<https://leetcode.com/problems/car-fleet-ii/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] = [positioni, speedi] represents:

positioni is the distance between the ith car and the beginning of the road in meters. It is guaranteed that positioni < positioni+1.

speedi is the initial speed of the ith 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 ith 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 <= positioni, speedi <= 10^6

positioni < positioni+1

**Attempt 1: 2024-09-28**

**Solution 1: Monotonic Increasing Stack (180 min)**

**Base on reverse order travesal from last to first car, the Monotonic Increasing Stack build up based on car fleet's speed perspective, since only later car (current car) with higher speed can catch up previous car (next car) with lower speed.**

lass Solution {

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

        int n = cars.length;

        // Array to store the collision times

        double[] result = new double[n];

        // Initialize collision times with -1.0, indicating no collision

        Arrays.fill(result, -1.0);

        // Stack to keep track of cars that have not collided yet

        Stack<Integer> stack = new Stack<>();

        // Traverse the cars array in reverse

        for(int i = n - 1; i >= 0; i--) {

            // Keep checking cars until we find a collision or run out of cars to check

            while(!stack.isEmpty()) {

                int j = stack.peek();

                int nextCarPosition = cars[j][0];

                int nextCarSpeed = cars[j][1];

                int curCarPosition = cars[i][0];

                int curCarSpeed = cars[i][1];

                // If the current car is faster than the next car, calculate the collision time

                if(curCarSpeed > nextCarSpeed) {

                    double collisionTime = (double)(nextCarPosition - curCarPosition) / (curCarSpeed - nextCarSpeed);

                    // Only record the collision if it happens before the next car collides with another car

                    if(result[j] == -1.0 || collisionTime <= result[j]) {

                        result[i] = collisionTime;

                        // Current car handling done, no need pop out next car's index

                        // from stack, directly move on to immediate previous car

                        break;

                    }

                }

                // If the current car is slower or the time of collision with the

                // next car is not valid, remove the next car from the stack as

                // there won't be a collision with the current car(pop next car

                // index from stack)

                stack.pop();

            }

            // Add the current car to the stack

            stack.push(i);

        }

        return result;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**We can use below 3 different inputs to simulate different scenarios:**

int[][] cars = new int[][]{{1,2},{2,1},{4,3},{7,2}};

int[][] cars = new int[][]{{1,2},{2,4},{4,3},{7,2}};

int[][] cars = new int[][]{{1,3},{2,4},{4,3},{7,2}};

**Refer to**

<https://algo.monster/liteproblems/1776>

**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.**

**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.**

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.**

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:

**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).

stk = []

n = len(cars)

ans = [-1] \* n

**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:

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

**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:

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:

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

**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.

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.

stk.pop()

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

stk.append(i)

**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:

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:

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:

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

stk = []

ans = [-1, -1, -1]

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

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

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:

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:

ans = [-1, 1.0, -1]

Car index 1 is then added to the stack.

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:

stk.pop()

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:

stk.append(2)

Returning the Result: The loop completes, and we return ans:

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.

**Refer to chatGPT**

**Problem Statement:**

There are n cars, and each car has a position and speed. All the cars are traveling in the same direction (increasing position). If two cars collide, they form a fleet, and the leading car will slow down to match the speed of the trailing car. The trailing car will never accelerate.

You are required to return an array of length n, where the i-th element is the time taken for car i to collide with another car or -1 if the car never collides.

**Approach:**

To solve this, we need to calculate the time of collision between each car and the car in front of it. We can do this using a **Monotonic Stack** (like Car Fleet I) to track which cars will collide and when.

**Key Ideas:**

If car i is moving faster than car i+1, they will eventually collide, and we can calculate the time of the collision.

We use a **stack** to store cars and their potential collision times.

When car i catches up with car j, we calculate the time of collision and propagate that time forward.

class Solution {

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

int n = cars.length;

double[] result = new double[n]; // Result array to store collision times

Arrays.fill(result, -1); // Initialize the result array with -1 (for cars that never collide)

Stack<Integer> stack = new Stack<>(); // Stack to store indices of cars

// Process cars from the second-to-last to the first car

for (int i = n - 2; i >= 0; i--) {

int pos = cars[i][0]; // Position of the current car

int speed = cars[i][1]; // Speed of the current car

// Process the stack to determine the earliest collision for car i

while (!stack.isEmpty()) {

int j = stack.peek();

int nextPos = cars[j][0]; // Position of the next car

int nextSpeed = cars[j][1]; // Speed of the next car

// If the current car is faster, compute the time of collision

if (speed > nextSpeed) {

double collisionTime = (double)(nextPos - pos) / (speed - nextSpeed);

// Check if this collision happens before the next car collides with another car

if (result[j] == -1 || collisionTime <= result[j]) {

result[i] = collisionTime;

break; // We found the first collision for car i

}

}

// If the current car is slower or the time of collision with the next car is not valid, pop the stack

stack.pop();

}

// Push the current car onto the stack for future comparisons

stack.push(i);

}

return result;

}

}

**Explanation:**

Initialization:

We initialize a result array result[] of size n, where each value is set to -1. This means that, by default, cars never collide unless we find a valid collision.

We use a stack to keep track of the cars that have not yet collided.

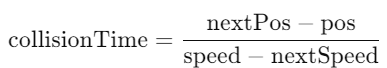
Traverse from the Second-to-Last Car to the First:

We traverse the cars from the second-last car to the first car (i.e., from right to left in the array). This way, we can evaluate potential collisions between the current car and cars in front of it.

Collision Calculation:

For each car, we check the car in front of it (the car on top of the stack).

If the current car is faster than the car in front, we compute the time of collision using the formula:



where pos and speed are for the current car, and nextPos and nextSpeed are for the car in front.

If this collision time is less than or equal to the previously calculated collision time for the car in front (or if the car in front never collides), then we record this time as the first collision for the current car.

If the current car is slower than the car in front or if the collision time is invalid, we pop the car from the stack.

Stack Push:

After processing the car, we push its index onto the stack so that future cars can use it for collision comparisons.

Return the Result:

After processing all the cars, the result array contains the collision times for each car, or -1 if the car never collides.

**Time Complexity:**

O(n): We only iterate through the list of cars once, and each car is pushed and popped from the stack at most once. This makes the time complexity linear.

**Space Complexity:**

O(n): The stack and result array both take linear space with respect to the number of cars.

**Example Walkthrough:**

For the input:

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

Car 3 (at position 7, speed 2) never collides, so the result for this car remains -1.

Car 2 (at position 4, speed 3) will catch up to car 3 in 3 units of time. Thus, result[2] = 3.00000.

Car 1 (at position 2, speed 1) never collides, so the result for this car remains -1.

Car 0 (at position 1, speed 2) will collide with car 1 in 1 unit of time. Thus, result[0] = 1.00000.

The final result array is:

[1.00000, -1.00000, 3.00000, -1.00000]

**Refer to**

[L853.Car Fleet (Ref.L402,L84,L1673,L1776)](note://WEBb3acaa13fcf7f22af84a1298449f9a43)

[L84.Largest Rectangle in Histogram](note://E2FC22D9715540C5A8F06F60FBBE886F)

[L402.Remove K Digits (Ref.L1673,L84)](note://09B83C91043B44D5BC818F72BEE20307)

[L1673.Find the Most Competitive Subsequence (Ref.L84,L402)](note://WEBd8e85b624521547b3108241deb26a6cb)