1751. Maximum Number of Events That Can Be Attended II Dynamic Programming Sorting Array Binary Search Hard

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

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.

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

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.

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

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

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 solution leverages the Python bisect_left function for binary search, and iteratively updates the dynamic programming table to arrive at the solution.

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:

The solution is implemented using Dynamic Programming (DP), which is a method for solving complex problems by breaking them

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

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

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

f[i][j] = max(f[i-1][j], f[p][j-1] + val)

current event starts (f[p][j - 1]) is added to the current event's value (val).

contain the maximum sum that can be achieved under the given conditions.

Solution Approach

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

used to store the maximum value that can be obtained by attending a certain number of events up to a certain point in time. 1 $f = [[0] * (k + 1) for _ in range(n + 1)]$ 3. Loop over sorted events, with each event enumerated as 1, 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.

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

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.

1 for j in range(1, k + 1):

Here's an explanation of each case:

previous event iteration.

compatible events.

the values.

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.

• f[i - 1][j] represents the value obtained by not attending the i-th event, hence just carrying over the value from the

of[p][j-1] + val represents the value obtained by attending the i-th event. The value from the event right before the

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. 1 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

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

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

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 n + 1 and columns k + 1. 1 f = I

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

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

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

manually.

· The events are:

Event 0: [1, 2, 4]

Event 1: [2, 3, 3]

Event 2: [3, 4, 2]

[0, 0, 0], # n=0 events

[0, 0, 0], # n=1 event

[0, 0, 0], # n=2 events

[0, 0, 0] # n=3 events

attending the current event or attending it: For Event 0:

 \circ j = 2: f[2][2] = max(f[1][2], f[0][1] + 3) becomes 7 (Event 0 + Event 1)

 \circ j = 1: f[1][1] = max(f[0][1], f[0][0] + 4) becomes 4

 \circ j = 2: f[1][2] = max(f[0][2], f[0][1] + 4) becomes 4

 \circ j = 1: f[2][1] = max(f[1][1], f[0][0] + 3) stays 4

 \circ j = 1: f[3][1] = max(f[2][1], f[0][0] + 2) stays 4

For Event 1, considering previous values:

For Event 2, considering previous values:

[0, 4, 4], # n=1 event, up to k=2

[0, 4, 7], # n=2 events, up to k=2

[0, 4, 7] # n=3 events, up to k=2

return dp_table[num_events][k]

int n = events.length;

// Fill in the DP table

return dpTable[n][k];

while (low < high) {

} else {

high = mid;

low = mid + 1;

int low = 0;

for (int i = 1; $i \ll n$; ++i) {

public int maxValue(int[][] events, int k) {

Arrays.sort(events, (a, b) -> a[1] - b[1]);

int[][] dpTable = new int[n + 1][k + 1];

int startTime = events[i - 1][0];

int value = events[i - 1][2];

for (int j = 1; $j \le k$; ++j) {

if (events[mid][1] >= endTime) {

// Sort the events array by the end time of each event.

// Start time and value of the current event

// Return the maximum value achievable using k events

private int binarySearch(int[][] events, int endTime, int high) {

int maxValue(vector<vector<int>>& events, int maxEvents) {

// Get the start time and value of the current event

sort(events.begin(), events.end(), [](const auto& event1, const auto& event2) {

// Sort the events based on their end time

return event1[1] < event2[1];</pre>

int dp[numEvents + 1][maxEvents + 1];

// Initialize the dp array with zero

for (int i = 1; i <= numEvents; ++i) {</pre>

int numEvents = events.size();

memset(dp, 0, sizeof(dp));

// Loop through all events

int previous = binarySearch(events, startTime, i - 1);

[0, 0, 0], # n=0 events

Python Solution

25

26

27

28

29

30

32

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

41

Java Solution

1 class Solution {

1 from bisect import bisect_left

from typing import List

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

 \circ 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:

Note that we were able to include the values of only the first two events, ignoring the third event since attending would overlap with

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

the second event. This gives us the maximum value of 7 for attending up to two events.

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]) 9 # The number of events 10 num_events = len(events) 11 12 # Prepare a DP table with dimensions (number of events + 1) x (k + 1)13 $dp_table = [[0] * (k + 1) for _ in range(num_events + 1)]$ 14 15 # Iterate through each event, starting with index 1 for convenience for i, (start_time, _, value) in enumerate(events, 1): 16 # Find the index of the last event that does not overlap with the current event 17 previous_index = bisect_left(events, start_time, hi=i - 1, key=lambda event: event[1]) 18 19 20 # Update DP table for each of the k attendances 21 for j in range(1, k + 1): 22 # The value is the max between not attending this event and attending it 23 dp_table[i][j] = max(24 dp_table[i - 1][j], # Not picking current event

// Create a DP table where f[i][j] will hold the maximum value for the first i events using j events.

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]

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

// Find the previous event that does not overlap with the current event

// The value for using j events including the i-th event is either:

// 2. The value of using j-1 events plus the value of the i-th event

// 1. The same as using the first i-1 events (not using the i-th event), or

dpTable[i][j] = Math.max(dpTable[i - 1][j], dpTable[previous][j - 1] + value);

// (when j-1 events are used up to the previously non-overlapping event).

// Performs a binary search to find the index of the latest event that finishes before 'endTime'.

int mid = (low + high) >> 1; // Equivalent to (low + high) / 2, but avoids overflow

42 43 return low; 44 45 46

C++ Solution

1 #include <vector>

2 #include <algorithm>

#include <cstring>

});

class Solution {

public:

9

10

11

12

13

14

15

16

17

18

19

20

21

32

33

34

35

36

37

38

39

40

41

42

43

44

45

22 int startTime = events[i - 1][0]; 23 int value = events[i - 1][2]; 24 25 // Create a vector that holds the start time 26 vector<int> currentTime = {startTime}; 27 // Find the last event that finishes before the current event starts int lastEventIndex = lower_bound(events.begin(), events.begin() + i - 1, currentTime, [](const auto& a, const auto& b) 29 return a[1] < b[0]; 30 31 }) - events.begin(); 32 33 // Loop through the numbers 1 to maxEvents for (int j = 1; j <= maxEvents; ++j) {</pre> 34 35 // Update the dp array with the max value achieved by either 36 // skipping the current event or attending it after the last non-overlapping event 37 dp[i][j] = max(dp[i - 1][j], dp[lastEventIndex][j - 1] + value);38 39 40 41 // Return the maximum value that can be achieved by attending maxEvents number of events 42 return dp[numEvents][maxEvents]; 43 44 }; 45 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 8 9 const dp: number[][] = new Array(numberOfEvents + 1) 10 .fill(0) 11 .map(() => new Array(maxEvents + 1).fill(0)); 12 13 // Function to search for the latest event that does not overlap 14 const binarySearchLatestNonOverlappingEvent = (startTime: number, high: number): number => { 15 let low = 0; let mid; while (low < high) { mid = (low + high) >> 1;18 if (events[mid][1] >= startTime) { 19 20 high = mid; 21 } else { 22 low = mid + 1;23 24 25 return low; }; 26 27 28 // Iterate over each event 29 for (let i = 1; i <= numberOfEvents; ++i) {</pre> // Destructure the event details 30 31 const [start, _, value] = events[i - 1];

const latestNonOverlappingEventIndex = binarySearchLatestNonOverlappingEvent(start, i - 1);

dp[i][j] = Math.max(dp[i - 1][j], dp[latestNonOverlappingEventIndex][j - 1] + value);

// Return the maximum value that can be obtained by attending at most 'maxEvents' events

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 and Space Complexity

// Find the latest event that does not overlap

// Update dynamic programming table

return dp[numberOfEvents][maxEvents];

for (let j = 1; j <= maxEvents; ++j) {</pre>

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:

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

For each event i (from 1 to n), we perform a bisect_left which takes 0(log i) time per event.

Then for each event i, we iterate through j from 1 to k, resulting in 0(k) operations per event.

The given code is a dynamic programming solution that solves a problem similar to the weighted job scheduling problem. The

• 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 $0(n \log n) + 0(nk \log n)$. 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

Time Complexity

large enough). Space Complexity

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

• We have n + 1 rows and k + 1 columns in the f table ([[0] * (k + 1) for _ in range(n + 1)]), thus the space complexity is

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