Solution: Gas Stations

Let's solve the Gas Stations problem using the Greedy Techniques pattern.

We'll cover the following

Statement

There are n gas stations along a circular route, where the amount of gas at the i^{th} station is `gas[i]`.

We have a car with an unlimited gas tank, and it costs `cost[i]` of gas to travel from the i^{th} station to the next $(i+1)^{th}$ station. We begin the journey with an empty tank at one of the gas stations.

Find the index of the gas station in the integer array `gas` such that if we start from that index we may return to the same index by traversing through all the elements, collecting `gas[i]` and consuming `cost[i]`.

- If it is not possible, return -1.
- If there exists such index, it is guaranteed to be unique.

Constraints:

- `gas.length` $==$ `cost.length`
- $1 \leq$ `gas.length`, `cost.length` $\leq 10^3$
- $0 \leq$ `gas[i]`, `cost[i]` $\leq 10^3$

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach would be to choose each station as a starting point in the outer loop and try to visit every other station while maintaining the current gas level in the inner loop. This will check for every gas station to be the starting gas station from where we can complete a round trip clockwise.

The time complexity of the naive approach would be $O(n^2)$, since we are using a nested loop.

Optimized approach using the greedy pattern

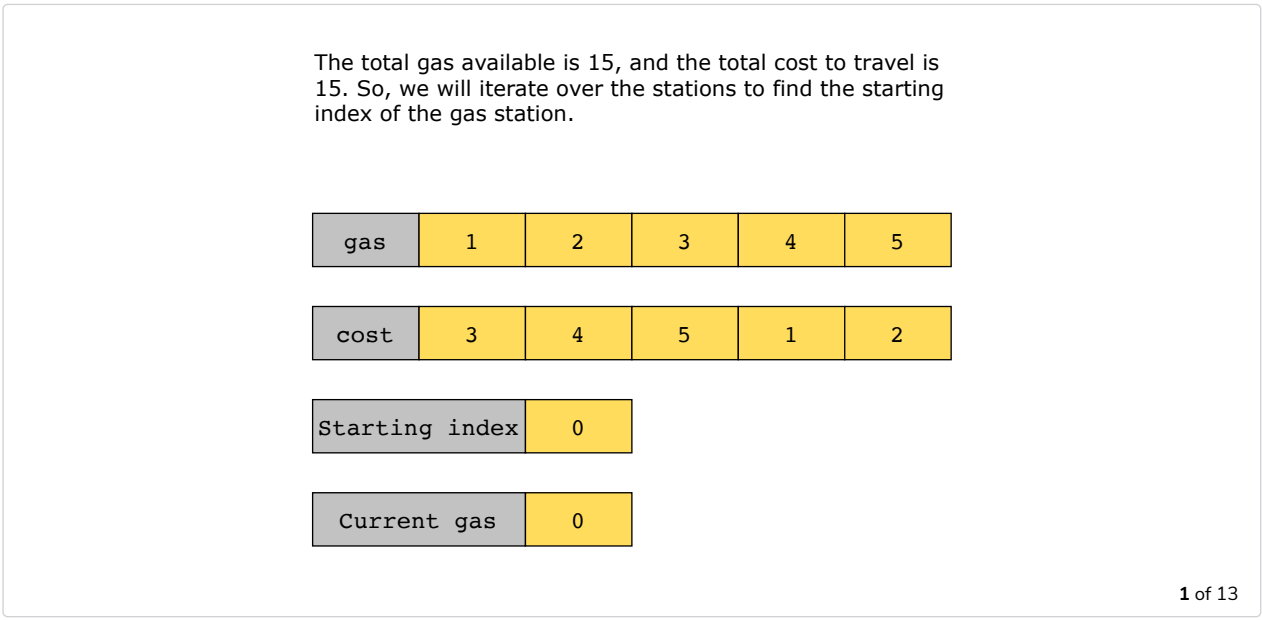
The greedy approach is to keep track of the amount of gas in the tank and the total cost of the journey. If we find that we cannot complete the journey starting from the current gas station, we reset the starting point to the next gas station and continue from there. This means we are making locally optimal choices at each step to find a solution. Therefore, this approach is considered a greedy algorithm.

The logic of the above algorithm is given below:

1. If the total cost of the journey is greater than the total amount of gas available at all the gas stations, then it is impossible to travel through all gas stations, so the function returns -1.
2. We will iterate through the gas stations from the start. While iterating these stations, we will perform the steps below:
    - At each gas station, we will calculate the amount of current gas available. We'll do this by subtracting the cost of the journey from the gas available at that station and adding it to the current gas available.
    - If the amount of currently available gas at any station becomes negative, we cannot travel further from the current station. Therefore, we reset the `currentGas` variable to 0 and start from the next gas station.
3. Return the index of the gas station from where we can start our journey in such a way that we can travel through all gas stations and reach the starting gas station again.

The starting index will be the index of that gas station from which we can start our journey in such a way that we can travel through all gas stations and reach the starting gas station again.

Let's look at the following illustration to get a better understanding of the solution:



Let's implement the algorithm as discussed above:

Java

```java
import java.util.*;

class GasStations {
    public static int gasStationJourney(int[] gas, int[] cost) {
        // Check if it is possible to complete the journey based on total gas and cost.
        if (Arrays.stream(cost).sum() > Arrays.stream(gas).sum()) {
            return -1;
        }
```

```
        }

        // Initialize variables for tracking total gas and starting index.
        int currentGas = 0;
        int startingIndex = 0;

        // Iterate over all gas stations in the array.
        for (int i = 0; i < gas.length; i++) {
            // Update current gas level by adding gas and subtracting cost at current station.
            currentGas += gas[i] - cost[i];

            // If the current gas level is negative, reset it to zero and update the starting index.
            if (currentGas < 0) {
                currentGas = 0;
                startingIndex = i + 1;
            }
        }

        // Return starting index of gas station that allows journey to be completed.
        return startingIndex;
    }
```

Solution summary

To recap, the solution to this problem can be divided into the following three parts:

1. If the sum of all values in the gas array is less than the sum of all values in the cost array, we return `-1`, because there will be no gas station from which we can start our journey to complete the round trip.
2. Traverse the gas list with `startingIndex` initially set to `0`.
    - We calculate `currentGas` left by incrementing `currentGas` (initially `0` at the starting point) to (`gas[i] - cost[i]`).
    - If at any point, our `currentGas` is less than `0`, it means we can not start from this index. Therefore, we increment `startingIndex` to `i+1` and reset `currentGas` to `0`.
3. Return `startingIndex` at the end of the traversal.

Time complexity

The time complexity is $O(n)$, since there are n elements in the array and we only visit each element once.

Space complexity

The space complexity is $O(1)$, since we don't use any additional data structures.

Back

Gas Stations

Next

Two City Scheduling

Mark as Completed