1353. Maximum Number of Events That Can Be Attended

Medium Greedy Array Sorting Heap (Priority Queue) Leetcode Link

Problem Description

In this problem, we are given a list of `events`, where each `event` is represented by a start day and an end day, indicating the duration during which the event takes place. We can choose to attend an event on any day from the start day to the end day inclusive. However, we can only attend one event at any given day. Our goal is to maximize the number of events that can be attended.

Intuition

The intuition behind the solution is to prioritize attending events based on their end dates because we want to ensure we do not miss out on events that are about to end. For this reason, a greedy algorithm works efficiently — sorting the events by their end times could help us attend as many as possible.

However, simply sorting by the end times is not adequate since we also have to consider the starting times. Therefore, we create a priority queue (min-heap) where we will keep the end days of events that are currently available to attend. We also use two variables to keep track of the minimum and maximum days we need to cover.

As we iterate through each day within the range, we do the following:

1. Remove any events that have already ended.
2. Add all events that start on the current day to the priority queue.
3. Attend the event that is ending soonest (if any are available).

By using a priority queue (min-heap), we ensure that we are always attending the event with the nearest end day, hence maximizing the number of events we can attend.

Solution Approach

The solution uses a greedy approach combined with a priority queue (min-heap) to facilitate the process of deciding which event to attend next. Specifically, it applies the following steps:

1. **Initialization:**
   - A dictionary `d` is used to map each start day to a list of its corresponding end days. This enables easy access to events starting on a particular day.
   - Two variables, `i` and `j`, are initialized to `inf` and `0`, respectively, to track the minimum start day and the maximum end day across all events.
2. **Building the dictionary:**
   - The solution iterates over each event and populates the dictionary `d` with the start day as the key and a list of end days as the value.
   - It also updates `i` to the minimum start day and `j` to the maximum end day encountered.
3. **Setting up a min-heap:**
   - A priority queue (implemented as a min-heap using a list `h`) is created to keep track of all the end days of the currently available events.
4. **Iterating over each day:**
   - For each day `s` in the range from the minimum start day `i` to the maximum end day `j` inclusive:
     - While there are events in the min-heap that have ended before day `s`, they are removed from the heap since they can no longer be attended.
     - All events starting on day `s` are added to the min-heap with their end days.
     - If the min-heap is not empty, it means there is at least one event that can be attended. The event with the earliest end day is attended (removed from the heap), and the answer count `ans` is incremented by one.
5. **Returning the result:**
   - After iterating through all the days, the `ans` variable that has been tracking the number of events attended gives us the maximum number of events that can be attended.

In summary, by using a combination of a dictionary to map start days to events, a min-heap to efficiently find the soonest ending event that can be attended, and iteration over each day, the solution efficiently computes the maximum number of events that one can attend.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Suppose we are given the following list of events:

```
Events = [[1,4], [4,4], [2,2], [3,4], [1,1]]
```

1. **Initialization:**
   - We create a dictionary `d`, and two variables `i = inf` and `j = 0`.
2. **Building the dictionary:**
   - We iterate over the events:
     - For event `[1,4]`, we update `d` with `{1: [4]}` and set `i = 1` and `j = 4`.
     - For event `[4,4]`, we update `d` with `{1: [4], 4: [4]}`. Variables `i` and `j` remain unchanged.
     - For event `[2,2]`, we update `d` with `{1: [4], 2: [2], 4: [4]}`. Variables `i` and `j` remain unchanged.
     - For event `[3,4]`, we update `d` with `{1: [4], 2: [2], 3: [4], 4: [4]}`. Variables `i` and `j` remain unchanged.
     - For event `[1,1]`, we update `d` with `{1: [4, 1], 2: [2], 3: [4], 4: [4]}`. Variables `i` and `j` remain unchanged.
3. **Setting up a min-heap:**
   - We initialize an empty min-heap list `h`.
4. **Iterating over each day:**
   - We have `i = 1` and `j = 4`, so we iterate from day `1` to day `4`.
   - On day `1`:
     - We add all end days of events starting on day `1` to `h`, so `h` becomes `[4, 1]`.
     - We pop `1` from `h` as it's the earliest end day, attend this event, and increment `ans` to `1`.
   - On day `2`:
     - There's no event ending before day `2`, so nothing is removed from `h`.
     - We add the end day of the event starting on day `2` to `h`, so `h` becomes `[4, 2]`.
     - We pop `2` from `h`, attend this event, and increment `ans` to `2`.
   - On day `3`:
     - There's no event ending before day `3`, so nothing is removed from `h`.
     - We add the end day of the event starting on day `3` to `h`, so `h` becomes `[4, 4]`.
     - We pop `4` from `h` (either one, as both have the same end day), attend this event, and increment `ans` to `3`.
   - On day `4`:
     - Since there is only one event with an end day of `4` left in `h`, we attend it and increment `ans` to `4`.
     - We also check for more events starting today which is one `[4, 4]` and add it to the heap.
     - We then attend this event and increment `ans` to `5`.
5. **Returning the result:**
   - After iterating through all days, we find that `ans = 5`, which means we could attend a total of 5 events.

In this example, by using the greedy approach outlined in the solution, we were methodically able to maximize the number of events that could be attended by ensuring we attend the ones ending soonest first.

Python Solution

```python
from collections import defaultdict
from heapq import heappush, heappop
from math import inf

class Solution:
    def maxEvents(self, events: List[List[int]]) -> int:
        # Create a default dictionary to hold events keyed by start date
        event_dict = defaultdict(list)

        # Initialize variables to track the earliest and latest event dates
        earliest_start, latest_end = inf, 0

        # Populate event_dict with events and update earliest_start and latest_end
        for start, end in events:
            event_dict[start].append(end)
            earliest_start = min(earliest_start, start)
            latest_end = max(latest_end, end)

        # Initialize an empty min-heap to store active events' end dates
        min_heap = []

        # Counter for the maximum number of events one can attend
        max_events_attended = 0

        # Iterate over each day within the range of event dates
        for day in range(earliest_start, latest_end + 1):
            # Remove events that have already ended
            while min_heap and min_heap[0] < day:
                heappop(min_heap)

            # Push all end dates of events starting today onto the heap
            for end in event_dict[day]:
                heappush(min_heap, end)

            # If there are any events available to attend today, attend one and increment count
            if min_heap:
                max_events_attended += 1
                heappop(min_heap)  # Remove the event that was attended

        # Return the total number of events attended
        return max_events_attended
```

Java Solution

```java
class Solution {
    public int maxEvents(int[][] events) {
        // Create a map to associate start days with a list of their respective end days
        Map<Integer, List<Integer>> dayToEventsMap = new HashMap<>();
        int earliestStart = Integer.MAX_VALUE; // Initialize earliest event start day
        int latestEnd = 0; // Initialize latest event end day

        // Process the events to populate the map and find the range of event days
        for (int[] event : events) {
            int startDay = event[0];
            int endDay = event[1];

            // Map the start day to the end day of the event
            dayToEventsMap.computeIfAbsent(startDay, k -> new ArrayList<>()).add(endDay);

            // Update earliest start and latest end
            earliestStart = Math.min(earliestStart, startDay);
            latestEnd = Math.max(latestEnd, endDay);
        }

        // Create a min-heap to manage event end days
        PriorityQueue<Integer> eventsEndingQueue = new PriorityQueue<>();

        int attendedEventsCount = 0; // Initialize the count of events attended

        // Iterate over each day within the range of event days
        for (int currentDay = earliestStart; currentDay <= latestEnd; ++currentDay) {
            // Remove past events that have already ended
            while (!eventsEndingQueue.isEmpty() && eventsEndingQueue.peek() < currentDay) {
                eventsEndingQueue.poll();
            }

            // Add new events that start on the current day
            List<Integer> eventsStartingToday = dayToEventsMap.getOrDefault(currentDay, Collections.emptyList());
            for (int endDay : eventsStartingToday) {
                eventsEndingQueue.offer(endDay);
            }

            // Attend the event that ends the earliest, if any are available
            if (!eventsEndingQueue.isEmpty()) {
                eventsEndingQueue.poll();
                ++attendedEventsCount; // Increment the count of events attended
            }
        }

        return attendedEventsCount;
    }
}
```

C++ Solution

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

using namespace std;

class Solution {
public:
    int maxEvents(vector<vector<int>>& events) {
        // Map to hold the events on each day
        unordered_map<int, vector<int>> eventsByStartDay;
        // Initialize the minimum and maximum days across all events
        int minDay = INT_MAX;
        int maxDay = 0;

        // Iterate through all the events
        for (auto& event : events) {
            int startDay = event[0];
            int endDay = event[1];
            // Map the end day of each event to its start day
            eventsByStartDay[startDay].push_back(endDay);
            // Update the minimum and maximum days
            minDay = min(minDay, startDay);
            maxDay = max(maxDay, endDay);
        }

        // Min-heap (priority queue) to keep track of the events' end days, prioritised by earliest end day
        priority_queue<int, vector<int>, greater<int>> minHeap;
        // Counter to hold the maximum number of events we can attend
        int maxEventsAttended = 0;

        // Iterate through each day from the earliest start day to the latest end day
        for (int day = minDay; day <= maxDay; ++day) {
            // Remove events that have already ended
            while (!minHeap.empty() && minHeap.top() < day) {
                minHeap.pop();
            }
            // Add all events starting on the current day to the min-heap
            for (int endDay : eventsByStartDay[day]) {
                minHeap.push(endDay);
            }
            // If we can attend an event, remove it from the heap and increase the count
            if (!minHeap.empty()) {
                maxEventsAttended++;
                minHeap.pop();
            }
        }

        // Return the maximum number of events that can be attended
        return maxEventsAttended;
    }
};
```

Typescript Solution

```typescript
// Importing necessary functionalities from standard TypeScript library
import { PriorityQueue } from 'typescript-collections';

// Function to determine the maximum number of events that can be attended
const maxEvents = (events: number[][]): number => {
  // A dictionary to hold events keyed by their start day
  const eventsByStartDay: { [key: number]: number[] } = {};

  // Initialize minimum and maximum days for all events
  let minDay: number = Number.MAX_SAFE_INTEGER;
  let maxDay: number = 0;

  // Populate the eventsByStartDay and define minimum and maximum days across all events
  events.forEach(event => {
    const [startDay, endDay] = event;
    eventsByStartDay[startDay] = eventsByStartDay[startDay] || [];
    eventsByStartDay[startDay].push(endDay);

    minDay = Math.min(minDay, startDay);
    maxDay = Math.max(maxDay, endDay);
  });

  // Using a TypeScript priority queue to manage events' end days
  const minHeap: PriorityQueue<number> = new PriorityQueue<number>((a, b) => a - b);

  // Counter for the maximum number of events attended
  let maxEventsAttended: number = 0;

  // Iterate from the minimum start day to the maximum end day
  for (let day = minDay; day <= maxDay; day++) {
    // Remove events that have already ended
    while (!minHeap.isEmpty() && minHeap.peek() < day) {
      minHeap.dequeue();
    }
    // Add new events that start on the current day to the heap
    if (eventsByStartDay[day]) {
      eventsByStartDay[day].forEach(endDay => {
        minHeap.enqueue(endDay);
      });
    }
    // Attend the event that ends the earliest
    if (!minHeap.isEmpty()) {
      maxEventsAttended++;
      minHeap.dequeue();
    }
  }

  // Return the total number of events that can be attended
  return maxEventsAttended;
};

// Typescript doesn't have a built-in PriorityQueue, but you can use the 'typescript-collections' library to match the desired functi
```

Time and Space Complexity

The given Python code aims to find the maximum number of events one can attend, given a list of events where each event is represented by a start and end day. The code uses a greedy algorithm with a min-heap to facilitate the process.

Time Complexity:

Let's analyze the time complexity step by step:

1. Building the dictionary `d` has a complexity of `O(N)`, where N is the number of events since we iterate through all the events once.
2. Populating the min-heap `h` on each day has a variable complexity. In the worst case, we could be adding all events to the heap on a single day which will be `O(N log N)` due to `N` heap insertions (heappush operations), each with `O(log N)` complexity.
3. The outer loop runs from the minimum start time `i` to the maximum end time `j`. Therefore, in the worst-case scenario, it would run `O(j - i)` times.
4. Inside this loop, we perform a heap pop operation for each day that an event ends before the current day. Since an event end can only be popped once, all these operations together sum up to `O(N log N)`, as each heappop operation is `O(log N)` and there are at most `N` such operations throughout the loop.
5. We also perform a heap pop operation when we can attend an event, and this happens at most `N` times (once for each event).

Adding these complexities, we have:

- For the worst case, a complexity of `O(N log N + (j - i))` for the loop, with `O(N log N)` potentially dominating the overall time complexity when `(j - i)` is not significantly larger than `N`.

In conclusion, the time complexity of the code is `O(N log N + (j - i))`. However, `(j - i)` may be considered negligible compared to `N log N` for large values of `N`, yielding an effective complexity of `O(N log N)`.

Space Complexity:

Let's analyze the space complexity:

1. The dictionary `d` can hold up to `N` entries in the form of lists, with each list containing at least one element, but potentially up to `N` end times in the worst case. Therefore the space required for `d` is `O(N)`.
2. The min-heap `h` also requires space which in the worst-case scenario may contain all `N` events at once. Thus, the space complexity due to the heap is `O(N)`.

The min-heap `h` and the dictionary `d` represent the auxiliary space used by the algorithm. Since they both have `O(N)` space complexity, the overall space complexity is also `O(N)`, assuming that the space required for input and output is not taken into consideration, which is standard in space complexity analysis.