871. Minimum Number of Refueling Stops **Dynamic Programming** Heap (Priority Queue) Hard Greedy Array

Problem Description

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

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

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 of 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

minimize the number of stops, it shouldn't stop at every station to refuel. Instead, the car should only stop when it is necessary—

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

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 consideration of necessary pauses for refueling ensures the optimal number of stops. By initializing the stations array with the

position a. Subtract this distance from the startFuel to simulate driving to the station.

car cannot reach the next station or the destination, and the function returns -1.

destination), return ans as the total number of refueling stops made.

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

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

Station at 3 miles offering 3 liters of fuel.

Station at 7 miles offering 2 liters of fuel.

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

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.

minimal number of stops needed to reach the target.

previous_station_position = 0

stations.append([target, 0])

if startFuel < 0:</pre>

return -1

heappush(fuel_maxheap, -fuel)

Process each station on the route

for position, fuel in stations:

The position of the last station we processed

Distance to the next station (or target)

while startFuel < 0 and fuel_maxheap:</pre>

Update the previous station position

distance = position - previous_station_position

refuel_stops += 1 # Increment the refuel counter

public int minRefuelStops(int target, int startFuel, int[][] stations) {

int numRefuels = 0; // The number of refueling stops made

numRefuels++; // Increment the refuel count

// to calculate the fuel needed to reach the target

for (int i = 0; i <= numStations; i++) {</pre>

startFuel += maxHeap.poll();

maxHeap.offer(stations[i][1]);

prevPosition = stations[i][0];

// Return the minimum number of stops to refuel

// Function to find the minimum number of refueling stops required.

let fuelMaxHeap = new PriorityQueue<number>((a, b) => b - a);

// Add a final station at the target position with 0 fuel.

// Takes the target distance, starting fuel level, and an array of fuel stations.

// Priority Queue (Max Heap) to store the fuel available at the stations.

// Initialize answer (minimum refueling stops) and previous station distance.

// Calculate the distance to the current station from the previous station.

// While the fuel is not enough to reach the next station and there's fuel in the max heap, refuel.

// If the fuel is still not enough, return -1 indicating it is not possible to reach the target.

function minRefuelStops(target: number, startFuel: number, stations: number[][]): number {

import { PriorityQueue } from './PriorityQueue'; // Assuming a PriorityQueue implementation available

return minStops;

1 // Importing array type for vectors.

stations.push([target, 0]);

for (let station of stations) {

startFuel -= distance;

if (startFuel < 0) {</pre>

fuelMaxHeap.enqueue(station[1]);

prevStationDist = station[0];

// Update the previous station distance.

return -1;

let minStops = 0, prevStationDist = 0;

// Iterate through the stations along the route.

let distance = station[0] - prevStationDist;

// Reduce the start fuel by the distance traveled.

while (startFuel < 0 && !fuelMaxHeap.isEmpty()) {</pre>

// Add the current station's fuel to the max heap.

startFuel += fuelMaxHeap.dequeue();

// Add the max fuel available from passed stations.

Typescript Solution

if (startFuel < 0) {</pre>

if (i < numStations) {</pre>

return -1;

return numRefuels;

// Max heap to store available fuel amounts at the stations we've passed

PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

int prevPosition = 0; // The previous position after the last fuel stop

// Loop through all stations, adding a 'virtual' station at the target

// Refuel using the station with the most fuel we've passed

// Update previous position to the current station's position

// Return the minimum number of refueling stops made to reach the target

// If we can't refuel and the current fuel is less than zero, we can't reach the target

// If it's not a virtual station, add the fuel from this station to the max heap

int numStations = stations.length; // Total number of fuel stations

The number of refueling stops we have made

from heapq import heappush, heappop

refuel_stops = 0

Python Solution

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

9

10

11

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

37

46

47

48

49

50

52

9

10

11

12

13

14

15

16

17

20

21

22

23

26

27

33

34

35

36

37

38

39

40

42

51 };

36 }

from typing import List

Continuing to the next station:

refuel.

Distance d from prev (0) to this station's position (3) is 3.

Distance d from prev (3) to this station's position (7) is 4.

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

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

point initially, and ans is the count of refueling stops.

target position (and zero fuel), the loop also conveniently handles the case of reaching the destination. **Solution Approach**

The number of times fuel is taken from the priority queue represents the minimum number of refueling stops required.

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

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

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

3. Loop through stations: For each gas station in the stations list, calculate the distance d from the prev position to this station's

stations list with 0 fuel.

solution:

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).

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).

8. Return the number of stops: After the loop completes (which happens when the car reaches the 'fake' gas station at the

The algorithm effectively determines the minimal number of refueling stops by always picking the refuel options that gave the

5. Check if destination is reachable: After attempting to refuel from the priority queue, if startFuel is still negative, it means 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
- target distance to reach: 10 miles

• stations: [[3, 3], [7, 2]], where the first element in each pair is the position of the station, and the second element is the

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.

4. Refuel if necessary: Since startFuel is 0, there's no need to refuel yet (no past stations to get fuel from).

Given:

Approach:

startFuel: 3 liters

fuel available at that station.

startFuel before arriving at this station is 0, and after subtracting the distance it becomes -4, which means the car needs to

8. **Push current station's fuel into the queue**: Push -2 into priority queue q.

 \circ 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.

Add the target as a station to make sure we process the journey's end

startFuel -= distance # Subtract the distance from the current fuel

startFuel becomes 3 - 3 = 0. The car arrives at the station with no fuel left.

 \circ 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.

7. Refuel if necessary: Pop the largest amount of fuel from q, which is -3 (representing 3 liters of fuel we passed earlier).

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

11. Check if destination is reachable: When attempting to refuel from an empty queue, the function should return -1 as the car

startFuel is still negative, pop the last from q. However, there is no fuel left, meaning the car cannot reach the destination.

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 = []

37 previous_station_position = position 38 39 # Return the number of refueling stops after processing all stations 40 return refuel_stops 41

Check if we need to refuel from passed stations (must take fuel from the station with the most fuel)

startFuel += -heappop(fuel_maxheap) # Get fuel from the heap (invert the negative value)

If we cannot reach the next station/target and there is no more fuel in the heap, return -1

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

```
12
                // Distance from the previous station or from the start if it's the first station
                int distance = (i < numStations ? stations[i][0] : target) - prevPosition;</pre>
13
                // Subtract the distance from the current fuel
14
15
                startFuel -= distance;
                // Use fuel from the max heap (previously visited stations) if we are out of fuel
16
17
                while (startFuel < 0 && !maxHeap.isEmpty()) {</pre>
```

Java Solution

1 class Solution {

```
C++ Solution
 1 #include <vector>
 2 #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;
10
11
12
           // Add a final station at the target position with 0 fuel
            stations.push back({target, 0});
13
14
15
           // Initialize answer (minimum refueling stops) and previous station
16
           int minStops = 0, prevStationDist = 0;
17
18
           // Iterate through the stations along the route
19
            for (auto& station : stations) {
               // Calculate the distance to the current station from the previous station
20
21
                int distance = station[0] - prevStationDist;
22
               // Reduce the start fuel by the distance traveled
                startFuel -= distance;
24
25
26
               // While the fuel is not enough to reach the next station and there's fuel in the max heap, refuel
27
               while (startFuel < 0 && !fuelMaxHeap.empty()) {</pre>
28
                    // Add the max fuel available from passed stations
                    startFuel += fuelMaxHeap.top();
29
                    fuelMaxHeap.pop();
30
31
32
                    // Increment the minimum stops as we've refueled
33
                    ++minStops;
34
35
36
               // If the fuel is still not enough, return -1 indicating it is not possible
37
               if (startFuel < 0) {</pre>
38
                    return -1;
39
40
               // Add the current station's fuel to the max heap
                fuelMaxHeap.push(station[1]);
43
               // Update the previous station distance
44
                prevStationDist = station[0];
45
```

28 29 // Increment the minimum stops as we've refueled. 30 minStops += 1; 31 32

```
43
44
45
       // Return the minimum number of stops to refuel.
46
       return minStops;
47 }
48
<u>Time and Space Complexity</u>
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.
```

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

time complexity becomes O(N * log N).

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

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 0(N).

Since a heappop operation could potentially occur for each station in the list (thus iterating through the list N times), the worst-case