<https://leetcode.com/problems/car-fleet/description/>

There are n cars at given miles away from the starting mile 0, traveling to reach the mile target.

You are given two integer array position and speed, both of length n, where position[i] is the starting mile of the ith car and speed[i] is the speed of the ith car in miles per hour.

A car cannot pass another car, but it can catch up and then travel next to it at the speed of the slower car.

A car fleet is a car or cars driving next to each other. The speed of the car fleet is the minimum speed of any car in the fleet.

If a car catches up to a car fleet at the mile target, it will still be considered as part of the car fleet.

Return the number of car fleets that will arrive at the destination.

**Example 1:**

**Input:** target = 12, position = [10,8,0,5,3], speed = [2,4,1,1,3]

**Output:** 3

**Explanation:**

The cars starting at 10 (speed 2) and 8 (speed 4) become a fleet, meeting each other at 12. The fleet forms at target.

The car starting at 0 (speed 1) does not catch up to any other car, so it is a fleet by itself.

The cars starting at 5 (speed 1) and 3 (speed 3) become a fleet, meeting each other at 6. The fleet moves at speed 1 until it reaches target.

**Example 2:**

**Input:** target = 10, position = [3], speed = [3]

**Output:** 1

**Explanation:**There is only one car, hence there is only one fleet.

**Example 3:**

**Input:** target = 100, position = [0,2,4], speed = [4,2,1]

**Output:** 1

**Explanation:**

The cars starting at 0 (speed 4) and 2 (speed 2) become a fleet, meeting each other at 4. The car starting at 4 (speed 1) travels to 5.

Then, the fleet at 4 (speed 2) and the car at position 5 (speed 1) become one fleet, meeting each other at 6. The fleet moves at speed 1 until it reaches target.

**Constraints:**

n == position.length == speed.length

1 <= n <= 10^5

0 < target <= 10^6

0 <= position[i] < target

All the values of position are **unique**.

0 < speed[i] <= 10^6

**本题的基本思路如下：**

**对每辆车的位置 positions[i] 按照逆序排序 (positions[i] 值大的排在前面)，为何是逆序呢？因为我们实际需要的是按顺序排序的 (target - positions[i])，物理含义是哪辆车离 target 近，哪辆车在排序的起始顺序上就排在前面，这样离 target 远的车起始顺序上排在后面，但是能依靠更快的速度到达 target 的话，就能体现物理意义上的超车，然而根据题目的定义，超车并不会发生，而是和前面的慢车融合为一个车队后以慢车的速度行驶到 target，当然，如果离 target 远的车在到达 target 之前一直赶不上前车就只能单独形成一个车队，这样才有了题目需要求的到达 target 的时候有多少个车队，而对于 stack 的使用则偏重于存储每个车队 (可以是一辆车) 到达 target 所需的时间，第一个车队和后面赶不上的车队就会在 stack 上占用一个位置，最终 stack 的大小就是车队的数量，而这个 stack 则适用于 Monotonic Increasing Stack，意味着后续赶不上的车队到达的时间必然大于前面的车队，反过来说，一旦能赶上的车队就会和前面的车队融合，所以反映在 stack 上存储的车队到达 target 的时间是单调递增的**

**Attempt 1: 2024-09-27**

**Solution 1: Sorting + Monotonic Increasing Stack (60min)**

**Style 1: Create 2D 'cars' array**

class Solution {

    public int carFleet(int target, int[] position, int[] speed) {

        int n = position.length;

        // Step 1: Create an array of cars with position and time to destination

        double[][] cars = new double[n][2];

        for(int i = 0; i < n; i++) {

            cars[i][0] = position[i]; // Position of the car

            cars[i][1] = (double) (target - position[i]) / speed[i]; // Time to reach the target

        }

        // Step 2: Sort cars by their position in descending order (farthest position away from target first)

        Arrays.sort(cars, Comparator.comparingDouble(car -> -car[0]));

        // Step 3: Use a real stack to count fleets

        int fleet = 0;

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

        // Traverse cars from the closest position to the target

        for(int i = 0; i < n; i++) {

            // If the stack is empty or the current car takes more time than the car in front, form a new fleet

            if(stack.isEmpty() || stack.peek() < cars[i][1]) {

                stack.push(cars[i][1]); // Push the current car's time onto the stack

            }

        }

        // The number of fleets is the size of the stack

        return stack.size();

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(n)

**Refer to chatGPT**

**Idea behind the Monotonic Stack approach:**

The idea of using a stack is based on the fact that we want to process the cars from the closest position to the target to the furthest position. This allows us to track which cars form a fleet as we go.

A **Monotonic Increasing Stack** will help us keep track of the time it takes for cars to reach the target. If a car takes longer than the car in front of it (closer to the target), it forms a new fleet. If it takes less or equal time, it joins the fleet of the car in front.

**Approach:**

Sort cars based on their **positions**.

Traverse the sorted cars from the one closest to the target to the furthest.

For each car, compute the **time** it will take to reach the target.

Use a stack to store these times:

If the current car takes more time than the car at the top of the stack, it forms a new fleet.

If the current car takes less or the same time, it merges with the fleet of the car in front.

**Monotonic Stack Solution in Java:**

class Solution {

public int carFleet(int target, int[] position, int[] speed) {

int n = position.length;

if (n == 0) return 0;

// Step 1: Create an array of cars with position and time to destination

double[][] cars = new double[n][2];

for (int i = 0; i < n; i++) {

cars[i][0] = position[i]; // Position of the car

cars[i][1] = (double) (target - position[i]) / speed[i]; // Time to reach the target

}

// Step 2: Sort cars by their position in descending order (farthest position away from target first)

Arrays.sort(cars, Comparator.comparingDouble(car -> -car[0]));

// Step 3: Use a real stack to count fleets

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

// Traverse cars from the closest position to the target

for (int i = 0; i < n; i++) {

double currentTime = cars[i][1];

// If the stack is empty or the current car takes more time than the car in front, form a new fleet

if (stack.isEmpty() || currentTime > stack.peek()) {

stack.push(currentTime); // Push the current car's time onto the stack

}

}

// The number of fleets is the size of the stack

return stack.size();

}

}

**Explanation of the Real Stack Solution:**

Sorting the Cars:

First, we calculate the time it will take for each car to reach the target, and we sort the cars based on their position in descending order (farthest cars come first).

Stack Operations:

We then use a stack to store the time it takes for each car to reach the target. Starting from the furthest car, we check whether the current car forms a new fleet.

If the stack is empty or the current car’s time is greater than the time on the top of the stack (i.e., the car can't catch up), we push this car’s time onto the stack.

If the current car’s time is less than or equal to the car on top of the stack, it joins the fleet with the car on top, so we don’t push its time.

Fleet Count:

At the end of the traversal, the size of the stack gives the number of distinct car fleets. Each element in the stack represents a fleet that will reach the destination at a different time.

**Refer to**

<https://leetcode.com/problems/car-fleet/solutions/3897504/monotonic-stack/>

**Intuition**

Imagine cars racing towards a finish line at different speeds. By considering their positions and speeds, you can sort the cars and calculate when each one reaches the finish line. Using a stack to track their times, you identify groups of cars that arrive together as fleets. The size of the stack then represents the count of car fleets.

Key Insight: Sorting cars by positions and calculating their arrival times helps find fleet formations efficiently..

**Approach**

Implement a monotonic stack.

**Complexity**

Time complexity: O(NlogN)

Space complexity: O(N)

**Code**

class Solution {

public int carFleet(int target, int[] position, int[] speed) {

if (position.length == 1) return 1;

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

int[][] posAndSpeed = new int[position.length][2];

for (int i = 0; i < position.length; i++) {

posAndSpeed[i][0] = position[i];

posAndSpeed[i][1] = speed[i];

}

Arrays.sort(posAndSpeed, java.util.Comparator.comparingInt(o -> o[0]));

for (int i = posAndSpeed.length - 1; i >= 0; i--) {

double currentTime = (double) (target - posAndSpeed[i][0]) / posAndSpeed[i][1];

if (!stack.isEmpty() && currentTime <= stack.peek()) {

continue;

} else {

stack.push(currentTime);

}

}

return stack.size();

}

}

**Style 2: No need create 2D 'cars' array**

class Solution {

    public int carFleet(int target, int[] position, int[] speed) {

        int n = position.length;

        // Must define 'indexes' as 'Integer' instead of 'int',

        // otherwise Arrays.sort() not able to work

// https://leetcode.com/discuss/feedback/2183318/lambda-in-java-sort-broken

        Integer[] indexes = new Integer[n];

        for(int i = 0; i < n; i++) {

            indexes[i] = i;

        }

// The sort is NOT default sort(e.g (a, b) -> a[0] - b[0])),

// instead it using modified comparator, so require object 'Integer'

// rather than primitive 'int'

        Arrays.sort(indexes, (a, b) -> position[b] - position[a]);

        // Use a real stack to count fleets

        int fleet = 0;

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

        // Traverse cars from the closest position to the target

        for(int i = 0; i < n; i++) {

            double time = 1.0 \* (target - position[indexes[i]]) / speed[indexes[i]];

            // If the stack is empty or the current car takes more time than the car in front, form a new fleet

            if(stack.isEmpty() || stack.peek() < time) {

                stack.push(time); // Push the current car's time onto the stack

            }

        }

        // The number of fleets is the size of the stack

        return stack.size();

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(n)

**Lambda in Java sort broken?**

**Refer to**

<https://leetcode.com/discuss/feedback/2183318/lambda-in-java-sort-broken>

Below are the expressions I'm using to sort an integer array.

int[] nums = new int[]{1, 2, 6, 3, 5, 4};

Arrays.sort(nums, (a, b) -> b - a);

Arrays.sort(nums, (int a, int b) -> b - a);

Both the expression doesn't seem to work at all, whilst giving me an error.

Line 3: error: no suitable method found for sort(int[],(a,b)->b - a)

Arrays.sort(nums, (a, b) -> b - a);

^

method Arrays.<T#1>sort(T#1[],Comparator<? super T#1>) is not applicable

(inference variable T#1 has incompatible bounds

equality constraints: int

lower bounds: Object)

method Arrays.<T#2>sort(T#2[],int,int,Comparator<? super T#2>) is not applicable

(cannot infer type-variable(s) T#2

(actual and formal argument lists differ in length))

where T#1,T#2 are type-variables:

T#1 extends Object declared in method <T#1>sort(T#1[],Comparator<? super T#1>)

T#2 extends Object declared in method <T#2>sort(T#2[],int,int,Comparator<? super T#2>)

Both the expression return similar error. I wonder why? Would really appreciate someone's input in this matter. TY!

In Java, **Sorting of primitive types with a Comparator is not supported.**

The Arrays#sort method with a Comparator looks like this

public static <T> void sort(T[] a, Comparator<? super T> c) { ... }

I see that you are trying to sort in reverse order. Probably you can try something like below.

Sort in ascending first, and then, reverse the array.

int[] nums = new int[]{1, 2, 6, 3, 5, 4};

// 1. Sort in ascending

Arrays.sort(nums);

// 2. Reverse the array

for (int i=0; i<nums.length/2; i++) {

int temp = nums[i];

nums[i] = nums[nums.length-i-1];

nums[nums.length-i-1] = temp;

}

System.out.println(Arrays.toString(nums));

Using Streams

int[] nums = new int[]{1, 2, 6, 3, 5, 4};

nums = Arrays.stream(nums)

.boxed()

.sorted(Collections.reverseOrder())

.mapToInt(Integer::intValue)

.toArray();

System.out.println(Arrays.toString(nums));

**Solution 2: No Stack required, just simulate Stack behavior (60min)**

class Solution {

    public int carFleet(int target, int[] position, int[] speed) {

        int n = position.length;

        // Must define 'indexes' as 'Integer' instead of 'int',

        // otherwise Arrays.sort() not able to work

        Integer[] indexes = new Integer[n];

        for(int i = 0; i < n; i++) {

            indexes[i] = i;

        }

        Arrays.sort(indexes, (a, b) -> position[b] - position[a]);

        // Count fleets without real stack

        int fleet = 0;

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

        double prevTime = 0;

        // Traverse cars from the closest position to the target

        for(int i = 0; i < n; i++) {

            double time = 1.0 \* (target - position[indexes[i]]) / speed[indexes[i]];

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

            if(time > prevTime) {

                fleet++;

                // Update the previous time

                prevTime = time;

            }

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

        }

        return fleet;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(n)

**Refer to chatGPT**

class Solution {

public int carFleet(int target, int[] position, int[] speed) {

int n = position.length;

if (n == 0) return 0;

// Step 1: Create an array of cars with position and time to destination

double[][] cars = new double[n][2];

for (int i = 0; i < n; i++) {

cars[i][0] = position[i]; // Position of the car

cars[i][1] = (double) (target - position[i]) / speed[i]; // Time to reach the target

}

// Step 2: Sort cars by their position in descending order (farthest first)

Arrays.sort(cars, Comparator.comparingDouble(car -> -car[0]));

// Step 3: Use a stack to count fleets

int fleets = 0;

double lastTime = 0;

// Traverse cars from the closest position to the target

for (int i = 0; i < n; i++) {

double currentTime = cars[i][1];

// If this car takes longer than the last car's fleet time, it forms a new fleet

if (currentTime > lastTime) {

fleets++; // Increment the fleet count

lastTime = currentTime; // Update the last fleet's time

}

}

return fleets;

}

}

**Explanation of the Monotonic Stack Solution:**

Sorting the cars:

First, we calculate the time it takes for each car to reach the target, then we sort the cars based on their starting positions in descending order. This allows us to process the furthest cars first.

Using a stack-like approach:

Instead of explicitly using a stack, we can simulate it using a variable (lastTime).

We iterate through the sorted cars and compare the time it takes for each car to reach the target.

If a car takes **more time** than the car in front of it (i.e., the top of the stack), it forms a new fleet (because it can't catch up).

If it takes **less or the same time**, it will join the fleet of the car in front, so we don’t increment the fleet count in that case.

Fleet counting:

We increment the fleet count whenever we encounter a car that cannot catch up with the fleet in front (i.e., it has a larger time to reach the destination than the previous car's time).

**Refer to**

<https://algo.monster/liteproblems/853>

**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 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 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:

**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])

**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

**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:

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]).

**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.

**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:

**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

Initialize Variables: ans = 0, pre = 0

Iterate through Sorted Indices: We iterate through idx as [0, 1, 3, 2].

For i = 0 (Car 1):

t = (12 - 10) / 2 = 1

Since t > pre, we increment ans to 1, and pre updates from 0 to 1.

For i = 1 (Car 2):

t = (12 - 8) / 4 = 1

Since t == pre, we do not increment ans, and pre remains 1.

For i = 3 (Car 4):

t = (12 - 5) / 3 ≈ 2.33

Since t > pre, we increment ans to 2, and pre updates from 1 to 2.33.

For i = 2 (Car 3):

t = (12 - 0) / 1 = 12

Since t > pre, we increment ans to 3, and pre updates from 2.33 to 12.

Return Fleet Count: We have ans = 3, indicating that all the cars will form 3 fleets 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 3 car fleets by the time they reach the destination.

**Solution Implementation**

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;

}

}

**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)

**Refer to**

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

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

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

[L1776.Car Fleet II (Ref.L853)](note://WEBa2081c5af253ebfba9c4be4b2daa7307)