Section 3: Greedy Algorithms

In this section, we're going to walk slowly, step-by-step through good problem-solving strategies applied to one algorithm design question. You might see a correct algorithm right away...it will still be worth it to practice the intermediate problem solving steps (because someday you *won't* see a correct algorithm right away).

1. Line Covering

Your new tow-truck company wants to be prepared to help along the highway during the next snowstorm. You have a list of n doubles, 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 one is at most 3 miles from **every** marked location. Find the locations which will allow you to place the minimal number of trucks while covering every marked location.

More formally, you will be given an array $A[]$, containing n `doubles` (in increasing order), representing the locations to cover.

Your task is to produce a list `sites` containing as few `doubles` as possible, such that for all i from 1 to n there is a j such that $|A[i] - \texttt{sites}[j]| \leq 3$. Note that the `sites` produced by your algorithm do not have to be at the same locations given by the marked locations.

1.1. Read and Understand the Problem

You can't solve a problem if you don't know what you're supposed to do! Read through the problem *slowly*. Remember that problems are written in mathematical English, which is likely to be much more dense than, say, a paragraph from a novel. You'll have to read more slowly (this advice still applies for our ridiculous word problems).

As you're reading, underline anything you don't understand. Rereading the problem a few times can often help (it's easier to understand details once you have the big picture in your brain).

We recommend asking yourself these questions to ensure you've understood the problem.

- Are there any technical terms in the problem you don't understand?
- What is the input type? (Array? Graph? Integer? Something else?)
- What is your return type? (Integer? List?)
- Are there any words that look like normal words, but are secretly technical terms (like "subsequence" or "list")? These words sometimes subtly add restrictions to the problem and can be easily missed.

For Today: Read the problem above and answer each of the four questions.

1.2. Generate Examples

Your first goal with the examples is to make sure you *really* understood the problem. You should generate two or three sample instances and the correct associated outputs. If you're working with others, these instances help make sure you've all interpreted the problem the same way. You should not think of these examples as debugging examples – `null` or the empty list is **not** a good example for this step. You can worry about edge cases at the end, once you have the main algorithm idea. You should be focused on the "normal" (not edge) case.

Your second goal is to get better intuition on the problem. You'll have to find the right answer to these instances. In doing so you might notice some patterns that will help you later.

For Today: Generate two examples with their associated outputs. Put some effort into these! The more different from each other they are, the more likely you are to catch mistakes later.

1.3. Come up with a Baseline

In a time-constrained setting (like a technical interview or an exam) you often want a "baseline" algorithm. An algorithm that you can implement and will give you the right answer, even if it might be slow. We're going to skip this step today, but you'll see it in future examples and in lectures.

For Today: Go to the next section

1.4. Brainstorm and analyze possible algorithms

It sometimes helps to ask "what kind of algorithm could I design?" This week the answer is going to be "a greedy algorithm" because we're learning about greedy algorithms. But by the end of the quarter you'll have a list of possible techniques.

- Does this problem remind me of any algorithms from class? What technique did we use there?
- Do I see a modeling opportunity (say a graph we could run an algorithm on, or a stable matching instance we could write)?
- Is there a way to improve the baseline algorithm (if you have one) to something faster?

For Today: You should use a greedy algorithm. Come up with at least two greedy ideas (which may or may not work). Run each of your ideas through the examples

1.5. Write an algorithm

Now we want to narrow down to our actual algorithm and get it working. From your two ideas, run them against your two example instances. Do they both work? Generate a few more instances and narrow down to just one possibility. If you eliminate both of your ideas, go back to the last step and generate a new rule; if you can't eliminate one of the two after multiple examples, try specifically to create an instance where they should behave differently (and if you still can't make them behave differently, just pick one to try).

As part of this process, you may want to jot-down some pseudocode, just to be sure you know exactly what your algorithm is.

Once you've got code you think might work, you need to convince yourself it really does work. For today, that's going to mean a proof.

For Today: Write the pseudocode for your algorithm.

1.6. Show your algorithm is correct

It doesn't matter if you write code if it doesn't work! Write

For Today: Write a proof of correctness.

1.7. Optimize and Analyze the running time

Now it's time to see if our algorithm is as efficient as possible. Flesh out any pseudocode you've written with enough detail to analyze the running time (do you need particular data structures?). Write and justify the big-O running time. Can you make your code more efficient? Can you give a reason why you shouldn't expect the code to be any faster?

For Today: Write the big-$\mathcal{O}$ of your code and justify the running time with a few sentences.

Extra Problems

2. Another Greedy Algorithm

You have a set, $\mathcal{X}$, of (possibly overlapping) intervals, which are (contiguous) subsets of $\mathbb{R}$. You wish to choose a subset $\mathcal{Y}$ of the intervals to cover the full set. Here, cover means for all $x \in \mathbb{R}$ if there is an $X \in \mathcal{X}$ such that $x \in X$ then there is a $Y \in \mathcal{Y}$ such that $x \in Y$.

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 to supporting your dreams by commissioning you to make art. It takes you one month to finish a commissioned piece (you are only working in your limited free-time). One of your friends will pay you at the beginning of the month to make their artwork.

While waiting for you to start their commission, your friends are going to place their money into bank accounts, which earn small (and varying!) rates of interest. Friend i earns interest at the rate of r_i, compounding monthly. I.e., 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; $r_i > 1$). Your friends generously decide to pay you both the principal and 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).