Dynamic Programming

Dynamic Programming is Design Technique

→ Related partially to Divide and Conquer and to Greedy Algorithms

→ Problem at hand subdivided into partial problems (subproblems) whose results are combined to solve overall problem

→ In Divide and Conquer, the subproblems are each solved only once, and so for problems to which D&C can be applied, it can be efficient

→ In Greedy algorithm, the subproblem can be solved so as to take a step that is also part of the solution to the overall problem.

→ Dynamic Programming applies where divide-and-conquer approach would cause repeated solution of same subproblem and no locally optimal step is possible that leads to globally optimal solution

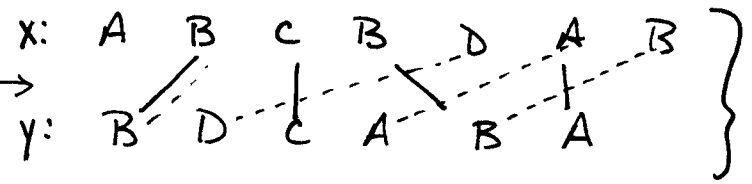
→ General strategy is to make a table and solve all subproblems once-and-only-once. Then build global solution from subproblems.

→ We'll go over these ideas again by working through specific examples

Problem: Longest Common Subsequence (LCS)

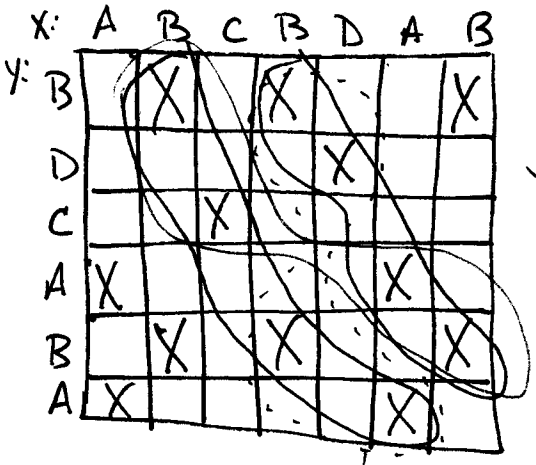
Given two sequences $X[1..m]$ and $y[1..n]$, find a longest subsequence common to both of them. ← (Gaps permitted)

not the =



→ BCBA is contained in each and is LCS
Is there another?
BDAB
BCAB
but not BDBA

no crossing allowed ⇒ implies backtracking



diagonal lines correspond to parallel

diagonal lines correspond to antiparallel

not allowed to backtrack

This helps visualize the problem, but this is not the table of solutions to subproblems that forms dynamic programming approach.

Uses

- Computational Biology. Sequences of DNA, RNA, or protein for difference genes (either within some species or between species). Identifies regions of related structure (sequence = primary struct.) that often corresponds to related function.
- Unix command "diff" compares lines of files

Imagine Brute-Force Algorithm

Examine every subsequence of $X[1..m]$ to see if contained in $Y[1..n]$

Analysis

- number of subsequences of X $\leq 2^m$
- time to check each against Y is $O(n)$ ← Can scan through Y "blanking out" elements that don't match.

W-C running time = $O(n\,2^m)$ → exponential (slow)

← Can you see one obvious speed up here? (If $m > n$ appropriately switch roles of X and Y)

Simplification

1. Focus on finding length of LCS
2. Extend alg. to find LCS itself

Strategy

Subproblem Identification

Consider prefixes of X and Y

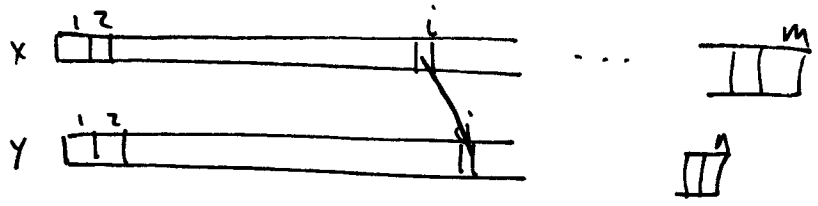
Let $c[i,j] = |LCS(X[1..i], Y[1..j])|$

then $c[m,n] = |LCS(X,Y)|$ ← overall problem (length only)

Theorem:

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

Illustration:



If $X[i]$ and $Y[j]$ are the same, then they could have been "added in last step." If different, then at least one of them is unused. To find which one (or both), look at two subproblems

(or have crossed)

← This is not a proof!

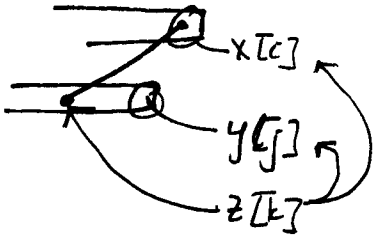
Proof:

(1) For $x[i] = y[c]$ case

Let $z[1..k] = LCS(x[1..i], y[1..j])$, where $c[i,j] = k$

If $z[k] \neq x[c]$ (and $x[c] = y[j]$) then z could be extended by adding $z[k+1] = x[c] = y[j]$, which contradicts $c[i,j] = k$

Note: True even for this case, because here $z[k]$ is still equal to $x[c]$ and $y[j]$, though both may not have been chosen for LCS



So, all we've shown is that $z[1..k-1]$ is CS of $x[1..i-1], y[1..j-1]$

Claim this is LCS. By indirect proof, assume not.

Assume w is longer subsequence ($|w| > k-1$). Then cut and paste $w \| z[k]$ is a CS of x and y with length $> k$, which is a contradiction.

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

(2) If $z[k] \neq x[i]$ then $z[1..k]$ is LCS of $x[1..i-1], y[1..j]$

If there were a longer CS, it would also be a longer CS of $x[1..i], y[1..j]$, violating assumption.

Thus, $z[k]$ is [LCS of $x[1..i-1], y[1..j]$] if $z[k] \neq x[i]$

Likewise, $z[k]$ is [LCS of $x[1..i], y[1..j-1]$] if $z[k] \neq y[j]$

taking maximum effectively tells us which case we are in

Dynamic Programming: Hallmark #1

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

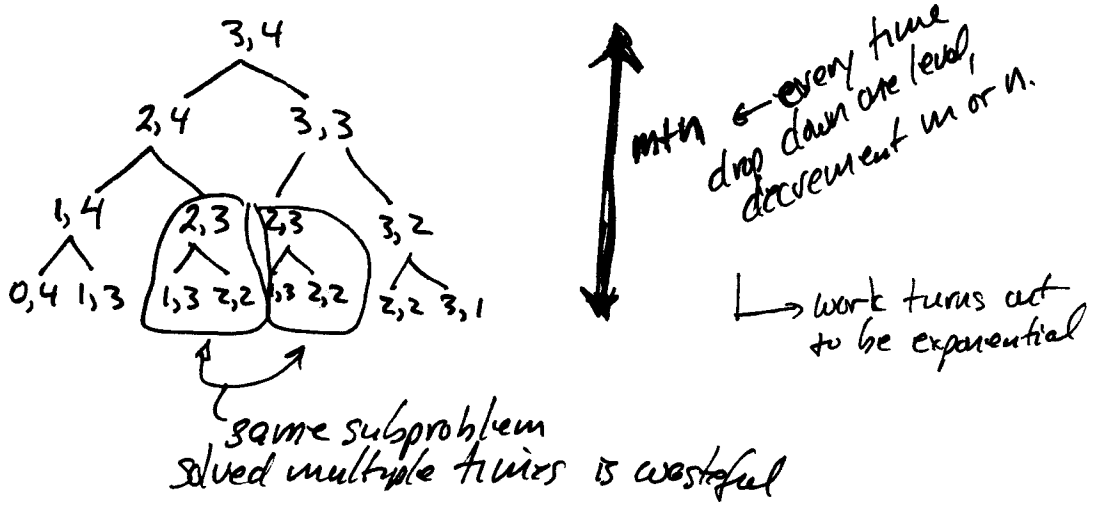
True here, because $c[i,j]$ expressed in terms of $c[i-1,j-1]$, $c[i-1,j]$, and $c[i,j-1]$

If $z = LCS(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

Recursive Algorithm for LCS

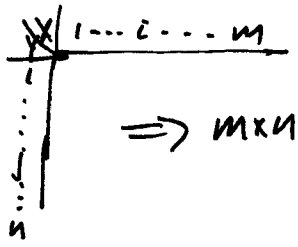
```
LCS (x, y, i, j)
  if   x[i] = y[j]
    then c[i,j] ← LCS(x,y, i-1,j-1) + 1
    else c[i,j] ← max{LCS(x,y,i-1,j), LCS(x,y,i,j-1)}
  return c[i,j]
```

Recursion Tree ($m=3$, $n=4$) for worst-case $x[i] \neq y[j]$



Dynamic Programming: Hallmark #2

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



$\Rightarrow$ $m \times n$ subproblems here

<u>One helpful algorithm</u>: MEMOization

Idea: After computing solution to subproblem, store in table to avoid re-doing work.

```
LCS (x, y, i, j)
  if c[i,j] = NIL
    then if x[i] = y[j]
           :                    same as before
         return c[i,j]
```

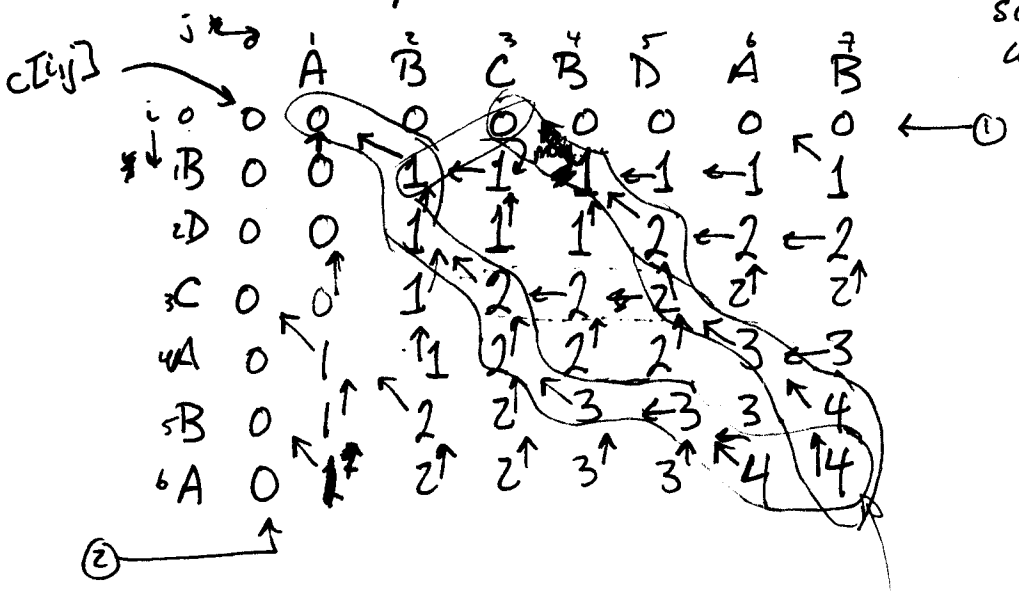
Analysis: Time = $\Theta(mn)$ $\rightarrow$ constant work per entry

Space = $\Theta(mn)$

Dynamic Programming Algorithm

Idea: Compute the table bottom up

Start with small subproblems and work toward larger



So, $C[m,n] = C[6,7] = 4$

and the LCS found is obtained by tracing back the arrows

@ BDAB upward
BCBA leftward
BCAB

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

Time: $\Theta(m \cdot n)$

Some savings & speedups possible

Memoization doesn't tell you the sequence

Biological Sequence Comparison - Proteins

X: } strings with 100's of elements selected from 20 AA's
Y: }

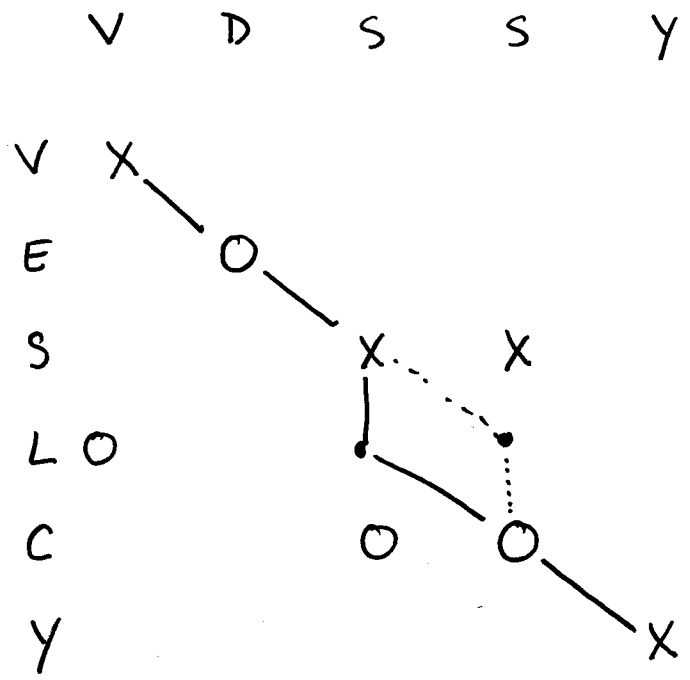
Care about perfect matches and "near" matches at individual locations. ← Chemically similar groups can replace one another (I,V). (E,D) (K,R)

Penalize insertion of gaps

Replace Algorithm:

$$c[i,j] = \max \text{ of } \begin{cases} c[i-1,j-1] + \sigma(x_i, y_j) & \text{(diagonal)} \\ c[i-1,j] - A & \text{(horizontal)} \\ c[i,j-1] - A & \text{(vertical)} \end{cases}$$

where A is (linear) penalty for introducing a gap and $\sigma(x_i, y_j)$ is a substitution matrix element that rewards identical matches highest, similar substitutions next, etc. pos score for pos matches.



V D S – S Y
V E S L C Y
x o x o x

not

V D S S – Y
V E S L C Y
x o x x

LCS: VSY

Summary:

1) Optimal substructure
2) Repeated subproblems

→ but not greedy property