More Dynamic Programming

Alex Chen

December 2, 2011

based on a past lecture by Andre

1 Introduction

Dynamic programming (DP) is an extremely useful technique for USACO (and for many programming-related tasks outside of contest programming). The basic idea is solving a large problem by solving smaller subproblems. A key idea in DP is never calculating the same thing twice, allowing us to turn many exponential time naive algorithms into polynomial time DP algorithms.

This DP lecture is not meant to be an complete introduction but rather a source of additional knowledge and ideas for those who already have some intuitive understanding of dynamic programming. This is also not to be very advanced or super comprehensive.

2 Strategy

Any DP algorithm is based on three characteristics, and once these three characteristics are figured out, then coding a DP algorithm is significantly easier. To illustrate, let's consider the coin change problem: given a set of N coins, each with some integer value, how many ways are there to use the coins to create M total money without any limitations on the number of times each coin can be used?

1. **State variables**: each value calculated in a DP algorithm can be linked to certain *state variables* that describe the "state." For example, we can solve the coin change problem by calculating $dp[i][j]$ if $dp[i][j]$ represents the total number of ways to produce j total money using the first i coins. Then, the state variables are i, the number of coins to use, and j, the target amount of money.

2. **Base cases**: before we start the DP algorithm, some set of values must already be initialized or else there is no basis from which to begin. In the coin change problem, we can set $dp[*][0]$ to 1 for all values of $*$ because no matter how many coins we use, there is only one way to produce a total of 0: by not using any coins at all. The other base case is $dp[0][*] = 0$ for all values of $* > 0$, because using no coins, it is not possible to produce any amount.

3. **Transtions**: there must be a formula to generate the values for new states based on values of old states. This is the heart of the DP and the hardest piece to figure out. For the coin change problem, we note that whenever we have a new coin, we have two choices: use it or don't. Thus, $dp[i][j] = dp[i-1][j] + dp[i][j-val[i]]$, where $val[i]$ denotes the monetary value of coin i. The first element in the sum denotes how many ways there were originally to produce j money, and the second element denotes the number of ways to create j money using previous states that also used the i^{th} coin.

The term *memoization* refers to storing values from previous subproblems for quick acess in future subproblems, especially useful when trying to calculate transitions. When the nature of the state variables allows for each storage within an array, then a simple table of values can serve as the means for memoization. The coin change problem is a good example of where array-based memoization is useful. In the case where using an array would result in a very sparse, inefficient, and wasteful (in terms of memory) table, it might be more useful to use a map of some sort. For example, the coin change problem can also be solved by storing

the DP table in $map < pair < int, int >, int >$, where the key, $pair < int, int >$ holds i and j and the value is the dp value itself.

2.1 Two Approaches

top-down In the top-down approach, we first try to calculate the solution. Upon realizing that some of the transitions utilize results of subproblems that have not yet been solved, we defer the calculation of the solution and recursively solve the subproblems first. The key difference between this and naive solutions is that the solutions to the subproblems are stored for later use.

bottom-up In the bottom-up approach, we start by calculating the solutions to subproblems and slowly build up to the solution. This is better in terms of stack space (recursion is not necessary anymore) but might be harder to code because you would have to figure out what order to solve the subproblems in.

2.2 Sliding Window

The sliding window trick is a DP technique that is used to save memory (and nothing else, really). The key idea behind the sliding window trick, which only works for certain DP problems, is that sometimes, once we have calculated a state, some of the previous states are no longer useful. In that case, why continue storing those states? With the sliding window trick, we simply overwrite it in the memory.

For example, consider the coin change problem again. $dp[i]$ is an array for row i, denoting the number of ways to make any amount using the first i coins. When calculating $dp[i]$, we sometimes refer to values from $dp[i-1]$, the number of ways to make values using the previous coins. However, we never look at $dp[i-2]$ again. Thus, there is no need to keep storing them and wasting memory. Thus, what the sliding window trick allows us to do is place the values of $dp[i]$ over the values of $dp[i-2]$ instead of creating a new row of the array for $dp[i]$. This whole trick allows us to perform the entire DP using only two rows.

Implementation-wise, it is helpful to consider the parities of numbers. For all i that are odd, we can store then in $dp[1]$, referring to values from $dp[0]$ in the transition. For all i that are even, we can store them in $dp[0]$.

Always remember to reset the sliding window when you are finished with a row—I, alongside many others, have made this mistake far too many times.

3 DP Types

There are many types of DP problems. A lot of them are straightforward but most of them require a good deal of creativity. The list below does not summarize every type, but gives a brief overview of some examples of DP. There is no "standard" dynamic programming algorithm or problem, for it is merely a technique that is extremely useful.

3.1 Knapsack

Knapsack problems come in several types, but all have a similar idea: you have a lot of items, each of which has a weight and a value. You have a knapsack that can only carry items totaling at most a certain total weight, and you wish to choose items to maximize the total value without breaking the knapsack.

Variations:

- *Integer weights versus decimal weights*: integer weight knapsack problems can be solved with DP much more easily, because it is not possible to have decimal array indices.
- *Single use versus unlimited usage*: sometimes, you can only use each item once and other times, you can use each item as many times as you want. There might also be multiple usage problems, where

there is some limit to the number of times you can use each item. However, multiple usage is just a case of single usage, because one can represent the multiple items as multiple copies the same item, of which each copy can only be used once. The difference between single use and unlimited use often shows up in code as deciding whether to iterate through values in the forward direction or the reverse direction.

- *Whole versus fractional*: most knapsack problems are whole knapsack problems, which means that you can only take whole number quantities of an item. In the fractional knapsack problem, you can take parts of each item. Fractional knapsack problems are easy: you can just sort the items based on their value to weight ratio and choose the items with the highest ratios greedily.
- *single knapsack versus multiple knapsacks*: if there are multiple knapsacks to put items in, the problem becomes significantly more challenging. Recursion sometimes works, but it depends on the problem.

Example: Farmer John wants to buy cheese, but he only has M money to use. There are N pieces of cheese, each of which has some cost and some value. Maximize the total value of the cheese that Farmer John can buy. Because Bessie likes variety, Farmer John may not buy more than one of each type of cheese.

Worked Solution: The state variables are i, the number of cheese types considered, and j, the total money spend. Let $dp[i][j]$ be the maximum value achievable with the first i pieces of cheese and at most j money spent. The base case is $dp[*][0] = 0$, because without spending money, you cannot have any value for cheese. The transition is: $dp[i][j] = \max(dp[i-1][j-weight[i]]+val[i], dp[i-1][j])$ (you buy the cheese or you don't).

Variation: What if you could buy as much of each cheese as you wanted? Then the transition would be $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-weight[i]] + val[i], dp[i][j-weight[i]] + val[i]$ (you don't buy the cheese, you buy the cheese, and you buy the cheese not for the first time). There are other possible transitions and the one shown here is not the most elegant.

Something to think about: With the above transition, two rows of a DP array are necessary (using the sliding window trick). There is a way to solve this using only one row. How? How does this method change based on whether Bessie cares about variety or not?

3.2 Interval-Based

Interval-based DP problems have state variables that represent an interval. Let's illustrate this with a sample problem (from the IOI in the 1990s):

Proble: There is a list of N numbers. Bessie and Farmer John take turns choosing numbers, but they can only take numbers from the ends of the list. Assuming both players play optimally, what is the maximum score that Bessie can achieve? The score is the total of all the selected numbers and Bessie always goes first.

Worked Solution: The state variables are i and j such that $dp[i][j]$ is the maximum score achievable on the subset of the board ranging from indices i to j. The base case is the trivial case: $dp[i][i] = val[i]$, because there is no choice but to take the only element. The transitions are fairly simple. The player who goes first can either take the leftmost item or the rightmost item. Precomputed sums are useful for this problem, and you will see why if you try and work out the transitions equation.

3.3 Tree-Based

Given a tree, you can solve a problem at the root sometimes by building solutions based on the children of nodes. That is, to calculate the solution based on the root, we must first calculate the solutions for the children, and their children, and so on.

Problem: You are given a rooted tree, where each node contains some value. Bessie starts at the root

and travels to some child, summing all the values on the way. Find the maximum possible sum, assuming Bessie chooses the best path.

Worked Solution: The state variable is the current node. This problem can be approached bottom-up or top-down, because it does not matter whether we think of Bessie going down from the root or Bessie going up from the leaves. This problem is extremely similiar to the commonly seen number triangle problem, where we want the maximum sum achievable in a path from the top of the triangle to the bottom of the triangle.

3.4 Row-by-Row

Row-by-row DP algorithms are just like what the name suggests: the state variable is the current row of some array. How does one store an entire row as a state variable? Usually, row-by-row DP problems have elements in rows that are binary: they can only be 0 or 1. Then, each row can be presented as a bitstring. Alternatively, in more complex problems, a map would have to be used where a vector containing the values of the rows are the keys.

Problem: (adapted from USACO) You are given a 400×9 grid. Some of the cells are bad. Farmer John wants to put cows in the grid such that no two cows are in cells that share an edge or a corner. How many ways are there to do it?

Worked Solution: They key observation is that $2^9 = 512$ is pretty small. Thus, we can use DP on this row by row. Let $dp[i][j]$ be the number of ways to place cows up to row i such that row i contains cows wherever there is a 1 in the bitstring representation of j. Then, for the transition, we have $dp[i][j] += dp[i-1][k]$ for all k that can be a previous row if j is a current row.

3.5 Large State

Sometimes the DP states are so large or complicated that it is not possible to store values in an array. This is where maps and recursion come in handy. Hashing is also useful, if maps are not convenient or are unaccessible.

3.6 Power of Two

Sometimes, we can build large solutions based on smaller solutions from powers of 2. This often allows us to find solutions that are $O(N \log N)$ or something of that nature. Building things from powers of 2 is a very common technique and not always associated with dynamic programming. Consider the problem of finding $a^b \pmod k$, where b is some large number, perhaps around 10^18. Instead of multiplying a by itself b times, which will surely exceed the time limit, we can compute $dp[i]$, where $dp[i] = a^{2^i}$. Note that $dp[i] = dp[i-1]^2$. The transition is very simple. Then, we can multiple some combination of these values together to find a^b. This is called **fast exponentiation**.

4 Problems

See the SCT Grader for some DP problems!