1094. Car Pooling

Medium | Array | Prefix Sum | Sorting | Simulation | Heap (Priority Queue) | Leetcode Link

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:
  - At the pick-up location `from`, we increase the value in the `d` array by `numPassengers`.
  - 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:

1. 2 passengers from kilometer 1 to kilometer 5
2. 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:

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

Python Solution

```python
from itertools import accumulate  # Import the accumulate function from itertools
from typing import List  # Import List for type annotation

class Solution:
    def carPooling(self, trips: List[List[int]], capacity: int) -> bool:
        # Initialize a list of zeros for all possible locations (0 to 1000)
        location_deltas = [0] * 1001

        # Loop over each trip in the trips list
        for passengers, start_location, end_location in trips:
            # Increase the passenger count at the start location
            location_deltas[start_location] += passengers
            # Decrease the passenger count at the end location
            location_deltas[end_location] -= passengers

        # Use accumulate to calculate the running total of passengers at each location
        # and verify if at any point the capacity is exceeded
        # all() will return True if all running totals are less than or equal to capacity
        return all(current_passengers <= capacity for current_passengers in accumulate(location_deltas))

# The code defines a class Solution with a method carPooling.
# The method takes a list of trips (where each trip is a list of [passengers, start_location, end_location])
# and the vehicle's capacity.
# It returns True if the vehicle can accommodate all the trips without exceeding the capacity at any point.

```

Java Solution

```java
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;
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    bool carPooling(std::vector<std::vector<int>>& trips, int capacity) {
        // Create an array to keep track of changes in car capacity at each point.
        int deltaCapacity[1001] = {0};

        // Loop through the trips to record the changes in capacity.
        for (const auto& trip : trips) {
            int passengers = trip[0];  // Number of passengers in the trip
            int from = trip[1];        // Starting point of the trip
            int to = trip[2];          // End point of the trip

            deltaCapacity[from] += passengers; // Increment passengers at the start
            deltaCapacity[to] -= passengers;   // Decrement passengers at the end
        }

        // Initialize the sum that will track current number of passengers in car
        int currentPassengers = 0;

        // Iterate through each point to calculate the capacity usage.
        for (int pointChange : deltaCapacity) {
            currentPassengers += pointChange;  // Adjust the current capacity

            // If at any point the number of passengers exceeds capacity, return false.
            if (currentPassengers > capacity) {
                return false;
            }
        }

        // If we made it through all the points without exceeding capacity, return true.
        return true;
    }
};

```

Typescript Solution

```typescript
function carPooling(trips: number[][], capacity: number): boolean {
    // Create an array to hold the net number of passengers at each point
    const deltaPassengers = new Array(1001).fill(0);

    // Fill the deltaPassengers array with passenger changes at each point
    for (const [numPassengers, start, end] of trips) {
        // Add the passengers to the start location
        deltaPassengers[start] += numPassengers;
        // Subtract the passengers from the end location
        deltaPassengers[end] -= numPassengers;
    }

    // Track the current number of passengers in the car
    let currentPassengers = 0;

    // Iterate through each point to determine if capacity is exceeded
    for (const passengerChange of deltaPassengers) {
        // Update the current number of passengers
        currentPassengers += passengerChange;

        // If the current passengers exceed the capacity, return false
        if (currentPassengers > capacity) {
            return false;
        }
    }

    // If the loop finishes without returning false, the carpooling is possible within the capacity
    return true;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is O(n + m), where `n` is the length of the `trips` array and `m` is the range of possible stops (fixed at 1001 in this case). The loop iterates over all elements in `trips` once, which results in O(n) time. The `accumulate` function processes the differential list which is of size m, taking O(m) time. Therefore, the total time complexity is the sum of these two parts, O(n + m).

Space Complexity

The space complexity of the code is O(m), where `m` is the fixed range of 1001 for the possible stops. The list `d` is created with a size of 1001 to track the number of passengers getting on and off at each stop. No other significant space is used, so the space complexity is determined by the size of this list, which is O(m).