CS 577: Introduction to Algorithms Fall 2020

How to Develop a Greedy Algorithm

Instructor: Dieter van Melkebeek TA: Drew Morgan and Ryan Moreno

This handout is meant to provide a basic structure to the problem solving and write-up steps for the greedy paradigm. Please note that these are not required patterns. Any solution that is clear, correct, sufficiently efficient, and analyzed (correctness and complexity) will get full credit.

Overview and Approach

Greedy algorithms only work in specific scenarios in which it's possible to know the optimal next decision for you algorithm based only on previous decisions. This is in contrast to DP algorithms in which you test each possible decision recursively and then select the optimal decision. Greedy algorithms are simple in that they consist of a simple greedy criteria without considering future decisions. However, the proof of correctness is often difficult because you have to prove *why* your greedy criteria works and you don't have to test every possible decision.

Suggested Format

1. **Algorithm Description**: ∼1-2 sentences for the idea plus a bit more for implementation as necessary. The greedy algorithm should be based on a simple heuristic. You should repeat the same type of decision at each step. For example, we've seen "pick the interval with the earliest finish time," "choose the largest item that fits," and "when deciding the vertex that came before v on a shortest path from s to v, choose the vertex u that minimizes the path from the start to u plus the length from u to v." As you come up with possible greedy criteria, you will need to decide which is correct. See the counterexample section for tips on disqualifying incorrect greedy criteria.

2. **Correctness**: Prove validity and optimality. For validity, if it's obvious that your algorithm maintains validity you can leave this out (see example solutions below). For optimality, you've learned two tools in this course: greedy stays ahead arguments and exchange arguments. Both compare some other valid solution S to the greedy solution G. It is often helpful to reorder S in the greedy order, making S and G easier to compare.

   (a) **Greedy Stays Ahead**
   - Clearly state the claim you will prove by induction. This claim will be something along the lines of "at every step, the greedy solution G stays ahead of every other valid solution S." You will need to decide what dictates the "steps" and what your quality measure is for "staying ahead."
   - Give a (usually inductive) proof of your claim.
   - Explain how your claim proves the optimality of the greedy algorithm.

   (b) **Exchange Argument**

- Show that if some valid and optimal solution S is not the same as the greedy solution G, then there is a way to change a decision made in S such that S gets closer to G. This is called an exchange argument because you will replace a decision made in S with a decision made in G. (Formally, you can see this as an inductive proof. Assuming S agrees with the first k decisions that G made, you show that you can exchange one of S's decisions with G's $(k+1)$st decision. Now S agrees with the first $k+1$ decisions that G made.)
- Prove that this exchange retains the validity and optimality of S. (Sometimes an exchange will make S strictly more optimal. In this case, S was not originally optimal which is a contradiction.)

3. **Complexity Analysis** ∼1-2 sentences

Finding Counterexamples

Sometimes we will explicitly ask you to develop a counterexample to an incorrect greedy algorithm. Other times, while coming up with your own greedy algorithm you may have multiple options you're considering. You will want to look for counterexamples to these potential greedy algorithms in order to decide which is correct. Here are some suggestions for finding counterexamples:

- A general rule is to start as small and simple as possible. See if the algorithm works on inputs of size $1, 2$, etc. Often an incorrect algorithm will have counterexamples at the first input size where it doesn't trivially work.

- In greedy algorithms, it helps to try to find where there are "ties" in the greedy rule such that breaking the tie one way breaks the algorithm. Usually you can adjust your example (e.g., change the numbers by very small amounts) so that the tie has to be broken in the counterexample's favor.

  As an example, consider unweighted interval scheduling. Sorting the intervals in order of non-decreasing start time doesn't work. To see this, we can start with two intervals: $[1, 2)$ and $[1, 10)$, which are tied according to the greedy criteria. We can pretend greed will break the tie in whatever way is convenient for us, so let's say it considers $[1, 10)$ first, since that's the bigger interval. Thus greed includes that interval in the solution, and skips the $[1, 2)$ interval. We can also prepare an interval $[2, 4)$ as part of the input: it conflicts with $[1, 10)$, but not with $[1, 2)$. So greed will output a solution with just one interval, while the optimal answer has two. We can force the tie to break in our favor by changing the $[1, 10)$ interval to $[0.999, 10)$.

- A general strategy is to infer what the algorithm is "trying" to do at a conceptual level. Then pretend to develop a proof of correctness and see where it gets stuck. Many times finding where the proof breaks down gives insight on how to build the counterexample. (This requires comfort and confidence with proving correctness, but is probably the most effective strategy overall.)

- For recursive algorithms (DP, D&C, and even greedy when formulated that way) suppose that the recursive calls really do work (otherwise the algorithm is wrong anyway). Then see if you can find an example where the body of the recursive function breaks down. Note this

is just a specialization of the previous bullet point. See Piazza post @364 for an example of this.

Example 1: Greedy Stays Ahead – HW05#4b

Algorithm Description Given the start and finish times of each class, $(s_i, f_i]$ for $i = 1 \dots n$, order these intervals by finish time. Schedule a visit for the earliest finish time of an uncovered class and repeat until all classes are covered.

Correctness The greedy solution is valid because we schedule visits until every class is covered. [Note: this is obvious enough you could leave it out without losing points]

We will show that the greedy solution G is optimal by showing that it stays ahead of any other valid solution S. First we will reorder S so that the visit times are listed in increasing order. We claim that **the kth visit in G is at least as late as the kth visit of any S.** Denote this as $g_k \geq s_k$. If a schedule has less than k visits, then we will say that the kth visit is scheduled at time ∞. We'll also deal with the $k = 0$ case here by saying that if G doesn't use any visits then there must not have been any classes, so having 0 visits is optimal.

Proof that $\forall k : g_k \geq s_k$:

- *Base case:* $k = 1$. The first visit for k is scheduled as the finish time for the first uncovered class, so $g_1 = f_1$. Since S is a valid schedule and s_1 is the earliest visit time in S, it must be the case that $s_1 \leq f_1$ or else the first class would be uncovered. So $g_1 \geq s_1$.
- *Inductive Hypothesis:* $g_k \geq s_k$
- *Inductive Step:* Want to show $g_{k+1} \geq s_{k+1}$

  Note that the classes in G that are still uncovered after the first k visits are exactly the classes with start times after g_k (because the solutions are in increasing order). The same holds for S. By the inductive hypothesis, we know $g_k \geq s_k$, so every class that's uncovered in G is also uncovered in S. Let class c be the first class that's uncovered by the first k visits in G. We know class c is also uncovered in S. By the greedy algorithm, $g_{k+1} = f_c$. Since class c is uncovered in S and we know S is valid, $s_{k+1} \leq f_c$. So $g_{k+1} \geq s_{k+1}$.

Note that this claim proves that G is optimal: If S schedules less than k visits, then we know that G also scheduled less than k visits (because $g_k \geq s_k = \infty$).

Running Time Analysis We sort the classes by finish time, which takes $O(n \log n)$. Then we make a single loop through the classes, scheduling a visit whenever we find an uncovered class (we can do this by keeping track of the time of the latest visit). This takes $O(n)$. So our overall runtime is $O(n \log n)$.

Example 2: Exchange Argument – HW06#4

Algorithm Description Let m_i and e_i for $i = 1, \ldots, n$ be the lengths of the morning and evening shifts, respectively. We order the m_i so that $m_1 \leq m_2 \leq \ldots \leq m_n$, order the e_i so that $e_1 \geq e_2 \geq \ldots \geq e_n$, and pair m_1 with e_1, m_2 with e_2, etc.

Correctness By exchange argument.
Fix any pairing P, and suppose $m_1 \leq \ldots \leq m_n$ are paired with (in order) $e_1, \ldots, e_n$.
P not greedy $\implies$ there exist adjacent positions $i < j$ such that $e_i < e_j$.
Let P' be P except pair $m_i \leftrightarrow e_j$ and $m_j \leftrightarrow e_i$.
We claim (proof below) that P' requires no more overtime than P.
Since this holds for every P not equal to the greedy strategy, and each swap reduces # of (i, j) with $e_i > e_j$, every solution can be transformed into the greedy solution with no increase in overtime.
$\implies$ Greedy is therefore optimal.

Proof that P' has no more overtime than P:
The overtime for all drivers in P equals that in P', except the drivers working m_i, m_j, e_i, and e_j.
In P, their extra overtime is

$$\max(m_i + e_i - d, 0) + \max(m_j + e_j - d, 0) = \max(m_i + e_i, d) + \max(m_j + e_j, d) - 2d$$

while in P' it is

$$\max(m_i + e_j - d, 0) + \max(m_j + e_i - d, 0) = \max(m_i + e_j, d) + \max(m_j + e_i, d) - 2d$$

so it suffices to prove

$$\max(m_i + e_i, d) + \max(m_j + e_j, d) \geq \max(m_i + e_j, d) + \max(m_j + e_i, d). \qquad (\star)$$

Case 1: $m_j + e_j \leq d$
Since $m_i \leq m_j$ and $e_i < e_j$, then also $m_i + e_i$, $m_i + e_j$, and $m_j + e_i$ are at most d.
$(\star)$ becomes $2d \geq 2d$. ✓

Case 2: $m_i + e_i > d$
Since $m_i \leq m_j$ and $e_i < e_j$, then also $m_i + e_j$, $m_j + e_i$, and $m_j + e_j$ are bigger than d.
$(\star)$ becomes $m_i + e_i + m_j + e_j \geq m_i + e_j + m_j + e_i$. ✓

Case 3: Otherwise
LHS $= d + m_j + e_j$
RHS $= \max(m_i + e_j + m_j + e_i, m_i + e_j + d, d + m_j + e_i, d + d)$

$$d + m_j + e_j \geq \begin{cases} m_i + e_j + m_j + e_i & \text{since } m_i + e_i \leq d \\ m_i + e_j + d & \text{since } m_j \geq m_i \\ d + m_j + e_i & \text{since } e_j \geq e_i \\ d + d & \text{since } m_j + e_j \geq d \end{cases}$$

so LHS $\geq$ RHS. ✓

Running Time Analysis The algorithm involves sorting two arrays of length n, and then pairing them together in order. The sorting takes $O(n \log n)$ time, and the pairing is $O(n)$, leading to $O(n \log n)$ overall.