Introduction to Dynamic Programming

"DP first; think later" ~ Muthu Chidambaram

Albert Gural

October 14, 2011

1 Introduction

Silver USACO contests will often include an optimization problem with a fairly obvious brute force approach with a runtime complexity around $O(2^N)$. Assuming that $N > 20$, brute force approaches are probably too slow. Many times the correct solution involves Dynamic Programming.

2 What is Dynamic Programming?

Dynamic Programming is the misleading name given to a general method of solving certain optimization problems. Chances are if a problem asks you to optimize something and doesn't involve a graph, you'll want to use Dynamic Programming. As always, there are exceptions (for example, for problems with small upper bounds on N, brute force might actually be the way to go).

Dynamic Programming works on problems that can be represented as a series of sub-states. We can solve the smallest or base state first, then work up from there building up to the solution. Since we only calculate each sub-state once, the runtime of dynamic programming solutions is polynomial.

2.1 Simple Example

A simple and overused example of dynamic programming is calculation of the Fibonacci numbers. Calculating the Fibonacci numbers recursively has an exponential time complexity $O(\varphi^N)$. But if we work from the bottom up calculating F_2, then F_3, etc, it's clear that this is $O(N)$.

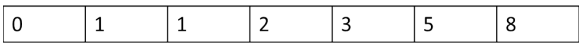

Figure 1: DP Array for Calculation of the First Few Fibonacci Numbers

2.2 Simple Example Extension - Sliding Windows

A lot of memory is wasted in keeping old Fibonacci values that are no longer used. In fact, we really only need to store three numbers at any given time (or perhaps fewer if you're tricky). Keeping only the necessary information is known as ***sliding window***, and you may find it critical for some USACO problems.

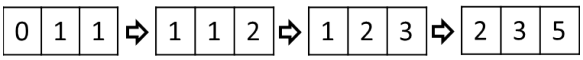

Figure 2: Calculation of Fibonacci Numbers Using DP and Sliding Window

3 Knapsack Algorithm

One very common DP problem you'll see on many Silver Contests is the Knapsack Problem. There are various forms of the Knapsack problem, but the general idea is that you have a "knapsack" with some capacity C. You have to fill it with any number of N types of objects of some weight $W[i]$ and value $V[i]$. Given that the sum of the weights must not exceed C, find the maximum value that can be stored in the knapsack. This specific form, where you have an infinite number of discrete objects, is often known as the Integer Knapsack Problem.

By way of example, consider the following case:

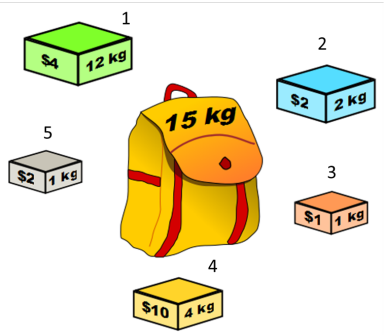

Figure 3: Example Knapsack Problem

3.1 Brute Force

We know that each of the N objects is *in* the knapsack some number of times. We could iterate over each object over the number of times it could be in the knapsack, but this is clearly too slow - exponential in N.

3.2 The *Wrong* Way

You might think that we can just greedily add the objects with the highest value to weight first. But consider the case $C = 10$ and we have two objects: $\{V_1 = 8, W_1 = 6\}$, $\{V_2 = 5, W_2 = 5\}$. We would greedily add object 1, yielding a value of 8. However, a cleverer person would add two object 2s, thus yielding a value of 10.

3.3 Dynamic Programming

Suppose instead of finding the maximum value obtainable with C as the capacity, we found the maximum value for some lower capacity limit. Finding the maximum for $C' = 0$, for example, is trivial. If we now take $C' = 1$, we can just find which objects have a weight of 1 and maximize over their values. In fact, as we increment C' we begin to notice a general pattern. Let's construct an array `dp[i]` that denotes the max value of a knapsack with capacity i. Then with a bit of thought, it's not hard to realize $dp[i] = \max_{\text{item } j} [dp[i - W[j]] + V[j]]$, assuming we initialize `dp[1...C]` to $-\infty$. Take a moment to understand why this works.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	2	4	6	10	12	13	16	20	22	24	26	30	32	34	36

Figure 4: DP Array - Top Row Represents Capacity, Bottom Row Represents Optimum Value

4 General Strategy

Now that we've seen an example problem and its dynamic programming solution, is there a general strategy for solving DP problems? In fact there is.

4.1 Definition of State Variables

In this step, we essentially need to figure out what the substates are and what variables will change for smaller or larger substates. We can then construct some array `dp` where accessing `dp[i][j][k]...` returns the desired answer for a given subproblem of those variables $i, j, k, \ldots$.

In the Knapsack example, our substates were the maximum values of the knapsack for smaller capacities. We could then base the substate on the capacity, creating a one-dimensional `dp` array. For some substate capacity i, `dp[i]` returned the maximum attainable value for that capacity.

4.2 Base Cases

Having figured out the state variables, we need to determine some base cases, usually at points where the state variables are very small.

In the Knapsack example, we might want to initialize `dp[0]` to 0 (since a knapsack of 0 capacity holds 0 value). We might also want to initialize the other elements of the array to $-\infty$ so that when we maximize, we make sure not to choose an unattainable capacity. If we use this method, we have to be careful when choosing the "maximum" that `dp[C]` is actually a positive number. Instead, we might choose the highest element of `dp` that is positive, since this will clearly be the maximum possible value of the knapsack of capacity C.

4.3 Transition Functions

Here's the part that usually requires lots of thought. We need to figure out the relationship between the elements in our array, and we need to be able to calculate them in a bottom-up approach so that our run time is polynomial.

In the Knapsack example, we chose the function $dp[i] = \max_{\text{item } j} [dp[i - W[j]] + V[j]]$. From this, we start at `dp[1]` and continue to `dp[C]`, making sure to traverse in that order (bottom-up). What made us choose this particular function is that we know we want to consider all the possible ways to get `dp[i]`, and we want to maximize its value. So for each object j, we simply check if the value of that object added to a knapsack of capacity $i - W[j]$ is greater than `dp[i]`.

5 A Harder Problem

Having shown a simple one dimensional Dynamic Programming problem, try solving this more complicated problem using the general method shown in section 4.

What are the state variables? What should be the definition of the `dp` array? What are the base cases? How should we initialize the array? What is the transitional function? What should we be careful of?

Another thing you should consider is the time and space complexity of your algorithm. If it's too memory heavy, can you use sliding window?

5.1 Dividing the Gold (Sherry Wu, 2010)

Bessie and Canmuu found a sack of $N (1 \leq N \leq 250)$ gold coins of values $V_i (1 \leq V_i \leq 2000)$ that they wish to divide as evenly as possible, although that is not always possible. What is the smallest difference between the values of the two piles, and how many ways are there to split the piles with this difference? (Note: $\{1, 1, 1, 1\}$ can be split in 6 ways.)

5.2 Analysis

This is a somewhat incomplete analysis of the steps you might take to solve this DP problem.

5.2.1 Definition of State Variables

Before looking for state variables, we need to figure out what we're looking for. We're looking for a *number of ways* to find some sum of coins, presumably such that the sum of the other coins is close to the first sum. If we look at it as a problem of just finding the number of ways to sum some value from some number of coins, we can see that there are two state variables.

The first is the set of coins we'll consider for taking a sum. Essentially this means we'll look at the first i coins out of the total N. The second variable is the possible sum of values of coins. Put together, we're looking for the number of ways to make some value j out of the first i coins. This gives us a two dimensional array `dp[i][j]`.

5.2.2 Base Cases

We can start by initializing the entire `dp` array to 0, since for most points, we'll start by assuming there are no ways to make a sum of value j out of the first i coins. We can do some further initialization by noticing that there is 1 way to make a value of 0 out of any number of coins. Similarly, there are 0 ways to make any value out of the first 0 coins, except for a value of 1.

5.2.3 Transition Functions

What we can do here is iterate along each of the possible values - the "j" axis. For each iteration, we iterate along the inclusion of an additional coin - the "i" axis. As we consider the inclusion of an additional coin, there are two cases. Include or don't include. With this, we can write out our transitional function.

$$dp[i][j] = dp[i-1][j] + dp[i-1][j - V[i]]$$

Again, take a moment to understand why this works.

5.2.4 Final Analysis

For this particular problem, completing the `dp` array isn't enough. First of all, the bounds of the problem make memory usage an issue. We can solve this by using sliding windows (the maximum value of a coin is 2000, so disregard any values prior to the current minus 2000). Another issue is that we're trying to find the number of ways to make a specific value using all the coins. This specific value is half the total value - or as close to half as possible. It shouldn't be too hard to figure out which element to choose at this point - but make sure to adhere to the problem statement. (Hint: don't choose an element that's 0!)

If you think you understand how to solve this problem, write out the `dp` array by hand and circle the element that will serve as the final answer.

6 DP Practice and Problems

The best way to get used to Dynamic Programming is to practice! MIT provides a fairly comprehensive list of problems that can be solved using dynamic programming:

`http://people.csail.mit.edu/bdean/6.046/dp/`

Additionally, a very cool list of practice DP problems will be provided as a supplement to this lecture.