731. My Calendar II

Medium **Design** Segment Tree Binary Search **Ordered Set** Leetcode Link

---

Problem Description

In this problem, you are tasked with creating a calendar system that can add new events without creating a situation wherein three events overlap in time—this is what is referred to as a "triple booking." Events are defined by their start and end times, with the start time being inclusive and the end time being exclusive, signified by the interval `[start, end)`. The problem requires the implementation of a class, `MyCalendarTwo`, which provides two functionalities:

1. Initializing the calendar object.
2. Booking an event (specified by its start and end) if doing so does not result in any triple booking. It returns `true` if the event can be added without a triple booking, otherwise `false`.

The objective is to efficiently manage a calendar by keeping track of events while ensuring at most two events may overlap, but not three. This requires careful tracking of each event's start and end times.

Intuition

The intuition behind the solution comes from the need to manage the overlaps efficiently. Given that double bookings are allowed but not triple bookings, we need to track whenever an event starts and ends, and how this impacts the existing timeline of bookings. Here, we can use a data structure such as the `SortedDict` from the `sortedcontainers` module which keeps the keys sorted and allows us to efficiently determine starting and ending points of events.

The approach is to increment the count at the event start time and decrement it at the event end time. Every time we attempt to book an event, we update the timeline with the start and end times. After adding the event to the calendar, we iterate through all time points in our `SortedDict` and maintain a running sum that represents the current number of overlapping events. If, at any time, this sum exceeds 2, it means we are trying to create a triple booking, which is not allowed. At this point, we need to revert this booking by decrementing the count at the start time and incrementing at the end time, and return `false` since we cannot book the event. If we successfully iterate through the entire sorted dictionary without the sum exceeding 2, the event is successfully booked without causing a triple booking, so we return `true`.

Solution Approach

The solution uses a class `MyCalendarTwo` that maintains a `SortedDict` from the `sortedcontainers` module. This dictionary will keep track of how many events are starting or ending at any given time. This approach is akin to employing a sweep line algorithm commonly used in computational geometry. The key idea is to "sweep" across the calendar and keep a count of concurrent events.

When booking a new event, we apply these steps in the `book` method:

1. Increment the counter for the event's start time: `self.sd[start] = self.sd.get(start, 0) + 1`. This represents the beginning of an event.

2. Decrement the counter for the event's end time: `self.sd[end] = self.sd.get(end, 0) - 1`. This signals the end of an event.

After updating the counters, we need to check for triple bookings:

3. Iterate over all values in our sorted dictionary using `self.sd.values()`. We keep a running sum `s` that represents the current number of overlapping events.

   a. For each value `v` in the `SortedDict`, we add it to our running sum `s += v`.

   b. If at any point our sum exceeds 2 (`if s > 2`), it signifies that the attempted booking would result in a triple booking, thus we revert the changes done in step 1 and 2 by decrementing the start time counter and incrementing the end time counter:

   ```
   self.sd[start] -= 1
   self.sd[end] += 1
   ```

   c. Since adding the event leads to a triple booking, we return `False`.

4. If we complete the iteration without our running sum ever exceeding 2, it means we have successfully added the event without causing a triple booking, and we return `True`.

The `SortedDict` data structure allows efficient insertion, deletion, and iteration, which is crucial for the performance of this algorithm. By incrementing start times and decrementing end times, we smartly keep track of ongoing events, and by checking the running sum, we enforce the no-triple-booking rule.

Example Walkthrough

Let's go through an example to illustrate the solution approach using the `MyCalendarTwo` class.

First, we initialize the `MyCalendarTwo` object:

```
my_calendar = MyCalendarTwo()
```

Our `SortedDict` starts empty as no events have been booked yet.

Now, let's try booking our first event `[10, 20)`:

```
result = my_calendar.book(10, 20)
```

- `self.sd[10]` becomes 1 because one event starts at time 10.
- `self.sd[20]` becomes -1 because one event ends at time 20.
- We iterate through the `SortedDict`, and the running sum `s` never exceeds 2, because we only have one event.
- The result is `True`, and the first event is successfully booked.

Now, suppose we book a second event `[15, 25)`:

```
result = my_calendar.book(15, 25)
```

- `self.sd[15]` gets incremented to 1, and since there is already one event overlapping at this time, the running total of events is now 2 at time 15 (as earlier, the running sum was 1 at 10, and then it becomes 2 at 15).
- `self.sd[25]` becomes -1.
- We iterate through the `SortedDict`, which now looks like this: `10: 1, 15: 1, 20: -1, 25: -1`, and the running sum `s` never exceeds 2.
- The result is `True`, and the second event is successfully booked.

Finally, let's try booking a third event `[20, 30)`:

```
result = my_calendar.book(20, 30)
```

- `self.sd[20]` remains unchanged because when an event ends, another begins at the same time.
- `self.sd[30]` becomes -1.
- The sorted dictionary now is `10: 1, 15: 1, 20: -1, 25: -1, 30: -1`, and while iterating, the running sum `s` does not exceed 2.
- The booking is successful as the sum `s` remains at most 2 at all points in time.

If we then attempt to book a fourth event `[10, 15)`:

```
result = my_calendar.book(10, 15)
```

- `self.sd[10]` would be incremented, going from 1 to 2, as there is now a second event starting at time 10.
- `self.sd[15]` would be decremented, going from 1 to 0.
- During iteration, when we reach time 15, the running sum `s` would become 3 (1 for the first event starting at 10, and 2 for the second and fourth events overlapping between 10 and 15) which exceeds our limit of 2.
- This would result in a triple booking, so we revert the changes (`self.sd[10]` goes back to 1 and `self.sd[15]` back to 1), and the result is `False`.

The event `[10, 15)` cannot be booked without causing a triple booking, and therefore, our `MyCalendarTwo` correctly returns `False`.

Python Solution

```python
from sortedcontainers import SortedDict

class MyCalendarTwo:
    def __init__(self):
        # Initialize a SortedDict to keep track of the booking times.
        # Keys are the start or end of an event, and values are the net number
        # of events starting or ending at that time.
        self.booking_counts = SortedDict()

    def book(self, start: int, end: int) -> bool:
        # Increment the count for the start time of the new event.
        self.booking_counts[start] = self.booking_counts.get(start, 0) + 1
        # Decrement the count for the end time of the new event.
        self.booking_counts[end] = self.booking_counts.get(end, 0) - 1

        # Initialize a running sum to track number of simultaneous events.
        running_sum = 0

        # Iterate over all booked time points (both start and end times).
        for count in self.booking_counts.values():
            # Update the running sum which represents the count of current
            # overlapping events at current time.
            running_sum += count

            # If there are more than 2 simultaneous events, it's a conflict.
            if running_sum > 2:
                # The booking is invalid, so we revert the increment and decrement
                # for the start and end time of the new event.
                self.booking_counts[start] -= 1
                self.booking_counts[end] += 1

                # Return False indicating booking was unsuccessful.
                return False

        # If no conflicts were found, return True indicating successful booking.
        return True

# Example usage of the MyCalendarTwo class:
# calendar = MyCalendarTwo()
# if calendar.book(10, 20):
#     print("Booking from 10 to 20 is successful.")
# else:
#     print("Booking from 10 to 20 is unsuccessful.")

```

Java Solution

```java
import java.util.Map;
import java.util.TreeMap;

public class MyCalendarTwo {

    // Use TreeMap to automatically keep the keys sorted
    private Map<Integer, Integer> timeMap = new TreeMap<>();

    // Default constructor (not explicitly needed unless more constructors are provided)
    public MyCalendarTwo() {
    }

    // Function to book a new event from start to end time
    public boolean book(int start, int end) {
        // Increase the counter at the start time
        timeMap.put(start, timeMap.getOrDefault(start, 0) + 1);

        // Decrease the counter at the end time
        timeMap.put(end, timeMap.getOrDefault(end, 0) - 1);

        int activeEvents = 0; // This will track the number of ongoing events

        // Iterate through the values in TreeMap
        for (int eventsCount : timeMap.values()) {
            // Increment the count of active events
            activeEvents += eventsCount;

            // If at any point there are more than 2 active events, this booking overlaps with two other events
            if (activeEvents > 2) {
                // The booking is not possible, so revert the changes
                timeMap.put(start, timeMap.get(start) - 1);
                timeMap.put(end, timeMap.get(end) + 1);

                // Return false as the booking overlaps and cannot be accepted
                return false;
            }
        }
        // The booking does not overlap with two or more events, so return true
        return true;
    }
}
```

C++ Solution

```cpp
#include <map>
using namespace std;

class MyCalendarTwo {
private:
    // Map to keep track of the number of bookings at any given point in time.
    map<int, int> bookings;

public:
    // Default constructor for MyCalendarTwo.
    MyCalendarTwo() {
        // The map is initially empty because no bookings have been made yet.
    }

    // Function to book a new event if it does not cause a triple booking.
    bool book(int start, int end) {
        // Increment the count for the start time.
        bookings[start]++;

        // Decrement the count for the end time.
        bookings[end]--;

        int count = 0; // To keep track of ongoing bookings.
        // Iterate over the map to check if there is any point in time
        // with more than two simultaneous bookings.
        for(auto& [time, bookingCount] : bookings) {
            count += bookingCount;
            // If there are more than two bookings at a certain time,
            // this means the current booking causes a triple booking.
            if (count > 2) {
                // Undo the changes - this booking is not allowed.
                bookings[start]--;
                bookings[end]++;

                // Return false as the booking cannot be made without causing a triple booking.
                return false;
            }
        }
        // If the loop completes without finding a triple booking,
        // the event can be successfully booked.
        return true;
    }
};

/**
 * The class definition and methods should not be changed as per the requirements.
 */

// Usage example:
// MyCalendarTwo* calendar = new MyCalendarTwo();
// bool canBook = calendar->book(10, 20); // Returns true if the booking is successful.
```

Typescript Solution

```typescript
// Represents the booking counters at various timestamps.
const bookings: { [key: number]: number } = {};

// Function used to book a new event if it does not cause a triple booking.
function book(start: number, end: number): boolean {
    // If the booking key doesn't exist, initialize to zero prior to incrementing.
    if (!bookings.hasOwnProperty(start)) bookings[start] = 0;
    if (!bookings.hasOwnProperty(end)) bookings[end] = 0;

    // Increment the count for the start time and decrement for the end time.
    bookings[start]++;
    bookings[end]--;

    let count = 0; // To keep track of the current number of overlapping bookings.

    // Sort the keys of the map since iteration order is not guaranteed in JavaScript/TypeScript.
    const sortedKeys = Object.keys(bookings).map(key => parseInt(key)).sort((a, b) => a - b);

    // Iterate over the sorted keys to check for triple bookings.
    for (const time of sortedKeys) {
        count += bookings[time];
        // If there are more than two bookings at any time, it's a triple booking.
        if (count > 2) {
            // Undo the increment/decrement operations, as the booking cannot be finalized.
            bookings[start]--;
            bookings[end]++;

            // Return false as the booking would lead to a triple booking.
            return false;
        }
    }

    // If there's no triple booking, the event can be booked.
    return true;
}

/*
 * Usage example:
 * let canBook = book(10, 20); // Returns true if the booking is successful without causing a triple booking.
 */
```

Time and Space Complexity

Time Complexity

The time complexity of the `book()` method is primarily dictated by three operations:

1. Insertions into `SortedDict`: Inserting a start or end key involves maintaining the sorted order, which runs in `O(log N)` time for each insertion, where `N` is the number of unique timestamps in `SortedDict`.
2. Value updates: Each call to `self.sd.get()` is `O(1)` since it's a simple dictionary operation, but the subsequent update back into the `SortedDict` is `O(1)` because we're not changing the keys, only the values.
3. Iterating over the values and checking for overlapping events: This operation has a time complexity of `O(N)` because we are iterating through each timestamp's value once.

The worst-case complexity is dominated by the third step if `N` is the number of unique events (i.e., not the total number of calls to `book()`). Therefore, the worst-case time complexity per `book()` call is `O(N)`.

Space Complexity

The space complexity is `O(N)` due to the storage requirements of the `SortedDict`, where `N` is the number of unique timestamps. The space required increases with each unique start or end timestamp that we add to the dict. There is no additional space usage that grows with the size of the input beyond what is used for the `SortedDict`.