2332. The Latest Time to Catch a Bus

Medium Array Two Pointers Binary Search Sorting Leetcode Link

Problem Description

You are at a bus station with a schedule of bus departures and passenger arrivals. Each bus has a specific departure time and can carry a maximum number of passengers, known as the capacity. Passengers arrive at various times and will wait for the next available bus they can board. The goal is to find the latest possible time you can arrive at the station to catch a bus without arriving at the same time as another passenger.

Intuition

The idea is first to sort both the `buses` and `passengers` arrays since we need to process them in ascending order. The essence of the algorithm is to simulate the boarding process for each bus by letting the earliest arriving passengers board first until the bus reaches capacity or there are no more passengers for that bus.

For each bus, we iterate over the sorted passenger list. We keep track of the count of passengers (`c`) that have boarded the bus by decrementing the capacity each time we find an eligible passenger. If a bus becomes full, we keep track of the last passenger who boarded (just before the capacity was reached). If a bus isn't full, it implies that even if you arrive at the time the bus departs, you could still board (assuming no other passenger arrives at exactly that time).

After processing all buses, if the last bus is not full, we can simply return its departure time as the latest time you can arrive. If the last bus is full, we need to find a time just before the last passenger boarded the last bus where there's no other passenger arriving. We do this by decrementing the arrive time of the last boarded passenger until we find a time point where no passenger has arrived.

This effectively gives us the latest time slot where you can arrive and not coincide with any other passenger, ensuring you can board the last bus.

Solution Approach

The solution implements a straightforward approach using sorting and iteration, which can be broken down into the following steps:

1. **Sorting:** Both `buses` and `passengers` arrays are sorted in increasing order to simplify the simulation of the boarding process. This is done using Python's built-in `.sort()` method on the arrays.

2. **Iterating over buses:** We iterate through each bus's departure time in the `buses` array keeping track of a counter `c` representing the current carrying capacity of the bus. We also maintain an index `j` to iterate through the `passengers` array.

3. **Boarding passengers:** While the bus has capacity (`c`) and there are waiting passengers (`j < len(passengers)`) who arrived on or before the bus's departure time (`passengers[j] <= t`), we decrement `c` and increment `j`. This simulates passengers boarding the bus.

4. **Finding the last passenger's arrival time:** If the final bus is at full capacity after simulating the boarding process, the variable `ans` is set to the arrival time of the last passenger who boarded. Otherwise, `ans` is set to the departure time of the last bus.

5. **Finding the latest arrival time:** We then decrement `ans` until we find a time that does not coincide with any passenger's arrival time. We do this by iterating backwards through the sorted `passengers` array with the index `j`. While `ans` is equal to `passengers[j]`, meaning a passenger is already at the station at that time, we decrement `ans` and `j`.

6. **Result:** The `ans` variable, after this loop, will hold the latest time at which you can arrive without coinciding with any passenger and still be able to catch the last bus.

The overall algorithm runs in O(n log n) due to the sorting of `buses` and `passengers`, followed by a single pass through each, which is O(n).

Example Walkthrough

Let's consider a specific example to illustrate the solution approach:

Suppose we have 3 buses with departure times `buses = [3, 9, 15]` and each bus has a capacity of 2 passengers. Passengers arrive at times `passengers = [2, 5, 7, 8]`.

Following the solution approach:

Step 1: Sorting

- Sort `buses` and `passengers`: `buses = [3, 9, 15]` (already sorted) `passengers = [2, 5, 7, 8]` (already sorted)

Step 2: Iterating over buses

- Start with the first bus at time `3`. Set `c` which represents the bus capacity to `2`.

Step 3: Boarding passengers

- First bus (time `3`): `passengers[0]` is `2`, which is before bus departure. So, one passenger boards. `c` becomes `1`.
- There are no more passengers that arrived before or at time `3`, so the first bus leaves with one seat empty.
- Second bus (time `9`): Reset `c` to `2`.
- `passengers[1]` is `5` and `passengers[2]` is `7`. Both passengers board since they arrived before bus departure time. `c` becomes `0`.
- Last bus (time `15`): Reset `c` to `2`.
- `passengers[3]` is `8`. This passenger boards. `c` becomes `1`.

Step 4: Finding the last passenger's arrival time

- The final bus is not full; it has one remaining capacity. So, `ans` initially is the departure time of the last bus; `ans = 15`.

Step 5: Finding the latest arrival time

- Since the last bus is not at full capacity, we do not need to adjust `ans`. You can arrive exactly at time `15` and still board.

Step 6: Result

- The latest time you can arrive at the bus station without coinciding with another passenger and being able to catch a bus is `ans = 15`.

In this example, no adjustments were needed in Step 5 because the last bus was not fully boarded. If the final bus had been at full capacity, we would decrement `ans` and compare it against the sorted list of `passengers` to ensure that there's no conflict before confirming the final `ans`.

Python Solution

```python
from typing import List

class Solution:
    def latest_time_catch_the_bus(self, buses: List[int], passengers: List[int], capacity: int) -> int:
        # Sort the buses and passengers in ascending order to process them in sequence
        buses.sort()
        passengers.sort()

        passenger_index = 0  # index of the current passenger in the sorted list
        # Iterate through buses to see how many passengers each can pick up
        for bus_arrival_time in buses:
            current_capacity = capacity  # Track the current bus's remaining capacity
            # Board passengers until the bus is full or no more passengers for the bus
            while current_capacity > 0 and passenger_index < len(passengers) and passengers[passenger_index] <= bus_arrival_time:
                # Load a passenger and decrease the available capacity
                current_capacity -= 1
                passenger_index += 1

        # Adjust the index back to the last boarded passenger
        passenger_index -= 1

        # Latest possible time to catch the bus is either bus's last arrival time
        # or just a minute before the last passenger boarded if the bus is full
        latest_time = buses[-1] if current_capacity > 0 else passengers[passenger_index]

        # If the bus is full, find the latest time by subtracting from the last boarded passenger's time,
        # making sure there's no passenger at that time already
        while passenger_index >= 0 and passengers[passenger_index] == latest_time:
            latest_time -= 1
            passenger_index -= 1

        # Return the latest time a passenger can catch the bus
        return latest_time

```

Java Solution

```java
class Solution {

    // Method to find the latest time you can catch the bus without modifying method names as per guidelines.
    public int latestTimeCatchTheBus(int[] buses, int[] passengers, int capacity) {
        // Sort the buses and passengers to process them in order.
        Arrays.sort(buses);
        Arrays.sort(passengers);

        // Passenger index and current capacity initialization
        int passengerIndex = 0, currentCapacity = 0;

        // Iterate through each bus
        for (int busTime : buses) {
            // Reset capacity for the new bus
            currentCapacity = capacity;

            // Load passengers until the bus is either full or all waiting passengers have boarded.
            while (currentCapacity > 0 && passengerIndex < passengers.length && passengers[passengerIndex] <= busTime) {
                currentCapacity--;
                passengerIndex++;
            }
        }

        // Decrement to get the last passenger's time or the bus's latest time if it's not full
        passengerIndex--;

        // Determine the latest time that you can catch the bus
        int latestTime;

        // If there is capacity left in the last bus, the latest time is the last bus's departure time.
        // Otherwise, it's the time just before the last passenger boarded.
        if (currentCapacity > 0) {
            latestTime = buses[buses.length - 1];
        } else {
            latestTime = passengers[passengerIndex];
        }

        // Ensure that the latest time is not the same as any passenger's arrival time.
        while (passengerIndex >= 0 && latestTime == passengers[passengerIndex]) {
            latestTime--;
            passengerIndex--;
        }

        // Return the latest time you can catch the bus
        return latestTime;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int latestTimeCatchTheBus(vector<int>& buses, vector<int>& passengers, int capacity) {
        // Sort the times when buses arrive
        sort(buses.begin(), buses.end());
        // Sort the times when passengers arrive
        sort(passengers.begin(), passengers.end());

        // Use two pointers for the passenger and bus times
        int passengerIndex = 0;
        int currentCapacity = 0;

        // Loop through each bus
        for (int busTime : buses) {
            // Reset the capacity for each bus
            currentCapacity = capacity;
            // Fill the bus until capacity is reached or no passengers are left to board before the bus time
            while (currentCapacity && passengerIndex < passengers.size() && passengers[passengerIndex] <= busTime) {
                --currentCapacity;
                ++passengerIndex;
            }
        }

        // Move back one passenger to see the last passenger who boarded or the bus's last available time
        --passengerIndex;
        int latestTime = currentCapacity ? buses.back() : passengers[passengerIndex];

        // Find the latest time you could arrive without coinciding with a passenger time
        while (passengerIndex >= 0 && latestTime == passengers[passengerIndex]) {
            --passengerIndex;
            --latestTime;
        }

        // Return the latest possible time to catch the bus
        return latestTime;
    }
};

```

Typescript Solution

```typescript
// Sort numeric arrays in ascending order
function sortNumbers(a: number, b: number): number {
    return a - b;
}

// Method to find the latest time to catch the bus
function latestTimeCatchTheBus(buses: number[], passengers: number[], capacity: number): number {
    // Sort the bus arrival times
    buses.sort(sortNumbers);
    // Sort the passenger arrival times
    passengers.sort(sortNumbers);

    let passengerIndex: number = 0;
    let currentCapacity: number = 0;

    // Loop through each bus arrival time
    for (let busTime of buses) {
        // Initialize the capacity to the maximum for each bus
        currentCapacity = capacity;

        // Fill the bus with passengers who arrived before or at the bus time
        while (currentCapacity > 0 && passengerIndex < passengers.length && passengers[passengerIndex] <= busTime) {
            --currentCapacity;
            ++passengerIndex;
        }
    }

    // If the last bus has capacity, the latest time is its departure time
    // Otherwise, start from the last passenger who boarded
    let latestTime: number = (currentCapacity > 0) ? buses[buses.length - 1] : passengers[passengerIndex - 1];

    // Look for a time just before a passenger arrival time
    while (passengerIndex > 0 && latestTime === passengers[passengerIndex - 1]) {
        // Decrement both the passenger index and time to find earlier time slot
        --passengerIndex;
        --latestTime;
    }

    // Return the latest time you could arrive to catch the bus
    return latestTime;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the sorting of the `buses` and `passengers` lists, and the iterations over these lists.

1. `buses.sort()` sorts the list of buses. Sorting a list of `n` elements has a time complexity of `O(n log n)`, where `n` is the number of buses in this context.
2. `passengers.sort()` sorts the list of passengers. The sorting has a time complexity of `O(m log m)`, where `m` is the number of passengers.
3. The for loop iterates over each bus – this is `O(n)` where `n` is the number of buses.
4. The nested while loop iterates over the passengers, but it only processes each passenger once in total, not once per bus. Hence, the total number of inner loop iterations is `O(m)` across all iterations of the outer loop, where `m` is the total number of passengers.

Adding these up, we get a time complexity of `O(n log n) + O(m log m) + O(n) + O(m)`. Since the `O(n log n)` and `O(m log m)` terms will be dominant for large `n` and `m`, we can simplify this to `O(n log n + m log m)`.

Space Complexity

The space complexity is determined by the additional memory used by the program.

1. The sorting algorithms for both `buses` and `passengers` lists typically have a space complexity of `O(1)` if implemented as an in-place sort such as Timsort (which is the case in Python's sort() function).
2. The additional variables `c`, `j`, and `ans` use constant space, which adds a space complexity of `O(1)`.

Thus, when not considering the space taken up by the input, the overall space complexity of the code would be `O(1)`. However, if considering the space used by the inputs themselves, we must acknowledge that the lists `buses` and `passengers` use `O(n + m)` space.

Therefore, the total space complexity, considering input space, is `O(n + m)`.