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

There is a car with capacity empty seats. The vehicle only drives east (i.e., it cannot turn around and drive west).

You are given the integer capacity and an array trips where trips[i] = [numPassengersi, fromi, toi] indicates that the ith trip has numPassengersi passengers and the locations to pick them up and drop them off are fromi and toi respectively. The locations are given as the number of kilometers due east from the car's initial location.

Return true *if it is possible to pick up and drop off all passengers for all the given trips, or* false *otherwise*.

**Example 1:**

Input: trips = [[2,1,5],[3,3,7]], capacity = 4

Output: false

**Example 2:**

Input: trips = [[2,1,5],[3,3,7]], capacity = 5

Output: true

**Constraints:**

* 1 <= trips.length <= 1000
* trips[i].length == 3
* 1 <= numPassengersi <= 100
* 0 <= fromi < toi <= 1000
* 1 <= capacity <= 105

**Attempt 1: 2023-11-28**

**Solution 1: Sort all intervals based on 'start' and push to Priority Queue based on interval 'end' (60 min)**

**The same Priority Queue way as L253/P5.5.Meeting Rooms II**

class Solution {

public boolean carPooling(int[][] trips, int capacity) {

PriorityQueue<int[]> minPQ = new PriorityQueue<>((a, b) -> a[2] - b[2]);

Arrays.sort(trips, (a, b) -> a[1] - b[1]);

// Must check initial element attempt to put on minPQ

// Test out by: trips={{9,0,1},{3,3,7}}, capacity=4

// Below is minPQ view (sort by 'to' in trips[i] = [numPassengers, from, to])

// 0 1 2 3 4 5 6 7 8 9 10

// | | -> 9 (already > capacity)

// | | -> 3

if(trips[0][0] > capacity) {

return false;

}

minPQ.offer(trips[0]);

int count = trips[0][0];

for(int i = 1; i < trips.length; i++) {

// Change if condition to while condition

// Test out by: trips={{3,2,8},{4,4,6},{10,8,9}}, capacity=11

// Below is minPQ view (sort by 'to' in trips[i] = [numPassengers, from, to])

// 1 2 3 4 5 6 7 8 9 10

// | | -> 4 (First remove {4,4,6})

// | | -> 3 (Have to remove {3,2,8} as well, use 'while')

// | | -> 10

while(!minPQ.isEmpty() && minPQ.peek()[2] <= trips[i][1]) {

int[] tmp = minPQ.poll();

count -= tmp[0];

}

if(count + trips[i][0] > capacity) {

return false;

}

minPQ.offer(trips[i]);

count += trips[i][0];

}

return true;

}

}

Time complexity: O(NlogN)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/car-pooling/solutions/1670052/c-2-approaches-priority-queue-with-intuition-easy-to-understand-explanation-stl/>

<https://leetcode.com/problems/car-pooling/solutions/317887/java-python-3-clean-codes-w-explanation-and-brief-analysis/>

**Method 1: PriorityQueue/heapqJava**use sort and PriorityQueue

* Sort the trips array by start location;
* Use a PriorityQueue to store the trips, order by ending location.
* Loop through the trips array, for each start location, keep polling out the arrays with corresponding end location <= current start location, for each polled out array, add the corresponding passengers to capacity; deduct passengers at current start location from capacity, if capacity < 0, return false.
* Repeat 3 till end, if never encounter false, return true.

Please refer to **@lcgt**'s excellent explanation as follows:

We first sort trips by start location, because that’s how the question is framed - we go in one direction and cannot turn back. That’s how we must iterate through the trips [array.Next](http://array.Next) we turn our attention to capacity -= trip[0] first. This is because pq will be empty initially empty anyway. Capacity goes down as long as a passenger gets in. That is straightforward. If capacity were ever to go below 0, we return false, that is also straightforward.

We put the whole trip’s information into the priority queue... why? Because we will eventually need the numPassengers and endLocation. Now that we have put our first trip into the PQ, we look at our next trip.

Now we have two possible scenarios:

* Our next trip’s start location were to be less than the end location of the trip we just stored (with lowest / nearest end location trips at the top of the heap / front of the queue). Skip the while block. Since we need to pick up more passengers, we once again decrease our capacity.
* Our next trip’s start location is greater than or equal to the trip’s end location that we just stored previously. We let out some passengers, and therefore we can increase our capacity again.

The PQ is like the “car” (maybe more like a big bus) in some sense, because it stores passengers such that the ones who will get off soon should sit near the front. The bus receives information about its next trip to make, and if the next trip’s start location is beyond what the current passengers destinations, those current passengers get off.

public boolean carPooling(int[][] trips, int capacity) {

Arrays.sort(trips, Comparator.comparing(trip -> trip[1]));

PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparing(trip -> trip[2]));

for (int[] trip : trips) {

while (!pq.isEmpty() && trip[1] >= pq.peek()[2]) // any passengers need to get off?

capacity += pq.poll()[0]; // more capacity as passengers out.

capacity -= trip[0]; // less capacity as passengers in.

if (capacity < 0) return false; // not enough capacity.

pq.offer(trip); // put into PriorityQueue the infomation at current location.

}

return true;

}

**Analysis:**

Time: O(nlogn),

space: O(n), n = trips.length.

**Solution 2: Sweep Line (30 min)**

class Solution {

public boolean carPooling(int[][] trips, int capacity) {

int[] timeline = new int[1001];

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

timeline[trips[i][1]] += trips[i][0];

timeline[trips[i][2]] -= trips[i][0];

}

int[] presum = new int[timeline.length + 1];

for(int i = 1; i < presum.length; i++) {

presum[i] = presum[i - 1] + timeline[i - 1];

}

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

if(presum[i] > capacity) {

return false;

}

}

return true;

}

}

===========================================================

Just merge two for loop as below

class Solution {

public boolean carPooling(int[][] trips, int capacity) {

int[] timeline = new int[1001];

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

timeline[trips[i][1]] += trips[i][0];

timeline[trips[i][2]] -= trips[i][0];

}

int[] presum = new int[timeline.length + 1];

for(int i = 1; i < presum.length; i++) {

presum[i] = presum[i - 1] + timeline[i - 1];

if(presum[i] > capacity) {

return false;

}

}

return true;

}

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://algo.monster/liteproblems/1094>

## Problem Description

In this problem, we are simulating a carpool scenario. A car has a certain number of empty seats (given by capacity), and it can only drive in one direction—east. We are provided with an array trips, where each element is a trip described by three integers: numPassengers, from, and to. These respectively represent the number of passengers for that trip, the kilometer mark where the passengers will be picked up, and the kilometer mark where they will be dropped off. Our task is to determine if the car can successfully complete all the given trips without ever exceeding its seating capacity. If it is possible to pick up and drop off all passengers for all the trips without going over capacity at any point, we return true. Otherwise, we return false.

## Intuition

The key insight to solving this problem is recognizing that we don't actually have to simulate the driving of the car along the eastward path. Instead, we can focus on the changes in the number of passengers at each pick-up and drop-off location. The original trips array tells us how many passengers get on and off at specific points, so we can tally these changes as they occur over the course of the car's journey.

Imagine a timeline where each point is a kilometer mark where some action takes place—a pick-up or a drop-off. We iterate through every trip and note the changes in passenger numbers at each relevant kilometer mark. The d array in the solution serves as this timeline, with indexes representing kilometer marks and values representing the change in the number of passengers at that mark.

When a trip starts (from), we add the number of passengers to the tally at that kilometer mark. When the trip ends (to), we subtract the number of passengers from the tally at the drop-off kilometer mark, since those passengers are no longer in the car.

After tallying all passenger changes, we use the accumulate function to simulate the succession of passengers over the journey. This function computes a running total that represents the number of passengers in the car at each kilometer mark. Finally, we check if this running total ever exceeds the car's capacity at any point. If it doesn't exceed the capacity, it means that the car can complete all trips successfully. Therefore, we return true if the car's capacity is never exceeded, or false if at some point there are too many passengers.

## Solution Approach

The solution utilizes a straightforward approach complemented by efficient use of data structures and algorithms. Here's a step-by-step breakdown of the implementation, showcasing the thought process behind the algorithm:

* The first data structure introduced is an array d of size 1001, initialized with zeros. This array is essential for recording the change in the number of passengers at each location along the trip. Why size 1001? This accounts for the maximum potential distance (from and to values) that could be specified in the problem (assuming they are within a reasonable range).
* We loop through each trip in the trips array using a for loop. In each iteration, we destructure the trip into numPassengers, from, and to. We are not concerned with the order of trips since we are looking at the overall effect on the car's capacity at each kilometer mark.
* For each trip:

1. At the pick-up location from, we increase the value in the d array by numPassengers.
2. At the drop-off location to, we decrease the value in the d array by numPassengers.

* The next step is to use the accumulate function from Python's itertools library. This function takes an iterable and returns a running total of the values. In our case, it generates a list where each element is the sum of passengers in the car up until that point. This list reflects the number of passengers present in the car at each kilometer mark of the journey.
* The final part of the solution involves checking if at any point the running total of passengers exceeds the car's capacity which is done using Python's all() function in conjunction with a generator expression. The expression inside all() checks for every kilometer: s <= capacity where s is an element from the running total obtained from accumulate(d). If all values are within capacity, all() returns true.
* The solution returns the result of this all() check. If the accumulated number of passengers at any point is greater than the car's capacity, the function will return false; otherwise, it will return true indicating successful completion of all trips within the given constraints.

This approach allows us to solve the problem without directly simulating the trip progress, which would be less efficient. We make just two passes: one for updating the d array, and another for accumulating and checking the sums against capacity, resulting in a time complexity that is linear with respect to the number of trips.

### Example Walkthrough

Let's use a small example to illustrate the solution approach.

Assume we are given the following parameters for our carpool problem:

capacity: 4

trips: [[2, 1, 5], [3, 3, 7]]

Here, the car has 4 empty seats. There are two trips to consider:

2 passengers from kilometer 1 to kilometer 5

3 passengers from kilometer 3 to kilometer 7

Now, we will walk through the solution step by step.

1.We initialize an array d of size 1001 with all zeros.

d = [0] \* 1001

2.We loop through each trip to update d. In our example:

* For the first trip [2, 1, 5]:

at from kilometer 1, add 2 (d[1] += 2)

at to kilometer 5, subtract 2 (d[5] -= 2)

* For the second trip [3, 3, 7]:

at from kilometer 3, add 3 (d[3] += 3)

at to kilometer 7, subtract 3 (d[7] -= 3)

After these updates, the array d would look like this (omitting zeros):

d[1] = 2

d[3] = 3

d[5] = -2

d[7] = -3

3. Next, we use the accumulate function to determine the number of passengers in the car at each point in the trip. The accumulate function will process the array d and give us:

[0, 2, 2, 5, 5, 3, 3, 0, 0, ..., 0] // Accumulated passenger counts

* This array represents the total number of passengers after each kilometer mark:

1. 0 passengers at the start
2. 2 passengers after kilometer 1
3. No change until kilometer 3
4. 5 passengers from kilometer 3 (since 2 were already in the car and 3 more joined)
5. Capacity is exceeded at this point (5 passengers > 4 capacity)
6. The count drops to 3 passengers after kilometer 5 (2 passengers leave)
7. No change until kilometer 7
8. All passengers have left by kilometer 7

* Finally, we check if the capacity of the car is ever exceeded using the all() function in conjunction with a generator expression:

result = all(s <= capacity for s in accumulate(d))

Since s is 5 at some point which is greater than the capacity of 4, the all() function would return false.

Therefore, our function that simulates whether all trips can be successfully completed given the capacity, with the input provided, would return false. The car cannot successfully complete all the given trips without exceeding its seating capacity.

class Solution {

// The function checks if it's possible to pick up and drop off all passengers for all trips

public boolean carPooling(int[][] trips, int capacity) {

// Initialize an array to record the cumulative changes in passenger count at each stop.

int[] passengerChanges = new int[1001];

// Iterate over all trips to record passenger pick-ups and drop-offs.

for (int[] trip : trips) {

int numberOfPassengers = trip[0]; // The number of passengers for the trip

int startLocation = trip[1]; // The starting location for the trip

int endLocation = trip[2]; // The ending location for the trip

// Add the number of passengers to the start location

passengerChanges[startLocation] += numberOfPassengers;

// Subtract the number of passengers from the end location

passengerChanges[endLocation] -= numberOfPassengers;

}

// Initialize the current number of passengers in the car to 0.

int currentPassengers = 0;

// Iterate over each location to update the number of passengers after each pick-up/drop-off.

for (int change : passengerChanges) {

currentPassengers += change; // Update the current number of passengers.

// If the current number of passengers exceeds capacity, return false.

if (currentPassengers > capacity) {

return false;

}

}

// If we've successfully accounted for all trips without exceeding capacity, return true.

return true;

}

}