871. Minimum Number of Refueling Stops

Hard | Greedy | Array | Dynamic Programming | Heap (Priority Queue) | Leetcode Link

Problem Description

This problem involves simulating a car's journey from a starting position to a destination, which is a certain number of miles away (`target`). The journey is not direct, as there are gas stations along the way, each with a certain amount of fuel available. The car starts with a certain amount of fuel (`startFuel`), and the goal is to determine the minimum number of stops at gas stations required to reach the destination. If the car can't reach the destination, the function should return `-1`. Importantly, the car consumes fuel at a rate of one liter per mile.

The gas stations are defined in an array called `stations`, where `stations[i]` contains two elements: the position of the station relative to the starting point, and the amount of fuel available at that station (`fuel[i]`). The car can refuel whenever it reaches a gas station and can take all the gas from that station. The problem's constraints allow the car to reach a gas station or the destination with exactly `0` fuel left and still be able to refuel or be considered arrived, respectively.

Intuition

The problem can be approached as a greedy algorithm combined with a priority queue. The intuition is to drive from one station to the next, using up fuel, and to always have enough fuel to reach the next station or the destination. However, since the car needs to minimize the number of stops, it shouldn't stop at every station to refuel. Instead, the car should only stop when it is necessary—meaning when it's running out of fuel.

To implement this strategy, use a max-heap (priority queue), where you can store the amount of fuel from stations the car has passed without stopping. If the car runs out of fuel before reaching the next station or the destination, it should refuel by taking the most significant amount of fuel it has passed. This is why using a max-heap is useful; it allows for accessing the largest amount of fuel quickly. The steps are as follows:

- Add each station's fuel to the priority queue when passing by without refueling.
- If the car needs to refuel (fuel drops below 0), it takes the largest available fuel from the priority queue, and that counts as a stop.
- If the car runs out of both fuel and potential refuels in the priority queue, it's impossible to reach the destination, and the function should return -1.
- Continue this process until the destination is reached.
- The number of times fuel is taken from the priority queue represents the minimum number of refueling stops required.

The consideration of necessary pauses for refueling ensures the optimal number of stops. By initializing the `stations` array with the target position (and zero fuel), the loop also conveniently handles the case of reaching the destination.

Solution Approach

The solution uses a priority queue to keep track of the gas amounts available at stations we've passed and a greedy algorithm approach to minimize the number of stops. Here's a step-by-step explanation of how the provided Python code implements this solution:

1. **Initialize priority queue and variables**: The `q` variable represents the priority queue (a max-heap), which is used to store the fuel amounts of gas stations that the car has passed. The `prev` variable stores the position of the last gas station, or the starting point initially, and `ans` is the count of refueling stops.

2. **Add the destination as a station**: To handle the arrival at the destination, it adds the destination itself as a gas station into the `stations` list with `0` fuel.

3. **Loop through stations**: For each gas station in the `stations` list, calculate the distance `d` from the `prev` position to this station's position `a`. Subtract this distance from the `startFuel` to simulate driving to the station.

4. **Refuel if necessary**: If `startFuel` drops below `0`, it means the car needs to refuel. It repeatedly pops the largest amount of fuel from the priority queue (note: Python's `heapq` module provides a min-heap, so we store negative values to simulate a max-heap). After each pop, increment the `ans` counter. Keep refueling (popping from the queue) until `startFuel` is non-negative (indicating the car can reach the next station or the destination).

5. **Check if destination is reachable**: After attempting to refuel from the priority queue, if `startFuel` is still negative, it means the car cannot reach the next station or the destination, and the function returns `-1`.

6. **Push current station's fuel into the queue**: Regardless of whether the car needed to refuel to reach the current station, push the negative of the current station's fuel amount `b` into the priority queue (in this way, we are preparing for a potential future refuel stop).

7. **Update `prev` position**: Update the `prev` variable to the current station's position for the next iteration.

8. **Return the number of stops**: After the loop completes (which happens when the car reaches the 'fake' gas station at the destination), return `ans` as the total number of refueling stops made.

The algorithm effectively determines the minimal number of refueling stops by always picking the refuel options that gave the maximum range expansion when necessary. This approach ensures that each stop contributes as much as possible towards progressing to the destination, thereby minimizing the total number of stops needed.

Example Walkthrough

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

Given:

- `target` distance to reach: 10 miles
- `startFuel`: 3 liters
- `stations`: `[[3, 3], [7, 2]]`, where the first element in each pair is the position of the station, and the second element is the fuel available at that station.

Approach:

Here is a step-by-step walkthrough applying the solution steps from the content above.

1. **Initialize priority queue and variables**: `q = []` (this is the max-heap for storing gas station fuel amounts), `prev = 0` (start position), and `ans = 0` (no refuel stops yet).

2. **Add the destination as a station**: Update `stations` to `[[3, 3], [7, 2], [10, 0]]` to include the target as a station with `0` fuel.

3. **Loop through the stations**: Starting at first station:

   - Station at `3` miles offering `3` liters of fuel.
   - Distance `d` from `prev` (0) to this station's position (3) is `3`.
   - `startFuel` becomes `3 - 3 = 0`. The car arrives at the station with no fuel left.
4. **Refuel if necessary**: Since `startFuel` is `0`, there's no need to refuel yet (no past stations to get fuel from).

5. **Push current station's fuel into the queue**: Push `-3` into priority queue `q`.

6. **Update `prev` position**: `prev` becomes `3`.

   Continuing to the next station:

   - Station at `7` miles offering `2` liters of fuel.
   - Distance `d` from `prev` (3) to this station's position (7) is `4`.
   - `startFuel` before arriving at this station is `0`, and after subtracting the distance it becomes `-4`, which means the car needs to refuel.
7. **Refuel if necessary**: Pop the largest amount of fuel from `q`, which is `-3` (representing `3` liters of fuel we passed earlier).

   - `startFuel` becomes `-4 + 3 = -1`. One stop has been made, so increment `ans` to `1`.
   - Since `startFuel` is still negative, we would need to stop if we had any more fuel in `q`, but since there's no more fuel in the queue, we continue.
8. **Push current station's fuel into the queue**: Push `-2` into priority queue `q`.

9. **Update `prev` position**: `prev` becomes `7`.

   Arriving at the destination (fake station at `10` miles):

   - Distance from `prev` (7) to destination (10) is `3`.
   - `startFuel` becomes `-1 - 3 = -4`, which requires another refuel.
10. **Refuel if necessary**: Pop the largest amount of fuel from `q`, which is now `-2` (representing `2` liters).

    - `startFuel` becomes `-4 + 2 = -2`. Another stop is made, increment `ans` to `2`.
    - `startFuel` is still negative, pop the last from `q`. However, there is no fuel left, meaning the car cannot reach the destination.
11. **Check if destination is reachable**: When attempting to refuel from an empty queue, the function should return `-1` as the car cannot reach the destination.

However, note that the example chose a scenario where we cannot reach the destination given the stations and starting fuel. If the stations provided more fuel or were positioned differently, the car could have reached the destination, and `ans` would represent the minimal number of stops needed to reach the target.

Python Solution

```python
from heapq import heappush, heappop
from typing import List

class Solution:
    def minRefuelStops(self, target: int, startFuel: int, stations: List[List[int]]) -> int:
        # Max-heap to store available fuel at stations we have passed
        fuel_maxheap = []

        # The position of the last station we processed
        previous_station_position = 0

        # The number of refueling stops we have made
        refuel_stops = 0

        # Add the target as a station to make sure we process the journey's end
        stations.append([target, 0])

        # Process each station on the route
        for position, fuel in stations:
            # Distance to the next station (or target)
            distance = position - previous_station_position
            startFuel -= distance  # Subtract the distance from the current fuel

            # Check if we need to refuel from passed stations (must take fuel from the station with the most fuel)
            while startFuel < 0 and fuel_maxheap:
                startFuel += -heappop(fuel_maxheap)  # Get fuel from the heap (invert the negative value)
                refuel_stops += 1  # Increment the refuel counter

            # If we cannot reach the next station/target and there is no more fuel in the heap, return -1
            if startFuel < 0:
                return -1

            # If the station offers fuel, add it to the heap (store as negative for max-heap)
            heappush(fuel_maxheap, -fuel)

            # Update the previous station position
            previous_station_position = position

        # Return the number of refueling stops after processing all stations
        return refuel_stops

```

Java Solution

```java
class Solution {
    public int minRefuelStops(int target, int startFuel, int[][] stations) {
        // Max heap to store available fuel amounts at the stations we've passed
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);
        int numStations = stations.length; // Total number of fuel stations
        int prevPosition = 0; // The previous position after the last fuel stop
        int numRefuels = 0; // The number of refueling stops made

        // Loop through all stations, adding a 'virtual' station at the target
        // to calculate the fuel needed to reach the target
        for (int i = 0; i <= numStations; i++) {
            // Distance from the previous station or from the start if it's the first station
            int distance = (i < numStations ? stations[i][0] : target) - prevPosition;
            // Subtract the distance from the current fuel
            startFuel -= distance;
            // Use fuel from the max heap (previously visited stations) if we are out of fuel
            while (startFuel < 0 && !maxHeap.isEmpty()) {
                // Refuel using the station with the most fuel we've passed
                startFuel += maxHeap.poll();
                numRefuels++; // Increment the refuel count
            }
            // If we can't refuel and the current fuel is less than zero, we can't reach the target
            if (startFuel < 0) {
                return -1;
            }
            // If it's not a virtual station, add the fuel from this station to the max heap
            if (i < numStations) {
                maxHeap.offer(stations[i][1]);
                // Update previous position to the current station's position
                prevPosition = stations[i][0];
            }
        }
        // Return the minimum number of refueling stops made to reach the target
        return numRefuels;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>
using namespace std;

class Solution {
public:
    // Method to find the minimum number of refueling stops required
    int minRefuelStops(int target, int startFuel, vector<vector<int>>& stations) {
        // Max heap to store the fuel available at the stations
        priority_queue<int> fuelMaxHeap;

        // Add a final station at the target position with 0 fuel
        stations.push_back({target, 0});

        // Initialize answer (minimum refueling stops) and previous station
        int minStops = 0, prevStationDist = 0;

        // Iterate through the stations along the route
        for (auto& station : stations) {
            // Calculate the distance to the current station from the previous station
            int distance = station[0] - prevStationDist;

            // Reduce the start fuel by the distance traveled
            startFuel -= distance;

            // While the fuel is not enough to reach the next station and there's fuel in the max heap, refuel
            while (startFuel < 0 && !fuelMaxHeap.empty()) {
                // Add the max fuel available from passed stations
                startFuel += fuelMaxHeap.top();
                fuelMaxHeap.pop();

                // Increment the minimum stops as we've refueled
                ++minStops;
            }

            // If the fuel is still not enough, return -1 indicating it is not possible
            if (startFuel < 0) {
                return -1;
            }

            // Add the current station's fuel to the max heap
            fuelMaxHeap.push(station[1]);

            // Update the previous station distance
            prevStationDist = station[0];
        }

        // Return the minimum number of stops to refuel
        return minStops;
    }
};

```

Typescript Solution

```typescript
// Importing array type for vectors.
import { PriorityQueue } from './PriorityQueue'; // Assuming a PriorityQueue implementation available

// Function to find the minimum number of refueling stops required.
// Takes the target distance, starting fuel level, and an array of fuel stations.
function minRefuelStops(target: number, startFuel: number, stations: number[][]): number {
    // Priority Queue (Max Heap) to store the fuel available at the stations.
    let fuelMaxHeap = new PriorityQueue<number>((a, b) => b - a);

    // Add a final station at the target position with 0 fuel.
    stations.push([target, 0]);

    // Initialize answer (minimum refueling stops) and previous station distance.
    let minStops = 0, prevStationDist = 0;

    // Iterate through the stations along the route.
    for (let station of stations) {
        // Calculate the distance to the current station from the previous station.
        let distance = station[0] - prevStationDist;

        // Reduce the start fuel by the distance traveled.
        startFuel -= distance;

        // While the fuel is not enough to reach the next station and there's fuel in the max heap, refuel.
        while (startFuel < 0 && !fuelMaxHeap.isEmpty()) {
            // Add the max fuel available from passed stations.
            startFuel += fuelMaxHeap.dequeue();

            // Increment the minimum stops as we've refueled.
            minStops += 1;
        }

        // If the fuel is still not enough, return -1 indicating it is not possible to reach the target.
        if (startFuel < 0) {
            return -1;
        }

        // Add the current station's fuel to the max heap.
        fuelMaxHeap.enqueue(station[1]);

        // Update the previous station distance.
        prevStationDist = station[0];
    }

    // Return the minimum number of stops to refuel.
    return minStops;
}

```

Time and Space Complexity

Time Complexity

The given code snippet involves iterating through the list of fuel stations once, which means the time complexity is at least `O(N)` where `N` is the number of fuel stations. However, since there are heap operations within the loop (specifically `heappop` and `heappush`), we have to consider their impact as well.

The worst-case time complexity for both `heappush` and `heappop` is `O(log M)`, where `M` is the number of elements in the heap. In the worst-case scenario, every station could potentially be added to the heap, so `M` can be at most `N`.

Since a `heappop` operation could potentially occur for each station in the list (thus iterating through the list `N` times), the worst-case time complexity becomes `O(N * log N)`.

To sum up, the total time complexity of the function is `O(N * log N)`.

Space Complexity

The space complexity of the function is primarily due to the usage of the priority queue (`q`). In the worst case, the priority queue could contain an entry for every station if no refueling was needed until the end, thus containing `N` elements. Hence, the space complexity is `O(N)` because that's the maximum amount of space that the priority queue will use. No other data structures in the solution use a significant amount of memory proportional to the input size, so they do not influence the space complexity beyond `O(N)`.