Topic Introduction

Today, we're diving into the world of **Greedy Algorithms** — a classic and powerful approach for solving optimization problems efficiently. You've probably heard of greedy algorithms before, but let's make sure you truly *get* when, why, and how to use them.

Greedy algorithms are all about making the locally optimal choice at each step, *hoping* that these choices add up to a globally optimal solution. Instead of exhaustively checking all possibilities (like brute-force or dynamic programming), greedy approaches move step by step, always picking what looks best *right now*.

How does it work?
Imagine you're collecting coins in a video game. At every fork, you choose the path with the most coins visible. If the problem is designed right, this simple, local strategy will get you the maximum loot. But beware: not every problem can be solved this way — greedy only works when the problem has the "greedy-choice property" (making a local optimum leads to a global optimum).

Why are greedy algorithms useful in interviews?

- They lead to elegant, efficient solutions.
- They can often be coded quickly.
- Recognizing which problems can be solved greedily is a valuable skill.

Quick Example (not one of today's main problems):
Suppose you want to give change for 63 cents using the least number of US coins. At each step, you take the largest coin possible (quarters, then dimes, etc.). This greedy approach works here because US coin denominations are compatible with it.

Today's three problems all use greedy strategies to optimize something: minimum resources, minimum waiting time, or finding a feasible solution in a circular route. Let's see what ties them together:

- **Gas Station:** Find the starting point to complete a loop where fuel and cost vary.
- **Candy:** Distribute candies to children so that each kid with a higher rating than their neighbors gets more, using the fewest candies.
- **Task Scheduler:** Schedule tasks with cooldowns to minimize total time.

What connects them?
Each asks, "How can we achieve the best (minimum or feasible) result when faced with constraints?" Greedy algorithms shine here by making decisions that look best at each step, leading to optimized solutions.

Let's tackle these one by one, starting with Gas Station.

Problem 1: Gas Station

Problem Statement (Rephrased):
Given two integer arrays `gas` and `cost`, each of length n, where `gas[i]` is the amount of fuel at station i and `cost[i]` is the fuel needed to travel from station i to station (i + 1). Starting with an empty tank at any station, can you complete the circuit? If so, return the starting station's index. If not, return -1.
LeetCode: Gas Station

Example:

Input:

`gas = [1,2,3,4,5]`

`cost = [3,4,5,1,2]`

Output: 3

Explanation:

- Start at station 3 (gas 4). You have enough fuel to get to station 4 (need 1, have 4, so 3 left).
- At station 4: You have 3+5=8, need 2 to get to 0, left with 6.
- At station 0: You have 6+1=7, need 3 to get to 1, left with 4.
- At station 1: 4+2=6, need 4 to get to 2, left with 2.
- At station 2: 2+3=5, need 5 to get to 3, left with 0.
- You completed the circuit!

Try this input yourself:

`gas = [2,3,4]`

`cost = [3,4,3]`

What should the output be?

Brute-Force Approach:

- Try starting at every station.
- For each, simulate the trip.
- Time complexity: O(n^2) (very slow for large n).

Optimal Greedy Approach:

Key insight:

- If the total gas is less than the total cost, it is impossible to complete the circuit.
- If you can't reach station i+1 from your current starting point, then none of the stations between your previous start and i+1 can be a valid starting point.

Steps:

- Track total gas and total cost — if gas < cost overall, return -1.
- Iterate through stations, keeping a running tank. If tank drops below zero, reset the starting station to the next index and reset the tank to zero.
- Return the starting station index.

Python Solution:

```
def canCompleteCircuit(gas, cost):
    total_gas = 0
    total_cost = 0
    tank = 0
    start = 0

    for i in range(len(gas)):
        total_gas += gas[i]
        total_cost += cost[i]
        tank += gas[i] - cost[i]
```

```
        # If tank is negative, can't start from current start
        if tank < 0:
            start = i + 1  # Try the next station as the start
            tank = 0       # Reset the tank for the next trial

    if total_gas < total_cost:
        return -1
    else:
        return start
```

Time & Space Complexity:

- Time: O(n)
- Space: O(1) (just a few variables)

Code Breakdown:

- `total_gas` and `total_cost` check if the trip is possible at all.
- `tank` keeps track of your current surplus or deficit.
- When `tank` goes negative, you know you can't have started from the previous start — so you move the start forward.
- If the total gas is enough, the `start` index at the end is the answer.

Step-by-step Trace:

For `gas = [1,2,3,4,5]`, `cost = [3,4,5,1,2]`:

- i = 0: tank = -2 — reset start to 1, tank=0
- i = 1: tank = -2 — reset start to 2, tank=0
- i = 2: tank = -2 — reset start to 3, tank=0
- i = 3: tank = 3
- i = 4: tank = 6

Total gas = 15, total cost = 15, so return start = 3.

Try this test case yourself:

`gas = [5,1,2,3,4]`, `cost = [4,4,1,5,1]`

Take a moment to solve this on your own before jumping into the code!

Problem 2: Candy

Problem Statement (Rephrased):

You are given an array `ratings` of children's ratings. Distribute candies so that each child has at least one candy, and any child with a higher rating than their immediate neighbor(s) gets more candies than them. Find the minimum total candies needed.

LeetCode: Candy

Why is this grouped here?

Like Gas Station, it's an optimization problem with a local constraint — and a greedy approach leads to the optimal solution.

Brute-Force Approach:

- Assign 1 candy to each, then keep increasing candies for children who violate the rule (multiple passes).
- Time complexity: O(n^2) or worse if not careful.

Optimal Greedy Approach:

Key insight:

- Two passes (left to right, right to left) are enough:
    - Left to right: If current rating > previous, give one more candy than previous.
    - Right to left: If current rating > next, give at least one more than next.

Steps:

- Give everyone 1 candy.
- Left pass: For each child, if their rating is higher than their left neighbor, increment their candy count.
- Right pass: For each child, if their rating is higher than their right neighbor, ensure their candy count is at least one more than the right neighbor.
- Sum up all candies.

Example:

`ratings = [1,0,2]`

- Left pass: [1,1,2]
- Right pass: [2,1,2]
- Total: 5

Try this test case:

`ratings = [1,2,2]`

What should the output be?

Step-by-step Trace:

Let's walk through `ratings = [1,3,2,2,1]`:

- Initial candies: [1,1,1,1,1]
- Left pass:
    - 3 > 1: candies[1] = candies[0] + 1 = 2 => [1,2,1,1,1]
    - 2 < 3: no change
    - 2 = 2: no change
    - 1 < 2: no change
- Right pass:
    - 1 < 2: no change
    - 2 = 2: no change
    - 2 > 1: candies[2] = max(candies[2], candies[3]+1) = 2 => [1,2,2,1,1]
    - 3 > 1: candies[1] = max(2, 2+1) = 3 => [1,3,2,1,1]
- Total: 1+3+2+1+1 = 8

Pseudocode:

```
function candy(ratings):
    n = length of ratings
    candies = [1] * n
```

```
    // Left to right
    for i = 1 to n-1:
        if ratings[i] > ratings[i-1]:
            candies[i] = candies[i-1] + 1

    // Right to left
    for i = n-2 downto 0:
        if ratings[i] > ratings[i+1]:
            candies[i] = max(candies[i], candies[i+1] + 1)

    return sum(candies)
```

Time & Space Complexity:

- Time: O(n)
- Space: O(n) (for the candies array)

Try this test case:

`ratings = [1,2,3,2,1]`

Problem 3: Task Scheduler

Problem Statement (Rephrased):

Given a list of tasks (represented by capital letters) and a non-negative integer n, each same task must be separated by at least n intervals. Find the least number of intervals needed to finish all tasks, allowing idle intervals if necessary.

LeetCode: Task Scheduler

What's new here?

This problem is more abstract: instead of a linear or circular sequence, you are handling counts and idle time. The greedy aspect is in how you arrange tasks to minimize idle slots.

Brute-Force Approach:

- Try all possible task orders, checking for cooldowns.
- Exponential time: infeasible!

Optimal Greedy Approach:

Key insight:

- The most frequent task determines the schedule's framework.
- Place the most frequent task as far apart as needed, filling gaps with other tasks or idles.

Steps:

- Count the frequency of each task.
- Find the highest frequency (`max_freq`).
- The minimum time is at least:

`(max_freq - 1) * (n + 1) + num_tasks_with_max_freq`

- The answer is the maximum of this formula and the total number of tasks.

Example:

`tasks = ["A","A","A","B","B","B"], n = 2`

- A and B both appear 3 times.
- Idle slots needed: (3-1) * (2+1) + 2 = 8
- Actual tasks = 6, so answer is 8.

Try this test case:

`tasks = ["A","A","A","B","B","B"], n = 0`

What should the output be?

Pseudocode:

```
function leastInterval(tasks, n):
    count = frequency of each task
    max_freq = max value in count
    num_max = number of tasks with max_freq

    part_count = max_freq - 1
    part_length = n + 1
    min_intervals = part_count * part_length + num_max

    return max(min_intervals, length of tasks)
```

Time & Space Complexity:

- Time: O(N), where N is the number of tasks
- Space: O(1) (since number of tasks types is limited to 26)

Try this test case:

`tasks = ["A","A","A","A","B","C","D","E","F","G"], n = 2`

Can you implement and test this in your favorite language?

Bonus reflection:

Notice how, in all three problems, the greedy strategy involves either advancing the start point, propagating local constraints, or maximizing resource use to minimize idle time. Can you think of other problems where a similar greedy framework applies?

Summary and Next Steps

Why these problems together?

Each uses a greedy algorithm to solve an optimization challenge — whether it's completing a circuit, distributing resources, or minimizing time. You've now seen how greedy thinking can be applied in different ways:

- Resetting the start point (Gas Station)
- Propagating constraints in both directions (Candy)
- Filling gaps as efficiently as possible (Task Scheduler)

Key Patterns to Remember:

- When local optimal choices lead to a global optimum, greedy is your friend.

- Always check if greedy works for the problem — sometimes you need proof or a counterexample.
- Don’t forget edge cases (like when total supply isn’t enough).

Common Traps:

- Assuming greedy always works — watch out for problems where local choices can block global optimum.
- Missing cases where you need to process both directions (like in Candy).
- Forgetting to check overall feasibility before proceeding (like total gas vs. cost).

Action List:

- Solve all three problems on your own — even the one with code provided.
- For Problem 2 and 3, try coding the brute-force solution and compare its performance.
- Explore other greedy problems (e.g., Jump Game, Interval Scheduling).
- Review your solutions for edge cases and clarity. Compare with others’ for alternative perspectives.
- If you get stuck, analyze the problem’s constraints — does a greedy strategy *really* apply?
- Keep practicing and reflecting — the more you see greedy, the quicker you’ll recognize it!

Happy coding and keep making those optimal choices!