1776. Car Fleet II

Hard Stack Array Math Monotonic Stack Heap (Priority Queue) Leetcode Link

Problem Description

The problem presents a scenario where `n` cars are moving on a single-lane road, all going the same direction. Each car has a starting position and speed, with higher-indexed cars being further down the road. The cars are simplified to points moving along a line, and when they collide, they form a single fleet with the speed of the slowest car involved in the collision. Our task is to determine the time each car takes to collide with the one in front, or -1 if no collision will occur. The answer needs to be within an error margin of `10^-5` of the actual value.

Intuition

To solve this problem, we use a stack to keep track of the cars whose collision times we need to calculate. We analyze the cars from the end of the array to the beginning, because a car's collision time will possibly affect only the cars behind it.

1. We initiate an empty stack and an array `ans` filled with -1, since initially, we assume that no car will collide with the car in front.
2. We iterate over the cars in reverse order:
   - If our stack is non-empty, we compare the current car's speed with the speed of the last car in the stack (which represents the nearest car in front with a collision time already computed, or yet to be computed).
   - If the speed of the current car is greater than the last car in the stack, a collision could occur. We then calculate the time it would take for the current car to collide with the last car in the stack.
   - If this collision time is less than or equal to the collision time of the last car in the stack (or the last car in the stack has no forthcoming collisions (`ans[j]` is -1)), we set this time as the answer for the current car's collision time, because it will collide before the car in the stack does or the car in the stack won't collide at all.
   - If the current car does not have a greater speed or if the collision time is greater than the collision time of the last car in the stack, we pop the last car from the stack because it will no longer collide with any previous cars.
3. After evaluating collision times, we add the index of the current car to the stack for the consideration of the following cars.

By the end of the iteration, the `ans` array will have the collision times for each car or -1 if no collision occurs.

Solution Approach

The solution uses a stack data structure to efficiently keep track of the cars and their potential collisions with the car directly in front of them. Below is a step-by-step walkthrough of the implementation detailing the algorithms, data structures, and patterns utilized:

1. **Initialization**: A stack `stk` is initialized to an empty list to keep track of car indices whose collision times need to be determined. An array `ans` is also initialized with `-1` for each of the `n` cars, indicating each car's collision time with the next car in front (initially set to -1 as we start assuming that there will be no collision).

```python
stk = []
n = len(cars)
ans = [-1] * n
```

2. **Processing Cars in Reverse**: The cars are processed from the end to the beginning to handle the propagation of collision times backward. This is done using a `for` loop that iterates in reverse:

```python
for i in range(n - 1, -1, -1):
```

3. **Collision Time Calculation**: Inside the loop, while there are still cars in the stack, the current car `i` is compared with the last car `j` in the stack, whether a collision is possible:

```python
j = stk[-1]
if cars[i][1] > cars[j][1]:
```

If `cars[i][1]`, the speed of the current car, is greater than that of car `j`, we calculate the time `t` at which car `i` would collide with car `j` using the relative speeds and positions:

```python
t = (cars[j][0] - cars[i][0]) / (cars[i][1] - cars[j][1])
```

4. **Potential Collision Validation**: Once a potential collision time `t` is calculated, we confirm if `t` is a valid collision time. The collision is considered valid if:
   - Car `j` has no collision (indicated by `ans[j]` which is `-1`), or
   - The collision with car `i` happens before car `j` would collide with another car.

```python
if ans[j] == -1 or t <= ans[j]:
    ans[i] = t
    break
```

If the collision time is not valid, the car `j` is removed from the stack as its collision time is no longer relevant to the cars behind it.

```python
stk.pop()
```

5. **Update Stack**: After processing potential collisions for car `i`, we add car `i` to the stack to potentially collide with the next cars.

```python
stk.append(i)
```

6. **Returning the Result**: After the loop completes, the `ans` array, which contains the collision times for each car or `-1` if the car doesn't collide, is finally returned:

```python
return ans
```

The use of the stack along with the reverse iteration of cars allows for efficient tracking and updating of collision times, ensuring each car's collision time is correctly calculated based on the conditions outlined in the problem.

Example Walkthrough

Let's use a small example with `n = 3` cars to illustrate the solution approach.

Suppose we have the following `cars` array, where each pair consists of the position and speed of the car:

```python
cars = [[3, 4], [2, 5], [1, 3]]
```

Here, the first car (index 2) is at position 1 with speed 3, the second car (index 1) is at position 2 with speed 5, and the third car (index 0) is at position 3 with speed 4.

Now, let's walk through the algorithm:

1. **Initialization**: We initialize the `stk` and `ans` arrays as empty and filled with `-1`, respectively.

```python
stk = []
ans = [-1, -1, -1]
```

2. **Processing Cars in Reverse**: We start with the last car (index 0) and move to the first car.

```python
for i in range(2, -1, -1):
```

3. **Collision Time Calculation**: Car index 0 has no one to collide with, so we add it to the stack and move to index 1.

For car index 1 (speed 5) and car index 0 in the stack (speed 4), calculate the time duration before a collision might occur:

```python
t = (3 - 2) / (5 - 4) = 1.0
```

Since car index 0 has no previous collision (`ans[0]` is `-1`), we update `ans[1]` with `t`:

```python
ans = [-1, 1.0, -1]
```

Car index 1 is then added to the stack.

4. **Potential Collision Validation**: Moving to car index 2 (speed 3), we check for a collision with car index 1 (speed 5). Since car index 2 is slower, no update to `ans` is needed, and car index 2 is not added to the stack, as it will never catch up. The last stack pop operation:

```python
stk.pop()
```

5. **Update Stack**: After processing potential collisions for car index 2, we add car index 2 to the stack, as it is still under consideration for car index 1:

```python
stk.append(2)
```

6. **Returning the Result**: The loop completes, and we return `ans`:

```python
return [-1, 1.0, -1]
```

This indicates that the car at index 1 takes 1.0 time units to collide with the car at index 0, and the other two cars do not collide with the car in front.

This example illustrates the algorithm's approach, efficiently determining the collision times of cars that are simply moving points on a line.

Python Solution

```python
from typing import List

class Solution:
    def get_collision_times(self, cars: List[List[int]]) -> List[float]:
        # Stack to keep track of cars indices that we're examining
        stack = []
        # The number of cars
        num_cars = len(cars)
        # Answer list initialized with -1, indicating
        # that initially we assume no collisions for each car
        collision_times = [-1] * num_cars

        # We iterate from the last car to the first car
        for i in range(num_cars - 1, -1, -1):
            # Resolve collisions in stack
            while stack:
                # Get the index of the last car in the stack
                j = stack[-1]
                # Check if the current car is faster than the car at the top of the stack
                if cars[i][1] > cars[j][1]:
                    # Calculate time taken to collide with the car in front
                    time_to_collide = (cars[j][0] - cars[i][0]) / (cars[i][1] - cars[j][1])
                    # If the car ahead has not collided or will not collide
                    # sooner than it takes for i to reach it, then we record this
                    # collision time for car i and stop looking for collisions
                    if collision_times[j] == -1 or time_to_collide <= collision_times[j]:
                        collision_times[i] = time_to_collide
                        break
                # If we do not find a collision or the current car is slower
                # than the last car in the stack, we should remove the last car
                # from consideration
                stack.pop()

            # Add the index of this car into the stack for potential
            # collision calculations with further preceding cars
            stack.append(i)

        # Return the list of times when each car would collide
        return collision_times

# Example of usage:
# sol = Solution()
# print(sol.get_collision_times([[1, 2], [2, 1], [4, 3]]))

```

Java Solution

```java
class Solution {
    public double[] getCollisionTimes(int[][] cars) {
        int numCars = cars.length; // Number of cars
        double[] collisionTimes = new double[numCars]; // Array to store the collision times
        // Initialize collision times with -1.0, indicating no collision
        Arrays.fill(collisionTimes, -1.0);
        Deque<Integer> stack = new ArrayDeque<>(); // Stack to keep track of cars that have not collided yet

        // Traverse the cars array in reverse
        for (int i = numCars - 1; i >= 0; --i) {
            // Keep checking cars until we find a collision or run out of cars to check
            while (!stack.isEmpty()) {
                int nextCarIndex = stack.peek(); // Get the index of the next car in the stack

                // If the current car is faster than the next car, calculate the collision time
                if (cars[i][1] > cars[nextCarIndex][1]) {
                    double timeUntilCollision = (double)(cars[nextCarIndex][0] - cars[i][0]) / (cars[i][1] - cars[nextCarIndex][1]);
                    // Only record the collision if it happens before the next car collides with another car
                    if (collisionTimes[nextCarIndex] < 0 || timeUntilCollision <= collisionTimes[nextCarIndex]) {
                        collisionTimes[i] = timeUntilCollision;
                        break;
                    }
                }
                stack.pop(); // Remove the next car from the stack as there won't be a collision with the current car
            }
            stack.push(i); // Add the current car to the stack
        }
        return collisionTimes; // Return the array containing the collision times for each car
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <stack>

class Solution {
public:
    vector<double> getCollisionTimes(vector<vector<int>>& cars) {
        int numCars = cars.size();                       // numCars holds the number of cars
        vector<double> collisionTimes(numCars, -1.0);  // Pre-fill the result with -1.0, indicating no collision
        stack<int> carIndices;                         // Stack to hold indices of cars which might collide

        // Iterate over the cars starting from the last one to the first
        for (int currentCar = numCars - 1; currentCar >= 0; --currentCar) {
            // Continue checking for potential collisions
            while (!carIndices.empty()) {
                int leadingCar = carIndices.top(); // Index of the leading car

                // Check if the current car is faster than the leading car
                if (cars[currentCar][1] > cars[leadingCar][1]) {
                    // Calculate the time of the collision
                    double timeToCollision = static_cast<double>(cars[leadingCar][0] - cars[currentCar][0]) /
                        (cars[currentCar][1] - cars[leadingCar][1]);

                    // If the leading car has no other collision or the calculated collision
                    // happens before the leading car's collision, update the result for the current car
                    if (collisionTimes[leadingCar] < 0 || timeToCollision <= collisionTimes[leadingCar]) {
                        collisionTimes[currentCar] = timeToCollision;
                        break;
                    }
                }
                // If the current car will not collide with the leading car, pop it from the stack
                carIndices.pop();
            }
            // Push the current car index into the stack for the future potential collisions
            carIndices.push(currentCar);
        }
        return collisionTimes; // Return the times at which each car will collide
    }
};

```

Typescript Solution

```typescript
// Define the car type as an array with two elements: position and speed of the car.
type Car = [number, number];

// Calculate collision times for an array of cars given their positions and speeds.
function getCollisionTimes(cars: Car[]): number[] {
    const numCars: number = cars.length;                        // The number of cars
    const collisionTimes: number[] = new Array(numCars).fill(-1.0); // Initialize collision times with -1.0 to represent no collisi
    const carIndices: number[] = [];                           // Stack to hold the indices of cars which might collide

    // Iterate through the cars from the last one to the first.
    for (let currentCar = numCars - 1; currentCar >= 0; --currentCar) {
        // While there are cars that may potentially collide.
        while (carIndices.length > 0) {
            const leadingCar = carIndices[carIndices.length - 1]; // Get the index of the car in front

            // If the current car is faster than the car in front, check for potential collision.
            if (cars[currentCar][1] > cars[leadingCar][1]) {
                // Calculate the time of collision between the current car and the car in front.
                const timeToCollision: number = (cars[leadingCar][0] - cars[currentCar][0]) /
                                                (cars[currentCar][1] - cars[leadingCar][1]);

                // Check if this is a valid collision time (before any collision the leading car might have).
                if (collisionTimes[leadingCar] === -1.0 || timeToCollision <= collisionTimes[leadingCar]) {
                    collisionTimes[currentCar] = timeToCollision;
                    break;
                }
            }

            // The current car won't collide with the leading car (either slower or collides later), remove leading car index from
            carIndices.pop();
        }

        // Add the current car index onto the stack to be considered for future collisions.
        carIndices.push(currentCar);
    }

    return collisionTimes; // Return the calculated times at which each car will collide
}

// Example usage:
// const cars: Car[] = [[1, 2], [2, 1], [4, 3]];
// const result = getCollisionTimes(cars);
// console.log(result); // This would print out the collision times for each car.

```

Time and Space Complexity

The given Python code uses a decreasing stack to solve the problem of finding the collision times of cars on a single lane road where each car is represented as a (position, speed) pair.

Time Complexity: The time complexity of the code is `O(n)`, where `n` is the number of cars. Each car is processed exactly once when iterating from the end of the list to the beginning. When a new car is processed, it either (1) becomes the new stack top and potentially removes several cars from the stack, or (2) finds a collision time and does not affect the stack. Since each car is pushed onto the stack and popped from the stack at most once, the total operations on the stack are linear with respect to the number of cars.

Space Complexity: The space complexity is also `O(n)`. The stack used in the algorithm can potentially store all the cars simultaneously if none of the cars is able to collide with the car in front of it (i.e., each car is slower than the one in front of it). The `ans` list, which stores collision times for each car, also takes up `n` space.

In summary, the algorithm efficiently processes each car exactly once and manages collisions using a stack structure that maintains cars that have the potential to collide in the future.