The Greedy Algorithm

Tom Morgan, Jacob Steinhardt

October 06, 2006

The Basics

The greedy algorithm is perhaps the most straightforward and powerful algorithm you will learn. The idea is simple: given a problem in which you are trying to minimize or maximize something and have some system you can operate on, you ask what the best thing you can do if you were to only do one. Next, you update the system and repeat.

When it works

The greedy algorithm can be used when a local change (as in taking the current best action) will not detrimentally affect the system as a whole. A classic example is as follows.

Say you are low on funds and decide to rob a convenience store. To your misfortune you find that the convenience store has been robbed by another criminal moments before hand. As the store now lacks any cash, you decide to steal and resell some of the store's donuts. The store has a variety of donuts which each can be sold for different amounts. For example, they may have ten donuts that are worth $2 each, five that are worth $3 each and three that are worth $1 each. You have room for exactly seven donuts in your pack. What is the maximum amount of money you can make?

The greedy approach is to say: I have seven spots, what is the best donut I can pick for the first spot? I take it and repeat. This amounts to taking as many of the most valuable ones as possible and then moving on to the next most. We quickly find that the answer in this case is to take five $3 donuts and two $2 donuts.

When it doesn't work

Taking the problem from before, it is no longer possible to solve if the donuts come in different sizes and we can not take fractions of donuts. For example if we have six spaces, and there is a donut of size four that is worth $8 and there are two donuts of size three that are worth $5 each. The greedy solution says to first take the $8 one but then we don't have room for any more, while optimally we could fit the two $5 donuts for a total value of $10.

Heuristics

The way in which the current best is evaluated is called a heuristic. Given the previous example (in which greedy doesn't work) possible heuristics include: the donut worth the most, the donut that takes the least space and (most logically) the donut with the highest ratio of value to size. What can make greedy problems difficult is having complicated heuristics.

When approaching a greedy problem (or a problem you are trying to get partial credit on) do not be afraid to use multiple heuristics. By running multiple greedy algorithms and taking the best result, you can maximize your partial credit and on occasion simulate more complicated (and accurate) heuristics.

Advanced Techniques

There are many techniques for proving or deriving Greedy algorithms, each requiring various levels of mathematical sophistication.

Usually, we can show that a Greedy algorithm work if all choices are independent of each other. Additionally, if the **swap** of two elements enacts only a local change (that is, all other elements remain unaffected, or the total value remains unaffected), then a Greedy algorithm will also work. Additionally, the Greedy can be **derived** by writing an inequality that will determine when the second element should come after the first. Then, a sort should be performed using this criterion as the comparison method. This usually works (see below proof; it will work if a similar proof applies). Here is an example:

(US Open 2006) N cows go through a milking queue. The first part of the process takes A_i seconds for cow i, and the second part takes B_i seconds for cow i. Only one farmer is available for each part of the process, so it is feasible that the second farmer will be doing nothing while the first farmer is stuck milking a cow, or there can be a backup if the second farmer takes too long milking a cow (though the first farmer can continue milking). Determine in $O(NlogN)$ time the minimum time it takes to milk al lthe cows.

First note that the total time is going to be the maximum value across i of $A_1 + A_2 + \ldots + A_i + B_i + B_{i+1} + \ldots + B_n$. Then, imagining the swap of two cows at the point where we switch from A's to B's, a bit of case analysis (when the maximal index for i changes and when it doesn't) shows us that the swap will decrease our time if and only if $min(B_2, A_1) > min(A_2, B_1)$. Additionally, in an optimal arrangement this will be true for every consecutive pair, and if it is true for every consecutive pair, it is true for every pair (this is not always true, but it is for this particular comparator). In particular, this means that there is only one such arrangement disregarding cases of equality. So we should be able to find it by performing a sort using this comparator. We do this and then we're done.

A second technique involves something called the Well-Ordering Principle, which basically works by proving a greedy by assuming a smallest case in which greedy fails, then finding a smaller case (this is impossible since we just specified we had the smallest case). Here is an example:

(Topcoder SRM 317) Each square in an RxC grid is either empty or has some color. You want to cut the grid, starting and ending at points adjacent to both an empty point and a non-empty point. A cut does not have to be straight, but it can never go through a monotone region, and ends as soon as it hits an empty point (which includes previous cuts, or a previous part of the current cut). Determine, if it is possible, the minimum number of cuts required to separate the grid into monotone regions. ($R \leq 50$, $C \leq 50$).

We claim that cutting greedily will work. We can take the smallest point at which greedy fails. Then make a cut greedily. One of the two halves of the area we just cut must then do worse than before, but in both cases greedy must work, since we just took the smallest point at which greedy should fail. Therefore cutting greedily will always work (there is also a solution that is easier to code and employs graph theory, which will be discussed later in the year).

Problems

1. **Fractional Knapsack [Traditional]**
   You are a thief, who has entered a jewelry store. In the store, there are N boxes, each with a type of jewel worth some value V and weighing some weight W. Your knapsack can only carry X units of weight. If there is an infinite amount of jewelry in each box, what is the maximum value you can fit in your knapsack if you can choose only to take part of a jewel?

2. **Barn Repair [1999 USACO Spring Open]**
   There is a long list of stalls, some of which need to be covered with boards. You can use up to N ($1 \leq N \leq 50$) boards, each of which may cover any number of consecutive stalls. Cover all the necessary stalls, while covering as few total stalls as possible.

3. **PlayGame [TopCoder 2004]**
   You are playing a computer game and a big fight is planned between two armies. You and your computer opponent will line up your respective units in two rows, with each of your units facing exactly one of your opponent's units and vice versa. Then, each pair of units, who face each other will fight and the stronger one will be victorious, while the weaker one will be captured. If two opposing units are equally strong, your unit will lose and be captured. You know how the computer will arrange its units, and must decide how to line up yours. You want to maximize the sum of the strengths of your units that are not captured during the battle.

4. **Indivisible [TopCoder 2005]**
   Given a positive integer n, you are to find the largest subset S of 1,...,n such that no member of S is divisible by another member of S. If there is more than one subset of the maximum size, choose the lexicographically earliest such subset. A subset S is lexicographically earlier than a subset T if the smallest element that is a member of one of the subsets, but not both, is a member of S. Return the subset S as a int[] in increasing order.

5. **MatrixTransforming [TopCoder 2006]**
   Given two matrices a and b, both composed of zeroes and ones, return the minimal number of operations necessary to transform matrix a into matrix b. An operation consists of flipping (one becomes zero and zero becomes one) all elements of some contiguous 3×3 submatrix. If a cannot be transformed into b, return -1.