2406. Divide Intervals Into Minimum Number of Groups

Medium Greedy Array Two Pointers Prefix Sum Sorting Heap (Priority Queue) Leetcode Link

Problem Description

You are given a list of intervals, and each interval is a list of two integers indicating the start and the end of that interval. The goal is to divide these intervals into groups such that no two intervals in the same group overlap each other. An overlap means that there is at least one number that is contained in both intervals. For instance, [1, 5] and [5, 8] are considered overlapping because they both contain the number 5. You need to find the minimum number of groups necessary to achieve this separation.

Intuition

The key to solving this problem is understanding that whenever we have a new interval, it can either be added to an existing group if it doesn't overlap with other intervals in that group, or we need to create a new group if it overlaps with all existing groups. The problem resembles organizing meeting rooms: each interval is like a meeting, and each group is a meeting room. The challenge is to book the minimum number of meeting rooms.

To figure out the solution, we can:

1. Sort the intervals based on the starting points. This way, we consider the intervals in the order of their starting times.
2. Use a min-heap to keep track of the end times of the last interval in each group. A min-heap is a binary tree where the parent node is always less than or equal to its child nodes; therefore, the smallest element is always at the root of the tree, and we can access it in constant time.
3. For each interval, check the root of the min-heap, which gives the earliest end time of all groups. If the start of the current interval is greater than the earliest end time, this means that we can add the interval to this group without overlaps, and thus, we replace the old end time with the end time of the current interval by popping the root of the heap and pushing the new end time.
4. If the start of the interval is not greater than the earliest end time in the heap, it means that it overlaps with every group, and we need to start a new group. So, we push the end time of this interval into the heap.
5. Continue this process until we have checked all intervals. The size of the heap at the end denotes the minimum number of groups needed, as each item in the heap represents a group's latest end time and all intervals within that group that do not overlap.

The given Python solution follows this approach, using the `heapq` module to manage the min-heap operations efficiently.

Solution Approach

The solution uses a min-heap to efficiently manage the end points of the intervals in the current groups and the sorting pattern to preprocess the intervals. The steps for the algorithm are as follows:

1. **Sorting the intervals**: The given intervals are sorted based on their starting point using `sorted(intervals)`. Sorting is crucial because it allows us to go through intervals sequentially, considering the earliest starting interval first.
2. **Initializing the min-heap**: A min-heap `h` is used to keep track of the end points of the intervals in the various groups. In Python, this is typically implemented using the `heapq` module.
3. **Iterating through the sorted intervals**: For every interval `(a, b)` in `sorted(intervals)`:
   - Check if there is an existing group that the interval can be added to without overlapping. This is done by examining the smallest end point in the min-heap `h[0]`. If `h[0] < a`, it indicates that there's a group whose last interval ends before the current interval starts, so they do not overlap.
   - If there's an overlap with all existing groups (`h[0] >= a`), the interval needs to be placed in a new group. Therefore, we push the end point `b` of the current interval onto the heap using `heappush(h, b)`.
   - If there is no overlap (`h[0] < a`), we remove the end point of the finished group from the heap using `heappop(h)` and then push the end point `b` of the current interval. This effectively updates the group end point to the current interval's end point without increasing the number of groups.
4. **Returning the result**: After all intervals have been processed, the number of elements in the min-heap corresponds to the number of groups needed. Each element in the heap represents the end time of a group that does not overlap with the others. The length of the heap `len(h)` is returned as the minimum number of groups required.

This algorithm ensures that all intervals are in exactly one group and that no two intervals in the same group intersect by always checking for the earliest possible group that an interval could belong to and creating a new group only when necessary.

Example Walkthrough

Let's take the following list of intervals as an example to walk through the solution approach:

```
Intervals: [[1, 4], [2, 5], [7, 9], [8, 10], [6, 8]]
```

Steps:

1. **Sorting the intervals**:
   - First, we sort the intervals based on their start times.
   - Sorted Intervals: [[1, 4], [2, 5], [6, 8], [7, 9], [8, 10]]
2. **Initializing the min-heap**:
   - We initiate a min-heap to keep track of end times of the formed groups.
   - Initial heap `h`: []
3. **Iterating through the sorted intervals**:
   - For the first interval `[1, 4]`, the heap is empty, so we push the end time `4` onto the heap.
     - Heap `h`: [4]
   - Next, consider interval `[2, 5]`. Since it starts before interval `[1, 4]` ends (`h[0] < 2` is false), we cannot add it to the same group. So, we push its end time `5` onto the heap.
     - Heap `h`: [4, 5]
   - Then, look at interval `[6, 8]`. Since it starts after interval `[1, 4]` ends (`h[0] < 6` is true), we pop `4` and push end time `8`.
     - Heap `h`: [5, 8]
   - The fourth interval `[7, 9]` starts before interval `[5, 8]` ends (`h[0] < 7` is false), hence we need a new group. We push end time `9`.
     - Heap `h`: [5, 8, 9]
   - Finally, we have interval `[8, 10]`. It starts after interval `[5, 8]` ends (`h[0] < 8` is true), so we pop `5` and push end time `10`.
     - Heap `h`: [8, 9, 10]
4. **Returning the result**:
   - With all intervals processed, the size of the heap is `3`, indicating that we need at least 3 groups to separate the intervals with no overlaps.
   - Hence, the minimum number of groups required is `3`.

By adhering to this method, we ensure each interval is placed in the correct group without overlaps, and the number of groups is minimized.

Python Solution

```python
from typing import List
import heapq

class Solution:
    def minGroups(self, intervals: List[List[int]]) -> int:
        # Initialize a min-heap to store end times of intervals
        min_heap = []

        # Sort the intervals based on starting time
        sorted_intervals = sorted(intervals, key=lambda x: x[0])

        # Loop through each interval in the sorted list
        for start, end in sorted_intervals:
            # If the heap is not empty and the smallest end time is less than
            # the current interval's start time, remove the interval from the heap
            # since it doesn't overlap with the current one.
            if min_heap and min_heap[0] < start:
                heapq.heappop(min_heap)

            # Add the current interval's end time to the heap
            heapq.heappush(min_heap, end)

        # The length of the heap gives the minimum number of overlapping groups.
        return len(min_heap)

# Example usage:
# sol = Solution()
# ans = sol.minGroups([[1,3],[2,4],[3,5]])
# print(ans)  # Expected output would be 2

```

Java Solution

```java
import java.util.Arrays;
import java.util.PriorityQueue;

class Solution {

    /**
    * This method calculates the minimum number of groups needed such that
    * each group consists of non-overlapping intervals from the given array of intervals.
    * Intervals are considered to have a start and an end, and overlapping intervals cannot
    * be in the same group.
    *
    * @param intervals Array of intervals where each interval is represented by a pair [start, end].
    * @return The minimum number of groups required.
    */
    public int minGroups(int[][] intervals) {
        // Sort the intervals based on the start time
        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

        // A priority queue to manage the end times of the intervals
        PriorityQueue<Integer> endTimeQueue = new PriorityQueue<>();

        // Iterate over all intervals
        for (int[] interval : intervals) {
            // If the queue is not empty and the smallest end time is less than the
            // start of the current interval, we can reuse this group for the new interval
            if (!endTimeQueue.isEmpty() && endTimeQueue.peek() < interval[0]) {
                endTimeQueue.poll(); // Remove the interval with the smallest end time
            }

            // Add the current interval's end time to the queue
            endTimeQueue.offer(interval[1]);
        }

        // The size of the priority queue indicates the minimum number of groups needed
        return endTimeQueue.size();
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>
#include <algorithm>

class Solution {
public:
    // Function to find the minimum number of groups required
    // so that no two intervals overlap within the same group.
    int minGroups(vector<vector<int>>& intervals) {
        // Sort the intervals based on their start times.
        std::sort(intervals.begin(), intervals.end());

        // Priority queue to store the end times of intervals.
        // The queue is ordered such that the smallest end time is at the top.
        std::priority_queue<int, std::vector<int>, std::greater<int>> endTimes;

        // Iterate over the sorted intervals
        for (const auto& interval : intervals) {
            // If the priority queue is not empty and the smallest end time is
            // less than the current interval's start time:
            if (!endTimes.empty() && endTimes.top() < interval[0]) {
                // This means the current interval does not overlap with the interval
                // that has the earliest end time. So, we can recycle this group.
                endTimes.pop();
            }
            // Push the current interval's end time into the priority queue.
            // This indicates that we have either used an existing group or created a new one.
            endTimes.push(interval[1]);
        }

        // The size of the priority queue indicates the number of groups required.
        // Each group has no overlapping intervals.
        return endTimes.size();
    }
};

```

Typescript Solution

```typescript
// Define the type for an interval.
type Interval = [number, number];

// Function to compare two intervals based on their start time.
function compareIntervals(a: Interval, b: Interval): number {
    return a[0] - b[0];
}

// Function to find the minimum number of groups required
// so that no two intervals overlap within the same group.
function minGroups(intervals: Interval[]): number {
    // Sort the intervals based on their start times.
    intervals.sort(compareIntervals);

    // Array to store the end times of intervals, functions like a priority queue.
    // This array is kept sorted such that the smallest end time is at the start.
    let endTimes: number[] = [];

    // Iterate over the sorted intervals
    for (const interval of intervals) {
        // Check if there is an existing group that this interval can join
        // without overlapping. This is the case if the earliest ending group
        // ends before the current interval starts.
        let canJoinGroup = false;
        for (let i = 0; i < endTimes.length; ++i) {
            if (endTimes[i] < interval[0]) {
                // The current interval can join this group, so update the end time.
                endTimes[i] = interval[1];
                canJoinGroup = true;
                break;
            }
        }

        // If the current interval could not join any group, create a new one.
        if (!canJoinGroup) {
            endTimes.push(interval[1]);
        }

        // Keep the endTimes array sorted, so the smallest end time is at the start.
        endTimes.sort((a, b) => a - b);
    }

    // The length of the endTimes array indicates the number of groups needed.
    // Each group contains no overlapping intervals.
    return endTimes.length;
}

```

Time and Space Complexity

Time Complexity

The given code consists of three main parts: sorting the intervals, iterating through the sorted intervals, and manipulating a heap for each interval.

1. The sort operation at the beginning has a time complexity of `O(N log N)`, where `N` is the number of intervals, because TimSort (Python's built-in sorting algorithm) is used.
2. Iterating through the sorted intervals has a linear time complexity of `O(N)` since we go through all intervals once.
3. For each interval, we might perform a `heap pop` and `heap push` operation, both of which have a time complexity of `O(log K)`, where `K` is the number of elements in the heap. In the worst-case scenario, each interval might be overlapping with all others, which would mean K can approach N, so we consider `O(log N)` as the complexity for these heap operations.

Combining these, the overall worst-case time complexity is `O(N log N + N log N)` which simplifies to `O(N log N)`, the dominant term being the sort and heap operations.

Space Complexity

The additional space used by the code is for the heap `h`. In the worst case, the heap can contain all `N` intervals if none of them can be merged, which means the space complexity is `O(N)` for storing heap elements.

Therefore, the overall space complexity of the code is `O(N)`.