Dynamic Programming

Dynamic Programming

- **Dynamic Programming** is an algorithm design technique based on divide and conquer that seeks to make recursive algorithms faster, by building the solution in reverse and reusing computations.

Rod Cutting

- Given rod of length n with prices p[1..n]
- Each length *i* gives us a certain number of dollars per foot: *p[i]/i*
- Let *L* be the length that maximizes dollars/foot
- So here is an idea: cut off as many pieces as we can of length *L*, then continue in the same manner with what is left

Challenge

- Come up with a list of prices for which this approach fails (with n == 4)
- See CLRS for details of rod-cutting problem

Dynamic Programming

- Main idea:
  - We take a recursive algorithm, which is always defined in terms of smaller subproblems.
  - We compute solutions to those subproblems in advance, so they're precomputed when we need them.
  - This can reduce many algorithms from exponential time to polynomial time. (Or poly. to linear, etc)
  - e.g. We start from the base case, and fill out a table (or array) of solutions. This is called **memoization**. (*from the word 'memo')

Longest Common Subsequence

There are 24 permutations of the four letters ACGT. These are:

AGCT	TGCA	TACG	TAGC
GACT	GTCA	ATCG	ATGC
GCAT	GCTA	ACTG	AGTC
CGAT	CGTA	CATG	GATC
CAGT	CTGA	CTAG	GTAC
ACGT	TCGA	TCAG	TGAC

There are 20!=2432902008176640000 permutations of the 20 amino acids. One such permutation is

Phe-Ser-Tyr-Sys-Leu-Trp-Pro-Hys-Arg-Glu-Ile-Thr-Asn-Met-Lys-Val-Ala-Asp-Gly-Glu

Longest Common Subsequence

- Two strings

  X = ACCG

  Y = CCAGA

  Problem: Find longest common *subsequence*.

  CS of length 1: {A} {C} {G}

  CS of length 2: {CC} {CG} {AG}

  CS of length 3: {CCG}

  CS of length 4: {}

Brute Force
(for Longest Common Subsequence)

X = ACCGGGTTACCGTTTAAAACCCGGGTAACCT
Size: N

Y = CCAGGACCAGGGACCGTTTACCAGCCTTAAACCA
Size: M

Algorithm.

$$\sum_{i=N}^{1} \frac{N!}{i!(N-i)!} = O(2^N)$$

N = X.size() – 1
for **i** = [N .. 1]
find all subsequences of X with length **i**
find all subsequences of Y with length **i**
if (there is a common subsequence) break;

LCS

- How can we use dynamic programming here?
- Consider a longest common subsequence of X and Y; does it contain LCSs of shorter strings?

Longest Common Subsequence

X = ACCGGGTTACCGTTTAAAACCCGGGTAACCT   Size: N
Y = CCAGGACCAGGGACCGTTTACCAGCCTTAAACCA   Size: M

Define: (a smaller problem)   (i < N and j < M)

C[**i**][**j**] : Length of LCS of sequence X[1..**i**] and Y[1..**j**]
: C[i][0] == 0 for all i
: C[0][j] == 0 for all j

Goal

Find C[N][M]

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Longest Common Subsequence

```
for (int i=0; i<X.size(); i++) C[i][0] = 0;
for (int j=0; j<Y.size(); j++) C[0][j] = 0;

for (int i=1; i<X.size(); i++)
   for (int j=1; j<Y.size(); j++) {
      if (X[i] == Y[j]) {
         C[i][j] = C[i-1][j-1] + 1;
       } else if ( C[i][j-1] > C[i-1][j] ) {
         C[i][j] = C[i][j-1];
       } else {
         C[i][j] = C[i-1][j] ;
       }
}
```

X: ACCGGGTTAC
Y: AGGACCA

0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1
0	1	1	1	1	2	2	2
0	1	1	1	1	2	3	3
0	1	2	2	2	2	3	3
0	1	2	3	3	3	3	3
0	1	2	3	3	3	3	3
0	1	2	3	3	3	3	3
0	1	2	3	3	3	3	3
0	1	2	3	4	4	4	4
0	1	2	3	4	5	5	5

Longest Common Subsequence

```
for (int i=0; i<X.size(); i++) C[i][0] = 0;
for (int j=0; j<Y.size(); j++) C[0][j] = 0;

for (int i=1; i<X.size(); i++)
   for (int j=1; j<Y.size(); j++) {
      if (X[i] == Y[j]) {
         C[i][j] = C[i-1][j-1] + 1;
         S[i][j] = 's';  // Same, X[i] or Y[i] is in LCS
      } else if ( C[i][j-1] > C[i-1][j] ) {
         C[i][j] = C[i][j-1];
         S[i][j] = 'j';  // LCS(X[1..i],Y[1..j] = LCS(X[1..i],Y[1..j-1]
      } else {
         C[i][j] = C[i-1][j] ;
         S[i][j] = 'i'; // LCS(X[1..i],Y[1..j] = LCS(X[1..i-1],Y[1..j]
     }
}
```

Print Longest Common Subsequence

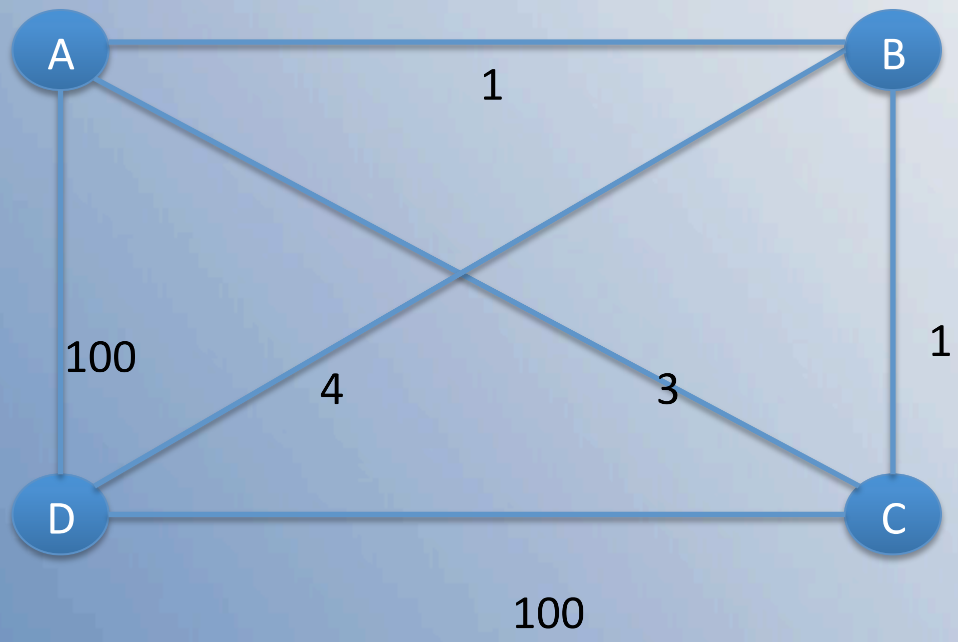
```
void printLCS(S, X, i, j) {
  if (i==0 || j == 0)
      return;
  if ('s' == S[i][j]) {
      printLCS(S, X, i-1, j-1);
      print X[i];
  } else if ('j' == S[i][j])  {
      printLCS(S, X, i, j-1);
  } else {
      printLCS(S, X, i-1, j);
  }
}
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Matrix “S” overlaid on “C”

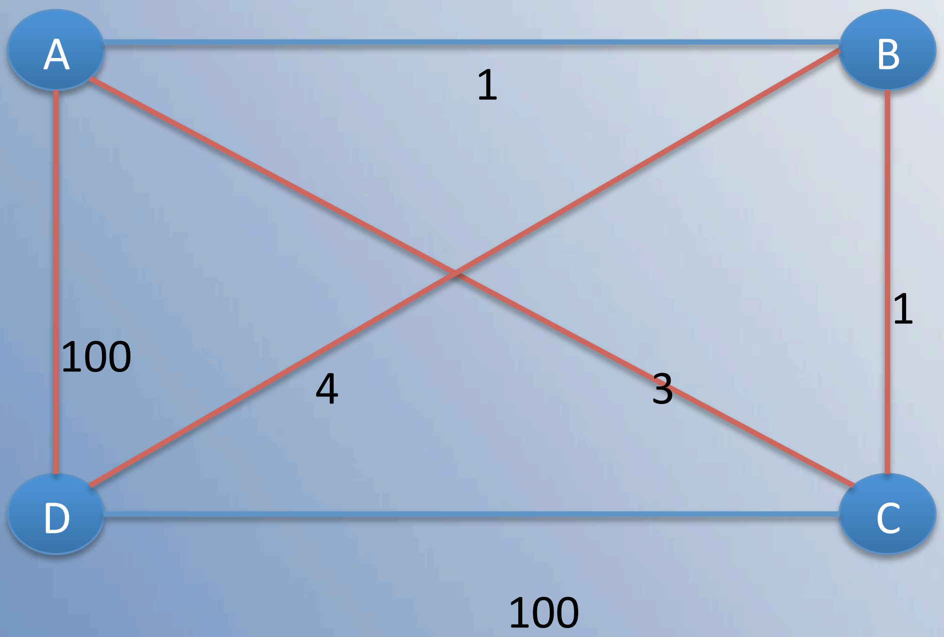
When DP does not apply

- Find shortest loop through this network:



When DP does not apply

- Find shortest loop through this network:



ACBDA – note we do NOT take shortest path from A to C

Dynamic Programming : Review

- Technique builds a "table" from the bottom up, or in some cases, stores the previous calculations in an array (called memoization).

- Enhances recursive algorithms, if:
  - Many overlapping, independent subproblems
  - We can save calls by solving problem in a different order.
  - Algorithm must have **Optimal Substructure**
    - Where optimal solution to a problem requires computing the optimal solution to subproblems.