853. Car Fleet

Medium | Stack | Array | Sorting | Monotonic Stack | Leetcode Link

Problem Description

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 `speed`. Each index `i` in the arrays corresponds to the `i`th 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 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.

Intuition

To solve this problem, an intuitive approach is to figure out how long it would take for each car to reach the destination independently and then see which cars would form fleets along the way. By sorting the cars based on their starting positions, we 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]`

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) 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.

Solution Approach

The solution is implemented in Python and is composed of the following steps:

1. **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 from closest to furthest relative to the destination.

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

2. **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.

   ```
   ans = pre = 0
   ```

3. **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]:
   ```

4. **Calculate Time to Reach Destination**: For each car, the time `t` to reach the destination is calculated using the formula:

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

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

5. **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 fleet count `ans` by 1.

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

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

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 of the algorithm is dominated by the sorting step, making the overall time complexity `O(n log n)` due to the sort, and the space complexity is `O(n)` for storing the sorted indices.

```
return ans
```

Example Walkthrough

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

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

- `position`: [10, 8, 0, 5]
- `speed`: [2, 4, 1, 3]
- `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)
|
| Car 4 (5 miles, 3 mph)
|
| Car 3 (0 miles, 1 mph)
|
Start
```

Following the solution approach:

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 `idx = [0, 1, 3, 2]`. Now the cars are sorted by proximity to the destination:

```
Target (12 miles)
|
| Car 1 (idx = 0)
|
| Car 2 (idx = 1)
|
| Car 4 (idx = 3)
|
| Car 3 (idx = 2)
|
Start
```

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

   For `i = 2` (Car 3):

   - `t = (12 - 0) / 1 = 12`
   - Since `t > pre`, we have `ans = 1` and `pre = 12`.

   For `i = 3` (Car 4):

   - `t = (12 - 5) / 3 ≈ 2.33`
   - Since `t < pre`, we do not increment `ans`, and `pre` remains `12`.

   For `i = 1` (Car 2):

   - `t = (12 - 8) / 4 = 1`
   - Since `t < pre`, we do not increment `ans`, and `pre` remains `12`.

   For `i = 0` (Car 1):

   - `t = (12 - 10) / 2 = 1`
   - Since `t < pre`, we do not increment `ans`, and `pre` remains `12`.
4. **Return Fleet Count**: We have `ans = 1`, indicating that all the cars will form one fleet by the time they reach the destination.

In summary, even though the cars started at different positions and had different speeds, they caught up with each other on the way to the target. Hence, there will be 1 car fleet by the time they reach the destination.

Python Solution

```python
from typing import List

class Solution:
    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
        fleet_count = 0
        previous_time = 0

        # Iterate over the cars from the one closest to the target to the furthest
        for i in car_indices_sorted_by_position[::-1]:  # Reverse iteration
            # Calculate the time needed for the current car to reach the target
            time_to_reach_target = (target - position[i]) / speed[i]
            # 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.
            if time_to_reach_target > previous_time:
                fleet_count += 1  # Increment fleet count
                previous_time = time_to_reach_target  # Update the time of the last fleet

        # Return the total number of fleets
        return fleet_count

```

Java Solution

```java
class Solution {

    // Function to count the number of car fleets that will arrive at the target
    public int carFleet(int target, int[] positions, int[] speeds) {
        // Number of cars
        int carCount = positions.length;
        // Array to hold the indices of the cars
        Integer[] indices = new Integer[carCount];

        // Populate the indices array with the array indices
        for (int i = 0; i < carCount; ++i) {
            indices[i] = i;
        }

        // Sort the indices based on the positions of the cars in descending order
        Arrays.sort(indices, (a, b) -> positions[b] - positions[a]);

        // Count of car fleets
        int fleetCount = 0;
        // The time taken by the previous car to reach the target
        double previousTime = 0;

        // Iterate through the sorted indices array
        for (int index : indices) {
            // Calculate the time taken for the current car to reach the target
            double timeToReach = 1.0 * (target - positions[index]) / speeds[index];

            // If the time taken is greater than the previous time, it forms a new fleet
            if (timeToReach > previousTime) {
                fleetCount++;
                previousTime = timeToReach; // Update the previous time
            }
            // If the time is less or equal, it joins the fleet of the previous car
        }
        // Return the total number of fleets
        return fleetCount;
    }
}

```

C++ Solution

```cpp
#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 Solution

```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 fleetCount = 0; // Counter for the number of car fleets.
    let previousTimeToTarget = 0; // Time taken for the previous car (or fleet) to reach the target.

    // Loop through the sorted car indices.
    for (const index of sortedIndices) {
        // Calculate time required for the current car to reach the target.
        const currentTimeToTarget = (targetDistance - positions[index]) / speeds[index];

        // If the current car takes longer to reach the target than the previous car (or fleet),
        // it cannot catch up and forms a new fleet.
        if (currentTimeToTarget > previousTimeToTarget) {
            fleetCount++; // Increment the fleet counter.
            previousTimeToTarget = currentTimeToTarget; // Update the time to match the new fleet's time.
        }
    }

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

```

Time and Space Complexity

The given Python code is to calculate the number of car fleets that will reach the target destination without any fleet getting caught up by another. The time complexity and space complexity of the code are analyzed below.

Time Complexity

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

- Sorting the list of positions: This is done by using the `.sort()` method, which typically has a time complexity of `O(N log N)`, where `N` is the length of the positions list.
- 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.
- Therefore, the overall time complexity of the function is dominated by the sorting operation and is `O(N log N)`.

Space Complexity

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: `O(1)`.
- Space taken by sorting depends on the implementation, but for most sorting algorithms such as Timsort (used in Python's sort method), it requires `O(N)` space in the worst case.
- Therefore, the overall space complexity of the function is `O(N)`.

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

- **Time Complexity:** `O(N log N)`
- **Space Complexity:** `O(N)`