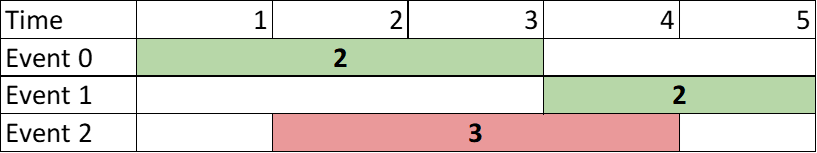
<https://leetcode.com/problems/two-best-non-overlapping-events/description/>

You are given a **0-indexed** 2D integer array of events where events[i] = [startTimei, endTimei, valuei]. The ith event starts at startTimei and ends at endTimei, and if you attend this event, you will receive a value of valuei. You can choose **at most** **two** **non-overlapping** events to attend such that the sum of their values is **maximized**.

Return **this maximum sum.**

Note that the start time and end time is **inclusive**: that is, you cannot attend two events where one of them starts and the other ends at the same time. More specifically, if you attend an event with end time t, the next event must start at or after t + 1.

**Example 1:**

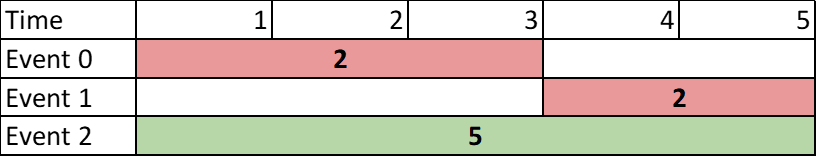


**Input:** events = [[1,3,2],[4,5,2],[2,4,3]]

**Output:** 4

**Explanation:** Choose the green events, 0 and 1 for a sum of 2 + 2 = 4.

**Example 2:**

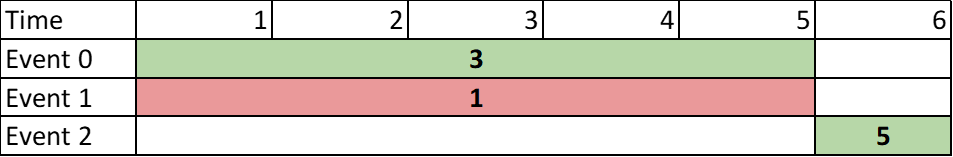


**Input:** events = [[1,3,2],[4,5,2],[1,5,5]]

**Output:** 5

**Explanation:** Choose event 2 for a sum of 5.

**Example 3:**



**Input:** events = [[1,5,3],[1,5,1],[6,6,5]]

**Output:** 8

**Explanation:** Choose events 0 and 2 for a sum of 3 + 5 = 8.

**Constraints:**

* 2 <= events.length <= 10^5
* events[i].length == 3
* 1 <= startTimei <= endTimei <= 10^9
* 1 <= valuei <= 10^6

**Attempt 1: 2023-12-19**

**本题就比L2008.Maximum Earnings From Taxi或者L1235.Maximum Profit in Job Scheduling多出一个条件，就是**You can choose **at most** **two** **non-overlapping** events to attend such that the sum of their values is **maximized，意味着我们只能选择2个events，而不是L2008，L1235中可以无限选择，所以在Native DFS中加入一个条件"count == 2"，注意，不是"count > 2" !!!**

**Solution 1: Native DFS (30min, TLE 35/63)**

**Wrong Solution**

class Solution {

    public int maxTwoEvents(int[][] events) {

        Arrays.sort(events, (a, b) -> a[0] - b[0]);

        return helper(events, 0, 0);

    }

    private int helper(int[][] events, int index, int count) {

        if(index == events.length || count > 2) {

            return 0;

        }

        int notTake = helper(events, index + 1, count);

        int i;

        for(i = index + 1; i < events.length; i++) {

            if(events[i][0] > events[index][1]) {

                break;

            }

        }

        int take = events[index][2] + helper(events, i, count + 1);

        return Math.max(notTake, take);

    }

}

**Step by Step**

**下图解释了2点：**

**1. 为何在recursion base condition中count == 2是必须的，而count > 2是错的**

**2.在recursion branch上有很多重复的部分可以用memo或者dp的方式存储，不用每次都计算**



**Correct Solution**

class Solution {

    public int maxTwoEvents(int[][] events) {

        Arrays.sort(events, (a, b) -> a[0] - b[0]);

        return helper(events, 0, 0);

    }

    private int helper(int[][] events, int index, int count) {

        if(index == events.length || count == 2) {

            return 0;

        }

        int notTake = helper(events, index + 1, count);

        int i;

        for(i = index + 1; i < events.length; i++) {

            if(events[i][0] > events[index][1]) {

                break;

            }

        }

        int take = events[index][2] + helper(events, i, count + 1);

        return Math.max(notTake, take);

    }

}

Time Complexity: O(2^m), m is the number of events.

Each recursion level we will take two branches, "not take event" and "take event", so worst case is we have to go through m depth recursion, then time complexity is O(2^m)

Space Complexity: O(m)

Recursion Stack: The depth of recursion may go up to m in the worst case if all events are taken sequentially with no overlap, which contributes to O(m) space complexity.

**Solution 2: DFS + Memoization (10min, TLE 45/63)**

class Solution {

    public int maxTwoEvents(int[][] events) {

        Arrays.sort(events, (a, b) -> a[0] - b[0]);

        // If not limit we can only choose 2 events, based on

        // 2 <= events.length <= 10^5 and 1 <= valuei <= 10^6

        // the 'Integer' memo cannot handle, but since we only

        // choose 2 events, it will never pass 2 \* 10^6, we

        // will still use 'Integer' memo

        // The second dimension of memo is 'count'(0,1,2)

        Integer[][] memo = new Integer[events.length][3];

        return helper(events, 0, 0, memo);

    }

    private int helper(int[][] events, int index, int count, Integer[][] memo) {

        if(index == events.length || count == 2) {

            return 0;

        }

        if(memo[index][count] != null) {

            return memo[index][count];

        }

        int notTake = helper(events, index + 1, count, memo);

        int i;

        for(i = index + 1; i < events.length; i++) {

            if(events[i][0] > events[index][1]) {

                break;

            }

        }

        int take = events[index][2] + helper(events, i, count + 1, memo);

        return memo[index][count] = Math.max(notTake, take);

    }

}

Time Complexity: O(m^2), m is the number of events

First m is for worst case may need m recursion on depth, second m is for each recursion internally we have a for loop to find the next recursion entry

Space Complexity: O(m\*3), m is the number of events

Memoization: The dfs function uses a memoization, which will store results for each unique argument(index) and (count). This could result in up to m\*3(3 comes from required condition as 'count' will be only three scenarios as 0, 1, 2) entries in the worst case, giving a space complexity of O(m\*3).

Recursion Stack: The depth of recursion may go up to m in the worst case if all events are taken sequentially with no overlap, which contributes to O(m) space complexity.

So Memoization cost more than Recursion Stack as O(m\*3), final cost is O(m\*3)

**Solution 3: DFS + Memoization + Binary Search (10min)**

**注意：与L2008，L1235的区别在于这里是Find Upper Boundary，因为关系是严格大于某个值，而不是大于等于某个值，相当于找到最右边的大于等于的值的坐标还要再向右移动一个位置才能获得严格大于某个值的效果**

class Solution {

    public int maxTwoEvents(int[][] events) {

        Arrays.sort(events, (a, b) -> a[0] - b[0]);

        // If not limit we can only choose 2 events, based on

        // 2 <= events.length <= 10^5 and 1 <= valuei <= 10^6

        // the 'Integer' memo cannot handle, but since we only

        // choose 2 events, it will never pass 2 \* 10^6, we

        // will still use 'Integer' memo

        // The second dimension of memo is 'count'(0,1,2)

        Integer[][] memo = new Integer[events.length][3];

        return helper(events, 0, 0, memo);

    }

    private int helper(int[][] events, int index, int count, Integer[][] memo) {

        if(index == events.length || count == 2) {

            return 0;

        }

        if(memo[index][count] != null) {

            return memo[index][count];

        }

        int notTake = helper(events, index + 1, count, memo);

        int i = binarySearch(events, events[index][1], index + 1);

        int take = events[index][2] + helper(events, i, count + 1, memo);

        return memo[index][count] = Math.max(notTake, take);

    }

    // Find upper boundary and plus 1 for strict larger than '>' relation

    private int binarySearch(int[][] events, int val, int lo) {

        int hi = events.length - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(events[mid][0] > val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // The 'lo - 1'(or 'hi') is upper boundary when '>=' target value,

        // the '+1' for strict larger than '>' target value

        return lo - 1 + 1;

    }

}

Time Complexity: O(mlogm), m is the number of events

Sorting the Rides: The initial step in the code is to sort the events, which consists of m events. This has a time complexity of O(m log m), where m is the number of events.

Dynamic Programming with Memoization (dfs function): The dfs function is used here to implement dynamic programming with memoization. The memoization ensures that each possible starting point of a event is computed at most once. Since memoization caches the results, the maximum number of distinct states to consider is m, the length of the sorted events list.

Binary Search (bisect\_left function): Inside the dfs function, binary search is used to find the next non-overlapping event, which has a time complexity of O(log m) per call.

Combining the dynamic programming computation with the binary search, for each of the m calls to dfs, a binary search operation is involved. Thus, each call may contribute up to O(log m) complexity. Given that there are m such calls, the total time complexity from the dynamic programming along with the binary searches is O(m log m).

Considering both sorting and the memoized dfs, the overall time complexity of the algorithm is O(m log m).

Space Complexity: O(m\*3), m is the number of events

Sorting: Sorting is done in-place, but it may still require O(log m) space for the stack frames used in the sort implementation.

Memoization: The dfs function uses a memoization, which will store results for each unique argument(index) and (count). This could result in up to m\*3(3 comes from required condition as 'count' will be only three scenarios as 0, 1, 2) entries in the worst case, giving a space complexity of O(m\*3).

Recursion Stack: The depth of recursion may go up to m in the worst case if all events are taken sequentially with no overlap, which contributes to O(m) space complexity.

Thus, combining the space requirements for the sorting, memoization, and recursion stack, the total space complexity is O(m\*3).

In conclusion, the time complexity of the code is O(m log m) and the space complexity is O(m\*3), where m is the number of events.

**Solution 4: DP (10min, TLE 45/63)**

class Solution {

    public int maxTwoEvents(int[][] events) {

        Arrays.sort(events, (a, b) -> a[0] - b[0]);

        // In DFS, top {index,conut} = {0,0} bottom {index,conut} = {events.length,2}

        int n = events.length;

        int[][] dp = new int[n + 1][3];

        dp[n][2] = 0;

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

            // Additional loop check than L2008, L1235 because of required condition:

            // You can choose at most two non-overlapping events to attend

            for(int count = 1; count >= 0; count--) {

                int i;

                for(i = index + 1; i < n; i++) {

                    if(events[i][0] > events[index][1]) {

                        break;

                    }

                }

                // 1. Not take current event -> dp[index + 1][count]

                // 2. Take current event -> events[index][2] + dp[i][count + 1]

                dp[index][count] = Math.max(dp[index + 1][count], events[index][2] + dp[i][count + 1]);

            }

        }

        return dp[0][0];

    }

}

Time Complexity: O(m^2\*2), m is the number of events

First m^2 is for worst case may need m recursion on two for loop to find the next recursion entry, the 2 is for additional 'count' limitation related loop, we will only loop on 2 scenarios as count = 1 or 0

Space Complexity: O(m\*3), m is the number of events, 3 comes from required condition as 'count' will be only three scenarios as 0, 1, 2

**Solution 5: DP + Binary Search (10min)**

class Solution {

    public int maxTwoEvents(int[][] events) {

        Arrays.sort(events, (a, b) -> a[0] - b[0]);

        // In DFS, top {index,conut} = {0,0} bottom {index,conut} = {events.length,2}

        int n = events.length;

        int[][] dp = new int[n + 1][3];

        dp[n][2] = 0;

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

            // Additional loop check than L2008, L1235 because of required condition:

            // You can choose at most two non-overlapping events to attend

            for(int count = 1; count >= 0; count--) {

                int i = binarySearch(events, events[index][1], index + 1);

                // 1. Not take current event -> dp[index + 1][count]

                // 2. Take current event -> events[index][2] + dp[i][count + 1]

                dp[index][count] = Math.max(dp[index + 1][count], events[index][2] + dp[i][count + 1]);

            }

        }

        return dp[0][0];

    }

    // Find upper boundary and plus 1 for strict larger than '>' relation

    private int binarySearch(int[][] events, int val, int lo) {

        int hi = events.length - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(events[mid][0] > val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // The 'lo - 1'(or 'hi') is upper boundary when '>=' target value,

        // the '+1' for strict larger than '>' target value

        return lo - 1 + 1;

    }

}

Time Complexity: O(mlogm\*2), m is the number of events

First mlogm is for worst case may need m recursion on a for loop with a binary search to find the next recursion entry, the 2 is for additional 'count' limitation related loop, we will only loop on 2 scenarios as count = 1 or 0

Space Complexity: O(m\*3), m is the number of events, 3 comes from required condition as 'count' will be only three scenarios as 0, 1, 2

**Refer to**

<https://leetcode.com/problems/two-best-non-overlapping-events/solutions/2645906/c-recursion-memoization-tabulation-01-knapsack/>

**Refer to**

<https://algo.monster/liteproblems/2054>

**这题直接看L2008的解释，有区别但不多**