Dynamic Programming Practice

Albert Gural

March 2, 2012

1 Introduction

By now, most of you should know what dynamic programming is and when to use it. However, actually figuring out how to use it can be a bit trickier. Often, the best way to understand is to do, so here are a few problems to work on. Instead of jumping to the computers, see if you can draw the DP array for sample input.

2 Example

Before setting you off on the problems, let's review a very common contest problem: the Multiple Knapsack problem.

2.1 Multiple Knapsack Problem

Given an unlimited number of several different types of objects numbered $1..N$ which have weight W_j and value V_j, determine the maximum value you can pack in a knapsack, if your knapsack can only hold a maximum total weight of C.

2.2 General Strategy

We will begin by trying to apply the general strategy.

1. **Definition of State Variables**: Figure out what the substates are and what variables change based on the substate. For the Knapsack problem, the variable that changes is capacity, i, and the substates are each capacity between 0 and C, inclusive.

2. **Base Cases**: Determine the base cases - find the solution for very small substates such that you can build on them in the next step. For the Knapsack problem, the base case is $dp[0] = 0$. In other words, a knapsack of capcity 0 will hold 0 value.

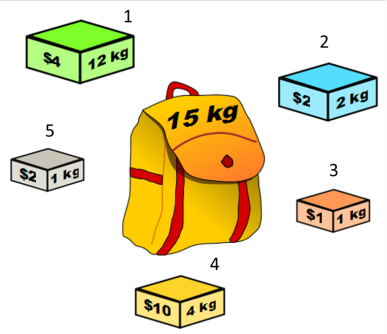
3. **Transition Functions**: Determine the relationship between higher substates and lower substates. Essentially, we are looking for a recursive relationship, much like the one for the Fibonacci sequence ($a_n = a_{n-1} + a_{n-2}$). For the Multiple Knapsack problem, the transition function is:

$$dp[i] = \max_{\text{item } j} [dp[i - W[j]] + V[j]]$$

   Here's why: Suppose we are at $dp[i]$ and the dp array is filled in for all subcapacities less than i. Now let's say we want to consider whether having item j in a knapsack of capacity i is good. Well, if the knapsack *does* contain item j, then the value of that knapsack will be the value of item j plus the value of the knapsack with capacity $i - W_j$. Maximizing over all items gives us the *best* possible value a knapsack of capacity i can have. Reread this paragraph if you didn't completely comprehend why the transition function works.

2.3 Sample Test Case

Here's a pictoral representation for the sample input.



2.4 Solution DP Array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	2	4	6	10	12	14	16	20	22	24	26	30	32	34	36

Figure 1: DP Array - Top Row Represents Substate Capacity, Bottom Row Represents Optimum Value

2.5 Analysis

First of all, what is the running time of this algorithm? Well, the program runs through C substates, and for each substate, it minimizes over all N objects. This is clearly an $O(CN)$ algorithm. We can kind of predict what the problem bounds should look like ($C < 100,000$, $N < 100$). Of course, any bounds on C and N such that $CN < 100,000,000$ is reasonable.

3 DP Practice Problems

Try to do these problems by first following the general strategy ("Definition of State Variables", "Base Cases", "Transition Functions"), then come up with a small test case and draw the DP array. You should also determine the runtime of your algorithm. See if you can figure out what the bounds of the problem should be!

3.1 0-1 Knapsack Problem [Traditional]

Same as the multiple knapsack problem, but you may only take one of each object (instead of unlimited numbers of each object).

3.2 Making Change [Traditional]

Given a certain number of coins $1..N$ of value V_i, find the minimum number of coins needed to get to a certain goal price. Bonus: Figure out the coins that would be used.

3.3 Balanced Partition [Traditional]

Given a set of integer values, determine the best way to partition them so that the two subsets would be as close to equal as possible.

3.4 Coin Picking Game [Traditional]

Given a row of coins $1..N$ of value V_i in a line and two players who take turns taking either the left-most coin or the right-most coin, determine the amount player 1 will get if both players play optimally.

3.5 Binary Cow Lines [Albert Gural, 2012]

Given binary strings S_i, find the number of N-length binary strings that do not contain any of the S_i as substrings. Each S_i is less than 8 characters long.

3.6 Cow Checkers [Amlesh Jayakumar, 2011]

A single checker piece is placed on a large checkerboard at location (X, Y), with North in the positive Y direction and East in the positive X direction. Two players take turns moving the checker piece. On each turn, a player may move the piece South, West, or Southwest. If both players play optimally, determine who will win.

4 More Practice

Brian Dean compiled a nice list of DP problems, a few of which I included as sample problems:

`http://people.csail.mit.edu/bdean/6.046/dp/`