Topic Introduction

Today, let's dive into the world of **Greedy Reconstruction and Optimization**. If you've ever tried to fit puzzle pieces together quickly or wanted to spend the least amount of money possible, you've already thought "greedily." In algorithms, a greedy strategy makes the *best local choice* at each step, hoping these choices will lead to a globally optimal solution.

What is a Greedy Algorithm?

A greedy algorithm builds up a solution piece by piece, always choosing the next piece that offers the most immediate benefit. Unlike dynamic programming, greedy algorithms never go back to reconsider their choices. This makes them fast and often simple — but only works when the problem exhibits the **greedy-choice property** (local optimum leads to global optimum).

How does it work?

At each step:

- Consider the available options.
- Pick the one that looks best right now.
- Repeat until finished.

When and why is it useful in interviews?

Greedy algorithms are interview favorites because:

- They test your ability to identify optimal substructure and greedy-choice property.
- Greedy solutions are usually efficient (linear or O(n log n)).
- Many "minimize" or "maximize" problems lend themselves to greedy thinking.

Quick Example (not from our problems)

Coin Change (minimum coins):
Given coins of certain denominations and a target amount, what is the minimum number of coins needed to make that amount? If you always pick the largest coin less than the remaining amount, that's a greedy approach. (Note: Works perfectly for US coins, but not all coin systems!)

Today's trio of challenges all center around *greedy reconstruction and optimization*:

- **Queue Reconstruction by Height:** Rebuild a queue to fit height and position rules.
- **Non-decreasing Array:** Find the least modifications needed to get a non-decreasing array.
- **Minimum Number of Arrows to Burst Balloons:** Shoot as few arrows as possible to burst all balloons represented by intervals.

Why are these grouped? Each requires a sequence of "best local" decisions to reconstruct or optimize a structure. Sometimes you're sorting and inserting, other times you're choosing when to act or adjust — but in each, greediness is the key to efficiency.

Problem 1: Queue Reconstruction by Height

LeetCode: Queue Reconstruction by Height

Problem Restated

Given a list of people, each described by two numbers `[height, k]`:

- `height`: the person's height.
- `k`: the number of people in front who are at least as tall.

Reconstruct the queue so that for every person, exactly `k` people in front of them have a height greater or equal to theirs.

Example Input:

```
[[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]
```

Example Output:

```
[[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]
```

Step-by-Step Example

Let's break down the first input:

- `[7,0]` means: Height 7, wants 0 people in front at least as tall.
- `[4,4]` means: Height 4, wants 4 people in front at least as tall.

How do we reconstruct?

Thought Process

This is a classic greedy reconstruction. If we insert people in a certain order, the queue will always satisfy everyone's requirement.

Try this by hand!

Write down the heights and k-values. Try arranging some by hand. You'll notice:

- Tall people can be placed first — their position won't be affected by shorter people.
- For each person, their `k` value is the correct *index* among those as tall or taller.

Additional Test Case

Input: `[[6,0],[5,0],[4,0],[3,2],[2,2],[1,4]]`
Try reconstructing this queue before reading the solution!

Brute Force

Try all possible permutations and check if they satisfy the `k` requirements.

Time Complexity: O(n!) — not practical for large inputs.

Optimal Approach

Greedy Sorting and Insertion:

- **Sort** people by:
  - Descending height (tallest first).
  - If heights are equal, ascending `k`.
- **Insert** each person into the queue at index `k`.
  - Because taller people are already in place, inserting by `k` maintains correctness.

Why does this work?

- When inserting a person, all previous people are as tall or taller.
- Inserting at position `k` means exactly `k` taller or equal people are before them.

Python Solution

```
def reconstructQueue(people):
    # Step 1: Sort by decreasing height, and for equal height, increasing k
    people.sort(key=lambda x: (-x[0], x[1]))
    queue = []
    # Step 2: Insert each person at the index equal to their k value
    for person in people:
        queue.insert(person[1], person)
    return queue
```

Complexity:

- Sorting: O(n log n)
- Insertion: Each insert can be O(n), for n people: O(n^2)
- **Total:** O(n^2) time, O(n) space

Code Breakdown

- `people.sort(...)` sorts by height descending, k ascending.
- `queue = []` initializes our result queue.
- For each person, `queue.insert(person[1], person)` puts them at the correct spot.

Trace Example

Input: `[[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]`

Sorted: `[[7,0],[7,1],[6,1],[5,0],[5,2],[4,4]]`

Inserting:

- `[7,0]` at 0: `[[7,0]]`
- `[7,1]` at 1: `[[7,0],[7,1]]`
- `[6,1]` at 1: `[[7,0],[6,1],[7,1]]`
- `[5,0]` at 0: `[[5,0],[7,0],[6,1],[7,1]]`
- `[5,2]` at 2: `[[5,0],[7,0],[5,2],[6,1],[7,1]]`
- `[4,4]` at 4: `[[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]`

Try this next:

Input: `[[6,0],[5,0],[4,0],[3,2],[2,2],[1,4]]`

What is the output?

Self-attempt prompt:

Take a moment to solve this on your own before jumping into the solution. Pen and paper can help visualize the insertions!

Problem 2: Non-decreasing Array

LeetCode: Non-decreasing Array

Problem Restated

Given an array, can you make it non-decreasing by modifying at most one element? (Non-decreasing: each element less than or equal to the next.)

Example Input: `[4,2,3]`

Example Output: `True` (Change 4 to 2 or 2 to 4)

Why is this similar?

You need to make the *minimum* number of changes (ideally one), always making the most “greedy” local fix at the first sign of trouble.

Brute Force

Try modifying every possible element and check if the resulting array is non-decreasing.

Time: O(n^2)

Optimal Greedy Approach

- Loop through the array.
- When you find `nums[i] > nums[i+1]`, you have a problem.
- You can modify *either* `nums[i]` or `nums[i+1]`, but only once in the whole array.

- Decide greedily:
  - If `i == 0` or `nums[i-1] <= nums[i+1]`, lower `nums[i]` to `nums[i+1]`.
  - Else, raise `nums[i+1]` to `nums[i]`.
- If you find more than one problem spot, return False.

Pseudocode

```
count = 0
for i from 0 to n-2:
    if nums[i] > nums[i+1]:
        count += 1
        if count > 1:
            return False
        if i == 0 or nums[i-1] <= nums[i+1]:
            nums[i] = nums[i+1]
        else:
            nums[i+1] = nums[i]
return True
```

Example

Input: `[3,4,2,3]`

- 4 > 2 at i=1
- nums[0]=3 <= 2? No, so raise nums[2]=2 to 4
- New array: `[3,4,4,3]`
- Now 4 > 3 at i=2 — second problem! So return False.

Output: `False`

Another Test Case

Try: `[5,7,1,8]`

Trace

Input: `[4,2,3]`

- 4 > 2 at i=0
- i==0, so lower 4 to 2
- Array: `[2,2,3]` which is non-decreasing.

Time and Space Complexity

- Time: O(n)
- Space: O(1)

Problem 3: Minimum Number of Arrows to Burst Balloons

LeetCode: Minimum Number of Arrows to Burst Balloons

Problem Restated

Given a list of balloons as intervals `[start, end]`, find the minimum number of arrows needed so every balloon is burst (an arrow bursts all balloons that overlap at its position).

What's different?

This is a classic greedy interval covering problem. You want to shoot as few arrows as possible, each bursting as many overlapping balloons as possible.

Brute Force

Try all combinations of arrow placements — way too slow!

Greedy Optimal Approach

- **Sort** all intervals by their `end` value.
- **Initialize** arrow count and track the position of the last arrow.
- For each balloon:
  - If its `start` is after the last arrow, shoot a new arrow at its `end`.
  - Else, it's already burst.

Pseudocode

```
sort balloons by end
arrows = 0
last_arrow = -infinity
for balloon in balloons:
    if balloon.start > last_arrow:
        arrows += 1
        last_arrow = balloon.end
return arrows
```

Example

Input: `[[10,16],[2,8],[1,6],[7,12]]`

- Sort by end: `[[1,6],[2,8],[7,12],[10,16]]`
- Shoot first arrow at 6 (covers [1,6] and [2,8])
- [7,12] starts at 7 > 6, so shoot at 12
- [10,16] starts at 10 <= 12, already burst
- Total: 2 arrows

Another Test Case

Try: `[[1,2],[2,3],[3,4],[4,5]]`
How many arrows do you need?

Time and Space Complexity

- Time: O(n log n) for sorting
- Space: O(1) (if sorting in-place)

Summary and Next Steps

Today, you tackled three classic greedy optimization and reconstruction problems:

- Queue reconstruction: Greedily sort and insert to satisfy placement rules.
- Non-decreasing array: Greedily make the minimal local modification.
- Bursting balloons: Greedily pick the earliest possible shot to cover the most.

Key Patterns:

- Sort first, then take local greedy steps.
- For reconstructions, insert in a way that preserves the property.
- For modifications, fix the first trouble spot in the best way possible.
- For coverings, pick the next action as late/early as possible to maximize coverage.

Common Pitfalls:

- Not sorting in the right order before greedy steps.
- Overcomplicating: Greedy is about simple, forward-only decisions.
- Forgetting to check if the greedy-choice property really holds.

Action List

- Try solving all three problems on your own — even if code was provided!
- For Problem 2 and 3, can you find a different approach (e.g., dynamic programming)?
- Explore other greedy problems, like “Merge Intervals” or “Activity Selection.”
- Dry-run your solution with edge cases, such as empty lists or all-equal elements.
- Check discussions for alternative solutions, and compare their clarity and efficiency.

Remember: Every time you try, you’re building the muscle to spot these greedy patterns more quickly! If you get stuck, review the step-by-step logic, and keep practicing. Happy coding!