Dynamic Programming

Hello, welcome to competitive programming.

Today we are going to discuss a technique that is extremely common called dynamic programming.

Objectives

Your objectives are to be able to describe how dynamic programming is different from greedy algorithms. You will know two different ways to implement dynamic programming: top down and bottom up, and you will see how dynamic programming can solve the Fibonacci sequence and the wedding shopping problem.

When Greediness Fails

You will remember that a greedy algorithm has two properties: first, that sub-problems have optimal solutions that can be combined to solve the main problem. Second, the algorithm has the Greedy Property, that the best choice locally is also the best choice globally.

In Dynamic programming, problems still have optimal subproblems, but now the subproblems can overlap each other. And worse, the greedy property does not hold; a choice that looks optimal now can cause trouble later.

Fibonacci

A Very Simple Example

Fibonacci Numbers. They're everywhere: plants, the golden ratio, and computer science lectures. The mathematical definition of the fibonacci sequence is very elegant, but don't you dare try to compute it that way!

Look at what happens if we try to compute the fifth Fibonacci number: f5 is equal to f4 plus f3, but then f4 is equal to f3 plus f2, and then f3 is equal to f2 plus f1. Computing it out this way has exponential time complexity. But this is completely unnecessary. The problem is we are recomputing solutions to subproblems over and over again.

The Naive Solution

Here's the naive solution. It's a very straight-forward implementation of the mathematical formula. If I run it on my laptop, it takes 52 seconds to compute the 50th Fibonacci number. If I were to compute the 51st, it would take almost twice as long as that.

One nice thing about this kind of computation though is that we can optimize it very easily by keeping a memoization table. We can use an array to check if we have attempted to compute this value before. Think of it as a web cache for your functions.

The Top Down Solution

To make it work you need to add three lines and modify a forth. First, declare an array large enough to hold the values you intend to compute. Second, in your computation, check the memoization array first; return the saved value if you already know it. Third, if you do have to perform the computation, update the memoization array on your way out. Finally, initialize your memoization array so you can tell which values you haven't seen yet.

It’s not a big change in the code, but notice it runs fast enough that it is below the resolution of the time command to say how quickly it ran.

This is called top down because we first attempt to compute the top of the tree, the final solution, and fill in the values of the table as we need them. It is very easy to modify a recursive solution to use Dynamic Programming. Another advantage is that some problems don’t need to compute all the elements of the table; we only compute the elements we actually needed to solve the problem.

The Bottom Up Solution

Another way to use Dynamic Programming is to compute from the bottom up. For Fibonacci, it means you initialize the bottom two elements as 1, and then compute the rest of the table up to the maximum.

A bottom up strategy typically uses a for loop.

Saving Space

The bottom up strategy has another advantage to it: if you only need the last row or two to deliver the answer, you can save space by not storing all the previous rows. This is called the space saving trick, and it could help if you need Dynamic Programming but don’t have a lot of memory.

In this example, i and j represent the values of the last two entries o the memoization array. On each recursive call, we update i and j to be the values of the next entries.

Of course, if you are going to make multiple queries on the same data, or if you want to be able to return the choices that make the optimal result, then you cannot use this method.

Wedding Shopping

The Problem

Let’s look at a more advanced problem. This is UVa 11450, Wedding Shopping. You may want to try this problem as we go along.

The basic idea is that you have some money and need to buy clothes to attend a wedding. There are C categories of clothes, and each category has different models costing different amounts. You want to look as nice as you can given the money you have, so you want to spend as much money as possible.

The thing that makes this difficult is that you must select one garment from each category. Clearly the greedy algorithm won’t work now, because if you spend too much money on a garment from an early category, you might run out before you can get a garment from a later category.

The problem specifies that there can be up to 20 categories, and each category can have up to 20 models. If you try to do an exhaustive search, you will run out of time because there are 20 to the 20 combinations you need to try.

If you have time, go ahead and download that problem and solve it using a top down recursive approach. The tests that you are given will run very quickly, giving you a false sense of security. Once you are ready, resume the video and we’ll go over the recursive enumeration solution.

The Recursive Version

Here is the naive recursive version. We have variables N, M, and C, for the number of test cases, the amount of money, and the number of categories. We have a costs array that stores the price of each of the models of garments.

The algorithm will start with the first category and pick a garment. Then it will recurse to the next category, deducting the price of the garment. It will do this for each of the models and take the one that maximizes the total amount of money spent.

To make this use dynamic programming, we create a memo array as before, but this time it needs to be two dimensional since the shop function takes two arguments.

The Memoized Version

Here is the memoized version. Again, like before, we only had to add a few lines of code. Note the use of a reference variable to make updating the memo table easier.

One thing you might wonder; what if you actually had to return the garments you used?

Returning the Decisions

It turns out this is not that difficult in this case. We just call shop again and see which garment purchase gave the optimal answer and assume that was the right one. This has very similar structure to the original shop function, but we only recurse on one state, the optimal one.

The Bottom Up Version

Here is the code to initialize the tables if we are using a bottom up style. We initialize an array `reachable` to have everything be false. The first loop sets the elements of the `reachable` array to true for each garment that is affordable at the start. We then use the `reachable` bits of the previous row to decide which entries to consider for the next row.

The code is not that tricky, but I do suggest you pause here to read it carefully and ensure you understand what it is doing, because in the next slides I'm going to show you a trick you can do with this code.

(pause)

So here’s the thing; notice that we only access rows `g` and `g-1` as we are computing the reachability array. This means we can use the space saving trick.

The Space Saving Trick

Here’s what we do. We define two new variables `cur` and `prev`, and use those instead of `g` and `g-1`. At each run of the loop, we swap their values. This way we only need two rows for the problem, instead of 20.

That’s it for this example.

Next
Transcript — Edmonds Karp →