853. Car Fleet

Medium

Problem Description

Sorting

<u>Array</u>

Monotonic Stack

speed. Each index i in the arrays corresponds to the ith car, with position[i] representing the initial position of the car and speed[i] being its speed in miles per hour. The key condition is that cars cannot overtake each other. When a faster car catches up to a slower one, it will slow down to form a "car fleet" and they will move together at the slower car's speed. This also applies to multiple cars forming a single fleet if they meet at the same point. Our task is to determine the number of car fleets that will eventually arrive at the destination. A car fleet can consist of just one

The problem presents a scenario where n cars are traveling towards a destination at different speeds and starting from different

positions on a single-lane road. The destination is target miles away. We are provided with two arrays of integers: position and

car, or multiple cars if they have caught up with each other along the way. Even if a car or fleet catches up with another fleet at the destination point, they are counted as one fleet.

ntuition

To solve this problem, an intuitive approach is to figure out how long it would take for each car to reach the destination

can iterate from the one closest to the destination to the one furthest from it. This allows us to determine which cars will catch up to each other. As we iterate backwards, we calculate the time t it takes for each car to reach the destination: t = (target - position[i]) / speed[i]

independently and then see which cars would form fleets along the way. By sorting the cars based on their starting positions, we

before reaching the destination, and thus we increment our fleet count ans. The current car's time then becomes the new pre.

This process is repeated until we have considered all cars. The total number of fleets (ans) is the answer we are looking for. This method ensures that we count all fleets correctly, regardless of how many cars they start with or how many they pick up along the way.

We compare this time with the time of the previously considered car (starting from the closest to the target). If t is greater than

the pre (previous car's time), this means that the current car will not catch up to the car(s) ahead (or has formed a new fleet)

The solution is implemented in Python and is composed of the following steps: Sort Car Indices by Position: A list of indices is created from 0 to n-1, which is then sorted according to the starting positions of the cars. This sorting step uses the lambda function as the key for the sorted function to sort the indices based on their

associated values in the position array. The resulting idx list will help us traverse the cars in order of their starting positions

ans = pre = 0

speed (speed[i]).

fleet count ans by 1.

complexity is O(n) for storing the sorted indices.

Solution Approach

Initialize Variables: Two variables are used, ans to count the number of car fleets, and pre to store the time taken by the previously processed car (or fleet) to reach the destination.

idx = sorted(range(len(position)), key=lambda i: position[i])

from closest to furthest relative to the destination.

Reverse Iterate through Sorted Indices: By iterating over the sorted indices in reverse order, the algorithm evaluates each car starting with the one closest to the destination. for i in idx[::-1]: Calculate Time to Reach Destination: For each car, the time t to reach the destination is calculated using the formula:

This formula calculates the time by taking the distance to the destination (target - position[i]) and dividing it by the car's

Evaluate Fleets: If the calculated time t is greater than the time of the last car (or fleet) pre, it implies that the current car will

not catch up to any car ahead of it before reaching the destination. Hence, we have found a new fleet, and we increment the

t = (target - position[i]) / speed[i]

Imagine we have 4 cars with positions and speeds given by the following arrays:

if t > pre: ans += 1pre = t

The current time t is now the new pre because it will serve as the comparison time for the next car in the iteration.

Return Fleet Count: After all the iterations, the variable ans holds the total number of car fleets, and its value is returned.

of the algorithm is dominated by the sorting step, making the overall time complexity 0(n log n) due to the sort, and the space

The primary data structure used here is a list for indexing the cars. The sorting pattern is essential to correctly pair cars that will become fleets, and the reverse iteration allows the algorithm to efficiently compare only the necessary cars. The time complexity

return ans

Example Walkthrough

• position: [10, 8, 0, 5]

Car 4 (5 miles, 3 mph)

Car 3 (0 miles, 1 mph)

Following the solution approach:

• speed: [2, 4, 1, 3]

Let's apply the solution approach to a small example to better understand how it works.

 target: 12 miles away First, let's visualize the initial state of the cars and the target: Target (12 miles) Car 1 (10 miles, 2 mph)

Car 2 (8 miles, 4 mph)

Start

Since t < pre, we do not increment ans, and pre remains 12.

way to the target. Hence, there will be 1 car fleet by the time they reach the destination.

def carFleet(self, target: int, position: List[int], speed: List[int]) -> int:

Pair each car's position with its index and sort the pairs in ascending order of positions

car_indices_sorted_by_position = sorted(range(len(position)), key=lambda idx: position[idx])

Initialize the count of car fleets and the time of the previously counted fleet

Iterate over the cars from the one closest to the target to the furthest

Calculate the time needed for the current car to reach the target

If this time is greater than the time of the previously counted fleet,

it means this car cannot catch up with that fleet and forms a new fleet.

previous_time = time_to_reach_target # Update the time of the last fleet

for i in car_indices_sorted_by_position[::-1]: # Reverse iteration

// Function to count the number of car fleets that will arrive at the target

// Sort the indices based on the positions of the cars in descending order

// Calculate the time taken for the current car to reach the target

previousTime = timeToReach; // Update the previous time

double timeToReach = 1.0 * (target - positions[index]) / speeds[index];

// If the time is less or equal, it joins the fleet of the previous car

// If the time taken is greater than the previous time, it forms a new fleet

time_to_reach_target = (target - position[i]) / speed[i]

fleet_count += 1 # Increment fleet count

public int carFleet(int target, int[] positions, int[] speeds) {

// Populate the indices array with the array indices

Arrays.sort(indices, (a, b) -> positions[b] - positions[a]);

// The time taken by the previous car to reach the target

if time_to_reach_target > previous_time:

Return the total number of fleets

int carCount = positions.length;

// Array to hold the indices of the cars

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

Integer[] indices = new Integer[carCount];

// Iterate through the sorted indices array

if (timeToReach > previousTime) {

// Return the total number of fleets

idx = [0, 1, 3, 2]. Now the cars are sorted by proximity to the destination: Target (12 miles)

1. Sort Car Indices by Position: After sorting the indices by the starting positions in descending order, we have the new order of car indexes as

Initialize Variables: ans = 0, pre = 0 Reverse Iterate through Sorted Indices: We iterate in reverse through idx as [2, 3, 1, 0].

Start

Car 1 (idx = 0)

Car 2 (idx = 1)

Car 4 (idx = 3)

Car 3 (idx = 2)

For i = 2 (Car 3): \circ t = (12 - 0) / 1 = 12Since t > pre, we have ans = 1 and pre = 12.

For i = 3 (Car 4):

 \circ t = (12 - 8) / 4 = 1

 \circ t = (12 - 10) / 2 = 1

Solution Implementation

from typing import List

fleet_count = 0

previous_time = 0

// Number of cars

indices[i] = i;

// Count of car fleets

double previousTime = 0;

for (int index : indices) {

fleetCount++;

int fleetCount = 0;

class Solution:

class Solution {

For i = 0 (Car 1):

- \circ t = (12 5) / 3 \approx 2.33 Since t < pre, we do not increment ans, and pre remains 12. For i = 1 (Car 2):
- Since t < pre, we do not increment ans, and pre remains 12. **Return Fleet Count**: We have ans = 1, indicating that all the cars will form one fleet by the time they reach the destination.
- **Python**
- return fleet_count Java

In summary, even though the cars started at different positions and had different speeds, they caught up with each other on the

};

```
return fleetCount;
C++
#include <algorithm> // For sort and iota functions
#include <vector> // For using vectors
class Solution {
public:
   int carFleet(int target, vector<int>& positions, vector<int>& speeds) {
       int numCars = positions.size(); // Number of cars
       vector<int> indices(numCars); // Initialize a vector to store indices
        iota(indices.begin(), indices.end(), 0); // Filling the indices vector with 0, 1, ..., numCars - 1
       // Sort cars by their position in descending order, so the farthest is first
       sort(indices.begin(), indices.end(), [&](int i, int j) {
            return positions[i] > positions[j];
       });
        int fleetCount = 0; // Initializing count of car fleets
       double lastFleetTime = 0; // Time taken by the last fleet to reach the target
       // Iterate over the sorted cars by their initial position
        for (int idx : indices) {
           double timeToTarget = 1.0 * (target - positions[idx]) / speeds[idx]; // Calculate the time to target
           // If the current car's time to target is greater than the last fleet's time,
           // it becomes a new fleet as it cannot catch up to the one in front
           if (timeToTarget > lastFleetTime) {
                fleetCount++; // Increment the number of fleets
                lastFleetTime = timeToTarget; // And update the last fleet's time to this car's time
       return fleetCount; // Return total number of fleets
TypeScript
function carFleet(targetDistance: number, positions: number[], speeds: number[]): number {
   const numCars = positions.length; // Number of cars.
   // Create an array of indices corresponding to cars, sorted in descending order of their starting positions.
   const sortedIndices = Array.from({ length: numCars }, (_, index) => index)
```

.sort((indexA, indexB) => positions[indexB] - positions[indexA]);

let previousTimeToTarget = 0; // Time taken for the previous car (or fleet) to reach the target.

// If the current car takes longer to reach the target than the previous car (or fleet),

previousTimeToTarget = currentTimeToTarget; // Update the time to match the new fleet's time.

const currentTimeToTarget = (targetDistance - positions[index]) / speeds[index];

Pair each car's position with its index and sort the pairs in ascending order of positions car_indices_sorted_by_position = sorted(range(len(position)), key=lambda idx: position[idx]) # Initialize the count of car fleets and the time of the previously counted fleet fleet_count = 0 previous_time = 0

class Solution:

from typing import List

```
Time Complexity
  The time complexity of the code can be analyzed as follows:
```

if time_to_reach_target > previous_time:

Return the total number of fleets

return fleet_count

length of the positions list.

Space Complexity

Time and Space Complexity

let fleetCount = 0; // Counter for the number of car fleets.

// it cannot catch up and forms a new fleet.

if (currentTimeToTarget > previousTimeToTarget) {

fleetCount++; // Increment the fleet counter.

return fleetCount; // Return the total number of car fleets.

// Calculate time required for the current car to reach the target.

def carFleet(self, target: int, position: List[int], speed: List[int]) -> int:

Iterate over the cars from the one closest to the target to the furthest

Calculate the time needed for the current car to reach the target

If this time is greater than the time of the previously counted fleet,

it means this car cannot catch up with that fleet and forms a new fleet.

previous_time = time_to_reach_target # Update the time of the last fleet

caught up by another. The time complexity and space complexity of the code are analyzed below.

for i in car_indices_sorted_by_position[::-1]: # Reverse iteration

time_to_reach_target = (target - position[i]) / speed[i]

fleet_count += 1 # Increment fleet count

// Loop through the sorted car indices.

for (const index of sortedIndices) {

- Therefore, the overall time complexity of the function is dominated by the sorting operation and is O(N log N).
- The space complexity of the code can be analyzed as follows:
- Additional space is required for the sorted indices array idx, which will be of size O(N). Variables ans and pre use constant space: 0(1).

• The for loop iterates over the list of positions in reverse: This has a time complexity of O(N) as it goes through each car once.

The given Python code is to calculate the number of car fleets that will reach the target destination without any fleet getting

• Sorting the list of positions: This is done by using the <code>.sort()</code> method, which typically has a time complexity of <code>O(N log N)</code>, where N is the

• Space taken by sorting depends on the implementation, but for most sorting algorithms such as Timsort (used in Python's sort method), it

Combining the time and space complexities, we get the following results:

• Therefore, the overall space complexity of the function is O(N).

• Time Complexity: 0(N log N)

Space Complexity: 0(N)

requires O(N) space in the worst case.