2054. Two Best Non-Overlapping Events Medium **Dynamic Programming** Heap (Priority Queue) Array Sorting **Binary Search**

searching of the next non-overlapping event using binary search.

Leetcode Link

You are provided with a list of events, each represented by a trio of integers: the start time, the end time, and the event's value. The

goal is to maximize the total value by attending up to two non-overlapping events. It's important to note that if two events share a start or end time, they are considered overlapping and thus cannot both be attended; to attend consecutive events, the next event must start after the previous event has ended.

Intuition

Problem Description

events. For any given event, the best strategy is to attend the event itself and then add it to the value of the next non-overlapping event (if

any) that yields the maximum value. To simplify this process, the events are sorted by their start time, which allows for efficient

To tackle this problem, think of it as two separate scenarios: either you attend only one event or you attend two non-overlapping

To streamline the search for the maximum value of a non-overlapping event that starts after a given end time, you precalculate and store the outcome in an array f, avoiding the need to compute it multiple times. This array holds the maximum event value from the current event to the last event. By updating this array from the end towards the start, you ensure that f[i] represents the maximum

event value from event i to the end. This approach enables you to easily find the maximum value that can be added to the current

event value. When considering a particular event, you find the next non-overlapping event by conducting a binary search to locate the index of the first event that starts after the current event's end time. Using binary search is efficient here due to the events being sorted. Once this index is obtained, you can add the value of the current event with the value stored at this index in your precalculated f array to get the total value if you were to attend both events.

Finally, you compare the value obtained by attending the current event and possibly the next non-overlapping event to the previously calculated maximum sum and continually update this maximum. This iterative approach ensures that by the time you

finish examining all events, you have determined the maximum total value that can be achieved by attending up to two nonoverlapping events. The solution capitalizes on sorted event start times and binary search for efficiency, combined with dynamic programming to precalculate possible future values, ensuring an optimized and speedy result.

1. Sorting: First, we sort the events by their start time. This step is crucial for the binary search that follows and ensures that when we look for the next non-overlapping event, we can do so efficiently.

2. Dynamic Programming: We prepare an array f which will, at each position i, store the maximum value of an event that starts

from i until the end of the array. This array represents the best future event value we can get if we decide to attend an event

starting from any position i. To populate this array, we iterate from the end of the list backward, constantly keeping track of the

The solution is built around a smart combination of sorting, binary search, dynamic programming, and greedy approach.

highest value seen so far.

Solution Approach

1 f = [events[-1][2]] * n2 for i in range(n - 2, -1, -1):

f[i] = max(f[i + 1], events[i][2])3. Greedy Approach: For each event, we consider the maximum sum we can get by attending the current event and then look

ahead to find the next possible non-overlapping event that we could attend. We use a greedy approach to always pick the next

best choice without considering the broader problem. 1 ans = 02 for _, e, v in events: idx = bisect_right(events, e, key=lambda x: x[0]) if idx < n: v += f[idx]ans = max(ans, v)

4. Binary Search: For finding the index of the first event that starts after the current event ends, we use the bisect_right function.

This standard library function implements a binary search algorithm that returns the index of the first element in the events that

```
maximum sum obtained by attending the current event and the best possible next non-overlapping event. We update the
maximum answer we have seen so far:
1 if idx < n:
      v += f[idx]
3 \text{ ans} = \max(\text{ans, v})
```

Let's illustrate the solution with a small example. Suppose we have the following events, where each event is a trio of integers (start

1. Sorting: We sort the events by their start time. However, the events are already sorted in this example, so we don't need to sort

By the end of the loop, ans will hold the maximum possible sum of the values of at most two non-overlapping events.

5. Final Calculation and Iteration: As we iterate over each event, for the current event, we set v to be its value, then we add to v

the value stored in f at the idx position found using binary search if it's within bounds. This total value of v now represents the

```
2. Dynamic Programming: We initialize the f array to prepare for maximum future event values. Since we have 4 events, our array
  will have 4 positions:
```

Now we populate f from right to left, keeping track of the highest value:

2 idx = bisect_right(events, 3, key=lambda x: x[0]) # idx for event (1, 3, 5)

4. Binary Search: We use bisect_right to efficiently find the non-overlapping event:

1 idx = bisect_right(events, e, key=lambda x: x[0])

def maxTwoEvents(self, events: List[List[int]]) -> int:

Initialize maximum value to be zero at the start.

max_value_after = [events[-1][2]] * n

for i in range(n - 2, -1, -1):

for _, end_time, value in events:

combined_value = value

if (left < num0fEvents) {</pre>

return maxTotalValue;

// Number of events

int ans = 0;

int n = events.size();

value += maxValueAfter[left];

int maxTwoEvents(vector<vector<int>>& events) {

// Sort events based on starting time

sort(events.begin(), events.end());

vector<int> future_max(n + 1);

// Initialize answer to zero

// Iterate over all events

for (auto& event : events) {

int value = event[2];

// Value of the current event

// Binary search boundaries

// after the current event ends

int mid = (left + right) / 2;

int left = 0, right = n;

while (left < right) {</pre>

} else {

for (int i = n - 1; i >= 0; --i) {

// Update the maximum total value if needed

maxTotalValue = Math.max(maxTotalValue, value);

// Future Max Value Array (f): stores the maximum value for events from i to n

// Perform binary search to find the smallest index of event starting

if (events[mid][0] > event[1]) { // event[mid] start time is after event finish time

// Build future_max array from the end to the start (reverse direction)

future_max[i] = max(future_max[i + 1], events[i][2]);

right = mid; // search in the left half

left = mid + 1; // search in the right half

Iterate over each event

1T 1dx < n:

else:

within bounds. Then update ans:

v = events[i][2] + f[idx]

v = events[0][2] + f[idx] # v = 5 (current value) + 6 (next non-overlap)

1 f = [5, 5, 5, 4] # Initialized with the last event's value

1 events = [(1, 3, 5), (2, 5, 6), (4, 6, 5), (7, 8, 4)]

Example Walkthrough

time, end time, value):

them again.

2 f = [0] * n

1 n = len(events) # n is 4

non-overlapping event:

1 ans = 0

3 if idx < n:

1 if idx < n:

 $3 \text{ ans} = \max(\text{ans}, v)$

Python Solution

class Solution:

10

12

13

14

15

16

17

20

22

23

24

27

28

29

30

31

32

33

34

33

34

35

36

37

38

39

40

42

41 }

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

39

40

41

42

43

44

45

46

47

48

49

50

51

});

return answer;

C++ Solution

public:

1 class Solution {

from bisect import bisect_right

max_value = 0

is greater than the e, which is the ending time of the current event:

1 idx = bisect_right(events, e, key=lambda x: x[0])

f[2] = max(f[3], events[2][2]) # f[2] is max(4, 5)f[1] = max(f[2], events[1][2]) # f[1] is max(5, 6)f[0] = max(f[1], events[0][2]) # f[0] is max(6, 5)After iteration, f becomes [6, 6, 5, 4].

3. Greedy Approach: We now iterate through events and use a greedy approach to sum values of the current event and the next

```
ans = max(ans, v) # ans is max(0, 11)
7 idx = bisect_right(events, 5, key=lambda x: x[0]) # idx for event (2, 5, 6)
8 # The rest of the events start after event (2, 5, 6) ends.
And so on for the remaining events. We continue updating ans with the maximum values obtained.
```

5. Final Calculation and Iteration: We update v with the sum of the current event's value and the max future value if the index is

After finishing the loop, ans will be the maximum sum of values that can be achieved by attending up to two non-overlapping events.

For this example, ans would be the maximum value obtained, demonstrating which events to choose to maximize the value.

Sort the events based on their start times. events.sort() n = len(events)

'max_value_after' holds the maximum value of any single event from index i to the end.

Fill 'max_value_after' by iterating from the second last to the first event.

max_value_after[i] = max(max_value_after[i + 1], events[i][2])

Find the first event that starts after the current event ends.

If such an event is found, add the value of the current event to

Update the maximum value with the larger of the two values.

Return the maximum value found which could be from two or one events.

idx = bisect_right(events, end_time, key=lambda x: x[0])

the maximum value found after the current event.

max_value = max(max_value, combined_value)

combined_value = value + max_value_after[idx]

```
35
           return max_value
36
37 # Note: Definition of 'List' is not given in the code, presumably it should
  # be imported from 'typing' (from typing import List) for the type annotations to work.
39
```

```
Java Solution
   class Solution {
       public int maxTwoEvents(int[][] events) {
           // Sort events by their start time
           Arrays.sort(events, (a, b) \rightarrow a[0] - b[0]);
            int numOfEvents = events.length;
           // 'maxValueAfter' array will store the maximum value from current event to the last event
           int[] maxValueAfter = new int[numOfEvents + 1];
            for (int i = numOfEvents - 1; i >= 0; --i) {
9
                maxValueAfter[i] = Math.max(maxValueAfter[i + 1], events[i][2]);
10
11
12
13
            int maxTotalValue = 0;
14
            for (int[] event : events) {
15
                int value = event[2]; // Value of the current event
16
17
               // Binary search to find the first event that starts after the current event ends
18
               int left = 0, right = numOfEvents;
20
                while (left < right) {</pre>
                    int mid = (left + right) >> 1;
21
22
                    if (events[mid][0] > event[1]) {
23
                        // If the event at 'mid' starts after current event ends, search in left half
24
                        right = mid;
25
                    } else {
                        // Otherwise search in the right half
26
27
                        left = mid + 1;
28
29
30
               // If there is an event that starts after the current one, add its value
31
```

36 37 38 39 40

```
// If there is a future event that does not overlap with the current event
 41
                 if (left < n) {
 42
                     value += future_max[left]; // Add max future event value to the current event value
 43
 44
 45
                 // Update the maximum value answer with the max value of the single event or
 46
                 // the current event paired with a max future event
 47
                 ans = max(ans, value);
 48
 49
             // Return the maximum value obtainable by attending at most two non-overlapping events
 50
 51
             return ans;
 52
 53 };
 54
Typescript Solution
     function maxTwoEvents(events: number[][]): number {
         // Sort events based on their starting time
         events.sort((a, b) => a[0] - b[0]);
  5
        // Number of events
         const n: number = events.length;
  8
         // Future max value array: stores the max value for events from index i to n
         const futureMax: number[] = new Array(n + 1).fill(0);
  9
 10
         // Build futureMax array from the end towards the start (in reverse direction)
 11
 12
         for (let i = n - 1; i >= 0; --i) {
 13
             futureMax[i] = Math.max(futureMax[i + 1], events[i][2]);
 14
 15
 16
        // Initialize the answer to zero
         let answer: number = 0;
 17
 18
 19
         // Iterate over all events
         events.forEach((event) => {
 20
 21
             // Value of the current event
 22
             let value = event[2];
 23
 24
             // Binary search boundaries
 25
             let left: number = 0, right: number = n;
 26
 27
             // Perform a binary search to find the smallest index of an event that starts
 28
             // after the current event ends
             while (left < right) {</pre>
 29
                 let mid = Math.floor((left + right) / 2);
 30
                 if (events[mid][0] > event[1]) { // event[mid] start time is after the current event's end time
 31
 32
                     right = mid; // Search in the left half
 33
                 } else {
                     left = mid + 1; // Search in the right half
 34
 35
 36
 37
 38
             // If there is a future event that does not overlap with the current event
```

The given code is designed to find the maximum value that can be obtained by attending at most two events, where events is a list of event intervals, each in the format [start, end, value]. Here's the analysis of its time and space complexity:

Time and Space Complexity

if (left < n) {

1. Sorting events: The initial sorting of the event list events.sort() has a time complexity of O(n log n), where n is the number of events.

2. Backward traversal to fill f: The loop that fills the list f with the maximum value of the events that come after each event has a

4. Summing up the time complexities from the above points, we have $0(n \log n) + 0(n) + 0(n \log n)$, which simplifies to $0(n \log n)$

value += futureMax[left]; // Add max future event value to the current event's value

// Update the maximum value answer with the max value of a single event or

// Return the maximum value obtainable by attending at most two non-overlapping events

// the current event paired with a max future event

time complexity of O(n) as it goes through the list of events once.

n) since the log n terms dominate the linear term.

answer = Math.max(answer, value);

Time Complexity

- 3. Binary search using bisect_right: In the worst case, the binary search is called for each event to find the index idx. Since bisect_right has a complexity of O(log n), and it's called inside a loop that runs n times, the total time complexity for this part is O(n log n).
- **Space Complexity** 1. Auxiliary list f: The space complexity is O(n) because of the additional list f, which has the same length as the list of events.
 - 2. Constant space: No other significant space-consuming structures are used, so we only consider the space for f.
- In conclusion, the total time complexity of the code is $0(n \log n)$ and the total space complexity is 0(n).