<https://leetcode.com/problems/describe-the-painting/description/>

There is a long and thin painting that can be represented by a number line. The painting was painted with multiple overlapping segments where each segment was painted with a **unique** color. You are given a 2D integer array segments, where segments[i] = [starti, endi, colori] represents the **half-closed segment** [starti, endi) with colori as the color.

The colors in the overlapping segments of the painting were **mixed** when it was painted. When two or more colors mix, they form a new color that can be represented as a **set** of mixed colors.

* For example, if colors 2, 4, and 6 are mixed, then the resulting mixed color is {2,4,6}.

For the sake of simplicity, you should only output the **sum** of the elements in the set rather than the full set.

You want to **describe** the painting with the **minimum** number of non-overlapping **half-closed segments** of these mixed colors. These segments can be represented by the 2D array painting where painting[j] = [leftj, rightj, mixj] describes a **half-closed segment** [leftj, rightj) with the mixed color **sum** of mixj.

For example, the painting created with segments = [[1,4,5],[1,7,7]] can be described by painting = [[1,4,12],[4,7,7]] because:

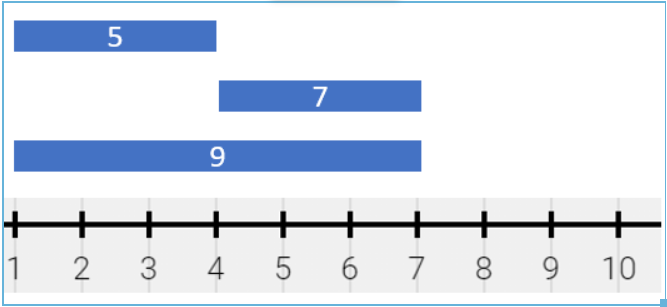
[1,4) is colored {5,7} (with a sum of 12) from both the first and second segments.

[4,7) is colored {7} from only the second segment.

Return *the 2D array* painting *describing the finished painting (excluding any parts that are* ***not*** *painted). You may return the segments in* ***any order***.

A **half-closed segment** [a, b) is the section of the number line between points a and b **including** point a and **not including** point b.

**Example 1:**



Input: segments = [[1,4,5],[4,7,7],[1,7,9]]

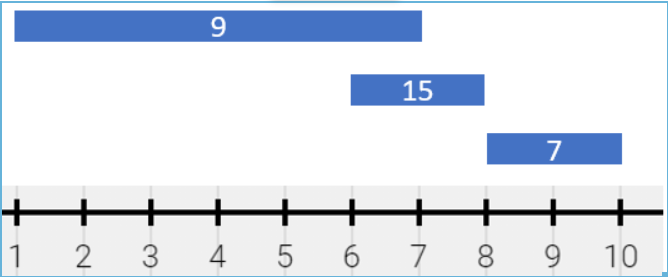
Output: [[1,4,14],[4,7,16]]

Explanation: The painting can be described as follows:

- [1,4) is colored {5,9} (with a sum of 14) from the first and third segments.

- [4,7) is colored {7,9} (with a sum of 16) from the second and third segments.

**Example 2:**



Input: segments = [[1,7,9],[6,8,15],[8,10,7]]

Output: [[1,6,9],[6,7,24],[7,8,15],[8,10,7]]

Explanation: The painting can be described as follows:

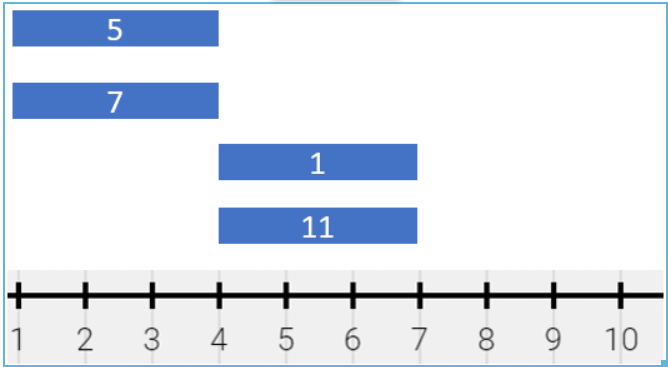
- [1,6) is colored 9 from the first segment.

- [6,7) is colored {9,15} (with a sum of 24) from the first and second segments.

- [7,8) is colored 15 from the second segment.

- [8,10) is colored 7 from the third segment.

**Example 3:**



Input: segments = [[1,4,5],[1,4,7],[4,7,1],[4,7,11]]

Output: [[1,4,12],[4,7,12]]

Explanation: The painting can be described as follows:

- [1,4) is colored {5,7} (with a sum of 12) from the first and second segments.

- [4,7) is colored {1,11} (with a sum of 12) from the third and fourth segments.

Note that returning a single segment [1,7) is incorrect because the mixed color sets are different.

**Constraints:**

* 1 <= segments.length <= 2 \* 10^4
* segments[i].length == 3
* 1 <= starti < endi <= 10^5
* 1 <= colori <= 10^9
* Each colori is distinct.

**Attempt 1: 2023-12-07**

**Solution 1: Sweep Line + Sorting (Treemap) + Two Pointers (30 min)**

class Solution {

public List<List<Long>> splitPainting(int[][] segments) {

// Sort all indexes with treemap

TreeMap<Integer, Long> map = new TreeMap<>();

for(int[] segment : segments) {

map.put(segment[0], map.getOrDefault(segment[0], (long)0) + segment[2]);

map.put(segment[1], map.getOrDefault(segment[1], (long)0) - segment[2]);

}

List<List<Long>> result = new ArrayList<>();

// 'count' is running count as presum

long count = 0;

// 'start' initialized as -1 as flag block adding section

// when 'start' & 'end' not filled by indexes recorded

// in treemap

long start = -1;

long end = -1;

for(Map.Entry<Integer, Long> e : map.entrySet()) {

// The tricky point is assigning 'end' first, and

// similar like find 'max' & 'second max' or like

// two pointers strategy 'faster' & 'slower', we

// have to assign 'max' and 'faster' first, then

// assign 'second max' and 'slower'

// Note: this line must before "if(start != -1 && count != 0)" condition

// which suppose only block 1st iteration when 'start' not filled in,

// in 2nd interation 'end' update to second index stored in treemap

// key, and that's the real first end index, now 'start' recorded

// previous 'end', and that's the real first start index

end = e.getKey();

// Additional condition 'count != 0' test out by:

// We don't store any section have default value 0

// Input segments = [[4,16,12],[9,10,15],[18,19,13],[3,13,20],[12,16,3],[2,10,10],[3,11,4],[13,16,6]]

// Use Testcase

// Output = [[2,3,10],[3,4,34],[4,9,46],[9,10,61],[10,11,36],[11,12,32],[12,13,35],[13,16,21],[16,18,0],[18,19,13]]

// Expected = [[2,3,10],[3,4,34],[4,9,46],[9,10,61],[10,11,36],[11,12,32],[12,13,35],[13,16,21],[18,19,13]]

if(start != -1 && count != 0) {

result.add(Arrays.asList(start, end, count));

}

// The running count (presum) binding with value at 'start' index

count += e.getValue();

start = end;

}

return result;

}

}

**Refer to**

<https://leetcode.com/problems/describe-the-painting/solutions/1359717/python-easy-solution-in-o-n-logn-with-detailed-explanation/>

### **Idea**

we can iterate on coordinates of each [start\_i, end\_i) from small to large and maintain the current mixed color.

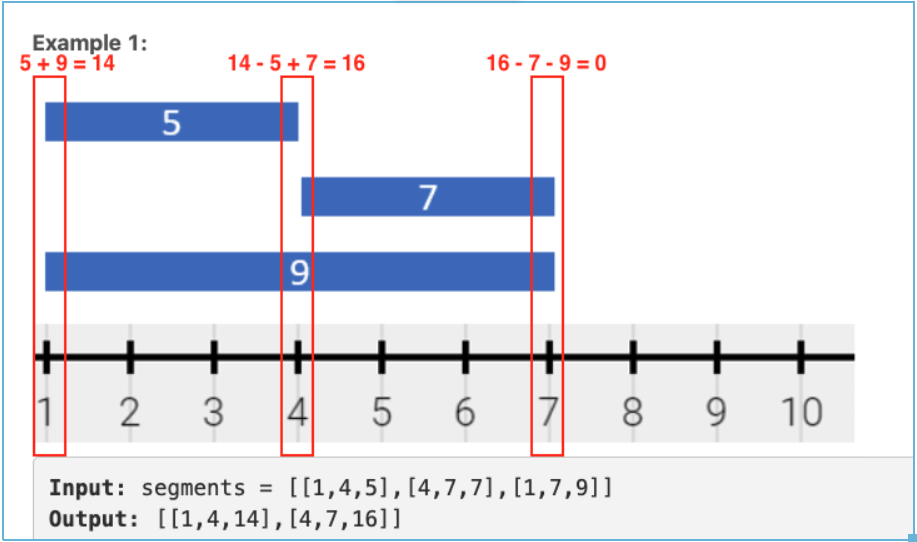
### **How can we maintain the mixed color?**

We can do addition if the current coordinate is the beginning of a [segment.In](http://segment.In) contrast, We do subtraction if the current coordinate is the end of a segment.

### **Examples**

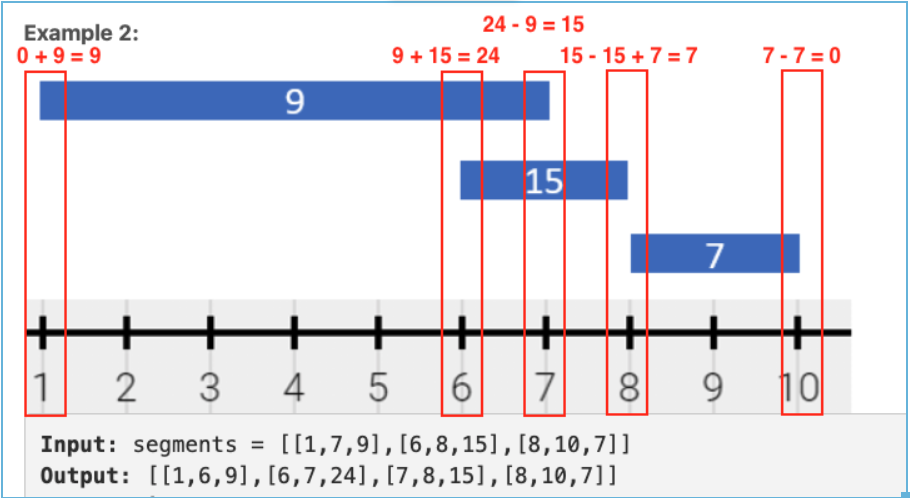
Let's give the example for understanding it easily.

#### **Example 1:**



There are 3 coordinates get involved. 1, 4, 7 respectively. At coordinate 1, there are two segment starting here. Thus, the current mixed color will be 5 + 9 = 14At coordinate 4, the segment [1, 4, 5] is end here and the segment [4, 7, 7] also start here. Thus, the current mixed color will be 14 - 5 + 7 = 16At coordinate 7, two segments are end here. Thus, the current mixed color will be 0So we know the final answer should be [1, 4, 14] and [4, 7, 16].

#### **Example 2:**



There are 5 coordinates get involved. 1, 6, 7, 8, 10 respectively. At coordinate 1, only the segment [1, 7, 9] start here. Thus, the current mixed color will be 0 + 9 = 9At coordinate 6, another segment [6, 8, 15] start here. Thus, the current mixed color will be 9 + 15 = 24At coordinate 7, the segment [1, 7, 9] is end here. Thus, the current mixed color will be 24 - 9 = 15At coordinate 8, the segment [6, 8, 15] is end here and the segment [8, 10, 7] also start here. Thus, the current mixed color will be 15 - 15 + 7 = 7At coordinate 10, the segment [8, 10, 7] is end here. Thus, the current mixed color will be 0So we know the final answer should be [1,6,9], [6,7,24], [7,8,15], [8,10,7].

### **Complexity**

* Time complexity: O(n \* log(n)) # We can use a fixed size array with length 10^5 to replace the default dict and bound the time complexity to O(n).
* Space complexity: O(n)

### **Code**

public List<List<Long>> splitPainting(int[][] segments) {

Map<Integer, Long> map = new TreeMap<>();

for (int[] s : segments) {

map.put(s[0], map.getOrDefault(s[0], 0l) + s[2]);

map.put(s[1], map.getOrDefault(s[1], 0l) - s[2]);

}

List<List<Long>> painting = new ArrayList<>();

int prev = 0;

long color = 0;

for (int curr : map.keySet()) {

// color == 0 means this segment is not painted

if (prev > 0 && color > 0) {

painting.add(Arrays.asList((long)prev, (long)curr, color));

}

color += map.get(curr);

prev = curr;

}

return painting;

}

**Refer to**

<https://algo.monster/liteproblems/1943>

## Problem Description

In this problem, we're asked to describe a painted mural represented as a series of segments on a number line, with each segment colored in a unique color. These segments can overlap, and when they do, the colors mix. The mixing of colors is not a blend but rather a union of distinct colors, and what's interesting is that you only need to report the sum of the mixed colors, rather than the set of colors themselves.

Each segment is defined by its start and end points, and its unique color value which is a number. What complicates the problem is that the mixing of colors from overlapping segments can create a patchwork of colored segments where each part may have a different mixed color.

Our task is to reduce this patchwork of segments into the minimum number of non-overlapping half-closed segments. These segments should capture the color mixing effect, and this final simplified description must exclude any part of the canvas that has not been painted. As an added twist, the segments in the output can be presented in any order.

We are provided with a 2D integer array segments where segments[i] = [start\_i, end\_i, color\_i]. We need to return another 2D array painting which represents these simplified half-closed segments [left\_j, right\_j, mix\_j] with mix\_j being the sum of colors on that segment.

A half-closed segment [a, b) includes point a but not point b.

## Intuition

To solve this problem, we need to find a way to efficiently represent the changes in color on the mural due to the overlapping segments. We need to account for when a new color starts (an increase in the sum of colors), when a color ends (a decrease in the sum), and maintain this accounting across the entire range of the painting.

The solution follows a common pattern in interval problems which involves marking the start and end of each segment with their corresponding color contributions using a dictionary or map. The key insight is to recognize that the coloring changes only at the endpoints of each segment (start and end). We don't care about the middle of the segments, since any internal points will just inherit the current mix of colors up to that point.

We use a dictionary to accumulate the color value (positive at the start, negative at the end) for each unique point on the number line where a color starts or ends. After processing all segments, we sort this dictionary to prepare for a sweep along the number line.

During the sweep, we accumulate the color values. If the accumulated color value is nonzero, it indicates that we have a painted segment. As we move from one point to the next, we detect changes in accumulated color values which mark the end of one segment and potentially the start of a new one.

The final result is obtained by creating segments between points where the color value changes from non-zero to another non-zero value. We ignore transitions from or to zero since this indicates an unpainted part of the mural.

By the end of this processing, we collect a list of the simplified half-closed segments, each with its sum of mixed colors, effectively compressing the painting into the desired output.

## Solution Approach

The solution utilizes a common technique in dealing with intervals or segments: marking the start and end points and then performing a "sweep" line approach, moving through the entire range of the mural to accumulate changes and construct the final set of segments.

Here is a step-by-step explanation of the process using a dictionary and a sorting strategy:

* **Use a Dictionary**: A dictionary, named d, is initialized to keep track of the start and end points of segments and how they influence the color changes at those points. Data structure choice is essential: a dictionary allows for efficient updates and lookups.
* **Mark the Changes**: Iterate through each segment, represented by (l, r, c), and modify the dictionary: increment the value at the start point l by c and decrement the value at the end point r by c. This sets up our "events" where color changes will occur.
* **Sort the Points**: Transform the dictionary into a sorted list, s, of tuples (k, v) where k is the point on the number line and v is the accumulated change in color at that point. Sorting by the number line ensures we'll process events in the correct order.
* **Accumulate Color Changes**: Iterate through the sorted list, accumulating the color values to track the current color sum. This accumulation reflects the presence of possibly multiple overlapping segments at each point.
* **Build Resulting Segments**: After accumulating the color values, iterate through the sorted list again, using s[i][1] to check if this segment should be added to the final output. A segment is created between consecutive points s[i][0] and s[i+1][0], with the color sum as the third element, but only if s[i][1] is non-zero. This inclusion check ensures that we don't consider unpainted parts of the mural.
* **Return the Result**: Construct the solution as a list of half-closed segments [s[i][0], s[i + 1][0], s[i][1]], where i ranges from 0 to n-2. Each of these segments has a start point, an end point, and a mixed color sum representing the painting described by the input array segments.

By employing this approach, the implementation details the specifics of processing intervals with efficient data structures like a dictionary for marking changes, sorting to prepare for a linear sweep, and accumulation to monitor the ongoing color sum. The code avoids unnecessary complexity, which can typically arise with overlapping intervals, by distilling the problem down to its essence: we only care about the points where color changes happen, and these are the starts and ends of the given segments.

### Example Walkthrough

Let's consider the following example to illustrate the solution approach. Suppose we're given the following array of segments as input:

segments = [[1, 4, 5], [3, 7, 6], [6, 8, 3]]

Each inner array represents a segment with start point, end point, and a unique color coded as a number.

* **Use a Dictionary**: We initiate a dictionary d to keep track of points and their corresponding color changes. Currently, d is empty.
* **Mark the Changes**: We process each segment in the segments array:

For the segment [1, 4, 5], we update d to reflect the start and end color effects:

d[1] is incremented by 5 (since 5 is the color at the start point).

d[4] is decremented by 5 (since 5 is the color at the end point).

For [3, 7, 6], we do analogously:

d[3] is incremented by 6.

d[7] is decremented by 6.

For [6, 8, 3], we update again:

d[6] is incremented by 3.

d[8] is decremented by 3.

After processing all segments, our dictionary may look something like this:

* **Sort the Points**: We then sort the keys of our dictionary d to get an ordered list of color change events. This list, s, becomes:
* **Accumulate Color Changes**: We iterate through s and keep track of the cumulative color. We keep adding (or subtracting if negative) the values corresponding to the points:

After (1, 5), cumulative color is 5.

After (3, 6), cumulative color increases to 11 (5 + 6).

At (4, -5), it decreases to 6 (11 - 5).

At (6, 3), it goes back to 9 (6 + 3).

And so on, until we go through all points.

* **Build Resulting Segments**: When the cumulative color value changes from one non-zero value to another, we add a new segment to our output result. After processing the sorted list, the output might look like this:
* We can see that each new segment in painting starts where the last one ended, representing the mixed colors appropriately.
* **Return the Result**: This painting now represents the simplified list of half-closed segments capturing the color mixing effect, which is exactly what the problem asks for.

By using this technique, we have efficiently compressed the original series of overlapping colored segments into a simplified and non-overlapping representation, accurately reflecting the mix of colors on the mural with the order being of no significance.

import java.util.TreeMap;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

import java.util.Objects;

import java.util.Arrays;

class Solution {

public List<List<Long>> splitPainting(int[][] segments) {

// A TreeMap stores unique keys in sorted order, along with their corresponding values.

// Here, keys track the position on the canvas and values track paint color changes.

TreeMap<Integer, Long> deltaMap = new TreeMap<>();

// Process each paint segment and update the deltas for start and end points

for (int[] segment : segments) {

int start = segment[0], end = segment[1];

long color = segment[2];

// Increase color value at the start point

deltaMap.put(start, deltaMap.getOrDefault(start, 0L) + color);

// Decrease color value at the end point

deltaMap.put(end, deltaMap.getOrDefault(end, 0L) - color);

}

// Prepare a list to store the result segments

List<List<Long>> result = new ArrayList<>();

long previousPosition = 0, currentPosition = 0;

long currentColorSum = 0;

// Go through each entry in the map to create the painted segments

for (Map.Entry<Integer, Long> entry : deltaMap.entrySet()) {

// If it's the first key, initialize the previous position.

if (Objects.equals(entry.getKey(), deltaMap.firstKey())) {

previousPosition = entry.getKey();

} else {

currentPosition = entry.getKey();

// If the color sum is positive, add the segment to the result list.

if (currentColorSum > 0) {

result.add(Arrays.asList(previousPosition, currentPosition, currentColorSum));

}

// Update the previous position for the next iteration

previousPosition = currentPosition;

}

// Update the color sum with the delta value

currentColorSum += entry.getValue();

}

return result;

}

}

### Time Complexity

The time complexity of the given code can be broken down into several parts:

**Iterating over segments:** The input to the function is segments, a list of lists where each sublist has the format [l, r, c]. The code uses a for loop to iterate over these segments, applying operations that are constant time to each (updating the dictionary d). Therefore, this part has a time complexity of O(m), where m is the number of segments.

**Creating the sorted list s:** The sorted function is then called on a list comprehension [k, v] for each key-value pair in the dictionary d. The sorted function has a time complexity of O(nlogn), where n is the number of distinct keys in the dictionary d.

**Iterating over sorted list s:** An in-place iteration to compute the cumulative sum of values is performed in a loop running from 1 to n. Since it's a simple accumulation of values, the time complexity is O(n) for this loop.

**List comprehension to construct the return list:** Lastly, a list comprehension is used to create the final list of painting segments that should be returned. The list comprehension iterates n - 1 times to check the value of the cumulative color and make a list of the segments. This operation has a time complexity of O(n).

Combining all of these, the overall time complexity of the function is determined by the most time-consuming operation, which in this case is the sorting step. So, the overall time complexity of the code is O(nlogn).

### Space Complexity

For space complexity:

**Dictionary d:** The space taken up by the dictionary is proportional to the number of distinct starting and ending points in the segments, which is O(n).

**Sorted list s:** The list s has the same number of elements as the dictionary d, which is O(n).

**Return list:** In the worst case, the return list will have as many segments as the input segments, because every segment might have a different color, which is O(m) space.

Overall, the space complexity combines the space for the dictionary d, the sorted list s, and the return list, which gives us O(n + m). Since m can be assumed to be less than or equal to 2n (each segment contributes two points at most), the space complexity can also be loosely represented as O(n).