Dynamic Programming for Contests

Saketh Are and Andre Kessler - 10/01/10

Dynamic Programming (DP) is a problem solving technique that entails breaking down a problem into easier, simpler subproblems of a similar structure. By storing the optimal solutions to each subproblem and reusing them as needed, the runtime for a recursively defined solution can be reduced drastically.

1 Introduction

A classic example of a problem which can be solved using dynamic programming is the task of computing the nth Fibonacci number. The Fibonacci numbers are defined by $F_n = F_{n-1} + F_{n-2}$ and $F_1 = F_2 = 1$. A naïve recursive approach might attempt to simply define a base case and a recurrence relation, computing the desired result in $O(2^N)$ time. However, a DP approach reduces this runtime to a much more favorable $O(N)$. This can be done in one of two ways: top-down or bottom-up. The top-down approach to DP is very similar to recursion. The same structure of base case and recurrence is used; however, we store the values we calculate, essentially turning them into additional base cases. This technique, known as memoization, eliminates unnecessary recomputation of results. Conversely, the bottom-up approach begins with the first two values, and builds up to the final result using the recurrence. Generally most of the work involved in a DP solution is formulating the subproblems and recurrence relations. Once you're sure that your solution is correct, the actual implementation is usually quite simple compared to other types of algorithms. For the rest of this lecture we will describe the various types of DP problems that appear on USACO contests, and provide a few practice problems for each type. When it comes to USACO problems, DP is one of the most useful techniques you can learn. You can pretty much expect at least one DP problem on every Silver and Gold contest.

2 Categories

Dynamic programming problems in computing contests tend to fall into a couple of different categories. This is by no means intended to be an exhaustive list, but this should be a helpful reference. Recognizing problems as knapsack-like or interval/tree based can lead to a solution more quickly, and thinking about how to deal with the base cases in each of these types is an instructive exercise.

2.1 Knapsack

The general integer knapsack problem goes like this: given a knapsack of a particular capacity and objects each of which have a size and a value, find the set of objects which maximizes the total value but whose total size is constrained to fit within the knapsack. Some knapsack problems will allow you to take an unlimited number of each type of object; others will only allow a limited number of objects. Our solution to this problem is to do dynamic programming on the maximum value that a knapsack of each size can have in it. Suppose we have a table of the maximum value that a knapsack of any size can have in it called `dp[1..N]`. To consider adding another item of size S, iterate over the `dp` array and see if placing the current object into the knapsack of size i yields a better set than the current best knapsack of size $S + i$. If we have an unlimited number of each object you'll want to be iterating forwards over the array; otherwise, iterate in reverse, from N to 1. For example, suppose the cows are trying to fit as much hay as possible inside a bin of limited capacity. There are a limited number of each type of bale. What is the "size" and "value" in this case? In what order should we iterate over the `dp` array?

2.2 1-D

This is a pretty generic class, as it includes all DP problems in which a state is describable by one variable. The Fibonacci numbers problem is an example of this, as the state is describable by exactly one integer: n. These are generally the simplest DP problems to solve since the state should definitely fit within memory.

2.3 Interval-Based

Dynamic programming problems involving expansion of an interval are known, surprisingly enough, as "interval-based" DP. Your base cases will be trivial intervals of a single element and you expand the length from there. For example, suppose you're playing a game in which there are N coins of differing values laid out in a row. On your turn, you can remove a coin from either end of the line. What is the maximum amount of money you can win, assuming your opponent plays optimally?

2.4 Tree-Based

Every DP problem is at its root tree-based! This is a result of our optimal substructure requirement and the fact that DP is in some sense recursion in reverse. Nevertheless, some DP problems are more clearly tree-based than others - especially the ones in which you're given a tree explictly in the problem statement. Most often the leaves of the tree will be the base cases, and you build up parent values by looking at their children.

2.5 Statespace BFS

When we do a recursive complete search, we're usually doing a depth-first search of the recursion tree. However, in some cases it's better to use a breadth-first search of the tree. Memoization and/or hashing allows us to ignore previously checked states while the basic properties of breadth first search mean that we find "shortest" solutions to the problem. This is best explained by an example. Suppose that you have a set of integers in the range $0..2^{16}$ and you want to combine them with the XOR operation to form the integer of minimum hamming distance (number of bits that differ) from some goal integer G. Find this integer and the minimum number of XOR operations needed to construct it.

2.6 Row-by-Row

Problems on arrays in which the next state only depends on the state of the previous X rows are generally solved via so-called "row-by-row" dynamic programming algorithms. Oftentimes, with small enough arrays, you can compress the previous state into a binary number. Because of this, row-by-row is really a subset of "state compression" DP problems – the total state space is far too large to consider, so we need to take advantage of symmetry wherever possible and reduce the possiblities. As an example of row-by-row, suppose you're given a grid representing a farm, with certain squares marked as unusable. Find the number of ways to plant non-adjacent squares while only planting on usable squares.

2.7 Large State

Frequently, the state space of the problem will be too large to hold within memory and there isn't an lot of nice symmetry we can use to compress it, like in row-by-row. In these cases, you will want to think a lot about how to express your state in fewer variables. The "sliding-window" trick is one example of this state space reduction and should be at the top of your mind when encountering these large state problems. Rotating arrays can also be useful for this. If the state space just absolutely too large for memory even after you've reduced it as much as possible, you should be thinking about memoization and hashing. Memoization allows you to avoid computing some unnecessary states and hashing will mean you can map the smaller space to an array in memory.

3 Practice

The main difficulty usually lies in recognizing that a problem *is* DP. Practice is the best way to get better at that, so here is a set of problems. Try to classify each problem here according to the categories we discussed earlier.

1. Given a sequence of real numbers $a_1, a_2, \ldots, a_n$, determine a contiguous subsequence $a_i, \ldots, a_j$ for which the sum of the elements of the subsequence is maximized.

2. Given a sequence $a_1, a_2, a_3, \ldots, a_n$, find the longest increasing subsequence in $O(N \log N)$ time.

3. Consider a two-dimensional map with a horizontal river passing through its center. There are N cities on the southern bank with x-coordinates $a_1, a_2, a_3, \ldots, a_N$ and N cities on the northern bank with x-coordinates $b_1, b_2, b_3, \ldots, b_N$. You want to connect as many north-south pairs of cities as possible with bridges such that no two bridges cross. However, when connecting cities, you can only connect city i on the northern bank to city i on the southern bank.

4. Given two strings A and B of lengths N and M respectively, you want to transform A into B using as few of the following three operations as possible: insert a character, delete a character, or replace a character with another. Extension: suppose inserting costs C_I, deletion costs C_D, and replacement costs C_R.

5. Given n arcs of a circle, each specified by two points on the circle as well as an associated cost, compute a minimum-cost collection of these arcs that covers the entire circle.

6. In the newest census of Jersey Cows and Holstein Cows, Wisconsin cows have earned three stalls in the Barn of Representatives. The Jersey Cows currently control the state's redistricting committee. They want to partition the state into three equally sized voting districts such that the Jersey Cows are guaranteed to win elections in at least two of the districts. Wisconsin has $3K$ ($1 \leq K \leq 60$) cities of population 1000, each with a known number of Jersey Cows. Find a way to partition the state into three districts, each with K cities, such that the Jersey Cows have the majority percentage in at least two of districts. (USACO Feb05)

7. Farmer John has decided to give each of his cows a cell phone in hopes to encourage their social interaction. This, however, requires him to set up cell phone towers on his N ($1 \leq N \leq 10,000$) pastures (conveniently numbered $1..N$) so they can all communicate. Exactly $N-1$ pairs of pastures are adjacent, and for any two pastures A and B ($1 \leq A \leq N$; $1 \leq B \leq N$; $A \neq B$) there is a sequence of adjacent pastures such that A is the first pasture in the sequence and B is the last. Farmer John can only place cell phone towers in the pastures, and each tower has enough range to provide service to the pasture it is on and all pastures adjacent to the pasture with the cell tower. Help him determine the minimum number of towers he must install to provide cell phone service to each pasture. (Jeffrey Wang, 2007)

8. Given n integers $a_1, a_2, \ldots, a_n$ each in the range $0 \ldots B$, how can you partition these into two sets having the closest possible sum?

9. Bessie is in a grocery store with a list of P items to purchase. The store has N aisles. Given the regular distances between aisles and shelves, along with the locations of the P items to purchase, find a route that minimizes the total distance Bessie has to walk. (Cox/Kolstad 2006)

10. A beetle finds itself on a thin horizontal branch. There are N drops of dew on that same branch, each holding M units of water. Their beetle-based integer coordinates are $x_1, x_2, \ldots, x_N$. The sun is shining brightly and evaporating one unit of water per minute. The beetle is so thirsty that it can drink a drop of dew in no time at all, and the beetle can crawl at one unit of length per minute. But will this pay off? That's what buzzes the beetle. Write a program which, given coordinates of dew drops, calculates the maximal amount of water that the beetle can possibly drink. Note that $0 \leq N \leq 300$, $1 \leq M \leq 1,000,000$, and $-10,000 \leq x_i \leq 10,000$.