Introduction to Dynamic Programming

Ryan Jian

November 8, 2013

"DP first; think later" - Mutty

1 Introduction

Dynamic programming (DP) is a technique to significantly reduce the runtime of algorithms solving certain types of problems, usually from exponential to polynomial time. More specifically, DP can be used if a problem exhibits:

1. *Optimal substructure*: The optimal solution to the problem can be computed from optimal solutions to its subproblems (typically described using **recursion**).
2. *Overlapping subproblems*: The optimal solutions to subproblems are reused over and over again.

The central idea of DP is that if we have overlapping subproblems, we can store the solutions to the subproblems to avoid recomputation.

2 Examples

2.1 Fibonacci

The simplest and also most overused example of DP is the calculation of the nth Fibonacci number 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 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Notice that every call to `fib()` calls itself twice. Thus, the naive solution of straight up implementing the recurrence relation takes exponential time. Notice that the subproblems are overlapping, for example for $n = 10$

```
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 recomputed 3 different times. To speed up our naive solution, we can employ DP by computing `fib()` for each value up to `n=10` and storing it for future use. One way we can implement this is to slightly modify our original recursive function to check a table to see if we've already computed that specific Fibonacci number. If we have, we use it, otherwise we compute and then store it. This is the *top-down* approach(also called *memoization*) because start from the overall problem and recurse down until we reach the smallest subproblems (the base cases).

```
initialize f[0..n] to all -1
def fib(n):
    if n == 0 or n == 1:
        return n
    if f[n] != -1:
        return f[n]
    else:
        f[n] = fib(n - 1) + fib(n - 2)
        return f[n]
output fib(n)
```

Another way is to take the *bottom-up* approach and start from the smallest subproblems and combine them into larger and larger subproblems. Usually this is the preferred way.

```
initialize f[n]
f[0] = 0
f[1] = 1
for i = 2 to n:
    f[i] = f[i - 1] + f[i - 2]
output f[n]
```

Regardless of our implementation strategy, we end up with a linear time and linear space algorithm, significantly better than our original exponential algorithm. In fact, we can get it down to constant space by using the bottom-up implementation and noticing that as we loop each stored value is only used to compute the next two Fibonacci numbers. As a result, we can discard everything but the two previous Fibonacci numbers.

```
f1 = 0, f2 = 1, f3 = 1
for i = 2 to n:
    f3 = f1 + f1
    f1 = f2
    f2 = f3
output f3
```

In general, this idea of discarding values in the table when they are no longer needed is known as the *sliding window trick*. Note that this cannot be used for every DP algorithm.

2.2 Integer Knapsack Problem

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

For this problem, and for almost all DP problems, the recursion isn't handed to us on a silver platter like it was with the Fibonacci numbers. The truth is that DP is easy to apply once we have an overlapping recursive solution to the problem. What is hard is coming up with a such a solution in the first place.

For this problem, we need to involve both the weight and the value in our recursion somehow. We can define $dp[w]$ to be the the maximum value we can obtain with a total of weight of w. Obviously $dp[0] = 0$. With a bit of thought, we realize that for the ith object, $dp[w] = \max(dp[w], dp[w - W[i]] + V[i])$ for every w from $W[i]$ to C. In psuedocode we have:

```
initialize dp[] to -infinity
dp[0] = 0
for i = 1 to N
    for w = C to 0
        if w >= W[i]:
```

```
            dp[w] = max(dp[w], dp[w - W[i]] + V[i])
output dp[C]
```

Notice that we looped backwards from C to 0. Looping forwards has problems. For example, for two instances of w, w_1 and w_2, where $w_1 < w_2$, if the jth weight equals $w_2 - w_1$ we could end up changing $dp[w_1]$. However, later on when we reach $dp[w_2]$, we change it based on the new value of $dp[w_1]$, not the value it had before adding the jth object, effectively using the object more than once.

Exercise: There is another way to formulate an overlapping recursive solution by defining $dp[v]$ to be the minimum weight that will give us a total value of v. Find the base case, find the correct recurrence, and write the entire DP algorithm in code.

3 Top-down vs Bottom-up

As we saw earlier, there are two approaches to implementing DP. Each turns out to have certain advantages and disadvantages. For top-down, once we have the naive recursive solution, it takes only a few modifications to add the table. In addition, when we compute the solution, we only end up computing the subproblems that the overall problem is dependent on. The downside is that recursion has a slight overhead compared to loops.

Bottom-up on the other hand requires us to convert our recursion into a series of loops. We now have to deal with the order in which we solve the subproblems. Furthermore we are forced to compute the solution for all possible subproblems, not just the ones that are used for a particular instance of the problem. However, the upside is that bottom-up has opportunities for optimizations that have no equivalent for top-down (one example is the aforementioned sliding window trick). As a result bottom-up is usually the preferred implementation, especially in Gold where certain runtime optimizations are needed for an algorithm to run in time.

4 General Strategy

A general strategy for solving DP problems is to find the following three things in the given order:

1. State Variables - The variables that identify a subproblem (the arguments to the recursive function), There is no one precise method to complete step, usually we just have to guess and see if we can complete the next two steps. Doing practice problems helps a lot with this as we start to develop an intuition for this step. **As a rule of thumb, the more state variables the slower the algorithm** (e.g. if we have state $dp[i][j][k]$ where i ranges from 0 to A, j from 0 to B, and k from 0 to C, then going bottom-up means the algorithm must take at least $O(ABC)$ time).
2. Base Case(s) - Because DP solves recursive problems, naturally there has to be a base case that varies depending on our choice of state variables. This is easy.
3. Recurrence Relation - The relationship between the optimal solution of a problem and the optimal solutions to its subproblems. This must result in overlapping subproblems otherwise we cannot use DP. Even if an overlapping recurrence is slower than a non-overlapping recurrence, DP will make the former much faster than the latter.

Once we have gone through these steps we are ready to code. Most of the time DP solutions are relatively short and easy to implement once you've solved the actual problem (not always true for more difficult problems).

5 Problems

If DP hasn't "clicked" for you yet (it certainly didn't the first time I learned it) then trying more problems will eventually do the trick.

You should begin with the traditional DP problems most of which can be found in any introductory algorithms textbook, or at the online list compiled by Brian Dean (`http://people.csail.mit.edu/bdean/6.046/dp/`). A couple of nontraditional problems are provided below:

1. (Traditional, Number Triangle) Given a triangle of numbers we start at the top and can move to adjacent numbers in the row below. Find the path from top to bottom with the greatest sum. For example, in the triangle. For example:
   1
   2 3
   1 5 9
   9 1 1 1
   the best path is 1-3-9-1, with sum 14.

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

3. (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 [{{}.

4. (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 $N(2 \leq N \leq 10^3)$, count how many ways start can be transformed into end using $K(0 \leq K \leq 10^5)$ split operations.

5. (USACO NOV09, xoinc) 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. 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?

6. (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)?

7. (USACO FEB12, cowids) FJ labels all of his 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 ($1 \leq N \leq 10^7$)