134. Gas Station

Medium Greedy Array Leetcode Link

Problem Description

In this problem, we are given `n` gas stations that are placed along a circular route. Each gas station has a certain amount of gas given in an array `gas[]` where `gas[i]` is the amount of gas available at the `i`th gas station. Additionally, we have an array `cost[]` where `cost[i]` represents the amount of gas needed to travel from the `i`th station to the next station `(i + 1)th`. The goal is to find out if it's possible to start at a gas station with an empty gas tank and travel around the entire route once without running out of gas. If it is possible, we need to return the index of the starting gas station, otherwise, we return -1. It is important to note that if an answer exists, it is unique.

Intuition

To solve this problem, we need to check if the total amount of gas is at least as much as the total cost. If it is not, we cannot complete the trip, and we return -1. Otherwise, a start point exists, and we need to find it.

The intuition behind the solution is to keep track of the total gas in the car (`s`) as we try to make the circuit. We start at the last gas station and work backwards to see if we can reach it from the second to last, third to last and so on. This is done because if we can't complete the circuit starting at station `i`, then we also can't complete it starting at any station before it, as we would have even less gas by the time we reached `i`.

1. We initialize two pointers `i` and `j` to the last station (index `n - 1`).
2. We then try to make the trip by simulating the journey, incrementing `j` and decrementing `i` as necessary as we simulate traveling around the circle. Every time we move to the next station, we update the total gas in the car `s` by adding the gas we get from the current station (`gas[j]`) and subtracting the cost to get to the next station (`cost[j]`).
3. If at any point our total `s` becomes negative, it means we can't reach the next station from our current starting point. Therefore, we need to change our starting point to the previous station by decrementing `i` and adding the gas available at the new starting station minus the cost to get to the next station.
4. We continue this process until we've checked all `n` stations.

If by the end of this process, `s` is still negative, it means we couldn't find a starting point that could complete the circuit; thus, we return -1. Otherwise, the `i` at which we finish is our starting point that can complete the circuit.

Solution Approach

The solution approach for this problem is based on the greedy algorithm. The implementation may seem a bit counter-intuitive at first glance because it works backwards, starting from the end of the loop rather than the beginning.

Here's a step-by-step guide through the algorithm:

- Initialize two pointers `i` and `j` to the last position in the arrays (`n - 1`). They will indicate the starting point of our journey (`i`) and the current station we are considering (`j`).
- Initialize two variables: `cnt` to keep track of how many stations we have considered so far, and `s` to represent the total amount of gas available subtracting the cost needed to get to the next station.
- Use a `while` loop to iterate until we have considered all `n` stations (`cnt < n`). For each iteration:
  - We add the net gas (gas at current station minus cost to next station) for station `j` to `s` (total gas available) and increment `cnt`.
  - We move to the next station `j` by using modular arithmetic `(j + 1) % n`, which wraps the index around to the start of the array when `j` reaches the end.
  - If ever our total available gas `s` drops below zero, it means we cannot reach the next station from our current start point, and thus we move our candidate starting point back one station (`i -= 1`).
  - We then incorporate this station's net gas into `s` and increment `cnt`.
- After finishing the loop, if our total gas `s` is negative, it means that we were not able to find a path that completes the circuit, and as per problem statement, we return `-1`.
- In case `s` is non-negative, it means that the car can traverse the entire circuit starting from gas station at index `i`. The position `i` is our answer, the index of the starting gas station.

Here are some insights into the data structures and patterns used:

- **Arrays:** The gas stations' gas and cost are given as arrays, and we are traversing these arrays to calculate the net gas.
- **Modular Arithmetic:** This is used to wrap the circular route, allowing us to move through the circular array repeatedly.
- **Greedy Algorithm:** We are using a greedy approach because at each step we take the best local decision: to move forward or change our start point, hoping to find a global optimum (a start point that lets us complete the circuit).

The overall complexity of the algorithm is O(n), because we are doing a single pass through the stations.

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Suppose we have 4 gas stations with the following `gas` and `cost` values:

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

We will walk through the steps of the algorithm described above using these values.

Step 1: Initialization

We set `i` and `j` to the last index which is 3 in this case.

- `i = 3`
- `j = 3`
- `cnt = 0` to count the stations considered.
- `s = 0` to keep track of the net gas.

Step 2: Start the Iteration

We start iterating while `cnt < n`, where `n = 4` is the number of gas stations.

Iteration 1:

- `j = 3`
- Add net gas of the current station `(j)` to `s`: `s += gas[j] - cost[j]` ⇒ `s += 4 - 1` ⇒ `s = 3`
- Increment `cnt`: `cnt = 1`
- Move to the next station (in this case, wrap around to the first) `j = (j + 1) % 4` ⇒ `j = 0`

Iteration 2:

- `j = 0`
- Add net gas of the current station to `s`: `s += gas[j] - cost[j]` ⇒ `s += 1 - 2` ⇒ `s = 2`
- Increment `cnt`: `cnt = 2`
- Move to the next station: `j = (j + 1) % 4` ⇒ `j = 1`

Iteration 3:

- `j = 1`
- Add net gas of the current station to `s`: `s += gas[j] - cost[j]` ⇒ `s += 2 - 3` ⇒ `s = 1`
- Increment `cnt`: `cnt = 3`
- Move to the next station: `j = (j + 1) % 4` ⇒ `j = 2`

Iteration 4:

- `j = 2`
- Before adding the net gas, we note that `cost[j]` is greater than `gas[j]` and if we added it, `s` would drop below zero. So, we need to change our start point.
- Decrement `i`: `i -= 1` ⇒ `i = 2`
- We won't add `gas[j] - cost[j]` to `s` just yet because we are potentially moving our start point.

Adjust Starting Point:

- Update `s` with the new start point's net gas: `s += gas[i] - cost[i]` ⇒ `s += 3 - 1` ⇒ `s = 3`
- Increment `cnt`: `cnt = 4`
- Since we have considered all stations (`cnt = n`), we break out of the loop.

Determining the Result:

At the end of the loop:

- `s = 3`
- `i = 2`

Since `s` is not negative, it means it's possible to complete the circuit, starting from station 2. So, our answer is `i = 2`.

It's important to note that the explanation above used explicit iterations for clarity, while in actual implementation, checking the total gas `s` against zero and updating the start point `i` would happen inside of the same iteration.

Python Solution

```python
from typing import List

class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        # Initialize the length of the gas and cost lists
        num_stations = len(gas)

        # Initialize pointers for traversing the gas stations
        start_index = end_index = num_stations - 1

        # Initialize counter for stations visited and total balance of gas
        stations_visited = total_gas_balance = 0

        # Loop until all stations have been visited
        while stations_visited < num_stations:
            # Update the total balance by adding current gas and subtracting current cost
            total_gas_balance += gas[end_index] - cost[end_index]

            # Increment the number of stations visited
            stations_visited += 1

            # Move to the next station, wrapping around if necessary
            end_index = (end_index + 1) % num_stations

            # While the total balance is negative and we haven't visited all stations
            # move the start index backwards and adjust the balance.
            while total_gas_balance < 0 and stations_visited < num_stations:
                # Move the start index to the previous station
                start_index = (start_index - 1 + num_stations) % num_stations

                # Update the total balance by adding gas and subtracting cost at the new start
                total_gas_balance += gas[start_index] - cost[start_index]

                # Increment the number of stations visited
                stations_visited += 1

        # If the total balance is negative, return -1 indicating the circuit cannot be completed,
        # otherwise return the start index
        return -1 if total_gas_balance < 0 else start_index
```

Java Solution

```java
class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        // n represents the total number of gas stations
        int n = gas.length;

        // Initialize index pointers for the circular route
        int start = n - 1; // Start from the last station
        int end = n - 1;   // End at the last station initially

        // Initialize a sum to keep track of the remaining gas and count of stations checked
        int sum = 0;
        int stationsChecked = 0;

        // Iterate until we've checked all stations
        while (stationsChecked < n) {
            // Calculate the remaining gas after visiting a station
            sum += gas[end] - cost[end];
            stationsChecked++; // Increment the number of stations checked
            end = (end + 1) % n; // Move to the next station circularly

            // If we have a deficit (sum < 0), try starting from an earlier station
            while (sum < 0 && stationsChecked < n) {
                start--; // Decrement start index to check an earlier station
                sum += gas[start] - cost[start]; // Update sum for the new start station
                stationsChecked++; // Increment the number of stations checked
            }
        }

        // If we have remaining gas (sum >= 0), return the starting station
        // Else, return -1 indicating the trip cannot be completed
        return sum >= 0 ? start : -1;
    }
}
```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    int canCompleteCircuit(std::vector<int>& gas, std::vector<int>& cost) {
        // 'n' represents the number of gas stations.
        int n = gas.size();

        // Start from the last gas station.
        int start = n - 1;

        // 'j' will be used to traverse the gas stations.
        int j = n - 1;

        // 'tours' will count how many gas stations we have considered.
        int tours = 0;

        // 'totalFuel' will keep track of our current fuel balance.
        int totalFuel = 0;

        // Loop through the gas stations to see where we can start.
        while (tours < n) {
            // Add net fuel (after consuming cost) at the current station.
            totalFuel += gas[j] - cost[j];

            // Move to the next gas station.
            tours++;
            j = (j + 1) % n;

            // If our total fuel goes negative, move start one station backward
            // and add the net fuel at that station to 'totalFuel'.
            while (totalFuel < 0 && tours < n) {
                // Only able to move backward if the total number of steps
                // considered is less than the number of gas stations.
                start--;
                totalFuel += gas[start] - cost[start];

                // Another station considered.
                tours++;
            }
        }

        // If after considering all stations the 'totalFuel' is still negative,
        // there is no way to complete the circuit. Otherwise, return the starting station.
        return totalFuel < 0 ? -1 : start;
    }
};
```

Typescript Solution

```typescript
// Function to determine if a vehicle can complete a circuit given the gas and cost.
function canCompleteCircuit(gas: number[], cost: number[]): number {
    // Total number of gas stations.
    const totalStations = gas.length;

    // Initialize pointers for current and next station.
    let currentStation = totalStations - 1;
    let nextStation = totalStations - 1;

    // Initialize surplus gas variable to store the surplus/deficit gas amount.
    let surplusGas = 0;

    // Counter for how many stations have been visited.
    let stationsVisited = 0;

    // Loop through all stations until all stations have been visited.
    while (stationsVisited < totalStations) {
        // Calculate current surplus by adding gas available and subtracting the cost.
        surplusGas += gas[nextStation] - cost[nextStation];

        // Increment station visited counter.
        stationsVisited++;

        // Move to next station, wrap around if necessary.
        nextStation = (nextStation + 1) % totalStations;

        // If surplus is negative and not all stations have been visited,
        // move current station counter backwards and add to surplusGas.
        while (surplusGas < 0 && stationsVisited < totalStations) {
            currentStation = (currentStation - 1 + totalStations) % totalStations; // Ensure currentStation stays within bounds.
            surplusGas += gas[currentStation] - cost[currentStation];

            // Increment stations visited to account for new calculation.
            stationsVisited++;
        }
    }

    // Return the starting station index if a circuit is possible, else return -1.
    return surplusGas >= 0 ? currentStation : -1;
}
```

Time and Space Complexity

The given Python code is intended to solve the gas station problem, which involves finding a starting gas station from which a vehicle can travel around a circular route without running out of gas, assuming the vehicle starts with an empty gas tank.

The time complexity of the provided code is `O(n)`, where `n` is the number of gas stations. This is because although there are nested while-loops, the outer loop and the inner loop combined ensure that each station is visited at most twice: once when moving forward (`cnt` and `j` increment) and possibly once when moving backward (`i` decrement). The condition `cnt < n` prevents the code from examining more than `n` elements.

The space complexity of the algorithm is `O(1)`. This is because the solution uses only a fixed number of variables (`n`, `i`, `j`, `cnt`, `s`) and does not allocate any additional space that would grow with the input size.