Design and Analysis of Algorithms – CSE 101

Dynammic Programming Algorithms : Design and Analysis

When is dynamic programming useful Dynamic programming is useful whenever you have a recursive procedure that is making *repeated* calls on *equivalent* sub-problems. Since these repeated calls themselves call the recursive procedure several times, this can lead to an unnecessary exponential blow-up in total time. In dynamic programming, we identify these repetitions, and *store and reuse* the answers, instead of repeating them from scratch. Normally, we then rewrite the program into an *iterative* algorithm that solves the sub-problems in *bottom-up* order. Dynamic programming can be used on any duplicative recursion, but the most typical applications are to *search and optimization* problems. In this case, we usually use a simple back-tracking procedure to obtain our initial recursive algorithm for the problem.

Basic Design Steps

1. Find a recursive solution for the problem. This is usually a back-tracking algorithm. Don't worry about optimizing the time for this algorithm. The actual time of the DP version has very little connection to the time of the BT algorithm it is based on. Usually, a simple BT will be more likely to have repeated sub-problems than a complicated one. So just keep it simple. Note that there can be problems with multiple BT algorithms, and the one that leads to the DP solution isn't necessarily the best BT algorithm. If you get stymied, "back-track" to another back-tracking approach. Note that the recursive procedure might be solving a more general problem. Generally, you should try to minimize the number of parameters that need to be carried forwards in the procedure.
2. Identify and classify repeated sub-problems. What kind of sub-problems does your algorithm use? Why might there be equivalent sub-problems? What do the problems at level i of the recursion look like? Think of a way to label each sub-problem that might come up. The labels should be tuples of small integers, keeping track of different parts of the sub-problem.
3. Use the labeling scheme to come up with the dimensions of arrays to store answers to sub-problems.
4. Identify what is "top-down" order. What changes in a sub-problem called by a procedure as opposed to the input problem? Then invert this order, to define a "bottom-up" order for sub-problems.
5. Now you have all the ingrediants for your DP algorithm. The outline will go something like:
   (a) Generic DP
   (b) Initialize array $Answer[possiblesub-problemlabel]$ to store the answers for sub-problems.
   (c) Fill in the values of the "base cases" of the recursion. For each (sub-problem where the recursion bottoms out) do: (fill in $Answer[sub-problem]$ according to base case of recursion)
   (d) In bottom-up order, fill in all other sub-problems according to recursion: For (first sub-problems in bottom-up order) to (last sub-problem in bottom up order) do: $Answer[sub-problem] \leftarrow$ Recursive body changing indices as needed and replacing recursive calls on sub-problem' with $Answer[sub-problem']$.
   (e) Return $Answer[mainproblem]$.
6. Correctness follows from the correctness of the recursion, since we do the same work as the recursion, just in different order and without repetitions. Time analysis is based on the new loop structure. Basically, it is Time= # of sub-problems * Time to do main body of recursion. Occasionally, the main body of the recursion has a variable time, and we sum up all the different times rather than using a fixed worst-case for the time of the main body.