Problem 1776: Car Fleet II

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

There are

n

cars traveling at different speeds in the same direction along a one-lane road. You are given an array

cars

of length

n

, where

cars[i] = [position

i

, speed

i

]

represents:

position

i

is the distance between the

i

th

car and the beginning of the road in meters. It is guaranteed that

position

i

< position

i+1

.

speed

i

is the initial speed of the

i

th

car in meters per second.

For simplicity, cars can be considered as points moving along the number line. Two cars collide when they occupy the same position. Once a car collides with another car, they unite and form a single car fleet. The cars in the formed fleet will have the same position and the same speed, which is the initial speed of the slowest

car in the fleet.

Return an array

answer

, where

answer[i]

is the time, in seconds, at which the

i

th

car collides with the next car, or

-1

if the car does not collide with the next car. Answers within

10

-5

of the actual answers are accepted.

Example 1:

Input:

cars = [[1,2],[2,1],[4,3],[7,2]]

Output:

[1.00000,-1.00000,3.00000,-1.00000]

Explanation:

After exactly one second, the first car will collide with the second car, and form a car fleet with speed 1 m/s. After exactly 3 seconds, the third car will collide with the fourth car, and form a car fleet with speed 2 m/s.

Example 2:

Input:

cars = [[3,4],[5,4],[6,3],[9,1]]

Output:

[2.00000,1.00000,1.50000,-1.00000]

Constraints:

1 <= cars.length <= 10

5

1 <= position

i

, speed

i

<= 10

6

position

i

< position

i+1

Code Snippets

C++:

```
class Solution {
public:
vector<double> getCollisionTimes(vector<vector<int>>& cars) {

}
};
```

Java:

```
class Solution {
public double[] getCollisionTimes(int[][] cars) {

}
}
```

Python3:

```
class Solution:
def getCollisionTimes(self, cars: List[List[int]]) -> List[float]:
```

Python:

```
class Solution(object):
def getCollisionTimes(self, cars):
"""
:type cars: List[List[int]]
:rtype: List[float]
"""
```

JavaScript:

```
/**
* @param {number[][]} cars
* @return {number[]}
*/
```

```
var getCollisionTimes = function(cars) {

};
```

TypeScript:

```
function getCollisionTimes(cars: number[][]): number[] {

};
```

C#:

```
public class Solution {
public double[] GetCollisionTimes(int[][] cars) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
double* getCollisionTimes(int** cars, int carsSize, int* carsColSize, int*
returnSize) {

}
```

Go:

```
func getCollisionTimes(cars [][]int) []float64 {

}
```

Kotlin:

```
class Solution {
fun getCollisionTimes(cars: Array<IntArray>): DoubleArray {

}
}
```

Swift:

```
class Solution {
func getCollisionTimes(_ cars: [[Int]]) -> [Double] {

}
}
```

Rust:

```
impl Solution {
pub fn get_collision_times(cars: Vec<Vec<i32>>) -> Vec<f64> {

}
}
```

Ruby:

```
# @param {Integer[][]} cars
# @return {Float[]}
def get_collision_times(cars)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $cars
* @return Float[]
*/
function getCollisionTimes($cars) {

}
}
```

Dart:

```
class Solution {
List<double> getCollisionTimes(List<List<int>> cars) {

}
}
```

Scala:

```
object Solution {
def getCollisionTimes(cars: Array[Array[Int]]): Array[Double] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec get_collision_times(cars :: [[integer]]) :: [float]
def get_collision_times(cars) do

end
end
```

Erlang:

```
-spec get_collision_times(Cars :: [[integer()]]) -> [float()].
get_collision_times(Cars) ->
.
```

Racket:

```
(define/contract (get-collision-times cars)
(-> (listof (listof exact-integer?)) (listof flonum?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Car Fleet II
* Difficulty: Hard
* Tags: array, math, stack, queue, heap
*
```

```
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
vector<double> getCollisionTimes(vector<vector<int>>& cars) {

}
};
```

Java Solution:

```
/**
* Problem: Car Fleet II
* Difficulty: Hard
* Tags: array, math, stack, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public double[] getCollisionTimes(int[][] cars) {

}
}
```

Python3 Solution:

```
"""
Problem: Car Fleet II
Difficulty: Hard
Tags: array, math, stack, queue, heap

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""
```

```
class Solution:
def getCollisionTimes(self, cars: List[List[int]]) -> List[float]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def getCollisionTimes(self, cars):
"""
:type cars: List[List[int]]
:rtype: List[float]
"""
```

JavaScript Solution:

```
/**
* Problem: Car Fleet II
* Difficulty: Hard
* Tags: array, math, stack, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[][]} cars
* @return {number[]}
*/
var getCollisionTimes = function(cars) {

};
```

TypeScript Solution:

```
/**
* Problem: Car Fleet II
* Difficulty: Hard
* Tags: array, math, stack, queue, heap
```

```
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function getCollisionTimes(cars: number[][]): number[] {

};
```

C# Solution:

```
/*
* Problem: Car Fleet II
* Difficulty: Hard
* Tags: array, math, stack, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public double[] GetCollisionTimes(int[][] cars) {

}
}
```

C Solution:

```
/*
* Problem: Car Fleet II
* Difficulty: Hard
* Tags: array, math, stack, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
```

```
* Note: The returned array must be malloced, assume caller calls free().
*/
double* getCollisionTimes(int** cars, int carsSize, int* carsColSize, int*
returnSize) {

}
```

Go Solution:

```
// Problem: Car Fleet II
// Difficulty: Hard
// Tags: array, math, stack, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func getCollisionTimes(cars [][]int) []float64 {

}
```

Kotlin Solution:

```
class Solution {
fun getCollisionTimes(cars: Array<IntArray>): DoubleArray {

}
}
```

Swift Solution:

```
class Solution {
func getCollisionTimes(_ cars: [[Int]]) -> [Double] {

}
}
```

Rust Solution:

```
// Problem: Car Fleet II
// Difficulty: Hard
// Tags: array, math, stack, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn get_collision_times(cars: Vec<Vec<i32>>) -> Vec<f64> {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} cars
# @return {Float[]}
def get_collision_times(cars)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $cars
* @return Float[]
*/
function getCollisionTimes($cars) {

}
}
```

Dart Solution:

```
class Solution {
List<double> getCollisionTimes(List<List<int>> cars) {

}
}
```

Scala Solution:

```
object Solution {
def getCollisionTimes(cars: Array[Array[Int]]): Array[Double] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec get_collision_times(cars :: [[integer]]) :: [float]
def get_collision_times(cars) do

end
end
```

Erlang Solution:

```
-spec get_collision_times(Cars :: [[integer()]]) -> [float()].
get_collision_times(Cars) ->
.
```

Racket Solution:

```
(define/contract (get-collision-times cars)
(-> (listof (listof exact-integer?)) (listof flonum?))
)
```