Greedy Algorithms

Nick Haliday, Owen Hoffman

2012-10-25

1 Intro

Greedy algorithms are rather tricky beasts. They are often very easy to conceive but much harder to prove correct. Fortunately, proof is not necessary for USACO, but you do want at least a strong intuition that an algorithm will work, and even that can be hard to develop when in comes to greedy algorithms. Greedy problems are probably the least straightforward of all non-ad-hoc problems to appear in bronze, so it's good to get some exposure early on.

2 The Basic Idea

The basic idea behind greedy algorithms is that a choice that is immediately or locally optimal is also globally optimal. In other words, we look for the best possible choice to make given our current solution, then add it to our solution and repeat. The problem with this idea is that it often just doesn't work. The trap greedy algorithms fall into is that they can only find local optima. Sometimes you need to make a bad choice early on in order to get a higher payoff later. Greedy algorithms do not work on problems where this is the case.

In textbooks, problems solvable by greedy algorithms are typically presented as having two properties: the greedy choice property, described in the last paragraph, and the optimal substructure property. The optimal substructure property simply states that optimal solutions to a given problem can be constructed using optimal solutions to simplified version of it. When we make a greedy choice we reduce the original problem to simplified version. Dynamic programming also relies on optimal substructure, but the key difference between the two is that greedy only makes one choice; DP makes several or possibly all choices. They achieve their efficiency by different means. Greedy simple cuts out swaths of subproblems as irrelevant. DP does not have that luxury, and relies on overlap between subproblems, so that solutions can be reused without having to be recomputed. This distinction is important because many problems that appear solvable with greedy actually require DP.

3 A Counterexample to Greedy

We'll start off with a couterexample instead of a correct algorithm. Wrong greedy algorithms tend to be harder to spot than right greedies. The classic example of a problem where greedy fails is the problem of finding the most efficient way to make change: given a sequence of coin denominations, $(v_i; 1 \leq i \leq n)$, with $v_1 = 1$, find the smallest number of coins you need to make a sum of value V. Clearly you can always find make the sum at least one way, just use V coins of value 1, but we want the optimal solution.

One greedy algorithm is to always take the highest value coin that still doesn't overshoot and add it to our current set until we have a sum of V. Unfortunately this fails for some values of v_i and V. A nice counterexample is $v = (1, 3, 4)$ and $V = 6$. In this case our algorithm chooses a coin of value 4, then 2 coins of value 1. This is not optimal. The best solution is 2 coins of value 3. (Funnily enough the greedy algorithm always works with some sets of denominations, including that of the United States.)

Note that you can actually solve this problem in $O(nV)$ time using a DP algorithm, but we won't go into that. Look up "knapsack" and "change-making" if you're interested.

There's really no good, general way to tell when a greedy algorithm is correct besides proving it correct or finding a counterexample. What you can do is build intuition by solving problems.

4 Sometimes It Works

Suppose you are choosing your schedule for next year. You know the start times and finish times of each class, and, being a crazy TJ student, you want to maximize the number of classes you can take. In fact this is all you care about. How many classes can you fit into a schedule? Classes may not overlap of course. We could try all subsets of intervals, checking if they form a valid schedule, but this would run in $O(n2^n)$, which is a pretty terrible running time.

To start, sort the intervals by finishing time, then by starting time. Then choose the first interval, add it your schedule, and skip ahead until you find one that starts after the one you just added, then repeat. The sweep through intervals is $O(n)$ but the initial sort raises the running time to $O(n \log n)$, which is still quite fast. Does this algorithm work? Let's try to prove that it does.

Is there ever a reason to not include the interval with the earliest finish time? It turns out there isn't. Choosing an interval with a higher finish time only restricts the number of remaining intervals we have to choose from, and it still only adds 1 to the size our set. So our algorithm works.

Note that this solution does not work if we're trying to maximize something else besides the size of our set, e. g., the number of hours spent in class. In that case you need to break out DP, but that's beyond the scope of this lecture.

5 Proof Techniques

The methods used to prove correctness can vary greatly, but there are some common themes. Often you can only show that your algorithm yields one of multiple optimal solutions, so you're trying to show it's at least as good as any other solution, not necessarily better. Some ways you can do this include taking any solution that doesn't look like one your greedy algorithm would produce, make a change or swap that makes it closer to such a solution, and show this change can only improve the solution, then repeatedly make this change. This is called an *exchange argument*.

Induction and proof by contradiction are also useful, though the details tend to vary even more compared to exchange arguments.

As an example of an exchange argument, a useful and simple inequality in math is the rearrangement inequality, which says that if we have two sequences of numbers (a_i) and (b_i), and we want to maximize the sum of the products of their elements

$$\sum_i a_i b_i$$

we should sort a and b so that the smallest elements of each form one product, then the next smallest of each another product, and so on.

To prove this, suppose we have a some elements out of order, say, two indicices i and j, such that $a_i \leq a_j$, but $b_i > b_j$. Originally the part of the sum they compose has value $a_i b_i + a_j b_j$. If we swap b_i and b_j, the new value will be $a_i b_j + a_j b_i$. Do we have $a_i b_j + a_j b_i \geq a_i b_i + a_j b_j$? In fact, we do. Subtract one side from the other and you get $a_i b_j - a_i b_i + a_j b_i - a_j b_j$, and we want to show this is greather than or equal to 0. There are lots of terms in common so let's try factoring. We get $(a_i - a_j)(b_j - b_i)$. But each of the terms in the product is greater than or equal to 0 by assumption. So we can set the entire product to be at least 0 and work backwards.

There may have not been an obvious algorithm going on there, but it was really just hidden by a sort. A lot of greedy algorithms can actually just be replaced by sorts, but the proof will be essentially the same.

6 Problems

1. You are recording some movies to a VHS cassete. You only have one unfortunately. You'll be watching each movie with friends over the coming weeks. You know each week Sreenath will sneak in through a window and rewind the casette to the beginning. Given that each week you have to fastforward through all the movies preceding the one you want to watch, in what order should you record the movies to minimize the total time spent fastforwarding? You're given the length of each movie as a sequence $(L_i; 1 \leq i \leq n)$.

2. Now you know that you will watch each movie movie a certain number of

times, given by a sequence $(F_i; 1 \leq i \leq n)$. Can you modify your algorithm from 1 to solve this problem?

3. Barn Repair [USACO Training Pages]

   It was a dark and stormy night that ripped the roof and gates off the stalls that hold Farmer John's cows. Happily, many of the cows were on vacation, so the barn was not completely full.

   The cows spend the night in stalls that are arranged adjacent to each other in a long line. Some stalls have cows in them; some do not. All stalls are the same width.

   Farmer John must quickly erect new boards in front of the stalls, since the doors were lost. His new lumber supplier will supply him boards of any length he wishes, but the supplier can only deliver a small number of total boards. Farmer John wishes to minimize the total length of the boards he must purchase.

   Given M (1 <= M <= 50), the maximum number of boards that can be purchased; S (1 <= S <= 200), the total number of stalls; C (1 <= C <= S) the number of cows in the stalls, and the C occupied stall numbers (1 <= stall_number <= S), calculate the minimum number of stalls that must be blocked in order to block all the stalls that have cows in them.

   Print your answer as the total number of stalls blocked.