Dynamic Programming

Haneul Shin

April 2018

1 Introduction

Dynamic Programming is a technique used to solve problems with recurrence relations. The idea behind Dynamic Programming is to divide a problem into various sub-problems and store their results to use later for bigger problems instead of recomputing these results.

As a motivating example, suppose you were asked to compute the n^{th} Fibonacci number, or F_n, for all n from 1 to 100. Recall that $F_0 = 0, F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. We compute each of the values as follows:

$$F_2 : F_2 = F_1 + F_0 = 1 + 0 = 1,$$

$$F_3 : F_2 = F_1 + F_0 = 1 + 0, F_3 = F_2 + F_1 = 1 + 1 = 2,$$

$$F_4 : F_2 = F_1 + F_0 = 1 + 0, F_3 = F_2 + F_1 = 1 + 1 = 2, F_4 = F_3 + F_2 = 2 + 1 = 3,$$

$$F_5 : F_2 = F_1 + F_0 = 1 + 0, F_3 = F_2 + F_1 = 1 + 1 = 2, F_4 = F_3 + F_2, F_5 = F_4 + F_3 = 3 + 2 = 5,$$

and so on. It's easy to see that we're wasting time by recomputing each of the values $F_2, F_3, ..., F_{n-1}$ to find F_n; we can simply use our previous results! This will save a lot of time as n gets large:

$$F_2 : F_2 = F_1 + F_0 = 1 + 0 = 1,$$

$$F_3 : F_3 = F_2 + F_1 = 1 + 1 = 2,$$

$$F_4 : F_4 = F_3 + F_2 = 2 + 1 = 3,$$

$$F_5 = F_4 + F_3 = 3 + 2 = 5,$$

and so on. This example demonstrates why Dynamic Programming is so powerful. By sacrificing some space for storage (in this example, storing the values of $F_0, F_1, ..., F_{n-1}$), we can save lots of time later.

We now discuss a classic problem involving dynamic programming.

2 Classic Problems

Problem 1 (Knapsack). Given n objects of weights $w_1, w_2, ..., w_n$ and values $v_1, v_2, \ldots, v_n$, find the maximum value of a subset of the objects with weight less than W (value and weight and additive).

The idea behind this problem is to fill the knapsack in the way that will optimize the value of the objects inside. As n gets large, we certainly don't want to test each subset from scratch. Thus, we approach this problem recursively.

Let $m[n+1][W+1]$ be a Boolean array, where $m[i][w]$ is the maximum possible value of a subset of the first i items with total weight at most w.

- If $i = 0$, then $m[0][w] = 0$ for any w.
- If $i > 0$ and $w[i] > w$, then $m[i][w] = m[i][w-1]$ since the new item's weight already exceeds the maximum weight.
- Otherwise, $m[i][w] = \max(m[i-1][w], m[i-1][w-w_i] + v_i)$, since we can either keep the optimal subset of the previous $i-1$ objects with weight at most w, or we can take the optimal subset of the previous $i-1$ objects with weight at most $w - w_i$ and add our current object.

Below is my implementation in C++.

```
int knap () {
    for (int j = 0; j <= W; j++) {
        m[0][j] = 0;
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= W; j++) {
            if (w[i] > j) {
                m[i][j] = m[i - 1][j];
            } else {
                m[i][j] = max (m[i - 1][j], m[i - 1][j - w[i]] + v[i]);
            }
        }
    }
    return m[n][W];
}
```

The following is another classic problem.

Problem 2 (Equal-Summed Subsets). Given a set of n elements $S = \{a_1, a_2, \ldots, a_n\}$, determine if S can be partitioned into two subsets such that the sum of the elements in both subsets is equal.

We first compute *sum*, the sum of the elements in S. If *sum* is odd, this is clearly impossible, so we only consider when *sum* is even, so the sum of the elements in the desired subsets must be $\frac{sum}{2}$. To see if we can construct a solution that satisfies the condition, we use Dynamic Programming.

Let $p[sum/2][n+1]$ be a Boolean 2D array such that $p[i][j]$ is true if there exists a subset of the first j elements that sums to i. Then, $part[0][i]$ is clearly true for all i, while $part[i][0]$ is false for all i.

To determine the rest of the values in the matrix, note that $p[i][j]$ is true if either $p[i][j-1]$ is true or if $p[i - a[j-1]][j-1]$ is true, or in other words, if we could achieve the sum with the first $j-1$ elements, or we could achieve exactly $a[j-1]$ less than the sum. Our final answer is thus $p[sum/2][n]$.

Below is my implementation in C++.

```
bool part (int a[], int n) {
    int sum = 0;
    bool p[sum / 2 + 1][n + 1];
    for (int i = 0; i < n; i++) {
        sum += a[i];
    }
    for (int j = 0; j <= n; j++) {
        p[0][j] = true;
    }
    for (int i = 1; i <= sum / 2; i++) {
        for (int j = 1; j <= n; j++) {
            if (p[i][j - 1] || (i >= a[j - 1] && p[i - a[j - 1]][j - 1])) {
                p[i][j] = true;
            }
        }
    }
    return p[sum / 2][n];
}
```

To solve the previous two problems, we used the **bottom-up** approach. We started with the simplest subproblems and used them to solve bigger problems.

3 Examples

Here are two examples of problems that can be solved using Dynamic Programming. I provide solutions, but I recommend that you try implementing them yourself.

Problem 3 (USACO Dec 15 Gold 2). Bessie has a maximum fullness of T ($1 \leq T \leq 5,000,000$). Eating an orange increases her fullness by A, and eating a lemon increases her fullness by B ($1 \leq A, B \leq T$). Additionally, if she wants, Bessie can drink water at most one time, which will instantly decrease her fullness by half (and will round down). Help Bessie determine the maximum fullness she can achieve!

This problem sounds quite similar to the Knapsack problem. The main difference here is that there are two states: before drinking and after drinking. The possible values of fullness after drinking depend on the values before drinking. Thus, we define $pos[2][t+1]$ such that $pos[0][t]$ is true if Bessie can achieve fullness t before drinking, and $pos[1][t]$ is true if Bessie can achieve fullness t after drinking.

We first determine which values are possible before drinking recursively. We begin by setting $pos[0][0] = true$, and thereafter, we have $pos[0][i]$ is true if $pos[0][i-a]$ or $pos[0][i-b]$ is true. Then, $pos[1][i]$ is true if $pos[0][2i]$ or $pos[0][2i+1]$ is true, or if $pos[1][i-a]$ or $pos[1][i-b]$ is true. We can now find the largest index i such that $pos[0][i]$ or $pos[1][i]$ is true, which yields our answer.

Problem 4 (USACO Jan 17 Gold 2, Adapted). Cows like to play a variant of the game “Rock, Paper, Scissors” they call ”Hoof, Paper, Scissors”. In the game, two cows count to three and then each simultaneously makes a gesture that represents either a hoof, a piece of paper, or a pair of scissors. Hoof beats scissors, scissors beats paper, and paper beats hoof. For example, if the first cow makes a ”hoof” gesture and the second a ”paper” gesture, then the second cow wins. Of course, it is also possible to tie, if both cows make the same gesture.

Farmer John wants to play against his prize cow, Bessie, at N games of ”Hoof, Paper, Scissors” ($1 \leq N \leq 100,000$). Bessie, being an expert at the game, can predict each of FJ’s gestures before he makes it. Unfortunately, Bessie, being a cow, is also very lazy. As a result, she tends to play the same gesture multiple times in a row. In fact, she is only willing to switch gestures at most K times over the entire set of games ($1 \leq K \leq 20$). For example, if $K = 2$, she might play ”hoof” for the first few games, then switch to ”paper” for a while, then finish the remaining games playing ”hoof”. Given the sequence of gestures FJ will be playing, please determine the maximum number of games that Bessie can win.

Bessie can choose K moves and when to play them within her sequence of N moves, so there is a total of $2^K \cdot \binom{N}{K}$ possible sequences of moves. As N gets large, checking all these cases individually would take far too long. However, we notice that different sequences that share the first x moves are related to each other, so we find a dynamic programming solution. We can define the 2d array $v[N+1][K+1][3]$ so that $v[I][j][k]$ is the maximum number of wins among sequences i moves long using j switches with last move k. From here, we can find a recurrence relation to build this array, which yields the desired answer. This solves the problem with a runtime of $O(NK)$.

4 Problems

Problem 5 (USACO Feb 17 Gold 2). Farmer John raises N breeds of cows ($1 \leq N \leq 1000$), conveniently numbered $1, ..., N$. Some pairs of breeds are friendlier than others, a property that turns out to be easily characterized in terms of breed ID: breeds a and b are friendly if $|a - b| \leq 4$, and unfriendly otherwise.

A long road runs through FJ's farm. There is a sequence of N fields on one side of the road (one designated for each breed), and a sequence of N fields on the other side of the road (also one for each breed). To help his cows cross the road safely, FJ wants to draw crosswalks over the road. Each crosswalk should connect a field on one side of the road to a field on the other side where the two fields have friendly breed IDs (it is fine for the cows to wander into fields for other breeds, as long as they are friendly). Each field can be accessible via at most one crosswalk (so crosswalks don't meet at their endpoints).

Given the ordering of N fields on both sides of the road through FJ's farm, please help FJ determine the maximum number of crosswalks he can draw over his road, such that no two intersect.

Problem 6 (USACO Dec 16 Gold 2). Every day, Farmer John walks through his pasture to check on the well-being of each of his cows. On his farm he has two breeds of cows, Holsteins and Guernseys. His H Holsteins are conveniently numbered $1, ..., H$ and his G Guernseys are conveniently numbered $1, ..., G$ ($1 \leq H \leq 1000, 1 \leq G \leq 1000$). Each cow is located at a point in the 2D plane (not necessarily distinct).

Farmer John starts his tour at Holstein 1, and ends at Holstein H.He wants to visit each cow along the way, and for convenience in maintaining his checklist of cows visited so far, he wants to visit the Holsteins and Guernseys in the order in which they are numbered. In the sequence of all $H + G$ cows he visits, the Holsteins numbered $1, ..., H$ should appear as a (not necessarily contiguous) subsequence, and likewise for the Guernseys. Otherwise stated, the sequence of all $H + G$ cows should be formed by interleaving the list of Holsteins numbered $1, ..., H$ with the list of Guernseys numbered $1, ..., H$.

When FJ moves from one cow to another cow traveling a distance of D, he expends D^2 energy. Please help him determine the minimum amount of energy required to visit all his cows according to a tour as described above.