Section 3: Greedy Algorithms

In this section, we're going to walk step-by-step through good problem-solving strategies applied to one algorithm design question. There are also a couple of problems at the end we won't have time to cover.

1. Line covering

Your new towing company wants to be prepared to help along the highway during the next snowstorm. You have a list of integers $t_1, t_2, \ldots, t_n$ in increasing order, representing mile markers on the highway where you think it is likely someone will need a tow (entrances/exits, merges, rest stops, etc.). To ensure you can help quickly, you want to place your tow trucks so that from every marker, at least one truck is at most 3 miles away. Find a minimum length list of sites where you can place tow trucks to satisfy the requirement, encoded as a list of integers $a_1, a_2, \ldots, a_m$ in increasing order. Note that the sites that you pick need not be a subset of the marked locations.

1.1. Getting started

When reading a long word problem, it is useful to summarize it. A common way is:

Input: …
Expected output: …

- mathematical definitions of any special words used above

(a) Write a summary of the above problem.

After summarizing the problem, you'll want to consider which big category of techniques you want to use.

- Does the problem remind you of an algorithm you've seen in class, like stable matching, graph algorithms, etc.? If so, you can try a reduction or modifying the known algorithm, like you've done on the past few homework sets.
- If not, you'll want to try techniques for developing algorithms from scratch. **Greedy algorithms** are the first category we're learning about. In the coming weeks, we'll also learn the techniques of divide and conquer and dynamic programming.

For today, since this problem doesn't sound like anything we've seen in class so far, we'll try to use a greedy technique!

1.2. Generating ideas

When developing an algorithm from scratch, it can be difficult to come up with ideas to get started, but there is a common method.

- First, **solve many examples** by hand. In the beginning, don't worry about the general strategy. But as you start to try larger examples, keep the techniques you've learned like greedy methods (and later dynamic programming, etc.) in mind. You will start to see **patterns** after enough examples to give you an idea.
- Your first idea will probably be wrong. Whenever you have an idea, you should **ask yourself**, "Can I break my strategy with a nasty example? Does my strategy ever waste time? Can I optimize it?" If it turns out that your strategy is bad, go back to examples to see how to modify it or to see what other ideas might work.
- After several cycles of this process, you will likely have an idea that you believe to be correct and efficient. At this point, you can exit the idea generating process and begin writing.

For greedy algorithms in particular, keep in mind that your strategies should meet the following criteria:

- Follows a rule to keep picking something
- Doesn't consider the future
- Doesn't go back to fix things

(Of course, considering the future or backtracking to fix mistakes are important things that are sometimes necessary, but the point of greedy algorithms is that very often, those more advanced ideas aren't needed. Check whether or not easy greedy rules first, before trying other techniques that we'll learn later in class.)

Coming up with many greedy ideas should be easy. But finding the correct greedy idea will usually require trial and error or insight, so don't be discouraged.

(b) We will practice the idea generating process.

(i) Solve these examples by hand. Don't worry too much about greedy strategies yet.

1, 2, 4, 10, 12

0, 1, 3, 5, 7, 8, 13, 14

(ii) Suppose you came up with the greedy idea, "Put a truck on the first uncovered marker." Check that this idea works on the above examples. Then, try to break this idea by coming up with an example where it doesn't work.

(iii) Come up with a new greedy idea that solves your new example. Does the idea work? If not, continue the process until you have a working idea.

1.3. Writing up your idea

Once you have an efficient, working idea, you should:

- Translate the idea into **pseudocode**. Recall that pseudocode is of form of writing more precise than English, but easier to understand than code. There are no hard rules, but see the handout from last week for common styles.
- **Prove** the pseudocode correct.
- Finally, write up the **running time** analysis.

(c) Write the pseudocode for the solution.

As a refresher, for the proof, always show **validity**, **termination**, and **correctness**.

- Correctness always means, "My algorithm's output matches the problem summary's expected output."
- For greedy algorithms, the expected output is always something about an optimal solution. So, there are two things to prove:
  - "My output is a valid solution.": In today's case, that means "The list $a_1, \ldots, a_m$ is in increasing order and covers all markers."
  - "My output is optimal.": In today's case, that means "All other valid solutions use at least m trucks."

Lastly, the optimality proofs of greedy algorithms also tend to have a consistent structure that you can use to help you. There are a few types:

- "Greedy stays ahead": For all other solutions, show by induction that at every step, your solution is at least as good.
- "Exchange argument": For all other solutions that differ from yours, show how to change a part of the other solution, so that the quality improves or stays the same (but never decreases).
- "Structural argument": (less common) Find a "hard subset" of the input that immediately implies why other solutions must also be as bad as yours (or worse).

(d) Write a proof that your pseudocode is correct.

(i) **Validity**:

(ii) **Termination**:

(iii) **"The output is in increasing order."**:

(iv) **"The output covers all markers."**:

(v) **"All other valid solutions use at least m trucks."**:

(e) Analyze and prove the running time with big-O in a few sentences.

The following problems will not be covered in section, but may be useful to think about.
We recommend trying them by yourself first. Solutions will be posted in the evening.

2. Minimizing covers again

You have a set, $\mathcal{X}$, of (possibly overlapping) closed intervals of $\mathbb{R}$. (The closed intervals of $\mathbb{R}$ are the sets commonly denoted $[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$.) You wish to choose a subset $\mathcal{Y}$ of the intervals to cover the full set. Here, cover means the union of all intervals in $\mathcal{X}$ is equal to the union of all intervals in $\mathcal{Y}$. Describe (and prove correct) an algorithm which gives you a cover with the fewest intervals.

3. Art commissions

You've just started a new one-person art company. You've convinced n of your friends to each put $\$c$ of their current money into a bank account, which will eventually be withdrawn when they commission you to make art, supporting your dreams. It takes you one month to finish a commissioned piece (you are only working in your limited free-time). At the beginning of every month, one of your friends will withdraw the entire value in their account to pay you to make their artwork.

The bank accounts all earn small (and varying) rates of interest. Friend i earns interest at the rate of r_i, compounding monthly. That is, the amount in their bank account is r_i times what it was at the start of the last month (until they withdraw their money, and $r_i > 1$). To reiterate, your friends decide to pay you both the original $\$c$ and all of the interest earned at the time you start their commission.

Describe an ordering to take the commissions that will maximize the amount you are paid (you may assume you know the r_i for each of your friends).