]$
          $p[i, j] \leftarrow$ "←"
return c and p

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

parent pointers

Why parent pointers?

- **Goal:** finding a longest common subsequence
  - p remembers if the c[*i*, *j*] computation used c[*i*-1, *j*-1], c[*i*, *j*-1], or c[*i*-1,*j*]
- Here is how they are used:
- Starting at c[*m*, *n*], look at parent pointer p[*m*,*n*]
  - if it points to (m-1,n-1), then x[m]=y[n] is part of *opt*
    - put x[m] at end of output string, jump to square (m-1,n-1) and continue building *opt* from there
  - else, build *opt* from squares (m-1,n) or (m,n-1) depending on where p[*m*,*n*] points
  - repeat

Constructing an LCS

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

initial call is PRINT-LCS (p, x, m, n)

running time: $O(m+n)$

Example

x: A B C B

y: B D C

	y_j	B	D	C
x_i	0	0	0	0
A	0	↑0	↑0	↑0
B	0	↖1	1 ←	1 ←
C	0	↑1	↑1	↖2
B	0	↖1	↑1	↑2

Lecture overview

- longest common subsequence:
  - the bottom-up approach
  - reconstructing the LCS: back-pointers
- **the DP DAG**
- knapsack

Generalization: Bottom-Up DP

- we've seen DP recurrences
  - which suggest recursive implementation
  - …with memoization to avoid re-computing intermediate results
- we've also seen "bottom up" implementations
  - order sub-problems in a way that allows answering bigger sub-problems using already computed solutions to smaller sub-problems
- how to get a good ordering?

The DP DAG

- define a graph representing DP
  - sub-problems are vertices
  - edge $x \rightarrow y$ if problem x depends on problem y
- what order of problem solving works?
  - need order where x follows y if $x \rightarrow y$
  - Topological Sort!
  - can do so if graph is a DAG
  - what if not?
    - cyclic problem dependency
    - can't use DP

Lecture overview

- longest common subsequence:
  - the bottom-up approach
  - reconstructing the LCS: back-pointers
- the DP DAG
- **knapsack**

Knapsack Problem

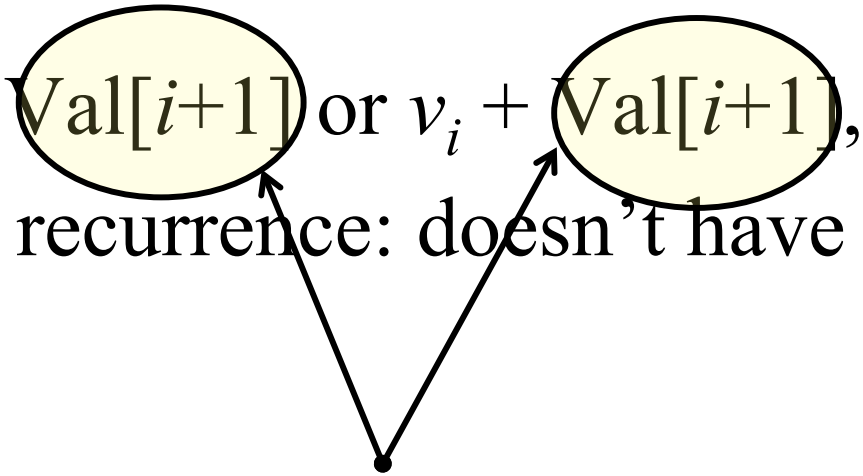
- Knapsack (or cart) of size S
- Collection of n items; item i has size s_i and value v_i
- Goal: choose subset with:
  - $\Sigma_i\, s_i < S$ (feasible, i.e. fits in knapsack)
  - maximize $\Sigma_i\, v_i$
- Ideas?
  - try all possible subsets: 2^n
  - greedy?
    - choose items maximizing value ?
    - choose items maximizing value/size
      - great, but what if items don't exactly fit (non-divisible items)?

Some bad and better news

- For arbitrary sizes, Knapsack is hard (NP-hard)
  - no polynomial time algorithm in 40 years of trying
  - it's exactly as hard as several thousand other important problems
  - and we haven't been able to find polynomial time algorithms for them for 40 years of trying either
  - most people think there is none
- Better news:
  - There is a DP algorithm if sizes are integers from a small range

First attempt for DP Algorithm

- subproblem?
  - Val[i] = Best value obtained if only items[i:n] were available to choose from
- recurrence?
  - Val[i] = best of Val[i+1] or v_i + Val[i+1],
- not a well-defined recurrence: doesn't have enough info



namely, in a correct recursion these should be different values

Second Attempt

- Solve a more complicated problem
  - of which the initial problem is a special case
- Val[i, X] = max value if one can choose from items [i : n] and available size is X
- Recurrence for Val[i, X]:
  - if $s_i > X$, then can't include i, so Val[i, X] = Val[i+1, X]
  - otherwise:
    - Val[i, X] = max(Val[$i + 1$, X], v_i + Val[$i + 1$, $X - s_i$])
- OPT = Val[1, S]

Analysis

$$\text{Val}[i, X] = \begin{cases} \text{Val}[i+1, X], & \text{if } s_i > X \\ \\ \max(\text{Val}[i+1, X], v_i + \text{Val}[i+1, X - s_i]) \end{cases}$$

- Is the recurrence a DAG?
  - yes, each problem depends on bigger *i* and smaller *X*
  - compute by decreasing *i* and increasing *X*
- Runtime?
  - O(*n S*) subproblems and work per subproblem is O(1)
  - So total time: O(*n S*)
- Is this polynomial?
- Looks polynomial but it isn't: to describe *S* need $\log_2 S$ bits
- "Pseudo-polynomial time": exponential dependence on numerical inputs, but polynomial dependence on everything else