1751. Maximum Number of Events That Can Be Attended II

Hard Array Binary Search Dynamic Programming Sorting

Leetcode Link

Problem Description

In this problem, you are provided with an array named `events`, where each element in this array is another array that contains three integers: `[startDay_i, endDay_i, value_i]`. The `i-th` element represents an event that begins on `startDay_i`, ends on `endDay_i`, and attending this event yields a reward of `value_i`. Additionally, an integer `k` is given, indicating the maximum number of events you can attend.

However, there are constraints: you can attend only one event at a time, and if you attend an event, you must be present for the entire duration of the event, from `startDay_i` to `endDay_i` inclusive. This means that you cannot attend part of an event, and you also can't jump into another event that overlaps in date with the current one.

The goal is to select events to attend in such a way that maximizes the sum of values received, without exceeding the max number of events (`k`) that you can attend. You must then return the maximum sum of values you can achieve.

Intuition

To solve this problem, a dynamic programming approach is used. The events are first sorted based on their end dates to facilitate sequential processing from the earliest end date to the latest.

The core of the approach is to construct a 2D array `f` where `f[i][j]` represents the maximum value that can be obtained by considering up to the `i-th` event and attending at most `j` events. By iterating through each event (denoted as `i`), we have the option to either attend this event or not. If the current event is attended, we add its value to the maximum value obtained by attending `j-1` events before the current event's start. In case the current event is not attended, we carry forward the maximum value of attending `j` events without considering the current event.

To check which other events could have been attended before the `i-th` event, a binary search is performed to find the rightmost event that ends before the current event's start, ensuring no overlap. This efficient search is possible due to the initial sorting of the events by their end dates.

Finally, the resultant value `f[n][k]` (where `n` is the number of events) will give us the maximum sum we can achieve by attending up to `k` events. The technique leverages the Python `bisect_left` function for binary search, and iteratively updates the dynamic programming table to arrive at the solution.

Solution Approach

The solution is implemented using Dynamic Programming (DP), which is a method for solving complex problems by breaking them down into simpler subproblems. It relies on the fact that the optimal solution to a problem depends upon the optimal solution to its subproblems.

Let's walk through the code:

1. Sort `events` by their end time to process them in order of completion.

```
events.sort(key=lambda x: x[1])
```

2. Initialize the DP table `f` with `n + 1` rows and `k + 1` columns filled with zeros, where `n` is the number of events. The DP table is used to store the maximum value that can be obtained by attending a certain number of events up to a certain point in time.

```
f = [[0] * (k + 1) for _ in range(n + 1)]
```

3. Loop over sorted events, with each event enumerated as `i`, starting from 1. This loop is responsible for filling the DP table with the maximum values for each scenario of attending up to `j` events when you are considering up to the `i-th` event.

```
for i, (st, _, val) in enumerate(events, 1):
```

4. Inside the loop, for each event, use binary search to find the rightmost event that ends before the current event starts (to ensure no overlap) using `bisect_left`.

```
p = bisect_left(events, st, hi=i - 1, key=lambda x: x[1])
```

5. Nested within is another loop over the range `1` to `k + 1` (the maximum number of events you can attend), denoted as `j`. For each possible number of events to attend, we are calculating the maximum value possible either by including the `i-th` event or excluding it.

```
for j in range(1, k + 1):
    f[i][j] = max(f[i - 1][j], f[p][j - 1] + val)
```

Here's an explanation of each case:

- `f[i - 1][j]` represents the value obtained by not attending the `i-th` event, hence just carrying over the value from the previous event iteration.
- `f[p][j - 1] + val` represents the value obtained by attending the `i-th` event. The value from the event right before the current event starts (`f[p][j - 1]`) is added to the current event's value (`val`).

6. Return the value `f[n][k]`, which is the answer to the problem. It represents the maximum sum that can be achieved by attending up to `k` events.

```
return f[n][k]
```

In this implementation, the data structure used is a 2D array/list for DP. Also, a binary search pattern is used for efficient searching of compatible events.

This DP approach ensures we are considering every combination of events we could possibly attend (up to `k` events) and updating our max sum at each step for each event choice. By doing this iteratively and considering all choices, we ensure that the result will contain the maximum sum that can be achieved under the given conditions.

Example Walkthrough

Let's illustrate the solution approach with a small example.

Suppose `events = [[1, 2, 4], [2, 3, 3], [3, 4, 2]]` and `k = 2`. We're allowed to attend up to 2 events to maximize the sum of the values.

Step 1: Sort the events by their end time. After sorting, the `events` array remains `[[1, 2, 4], [2, 3, 3], [3, 4, 2]]`.

Step 2: Initialize the DP table `f` with rows equivalent to the number of `events + 1` and columns `k + 1`.

```
1 f = [
2     [0, 0, 0], # no events
3     [0, 0, 0], # no events
4     [0, 0, 0], # no events
5     [0, 0, 0]  # no events
6 ]
```

Step 3: Begin looping over sorted events and enumerate the events starting from 1:

- The events are:
  - Event 0: [1, 2, 4]
  - Event 1: [2, 3, 3]
  - Event 2: [3, 4, 2]

Step 4: Use binary search to find the rightmost event that does not conflict with the current one. First, we check for each event manually.

- For Event 0, we look before event 0: no events.
- For Event 1, we look before event 1: event 0 ends before event 1 starts. `p = 0`
- For Event 2, we look before event 2: event 1 ends as event 2 starts, so only event 0 does not conflict. `p = 0`

Step 5: Inside the loop for `i` (events), loop for `j` (number of events to attend up to `k`). For each `j`, we compute the `max` of not attending the current event or attending it:

- For Event 0:
  - `j = 1`: `f[1][1] = max(f[0][1], f[0][0] + 4)` becomes `4`
  - `j = 2`: `f[1][2] = max(f[0][2], f[0][1] + 4)` becomes `4`
- For Event 1, consulting previous values:
  - `j = 1`: `f[2][1] = max(f[1][1], f[0][0] + 3)` stays `4`
  - `j = 2`: `f[2][2] = max(f[1][2], f[0][1] + 3)` becomes `7` (Event 0 + Event 1)
- For Event 2, considering previous values:
  - `j = 1`: `f[3][1] = max(f[2][1], f[0][0] + 2)` stays `4`
  - `j = 2`: `f[3][2] = max(f[2][2], f[0][1] + 2)` stays `7` (cannot attend Event 2 after Event 1 due to overlap)

The updated `f` table after processing would look like:

```
1 f = [
2     [0, 0, 0], # no events
3     [0, 4, 4], # no events, up to no2
4     [0, 4, 7], # no events, up to no2
5     [0, 4, 7]  # no events, up to no2
6 ]
```

Step 6: Return `f[n][k]` which is `7`, representing the maximum sum of values by attending up to `k=2` events.

Note that we were able to include the values of only the first two events. Ignoring the third event since attending would overlap with the second event. This gives us the maximum value of `7` for attending up to two events.

Python Solution

```
from bisect import bisect_left
from typing import List

class Solution:
    def max_value(self, events: List[List[int]], k: int) -> int:
        # First, we sort the events based on their ending time.
        events.sort(key=lambda event: event[1])

        # The number of events
        num_events = len(events)

        # Prepare a DP table with dimensions (number of events + 1) x (k + 1)
        dp_table = [[0 for _ in range(k + 1)] for _ in range(num_events + 1)]

        # Iterate through each event, starting with index 1 for convenience
        for i, (start_time, _, value) in enumerate(events, 1):
            # Find the index of the last event that does not overlap with the current event
            previous_index = bisect_left(events, start_time, hi=i - 1, key=lambda event: event[1])

            # Update DP table for each of the k attendances
            for j in range(1, k + 1):
                # The value is the max between not attending this event and attending it
                dp_table[i][j] = max(
                    dp_table[i - 1][j],  # Not picking current event
                    dp_table[previous_index][j - 1] + value  # Picking current event
                )

        # The maximum value for attending at most k events is stored at dp_table[num_events][k]
        return dp_table[num_events][k]

# The function 'max_value' can be called with a list of events and a number k to find the result.
```

Java Solution

```
class Solution {
    public int maxValue(int[][] events, int k) {
        // Sort the events array by the end time of each event.
        Arrays.sort(events, (a, b) -> a[1] - b[1]);
        int n = events.length;

        // Create a DP table f[][][] will hold the maximum value for the first i events using j events.
        int[][] dpTable = new int[n + 1][k + 1];

        // Fill in the DP table
        for (int i = 1; i <= n; ++i) {
            // Start time and end of the current event
            int startTime = events[i - 1][0];
            int value = events[i - 1][2];

            // Find the previous event that does not overlap with the current event
            int previous = binarySearch(events, startTime, i - 1);

            for (int j = 1; j <= k; ++j) {
                // The value by using j events including the i-th event is either:
                // 1) The same as using j events without including the i-th event, or
                // 2) The value of using j-1 events plus the value of the i-th event
                //    (when events are sorted up to the previously non-overlapping event).
                dpTable[i][j] = Math.max(dpTable[i - 1][j], dpTable[previous][j - 1] + value);
            }
        }

        // Return the maximum value achievable using k events
        return dpTable[n][k];
    }

    // Performs a binary search to find the index of the latest event that finishes before 'endTime'.
    private int binarySearch(int[][] events, int endTime, int high) {
        int low = 0;
        while (low < high) {
            int mid = (low + high) >> 1; // Equivalent to (low + high) / 2, but avoids overflow
            if (events[mid][1] >= endTime) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }
        return low;
    }
}
```

C++ Solution

```
#include <vector>
#include <algorithm>
#include <cstring>

class Solution {
public:
    int maxValue(vector<vector<int>>& events, int maxEvents) {
        // Sort the events based on their end time.
        sort(events.begin(), events.end(), [](const auto& event1, const auto& event2) {
            return event1[1] < event2[1];
        });

        int numEvents = events.size();
        int dpTable[events.size() + 1][maxEvents + 1];

        // Initialize the dp array with zero
        memset(dpTable, 0, sizeof(dpTable));

        // Loop through all events
        for (int i = 1; i <= numEvents; ++i) {
            // Get the start time and value of the current event
            int startTime = events[i - 1][0];
            int value = events[i - 1][2];

            // Create a vector that holds the start time
            vector<int> currentTime{startTime};

            // Find the latest event that finishes before the current event starts
            int lastNonOverlappingEventIndex = lower_bound(events.begin(), events.begin() + i - 1, currentTime, [](const auto& x, const auto& b) {
                return x[1] < b[0];
            }) - events.begin();

            // Loop through the numbers 1 to maxEvents
            for (int j = 1; j <= maxEvents; ++j) {
                // Update the dp array with the max value achieved by either
                // skipping the current event or attending it after the last non-overlapping event
                dpTable[i][j] = max(dpTable[i - 1][j], dpTable[lastNonOverlappingEventIndex][j - 1] + value);
            }
        }

        // Return the maximum value that can be achieved by attending maxEvents number of events
        return dpTable[numEvents][maxEvents];
    }
};
```

Typescript Solution

```
function maxValue(events: number[][], maxEvents: number): number {
    // Sort the events based on their end time
    events.sort((a, b) => a[1] - b[1]);

    // Number of events
    const numberOfEvents = events.length;

    // Initialize the dynamic programming table with 0's
    const dp: number[][] = new Array(numberOfEvents + 1)
                               .fill(0)
                               .map(() => new Array(maxEvents + 1).fill(0));

    // Function to search for the latest event that does not overlap
    const binarySearchLatestNonOverlappingEvent = (startTime: number, high: number): number => {
        let low = 0;
        let mid;
        while (low < high) {
            mid = (low + high) >> 1;
            if (events[mid][1] >= startTime) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }
        return low;
    };

    // Iterate over each event
    for (let i = 1; i <= numberOfEvents; ++i) {
        // Destructure the event details
        const [start, , value] = events[i - 1];

        // Find the latest event that does not overlap
        const lastNonOverlappingEventIndex = binarySearchLatestNonOverlappingEvent(start, i - 1);

        // Update dynamic programming table
        for (let j = 1; j <= maxEvents; ++j) {
            dp[i][j] = Math.max(dp[i - 1][j], dp[lastNonOverlappingEventIndex][j - 1] + value);
        }
    }

    // Return the maximum value that can be achieved by attending at most `maxEvents` events
    return dp[numberOfEvents][maxEvents];
}
```

Time and Space Complexity

The given code is a dynamic programming solution that solves a problem similar to the weighted job scheduling problem. The solution involves finding the maximum value that can be obtained by attending up to `k` non-overlapping events from a list of events given their start time, end time, and value.

Time Complexity

The time complexity of the provided code is determined by two main operations: sorting the events and populating the `f` table.

1. **Sorting**: The events list is sorted based on the end time using the `sort` function. If there are `n` events, the time complexity for sorting is $O(n \log n)$.
2. **DP Table Calculation**:
   - For each event `i` (from `1` to `n`), we perform a `bisect_left` which takes $O(\log i)$ time per event.
   - Then for each event `i`, we iterate through `j` from `1` to `k`, resulting in $O(k)$ operations per event.
   - Therefore, for all `n` events, we will have $O(nk \log n)$ work inside the loop for updating the `f` table.

Combining both steps, the total time complexity becomes $O(n \log n + nk \log n)$. Since both terms dominate for different ranges of `n` and `k`, we usually express this as $O(nk \log n)$ as it's the term that will be larger in the worst-case scenario (where `k` is large enough).

Space Complexity

The space complexity of the code is mainly determined by the size of the `f` table which is a two-dimensional list in Python.

- We have `n + 1` rows and `k + 1` columns in the `f` table ($[[0] * (k + 1) \text{ for } \_ \text{ in } \text{range}(n + 1)]$), thus the space complexity is $O(nk)$.

To summarize, the time complexity is $O(nk \log n)$ and the space complexity is $O(nk)$.