Yet Another Dynamic Programming Lecture

Jason Lam

February 12, 2016

"Unfortunately, programmer, no one can be told what DP is.
You have to try it for yourself."
– Stolen from a old SCT lecture

1 Introduction

Dynamic Programming, or DP for short, is a useful technique that is used to significantly reduce the runtime of algorithms in certain problems. The basic idea behind DP is that when faced with a problem that involves many overlapping subproblems, we can store the partial solutions to the problems to avoid recomputing the subproblems. This lecture heavily borrows from Jeff Chen, Ryan Jian, and Rajat Khanna's lectures on DP.

2 Examples

2.1 Fibonacci

The simplest way to see DP in action is to apply it in the calculation of the nth Fibonacci number, defined as $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$. The naive solution is as follows:

```
int fib(int N) {
    if (N == 0 || N == 1)
        return N;
    return (fib(N-1) + fib(N-2))
}
```

Unfortunately, its runtime of $\mathbf{O(2^n)}$ makes this solution too inefficent. The function calls itself twice, and those functions call themselves twice, etc. We can note that not all of these function calls are necessary, so we use DP. Rewriting the function to store previously generated values, we get this:

```
initialize F[]
int fib(int N) {
    F[1] = 1;
    F[2] = 1;
    for (int i = 3; i <= N; i++)
        F[i] = F[i-1] + F[i-2];
    return F[N];
}
```

By using `F[]` as a cache array, we are able to get a runtime of $\mathbf{O(n)}$, which is a huge improvement. We can optimize this solution even further by using the *sliding window trick*, where we take advantage of the fact that we only care about the last two elements in the cache and save memory by deleting all the elements that came before, but that doesn't fall under the scope of DP.

2.2 Integer Knapsack

Now with that trivial problem out of the way, we can move on to a more complex application of DP. Given N types of objects, with the ith object having value V`[i]` and weight W`[i]`, what is the maximum value we can pick up without exceeding a weight C?

The recursion in this problem is much less obvious than the last one, but once it is identified, coding the solution and applying DP becomes pretty simple. For this particular problem, we have to account for both the weight and the value of each object. We can define `F[c]` to be the maximum value we can store within a weight of `c`, and set `F[0]` to be `0`. By looking at the idea of adding an object of weight W from the perspective of decreasing `c` by W, we can see that a formula for `F[c]` would be `max(F[c], F[c-W[i]] + V[i])`, where we loop through all the is from 1 to N. The solution would look like this:

```
initialize F[] to -infinity
F[0] = 0;
int knapsack(int C) {
    if (F[C] > -infinity)
        return F[C];
    for (int i = 1; i <= N; i++) {
        if (C - W[i] > 0) {
            F[C] = max(F[C], V[i] + knapsack(C - W[i]));
        }
    }
    return F[C];
}
```

3 General Strategy

1. **Identify state variables.** The state variables identify a subproblem. In the Fibonacci problem, it would just be n, while in the integer knapsack problem it would be the weight. Having more state variables slows the algorithm down.

2. **Find base case(s).** Just like in any other recursive solution, the base cases have to be set. This should be simple.

3. **Figure out recursive relations.** Somehow, the subproblems have to relate back to the "big" problem. There must be overlapping subproblems for DP to be effective.

4 Problems

For additional practice, here are the first five problems on Brian Dean's list of traditional DP problems (`http://people.cs.clemson.edu/~bcdean/dp_practice/`). The solutions are all available on the website as Flash animations.

1. **Maximum Value Contiguous Subsequence.** Given a sequence of n real numbers A(1) ... A(n), determine a contiguous subsequence A(i) ... A(j) for which the sum of elements in the subsequence is maximized.

2. **Making Change.** You are given n types of coin denominations of values v(1) ¡ v(2) ¡ ... ¡ v(n) (all integers). Assume v(1) = 1, so you can always make change for any amount of money C. Give an algorithm which makes change for an amount of money C with as few coins as possible.

3. **Longest Increasing Subsequence.** Given a sequence of n real numbers A(1) ... A(n), determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence form a strictly increasing sequence.

4. **Box Stacking.** You are given a set of n types of rectangular 3-D boxes, where the i^{th} box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

5. **Building Bridges.** Consider a 2-D map with a horizontal river passing through its center. There are n cities on the southern bank with x-coordinates a(1) ... a(n) and n cities on the northern bank with x-coordinates b(1) ... b(n). You want to connect as many north-south pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, you can only connect city i on the northern bank to city i on the southern bank.