Dynamic Programming

Dynamic Programming

- Another strategy for designing algorithms is *dynamic programming*
  - A metatechnique, not an algorithm
    (like divide & conquer)
  - The word “programming” is historical and predates computer programming
- Use when problem breaks down into recurring small subproblems

Properties of a problem that can be solved with dynamic programming

- Simple Subproblems
  - We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal Substructure of the problems
  - The solution to the problem must be a composition of subproblem solutions
- Subproblem Overlap
  - Optimal subproblems to unrelated problems can contain subproblems in common

Dynamic Programming Example: Longest Common Subsequence

- *Longest common subsequence* (*LCS*) problem:
  - Given two sequences x[1..m] and y[1..n], find the longest subsequence which occurs in both
  - Ex: x = {A B C B D A B }, y = {B D C A B A}
  - {B C} and {A A} are both subsequences of both
    - *What is the LCS?*
  - Brute-force algorithm: For every subsequence of x, check if it's a subsequence of y
    - *How many subsequences of x are there?*
    - *What will be the running time of the brute-force alg?*

LCS Algorithm

- Brute-force algorithm: 2^m subsequences of x to check against n elements of y: $O(n\, 2^m)$
- We can do better: for now, let's only worry about the problem of finding the *length* of LCS
  - When finished we will see how to backtrack from this solution back to the actual LCS
- Notice LCS problem has optimal substructure
  - Subproblems: LCS of pairs of *prefixes* of x and y

Finding LCS Length

- Define c[*i*,*j*] to be the length of the LCS of x[1..*i*] and y[1..*j*]
  - *What is the length of LCS of x and y?*
- Theorem:

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

- *What is this really saying?*

Longest Common Subsequence (LCS)

- Define X_i, Y_j to be prefixes of X and Y of length i and j; m = |X|, n = |Y|
- We store the length of LCS(X_i, Y_j) in c[i,j]
- Trivial cases: LCS(X_0 , Y_j) and LCS(X_i, Y_0) is empty (so c[0,j] = c[i,0] = 0)
- Recursive formula for c[i,j]:

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

c[m,n] is the final solution

Longest Common Subsequence (LCS)

- After we have filled the array *c[ ]*, we can use this data to find the characters that constitute the Longest Common Subsequence

- Algorithm runs in $O(m*n)$, which is *much* better than the brute-force algorithm: $O(n\ 2^m)$