Section 3: Solutions

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.

Solution:

- "cover" means "place a truck at most 3 miles from the location."
- The input type is a `double[]`.
- The return type is a `double[]`.
- list might be a technical term (though it doesn't make a difference if it is).

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.

Solution:

There are multiple examples and outputs you can pick for the best solution, here are some we thought of:

- Input: [0,1,2,3,4,5,6,7] Output: [3,7]
- Input: [0,6,7,8,9,13] Output: [3,10]
- Input: [0,1,2,3, 100,101,102,103] Output: [1.5,101.5]

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

Solution:

Here are some ideas we thought of (you might have thought of others)

- Choose the `double` which will cover the most remaining elements of A.
- For $A[i]$ being the leftmost (remaining, uncovered) element, add $A[i] + 3$ to `sites` (the right-most spot which will cover the left-most element).
- Start with $A[1] + 3$, and thereafter if your previous point was p, choose $p + 6$
- Pick the left-most (remaining, uncovered) $A[i]$.

Solution:

With our examples, we can eliminate many of these options.
[0,6,7,8,9,13]. The solution $[3, 10]$ covers all elements. This example eliminates the first option, which will find $[7.5, 0, 13]$. This example also eliminates the last option, which will produce $[0, 6, 13]$.
[0,1,2,3, 100,101,102,103] The solution $[1.5, 101.5]$ covers all elements. This example eliminates the third option, which will be $[3, 9, 15, ..., 99, 105]$.

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.

Solution:

```
function PLACEMENTS(A[1..n])
    uncoveredIndex ← 1
    sites ← empty list
    while uncoveredIndex ≤ n do
        newSite ← A[uncoveredIndex] + 3
        append newSite to sites.
        while uncoveredIndex ≤ n and |A[uncoveredIndex] − newSite| ≤ 3 do
            uncoveredIndex++
    return sites
```

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.

Solution:

We prove the claim using the "greedy stays ahead" format.
Let OPT be an optimal solution (sorted in increasing order), and let ALG be the solution generated by our algorithm (similarly sorted). We will show that for all i, the first i elements of OPT cover at most as many elements of A as the first i elements of ALG
We proceed by induction on n.
Base Case: $n = 1$
The left-most chosen site must cover the left-most element of A. ALG chooses the rightmost point that still allows for $A[1]$ to be covered, so it must cover every element that OPT's first site does.
IH: Suppose that the claim holds for $i = 1, ..., k$.
IS: Let $A[\ell]$ be the left-most element not covered by the first k entries of ALG. By IH, $A[\ell]$ is also not covered by the first k entries of OPT.
Note that uncoveredIndex will be ℓ at this point of the algorithm; it begins at index 1 and increases to the left-most-uncovered index with the inner-most while-loop at each step. Thus the next site we choose is $A[\ell]+3$. This is the rightmost point which can still cover $A[\ell]$ (any further right point is too far away to cover it), since $A[\ell]$ is uncovered by OPT as well, the next element of OPT cannot be greater than $A[\ell] + 3$ (if it were, by the sorted ordering, $A[\ell]$ would remain uncovered). Thus we have that ALG continues to cover at least as many elements as OPT.

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.

Solution:

The loop runs in $\mathcal{O}(n)$ time. It might look worse at first glance, but it's not! Every iteration of the inner loop increases `uncoveredIndex`, so those lines run at most n times **total** (across all iterations of the outer-loop). Similarly, every line in the outer-loop executes at most n times, since every new site covers at least one element of A and therefore increases `uncoveredIndex`. Each step can be implemented in $\mathcal{O}(1)$ time, so the total time is $\mathcal{O}(n)$.

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.

Solution:

Key Idea Taking the earliest-start-time (breaking ties with the longest interval) will guarantee all points are covered and will "stay ahead" of any other solution.
Algorithm

function IntervalCovering(()$\mathcal{X}$)
Sort $\mathcal{X}$ by start time
$\mathcal{Y} \leftarrow \varnothing$
while $\mathcal{X} \neq \varnothing$ **do**
Let I be element remaining in $\mathcal{X}$ with earliest start time, breaking ties by taking latest end time.
$\mathcal{Y} \leftarrow \mathcal{Y} \cup \{I\}$
Delete all elements of $\mathcal{X}$ that are fully covered by $\mathcal{Y}$
For any remaining elements of $\mathcal{X}$ that start before I ends, set their start time equal to I's end time

Correctness Let $\mathsf{ALG} = a_1, a_2, ..., a_k$ be the list of intervals found by the algorithm, and let $\mathsf{OPT} = o_1, ..., o_j$ be the list of intervals in an optimal cover. In both cases, let these lists be sorted by increasing start time.
We claim the following:
Lemma 1. *for all i,* $\mathsf{END}(a_i) \geq \mathsf{END}(o_i)$.

Proof. Base Case: Let ℓ be the left-most point of any interval in $\mathcal{X}$. To be valid covers, both ALG and OPT must cover ℓ. Since the algorithm starts by sorting $\mathcal{X}$, the first step of ALG chooses an interval covering ℓ. By the tie-breaking of the sort, $\mathsf{END}(a_1)$ is the right-most point in any interval containing ℓ. Since OPT also covers ℓ, in sorted order o_1 must be an interval covering ℓ, thus $\mathsf{END}(a_1) \geq \mathsf{END}(o_1)$.
IH: Suppose $\mathsf{END}(a_k) \geq \mathsf{END}(o_k)$.
IS: Let ℓ_{k+1} be the left-most point in $\mathcal{X}$ not covered by any of $a_1, ..., a_k$. By IH, o_k also does not cover ℓ_{k+1}. Since OPT is a valid cover and sorted, o_{k+1} must cover ℓ_{k+1}. Now, consider the execution of the algorithm: when it added a_k, it deleted all elements covered by a_k (and already deleted everything earlier), thus, when choosing a_{k+1}, it considered only intervals containing ℓ_{k+1} and chose the one which reached the farthest right, by the tie-breaking order. Thus, o_{k+1} was an option for the algorithm, and it chose the farthest-right-reaching, so we have $\mathsf{END}(a_{k+1}) \geq \mathsf{END}(o_{k+1})$. □

With the Lemma proven, we observe that ALG is a minimum-sized cover. By construction, ALG covers every point in $\mathcal{X}$. Until the last interval a_k is added to ALG, there is still a point not covered by $\mathcal{Y}$ (as we delete all intervals that have all points covered); by the lemma $\mathsf{END}(\mathsf{ALG}_{k-1}) \geq \mathsf{END}(\mathsf{OPT}_{k-1})$, so OPT is not a cover until the final interval is added. Thus both OPT and ALG must contain the same number of intervals, and ALG is also optimal.
Running Time Note that the whole algorithm, including the deletion step, can be performed with a simple iteration — intervals in $\mathcal{X}$ will by covered by $\mathcal{Y}$ upon the addition of I only if their start time (and end time) is before I's end-time. We sort $\mathcal{X}$ at the beginning, and the remainder of the algorithm (scanning the list a single time, deleting or including each element, searching for the tie-breaking element up to a maximum of n total times) runs in n time, so the function runs in $\mathcal{O}(n \log(n))$ time total, dominated by sorting time.

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).

Solution:

You should take on the commissions in increasing order of r_i (starting with the smallest). Intuitively, you give the fastest growing account the longest to grow.
We will write an **exchange argument** for our proof.
Let OPT be an optimal solution, and ALG be the solution we have. Suppose, for the sake of contradiction, that OPT and ALG are different from each other.
Since ALG keeps elements in sorted order, it must be that OPT is not in sorted order. Then there will be a consecutive pair of elements; call them $j, j+1$ where $r_j > r_{j+1}$. We will show that swapping these elements increases our earnings. In OPT, we get $\$cr_j^j + cr_{j+1}^{j+1}$ from j and $j+1$. Now suppose we swap just j and $j+1$. We will instead earn $cr_{j+1}^j + cr_j^{j+1}$
Observe that with this swap we have another ordering, and have not affected the amount earned for any other commission. Thus if $cr_{j+1}^j + cr_j^{j+1} > cr_j^j + cr_{j+1}^{j+1}$, we will have found a better option than OPT, giving our desired contradiction.
To show the desired inequality, we begin with the observation that:

$$r_j^j(r_j - 1) > r_{j+1}^j(r_{j+1} - 1)$$

which follows from $r_j > r_{j+1}$ (that this pair is not in sorted order) and that all terms in the expression are positive (since the rates themselves are greater than 1).
Distributing, we have

$$r_j^{j+1} - r_j^j > r_{j+1}^{j+1} - r_{j+1}^j$$

Rearranging, so everything is positive we have

$$r_j^{j+1} + r_{j+1}^j > r_{j+1}^{j+1} + r_j^j$$

Multiplying by c and reordering the terms, we have

$$cr_{j+1}^j + cr_j^{j+1} > cr_j^j + cr_{j+1}^{j+1}$$

as desired. We thus have that swapping will produce a better ordering than OPT, a contradiction.

Solution:

Remarks:

- The algebra in the proof above is much more easily discovered backwards (starting from the desired conclusion), but remember to put proofs in proper logical order.
- This proof relies on the r_i being greater than 1. The problem is different if that isn't the case!
- You could also write a "greedy stays ahead" proof! Though you'd have to keep track of both the rates themselves and the amounts in the accounts to write the proof.