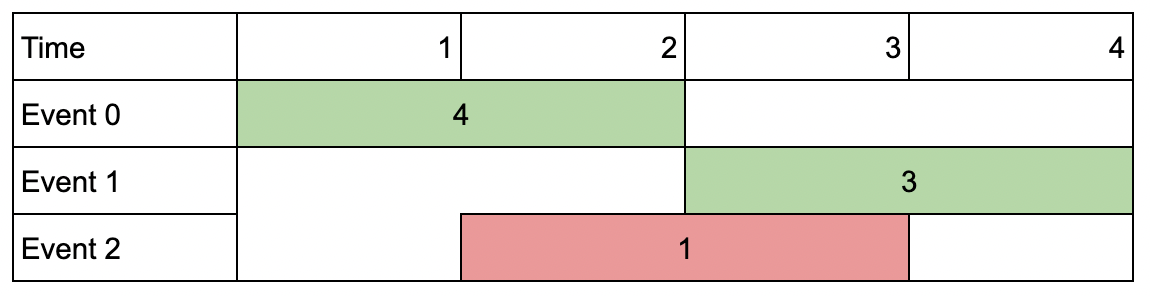
<https://leetcode.com/problems/maximum-number-of-events-that-can-be-attended-ii/description/>

You are given an array of events where events[i] = [startDayi, endDayi, valuei]. The ith event starts at startDayi and ends at endDayi, and if you attend this event, you will receive a value of valuei. You are also given an integer k which represents the maximum number of events you can attend.

You can only attend one event at a time. If you choose to attend an event, you must attend the **entire** event. Note that the end day is **inclusive**: that is, you cannot attend two events where one of them starts and the other ends on the same day.

Return **the maximum sum of values that you can receive by attending events.**

**Example 1:**

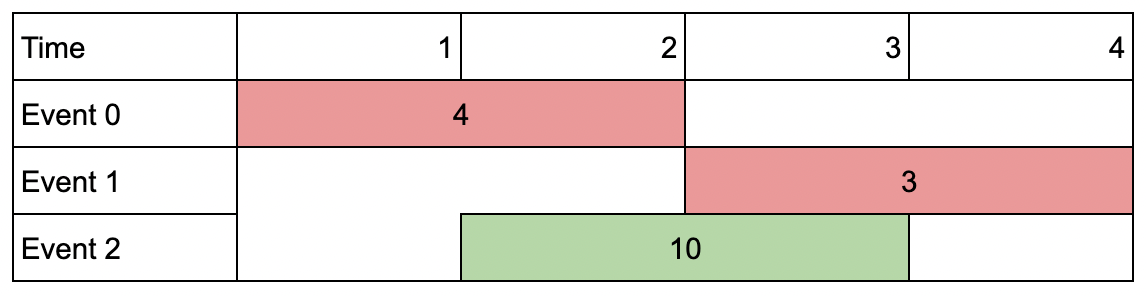


**Input:** events = [[1,2,4],[3,4,3],[2,3,1]], k = 2

**Output:** 7

**Explanation:** Choose the green events, 0 and 1 (0-indexed) for a total value of 4 + 3 = 7.

**Example 2:**

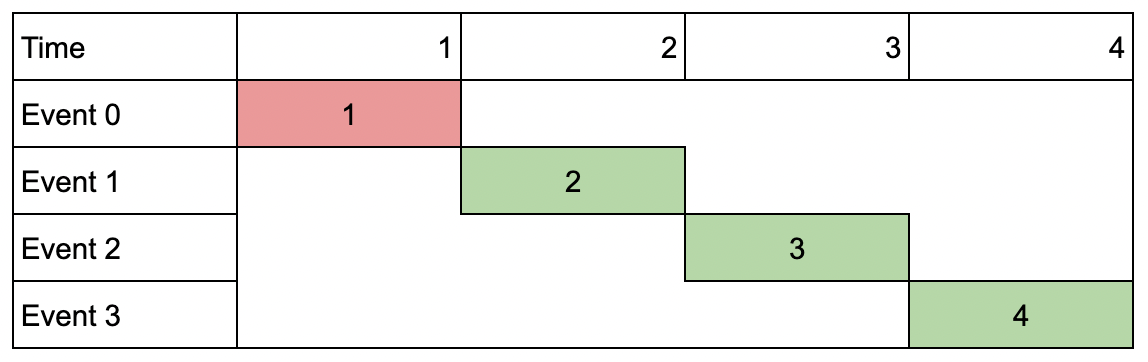


**Input:** events = [[1,2,4],[3,4,3],[2,3,10]], k = 2

**Output:** 10

**Explanation:** Choose event 2 for a total value of 10.Notice that you cannot attend any other event as they overlap, and that you do **not** have to attend k events.

**Example 3:**



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

**Output:** 9

**Explanation:** Although the events do not overlap, you can only attend 3 events. Pick the highest valued three.

**Constraints:**

* 1 <= k <= events.length
* 1 <= k \* events.length <= 10^6
* 1 <= startDayi <= endDayi <= 10^9
* 1 <= valuei <= 10^6

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

**本题和L2054.Two Best Non-Overlapping Events (Ref.L2008). 完全一致**

**Solution 1: Native DFS (10min, TLE 64/69)**

class Solution {

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

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

        return helper(events, 0, k);

    }

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

        if(index == events.length || count == 0) {

            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)**

class Solution {

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

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

        Integer[][] memo = new Integer[events.length + 1][k + 1];

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

    }

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

        if(index == events.length || count == 0) {

            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\*(k + 1)), 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\*(k + 1)(k comes from required condition as 'count' will be 'k + 1' scenarios) entries in the worst case, giving a space complexity of O(m\*(k + 1)).

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\*(k + 1)), final cost is O(m\*(k + 1))

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

class Solution {

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

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

        Integer[][] memo = new Integer[events.length + 1][k + 1];

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

    }

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

        if(index == events.length || count == 0) {

            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 target

    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;

            }

        }

        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\*(k + 1)), 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\*(k + 1)(k comes from required condition as 'count' will be 'k + 1' scenarios) entries in the worst case, giving a space complexity of O(m\*(k + 1)).

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\*(k + 1)).

In conclusion, the time complexity of the code is O(m log m) and the space complexity is O(m\*(k + 1)), where m is the number of events.

**Solution 4: DP (10min)**

class Solution {

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

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

        int n = events.length;

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

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

        dp[n][0] = 0;

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

            // Additional loop check than L2008, L1235 (same as L2054) because of

            // required condition:

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

            for(int count = 1; count <= k; 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][k];

    }

}

Time Complexity: O(m^2\*k), 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 k is for additional 'count' limitation related loop, we will only loop on k scenarios

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

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

class Solution {

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

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

        int n = events.length;

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

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

        dp[n][0] = 0;

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

            // Additional loop check than L2008, L1235 (same as L2054) because of

            // required condition:

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

            for(int count = 1; count <= k; 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][k];

    }

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

    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;

            }

        }

        return lo - 1 + 1;

    }

}

Time Complexity: O(mlogm\*k), 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 k is for additional 'count' limitation related loop, we will only loop on k scenarios

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

**这题直接看L2054的解释，没有区别**