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.

Greedy Algorithm

贪心算法

译 by Lucky Crazy & Felicia Crazy

样例：牛棚修理 [1999 USACO 春季公开赛]

Farmer John 有一列牛棚，在一次暴风中，牛棚的一整面墙都被吹倒了，但还好不是每一间牛棚都有牛。Farmer John 决定卖木料来修理牛棚，然而，刻薄的木材提供商却只能提供有限块的木料（木料的长度不限），现在告诉你关着牛的牛棚号，和提供的木材个数 N，你的任务是编程求出最小的木块长度和。（1 <= N <= 50）

贪心思想：

贪心思想的本质是每次都形成局部最优解，换一种方法说，就是每次都处理出一个最好的方案。例如：在样例中，若已经发现 N = 5 时的最优解，那么我们可以直接利用 N = 5 的最优解构成 N = 4 的最优解，而不用去考虑那些 N = 4 时的其他非最优解。

贪心算法的最大特点就是快。通常，二次方级的存储要浪费额外的空间，而且很不幸，那些空间经常得不出正解。但是，当使用贪心算法时，这些空间可以帮助算法更容易实现且更快执行。

贪心的难点：

贪心算法有两大难点：

如何贪心：

怎样才能从众多可行解中找到最优解呢？其实，大部分都是有规律的。在样例中，贪心就有很明显的规律。但你得到了 N = 5 时的最优解后，你只需要在已用上的 5 块木板中寻找最靠近的两块，然后贴上中间的几个牛棚，使两块木板变成一块。这样生成的 N = 4 的解必定最优。因为这样木板的浪费最少。同样，其他的贪心题也会有这样的性质。正因为贪心有如此性质，它才能比其他算法要快。

贪心的正确性：

要证明贪心性质的正确性，才是贪心算法的真正挑战，因为并不是每次局部最优解都会与整体最优解之间有联系，往往靠贪心生成的解不是最优解。这样，贪心性质的证明就成了贪心算法正确的关键。一个你想出的贪心性质也许是错的，即使它在大部分数据中都是可行的，你必须考虑到所有可能出现的特殊情况，并证明你的贪心性质在这些特殊情况中仍然正确。这样经过千锤百炼的性质才能构成一个正确的贪心。

在样例中，我们的贪心性质是正确的。如下：

假设我们的答案盖住了较大的空牛棚连续列，而不是较小的。那么我们把那部分盖空牛棚的木板锯下来，用来把较小的空牛棚连续列盖住，还会有剩余。那么锯掉它们！还给木材商！同时我们的解也变小了。也就是说，我们获得更优的解。所以，靠盖住较大空牛棚连续列的方法无法获得最优解，我们也应该尽量贪心那些距离小的木板合并。

如果仍有一个空牛棚连续列与我们的答案盖住的那个相同，我们同样使用上述的方法。会发现获得的新解与原解相同，那么不论我们选哪个，结果都将一样。

由此可见，如果我们合并的两块木板间距离最短，那么总能获得最优解。所以，在解题的每一步中，我们都只需要寻找两块距离最小的木板并合并它们。这样，我们获得的解必定最优。

结论：

如果有贪心性质存在，那么一定要采用！因为它容易编写，容易调试，速度极快，并且节约空间。几乎可以说，它是所有算法中最好的。但是应该注意，别陷入证明不正确贪心性质的泥塘中无法自拔，因为贪心算法的适用范围并不大，而且有一部分极难证明，若是没有把握，最好还是不要冒险，因为还有其他算法会比它要保险。

类似问题：

三值排序问题 [IOI 1996]

有一个由 N 个数值均为 1、2 或 3 的数构成的序列（N<= 1000），其值无序，现要求你用最少的交换次数将序列按升序顺序排列。

算法：排序后的序列分为三个部分：排序后应存储 1 的部分，排序后应存储 2 的部分和排序后应存储 3 的部分，贪心排序法应交换尽量多的交换后位置正确的（2，1）、（3，1）和（3，2）数对。当这些数对交换完毕后，再交换进行两次交换后位置正确的（1，2，3）三个数。

分析：很明显，每一次交换都可以改变两个数的位置，若经过一次交换以后，两个数的位置都由错误变为了正确，那么它必定最优。同时我们还可发现，经过两次交换后，我们可以随意改变 3 个数的位置。那么如果存在三个数恰好为 1，2 和 3，且位置都是错误的，那么进行两次交换使它们位置正确也必定最优。有由于该题具有最优子结构性质，我们的贪心算法成立。

货币系统 -- 一个反例［已删节］

奶牛王国刚刚独立，王国中的奶牛们要求设立一个货币系统，使得这个货币系统最好。现在告诉你一个货币系统所包含的货币面额种类（假设全为硬币）以及所需要找的钱的大小，请给出用该货币系统找出该钱数，并且要求硬币数尽量少。

算法：每次都选择面额不超过剩余钱数但却最大的一枚硬币。例如：有货币系统为{1，2，5，10}，要求找出 16，那么第一次找出 10，第二次找出 5，第三次找出 1，恰好为最优解。

错误分析：其实可以发现，这种算法并不是每一次都能构成最优解。反例如：货币系统{1，5，8，10}，同样找 16，贪心的结果是 10，5，1 三枚，但用两枚 8 的硬币才是最优解。因为这样，贪心的性质不成立，如此解题也是错的。

拓扑排序

给你一些物品的集合，然后给你一些这些物品的摆放顺序的约束，如"物品 A 应摆放在物品 B 前"，请给出一个这些物品的摆放方案，使得所有约束都可以得到满足。

算法：对于给定的物品创建一个有向图，A 到 B 的弧表示"物品 A 应摆放在物品 B 前”。以任意顺序对每个物品进行遍历。每当你找到一个物品，他的入度为 0，那么贪心地将它放到当前序列的末尾，删除它所有的出弧，然后对它的出弧指向的所有结点进行递归，用同样的算法。如果这个算法遍历了所有的物品，但却没有把所有的物品排序，那就意味着没有满足条件的解。