Design and Analysis of Algorithms – CSE 101
Greedy Algorithms : Design and Correctness

1 Search and Optimization Problems

The problems solved by greedy algorithms are mostly optimization problems. These problems can be put in the following format:

Instance What does the input look like?

Solution format What does the output look like?

Constraints What properties does the output have to have to count as a solution?

Objective function What quantity are we trying to maximize or minimize?

If there is any solution of the format that meets the constraints, we want to find one that is optimal for the objective function. There can be a huge number of optimal solutions, so we just need to output one optimal solution. In other words, if our algorithm produces AS and OS is another solution that also meets the constraints, and Obj is the objective function, we need to guarantee $Obj(AS) \geq Obj(OS)$ for a maximization problem, or $Obj(AS) \leq Obj(OS)$ for a minimization problem.

If there is no solution that meets the constraints, our algorithm should return "no solution" or similar message.

For example, in the spanning tree problem, the instance is an undirected graph with non-negative edge weights, and the solution format is a subset of the edges. The constraint is that the sub-graph formed by these edges is connected. The objective is to minimize the total weights of elements in the spanning tree. A correct minimum spanning tree algorithm for a connected graph should return such a tree, and there should not be a tree that connects the graph with smaller cost. If the graph is not connected, there is no possible solution, so our algorithm should return "False" or "No solution".

2 Coming up with greedy strategies

If we think of trying to search through all possible solutions, we usually end up with an exponential time exhaustive search algorithm. Instead, we try to break up the search into a series of easier decisions. In a greedy algorithm, we don't reconsider decisions once made, so each decision had better be "correct", i.e., each decision has to at least be consistent.

1. View each solution as making a series of *decisions*. Identify the *options* for each decision. The format for solutions is a big clue as to how to do this. For example, if the format is an array, for each index, we must choose the value of that array element. In the spanning tree example, we can view a solution as deciding the next edge to add $n-1$ times. There are $n-1$ decision points, and for each, the options are the remaining edges that connect two separate components.

2. Pick a single choice, and do a case analysis of the options. What options are possible at each choice? How do the constraints and previous choices limit the options? For example, if we have previously added edges that connect two nodes (indirectly), we will never choose to add an edge between them.

3. Find a conjecture for an "easy" decision, one where the case analysis suggests that one case is better than the other. In the spanning tree example, it always seems better to include the edge of least weight. If we don't include it, we'd have to later include some other edge to connect its endpoints, and that one would cost at least as much.

   One rule of thumb in coming up with conjectures is to weigh both the immediate cost or benefit of a decision with the opportunity costs or benefits making it entails. A typical type of problem is when we are selecting a subset of objects and cannot pick objects that "conflict" in some sense. Here, the immediate benefit is how much an object contributes directly to the objective function. But the opportunity cost is that we cannot pick any of the conflicting objects. Of course, we might not be able to pick all of the conflicting objects because they might also conflict with each other. So a more accurate opportunity cost is the maximum contribution of a subset of objects that conflict with our choice, but do not conflict with each other. Think about extreme examples for this subset.

Recursive vs. iterative

Once you've decided on a conjecture, your strategy can be thought of either as a recursive procedure or an iterative procedure.

If we think about it recursively , it looks something like:

$Greedy(Instance)$.

(a) Find the first greedy decision, g_1.

(b) Simplify the problem once you've chosen g_1, using the constraints, to get a smaller instance $Instance'$.

(c) $GS' := Greedy(Instance')$.

(d) Return $g_1 + GS'$, where $+$ represents a logical way to combine the two.

For example, $Kruskal(G)$.

(a) Find the smallest weight edge $e = \{u, v\}$.

(b) Let G' be the graph where we contract u and v to a single vertex, with all edges out of either u or v.

(c) $T' := Kruskal(G')$.

(d) Return $T' \cup \{e\}$.

However, implementing the algorithm iteratively is usually more efficient. $Greedy(Instance)$.

```
(a) Intialize an empty solution $GS$, $i := 1$..
(b) Until there are no more decisions to make do:
(c)     Find the next greedy decision $g_i$, $i++$, Add $g_i$ to $GS$.
(d) Return $GS$
```

Example: $Kruskal(G)$.

```
(a) Sort the edges by weight from smallest to largest.
(b) Initialize an empty set of edges $T$, and a pointer into the sorted list.
(c) Repeat until $n-1$ edges have been added:
(d)     Add the next edge in the list whose endpoints are not already connected to $T$.
(e) Return $T$.
```

These are really the same strategy, but for proving correctness, the recursive viewpoint is usually easier to work with, and for coming up with a fast implementation, the iterative viewpoint is usually better.

4. Prove your conjecture works. Usually, your greedy algorithm is wrong, even when it seems right. Most optimization problems cannot be solved by greedy algorithms; you need to re-consider possible decisions, and there is no simple rule that makes one decision clearly better than the others.. We'll look at several proof strategies in the next section. The important thing is that proving your conjecture is a way of logically forcing a skeptical person that your strategy works. Until you've proved your algorithm works, you shouldn't believe it works.

5. Find an efficient implementation. How do we want to preprocess the data items to make the next decision easy to find? How does making one decision change how we store the rest of the input? Think about data structures that can keep finding new decision points quickly. (See the data structures in algorithms guide for more discussion.)

3 Two example problems

We'll use two related example problems to illustrate the various proof techniques for showing greedy algorithms are correct.

Say you have a start-up company, and you need to get as much capital as possible before your company goes public. The plan is for your company to perform $k \geq 1$ projects and then go public. You have as input k, your initial capital C_0, and a list of n potential projects, $Proj_i$, each with a required capital $RC_i \geq 0$ and a profit for completing it, $Prof_i > 0$. Each project you do adds $Prof_i$ to your capital, and you can only do projects that have RC_i at most your current capital. In version 1, you can repeat projects any number of times; in version 2, you can only do each project once. In either case, your objective is to maximize your capital after k projects. In version 3, you can only do each project once, but there is no limit on how many projects you can do before going public.

For all of these, the obvious greedy strategy is to perform the task that you can perform (because you have enough capital, and in some versions, you have not already done it) that gives the maximum profit. The intuition is that this is both the greatest immediate benefit, and creates the best opportunities, since the more profit we gain, the greater our capital will be for future choices. So this is the rare type of problem where immediate benefits and future opportunities agree with each other, rather than being a trade-off.

4 The modify-the-solution method, format 1

The challenge in proving a greedy solution GS is optimal is we need to compare it with a totally unknown solution OS. (You can think of OS as standing for "other solution" or "optimal solution"). In the modify-the-solution method, we compare indirectly, by morphing OS into GS through a series of steps. In the first format, we think of GS as being generated recursively, and the morphing process as also being performed recursively in the same way. That allows us just to justify the first step of GS, and only worry about later steps when we inductively prove the recursion correct. By "justify the first move", here's what we mean: Let g_1 be the first greedy move. We want to show that by picking g_1, it is still possible for GS to be optimal. In other words, we want to show that there is an optimal solution that includes g_1. Think of OS as any optimal solution. If it already picks g_1, we have an optimal solution including g_1. If it doesn't, we need to rearrange it so that it does. In the process of rearranging, we must make sure it still meets the constraints, and that it is still optimal, ie., the rearranged solution OS' must be as good as OS.

This gives the following format for the first step lemma:

MSM Lemma:

Let g_1 be the first choice of the greedy algorithm. Let OS be any solution that meets all the constraints and does not choose g_1. Then there is a solution OS' that does choose g_1 , meets all the constraints, and and is at least as good as OS.

Many of the phrases need to be translated into the terms of the problem. g_1 is defined by the greedy algorithm. "Meets all the constraints" is defined by the constraints in the problem description. "Is at least as good" is defined by the objective function in the problem description.

The logical structure of this lemma also guides how the proof should go. The first two sentences tell you what ingrediants you have (g_1, OS) to work with and what you know about them. Since you need to show OS' exists, you need to use these ingrediants to construct OS'. Then the last part of that statement is what you need to show about OS'.

So the MTS proof goes through these steps:

1. State what you know: g_1 meets the condition for the first choice of the greedy algorithm, OS meets all of the constraints for the problem
2. Define OS', in terms of g_1 and OS, to include g_1. There might be multiple cases.

3. Show that OS' meets all of the constraints
4. Compare the objective functions of OS and OS', and see that OS' is the same or better.

If step 2 has multiple cases, you'll need to do steps 3 and 4 for each case.

Examples:

All versions, statement:

Let $Proj_1$ be the project with maximum profit among those whose required capital is at most C_0. Let OS be any schedule that does not perform $Prof_1$ first. Then there is a schedule OS' that performs $Proj_1$ first where we obtain at least as much total capital in OS' as we do in OS.

Proof: In version 1, we can just define OS' to be identical to OS in later projects, but perform $Proj_1$ as the first project. Say that OS does $Proj_i$ first. Then we know $CR_i \leq C_0$ or it could not be performed first in OS. $Prof_1 \geq Prof_i$, since the $Proj_1$ is defined to be the poject with maximum profit among those whose required capital is at most C_0. Thus, we will have at least as much accumulated capital in each step in OS' as we do in OS. In particular, we will have sufficient capital in OS' to perform each project, since we did in OS, and we have at least as much in OS'. Finally, also at the end, we will have at least as much total capital in OS' as we do in OS.

Versions 2 and 3: If OS never performs $Proj_1$, we can use the same argument as above. But if OS performs $Proj_1$ as the j'th project, we cannot just schedule $Proj_1$ first, since then we will be doing the same project twice. So we need an alternative project for the jth project. But we've just stopped doing $Proj_i$. So in this case, we switch the order of $Proj_1$ and $Proj_1$. Namely, OS' does the same projects as OS, but does $Proj_1$ first, and $Proj_i$ as the j'th project. As before, $Proj_1$ must have at least as much profit as $Proj_i$. So when we switch, we'll have more capital in OS' between projects 1 and j, so all intermediate projects will meet the capital requirement. After project j, OS and OS' will have finished the same set of projects, and so will have equal amounts of capital, so capital requirements will be met after project j also. Thus, OS' meets all constraints. Also, at the end, OS and OS' have the same amount of capital.

4.1 Induction

Since this justifies just the first move, we'll need to then use this lemma in an induction proof, to say that other moves are handled recursively in the same manner.

Induction format: For any instance I of size N, $GS(I)$ is an optimal solution.

By strong induction. Base case: $N = 0, N = 1$. Usually, this is trivial, since there are no choices or only one choice.

Assume for every instance I_2 of size $0 \leq n \leq N-1$, $GS(I_2)$ is optimal for I_2. Let I be an instance of size N. Then $GS(I) = g_1 + GS(I_2)$ for some smaller I_2, thinking about the greedy algorithm recursively. By the MTS lemma, there is an optimal solution OS' that also includes g_1. $OS' = g_1 + OS_2$ for some other solution OS_2 of instance I_2. Then by the induction hypothesis, $GS(I_2)$ is at least as good as OS_2. Then $GS(I)$ is at least as good as OS', since both combine the sub-problems with the same first move. Since OS' was optimal, $GS(I)$ is also optimal.

Example: In version 1, the only parameter decreasing is k, the number of allowed projects. So our induction is on k.

Version 1: We prove by strong induction that the greedy solution is optimal for every instance by induction on k, the allowed number of projects. If $k = 0$, there is only the empty schedule, which gains final capital C_0.

Assume the greedy strategy is optimal for any instance with $0 \leq K \leq k-1$ allowed projects. Consider an instance with k allowed projects. Let $Proj_1$ be the project chosen by the greedy algorithm. Consider the instance with initial capital $C_0 + Prof_1$ and $k-1$ allowed projects. The greedy algorithm can be viewed as first performing $Proj_1$, and then recursively solving the above instance. We showed in the MTS solution that there is an optimal solution OS' that does $Proj_1$ first. Then the rest of the solution OS' is also a solution to the above instance with $k-1$ projects. Note that since we added $Prof_1$ to initial capital in the smaller instance, the greedy algorithm on the k step instance gets exactly the same final capital as it does on the $k-1$ step instance, as does OS' . By the induction hypothesis, the greedy solution to that smaller instance has at least as much final capital as does the remaining part of OS'. Since these are the same quantities as the final capital on the whole instance, the greedy solution to the k step instance gets at least as much capital as does OS', which is optimal. So the greedy solution is optimal for schedules with k projects.

Thus, by strong induction, the greedy schedule is optimal for any number of projects.

Version 2 would be identical, except we would remove $Proj_1$ from the smaller instance. Note that we could do the induction on either k or n for version 2, since both decrease.

Version 3 is the same as version 2, except there is no k, so we would have to do the induction on n.

5 Modify-the-solution, iterative format

Some students find reasoning about the recursive version of the greedy solution confusing. Here's a different format but equivalent logic that avoids recursion. In this version, we carry out the whole sequence of transformations, that starts with an optimal solution OS and morphs it into the greedy solution. Instead of doing the modfication lemma and then using it in an induction argument, we combine the induction and modification.

Format

IterMTS : Let $g_1, , ..g_T$ be the decisions made (in order) by the greedy strategy. For each $0 \leq i \leq T$, there is an optimal solution OS_i that includes choices $g_1, ...g_i$.

We prove this by induction on i. For $i = 0$ we can let OS_0 be any optimal solution, since it doesn't have to agree with any greedy decisions.

The induction step is similar to the first MTS format, except we have $g_1, ..g_i$ to cope with not just g_1. Assume there is an optimal solution OS_{i-1} that includes $g_1, ..g_{i-1}$. If it also includes g_i, we set $OS_i = OS_{i-1}$. Otherwise, we need to

1. Define OS_i in a way that leaves $g_1, , g_{i-1}$ in place, but changes the i'th move of OS_{i-1} to g_i
2. Prove that OS_i meets the constraints
3. Compare the objective functions of OS_i and OS_{i-1}, showing that OS_i is just as good, so also optimal.

At the end, we conclude that $OS_T = GS$ is optimal, since OS_T includes all of the greedy decisions.

5.1 Example

I'll just show this for Version 2, the most complex.

Let $Proj_1, ..Proj_k$ be the projects that the greedy schedule performs, in that order.

We prove by induction on i that there is an optimal schedule OS_i that performs $Proj_1, ..Proj_i$ as its first i projects, in that order.

For $i = 0$, we can let OS_0 be any optimal schedule.

Assume OS_i is an optimal schedule that performs $Proj_1, ..Proj_i$ as its first i jobs. Then at the start of the $i+1$'st job, OS_i and GS have the same capital, having performed the same jobs. If OS_i performs $Proj_{i+1}$ next, we can let $OS_{i+1} = OS_i$. Otherwise, say it performs $Proj_l$ next.

If OS_i peforms $Proj_{i+1}$ as the j'th project for $j > i + 1$, we can create OS_{i+1} by swapping $Proj_l$ to the j'th project and performing $Proj_{i+1}$ now. $Prof_{i+1} \geq Prof_l$, because both were available projects when the greedy schedule chose $Proj_{i+1}$. So between the $i+1$'st project and the j'th, OS_{i+1} will always have at least as much capital as OS_i, and so intermediate projects will have sufficient capital in OS_i. We had enough capital to do project l as the $i + 1$'st and we will have more capital than that when we do it as project j. Finally, after project j, OS_i and OS_{i+1} have done the same jobs, so the capital will be equal. Thus, all capital constraints are met in OS_{i+1}, and its final capital is equal to that of OS_{i+1}.

If OS_i never performs $Proj_{i+1}$, we can replace the $i+1$'st project in OS_i with $Proj_{i+1}$ and keep all others the same to define OS_{i+1}. The capital will only have increased at each step, so all projects will meet capital constraints, and we will have at least as much capital at the end, so OS_{i+1} is also optimal.

Thus, in either case, we have created an optimal schedule OS_{i+1} containing the first $i + 1$ greedy projects. By induction on i, there is such an OS_i for all i.

In particular, for $i = k$, OS_k is optimal and is identical to GS. Therefore GS is optimal.

6 Greedy stays ahead

A less general but sometimes more intutive method is greedy-stays-ahead. Here, we line up the steps of the greedy solution and the optimal solution, and show that by some measure, GS is "ahead" of OS at all times. This involves an induction on the number of decisions made, because we need "at all times" as an invariant. This method is not as general as MTS. For the above problems , I can prove version 1 optimal using GSA, but I don't know GSA proofs for the other versions.

The steps are:

1. Define the progress measure
2. Define some way of ordering the choices of OS to line up with GS

3. Prove by induction on i, the number of choices, that after i steps, GS has made at least as much progress as OS.

4. Use this at the last step of GS to show that GS is at least as good as OS

6.1 Example

Version 1:

Let the GS perform $Proj_1.,,.Proj_k$, and let an optimal solution OS perform $Proj'_1,..Proj'_k$. We claim that after the i'th project, GS has at least as much capital as OS.

For $i = 0$, this is true, because both have C_0 capital.

Assume GS has capital C_i after i projects, OS has C'_i capital after i projectsi, and $C_i \geq C'_i$. Then the next project in GS, $Proj_{i+1}$ has the greatest profit of any project with required capital at most C_i. Also, $Proj'_{i+1}$ must have required capital at most $C'_i \leq C_i$, so it is some project with required capital at most C_i. Thus, the profit from $Proj_{i+1}$ is at least that of $Proj' + i + 1$. Thus, GS will have at least as much capital after $i + 1$ projects as does OS, since it started with at least as much and gained at least as much.

By induction, then, this holds for all $0 \leq i \leq k$. In particular, for $i = k$, GS ends with at least as much capital as optimal solution OS, and so is optimal.

7 Achieves the bound method

In the achieves the bound method, we identify an obstacle n a problem instance that would prevent an algorithm from doing better. For example, "If there are K events going on at the same time, then we need at least K rooms to hold them in." Here, the obstacle is the K events, at the same time, and the obstacle gives a lower bound of K on the cost of a solution. This obstacle will allow us to indirectly compare GS to OS. The obstacle is a bound on OS, and so to prove GS is at least as good as OS, we need to show that GS achieves this bound.

1. Define the bound

2. Show that for any solution OS, its objective function cannot be better than this bound

3. Show that the objective function of the greedy solution equals this bound.

4. Conclude that greedy is at least as good as any other solution.

7.1 Example:

Again, I don't know AtB proofs for all versions, only version 3. If the greedy algorithm doesn't do all projects in version 3, why does it stop? If its capital at the end is C, there can't be any projects that it didn't do with $RC \leq C$, or it would continue. It can't have done any projects with $RC > C$, because it never had capital greater than C. So it must have done exactly the projects with $RC \leq C$, and gotten total capital C.

Define a *bounding value* as an amount of capital C so that C_0 plus the profits of all projects with $RC \leq C$ is C.

As we said above, if the greedy solution doesn't do all projects, its final capital is a bounding value.

Let C be a bounding value, and OS any solution. We claim that OS can never have capital $> C$. It starts with capital $C_0 \leq C$, and until the first time when it has capital $> C$, can only perform projects with required capital at most C. It can only perform such projects once, so while it performs such projects, it can only have capital at most C_0 plus the sum of all the profits for such projects, which is C. So there can never be a first time when OS has capital $> C$. so it never does.

IF GS performs all projects, it is optimal. Otherwise, the amount of capital GS gets is a bounding value C, and so any other solution OS gets at most C total. Thus, in either case GS is optimal.