Greedy Algorithm

Greedy Algorithm

Sample Problem: 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 <= N <= 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.

The Idea

The basic idea behind greedy algorithms is to build large solutions up from smaller ones. Unlike other approaches, however, greedy algorithms keep only the best solution they find as they go along. Thus, for the sample problem, to build the answer for N = 5, they find the best solution for N = 4, and then alter it to get a solution for N = 5. No other solution for N = 4 is ever considered.

Greedy algorithms are **fast**, generally linear to quadratic and require little extra memory. Unfortunately, they usually aren't correct. But when they do work, they are often easy to implement and fast enough to execute.

Problems

There are two basic problems to greedy algorithms.

How to Build

How does one create larger solutions from smaller ones? In general, this is a function of the problem. For the sample problem, the most obvious way to go from four boards to five boards is to pick a board and remove a section, thus creating two boards from one. You should choose to remove the largest section from any board which covers only stalls which don't need covering (so as to minimize the total number of stalls covered).

To remove a section of covered stalls, take the board which spans those stalls, and make into two boards: one of which covers the stalls before the section, one of which covers the stalls after the section.

Does it work?

The real challenge for the programmer lies in the fact that greedy solutions don't always work. Even if they seem to work for the sample input, random input, and all the cases you can think of, if there's a case where it won't work, at least one (if not more!) of the judges' test cases will be of that form.

For the sample problem, to see that the greedy algorithm described above works, consider the following:

Assume that the answer doesn't contain the large gap which the algorithm removed, but does contain a gap which is smaller. By combining the two boards at the end of the

smaller gap and splitting the board across the larger gap, an answer is obtained which uses as many boards as the original solution but which covers fewer stalls. This new answer is better, so therefore the assumption is wrong and we should always choose to remove the largest gap.

If the answer doesn't contain this particular gap but does contain another gap which is just as large, doing the same transformation yields an answer which uses as many boards and covers as many stalls as the other answer. This new answer is just as good as the original solution but no better, so we may choose either.

Thus, there exists an optimal answer which contains the large gap, so at each step, there is always an optimal answer which is a superset of the current state. Thus, the final answer is optimal.

Conclusions

If a greedy solution exists, use it. They are easy to code, easy to debug, run quickly, and use little memory, basically defining a good algorithm in contest terms. The only missing element from that list is correctness. If the greedy algorithm finds the correct answer, go for it, but don't get suckered into thinking the greedy solution will work for all problems.

Sample Problems

Sorting a three-valued sequence [IOI 1996]

You are given a three-valued (1, 2, or 3) sequence of length up to 1000. Find a minimum set of exchanges to put the sequence in sorted order.

Algorithm The sequence has three parts: the part which will be 1 when in sorted order, 2 when in sorted order, and 3 when in sorted order. The greedy algorithm swaps as many as possible of the 1's in the 2 part with 2's in the 1 part, as many as possible 1's in the 3 part with 3's in the 1 part, and 2's in the 3 part with 3's in the 2 part. Once none of these types remains, the remaining elements out of place need to be rotated one way or the other in sets of 3. You can optimally sort these by swapping all the 1's into place and then all the 2's into place.

Analysis: Obviously, a swap can put at most two elements in place, so all the swaps of the first type are optimal. Also, it is clear that they use different types of elements, so there is no ``interference'' between those types. This means the order does not matter. Once those swaps have been performed, the best you can do is two swaps for every three elements not in the correct location, which is what the second part will achieve (for example, all the 1's are put in place but no others; then all that remains are 2's in the 3's place and vice-versa, and which can be swapped).

Friendly Coins - A Counterexample [abridged]

Given the denominations of coins for a newly founded country, the Dairy Republic, and some monetary amount, find the smallest set of coins that sums to that amount. The Dairy Republic is guaranteed to have a 1 cent coin.

Algorithm: Take the largest coin value that isn't more than the goal and iterate on the total minus this value.

(Faulty) Analysis: Obviously, you'd never want to take a smaller coin value, as that would mean you'd have to take more coins to make up the difference, so this algorithm works.

Maybe not: Okay, the algorithm usually works. In fact, for the U.S. coin system {1, 5, 10, 25}, it always yields the optimal set. However, for other sets, like {1, 5, 8, 10} and a goal of 13, this greedy algorithm would take one 10, and then three 1's, for a total of four coins, when the two coin solution {5, 8} also exists.

Topological Sort

Given a collection of objects, along with some ordering constraints, such as "A must be before B," find an order of the objects such that all the ordering constraints hold.

Algorithm: Create a directed graph over the objects, where there is an arc from A to B if "A must be before B." Make a pass through the objects in arbitrary order. Each time you find an object with in-degree of 0, greedily place it on the end of the current ordering, delete all of its out-arcs, and recurse on its (former) children, performing the same check. If this algorithm gets through all the objects without putting every object in the ordering, there is no ordering which satisfies the constraints.