Even More Dynamic Programming

Ryan Jian

October 12, 2012

Stolen from basically every one of the past... 7 SCT DP lectures..

1 Introduction

Dynamic programming (DP) is a powerful problem solving technique that efficiently solves problems with overlapping subproblems and optimal substructure. The idea is that the optimal solution to a problem can be found using the optimal solutions to smaller subproblems. By storing and reusing these subproblems, the runtime of an algorithm can be lowered drastically, usually from exponential to polynomial time.

2 The Method

In order to solve a DP problem you have to determine the following:

1. Whether it's DP - This is not as easy as it sounds. You can sometimes tell based on the bounds of the problem or if you figure out a naive solution and find that it has exponential running time.
2. State Variable(s) - The variables that identify a subproblem. There is no one precise method to do this step, usually you just have to guess and see if you can complete the next two steps. Doing practice problems helps a lot with this as you start to recognize patterns in the states that different problems require. Keep in mind that the more variables you use the longer your algorithm will take.
3. Base Case(s) - Because DP solves recursive problems, naturally there has to be a base case. The base case can vary depending on your choice of state variables, however it is very easy to figure out once you've done so.
4. Recurrence Relation / Transitions - The relationship between the optimal solution of a problem and the optimal solution of its subproblems. This step can be pretty difficult, and once again, practice makes perfect. If you find a working transition but it doesn't run within the bounds of the problem, this means usually one of two things. You could have picked a state with too many variables and a simpler state exists and thus a faster transition. For harder problems, it could also mean you need to use a data structure to further lower the time complexity, with common examples being prefix sums and heaps.

   Once you've gone through these steps, you're ready to code. DP solutions are relatively easy to implement once you've solved the actual problem, although some harder problems can be much trickier.

3 Examples

3.1 Fibonacci

The most common example used to introduce DP is calculating the Fibonacci numbers, which are defined by the recurrence relation: $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0$ and $F_1 = 1$ In code, this is:

```
def fib(n):
    if n == 0 || n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Notice that every call to fib calls itself twice, so this naive solution takes exponential time. We also notice that the subproblems are overlapping, for example if we call fib(10)

```
fib(10)
= fib(9) + fib(8)
= (fib(8) + fib(7)) + (fib(7) + fib(6))
= ((fib(7 + fib(6)) + (fib(6) + fib(5))) + ((fib(6) + fib(5)) + (fib(5) + fib(4)))
...
```

Already `fib(5)` is being called and recalculated 3 different times. The idea behind DP is that we would calculate `fib(5)` once and store it in memory so we wouldn't have to spend extra time recalculating it.

Since the problem already gave us the state variables, the base case, and the transitions, let's go ahead an implement a DP algorithm:

```
def fib(n):
    init f[n]
    f[0] = 0
    f[1] = 1
    for i = 2 to n:
        f[i] = f[i - 1] + f[i - 2]
    return f[n]
```

3.2 Integer Knapsack Problem

You're given N objects where the ith object has weight $W[i]$ and value $V[i]$. You want to choose some combination of objects that maximizes the total value such that the total weight does not exceed C.

Clearly we need to involve the weight in the state somehow. Additionally, if we use some item, we need to make sure that we never use it again. With that in mind, we define the state to be $\text{dp}[n][w]$, meaning the maximum value attainable using a subset of the first n cows with total weight w. The n parameter allows us to control which cows are considered. Then $\text{dp}[n][w] = \max(\text{dp}[n-1][w], V[n] + \text{dp}[n-1][w - W[n]])$ (we can either include the nth cow or leave it out). You can actually get rid of the n parameter entirely using the sliding window trick (which we are about to discuss) with a little bit of trickiness.

4 Sliding Window

A trick we can perform to save some memory is to observe that the solutions to some subproblems are never used again for the calculation of other subproblems, meaning they can be overwritten, thus conserving memory.

For the calculation of Fibonacci numbers, notice how the nth Fibonacci number depends only on the previous two. Because of this, we can just continually store and overwrite the value of the last two numbers to reduce the space from $O(N)$ to $O(1)$

```
def fib(n):
    if n < 2:
        return n
    f0 = 0, f1 = 1, f2 = 0
    for i = 2 to n:
        f2 = f0 + f1
        f0 = f1
        f1 = f2
    return f2
```

While this can be used the Fibonacci problem and others, it doesn't always apply.

5 Implementation Strategies

DP can be implemented two different ways, top-down or bottom-up. So far we've been using bottom-up, which is the more common of the two, epsecially for USACO where a majority of the analyses are implemented as such. However both approaches have their strengths and weaknesses.

For top-down (also called memoization), what we would define some sort of table to store or the optimal solutions to different subproblems. Before calculating a subproblem, we would check if a solution for it existed in the table. If so, then we would just return that, otherwise we would recurse. After calculating each subproblem the result would be stored in the table to save recalculation. This has the advantage that for certain problems only the subproblems that actually contribute to our final answer will be used, whereas in bottom-up all subproblems are calculated.

In bottom-up we typically get rid of the recursion, and calculate the subproblems starting from the base case and slowly go up using the recurrence relation. This can be a bit harder because when the order in which we iterate through the different subproblems matters. However the advantage here is that there is no overhead from recursion.

6 Problems

Doing practice problems is important for any topic, but especially DP, since it is a technique that can be used to solve so many different problems. You should begin with the traditional DP problems most of which can be found in a list compiled by Brian Dean at `http://people.csail.mit.edu/bdean/6.046/dp/` Some more untraditional problems are included below in approximate order of increasing difficulty.

As with all problems, you should NEVER read the analysis until AFTER you solve the problem.

1. (Traditional, Subset Sum Problem) Given a set of integers, find a non empty subset of that sums to zero (Hint: Knapsack).

2. (SPOJ, SQRBR) Find the number of properly balanced bracket expressions of length $2N$ ($1 \leq N \leq 19$) with K ($1 \leq K \leq N$) opening brackets at any position from 0 to $2N$. An example of properly balanced bracket expression would be [][], [[]], [[[]][]], but not][or [[[].

3. (Codeforces Croc Champ 2012 - Round 2, Word Cut) A split operation is defined as transforming a word $w = xy$ into a word $u = yx$. For example "wordcut" can become "cutword". Given two words, *start* and *end*, both of length N ($2 \leq \mathrm{N} \leq 10^3$), count how many ways *start* can be transformed into *end* using K ($0 \leq \mathrm{K} \leq 10^5$) split operations.

4. (USACO DEC11, umbrella) Today is a rainy day! FJ's N ($1 \leq N \leq 5{,}000$) cows are not particularly fond of getting wet. The cows are standing in roofless stalls arranged on a number line. The stalls span X-coordinates from 1 to M ($1 \leq M \leq 100{,}000$). Cow i stands at a stall on coordinate X_i ($1 \leq X_i \leq M$). No two cows share stalls.

   To protect the cows from the rain, FJ wants to buy them umbrellas. An umbrella that spans coordinates X_i to X_j ($X_i \leq X_j$) has a width of $X_j - X_i + 1$. It costs C_W ($1 \leq C_W \leq 1{,}000{,}000$) to buy an umbrella of width W. Larger umbrellas do not necessarily cost more than smaller umbrellas.

   Help FJ find the minimum cost it takes to purchase a set of umbrellas that will protect every cow from the rain. Note that the set of umbrellas in an optimal solution might overlap to some extent.

5. (2007 Canadian Computing Competition Stage 1, Bowling for Numbers) At the Canadian Carnival Competition (CCC), a popular game is Bowling for Numbers. A large number of bowling pins are lined up in a row. Each bowling pin has a number printed on it, which is the score obtained from knocking over that pin. The player is given a number of bowling balls; each bowling ball is wide enough to knock over a few consecutive and adjacent pins.

   For example, one possible sequence of pins is: 2 8 5 1 9 6 9 3 2

If Alice was given two balls, each able to knock over three adjacent pins, the maximum score Alice could achieve would be 39, the sum of two throws: $2 + 8 + 5 = 15$, and $9 + 6 + 9 = 24$.

Bob has a strategy where he picks the shot that gives him the most score, then repeatedly picks the shot that gives him the most score from the remaining pins. This strategy doesn't always yield the maximum score, but it is close. On the test data, such a strategy would get a score of 20%.

6. (USACO FEB12, cowids) Being a secret computer geek, Farmer John labels all of his cows with binary numbers. However, he is a bit superstitious, and only labels cows with binary numbers that have exactly K "1" bits ($1 \leq K \leq 10$). The leading bit of each label is always a "1" bit, of course. FJ assigns labels in increasing numeric order, starting from the smallest possible valid label – a K-bit number consisting of all "1" bits. Unfortunately, he loses track of his labeling and needs your help: please determine the Nth label he should assign ($1 \leq N \leq 10^7$).

7. (USACO NOV09, xoinc) FJ's cows like to play coin games so FJ has invented with a new two-player coin game called Xoinc for them.

   Initially a stack of N ($5 \leq N \leq$ 2,000) coins sits on the ground; coin i from the top has integer value C ($1 \leq C_i \leq$ 100,000).

   The first player starts the game by taking the top one or two coins (C_1 and maybe C_2) from the stack. If the first player takes just the top coin, the second player may take the following one or two coins in the next turn. If the first player takes two coins then the second player may take the top one, two, three or four coins from the stack. In each turn, the current player must take at least one coin and at most two times the amount of coins last taken by the opposing player. The game is over when there are no more coins to take.

   Assuming the second player plays optimally to maximize his own winnings, what is the highest total value that the first player can have when the game is over?

8. (USACO NOV05, ants) Bessie was poking around the ant hill one day watching the ants march to and fro while gathering food.She realized that many of the ants were siblings, indistinguishable from one another. She also realized the sometimes only one ant would go for food, sometimes a few, and sometimes all of them. This made for a large number of different sets of ants! Being a bit mathematical, Bessie started wondering. Bessie noted that the hive has T ($1 \leq$ T $\leq$ 1,000) families of ants which she labeled 1...T A ants altogether). Each family had some number N_i($1 \leq N_i \leq$ 100) of ants. How many groups of sizes $S, S+1, ..., B$ ($1 \leq S \leq B \leq A$) can be formed (mod 1,000,000)?