## Dynamic Programming

12-2-2003

## Opening Discussion

- What do you remember about our discussion of dynamic programming from last class? What are the steps in coming up with a DP solution to a problem?
- Do you have any questions about the assignment?
- Scheme and the largest factorial in Java.

## Optimal Substructure

- In order for a problem to be approachable with dynamic programming, it must have optimal substructure. So the optimal solution to the full problem must be built from optimal solutions to smaller problems.
- In graphs a shortest path has this, a longest path does not.

## Assembly Line Problem

- As simple example of this is the assembly line problem from the book. You have two assembly lines next to one another they do things at different speeds. Normally cars go straight through them, but rush jobs want the fastest path. That might include some transfers, but transfers cost some time.
- At each station the optimal path either comes from the previous station on that line, or the previous one on the other line.

## Assembly Line Recursion/Solution

- Solving this requires only the recursive function and two arrays, one for each assembly line.

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & j = 1 \\ a_{1,j} + \min\left(f_1[j-1], f_2[j-1] + t_{2,j-1}\right) & j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & j = 1 \\ a_{2,j} + \min\left(f_2[j-1], f_1[j-1] + t_{1,j-1}\right) & j \geq 2 \end{cases}$$

## 0/1 Knapsack Problem

- Last time we began discussion this problem. We need to consider if our approach gives us optimal substructure.
- To do that the recursive arguments must be the number of the object being considered and the weight we are currently carrying. This makes things harder for non-integer weights.

$$v(i,w) = \begin{cases} 0 & i < 1 \vee w \leq 0 \\ v(i-1,w) & w < w_i \\ \max\left(v(i-1,w), v(i-1,w-w_i) + v_i\right) & otherwise \end{cases}$$

## Solving 0/1 Knapsack

- To solve this problem we need a 2D array with one dimension that goes up to the number of items and a second dimension that goes up to the maximum weight that we can hold.
- We simply fill this in starting from the bottom left corner and the value that comes out in the top right corner is the final answer.

## Reconstructing the Solution

- The methods we have talked about only tell us a final value, not how to get it. To find that, we simply track back from the answer we got and reconstruct how we got there.
- Let's see how this works in our examples.

## Memoization

- Because we have trained our brains to think in a top-down approach a lot of the time, it can often be helpful to use a top-down style like we would with recursion, but simply save the intermediate answers. For this we have an array like in DP that we pass through the recursive calls. When we need something we try looking in the array before making the recursive call.

## Longest Common Subsequence

- For this problem, we want to find the longest sequence of characters that is found in two strings. The sequence must be found in both in order, but doesn't have to be consecutive.
- For example, "Happy Birthday" and "Have a nice day" share the common subsequence "Ha iday"

## Solution to LCS

- Does this problem have optimal substructure?
- How can we characterize a solution?
- What does the recursive function look like for finding the length of the LCS? To answer this last question let's think of what is it a function of, and what the possibilities are for a given value.

## Minute Essay

- Write a piece of code (or pseudocode) that would solve one of the DP problems we talked about today.
- Assignment #6 is due a week from today. Assignment #7 has been posted as well. I'd like for it to be submitted by the 15th.